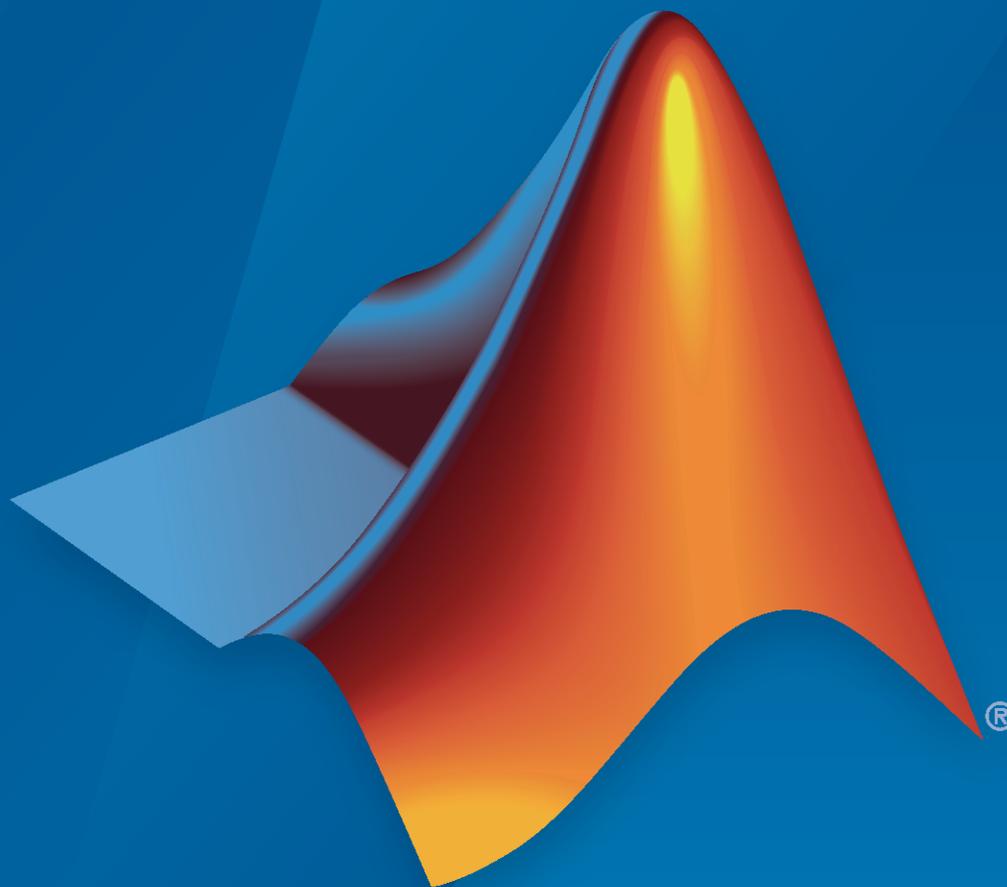


Bioinformatics Toolbox™

User's Guide



MATLAB®

R2026a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Bioinformatics Toolbox™ User's Guide

© COPYRIGHT 2003–2026 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2003	Online only	New for Version 1.0 (Release 13SP1+)
June 2004	Online only	Revised for Version 1.1 (Release 14)
November 2004	Online only	Revised for Version 2.0 (Release 14SP1+)
March 2005	Online only	Revised for Version 2.0.1 (Release 14SP2)
May 2005	Online only	Revised for Version 2.1 (Release 14SP2+)
September 2005	Online only	Revised for Version 2.1.1 (Release 14SP3)
November 2005	Online only	Revised for Version 2.2 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.2.1 (Release 2006a)
May 2006	Online only	Revised for Version 2.3 (Release 2006a+)
September 2006	Online only	Revised for Version 2.4 (Release 2006b)
March 2007	Online only	Revised for Version 2.5 (Release 2007a)
April 2007	Online only	Revised for Version 2.6 (Release 2007a+)
September 2007	Online only	Revised for Version 3.0 (Release 2007b)
March 2008	Online only	Revised for Version 3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.5 (Release 2010a)
September 2010	Online only	Revised for Version 3.6 (Release 2010b)
April 2011	Online only	Revised for Version 3.7 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 4.3.1 (Release 2013b)
March 2014	Online only	Revised for Version 4.4 (Release 2014a)
October 2014	Online only	Revised for Version 4.5 (Release 2014b)
March 2015	Online only	Revised for Version 4.5.1 (Release 2015a)
September 2015	Online only	Revised for Version 4.5.2 (Release 2015b)
March 2016	Online only	Revised for Version 4.6 (Release 2016a)
September 2016	Online only	Revised for Version 4.7 (Release 2016b)
March 2017	Online only	Revised for Version 4.8 (Release 2017a)
September 2017	Online only	Revised for Version 4.9 (Release 2017b)
March 2018	Online only	Revised for Version 4.10 (Release 2018a)
September 2018	Online only	Revised for Version 4.11 (Release 2018b)
March 2019	Online only	Revised for Version 4.12 (Release 2019a)
September 2019	Online only	Revised for Version 4.13 (Release 2019b)
March 2020	Online only	Revised for Version 4.14 (Release 2020a)
September 2020	Online only	Revised for Version 4.15 (Release 2020b)
March 2021	Online only	Revised for Version 4.15.1 (Release 2021a)
September 2021	Online only	Revised for Version 4.15.2 (Release 2021b)
March 2022	Online only	Revised for Version 4.16 (Release 2022a)
September 2022	Online only	Revised for Version 4.16.1 (Release 2022b)
March 2023	Online only	Revised for Version 4.17 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Revised for Version 25.2 (R2025b)
March 2026	Online only	Revised for Version 26.1 (R2026a)

1

Getting Started

Bioinformatics Toolbox Product Description	1-2
Product Overview	1-3
Features	1-3
Expected Users	1-4
Data Formats and Databases	1-5
Sequence Alignments	1-7
Sequence Utilities and Statistics	1-8
Protein Property Analysis	1-9
Phylogenetic Analysis	1-10
Microarray Data Analysis Tools	1-11
Microarray Data Storage	1-12
Mass Spectrometry Data Analysis	1-13
Statistical Learning and Visualization	1-15
Prototyping and Development Environment	1-16
Data Visualization	1-17
Exchange Bioinformatics Data Between Excel and MATLAB	1-18
Use Excel and MATLAB Together	1-18
About the Example	1-18
Set System Path and Enable Add-In	1-18
Download Spreadsheet with Filtered Yeast Data	1-18
Run the Example for the Entire Data Set	1-19
Edit Formulas to Run the Example on a Subset of the Data	1-20
Use the Spreadsheet Link product to Interact With the Data in MATLAB	1-21
Working with Whole Genome Data	1-24
Comparing Whole Genomes	1-31

Work with Next-Generation Sequencing Data	2-2
Overview	2-2
What Files Can You Access?	2-2
Before You Begin	2-3
Create a BioIndexedFile Object to Access Your Source File	2-3
Determine the Number of Entries Indexed by a BioIndexedFile Object ...	2-3
Retrieve Entries from Your Source File	2-4
Read Entries from Your Source File	2-4
Manage Sequence Read Data in Objects	2-6
Overview	2-6
Represent Sequence and Quality Data in a BioRead Object	2-7
Represent Sequence, Quality, and Alignment/Mapping Data in a BioMap Object	2-8
Retrieve Information from a BioRead or BioMap Object	2-10
Set Information in a BioRead or BioMap Object	2-12
Determine Coverage of a Reference Sequence	2-12
Construct Sequence Alignments to a Reference Sequence	2-13
Filter Read Sequences Using SAM Flags	2-14
Store and Manage Feature Annotations in Objects	2-16
Represent Feature Annotations in a GFFAnnotation or GTFAnnotation Object	2-16
Construct an Annotation Object	2-16
Retrieve General Information from an Annotation Object	2-16
Access Data in an Annotation Object	2-17
Use Feature Annotations with Sequence Read Data	2-18
Bioinformatics Toolbox Software Support Packages	2-21
Install Support Package	2-21
Available Support Packages	2-21
Count Features from NGS Reads	2-23
Identifying Differentially Expressed Genes from RNA-Seq Data	2-32
Visualize NGS Data Using Genomics Viewer App	2-53
Open the App	2-53
Add Tracks by Importing Data	2-54
Visualize Single Nucleotide Variation in Cytochrome P450	2-54
Exploring Genome-Wide Differences in DNA Methylation Profiles	2-59
Exploring Protein-DNA Binding Sites from Paired-End CHIP-Seq Data	2-80
Working with Illumina/Solexa Next-Generation Sequencing Data	2-98
Bioinformatics Pipeline SplitDimension	2-108
Specify SplitDimension to Select Which Input Array Dimensions to Split	2-108

Provide Compatible Array sizes	2-108
Default Value of SplitDimension for Built-In Pipeline Blocks	2-109
Show split dimensions in Biopipeline Designer	2-111
Split Input SAM Files and Assemble Transcriptomes Using Bioinformatics Pipeline	2-112
Bioinformatics Pipeline Run Mode	2-114
Create Simple Pipeline to Plot Sequence Quality Data Using Biopipeline Designer	2-115
Count RNA-Seq Reads Using Biopipeline Designer	2-127
Bioinformatics Pipeline Block Libraries	2-136
Analyze Gene Expression Profiles Using Biopipeline Designer	2-139
Predict Protein Secondary Structure Using Biopipeline Designer	2-150

Sequence Analysis

3

Exploring a Nucleotide Sequence Using Command Line	3-2
Overview of Example	3-2
Searching the Web for Sequence Information	3-2
Reading Sequence Information from the Web	3-4
Determining Nucleotide Composition	3-5
Determining Codon Composition	3-8
Open Reading Frames	3-11
Amino Acid Conversion and Composition	3-13
Compare Sequences Using Sequence Alignment Algorithms	3-15
View and Align Multiple Sequences	3-22
Overview of the Sequence Alignment App	3-22
Visualize Multiple Sequence Alignment	3-22
Adjust Sequence Alignments Manually	3-23
Rearrange Rows	3-31
Generate Phylogenetic Tree from Aligned Sequences	3-33
Analyzing Synonymous and Nonsynonymous Substitution Rates	3-36
Investigating the Bird Flu Virus	3-46
Exploring Primer Design	3-57
Identifying Over-Represented Regulatory Motifs	3-67
Predicting and Visualizing the Secondary Structure of RNA Sequences	3-78

Using HMMs for Profile Analysis of a Protein Family	3-90
Predicting Protein Secondary Structure Using a Neural Network	3-107
Calculating and Visualizing Sequence Statistics	3-124
Aligning Pairs of Sequences	3-138
Assessing the Significance of an Alignment	3-144
Using Scoring Matrices to Measure Evolutionary Distance	3-153
Calling Bioperl Functions from MATLAB	3-157
Accessing NCBI Entrez Databases with E-Utilities	3-169

Microarray Analysis

4

Managing Gene Expression Data in Objects	4-2
Representing Expression Data Values in DataMatrix Objects	4-5
Overview of DataMatrix Objects	4-5
Constructing DataMatrix Objects	4-5
Getting and Setting Properties of a DataMatrix Object	4-6
Accessing Data in DataMatrix Objects	4-6
Representing Expression Data Values in ExptData Objects	4-9
Overview of ExptData Objects	4-9
Constructing ExptData Objects	4-9
Using Properties of an ExptData Object	4-10
Using Methods of an ExptData Object	4-10
References	4-11
Representing Sample and Feature Metadata in MetaData Objects	4-12
Overview of MetaData Objects	4-12
Constructing MetaData Objects	4-13
Using Properties of a MetaData Object	4-15
Using Methods of a MetaData Object	4-15
Representing Experiment Information in a MIAME Object	4-16
Overview of MIAME Objects	4-16
Constructing MIAME Objects	4-16
Using Properties of a MIAME Object	4-17
Using Methods of a MIAME Object	4-18
Representing All Data in an ExpressionSet Object	4-19
Overview of ExpressionSet Objects	4-19
Constructing ExpressionSet Objects	4-21
Using Properties of an ExpressionSet Object	4-21
Using Methods of an ExpressionSet Object	4-22

Analyzing Illumina Bead Summary Gene Expression Data	4-23
Detecting DNA Copy Number Alteration in Array-Based CGH Data	4-44
Analyzing Array-Based CGH Data Using Bayesian Hidden Markov Modeling	4-60
Visualizing Microarray Data	4-74
Gene Expression Profile Analysis	4-95
Working with GEO Series Data	4-112
Identifying Biomolecular Subgroups Using Attractor Metagenes	4-123
Working with the Clustergram Function	4-135
Working with Objects for Microarray Experiment Data	4-152

Phylogenetic Analysis

5

Using the Phylogenetic Tree App	5-2
Overview of the Phylogenetic Tree App	5-2
Opening the Phylogenetic Tree App	5-2
File Menu	5-3
Tools Menu	5-11
Window Menu	5-17
Help Menu	5-18
Building a Phylogenetic Tree for the Hominidae Species	5-19
Analyzing the Origin of the Human Immunodeficiency Virus	5-25
Bootstrapping Phylogenetic Trees	5-32

Mass Spectrometry and Bioanalytics

6

Preprocessing Raw Mass Spectrometry Data	6-2
Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling	6-19
Identifying Significant Features and Classifying Protein Profiles	6-38
Differential Analysis of Complex Protein and Metabolite Mixtures Using Liquid Chromatography/Mass Spectrometry (LC/MS)	6-52

Genetic Algorithm Search for Features in Mass Spectrometry Data . . .	6-71
Batch Processing of Spectra Using Sequential and Parallel Computing	6-77

Getting Started

- “Bioinformatics Toolbox Product Description” on page 1-2
- “Product Overview” on page 1-3
- “Data Formats and Databases” on page 1-5
- “Sequence Alignments” on page 1-7
- “Sequence Utilities and Statistics” on page 1-8
- “Protein Property Analysis” on page 1-9
- “Phylogenetic Analysis” on page 1-10
- “Microarray Data Analysis Tools” on page 1-11
- “Microarray Data Storage” on page 1-12
- “Mass Spectrometry Data Analysis” on page 1-13
- “Statistical Learning and Visualization” on page 1-15
- “Prototyping and Development Environment” on page 1-16
- “Data Visualization” on page 1-17
- “Exchange Bioinformatics Data Between Excel and MATLAB” on page 1-18
- “Working with Whole Genome Data” on page 1-24
- “Comparing Whole Genomes” on page 1-31

Bioinformatics Toolbox Product Description

Read, analyze, and visualize genomic and proteomic data

Bioinformatics Toolbox provides algorithms and apps for building bioinformatics pipelines, Next Generation Sequencing (NGS), microarray analysis, mass spectrometry, and gene ontology. You can read genomic and proteomic data and explore this data with the Genomics Viewer app and visualize with spatial heatmaps and clustergrams. The toolbox (with Statistics and Machine Learning Toolbox™) provides statistical, regression, classification, and (with Deep Learning Toolbox™) deep learning techniques to build complete analysis pipelines.

The Biopipeline Designer app lets you interactively build bioinformatics pipelines for genomic data analysis, locally or in the cloud. You can use a combination of built-in bioinformatics pipeline blocks that integrate proven NGS libraries and custom blocks to extend analyses with community tools. You can create a pipeline to preprocess reads, map them to a reference genome, and perform statistical analysis, like RNA-Seq differential expression analysis or ChIP-Seq genome-wide analysis.

Product Overview

Features

The Bioinformatics Toolbox product extends the MATLAB® environment to provide an integrated software environment for genome and proteome analysis. Scientists and engineers can answer questions, solve problems, prototype new algorithms, and build applications for drug discovery and design, genetic engineering, and biological research. An introduction to these features will help you to develop a conceptual model for working with the toolbox and your biological data.

The Bioinformatics Toolbox product includes many functions to help you with genome and proteome analysis. Most functions are implemented in the MATLAB programming language, with the source available for you to view. This open environment lets you explore and customize the existing toolbox algorithms or develop your own.

You can use the basic bioinformatic functions provided with this toolbox to create more complex algorithms and applications. These robust and well-tested functions are the functions that you would otherwise have to create yourself.

Toolbox features and functions fall within these categories:

- **Data formats and databases** — Connect to Web-accessible databases containing genomic and proteomic data. Read and convert between multiple data formats.
- **High-throughput sequencing** — Gene expression and transcription factor analysis of next-generation sequencing data, including RNA-Seq and ChIP-Seq.
- **Sequence analysis** — Determine the statistical characteristics of a sequence, align two sequences, and multiply align several sequences. Model patterns in biological sequences using hidden Markov model (HMM) profiles.
- **Phylogenetic analysis** — Create and manipulate phylogenetic tree data.
- **Microarray data analysis** — Read, normalize, and visualize microarray data.
- **Mass spectrometry data analysis** — Analyze and enhance raw mass spectrometry data.
- **Statistical learning** — Classify and identify features in data sets with statistical learning tools.
- **Programming interface** — Use other bioinformatic software (BioPerl and BioJava) within the MATLAB environment.

The field of bioinformatics is rapidly growing and will become increasingly important as biology becomes a more analytical science. The toolbox provides an open environment that you can customize for development and deployment of the analytical tools you will need.

- **Prototype and develop algorithms** — Prototype new ideas in an open and extensible environment. Develop algorithms using efficient string processing and statistical functions, view the source code for existing functions, and use the code as a template for customizing, improving, or creating your own functions. See “Prototyping and Development Environment” on page 1-16.
- **Visualize data** — Visualize sequences and alignments, gene expression data, phylogenetic trees, mass spectrometry data, protein structure, and relationships between data with interconnected graphs. See “Data Visualization” on page 1-17.
- **Share and deploy applications** — Use an interactive GUI builder to develop a custom graphical front end for your data analysis programs. Create standalone applications that run separately from the MATLAB environment.

Expected Users

The Bioinformatics Toolbox product is intended for computational biologists and research scientists who need to develop new algorithms or implement published ones, visualize results, and create standalone applications.

- **Industry/Professional** — Increasingly, drug discovery methods are being supported by engineering practice. This toolbox supports tool builders who want to create applications for the biotechnology and pharmaceutical industries.
- **Education/Professor/Student** — This toolbox is well suited for learning and teaching genome and proteome analysis techniques. Educators and students can concentrate on bioinformatic algorithms instead of programming basic functions such as reading and writing to files.

While the toolbox includes many bioinformatic functions, it is not intended to be a complete set of tools for scientists to analyze their biological data. However, the MATLAB environment is ideal for rapidly designing and prototyping the tools you need.

Data Formats and Databases

The Bioinformatics Toolbox lets you access many of the databases on the web and other online data repositories. It lets you copy data into the MATLAB workspace, and read and write to files with standard bioinformatic formats. It also reads many common genome file formats so that you do not have to write and maintain your own file readers.

Web-based databases — You can directly access public databases on the Web and copy sequence and gene expression information into the MATLAB environment.

The sequence databases currently supported are GenBank® (`getgenbank`), GenPept (`getgenpept`), European Molecular Biology Laboratory (EMBL) (`getembl`), and Protein Data Bank (PDB) (`getpdb`). You can also access data from the NCBI Gene Expression Omnibus (GEO) Web site by using a single function (`getgeodata`).

Get multiply aligned sequences (`gethmmalignment`), hidden Markov model profiles (`gethmmprof`), and phylogenetic tree data (`gethmmtree`) from the PFAM database.

Gene Ontology database — Load the database from the Web into a gene ontology object (`geneont`). Select sections of the ontology with methods for the `geneont` object (`getancestors`, `getdescendents`, `getmatrix`, `getrelatives`), and manipulate data with utility functions (`goannotread`, `num2goid`).

Read data from instruments — Read data generated from gene sequencing instruments (`scfread`, `joinseq`, `traceplot`), mass spectrometers (`jcampread`), and Agilent® microarray scanners (`agferead`).

Reading data formats — The toolbox provides a number of functions for reading data from common bioinformatic file formats.

- Sequence data: GenBank (`genbankread`), GenPept (`genpeptread`), EMBL (`emblread`), PDB (`pdbread`), and FASTA (`fastaread`)
- Multiply aligned sequences: ClustalW and GCG formats (`multialignread`)
- Gene expression data from microarrays: Gene Expression Omnibus (GEO) data (`geosoftread`), GenePix® data in GPR and GAL files (`gprread`, `galread`), SPOT data (`sptread`), and Imagen® results files (`imageneread`)
- Hidden Markov model profiles: PFAM-HMM file (`pfamhmmread`)

Writing data formats — The functions for getting data from the Web include the option to save the data to a file. However, there is a function to write data to a file using the FASTA format (`fastawrite`).

BLAST searches — Request Web-based BLAST searches (`blastncbi`), get the results from a search (`getblast`) and read results from a previously saved BLAST formatted report file (`blastread`).

The MATLAB environment has built-in support for other industry-standard file formats including Microsoft® Excel® and comma-separated-value (CSV) files. Additional functions perform ASCII and low-level binary I/O, allowing you to develop custom functions for working with any data format.

See Also

More About

- “High-Throughput Sequencing”
- “Microarray Analysis”
- “Sequence Analysis”
- “Structural Analysis”
- “Mass Spectrometry and Bioanalytics”

Sequence Alignments

You can select from a list of analysis methods to compare nucleotide or amino acid sequences using pairwise or multiple sequence alignment functions.

Pairwise sequence alignment — Efficient implementations of standard algorithms such as the Needleman-Wunsch (`nwalign`) and Smith-Waterman (`swalign`) algorithms for pairwise sequence alignment. The toolbox also includes standard scoring matrices such as the PAM and BLOSUM families of matrices (`blosum`, `dayhoff`, `gonnet`, `nuc44`, `pam`). Visualize sequence similarities with `seqdotplot`.

Multiple sequence alignment — Functions for multiple sequence alignment (`multialign`, `profalign`) and functions that support multiple sequences (`multialignread`, `fastaread`). There is also a graphical interface (`seqalignviewer`) for viewing the results of a multiple sequence alignment and manually making adjustment.

Multiple sequence profiles — Implementations for multiple alignment and profile hidden Markov model algorithms (`gethmmprof`, `gethmmalignment`, `gethmmtree`, `pfamhmmread`, `hmmprofalign`, `hmmprofestimate`, `hmmprofgenerate`, `hmmprofmerge`, `hmmprofstruct`, `showhmmprof`).

Biological codes — Look up the letters or numeric equivalents for commonly used biological codes (`aminolookup`, `baselookup`, `geneticcode`, `revgeneticcode`).

See Also

More About

- “Sequence Utilities and Statistics” on page 1-8
- “Sequence Analysis”
- “Data Formats and Databases” on page 1-5

Sequence Utilities and Statistics

You can manipulate and analyze your sequences to gain a deeper understanding of the physical, chemical, and biological characteristics of your data. Use a graphical user interface (GUI) with many of the sequence functions in the toolbox.

Sequence conversion and manipulation — The toolbox provides routines for common operations, such as converting DNA or RNA sequences to amino acid sequences, that are basic to working with nucleic acid and protein sequences (`aa2int`, `aa2nt`, `dna2rna`, `rna2dna`, `int2aa`, `int2nt`, `nt2aa`, `nt2int`, `seqcomplement`, `seqrcomplement`, `seqreverse`).

You can manipulate your sequence by performing an *in silico* digestion with restriction endonucleases (`restrict`) and proteases (`cleave`).

Sequence statistics — Determine various statistics about a sequence (`aacount`, `basecount`, `codoncount`, `dimercount`, `nmercount`, `ntdensity`, `codonbias`, `cpgisland`, `oligoprop`), or search for specific patterns within a sequence (`seqwordcount`). In addition, you can create random sequences for test cases (`randseq`).

Sequence utilities — Determine a consensus sequence from a set of multiply aligned amino acid, nucleotide sequences (`seqconsensus`, or a sequence profile (`seqprofile`)). Format a sequence for display (`seqdisp`) or graphically show a sequence alignment with frequency data (`seqlogo`).

Additional MATLAB functions efficiently handle string operations with regular expressions (`regexp`, `seq2regexp`) to look for specific patterns in a sequence and search through a library for string matches (`seqmatch`).

Look for possible cleavage sites in a DNA/RNA sequence by searching for palindromes (`palindromes`).

See Also

More About

- “Sequence Alignments” on page 1-7
- “Sequence Analysis”
- “Protein and Amino Acid Sequence Analysis”
- “Data Formats and Databases” on page 1-5

Protein Property Analysis

You can use a collection of protein analysis methods to extract information from your data. You can determine protein characteristics and simulate enzyme cleavage reactions. The toolbox provides functions to calculate various properties of a protein sequence, such as the atomic composition (`atomiccomp`), molecular weight (`molweight`), and isoelectric point (`isoelectric`). You can cleave a protein with an enzyme (`cleave`, `rebasecuts`) and create distance and Ramachandran plots for PDB data (`pdbdistplot`, `ramachandran`). The toolbox contains a graphical user interface for protein analysis (`proteinplot`) and plotting 3-D protein and other molecular structures with information from molecule model files, such as PDB files.

Amino acid sequence utilities — Calculate amino acid statistics for a sequence (`aaccount`) and get information about character codes (`aminolookup`).

See Also

More About

- “Protein and Amino Acid Sequence Analysis”
- “Structural Analysis”

Phylogenetic Analysis

Phylogenetic analysis is the process you use to determine the evolutionary relationships between organisms. The results of an analysis can be drawn in a hierarchical diagram called a cladogram or phylogram (phylogenetic tree). The branches in a tree are based on the hypothesized evolutionary relationships (phylogeny) between organisms. Each member in a branch, also known as a monophyletic group, is assumed to be descended from a common ancestor. Originally, phylogenetic trees were created using morphology, but now, determining evolutionary relationships includes matching patterns in nucleic acid and protein sequences. The Bioinformatics Toolbox provides the following data structure and functions for phylogenetic analysis.

Phylogenetic tree data — Read and write Newick-formatted tree files (`phytreeread`, `phytreewrite`) into the MATLAB Workspace as phylogenetic tree objects (`phytree`).

Create a phylogenetic tree — Calculate the pairwise distance between biological sequences (`seqpdist`), estimate the substitution rates (`dnds`, `dndsm1`), build a phylogenetic tree from pairwise distances (`seqlinkage`, `seqneighjoin`, `reroot`), and view the tree in an interactive GUI that allows you to view, edit, and explore the data (`phytreeviewer` or `view`). This GUI also allows you to prune branches, reorder, rename, and explore distances.

Phylogenetic tree object methods — You can access the functionality of the `phytreeviewer` user interface using methods for a phylogenetic tree object (`phytree`). Get property values (`get`) and node names (`getbyname`). Calculate the patristic distances between pairs of leaf nodes (`pdist`, `weights`) and draw a phylogenetic tree object in a MATLAB Figure window as a phylogram, cladogram, or radial treeplot (`plot`). Manipulate tree data by selecting branches and leaves using a specified criterion (`select`, `subtree`) and removing nodes (`prune`). Compare trees (`getcanonical`) and use Newick-formatted strings (`getnewickstr`).

See Also

More About

- “Sequence Utilities and Statistics” on page 1-8
- “Sequence Analysis”

Microarray Data Analysis Tools

The MATLAB environment is widely used for microarray data analysis, including reading, filtering, normalizing, and visualizing microarray data. However, the standard normalization and visualization tools that scientists use can be difficult to implement. The toolbox includes these standard functions:

Microarray data — Read ImaGene results files (`imageneread`), SPOT files (`sptread`) and Agilent microarray scanner files (`agferead`). Read GenePix GPR files (`gprread`) and GAL files (`galread`). Get Gene Expression Omnibus (GEO) data from the Web (`getgeodata`) and read GEO data from files (`geosoftread`).

A utility function (`magetfield`) extracts data from one of the microarray reader functions (`gprread`, `agferead`, `sptread`, `imageneread`).

Microarray normalization and filtering — The toolbox provides a number of methods for normalizing microarray data, such as lowess normalization (`malowess`) and mean normalization (`manorm`), or across multiple arrays (`quantilenorm`). You can use filtering functions to clean raw data before analysis (`geneentropyfilter`, `genelowvalfilter`, `generangefilter`, `genevarfilter`), and calculate the range and variance of values (`exprprofrange`, `exprprofvar`).

Microarray visualization — The toolbox contains routines for visualizing microarray data. These routines include spatial plots of microarray data (`maimage`, `redgreencmap`), box plots (`maboxplot`), loglog plots (`maloglog`), and intensity-ratio plots (`mairplot`). You can also view clustered expression profiles (`clustergram`, `redgreencmap`). You can create 2-D scatter plots of principal components from the microarray data (`mapcaplot`).

The toolbox accesses statistical routines to perform cluster analysis and to visualize the results, and you can view your data through statistical visualizations such as dendrograms, classification, and regression trees.

See Also

More About

- “Microarray Data Storage” on page 1-12
- “Microarray Analysis”

Microarray Data Storage

The Bioinformatics Toolbox includes functions, objects, and methods for creating, storing, and accessing microarray data.

The object constructor function, `DataMatrix`, lets you create a `DataMatrix` object to encapsulate data and metadata from a microarray experiment. A `DataMatrix` object stores experimental data in a matrix, with rows typically corresponding to gene names or probe identifiers, and columns typically corresponding to sample identifiers. A `DataMatrix` object also stores metadata, including the gene names or probe identifiers (as the row names) and sample identifiers (as the column names).

You can reference microarray expression values in a `DataMatrix` object the same way you reference data in a MATLAB array, that is, by using linear or logical indexing. Alternately, you can reference this experimental data by gene (probe) identifiers and sample identifiers. Indexing by these identifiers lets you quickly and conveniently access subsets of the data without having to maintain additional index arrays.

Many MATLAB operators and arithmetic functions are available to `DataMatrix` objects by means of methods. These methods let you modify, combine, compare, analyze, plot, and access information from `DataMatrix` objects. Additionally, you can easily extend the functionality by using general element-wise functions, `dmarrayfun` and `dmbsxfun`, and by manually accessing the properties of a `DataMatrix` object.

Note For more information on creating and using `DataMatrix` objects, see “Representing Expression Data Values in `DataMatrix` Objects” on page 4-5.

See Also

More About

- “Microarray Data Analysis Tools” on page 1-11
- “Microarray Analysis”

Mass Spectrometry Data Analysis

The mass spectrometry functions preprocess and classify raw data from SELDI-TOF and MALDI-TOF spectrometers and use statistical learning functions to identify patterns.

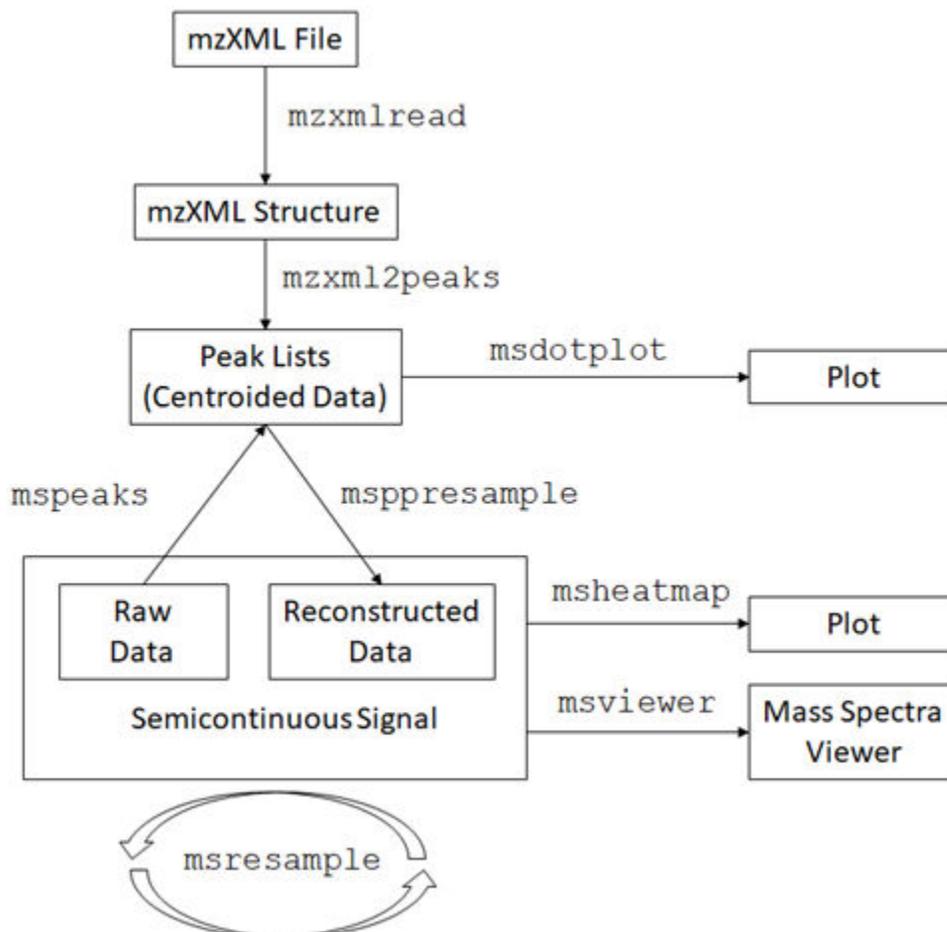
Reading raw data — Load raw mass/charge and ion intensity data from comma-separated-value (CSV) files, or read a JCAMP-DX-formatted file with mass spectrometry data (`jcampread`) into the MATLAB environment.

You can also have data in TXT files and use the `importdata` function.

Preprocessing raw data — Resample high-resolution data to a lower resolution (`msresample`) where the extra data points are not needed. Correct the baseline (`msbackadj`). Align a spectrum to a set of reference masses (`msalign`) and visually verify the alignment (`msheatmap`). Normalize the area between spectra for comparing (`msnorm`), and filter out noise (`mslowess` and `mssgolay`).

Spectrum analysis — Load spectra into a GUI (`msviewer`) for selecting mass peaks and further analysis.

The following graphic illustrates the roles of the various mass spectrometry functions in the toolbox.



See Also

More About

- “Mass Spectrometry and Bioanalytics”
- “Data Formats and Databases” on page 1-5

Statistical Learning and Visualization

You can classify and identify features in data sets, set up cross-validation experiments, and compare different classification methods.

The toolbox provides functions that build on the classification and statistical learning tools in the Statistics and Machine Learning Toolbox software (`classify`, `kmeans`, `fitctree`, and `fitrtree`).

These functions include imputation tools (`knnimpute`), and K-nearest neighbor classifiers (`fitcknn`).

Other functions include set up of cross-validation experiments (`crossvalind`) and comparison of the performance of different classification methods (`classperf`). In addition, there are tools for selecting diversity and discriminating features (`rankfeatures`, `randfeatures`).

Prototyping and Development Environment

The MATLAB environment lets you prototype and develop algorithms and easily compare alternatives.

- **Integrated environment** — Explore biological data in an environment that integrates programming and visualization. Create reports and plots with the built-in functions for mathematics, graphics, and statistics.
- **Open environment** — Access the source code for the toolbox functions. The toolbox includes many of the basic bioinformatics functions you will need to use, and it includes prototypes for some of the more advanced functions. Modify these functions to create your own custom solutions.
- **Interactive programming language** — Test your ideas by typing functions that are interpreted interactively with a language whose basic data element is an array. The arrays do not require dimensioning and allow you to solve many technical computing problems,

Using matrices for sequences or groups of sequences allows you to work efficiently and not worry about writing loops or other programming controls.

- **Programming tools** — Use a visual debugger for algorithm development and refinement and an algorithm performance profiler to accelerate development.

Data Visualization

You can visually compare pairwise sequence alignments, multiply aligned sequences, gene expression data from microarrays, and plot nucleic acid and protein characteristics. The 2-D and volume visualization features let you create custom graphical representations of multidimensional data sets. You can also create montages and overlays, and export finished graphics to an Adobe® PostScript® image file or copy directly into Microsoft PowerPoint®.

Exchange Bioinformatics Data Between Excel and MATLAB

In this section...

“Use Excel and MATLAB Together” on page 1-18

“About the Example” on page 1-18

“Set System Path and Enable Add-In” on page 1-18

“Download Spreadsheet with Filtered Yeast Data” on page 1-18

“Run the Example for the Entire Data Set” on page 1-19

“Edit Formulas to Run the Example on a Subset of the Data” on page 1-20

“Use the Spreadsheet Link product to Interact With the Data in MATLAB” on page 1-21

Use Excel and MATLAB Together

If you have bioinformatics data in an Excel (2007 or newer) spreadsheet, use Spreadsheet Link to:

- Connect Excel with the MATLAB Workspace to exchange data
- Use MATLAB and Bioinformatics Toolbox computational and visualization functions

About the Example

Note The following example assumes you have Spreadsheet Link software installed on your system.

The Excel file used in the following example contains data from DeRisi, J.L., Iyer, V.R., and Brown, P.O. (Oct. 24, 1997). Exploring the metabolic and genetic control of gene expression on a genomic scale. *Science* 278(5338), 680-686. PMID: 9381177. The data was filtered using the steps described in “Gene Expression Profile Analysis” on page 4-95.

Set System Path and Enable Add-In

- 1 If not already done, modify your system path to include the MATLAB root folder as described in the Spreadsheet Link documentation.
- 2 If not already done, enable the Spreadsheet Link Add-In as described in “Add-In Setup” (Spreadsheet Link).
- 3 Close MATLAB and Excel if they are open.
- 4 Start Excel. MATLAB and Spreadsheet Link software automatically start.

Download Spreadsheet with Filtered Yeast Data

- 1 Download the provided `Filtered_Yeastdata.xlsx` and open it in Excel.
- 2 In the Excel software, enable macros. Click the **Developer** tab, and then select **Macro Security** from the Code group. If the **Developer** tab is not displayed on the Excel ribbon, consult Excel Help to display it. If you encounter the “Can't find project or library” error, you might need to update the references in the Visual Basic software. Open Visual Basic by clicking the **Developer**

tab and selecting **Visual Basic**. Then select **Tools > References > SpreadsheetLink**. If the **MISSING: exlink2007.xlam** check box is selected, clear it.

Run the Example for the Entire Data Set

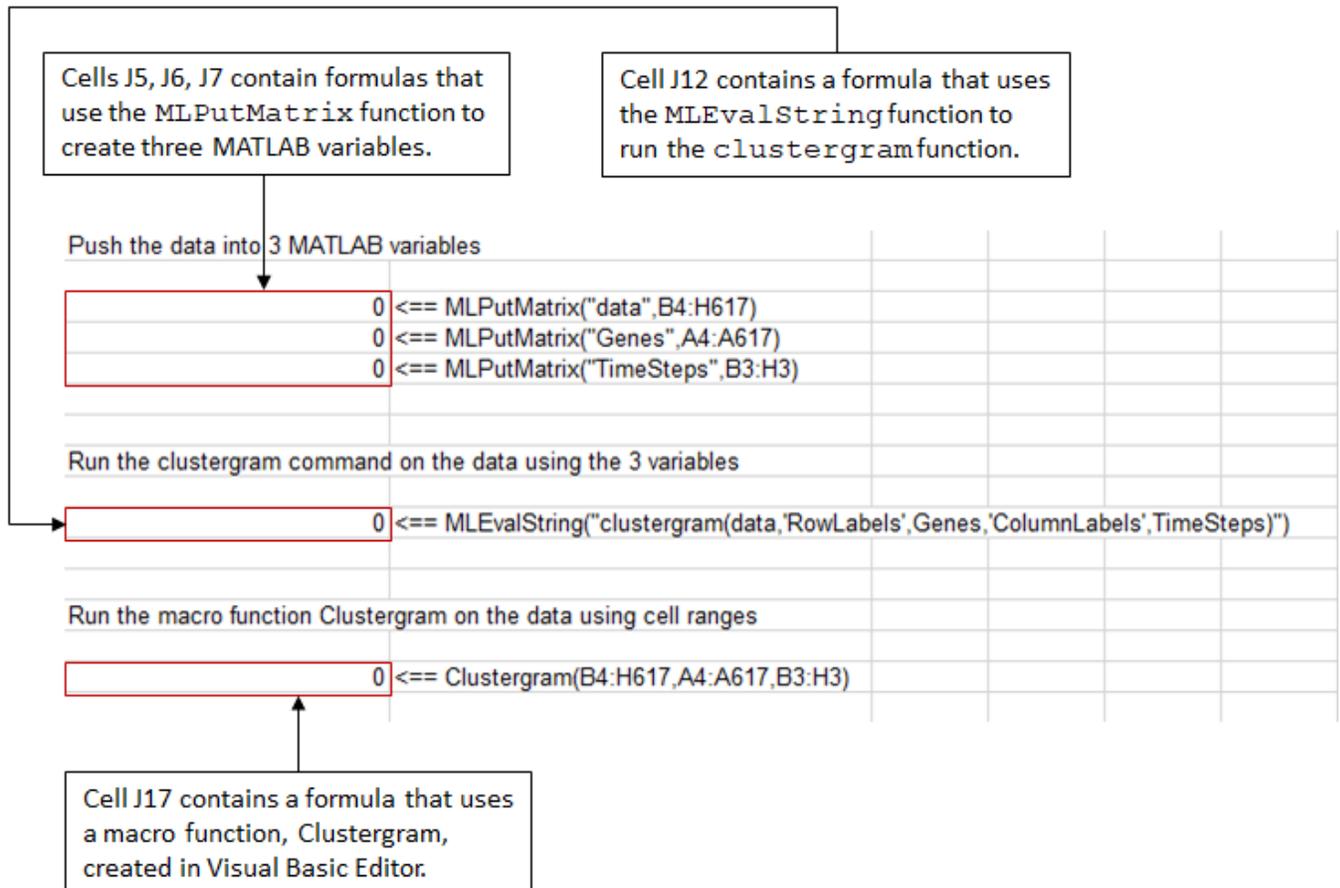
- 1 In the provided Excel file, note that columns A through H contain data from DeRisi et al. Also note that cells J5, J6, J7, and J12 contain formulas using Spreadsheet Link functions `MLPutMatrix` and `MLEvalString`.

Tip To view a cell's formula, select the cell, and then view the formula in the formula bar

 _____ at the top of the Excel window.

- 2 Execute the formulas in cells J5, J6, J7, and J12, by selecting the cell, pressing **F2**, and then pressing **Enter**.

Each of the first three cells contains a formula using the Spreadsheet Link function `MLPutMatrix`, which creates a MATLAB variable from the data in the spreadsheet. Cell J12 contains a formula using the Spreadsheet Link function `MLEvalString`, which runs the Bioinformatics Toolbox `clustergram` function using the three variables as input. For more information on adding formulas using Spreadsheet Link functions, see "Create Diagonal Matrix Using Worksheet Cells" (Spreadsheet Link).



The diagram illustrates the following Excel formulas and their descriptions:

- Cells J5, J6, J7:** Contain formulas that use the `MLPutMatrix` function to create three MATLAB variables.


```
0 <== MLPutMatrix("data",B4:H617)
0 <== MLPutMatrix("Genes",A4:A617)
0 <== MLPutMatrix("TimeSteps",B3:H3)
```
- Cell J12:** Contains a formula that uses the `MLEvalString` function to run the `clustergram` function.


```
0 <== MLEvalString("clustergram(data,'RowLabels',Genes,'ColumnLabels',TimeSteps)")
```
- Cell J17:** Contains a formula that uses a macro function, `Clustergram`, created in Visual Basic Editor.


```
0 <== Clustergram(B4:H617,A4:A617,B3:H3)
```

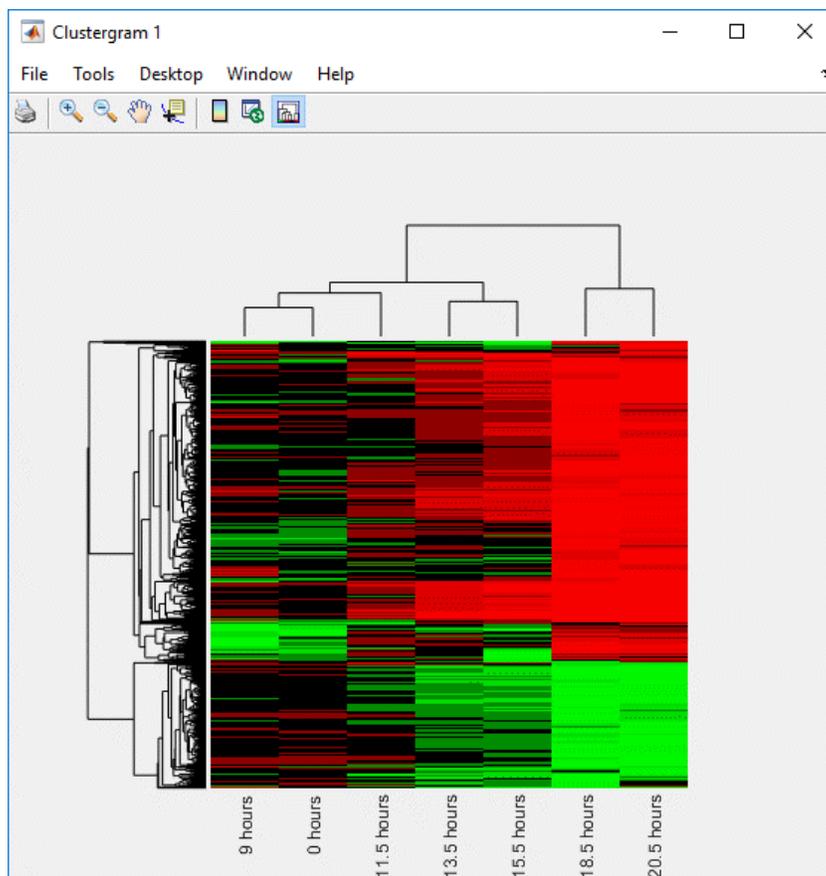
- 3 Note that cell J17 contains a formula using a macro function Clustergram, which was created in the Visual Basic® Editor. Running this macro does the same as the formulas in cells J5, J6, J7, and J12. Optionally, view the Clustergram macro function by clicking the **Developer** tab, and then

clicking the Visual Basic button . (If the **Developer** tab is not on the Excel ribbon, consult Excel Help to display it.)

For more information on creating macros using Visual Basic Editor, see “Create Diagonal Matrix Using VBA Macro” (Spreadsheet Link).

- 4 Execute the formula in cell J17 to analyze and visualize the data:
 - a Select cell **J17**.
 - b Press **F2**.
 - c Press **Enter**.

The macro function Clustergram runs creating three MATLAB variables (data, Genes, and TimeSteps) and displaying a Clustergram window containing dendrograms and a heat map of the data.



Edit Formulas to Run the Example on a Subset of the Data

- 1 Edit the formulas in cells J5 and J6 to analyze a subset of the data. Do this by editing the formulas' cell ranges to include data for only the first 30 genes:

- a Select cell **J5**, and then press **F2** to display the formula for editing. Change **H617** to **H33**, and then press **Enter**.

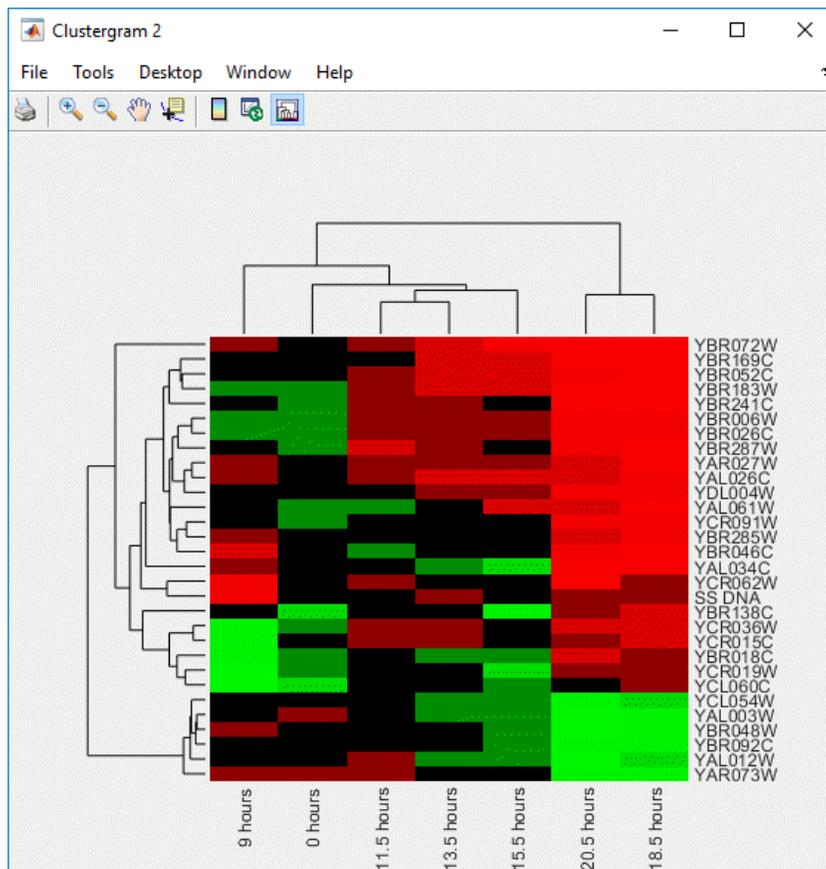
```
=MLPutMatrix("data",B4:H33)
```

- b Select cell **J6**, then press **F2** to display the formula for editing. Change **A617** to **A33**, and then press **Enter**.

```
=MLPutMatrix("Genes",A4:A33)
```

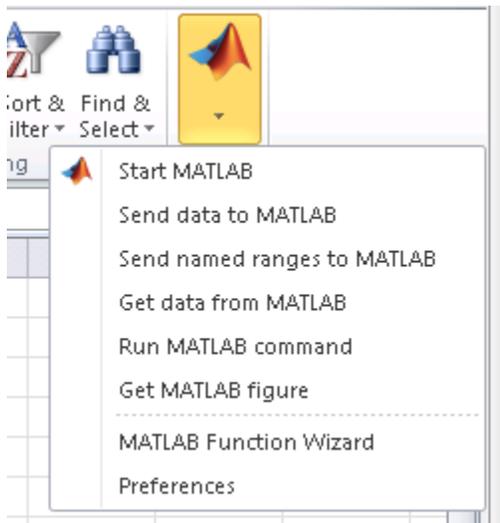
- 2 Run the formulas in cells J5, J6, J7, and J12 to analyze and visualize a subset of the data:

- a Select cell **J5**, press **F2**, and then press **Enter**.
 b Select cell **J6**, press **F2**, and then press **Enter**.
 c Select cell **J7**, press **F2**, and then press **Enter**.
 d Select cell **J12**, press **F2**, and then press **Enter**.



Use the Spreadsheet Link product to Interact With the Data in MATLAB

Use the MATLAB group on the right side of the **Home** tab to interact with the data:



For example, create a variable in MATLAB containing a 3-by-7 matrix of the data, plot the data in a Figure window, and then add the plot to your spreadsheet:

- 1 Click-drag to select cells **B5** through **H7**.

0.305	0.146	-0.129	-0.444	-0.707	-1.499	-1.935
0.157	0.175	0.467	-0.379	-0.52	-1.279	-2.125
0.246	0.796	0.384	0.981	1.02	1.646	1.157

- 2 From the MATLAB group, select **Send data to MATLAB**.
- 3 Type **YAGenes** for the variable name, and then click **OK**.

The variable **YAGenes** is added to the MATLAB Workspace as a 3-by-7 matrix.

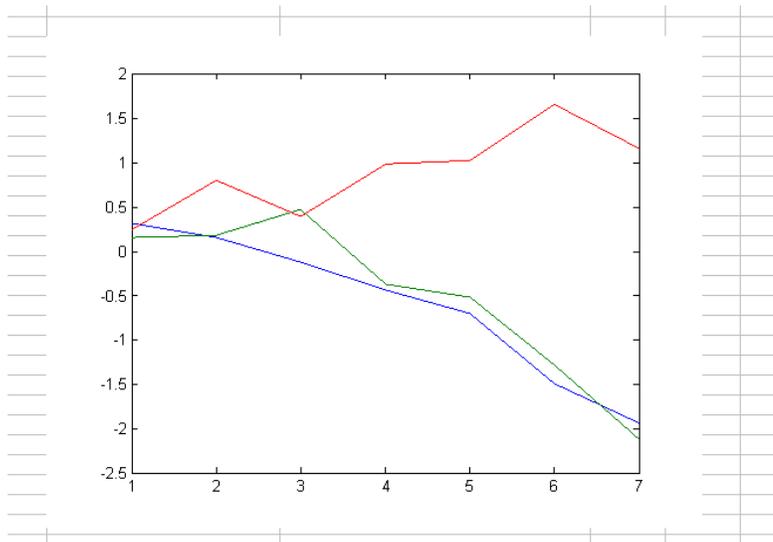
- 4 From the MATLAB group, select **Run MATLAB command**.
- 5 Type **plot(YAGenes')** for the command, and then click **OK**.

A Figure window displays a plot of the data.

Note Make sure you use the ' (transpose) symbol when plotting the data in this step. You need to transpose the data in YAGenes so that it plots as three genes over seven time intervals.

- 6 Select cell **J20**, and then click from the MATLAB group, select **Get MATLAB figure**.

The figure is added to the spreadsheet.



Working with Whole Genome Data

This example shows how to create a memory mapped file for sequence data and work with it without loading all the genomic sequence into memory. Whole genomes are available for human, mouse, rat, fugu, and several other model organisms. For many of these organisms one chromosome can be several hundred million base pairs long. Working with such large data sets can be challenging as you may run into limitations of the hardware and software that you are using. This example shows one way to work around these limitations in MATLAB®.

Large Data Set Handling Issues

Solving technical computing problems that require processing and analyzing large amounts of data puts a high demand on your computer system. Large data sets take up significant memory during processing and can require many operations to compute a solution. It can also take a long time to access information from large data files.

Computer systems, however, have limited memory and finite CPU speed. Available resources vary by processor and operating system, the latter of which also consumes resources. For example:

32-bit processors and operating systems can address up to $2^{32} = 4,294,967,296 = 4$ GB of memory (also known as virtual address space). Windows® XP and Windows® 2000 allocate only 2 GB of this virtual memory to each process (such as MATLAB). On UNIX®, the virtual memory allocated to a process is system-configurable and is typically around 3 GB. The application carrying out the calculation, such as MATLAB, can require storage in addition to the user task. The main problem when handling large amounts of data is that the memory requirements of the program can exceed that available on the platform. For example, MATLAB generates an "out of memory" error when data requirements exceed approximately 1.7 GB on Windows XP.

For more details on memory management and large data sets, see "Profile and Improve Performance".

On a typical 32-bit machine, the maximum size of a single data set that you can work with in MATLAB is a few hundred MB, or about the size of a large chromosome. Memory mapping of files allows MATLAB to work around this limitation and enables you to work with very large data sets in an intuitive way.

Whole Genome Data Sets

The latest whole genome data sets can be downloaded from the Ensembl Website. The data are provided in several formats. These are updated regularly as new sequence information becomes available. This example will use human DNA data stored in FASTA format. Chromosome 1 is (in the GRCh37.56 Release of September 2009) a 65.6 MB compressed file. After uncompressing the file it is about 250MB. MATLAB uses 2 bytes per character, so if you read the file into MATLAB, it will require about 500MB of memory.

This example assumes that you have already downloaded and uncompressed the FASTA file into your local directory. Change the name of the variable `FASTAfilename` if appropriate.

```
FASTAfilename = 'Homo_sapiens.GRCh37.56.dna.chromosome.1.fa';  
fileInfo = dir(which(FASTAfilename));
```

Memory Mapped Files

Memory mapping allows MATLAB to access data in a file as though it is in memory. You can use standard MATLAB indexing operations to access data. See the documentation for `memmapfile` for more details.

You could just map the FASTA file and access the data directly from there. However the FASTA format file includes new line characters. The `memmapfile` function treats these characters in the same way as all other characters. Removing these before memory mapping the file will make indexing operations simpler. Also, memory mapping does not work directly with character data so you will have to treat the data as 8-bit integers (`uint8` class). The function `nt2int` in the Bioinformatics Toolbox™ can be used to convert character information into integer values. `int2nt` is used to convert back to characters.

First open the FASTA file and extract the header.

```
fidIn = fopen(FASTAfilename, 'r');
header = fgetl(fidIn)

header =

    '>1 dna:chromosome chromosome:GRCh37:1:1:249250621:1'
```

Open the file to be memory mapped.

```
[fullPath, filename, extension] = fileparts(FASTAfilename);
mmFilename = [filename '.mm']
fidOut = fopen(mmFilename, 'w');

mmFilename =

    'Homo_sapiens.GRCh37.56.dna.chromosome.1.mm'
```

Read the FASTA file in blocks of 1MB, remove new line characters, convert to `uint8`, and write to the MM file.

```
newLine = sprintf('\n');
blockSize = 2^20;
while ~feof(fidIn)
    % Read in the data
    charData = fread(fidIn, blockSize, '*char');
    % Remove new lines
    charData = strrep(charData, newLine, '');
    % Convert to integers
    intData = nt2int(charData);
    % Write to the new file
    fwrite(fidOut, intData, 'uint8');
end
```

Close the files.

```
fclose(fidIn);
fclose(fidOut);
```

The new file is about the same size as the old file but does not contain new lines or the header information.

```
mmfileInfo = dir(mmFilename);
```

Accessing the Data in the Memory Mapped File

The command `memmapfile` constructs a `memmapfile` object that maps the new file to memory. In order to do this, it needs to know the format of the file. The format of this file is simple, though much more complicated formats can be mapped.

```
chr1 = memmapfile(mmFilename, 'format', 'uint8');
```

The MEMMAPFILE Object

The `memmapfile` object has various properties. `Filename` stores the full path to the file. `Writable` indicates whether or not the data can be modified. Note that if you do modify the data, this will also modify the original file. `Offset` allows you to specify the space used by any header information. `Format` indicates the data format. `Repeat` is used to specify how many blocks (as defined by `Format`) to map. This can be useful for limiting how much memory is used to create the memory map. These properties can be accessed in the same way as other MATLAB data. For more details see `type help memmapfile` or `doc memmapfile`.

```
chr1.Data(1:10)
```

```
ans =
```

```
10x1 uint8 column vector
```

```
15  
15  
15  
15  
15  
15  
15  
15  
15  
15
```

You can access any region of the data using indexing operations.

```
chr1.Data(10000000:10000010)'
```

```
ans =
```

```
1x11 uint8 row vector
```

```
1 1 2 2 2 2 3 4 2 4 2
```

Remember that the nucleotide information was converted to integers. You can use `int2nt` to get the sequence information back.

```
int2nt(chr1.Data(10000000:10000010)')
```

```
ans =
```

```
'AACCCCGTCTC'
```

Or use `seqdisp` to display the sequence.

```
seqdisp(chr1.Data(10000000:10001000)')
```

```
ans =
```

```
17×71 char array
```

```
'  1 AACCCCGTCT CTACAATAAA TTAAAATATT AGCTGGGCAT GGTGGTGTGT GCTTGTAGTC '
' 61 CCAGCTACTT GGCGGGCTGA GGTGGGAGAA TCATCCAAGC CTTGGAGGCA GAGGTTGCAG '
'121 TGAGCTGAGA TTGTGACACT GCACTCCAGC CTGGGAGACA GAGTGAGACT CCTACTCAAA '
'181 AAAAAACAAA AAACAAAAAA CAAACCACAA AACTTTCCAG GTAACCTTATT AAAACATGTT '
'241 TTTTGTGTTG TTTGAGACAG AGTCTTGCTC TGTCGCCAG GCTGGAGTGC AGTGGAGCAA '
'301 TCTCAGCTCA CTGCAAGCTC CGCCTCCCGG GTTCACACCA TTCTCCTGCC TCAGCCTCCC '
'361 GAGTAGCTAG GACTATAGGC ACCCGCCACC AGGCCAGCT TATTTTTTTT GTATTTTTTA '
'421 GTAGAGACGG GGTTTCATCG TGTTAGCCAG GATGGTCTCG ATCTCCTGAC CTCGTGATCC '
'481 GCCCACCTCA GCCTCCCAA GTGCTGGGAT TACAGGCGTG AGCCACTGCA CCCGGCCTAG '
'541 TTTTGTGATA TTTTGTGATA TAGAGACAGG GTTTCACCAT GTTAGCCAGG ATGGTCTCAA '
'601 TCTCCTGACC TCGTGATCCG CCCGCCTCGG CCTCCCAAAG TGCTGGGGTT ACAGGCCTGA '
'661 GCCACGCGAC ACAGCATTAA AGCATGTTTT ATTTTCCTAC ACATAATGAA ATCATTACCA '
'721 GATGATTTGA CATGTGTACT TCATTGGAGA GGATTCTTAC AGTATATTCA AAATTAATA '
'781 TAATGACAAA AAATTACTAC CTAATCTATT AAAATTGGCA TAAGTCATCT ATGATCATT '
'841 ATGATATGCA AACATAACA AGTATTATAC CCAGAAGTGT AATTTATTGT AGCTACATCT '
'901 TATGTATAAT AGTTTAGTGG ATTTTCTCTG GAAATTGTCC ATTTTAATTT TTCTCTTAAG '
'961 TCTGTGGAAT TTTCCAGTAA AAGTCAAGGC AAACCCAAGA T'
```

Analysis of the Whole Chromosome

Now that you can easily access the whole chromosome, you can analyze the data. This example shows one way to look at the GC content along the chromosome.

You extract blocks of 500000bp and calculate the GC content.

Calculate how many blocks to use.

```
numNT = numel(chr1.Data);
blockSize = 500000;
numBlocks = floor(numNT/blockSize);
```

One way to help MATLAB performance when working with large data sets is to "preallocate" space for data. This allows MATLAB to allocate enough space for all of the data rather than having to grow the array in small chunks. This will speed things up and also protect you from problems of the data getting too large to store. For more details on pre-allocating arrays, see: <https://www.mathworks.com/matlabcentral/answers/99124-how-do-i-pre-allocate-memory-when-using-matlab>

An easy way to preallocate an array is to use the `zeros` function.

```
ratio = zeros(numBlocks+1,1);
```

Loop over the data looking for C or G and then divide this number by the total number of A, T, C, and G. This will take about a minute to run.

```
A = nt2int('A'); C = nt2int('C'); G = nt2int('G'); T = nt2int('T');
```

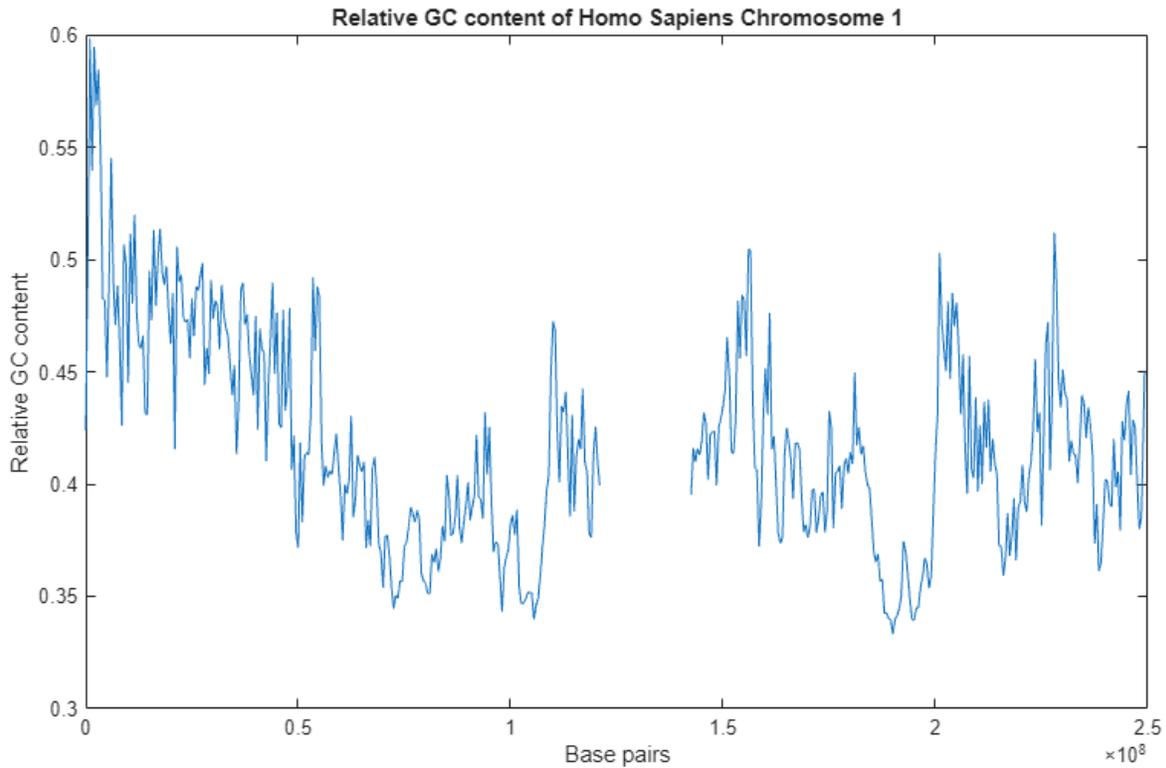
```
for count = 1:numBlocks
    % calculate the indices for the block
    start = 1 + blockSize*(count-1);
    stop = blockSize*count;
    % extract the block
    block = chr1.Data(start:stop);
    % find the GC and AT content
    gc = (sum(block == G | block == C));
    at = (sum(block == A | block == T));
    % calculate the ratio of GC to the total known nucleotides
    ratio(count) = gc/(gc+at);
end
```

The final block is smaller so treat this as a special case.

```
block = chr1.Data(stop+1:end);
gc = (sum(block == G | block == C));
at = (sum(block == A | block == T));
ratio(end) = gc/(gc+at);
```

Plot of the GC Content for the Homo Sapiens Chromosome 1

```
xAxis = [1:blockSize:numBlocks*blockSize, numNT];
plot(xAxis,ratio)
xlabel('Base pairs');
ylabel('Relative GC content');
title('Relative GC content of Homo Sapiens Chromosome 1')
```



The region in the center of the plot around 140Mbp is a large region of Ns.

```
seqdisp(chr1.Data(140000000:140001000))
```

```
ans =
```

```
17x71 char array
```

```
'  1  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
' 61  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'121  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'181  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'241  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'301  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'361  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'421  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'481  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'541  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'601  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'661  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'721  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'781  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'841  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'901  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
'961  NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN NNNNNNNNNN '
```

Finding Regions of High GC Content

You can use `find` to identify regions of high GC content.

```
indices = find(ratio>0.5);  
ranges = [(1 + blockSize*(indices-1)), blockSize*indices];  
fprintf('Region %d:%d has GC content %f\n',[ranges ,ratio(indices)]')
```

```
Region 500001:1000000 has GC content 0.501412  
Region 1000001:1500000 has GC content 0.598332  
Region 1500001:2000000 has GC content 0.539498  
Region 2000001:2500000 has GC content 0.594508  
Region 2500001:3000000 has GC content 0.568620  
Region 3000001:3500000 has GC content 0.584572  
Region 3500001:4000000 has GC content 0.548137  
Region 6000001:6500000 has GC content 0.545072  
Region 9000001:9500000 has GC content 0.506692  
Region 10500001:11000000 has GC content 0.511386  
Region 11500001:12000000 has GC content 0.519874  
Region 16000001:16500000 has GC content 0.513082  
Region 17500001:18000000 has GC content 0.513392  
Region 21500001:22000000 has GC content 0.505598  
Region 156000001:156500000 has GC content 0.504446  
Region 156500001:157000000 has GC content 0.504090  
Region 201000001:201500000 has GC content 0.502976  
Region 228000001:228500000 has GC content 0.511946
```

If you want to remove the temporary file, you must first clear the `memmapfile` object.

```
clear chr1  
delete(mmFilename)
```

Comparing Whole Genomes

This example shows how to compare whole genomes for organisms, which allows you to compare the organisms at a very different resolution relative to single gene comparisons. Instead of just focusing on the differences between homologous genes you can gain insight into the large-scale features of genomic evolution.

This example uses two strains of Chlamydia, *Chlamydia trachomatis* and *Chlamydophila pneumoniae*. These are closely related bacteria that cause different, though both very common, diseases in humans. Whole genomes are available in the GenBank® database for both organisms.

Retrieving the Genomes

You can download these genomes using the `getgenbank` function. First, download the *Chlamydia trachomatis* genome. Notice that the genome is circular and just over one million bp in length. These sequences are quite large so may take a while to download.

```
seqtrachomatis = getgenbank('NC_000117');
```

Next, download *Chlamydophila pneumoniae*. This genome is also circular and a little longer at 1.2 Mbp.

```
seqpneumoniae = getgenbank('NC_002179');
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated. Hence, the results of this example might be slightly different when you use up-to-date datasets.

```
load('chlamydia.mat','seqtrachomatis','seqpneumoniae')
```

A very simple approach for comparing the two genomes is to perform pairwise alignment between all genes in the genomes. Given that these are bacterial genomes, a simple approach would be to compare all ORFs in the two genomes. However, the GenBank data includes more information about the genes in the sequences. This is stored in the CDS field of the data structure. *Chlamydia trachomatis* has 895 coding regions, while *Chlamydophila pneumoniae* has 1112.

```
M = numel(seqtrachomatis.CDS)
N = numel(seqpneumoniae.CDS)
```

```
M =
    895
```

```
N =
   1112
```

Most of the CDS records contain the translation to amino acid sequences. The first CDS record in the *Chlamydia trachomatis* data is a hypothetical protein of length 591 residues.

```
seqtrachomatis.CDS(1)
```

```
ans =  
  
  struct with fields:  
  
    location: 'join(1041920..1042519,1..1176)'  
    gene: []  
    product: 'hypothetical protein'  
    codon_start: '1'  
    indices: [1041920 1042519 1 1176]  
    protein_id: 'NP_219502.1'  
    db_xref: 'GeneID:884145'  
    note: []  
    translation: 'MSIRGVGGNGNSRIPSHNGDGSNRRSQNTKGNKVEDRVCSLYSSRSNENRESPYAVVDVSSMIESTPTSGETTRASR  
    text: [19x58 char]
```

The fourth CDS record is for the *gatA* gene, which has product glutamyl-tRNA amidotransferase subunit A. The length of the product sequence is 491 residues.

```
seqtrachomatis.CDS(4)
```

```
ans =  
  
  struct with fields:  
  
    location: '2108..3583'  
    gene: 'gatA'  
    product: [2x47 char]  
    codon_start: '1'  
    indices: [2108 3583]  
    protein_id: 'NP_219505.1'  
    db_xref: 'GeneID:884087'  
    note: [7x58 char]  
    translation: 'MYRKSALRLDAVVNRELSVTAITEYFYHRIESHDEQIGAFSLCKERALLRASRIDDKLAKGDPIGLLAGIPIGVKDI  
    text: [26x58 char]
```

A few of the *Chlamydomonas reinhardtii* CDS have empty translations. Fill them in as follows. First, find all empty translations, then display the first empty translation.

```
missingPn = find(cellfun(@isempty,{seqpneumoniae.CDS.translation}));  
seqpneumoniae.CDS(missingPn(1))
```

```
ans =  
  
  struct with fields:  
  
    location: 'complement(73364..73477)'  
    gene: []  
    product: 'hypothetical protein'  
    codon_start: '1'  
    indices: [73477 73364]  
    protein_id: 'NP_444613.1'  
    db_xref: 'GeneID:963699'  
    note: 'hypothetical protein; identified by Glimmer2'
```

```
translation: []
text: [10×52 char]
```

The function `featureparse` extracts features, such as the CDS, from the sequence structure. You can then use `cellfun` to apply `nt2aa` to the sequences with missing translations.

```
allCDS = featureparse(seqpneumoniae, 'Feature', 'CDS', 'Sequence', true);
missingSeqs = cellfun(@nt2aa, {allCDS(missingPn).Sequence}, 'uniform', false);
[seqpneumoniae.CDS(missingPn).translation] = deal(missingSeqs{:});
seqpneumoniae.CDS(missingPn(1))
```

```
ans =
```

```
struct with fields:
```

```
location: 'complement(73364..73477)'
gene: []
product: 'hypothetical protein'
codon_start: '1'
indices: [73477 73364]
protein_id: 'NP_444613.1'
db_xref: 'GeneID:963699'
note: 'hypothetical protein; identified by Glimmer2'
translation: 'MLTDQRKHIQMLHKHNSIEIFLSNMVVEVKLFFKTLK*'
text: [10×52 char]
```

Performing Gene Comparisons

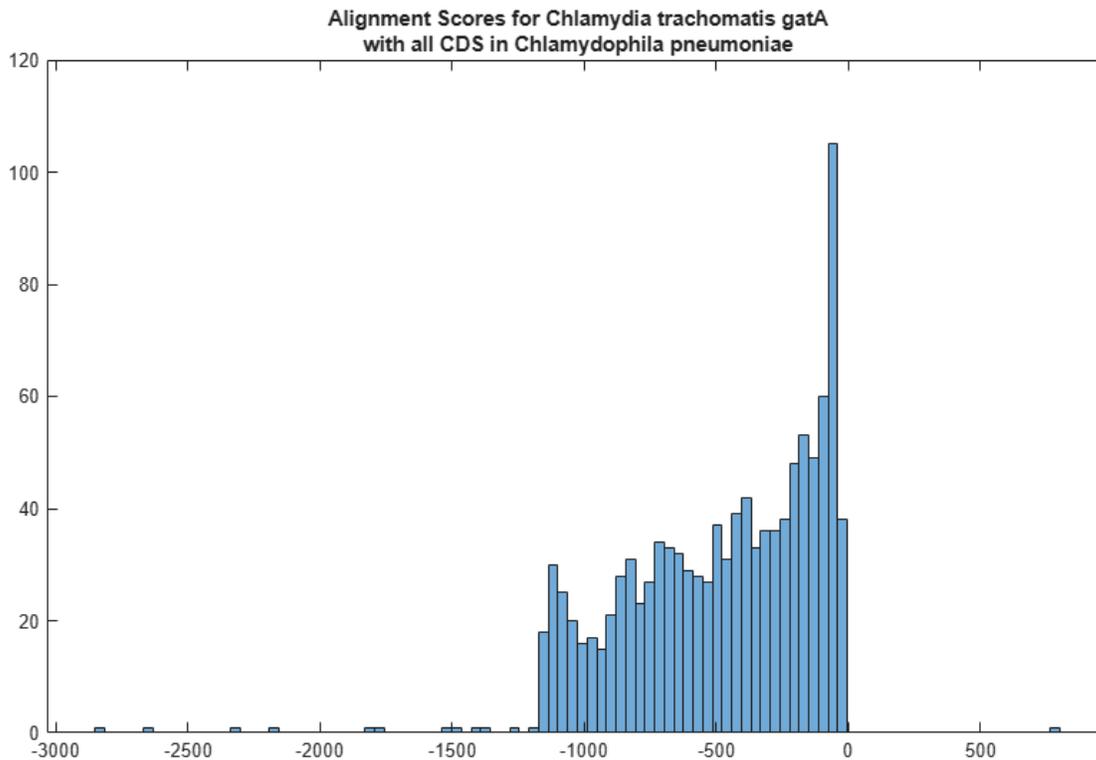
To compare the `gatA` gene in *Chlamydia trachomatis* with all the CDS genes in *Chlamydophila pneumoniae*, put a for loop around the `nwalign` function. You could alternatively use local alignment (`swalign`).

```
tic
gatAScores = zeros(1,N);
for inner = 1:N
    gatAScores(inner) = nwalign(seqtrachomatis.CDS(4).translation,...
    seqpneumoniae.CDS(inner).translation);
end
toc % |tic| and |toc| are used to report how long the calculation takes.
```

```
Elapsed time is 2.073756 seconds.
```

A histogram of the scores shows a large number of negative scores and one very high positive score.

```
histogram(gatAScores,100)
title(sprintf(['Alignment Scores for Chlamydia trachomatis %s\n',...
'with all CDS in Chlamydophila pneumoniae'],seqtrachomatis.CDS(4).gene))
```



As expected, the high scoring match is with the gatA gene in *Chlamydomphila pneumoniae*.

```
[gatABest, gatABestIdx] = max(gatAScores);
seqpneumoniae.CDS(gatABestIdx)
```

ans =

struct with fields:

```
location: 'complement(838828..840306) '
gene: 'gatA'
product: [2×47 char]
codon_start: '1'
indices: [840306 838828]
protein_id: 'NP_445311.1'
db_xref: 'GeneID:963139'
note: [7×58 char]
translation: 'MYRYSALELAKAVTLGELTATGVTQHFFHRIEEAEGQVGAFISLCKEQALEQAEIDKKRSRGEPLGKLAGVPVGIKDI'
text: [26×58 char]
```

The pairwise alignment of one gene from *Chlamydia trachomatis* with all genes from *Chlamydomphila pneumoniae* takes just under a minute on an Intel® Pentium 4, 2.0 GHz machine running Windows® XP. To do this calculation for all 895 CDS in *Chlamydia trachomatis* would take about 12 hours on the same machine. Uncomment the following code if you want to run the whole calculation.

```

scores = zeros(M,N);
parfor outer = 1:M
    theScore = zeros(1,N);
    theSeq = seqtrachomatis.CDS(outer).translation;
    for inner = 1:N
        theScore(inner) = ...
            nwalign(theSeq,...
                seqpneumoniae.CDS(inner).translation);
    end
    scores(outer,:) = theScore;
end

```

Note the command `parfor` is used in the outer loop. If your machine is configured to run multiple *labs* then the outer loop will be executed in parallel. For a full understanding of this construct, see `doc parfor`.

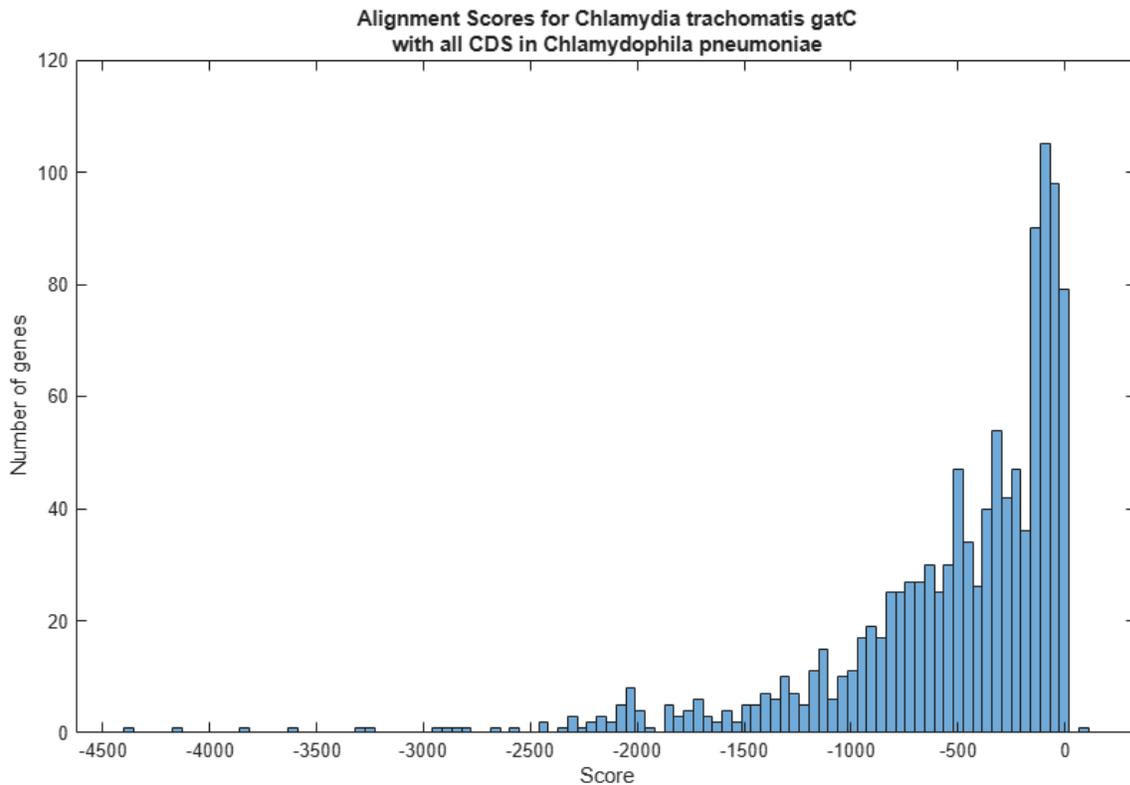
Investigating the Meaning of the Scores

The distributions of the scores for several genes show a pattern. The CDS(3) of *Chlamydia trachomatis* is the `gatC` gene. This has a relatively short product, aspartyl/glutamyl-tRNA amidotransferase subunit C, with only 100 residues.

```

gatCScores = zeros(1,N);
for inner = 1:N
    gatCScores(inner) = nwalign(seqtrachomatis.CDS(3).translation,...
        seqpneumoniae.CDS(inner).translation);
end
figure
histogram(gatCScores,100)
title(sprintf(['Alignment Scores for Chlamydia trachomatis %s\n',...
    'with all CDS in Chlamydophila pneumoniae'],seqtrachomatis.CDS(3).gene))
xlabel('Score');ylabel('Number of genes');

```



The best score again corresponds to the same gene in the *Chlamydomophila pneumoniae*.

```
[gatCBest, gatCBestIdx] = max(gatCScores);
seqpneumoniae.CDS(gatCBestIdx).product
```

```
ans =
```

```
2×47 char array
```

```
'aspartyl/glutamyl-tRNA amidotransferase subunit'  
'C'
```

CDS(339) of *Chlamydia trachomatis* is the *uvrA* gene. This has a very long product, excinuclease ABC subunit A, of length 1786.

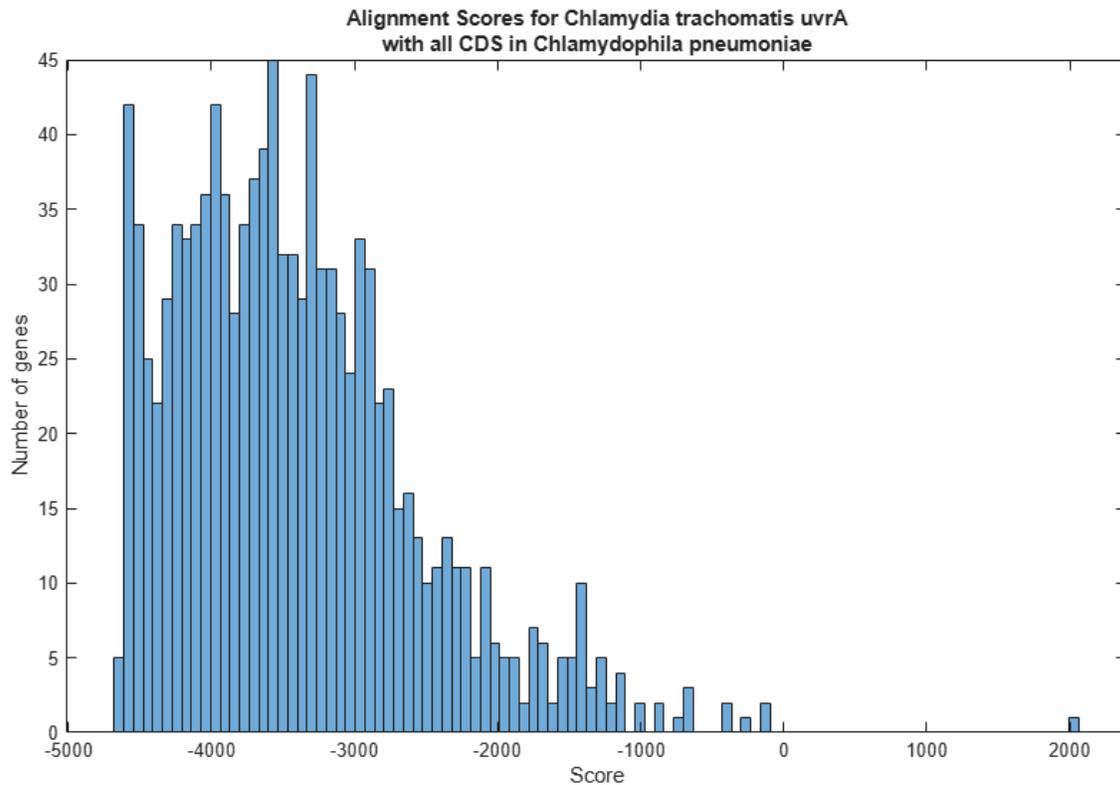
```
uvrAScores = zeros(1,N);
for inner = 1:N
    uvrAScores(inner) = nalign(seqtrachomatis.CDS(339).translation,...
        seqpneumoniae.CDS(inner).translation);
end
figure
histogram(uvrAScores,100)
title(sprintf(['Alignment Scores for Chlamydia trachomatis %s\n',...
    'with all CDS in Chlamydomophila pneumoniae'],seqtrachomatis.CDS(339).gene))
xlabel('Score');ylabel('Number of genes');
```

```
[uvrABest, uvrABestIdx] = max(uvrAScores);
seqpneumoniae.CDS(uvrABestIdx)
```

```
ans =
```

```
struct with fields:
```

```
location: '716887..722367'
gene: []
product: 'excinuclease ABC subunit A'
codon_start: '1'
indices: [716887 722367]
protein_id: 'NP_445220.1'
db_xref: 'GeneID:963214'
note: [6×58 char]
translation: 'MKSLPVYVSGIKVRNLKNVSIHFNSEEIVLLTGVVSGSGKSSIAFDTLYAAGRKRYISTLPTFFATTITTLPNPKVEEII'
text: [46×58 char]
```



The distribution of the scores is affected by the length of the sequences, with very long sequences potentially having much higher or lower scores than shorter sequences. You can normalize for this in a number of ways. One way is to divide by the length of the sequences.

```
lnormgatABest = gatABest./length(seqtrachomatis.CDS(4).product)
lnormgatCBest = gatCBest./length(seqtrachomatis.CDS(3).product)
lnormuvrABest = uvrABest./length(seqtrachomatis.CDS(339).product)
```

```
lnormgatABest =
```

```
16.8794
```

```
lnormgatCBest =
```

```
2.2695
```

```
lnormuvrABest =
```

```
78.9615
```

An alternative normalization method is to use the self alignment score, that is the score from aligning the sequence with itself.

```
gatASelf = nwalign(seqtrachomatis.CDS(4).translation,...  
    seqtrachomatis.CDS(4).translation);  
gatCSelf = nwalign(seqtrachomatis.CDS(3).translation,...  
    seqtrachomatis.CDS(3).translation);  
uvrASelf = nwalign(seqtrachomatis.CDS(339).translation,...  
    seqtrachomatis.CDS(339).translation);  
normgatABest = gatABest./gatASelf  
normgatCBest = gatCBest./gatCSelf  
normuvrABest = uvraBest./uvrASelf
```

```
normgatABest =
```

```
0.7380
```

```
normgatCBest =
```

```
0.5212
```

```
normuvrABest =
```

```
0.5253
```

Using Sparse Matrices to Reduce Memory Usage

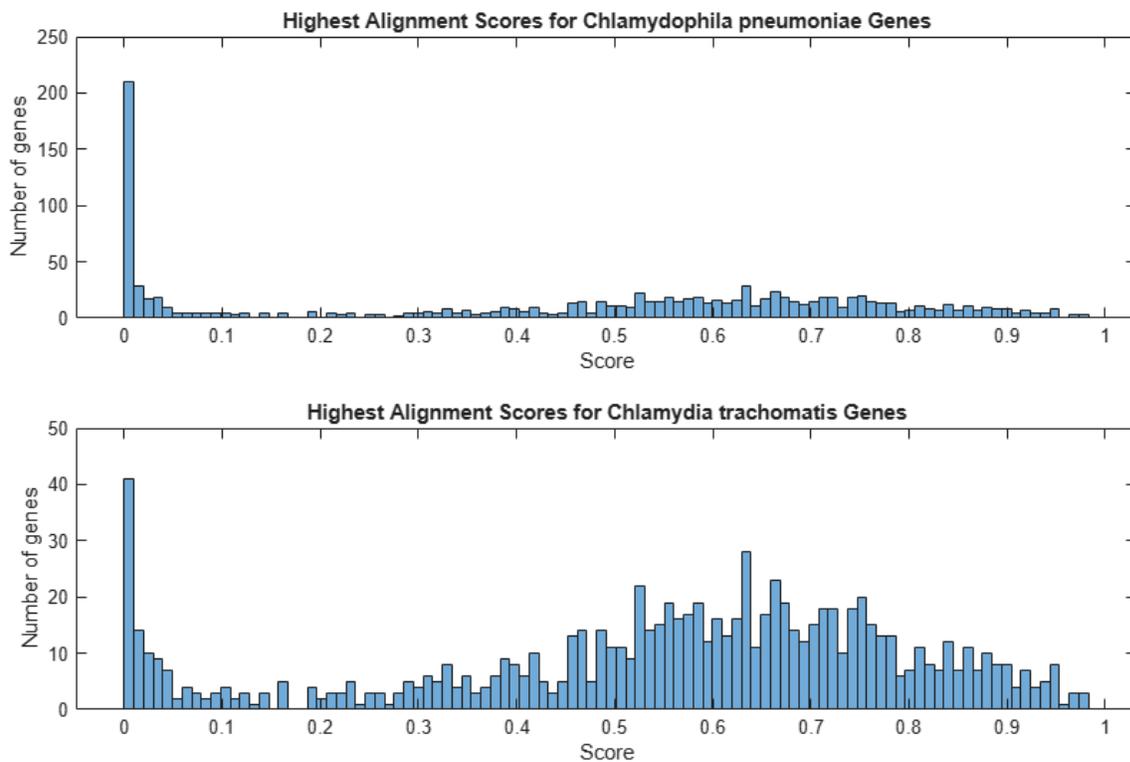
The all-against-all alignment calculation not only takes a lot of time, it also generates a large matrix of scores. If you are looking for similar genes across species, then the scores that are interesting are the positive scores that indicate good alignment. However, most of these scores are negative, and the actual values are not particularly useful for this type of study. Sparse matrices allow you to store the interesting values in a more efficient way.

The sparse matrix, `spScores`, in the MAT-file `chlamydia.mat` contains the positive values from the all against all pairwise alignment calculation normalized by self-alignment score.

```
load('chlamydia.mat','spScores')
```

With the matrix of scores you can look at the distribution of scores of *Chlamydomophila pneumoniae* genes aligned with *Chlamydia trachomatis* and the converse of this, *Chlamydia trachomatis* genes aligned with *Chlamydomophila pneumoniae* genes

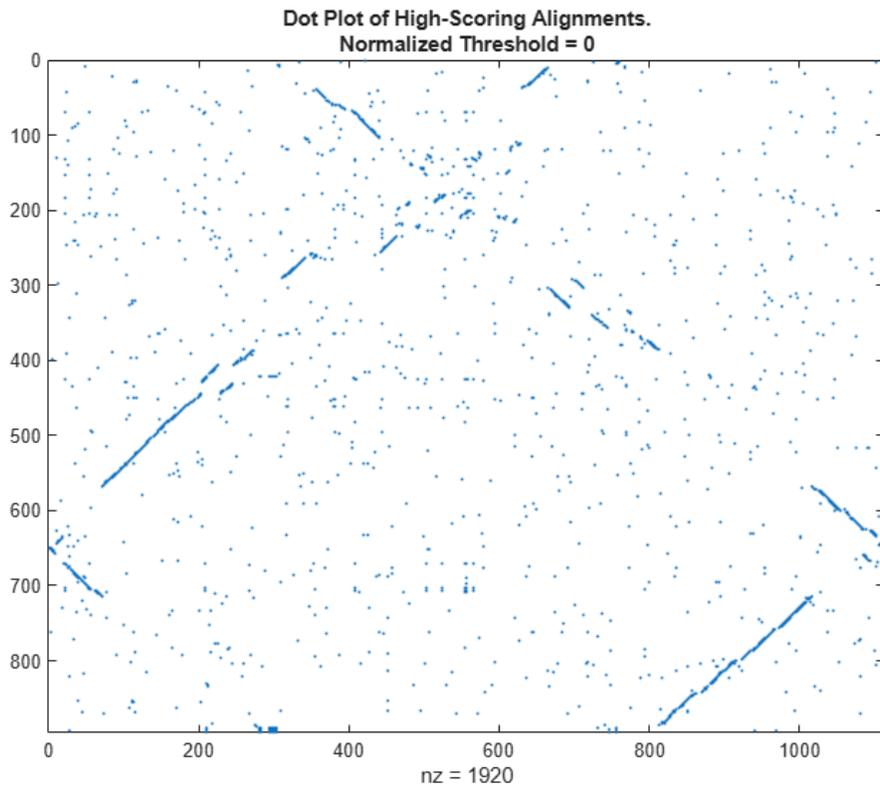
```
figure
subplot(2,1,1)
histogram(max(spScores),100)
title('Highest Alignment Scores for Chlamydomophila pneumoniae Genes')
xlabel('Score');ylabel('Number of genes');
subplot(2,1,2)
histogram(max(spScores,[],2),100)
title('Highest Alignment Scores for Chlamydia trachomatis Genes')
xlabel('Score');ylabel('Number of genes');
```



Remember that there are 1112 CDS in *Chlamydomophila pneumoniae* and only 895 in *Chlamydia trachomatis*. The high number of zero scores in the top histogram indicates that many of the extra CDS in *Chlamydomophila pneumoniae* do not have good matches in *Chlamydia trachomatis*.

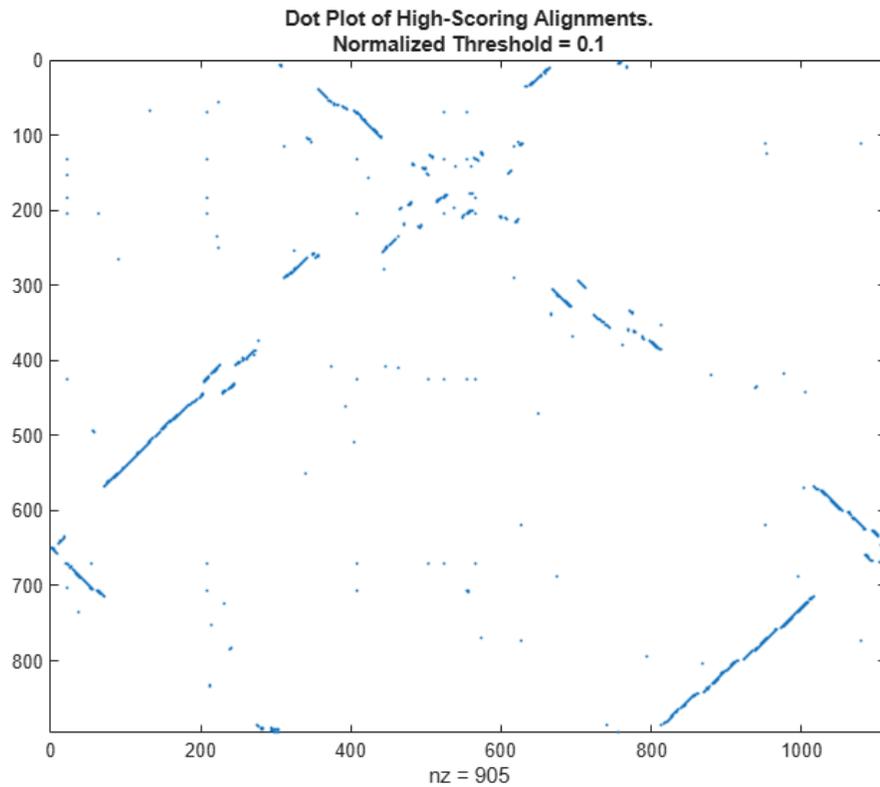
Another way to visualize the data is to look at the positions of points in the scores matrix that are positive. The sparse function `spy` is an easy way to quickly view dotplots of matrices. This shows some interesting structure in the positions of the high scoring matches.

```
figure
spy(spScores > 0)
title(sprintf('Dot Plot of High-Scoring Alignments.\nNormalized Threshold = 0'))
```



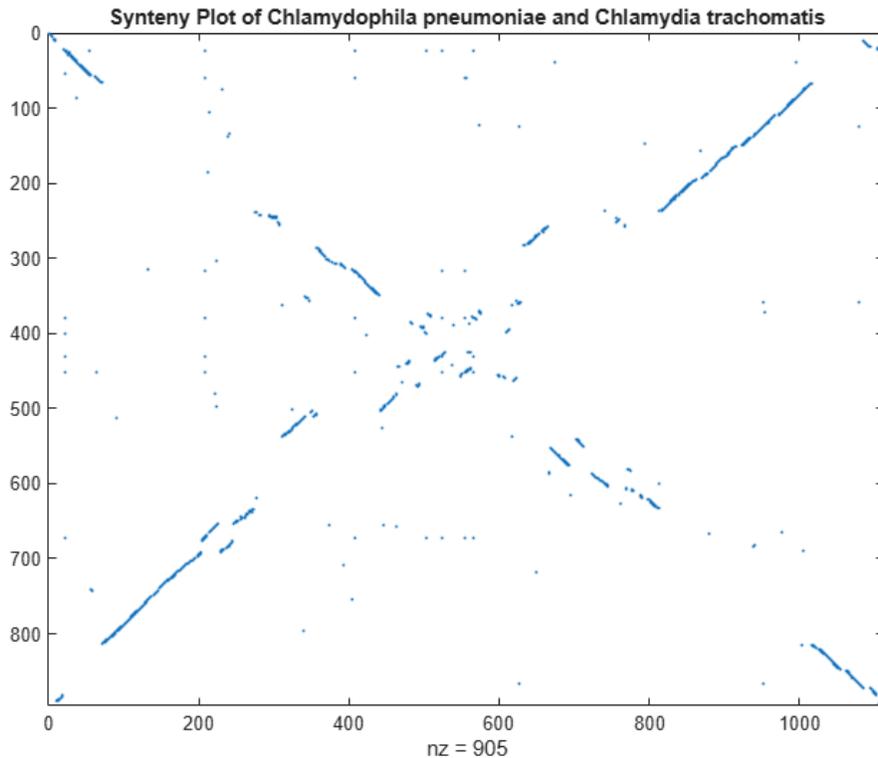
Raise the threshold a little higher to see clear diagonal lines in the plot.

```
spy(spScores > .1)
title(sprintf('Dot Plot of High-Scoring Alignments.\nNormalized Threshold = 0.1'))
```



Remember that these are circular genomes, and it seems that the starting points in GenBank are arbitrary. Permute the scores matrix so that the best match of the first CDS in *Chlamydomonas reinhardtii* is in the first row to see a clear diagonal plot. This shows the synteny between the two organisms.

```
[bestScore bestMatch] = max(spScores(:,1));
spy(spScores([bestMatch:end 1:bestMatch-1],:)>.1);
title('Synteny Plot of Chlamydomonas reinhardtii and Chlamydia trachomatis')
```



Looking for Homologous Genes

Genes in different genomes that are related to each other are said to be homologous. Similarity can be by speciation (orthologous genes) or by replication (paralogous genes). Having the scoring matrix lets you look for both types of relationships.

The most obvious way to find orthologs is to look for the highest scoring pairing for each gene. If the score is significant then these best reciprocal pairs are very likely to be orthologous.

```
[bestScores, bestIndices] = max(spScores);
```

The variable `bestIndices` contains the index of the best reciprocal pairs for the genes in *Chlamydomonas reinhardtii*. Sort the best scores and create a table to compare the description of the best reciprocal pairs and discover very high similarity between the highest scoring best reciprocal pairs.

```
[orderedScores, permScores] = sort(full(bestScores), 'descend');
matches = [num2cell(orderedScores)', num2cell(bestIndices(permScores))', ...
          num2cell((permScores))', ...
          {seqtrachomatis.CDS(bestIndices(permScores)).product; ...
          seqpneumoniae.CDS((permScores)).product; }'];

for count = 1:7
    fprintf(['Score %f\nChlamydia trachomatis Gene   : %s\n', ...
           'Chlamydomonas reinhardtii Gene : %s\n\n'], ...
          matches{count,1}, matches{count,4}, matches{count,5})
end
```

```

Score 0.982993
Chlamydia trachomatis Gene      : 50S ribosomal protein L36
Chlamydophila pneumoniae Gene  : 50S ribosomal protein L36

Score 0.981818
Chlamydia trachomatis Gene      : 30S ribosomal protein S15
Chlamydophila pneumoniae Gene  : 30S ribosomal protein S15

Score 0.975422
Chlamydia trachomatis Gene      : integration host factor alpha-subunit
Chlamydophila pneumoniae Gene  : integration host factor beta-subunit

Score 0.971647
Chlamydia trachomatis Gene      : 50S ribosomal protein L16
Chlamydophila pneumoniae Gene  : 50S ribosomal protein L16

Score 0.970105
Chlamydia trachomatis Gene      : 30S ribosomal protein S10
Chlamydophila pneumoniae Gene  : 30S ribosomal protein S10

Score 0.969554
Chlamydia trachomatis Gene      : rod shape-determining protein MreB
Chlamydophila pneumoniae Gene  : rod shape-determining protein MreB

Score 0.953654
Chlamydia trachomatis Gene      : hypothetical protein
Chlamydophila pneumoniae Gene  : hypothetical protein

```

You can use the Variable Editor to look at the data in a spreadsheet format.

```
open('matches')
```

Compare the descriptions to see that the majority of the best reciprocal pairs have identical descriptions.

```
exactMatches = strcmpi(matches(:,4),matches(:,5));
sum(exactMatches)
```

```
ans =
```

```
808
```

Perhaps more interesting are the best reciprocal pairs where the descriptions are not identical. Some are simply differences in how the same gene is described, but others show quite different descriptions.

```

mismatches = matches(~exactMatches,:);
for count = 1:7
    fprintf(['Score %f\nChlamydia trachomatis Gene      : %s\n',...
            'Chlamydophila pneumoniae Gene : %s\n\n'],...
            mismatches{count,1}, mismatches{count,4}, mismatches{count,5})
end

```

```

Score 0.975422
Chlamydia trachomatis Gene      : integration host factor alpha-subunit

```

Chlamydomonas reinhardtii Gene : integration host factor beta-subunit

Score 0.929565

Chlamydia trachomatis Gene : low calcium response D

Chlamydomonas reinhardtii Gene : type III secretion inner membrane protein SctV

Score 0.905000

Chlamydia trachomatis Gene : NrdR family transcriptional regulator

Chlamydomonas reinhardtii Gene : transcriptional regulator NrdR

Score 0.903226

Chlamydia trachomatis Gene : Yop proteins translocation protein S

Chlamydomonas reinhardtii Gene : type III secretion inner membrane protein SctS

Score 0.896212

Chlamydia trachomatis Gene : ATP-dependent protease ATP-binding subunit ClpX

Chlamydomonas reinhardtii Gene : ATP-dependent protease ATP-binding protein ClpX

Score 0.890705

Chlamydia trachomatis Gene : ribonuclease E

Chlamydomonas reinhardtii Gene : ribonuclease G

Score 0.884234

Chlamydia trachomatis Gene : ClpC protease ATPase

Chlamydomonas reinhardtii Gene : ATP-dependent Clp protease ATP-binding protein

View data for mismatches.

```
open('mismatches')
```

Once you have the scoring matrix this opens up many possibilities for further investigation. For example, you could look for CDS where there are multiple high scoring reciprocal CDS. See Cristianini and Hahn [1] for further ideas.

References

[1] Cristianini, N. and Hahn, M.W., "Introduction to Computational Genomics: A Case Studies Approach", Cambridge University Press, 2007.

See Also

getgenbank | nwalignment | featureparse

High-Throughput Sequence Analysis

- “Work with Next-Generation Sequencing Data” on page 2-2
- “Manage Sequence Read Data in Objects” on page 2-6
- “Store and Manage Feature Annotations in Objects” on page 2-16
- “Bioinformatics Toolbox Software Support Packages” on page 2-21
- “Count Features from NGS Reads” on page 2-23
- “Identifying Differentially Expressed Genes from RNA-Seq Data” on page 2-32
- “Visualize NGS Data Using Genomics Viewer App” on page 2-53
- “Exploring Genome-Wide Differences in DNA Methylation Profiles” on page 2-59
- “Exploring Protein-DNA Binding Sites from Paired-End ChIP-Seq Data” on page 2-80
- “Working with Illumina/Solexa Next-Generation Sequencing Data” on page 2-98
- “Bioinformatics Pipeline SplitDimension” on page 2-108
- “Split Input SAM Files and Assemble Transcriptomes Using Bioinformatics Pipeline” on page 2-112
- “Bioinformatics Pipeline Run Mode” on page 2-114
- “Create Simple Pipeline to Plot Sequence Quality Data Using Biopipeline Designer” on page 2-115
- “Count RNA-Seq Reads Using Biopipeline Designer” on page 2-127
- “Bioinformatics Pipeline Block Libraries” on page 2-136
- “Analyze Gene Expression Profiles Using Biopipeline Designer” on page 2-139
- “Predict Protein Secondary Structure Using Biopipeline Designer” on page 2-150

Work with Next-Generation Sequencing Data

In this section...

“Overview” on page 2-2

“What Files Can You Access?” on page 2-2

“Before You Begin” on page 2-3

“Create a BioIndexedFile Object to Access Your Source File” on page 2-3

“Determine the Number of Entries Indexed by a BioIndexedFile Object” on page 2-3

“Retrieve Entries from Your Source File” on page 2-4

“Read Entries from Your Source File” on page 2-4

Overview

Many biological experiments produce huge data files that are difficult to access due to their size, which can cause memory issues when reading the file into the MATLAB Workspace. You can construct a `BioIndexedFile` object to access the contents of a large text file containing nonuniform size entries, such as sequences, annotations, and cross-references to data sets. The `BioIndexedFile` object lets you quickly and efficiently access this data without loading the source file into memory.

You can use the `BioIndexedFile` object to access individual entries or a subset of entries when the source file is too big to fit into memory. You can access entries using indices or keys. You can read and parse one or more entries using provided interpreters or a custom interpreter function.

Use the `BioIndexedFile` object in conjunction with your large source file to:

- Access a subset of the entries for validation or further analysis.
- Parse entries using a custom interpreter function.

What Files Can You Access?

You can use the `BioIndexedFile` object to access large text files.

Your source file can have these application-specific formats:

- FASTA
- FASTQ
- SAM

Your source file can also have these general formats:

- **Table** — Tab-delimited table with multiple columns. Keys can be in any column. Rows with the same key are considered separate entries.
- **Multi-row Table** — Tab-delimited table with multiple columns. Keys can be in any column. Contiguous rows with the same key are considered a single entry. Noncontiguous rows with the same key are considered separate entries.
- **Flat** — Flat file with concatenated entries separated by a character vector, typically `//`. Within an entry, the key is separated from the rest of the entry by a white space.

Before You Begin

Before constructing a `BioIndexedFile` object, locate your source file on your hard drive or a local network.

When you construct a `BioIndexedFile` object from your source file for the first time, you also create an auxiliary index file, which by default is saved to the same location as your source file. However, if your source file is in a read-only location, you can specify a different location to save the index file.

Tip If you construct a `BioIndexedFile` object from your source file on subsequent occasions, it takes advantage of the existing index file, which saves time. However, the index file must be in the same location or a location specified by the subsequent construction syntax.

Tip If insufficient memory is not an issue when accessing your source file, you may want to try an appropriate read function, such as `genbankread`, for importing data from GenBank files.

Additionally, several read functions such as `fastaread`, `fastqread`, `samread`, and `sffread` include a `Blockread` property, which lets you read a subset of entries from a file, thus saving memory.

Create a BioIndexedFile Object to Access Your Source File

To construct a `BioIndexedFile` object from a multi-row table file:

- 1 Create a variable containing the full absolute path of your source file. For your source file, use the `yeastgenes.sgd` file, which is included with the Bioinformatics Toolbox software.
- 2 Use the `BioIndexedFile` constructor function to construct a `BioIndexedFile` object from the `yeastgenes.sgd` source file, which is a multi-row table file. Save the index file in the Current Folder. Indicate that the source file keys are in column 3. Also, indicate that the header lines in the source file are prefaced with `!`, so the constructor ignores them.

```
sourcefile = which('yeastgenes.sgd');
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
    'KeyColumn', 3, 'HeaderPrefix', '!')
```

The `BioIndexedFile` constructor function constructs `gene2goObj`, a `BioIndexedFile` object, and also creates an index file with the same name as the source file, but with an `IDX` extension. It stores this index file in the Current Folder because we specified this location. However, the default location for the index file is the same location as the source file.

Caution Do not modify the index file. If you modify it, you can get invalid results. Also, the constructor function cannot use a modified index file to construct future objects from the associated source file.

Determine the Number of Entries Indexed by a BioIndexedFile Object

To determine the number of entries indexed by a `BioIndexedFile` object, use the `NumEntries` property of the `BioIndexedFile` object. For example, for the `gene2goObj` object:

```
gene2goObj.NumEntries
```

```
ans =
    6476
```

Note For a list and description of all properties of the object, see `BioIndexedFile`.

Retrieve Entries from Your Source File

Retrieve entries from your source file using either:

- The index of the entry
- The entry key

Retrieve Entries Using Indices

Use the `getEntryByIndex` method to retrieve a subset of entries from your source file that correspond to specified indices. For example, retrieve the first 12 entries from the `yeastgenes.sgd` source file:

```
subset_entries = getEntryByIndex(gene2goObj, [1:12]);
```

Retrieve Entries Using Keys

Use the `getEntryByKey` method to retrieve a subset of entries from your source file that are associated with specified keys. For example, retrieve all entries with keys of AAC1 and AAD10 from the `yeastgenes.sgd` source file:

```
subset_entries = getEntryByKey(gene2goObj, {'AAC1' 'AAD10'});
```

The output `subset_entries` is a character vector of concatenated entries. Because the keys in the `yeastgenes.sgd` source file are not unique, this method returns all entries that have a key of AAC1 or AAD10.

Read Entries from Your Source File

The `BioIndexedFile` object includes a `read` method, which you can use to read and parse a subset of entries from your source file. The `read` method parses the entries using an interpreter function specified by the `Interpreter` property of the `BioIndexedFile` object.

Set the Interpreter Property

Before using the `read` method, make sure the `Interpreter` property of the `BioIndexedFile` object is set appropriately.

If you constructed a <code>BioIndexedFile</code> object from ...	The <code>Interpreter</code> property ...
A source file with an application-specific format (FASTA, FASTQ, or SAM)	By default is a handle to a function appropriate for that file type and typically does not require you to change it.

If you constructed a <code>BioIndexedFile</code> object from ...	The Interpreter property ...
A source file with a table, multi-row table, or flat format	By default is <code>[]</code> , which means the interpreter is an anonymous function in which the output is equivalent to the input. You can change this to a handle to a function that accepts a character vector of one or more concatenated entries and returns a structure or an array of structures containing the interpreted data.

There are two ways to set the `Interpreter` property of the `BioIndexedFile` object:

- When constructing the `BioIndexedFile` object, use the `Interpreter` property name/property value pair
- After constructing the `BioIndexedFile` object, set the `Interpreter` property

Note For more information on setting the `Interpreter` property of the object, see `BioIndexedFile`.

Read a Subset of Entries

The `read` method reads and parses a subset of entries that you specify using either entry indices or keys.

Example

To quickly find all the gene ontology (GO) terms associated with a particular gene because the entry keys are gene names:

- 1 Set the `Interpreter` property of the `gene2goObj` `BioIndexedFile` object to a handle to a function that reads entries and returns only the column containing the GO term. In this case the interpreter is a handle to an anonymous function that accepts character vectors and extracts those that start with the characters `GO`.

```
gene2goObj.Interpreter = @(x) regexp(x, 'GO:\d+', 'match')
```

- 2 Read only the entries that have a key of `YAT2`, and return their GO terms.

```
GO_YAT2_entries = read(gene2goObj, 'YAT2')
```

```
GO_YAT2_entries =
```

```
'GO:0004092' 'GO:0005737' 'GO:0006066' 'GO:0006066' 'GO:0009437'
```

Manage Sequence Read Data in Objects

In this section...

“Overview” on page 2-6

“Represent Sequence and Quality Data in a BioRead Object” on page 2-7

“Represent Sequence, Quality, and Alignment/Mapping Data in a BioMap Object” on page 2-8

“Retrieve Information from a BioRead or BioMap Object” on page 2-10

“Set Information in a BioRead or BioMap Object” on page 2-12

“Determine Coverage of a Reference Sequence” on page 2-12

“Construct Sequence Alignments to a Reference Sequence” on page 2-13

“Filter Read Sequences Using SAM Flags” on page 2-14

Overview

High-throughput sequencing instruments produce large amounts of sequence read data that can be challenging to store and manage. Using objects to contain this data lets you easily access, manipulate, and filter the data.

Bioinformatics Toolbox includes two objects for working with sequence read data.

Object	Contains This Information	Construct from One of These
BioRead	<ul style="list-style-type: none"> Sequence headers Read sequences Sequence qualities (base calling) 	<ul style="list-style-type: none"> FASTQ file SAM file FASTQ structure (created using the <code>fastqread</code> function) SAM structure (created using the <code>samread</code> function) Cell arrays containing header, sequence, and quality information (created using the <code>fastqread</code> function)
BioMap	<ul style="list-style-type: none"> Sequence headers Read sequences Sequence qualities (base calling) Sequence alignment and mapping information (relative to a single reference sequence), including mapping quality 	<ul style="list-style-type: none"> SAM file BAM file SAM structure (created using the <code>samread</code> function) BAM structure (created using the <code>bamread</code> function) Cell arrays containing header, sequence, quality, and mapping/alignment information (created using the <code>samread</code> or <code>bamread</code> function)

Represent Sequence and Quality Data in a BioRead Object

Prerequisites

A `BioRead` object represents a collection of sequence reads. Each element in the object is associated with a sequence, sequence header, and sequence quality information.

Construct a `BioRead` object in one of two ways:

- **Indexed** — The data remains in the source file. Constructing the object and accessing its contents is memory efficient. However, you cannot modify object properties, other than the `Name` property. This is the default method if you construct a `BioRead` object from a FASTQ- or SAM-formatted file.
- **In Memory** — The data is read into memory. Constructing the object and accessing its contents is limited by the amount of available memory. However, you can modify object properties. When you construct a `BioRead` object from a FASTQ structure or cell arrays, the data is read into memory. When you construct a `BioRead` object from a FASTQ- or SAM-formatted file, use the `InMemory` name-value pair argument to read the data into memory.

Construct a BioRead Object from a FASTQ- or SAM-Formatted File

Note This example constructs a `BioRead` object from a FASTQ-formatted file. Use similar steps to construct a `BioRead` object from a SAM-formatted file.

Use the `BioRead` constructor function to construct a `BioRead` object from a FASTQ-formatted file and set the `Name` property:

```
BRObj1 = BioRead('SRR005164_1_50.fastq', 'Name', 'MyObject')
```

```
BRObj1 =
```

```
    BioRead with properties:
```

```
    Quality: [50x1 File indexed property]
    Sequence: [50x1 File indexed property]
    Header: [50x1 File indexed property]
    NSeqs: 50
    Name: 'MyObject'
```

The constructor function constructs a `BioRead` object and, if an index file does not already exist, it also creates an index file with the same file name, but with an `.IDX` extension. This index file, by default, is stored in the same location as the source file.

Caution Your source file and index file must always be in sync.

- After constructing a `BioRead` object, do not modify the index file, or you can get invalid results when using the existing object or constructing new objects.
 - If you modify the source file, delete the index file, so the object constructor creates a new index file when constructing new objects.
-

Note Because you constructed this `BioRead` object from a source file, you cannot modify the properties (except for `Name`) of the `BioRead` object.

Represent Sequence, Quality, and Alignment/Mapping Data in a `BioMap` Object

Prerequisites

A `BioMap` object represents a collection of sequence reads that map against a single reference sequence. Each element in the object is associated with a read sequence, sequence header, sequence quality information, and alignment/mapping information.

When constructing a `BioMap` object from a BAM file, the maximum size of the file is limited by your operating system and available memory.

Construct a `BioMap` object in one of two ways:

- **Indexed** — The data remains in the source file. Constructing the object and accessing its contents is memory efficient. However, you cannot modify object properties, other than the `Name` property. This is the default method if you construct a `BioMap` object from a SAM- or BAM-formatted file.
- **In Memory** — The data is read into memory. Constructing the object and accessing its contents is limited by the amount of available memory. However, you can modify object properties. When you construct a `BioMap` object from a structure, the data stays in memory. When you construct a `BioMap` object from a SAM- or BAM-formatted file, use the `InMemory` name-value pair argument to read the data into memory.

Construct a `BioMap` Object from a SAM- or BAM-Formatted File

Note This example constructs a `BioMap` object from a SAM-formatted file. Use similar steps to construct a `BioMap` object from a BAM-formatted file.

- 1 If you do not know the number and names of the reference sequences in your source file, determine them using the `saminfo` or `baminfo` function and the `ScanDictionary` name-value pair argument.

```
samstruct = saminfo('ex2.sam', 'ScanDictionary', true);
samstruct.ScannedDictionary

ans =

    'seq1'
    'seq2'
```

Tip The previous syntax scans the entire SAM file, which is time consuming. If you are confident that the Header information of the SAM file is correct, omit the `ScanDictionary` name-value pair argument, and inspect the `SequenceDictionary` field instead.

- 2 Use the `BioMap` constructor function to construct a `BioMap` object from the SAM file and set the `Name` property. Because the SAM-formatted file in this example, `ex2.sam`, contains multiple reference sequences, use the `SelectRef` name-value pair argument to specify one reference sequence, `seq1`:

```
BMObj2 = BioMap('ex2.sam', 'SelectRef', 'seq1', 'Name', 'MyObject')
```

```
BMObj2 =
```

```
BioMap with properties:
```

```
SequenceDictionary: 'seq1'
  Reference: [1501x1 File indexed property]
  Signature: [1501x1 File indexed property]
  Start: [1501x1 File indexed property]
MappingQuality: [1501x1 File indexed property]
  Flag: [1501x1 File indexed property]
  MatePosition: [1501x1 File indexed property]
  Quality: [1501x1 File indexed property]
  Sequence: [1501x1 File indexed property]
  Header: [1501x1 File indexed property]
  NSeqs: 1501
  Name: 'MyObject'
```

The constructor function constructs a `BioMap` object and, if index files do not already exist, it also creates one or two index files:

- If constructing from a SAM-formatted file, it creates one index file that has the same file name as the source file, but with an `.IDX` extension. This index file, by default, is stored in the same location as the source file.
- If constructing from a BAM-formatted file, it creates two index files that have the same file name as the source file, but one with a `.BAI` extension and one with a `.LINEARINDEX` extension. These index files, by default, are stored in the same location as the source file.

Caution Your source file and index files must always be in sync.

- After constructing a `BioMap` object, do not modify the index files, or you can get invalid results when using the existing object or constructing new objects.
 - If you modify the source file, delete the index files, so the object constructor creates new index files when constructing new objects.
-

Note Because you constructed this `BioMap` object from a source file, you cannot modify the properties (except for `Name` and `Reference`) of the `BioMap` object.

Construct a `BioMap` Object from a SAM or BAM Structure

Note This example constructs a `BioMap` object from a SAM structure using `samread`. Use similar steps to construct a `BioMap` object from a BAM structure using `bamread`.

- 1 Use the `samread` function to create a SAM structure from a SAM-formatted file:

```
SAMStruct = samread('ex2.sam');
```

- 2 To construct a valid `BioMap` object from a SAM-formatted file, the file must contain only one reference sequence. Determine the number and names of the reference sequences in your SAM-

formatted file using the `unique` function to find unique names in the `ReferenceName` field of the structure:

```
unique({SAMStruct.ReferenceName})
```

```
ans =
```

```
    'seq1'    'seq2'
```

- 3 Use the `BioMap` constructor function to construct a `BioMap` object from a SAM structure. Because the SAM structure contains multiple reference sequences, use the `SelectRef` name-value pair argument to specify one reference sequence, `seq1`:

```
BMObj1 = BioMap(SAMStruct, 'SelectRef', 'seq1')
```

```
BMObj1 =
```

BioMap with properties:

```
SequenceDictionary: {'seq1'}
Reference: {1501x1 cell}
Signature: {1501x1 cell}
Start: [1501x1 uint32]
MappingQuality: [1501x1 uint8]
Flag: [1501x1 uint16]
MatePosition: [1501x1 uint32]
Quality: {1501x1 cell}
Sequence: {1501x1 cell}
Header: {1501x1 cell}
NSeqs: 1501
Name: ''
```

Retrieve Information from a BioRead or BioMap Object

You can retrieve all or a subset of information from a `BioRead` or `BioMap` object.

Retrieve a Property from a BioRead or BioMap Object

You can retrieve a specific property from elements in a `BioRead` or `BioMap` object.

For example, to retrieve all headers from a `BioRead` object, use the `Header` property as follows:

```
allHeaders = BRObj1.Header;
```

This syntax returns a cell array containing the headers for all elements in the `BioRead` object.

Similarly, to retrieve all start positions of aligned read sequences from a `BioMap` object, use the `Start` property of the object:

```
allStarts = BMObj1.Start;
```

This syntax returns a vector containing the start positions of aligned read sequences with respect to the position numbers in the reference sequence in a `BioMap` object.

Retrieve Multiple Properties from a BioRead or BioMap Object

You can retrieve multiple properties from a `BioRead` or `BioMap` object in a single command using the `get` method. For example, to retrieve both start positions and headers information of a `BioMap` object, use the `get` method as follows:

```
multiProp = get(BMobj1, {'Start', 'Header'});
```

This syntax returns a cell array containing all start positions and headers information of a `BioMap` object.

Note Property names are case sensitive.

For a list and description of all properties of a `BioRead` object, see `BioRead` class. For a list and description of all properties of a `BioMap` object, see `BioMap` class.

Retrieve a Subset of Information from a BioRead or BioMap Object

Use specialized `get` methods with a numeric vector, logical vector, or cell array of headers to retrieve a subset of information from an object. For example, to retrieve the first 10 elements from a `BioRead` object, use the `getSubset` method:

```
newBRobj = getSubset(BRobj1, [1:10]);
```

This syntax returns a new `BioRead` object containing the first 10 elements in the original `BioRead` object.

For example, to retrieve the first 12 positions of sequences with headers `SRR005164.1`, `SRR005164.7`, and `SRR005164.16`, use the `getSubsequence` method:

```
subSeqs = getSubsequence(BRobj1, ...
    {'SRR005164.1', 'SRR005164.7', 'SRR005164.16'}, [1:12])
subSeqs =
    'TGGCTTTAAAGC'
    'CCCGAAAGCTAG'
    'AATTTTGCGGCT'
```

For example, to retrieve information about the third element in a `BioMap` object, use the `getInfo` method:

```
Info_3 = getInfo(BMobj1, 3);
```

This syntax returns a tab-delimited character vector containing this information for the third element:

- Sequence header
- SAM flags for the sequence
- Start position of the aligned read sequence with respect to the reference sequence
- Mapping quality score for the sequence
- Signature (CIGAR-formatted character vector) for the sequence
- Sequence

- Quality scores for sequence positions

Note Method names are case sensitive.

For a complete list and description of methods of a `BioRead` object, see `BioRead` class. For a complete list and description of methods of a `BioMap` object, see `BioMap` class.

Set Information in a `BioRead` or `BioMap` Object

Prerequisites

To modify properties (other than `Name` and `Reference`) of a `BioRead` or `BioMap` object, the data must be in memory, and not indexed. To ensure the data is in memory, do one of the following:

- Construct the object from a structure as described in “Construct a `BioMap` Object from a SAM or BAM Structure” on page 2-9.
- Construct the object from a source file using the `InMemory` name-value pair argument.

Provide Custom Headers for Sequences

First, create an object with the data in memory:

```
BRObj1 = BioRead('SRR005164_1_50.fastq','InMemory',true);
```

To provide custom headers for sequences of interest (in this case sequences 1 to 5), do the following:

```
BRObj1.Header(1:5) = {'H1', 'H2', 'H3', 'H4', 'H5'};
```

Alternatively, you can use the `setHeader` method:

```
BRObj1 = setHeader(BRObj1, {'H1', 'H2', 'H3', 'H4', 'H5'}, [1:5]);
```

Several other specialized `set` methods let you set the properties of a subset of elements in a `BioRead` or `BioMap` object.

Note Method names are case sensitive.

For a complete list and description of methods of a `BioRead` object, see `BioRead` class. For a complete list and description of methods of a `BioMap` object, see `BioMap` class.

Determine Coverage of a Reference Sequence

When working with a `BioMap` object, you can determine the number of read sequences that:

- Align within a specific region of the reference sequence
- Align to each position within a specific region of the reference sequence

For example, you can compute the number, indices, and start positions of the read sequences that align within the first 25 positions of the reference sequence. To do so, use the `getCounts`, `getIndex`, and `getStart` methods:

```
Cov = getCounts(BMObj1, 1, 25)
```

```

Cov =
    12
Indices = getIndex(BMObj1, 1, 25)
Indices =
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
   11
   12

startPos = getStart(BMObj1, Indices)
startPos =
    1
    3
    5
    6
    9
   13
   13
   15
   18
   22
   22
   24

```

The first two syntaxes return the number and indices of the read sequences that align within the specified region of the reference sequence. The last syntax returns a vector containing the start position of each aligned read sequence, corresponding to the position numbers of the reference sequence.

For example, you can also compute the number of the read sequences that align to *each* of the first 10 positions of the reference sequence. For this computation, use the `getBaseCoverage` method:

```

Cov = getBaseCoverage(BMObj1, 1, 10)
Cov =
    1    1    2    2    3    4    4    4    5    5

```

Construct Sequence Alignments to a Reference Sequence

It is useful to construct and view the alignment of the read sequences that align to a specific region of the reference sequence. It is also helpful to know which read sequences align to this region in a `BioMap` object.

For example, to retrieve the alignment of read sequences to the first 12 positions of the reference sequence in a `BioMap` object, use the `getAlignment` method:

```
[Alignment_1_12, Indices] = getAlignment(BMObj2, 1, 12)
```

```
Alignment_1_12 =
```

```
CACTAGTGGCTC
  CTAGTGGCTC
    AGTGGCTC
      GTGGCTC
        GCTC
```

```
Indices =
```

```
1
2
3
4
5
```

Return the headers of the read sequences that align to a specific region of the reference sequence:

```
alignedHeaders = getHeader(BMObj2, Indices)
```

```
alignedHeaders =
```

```
'B7_591:4:96:693:509'
'EAS54_65:7:152:368:113'
'EAS51_64:8:5:734:57'
'B7_591:1:289:587:906'
'EAS56_59:8:38:671:758'
```

Filter Read Sequences Using SAM Flags

SAM- and BAM-formatted files include the status of 11 binary flags for each read sequence. These flags describe different sequencing and alignment aspects of a read sequence. For more information on the flags, see the SAM Format Specification. The `filterByFlag` method lets you filter the read sequences in a `BioMap` object by using these flags.

Filter Unmapped Read Sequences

- 1 Construct a `BioMap` object from a SAM-formatted file.

```
BMObj2 = BioMap('ex1.sam');
```

- 2 Use the `filterByFlag` method to create a logical vector indicating the read sequences in a `BioMap` object that are mapped.

```
LogicalVec_mapped = filterByFlag(BMObj2, 'unmappedQuery', false);
```

- 3 Use this logical vector and the `getSubset` method to create a new `BioMap` object containing only the mapped read sequences.

```
filteredBMObj_1 = getSubset(BMObj2, LogicalVec_mapped);
```

Filter Read Sequences That Are Not Mapped in a Pair

- 1 Construct a `BioMap` object from a SAM-formatted file.

```
BMObj2 = BioMap('ex1.sam');
```

- 2 Use the `filterByFlag` method to create a logical vector indicating the read sequences in a `BioMap` object that are mapped in a proper pair, that is, both the read sequence and its mate are mapped to the reference sequence.

```
LogicalVec_paired = filterByFlag(BMObj2, 'pairedInMap', true);
```

- 3 Use this logical vector and the `getSubset` method to create a new `BioMap` object containing only the read sequences that are mapped in a proper pair.

```
filteredBMObj_2 = getSubset(BMObj2, LogicalVec_paired);
```

Store and Manage Feature Annotations in Objects

In this section...

“Represent Feature Annotations in a GFFAnnotation or GTFAnnotation Object” on page 2-16

“Construct an Annotation Object” on page 2-16

“Retrieve General Information from an Annotation Object” on page 2-16

“Access Data in an Annotation Object” on page 2-17

“Use Feature Annotations with Sequence Read Data” on page 2-18

Represent Feature Annotations in a GFFAnnotation or GTFAnnotation Object

The GFFAnnotation and GTFAnnotation objects represent a collection of feature annotations for one or more reference sequences. You construct these objects from GFF (General Feature Format) and GTF (Gene Transfer Format) files. Each element in the object represents a single annotation. The properties and methods associated with the objects let you investigate and filter the data based on reference sequence, a feature (such as CDS or exon), or a specific gene or transcript.

Construct an Annotation Object

Use the GFFAnnotation constructor function to construct a GFFAnnotation object from either a GFF- or GTF-formatted file:

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff')
```

```
GFFAnnotObj =
```

```
    GFFAnnotation with properties:
```

```
        FieldNames: {1x9 cell}
        NumEntries: 3331
```

Use the GTFAnnotation constructor function to construct a GTFAnnotation object from a GTF-formatted file:

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf')
```

```
GTFAnnotObj =
```

```
    GTFAnnotation with properties:
```

```
        FieldNames: {1x11 cell}
        NumEntries: 308
```

Retrieve General Information from an Annotation Object

Determine the field names and the number of entries in an annotation object by accessing the FieldNames and NumEntries properties. For example, to see the field names for each annotation object constructed in the previous section, query the FieldNames property:

```
GFFAnnotObj.FieldNames
```

```
ans =
    Columns 1 through 6
    'Reference'    'Start'    'Stop'    'Feature'    'Source'    'Score'
    Columns 7 through 9
    'Strand'    'Frame'    'Attributes'
```

```
GTFAnnotObj.FieldNameNames
```

```
ans =
    Columns 1 through 6
    'Reference'    'Start'    'Stop'    'Feature'    'Gene'    'Transcript'
    Columns 7 through 11
    'Source'    'Score'    'Strand'    'Frame'    'Attributes'
```

Determine the range of the reference sequences that are covered by feature annotations by using the `getRange` method with the annotation object constructed in the previous section:

```
range = getRange(GFFAnnotObj)
range =
    3631    498516
```

Access Data in an Annotation Object

Create a Structure of the Annotation Data

Creating a structure of the annotation data lets you access the field values. Use the `getData` method to create a structure containing a subset of the data in a `GFFAnnotation` object constructed in the previous section.

```
% Extract annotations for positions 1 through 10000 of the
% reference sequence
AnnotStruct = getData(GFFAnnotObj,1,10000)

AnnotStruct =
60x1 struct array with fields:
    Reference
    Start
    Stop
    Feature
    Source
    Score
    Strand
    Frame
    Attributes
```

Access Field Values in the Structure

Use dot indexing to access all or specific field values in a structure.

For example, extract the start positions for all annotations:

```
Starts = AnnotStruct.Start;
```

Extract the start positions for annotations 12 through 17. Notice that you must use square brackets when indexing a range of positions:

```
Starts_12_17 = [AnnotStruct(12:17).Start]
```

```
Starts_12_17 =
```

```
    4706    5174    5174    5439    5439    5631
```

Extract the start position and the feature for the 12th annotation:

```
Start_12 = AnnotStruct(12).Start
```

```
Start_12 =
```

```
    4706
```

```
Feature_12 = AnnotStruct(12).Feature
```

```
Feature_12 =
```

```
CDS
```

Use Feature Annotations with Sequence Read Data

Investigate the results of HTS sequencing experiments by using `GFFAnnotation` and `GTFAnnotation` objects with `BioMap` objects. For example, you can:

- Determine counts of sequence reads aligned to regions of a reference sequence associated with specific annotations, such as in RNA-Seq workflows.
- Find annotations within a specific range of a peak of interest in a reference sequence, such as in ChIP-Seq workflows.

Determine Annotations of Interest

- 1 Construct a `GTFAnnotation` object from a GTF-formatted file:

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

- 2 Use the `getReferenceNames` method to return the names for the reference sequences for the annotation object:

```
refNames = getReferenceNames(GTFAnnotObj)
```

```
refNames =
```

```
    'chr2'
```

- 3 Use the `getFeatureNames` method to retrieve the feature names from the annotation object:

```
featureNames = getFeatureNames(GTFAnnotObj)
```

```
featureNames =
  'CDS'
  'exon'
  'start_codon'
  'stop_codon'
```

- 4 Use the `getGeneNames` method to retrieve a list of the unique gene names from the annotation object:

```
geneNames = getGeneNames(GTFAnnotObj)
```

```
geneNames =
  'uc002qvu.2'
  'uc002qvv.2'
  'uc002qvw.2'
  'uc002qvx.2'
  'uc002qvy.2'
  'uc002qvz.2'
  'uc002qwa.2'
  'uc002qwb.2'
  'uc002qwc.1'
  'uc002qwd.2'
  'uc002qwe.3'
  'uc002qwf.2'
  'uc002qwg.2'
  'uc002qwh.2'
  'uc002qwi.3'
  'uc002qwk.2'
  'uc002qwl.2'
  'uc002qwm.1'
  'uc002qwn.1'
  'uc002qwo.1'
  'uc002qwp.2'
  'uc002qwq.2'
  'uc010ewe.2'
  'uc010ewf.1'
  'uc010ewg.2'
  'uc010ewh.1'
  'uc010ewi.2'
  'uc010yim.1'
```

The previous steps gave us a list of available reference sequences, features, and genes associated with the available annotations. Use this information to determine annotations of interest. For instance, you might be interested only in annotations that are exons associated with the `uc002qvv.2` gene on chromosome 2.

Filter Annotations

Use the `getData` method to filter the annotations and create a structure containing only the annotations of interest, which are annotations that are exons associated with the `uc002qvv.2` gene on chromosome 2.

```
AnnotStruct = getData(GTFAnnotObj, 'Reference', 'chr2', ...
  'Feature', 'exon', 'Gene', 'uc002qvv.2')
```

```
AnnotStruct =
```

12x1 struct array with fields:

```
Reference
Start
Stop
Feature
Gene
Transcript
Source
Score
Strand
Frame
Attributes
```

The return structure contains 12 elements, indicating there are 12 annotations that meet your filter criteria.

Extract Position Ranges for Annotations of Interest

After filtering the data to include only annotations that are exons associated with the uc002qvv.2 gene on chromosome 2, use the Start and Stop fields to create vectors of the start and end positions for the ranges associated with the 12 annotations.

```
StartPos = [AnnotStruct.Start];
EndPos = [AnnotStruct.Stop];
```

Determine Counts of Sequence Reads Aligned to Annotations

Construct a BioMap object from a BAM-formatted file containing sequence read data aligned to chromosome 2.

```
BMObj3 = BioMap('ex3.bam');
```

Then use the range for the annotations of interest as input to the `getCounts` method of a BioMap object. This returns the counts of short reads aligned to the annotations of interest.

```
counts = getCounts(BMObj3,StartPos,EndPos,'independent', true)
```

```
counts =
```

```
1399
     1
     54
    221
     97
    125
     0
     1
     0
     65
     9
    12
```

Bioinformatics Toolbox Software Support Packages

Bioinformatics Toolbox provides support packages for various next-generation sequencing workflows and analyses. To make a support package available in your MATLAB command line, you must first install it.

Install Support Package

Follow these steps to install a support package.

- 1 In the **Environment** section of the MATLAB toolstrip, select **Add-Ons > Get Add-Ons**.
- 2 In the Add-On Explorer, search for the support package that you want to install by entering its name.
- 3 Install the support package.

For details about installing add-ons, see “Get and Manage Add-Ons”. For other information, see “Add-Ons”.

Available Support Packages

The following table lists all the Bioinformatics Toolbox support packages that are available for download as Add-Ons.

Support Package Name	Version [†]	Corresponding MATLAB functions	Supported OS
Bowtie 2 Support Package for Bioinformatics Toolbox [1] (download link)	2.5.1	bowtie2, bowtie2build, bowtie2inspect.	Windows ^{®‡} , Mac, and Linux [®]
Cufflinks Support Package for the Bioinformatics Toolbox [2] (download link)	2.2.1	cufflinks, cuffcompare, cuffdiff, cuffgffread, cuffgtf2sam, cuffmerge, cuffnorm, cuffquant.	Windows [‡] , Mac, and Linux
BWA Support Package for Bioinformatics Toolbox [3][4] (download link)	0.7.17	bwaindex, bwamem.	Windows [‡] , Mac, and Linux
SRA Toolkit for Bioinformatics Toolbox [5] (download link)	3.0.6	srafasterdump, srasamdump	Windows, Mac, and Linux
BLAST+ Support Package for Bioinformatics Toolbox [6][7] (download link)	2.14.0	blastplusdatabase, blastplus, blastplustoptions	Windows, Mac, and Linux

[†]Version of the original (third-party) software

[‡] You need to install Windows Subsystem for Linux (WSL) and a Linux distribution on your Windows machine. For details on installing WSL, see here.

See Also

More About

- “Count Features from NGS Reads” on page 2-23
- “High-Throughput Sequencing”

References

- [1] Langmead, Ben, and Steven L Salzberg. “Fast Gapped-Read Alignment with Bowtie 2.” *Nature Methods* 9, no. 4 (April 2012): 357–59. <https://doi.org/10.1038/nmeth.1923>.
- [2] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.
- [3] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. <https://doi.org/10.1093/bioinformatics/btp698>.
- [4] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. <https://doi.org/10.1093/bioinformatics/btp324>.
- [5] SRA Toolkit Development Team <https://github.com/ncbi/sra-tools/wiki/01.-Downloading-SRA-Toolkit>
- [6] Camacho, Christiam, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. “BLAST+: Architecture and Applications.” *BMC Bioinformatics* 10, no. 1 (December 2009): 421.
- [7] “BLAST: Basic Local Alignment Search Tool.” <https://blast.ncbi.nlm.nih.gov/Blast.cgi>.

Count Features from NGS Reads

This example shows how to count features from paired-end sequencing reads after aligning them to the whole human genome curated by the Genome Reference Consortium. This example uses Genome Reference Consortium Human Build 38 patch release 12 (GRCh38.p12) as the human genome reference.

Prerequisites and Data Set

This example works on the UNIX® and Mac platforms only. Download the Bioinformatics Toolbox™ Interface for Bowtie Aligner support package from the Add-On Explorer. For details, see “Bioinformatics Toolbox Software Support Packages” on page 2-21.

This example assumes you have:

- Downloaded and extracted the RefSeq assembly from Genome Reference Consortium Human Build 38 patch release 12 (GRCh38.p12).
- Downloaded and organized some paired-end reads data. This example uses the exome sequencing data from the 1000 genomes project. Paired-end reads are indicated by '_1' and '_2' in the filenames.

Build Index

Construct an index for aligning reads to the reference using `bowtie2build`. The file `GCF_000001405.38_GRCh38.p12_genomic.fna` contains the human reference genome in the FASTA format. `bowtieIdx` is the base name of the reference index files. The `'--threads 8'` option specifies the number of parallel threads to build index files faster. You do not need to specify full file paths for `*.fna` or `*.index` files if you are running the example from the same folder location. Specify the full paths if you wish to store the files elsewhere or run the example from a different folder.

```
bowtieIdx = 'GCF_000001405.38_GRCh38.p12_genomic.index';
buildFlag = bowtie2build('GCF_000001405.38_GRCh38.p12_genomic.fna',...
                        bowtieIdx,'--threads 8');
```

Align Reads to Reference

Align paired-end reads to the reference using `bowtie2`. You can create a `Bowtie2AlignOptions` object to specify different options, such as the number of parallel threads to use.

```
opt          = Bowtie2AlignOptions;
opt.NumThreads = 8;
reads1       = 'HG00096_1.fastq';
reads2       = 'HG00096_2.fastq';
bowtie2(bowtieIdx, reads1, reads2, 'HG00096.sam', opt);
```

Selectively Align to Gene of Interest

SAM files can be very large. Use `BioMap` to select only the data for the correct reference. For this example, consider `APOE`, which is a gene on Chromosome 19 linked to Alzheimer's disease. Create a smaller BAM file for `APOE` to improve performance.

```
apoeRef = 'NC_000019.10'; % Reference name for Chromosome 19 in HG38
bm      = BioMap('HG00096.sam', 'SelectReference', apoeRef);
write(bm, 'HG00096.bam', 'Format', 'bam');
```

```
Warning: Found invalid tag in header type: 'PG'. Ignoring tag 'PN:bowtie2'.
Warning: The read sequences in input SAM file do not appear to be ordered
according to the start position of their alignments with the reference
sequence. Because of this, there will be a decrease in performance when
accessing the reads. For maximum performance, order the read sequences in the
SAM file, before creating a BioMap object.
```

Summarize Read Counts

Use `featurecount` to compare the number of transcripts for each APOE variant using a GTF file. A full table of features is included in the GRCh38.p12 assembly in GFF format, which can be converted to GTF using `cuffgffread`. This example uses a simplified GTF based on APOE transcripts. `APOE_gene.gtf` is included with the software.

```
[FeatTable, Summary] = featurecount('APOE_gene.gtf', 'HG00096.bam', ...
                                   'Metafeature', 'transcript_id');
```

```
Processing GTF file APOE_gene.gtf ...
Processing BAM file HG00096.bam ...
Processing reference NC_000019.10 ...
10000 reads processed ...
20000 reads processed ...
30000 reads processed ...
40000 reads processed ...
50000 reads processed ...
60000 reads processed ...
70000 reads processed ...
80000 reads processed ...
90000 reads processed ...
100000 reads processed ...
110000 reads processed ...
120000 reads processed ...
130000 reads processed ...
140000 reads processed ...
150000 reads processed ...
160000 reads processed ...
170000 reads processed ...
180000 reads processed ...
190000 reads processed ...
200000 reads processed ...
210000 reads processed ...
220000 reads processed ...
230000 reads processed ...
240000 reads processed ...
250000 reads processed ...
260000 reads processed ...
270000 reads processed ...
280000 reads processed ...
290000 reads processed ...
300000 reads processed ...
310000 reads processed ...
320000 reads processed ...
330000 reads processed ...
340000 reads processed ...
350000 reads processed ...
360000 reads processed ...
370000 reads processed ...
380000 reads processed ...
```

```
390000 reads processed ...
400000 reads processed ...
410000 reads processed ...
420000 reads processed ...
430000 reads processed ...
440000 reads processed ...
450000 reads processed ...
460000 reads processed ...
470000 reads processed ...
480000 reads processed ...
490000 reads processed ...
500000 reads processed ...
510000 reads processed ...
520000 reads processed ...
530000 reads processed ...
540000 reads processed ...
550000 reads processed ...
560000 reads processed ...
570000 reads processed ...
580000 reads processed ...
590000 reads processed ...
600000 reads processed ...
610000 reads processed ...
620000 reads processed ...
630000 reads processed ...
640000 reads processed ...
650000 reads processed ...
660000 reads processed ...
670000 reads processed ...
680000 reads processed ...
690000 reads processed ...
700000 reads processed ...
710000 reads processed ...
720000 reads processed ...
730000 reads processed ...
740000 reads processed ...
750000 reads processed ...
760000 reads processed ...
770000 reads processed ...
780000 reads processed ...
790000 reads processed ...
800000 reads processed ...
810000 reads processed ...
820000 reads processed ...
830000 reads processed ...
840000 reads processed ...
850000 reads processed ...
860000 reads processed ...
870000 reads processed ...
880000 reads processed ...
890000 reads processed ...
900000 reads processed ...
910000 reads processed ...
920000 reads processed ...
930000 reads processed ...
940000 reads processed ...
950000 reads processed ...
960000 reads processed ...
```

970000 reads processed ...
980000 reads processed ...
990000 reads processed ...
1000000 reads processed ...
1010000 reads processed ...
1020000 reads processed ...
1030000 reads processed ...
1040000 reads processed ...
1050000 reads processed ...
1060000 reads processed ...
1070000 reads processed ...
1080000 reads processed ...
1090000 reads processed ...
1100000 reads processed ...
1110000 reads processed ...
1120000 reads processed ...
1130000 reads processed ...
1140000 reads processed ...
1150000 reads processed ...
1160000 reads processed ...
1170000 reads processed ...
1180000 reads processed ...
1190000 reads processed ...
1200000 reads processed ...
1210000 reads processed ...
1220000 reads processed ...
1230000 reads processed ...
1240000 reads processed ...
1250000 reads processed ...
1260000 reads processed ...
1270000 reads processed ...
1280000 reads processed ...
1290000 reads processed ...
1300000 reads processed ...
1310000 reads processed ...
1320000 reads processed ...
1330000 reads processed ...
1340000 reads processed ...
1350000 reads processed ...
1360000 reads processed ...
1370000 reads processed ...
1380000 reads processed ...
1390000 reads processed ...
1400000 reads processed ...
1410000 reads processed ...
1420000 reads processed ...
1430000 reads processed ...
1440000 reads processed ...
1450000 reads processed ...
1460000 reads processed ...
1470000 reads processed ...
1480000 reads processed ...
1490000 reads processed ...
1500000 reads processed ...
1510000 reads processed ...
1520000 reads processed ...
1530000 reads processed ...
1540000 reads processed ...

```
1550000 reads processed ...
1560000 reads processed ...
1570000 reads processed ...
1580000 reads processed ...
1590000 reads processed ...
1600000 reads processed ...
1610000 reads processed ...
1620000 reads processed ...
1630000 reads processed ...
1640000 reads processed ...
1650000 reads processed ...
1660000 reads processed ...
1670000 reads processed ...
1680000 reads processed ...
1690000 reads processed ...
1700000 reads processed ...
1710000 reads processed ...
1720000 reads processed ...
1730000 reads processed ...
1740000 reads processed ...
1750000 reads processed ...
1760000 reads processed ...
1770000 reads processed ...
1780000 reads processed ...
1790000 reads processed ...
1800000 reads processed ...
1810000 reads processed ...
1820000 reads processed ...
1830000 reads processed ...
1840000 reads processed ...
1850000 reads processed ...
1860000 reads processed ...
1870000 reads processed ...
1880000 reads processed ...
1890000 reads processed ...
1900000 reads processed ...
1910000 reads processed ...
1920000 reads processed ...
1930000 reads processed ...
1940000 reads processed ...
1950000 reads processed ...
1960000 reads processed ...
1970000 reads processed ...
1980000 reads processed ...
1990000 reads processed ...
2000000 reads processed ...
2010000 reads processed ...
2020000 reads processed ...
2030000 reads processed ...
2040000 reads processed ...
2050000 reads processed ...
2060000 reads processed ...
2070000 reads processed ...
2080000 reads processed ...
2090000 reads processed ...
2100000 reads processed ...
2110000 reads processed ...
2120000 reads processed ...
```

```
2130000 reads processed ...
2140000 reads processed ...
2150000 reads processed ...
2160000 reads processed ...
2170000 reads processed ...
2180000 reads processed ...
2190000 reads processed ...
2200000 reads processed ...
2210000 reads processed ...
2220000 reads processed ...
2230000 reads processed ...
2240000 reads processed ...
2250000 reads processed ...
2260000 reads processed ...
2270000 reads processed ...
2280000 reads processed ...
2290000 reads processed ...
2300000 reads processed ...
2310000 reads processed ...
2320000 reads processed ...
2330000 reads processed ...
2340000 reads processed ...
2350000 reads processed ...
2360000 reads processed ...
2370000 reads processed ...
2380000 reads processed ...
2390000 reads processed ...
2400000 reads processed ...
2410000 reads processed ...
2420000 reads processed ...
2430000 reads processed ...
2440000 reads processed ...
2450000 reads processed ...
2460000 reads processed ...
2470000 reads processed ...
2480000 reads processed ...
2490000 reads processed ...
2500000 reads processed ...
2510000 reads processed ...
2520000 reads processed ...
2530000 reads processed ...
2540000 reads processed ...
2550000 reads processed ...
2560000 reads processed ...
2570000 reads processed ...
2580000 reads processed ...
2590000 reads processed ...
2600000 reads processed ...
2610000 reads processed ...
2620000 reads processed ...
2630000 reads processed ...
2640000 reads processed ...
2650000 reads processed ...
2660000 reads processed ...
2670000 reads processed ...
2680000 reads processed ...
2690000 reads processed ...
2700000 reads processed ...
```

```
2710000 reads processed ...
2720000 reads processed ...
2730000 reads processed ...
2740000 reads processed ...
2750000 reads processed ...
2760000 reads processed ...
2770000 reads processed ...
2780000 reads processed ...
2790000 reads processed ...
2800000 reads processed ...
2810000 reads processed ...
2820000 reads processed ...
2830000 reads processed ...
2840000 reads processed ...
2850000 reads processed ...
2860000 reads processed ...
2870000 reads processed ...
2880000 reads processed ...
2890000 reads processed ...
2900000 reads processed ...
2910000 reads processed ...
2920000 reads processed ...
2930000 reads processed ...
2940000 reads processed ...
2950000 reads processed ...
2960000 reads processed ...
2970000 reads processed ...
2980000 reads processed ...
2990000 reads processed ...
3000000 reads processed ...
3010000 reads processed ...
3020000 reads processed ...
3030000 reads processed ...
3040000 reads processed ...
3050000 reads processed ...
3060000 reads processed ...
3070000 reads processed ...
3080000 reads processed ...
3090000 reads processed ...
3100000 reads processed ...
3110000 reads processed ...
3120000 reads processed ...
3130000 reads processed ...
3140000 reads processed ...
3150000 reads processed ...
3160000 reads processed ...
3170000 reads processed ...
3180000 reads processed ...
3190000 reads processed ...
3200000 reads processed ...
3210000 reads processed ...
3220000 reads processed ...
3230000 reads processed ...
3240000 reads processed ...
3250000 reads processed ...
3260000 reads processed ...
3270000 reads processed ...
3280000 reads processed ...
```

3290000 reads processed ...
3300000 reads processed ...
3310000 reads processed ...
3320000 reads processed ...
3330000 reads processed ...
3340000 reads processed ...
3350000 reads processed ...
3360000 reads processed ...
3370000 reads processed ...
3380000 reads processed ...
3390000 reads processed ...
3400000 reads processed ...
3410000 reads processed ...
3420000 reads processed ...
3430000 reads processed ...
3440000 reads processed ...
3450000 reads processed ...
3460000 reads processed ...
3470000 reads processed ...
3480000 reads processed ...
3490000 reads processed ...
3500000 reads processed ...
3510000 reads processed ...
3520000 reads processed ...
3530000 reads processed ...
3540000 reads processed ...
3550000 reads processed ...
3560000 reads processed ...
3570000 reads processed ...
3580000 reads processed ...
3590000 reads processed ...
3600000 reads processed ...
3610000 reads processed ...
3620000 reads processed ...
3630000 reads processed ...
3640000 reads processed ...
3650000 reads processed ...
3660000 reads processed ...
3670000 reads processed ...
3680000 reads processed ...
3690000 reads processed ...
3700000 reads processed ...
3710000 reads processed ...
3720000 reads processed ...
3730000 reads processed ...
3740000 reads processed ...
3750000 reads processed ...
3760000 reads processed ...
3770000 reads processed ...
3780000 reads processed ...
3790000 reads processed ...
3800000 reads processed ...
3810000 reads processed ...
3820000 reads processed ...
3830000 reads processed ...
3840000 reads processed ...
3850000 reads processed ...
3860000 reads processed ...

```
3870000 reads processed ...
3880000 reads processed ...
3890000 reads processed ...
3900000 reads processed ...
3910000 reads processed ...
3920000 reads processed ...
3930000 reads processed ...
3940000 reads processed ...
3950000 reads processed ...
3960000 reads processed ...
3970000 reads processed ...
Done.
```

See Also

[bamsort](#) | [samsort](#) | [bwamem](#) | [bowtie2](#) | [bowtie2build](#) | [featurecount](#) | [BioMap](#) | [cuffgffread](#) | [cufflinks](#)

Identifying Differentially Expressed Genes from RNA-Seq Data

This example shows how to test RNA-Seq data for differentially expressed genes using a negative binomial model.

Introduction

A typical differential expression analysis of RNA-Seq data consists of normalizing the raw counts and performing statistical tests to reject or accept the null hypothesis that two groups of samples show no significant difference in gene expression. This example shows how to inspect the basic statistics of raw count data, how to determine size factors for count normalization and how to infer the most differentially expressed genes using a negative binomial model.

The dataset for this example comprises of RNA-Seq data obtained in the experiment described by Brooks et al. [1]. The authors investigated the effect of siRNA knock-down of *pasilla*, a gene known to play an important role in the regulation of splicing in *Drosophila melanogaster*. The dataset consists of 2 biological replicates of the control (untreated) samples and 2 biological replicates of the knock-down (treated) samples.

Inspecting Read Count Tables for Genomic Features

The starting point for this analysis of RNA-Seq data is a count matrix, where the rows correspond to genomic features of interest, the columns correspond to the given samples and the values represent the number of reads mapped to each feature in a given sample.

The included file `pasilla_count_noMM.mat` contains two tables with the count matrices at the gene level and at the exon level for each of the considered samples. You can obtain similar matrices using the function `featurecount`.

```
load pasilla_count_noMM.mat
```

```
% preview the table of read counts for genes
head(geneCountTable)
```

ID	Reference	untreated3	untreated4	treated2	treated3
"FBgn0000003"	"3R"	0	1	1	2
"FBgn0000008"	"2R"	142	117	138	132
"FBgn0000014"	"3R"	20	12	10	19
"FBgn0000015"	"3R"	2	4	0	1
"FBgn0000017"	"3L"	6591	5127	4809	6027
"FBgn0000018"	"2L"	469	530	492	574
"FBgn0000024"	"3R"	5	6	10	8
"FBgn0000028"	"X"	0	0	2	1

Note that when counting is performed without summarization, the individual features (exons in this case) are reported with their metafeature assignment (genes in this case) followed by the start and stop positions.

```
% preview the table of read counts for exons
head(exonCountTable)
```

ID	Reference	untreated3	untreated4	treated2	treated3
"FBgn0000003_2648220_2648518"	"3R"	0	0	0	0
"FBgn0000008_18024938_18025756"	"2R"	0	1	0	0
"FBgn0000008_18050410_18051199"	"2R"	13	9	14	11
"FBgn0000008_18052282_18052494"	"2R"	4	2	5	0
"FBgn0000008_18056749_18058222"	"2R"	32	27	26	23
"FBgn0000008_18058283_18059490"	"2R"	14	18	29	22
"FBgn0000008_18059587_18059757"	"2R"	1	4	3	0
"FBgn0000008_18059821_18059938"	"2R"	0	0	2	0

You can annotate and group the samples by creating a logical vector as follows:

```
samples = geneCountTable(:,3:end).Properties.VariableNames;
untreated = strncmp(samples,'untreated',length('untreated'));
treated = strncmp(samples,'treated',length('treated'))
```

```
untreated =
```

```
1×4 logical array
```

```
1 1 0 0
```

```
treated =
```

```
1×4 logical array
```

```
0 0 1 1
```

Plotting the Feature Assignments

The included file also contains a table `geneSummaryTable` with the summary of assigned and unassigned SAM entries. You can plot the basic distribution of the counting results by considering the number of reads that are assigned to the given genomic features (exons or genes for this example), as well as the number of reads that are unassigned (i.e. not overlapping any feature) or ambiguous (i.e. overlapping multiple features).

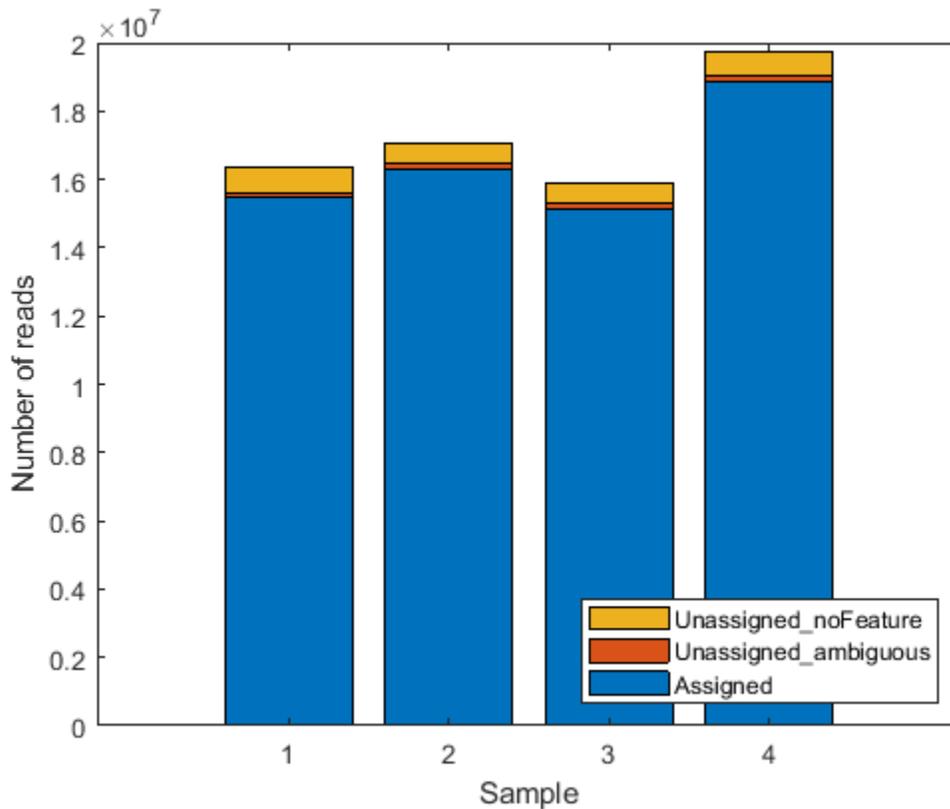
```
st = geneSummaryTable({'Assigned','Unassigned_ambiguous','Unassigned_noFeature'},:)
bar(table2array(st),'stacked');
legend(st.Properties.RowNames,'Interpreter','none','Location','southeast');
xlabel('Sample')
ylabel('Number of reads')
```

```
st =
```

```
3×4 table
```

	untreated3	untreated4	treated2	treated3
Assigned	1.5457e+07	1.6302e+07	1.5146e+07	1.8856e+07
Unassigned_ambiguous	1.5708e+05	1.6882e+05	1.6194e+05	1.9977e+05

Unassigned_noFeature 7.5455e+05 5.8309e+05 5.8756e+05 6.8356e+05



Note that a small fraction of the alignment records in the SAM files is not reported in the summary table. You can notice this in the difference between the total number of records in a SAM file and the total number of records processed during the counting procedure for that same SAM file. These unreported records correspond to the records mapped to reference sequences that are not annotated in the GTF file and therefore are not processed in the counting procedure. If the gene models account for all the reference sequences used during the read mapping step, then all records are reported in one of the categories of the summary table.

```
geneSummaryTable{'TotalEntries', :} - sum(geneSummaryTable{2:end, :})
```

```
ans =
```

```
89516      95885      98207      104629
```

Plotting Read Coverage Across a Given Chromosome

When read counting is performed without summarization using the function `featurecount`, the default IDs are composed by the attribute or metafeature (by default, `gene_id`) followed by the start and the stop positions of the feature (by default, `exon`). You can use the exon start positions to plot the read coverage across any chromosome in consideration, for example chromosome arm 2L.

```
% consider chromosome arm 2L
```

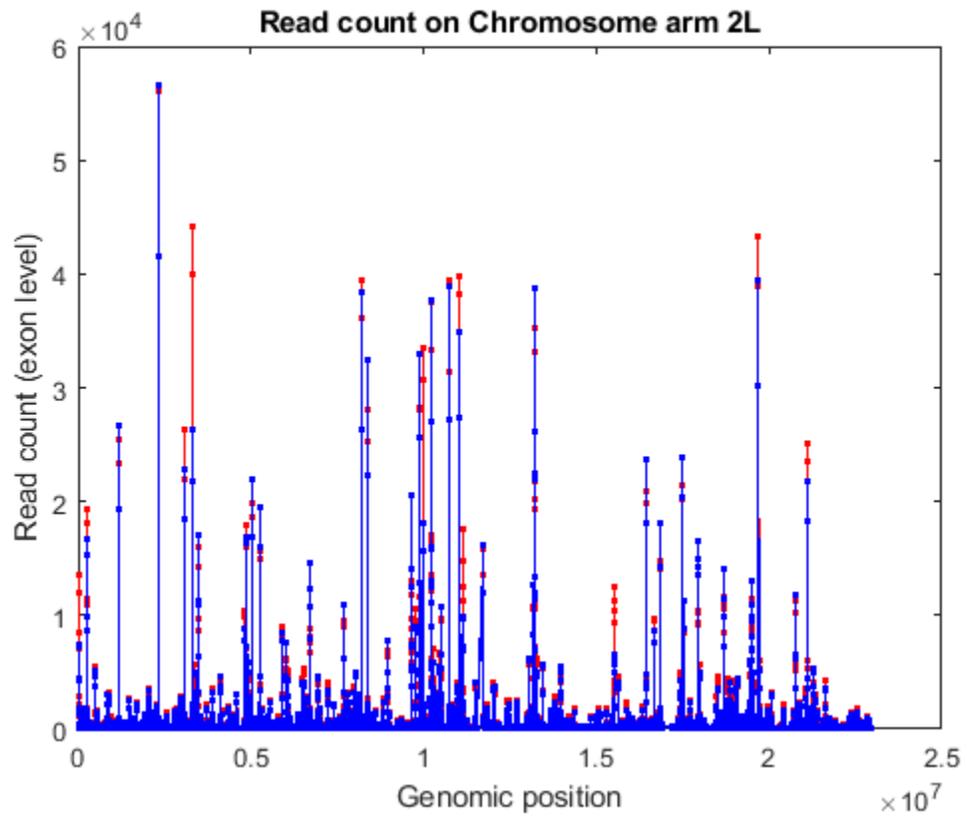
```
chr2L = strcmp(exonCountTable.Reference, '2L');
exonCount = exonCountTable{:,3:end};

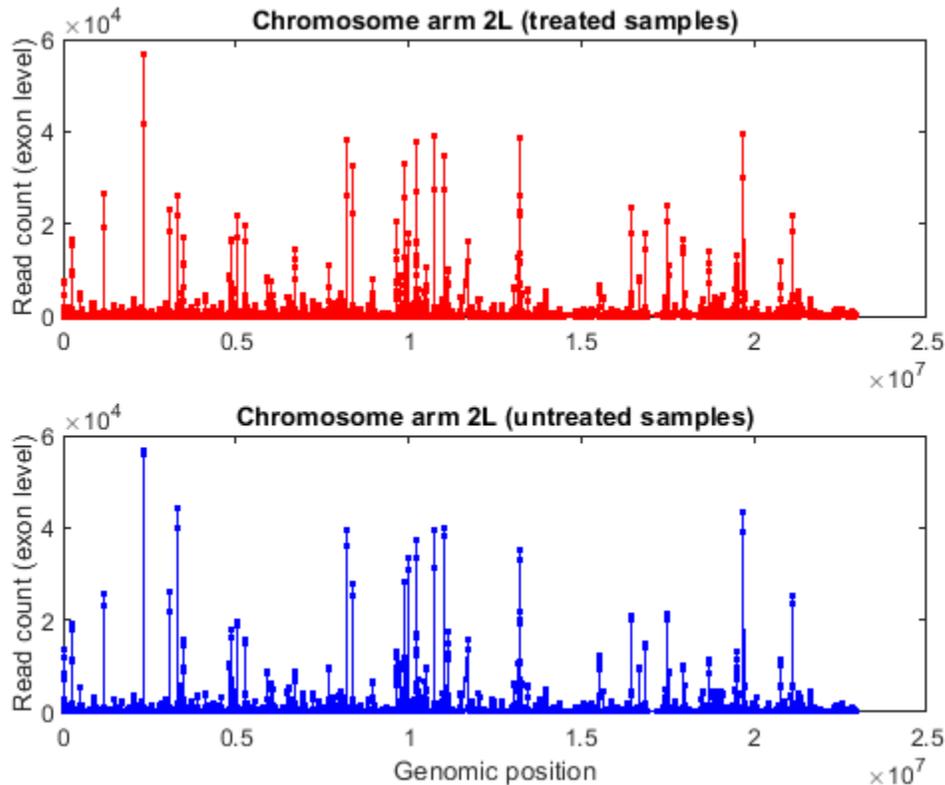
% retrieve exon start positions
exonStart = regexp(exonCountTable{chr2L,1}, '_(\d+)_', 'tokens');
exonStart = [exonStart{:}];
exonStart = cellfun(@str2num, [exonStart{:}]);

% sort exon by start positions
[~,idx] = sort(exonStart);

% plot read coverage along the genomic coordinates
figure;
plot(exonStart(idx),exonCount(idx,treated),'.-r',...
exonStart(idx),exonCount(idx,untreated),'.-b');
xlabel('Genomic position');
ylabel('Read count (exon level)');
title('Read count on Chromosome arm 2L');

% plot read coverage for each group separately
figure;
subplot(2,1,1);
plot(exonStart(idx),exonCount(idx,untreated),'.-r');
ylabel('Read count (exon level)');
title('Chromosome arm 2L (treated samples)');
subplot(2,1,2);
plot(exonStart(idx),exonCount(idx,treated),'.-b');
ylabel('Read count (exon level)');
xlabel('Genomic position');
title('Chromosome arm 2L (untreated samples)');
```





Alternatively, you can plot the read coverage considering the starting position of each gene in a given chromosome. The file `pasilla_geneLength.mat` contains a table with the start and stop position of each gene in the corresponding gene annotation file.

```
% load gene start and stop position information
load pasilla_geneLength
head(geneLength)
```

ID	Name	Reference	Start	Stop
"FBgn0037213"	"CG12581"	3R	380	10200
"FBgn0000500"	"Dsk"	3R	15388	16170
"FBgn0053294"	"CR33294"	3R	17136	21871
"FBgn0037215"	"CG12582"	3R	23029	30295
"FBgn0037217"	"CG14636"	3R	30207	41033
"FBgn0037218"	"aux"	3R	37505	53244
"FBgn0051516"	"CG31516"	3R	44179	45852
"FBgn0261436"	"DhpD"	3R	53106	54971

```
% consider chromosome 3 ('Reference' is a categorical variable)
chr3 = (geneLength.Reference == '3L') | (geneLength.Reference == '3R');
sum(chr3)
```

```
% consider the counts for genes in chromosome 3
```

```

counts = geneCountTable(:,3:end);
[~,j,k] = intersect(geneCountTable(:, 'ID'),geneLength{chr3, 'ID'});
gstart = geneLength{k, 'Start'};
gcounts = counts(j,:);

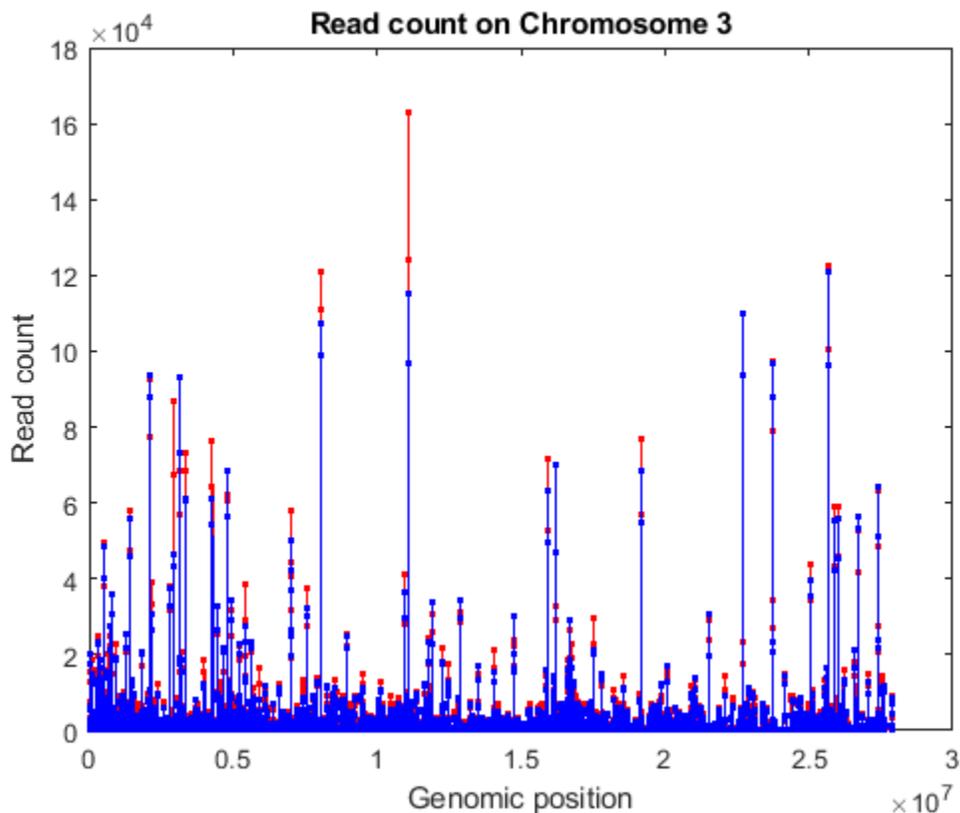
% sort according to ascending start position
[~,idx] = sort(gstart);

% plot read coverage by genomic position
figure;
plot(gstart(idx), gcounts(idx,treated), '-r',...
     gstart(idx), gcounts(idx,untreated), '-b');
xlabel('Genomic position')
ylabel('Read count');
title('Read count on Chromosome 3');

```

```
ans =
```

```
6360
```



Normalizing Read Counts

The read count in RNA-Seq data has been found to be linearly related to the abundance of transcripts [2]. However, the read count for a given gene depends not only on the expression level of the gene, but also on the total number of reads sequenced and the length of the gene transcript. Therefore, in

order to infer the expression level of a gene from the read count, we need to account for the sequencing depth and the gene transcript length. One common technique to normalize the read count is to use the RPKM (Read Per Kilobase Mapped) values, where the read count is normalized by the total number of reads yielded (in millions) and the length of each transcript (in kilobases). This normalization technique, however, is not always effective since few, very highly expressed genes can dominate the total lane count and skew the expression analysis.

A better normalization technique consists of computing the effective library size by considering a size factor for each sample. By dividing each sample's counts by the corresponding size factors, we bring all the count values to a common scale, making them comparable. Intuitively, if sample A is sequenced N times deeper than sample B, the read counts of non-differentially expressed genes are expected to be on average N times higher in sample A than in sample B, even if there is no difference in expression.

To estimate the size factors, take the median of the ratios of observed counts to those of a pseudo-reference sample, whose counts can be obtained by considering the geometric mean of each gene across all samples [3]. Then, to transform the observed counts to a common scale, divide the observed counts in each sample by the corresponding size factor.

```
% estimate pseudo-reference with geometric mean row by row
pseudoRefSample = geomean(counts,2);
nz = pseudoRefSample > 0;
ratios = bsxfun(@rdivide,counts(nz,:),pseudoRefSample(nz));
sizeFactors = median(ratios,1)
```

```
sizeFactors =
    0.9374    0.9725    0.9388    1.1789
```

```
% transform to common scale
normCounts = bsxfun(@rdivide,counts,sizeFactors);
normCounts(1:10,:)
```

```
ans =
    1.0e+03 *
         0    0.0010    0.0011    0.0017
    0.1515    0.1203    0.1470    0.1120
    0.0213    0.0123    0.0107    0.0161
    0.0021    0.0041         0    0.0008
    7.0315    5.2721    5.1225    5.1124
    0.5003    0.5450    0.5241    0.4869
    0.0053    0.0062    0.0107    0.0068
         0         0    0.0021    0.0008
    1.2375    1.1753    1.2122    1.2003
         0         0         0    0.0008
```

You can appreciate the effect of this normalization by using the function `boxplot` to represent statistical measures such as median, quartiles, minimum and maximum.

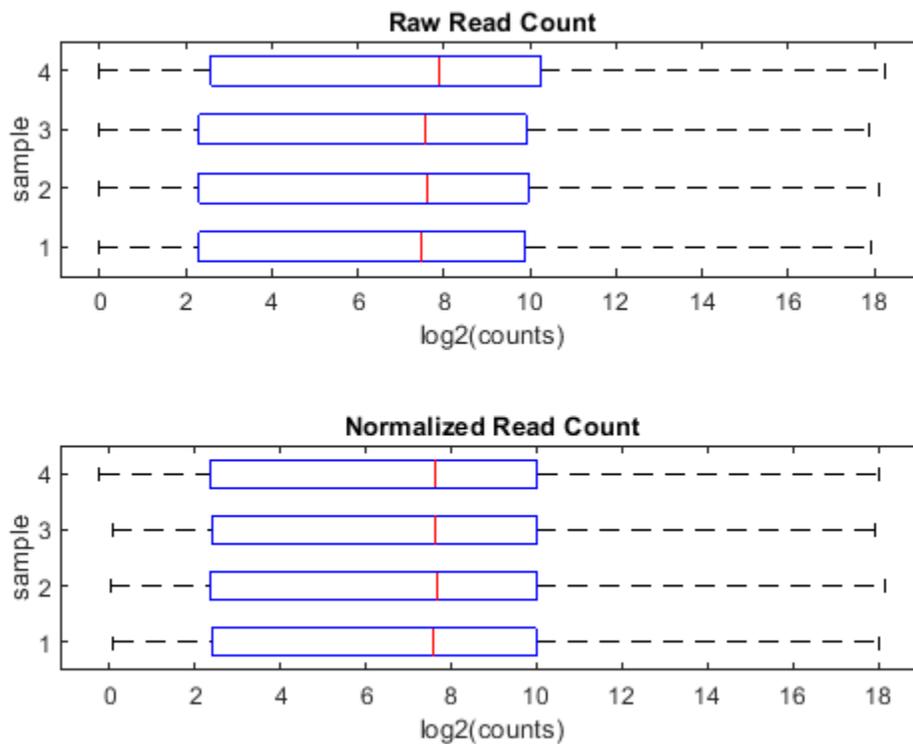
```

figure;

subplot(2,1,1)
maboxplot(log2(counts), 'title', 'Raw Read Count', 'orientation', 'horizontal')
ylabel('sample')
xlabel('log2(counts)')

subplot(2,1,2)
maboxplot(log2(normCounts), 'title', 'Normalized Read Count', 'orientation', 'horizontal')
ylabel('sample')
xlabel('log2(counts)')

```



Computing Mean, Dispersion and Fold Change

In order to better characterize the data, we consider the mean and the dispersion of the normalized counts. The variance of read counts is given by the sum of two terms: the variation across samples (raw variance) and the uncertainty of measuring the expression by counting reads (shot noise or Poisson). The raw variance term dominates for highly expressed genes, whereas the shot noise dominates for lowly expressed genes. You can plot the empirical dispersion values against the mean of the normalized counts in a log scale as shown below.

```

% consider the mean
meanTreated = mean(normCounts(:,treated),2);
meanUntreated = mean(normCounts(:,untreated),2);

% consider the dispersion

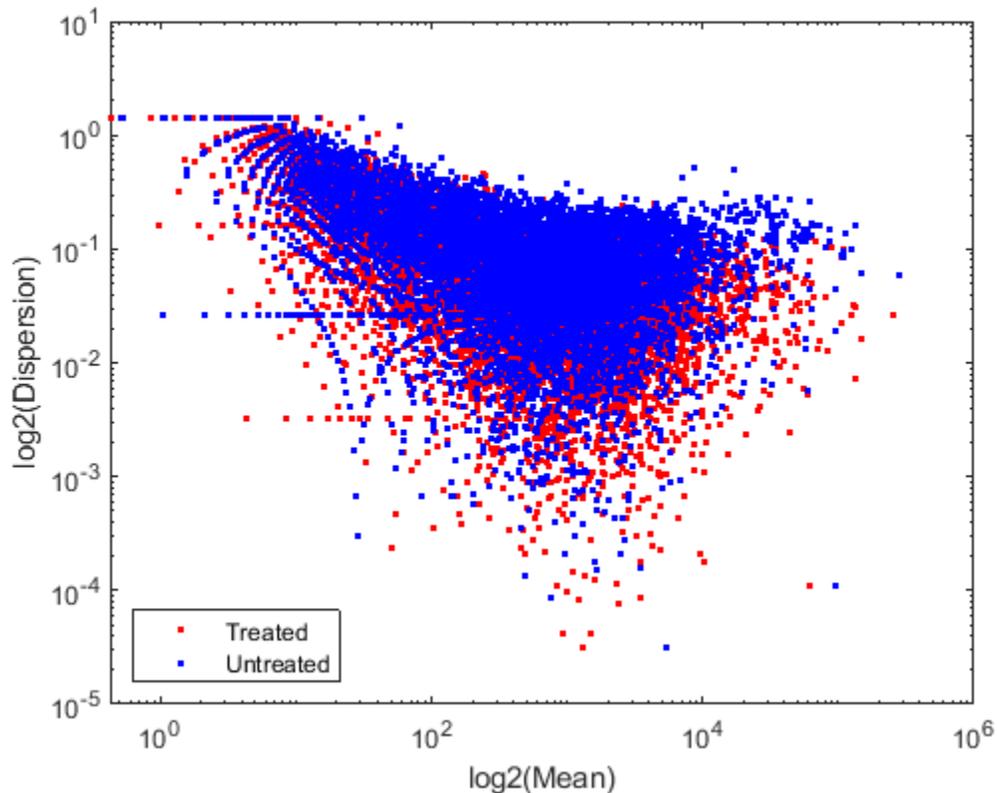
```

```

dispTreated = std(normCounts(:,treated),0,2) ./ meanTreated;
dispUntreated = std(normCounts(:,untreated),0,2) ./ meanUntreated;

% plot on a log-log scale
figure;
loglog(meanTreated,dispTreated,'r. ');
hold on;
loglog(meanUntreated,dispUntreated,'b. ');
xlabel('log2(Mean)');
ylabel('log2(Dispersion)');
legend('Treated','Untreated','Location','southwest');

```



Given the small number of replicates, it is not surprising to expect that the dispersion values scatter with some variance around the true value. Some of this variance reflects sampling variance and some reflects the true variability among the gene expressions of the samples.

You can look at the difference of the gene expression among two conditions, by calculating the fold change (FC) for each gene, i.e. the ratio between the counts in the treated group over the counts in the untreated group. Generally these ratios are considered in the log₂ scale, so that any change is symmetric with respect to zero (e.g. a ratio of 1/2 or 2/1 corresponds to -1 or +1 in the log scale).

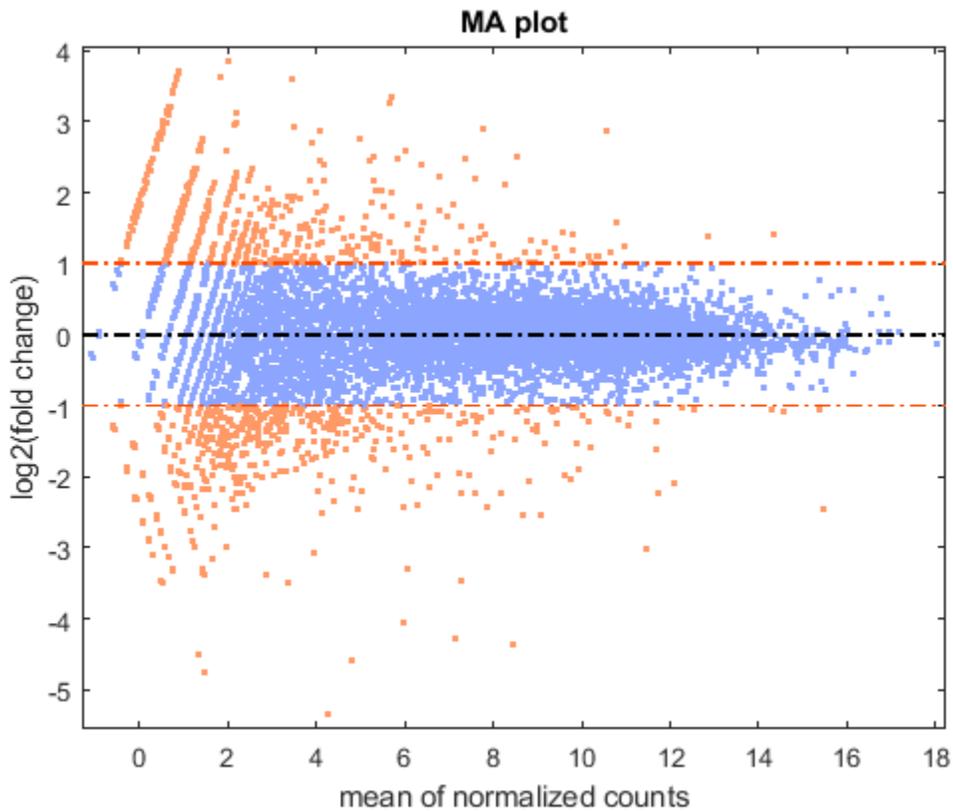
```

% compute the mean and the log2FC
meanBase = (meanTreated + meanUntreated) / 2;
foldChange = meanTreated ./ meanUntreated;
log2FC = log2(foldChange);

```

```
% plot mean vs. fold change (MA plot)
mairplot(meanTreated, meanUntreated, 'Type', 'MA', 'Plotonly', true);
set(get(gca, 'Xlabel'), 'String', 'mean of normalized counts')
set(get(gca, 'Ylabel'), 'String', 'log2(fold change)')
```

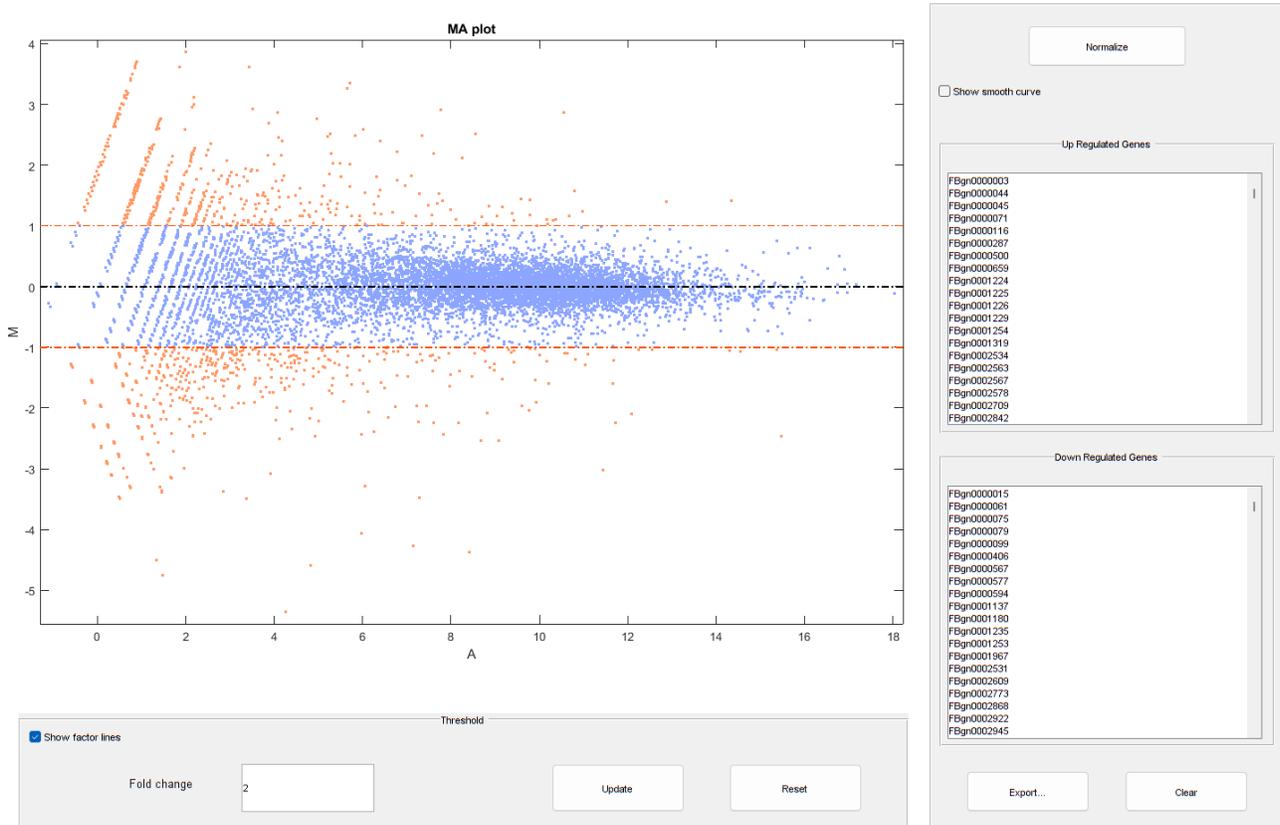
Warning: Zero values are ignored



It is possible to annotate the values in the plot with the corresponding gene names, interactively select genes, and export gene lists to the workspace by calling the `mairplot` function as illustrated below:

```
mairplot(meanTreated, meanUntreated, 'Labels', geneCountTable.ID, 'Type', 'MA');
```

Warning: Zero values are ignored



It is convenient to store the information about the mean value and fold change for each gene in a table. You can then access information about a given gene or a group of genes satisfying specific criteria by indexing the table by gene names.

```
% create table with statistics about each gene
geneTable = table(meanBase,meanTreated,meanUntreated,foldChange,log2FC);
geneTable.Properties.RowNames = geneCountTable.ID;
```

```
% summary
summary(geneTable)
```

Variables:

meanBase: 11609×1 double

Values:

Min	0.21206
Median	201.24
Max	2.6789e+05

meanTreated: 11609×1 double

```

Values:
      Min           0
      Median       201.54
      Max         2.5676e+05

meanUntreated: 11609x1 double

Values:
      Min           0
      Median       199.44
      Max         2.7903e+05

foldChange: 11609x1 double

Values:
      Min           0
      Median       0.99903
      Max          Inf

log2FC: 11609x1 double

Values:
      Min          -Inf
      Median      -0.001406
      Max          Inf

% preview
head(geneTable)

      meanBase    meanTreated    meanUntreated    foldChange    log2FC
-----
FBgn0000003    0.9475      1.3808      0.51415      2.6857      1.4253
FBgn0000008    132.69     129.48     135.9      0.95277     -0.069799
FBgn0000014    15.111     13.384     16.838     0.79488     -0.33119
FBgn0000015    1.7738     0.42413    3.1234     0.13579     -2.8806
FBgn0000017    5634.6     5117.4     6151.8     0.83186     -0.26559
FBgn0000018    514.08     505.48     522.67     0.96711     -0.048243
FBgn0000024    7.2354     8.7189     5.752      1.5158     0.60009
FBgn0000028    0.74465    1.4893     0          Inf         Inf

% access information about a specific gene
myGene = 'FBgn0261570';
geneTable(myGene,:)
geneTable(myGene,{'meanBase','log2FC'})

% access information about a given gene list
myGeneSet = ["FBgn0261570","FBgn0261573","FBgn0261575","FBgn0261560"];
geneTable(myGeneSet,:)

```

```
ans =
```

```
1x5 table
```

	meanBase	meanTreated	meanUntreated	foldChange	log2FC
FBgn0261570	4435.5	4939.1	3931.8	1.2562	0.32907

```
ans =
```

```
1x2 table
```

	meanBase	log2FC
FBgn0261570	4435.5	0.32907

```
ans =
```

```
4x5 table
```

	meanBase	meanTreated	meanUntreated	foldChange	log2FC
FBgn0261570	4435.5	4939.1	3931.8	1.2562	0.32907
FBgn0261573	2936.9	2954.8	2919.1	1.0122	0.01753
FBgn0261575	4.3776	5.6318	3.1234	1.8031	0.85047
FBgn0261560	2041.1	1494.3	2588	0.57738	-0.7924

Inferring Differential Expression with a Negative Binomial Model

Determining whether the gene expressions in two conditions are statistically different consists of rejecting the null hypothesis that the two data samples come from distributions with equal means. This analysis assumes the read counts are modeled according to a negative binomial distribution (as proposed in [3]). The function `rnaseqde` performs this type of hypothesis testing with three possible options to specify the type of linkage between the variance and the mean.

By specifying the link between variance and mean as an identity, we assume the variance is equal to the mean, and the counts are modeled by the Poisson distribution [4]. "IDColumns" specifies columns from the input table to append to the output table to help keep data organized.

```
diffTableIdentity = rnaseqde(geneCountTable, ["untreated3", "untreated4"], ["treated2", "treated3"],
```

```
% Preview the results.
```

```
head(diffTableIdentity)
```

ID	Mean1	Mean2	Log2FoldChange	PValue	AdjustedPValue
"FBgn0000003"	0.51415	1.3808	1.4253	0.627	0.75892
"FBgn0000008"	135.9	129.48	-0.069799	0.48628	0.64516
"FBgn0000014"	16.838	13.384	-0.33119	0.44445	0.61806
"FBgn0000015"	3.1234	0.42413	-2.8806	0.05835	0.12584

```

"FBgn0000017" 6151.8 5117.4 -0.26559 2.864e-42 6.0233e-41
"FBgn0000018" 522.67 505.48 -0.048243 0.39015 0.5616
"FBgn0000024" 5.752 8.7189 0.60009 0.35511 0.52203
"FBgn0000028" 0 1.4893 Inf 0.252 0.39867

```

Alternatively, by specifying the variance as the sum of the shot noise term (i.e. mean) and a constant multiplied by the squared mean, the counts are modeled according to a distribution described in [5]. The constant term is estimated using all the rows in the data.

```
diffTableConstant = rnaseqde(geneCountTable, ["untreated3", "untreated4"], ["treated2", "treated3"], Va
```

```
% Preview the results.
head(diffTableConstant)
```

ID	Mean1	Mean2	Log2FoldChange	PValue	AdjustedPValue
"FBgn0000003"	0.51415	1.3808	1.4253	0.62769	0.7944
"FBgn0000008"	135.9	129.48	-0.069799	0.53367	0.72053
"FBgn0000014"	16.838	13.384	-0.33119	0.45592	0.68454
"FBgn0000015"	3.1234	0.42413	-2.8806	0.058924	0.16938
"FBgn0000017"	6151.8	5117.4	-0.26559	8.5529e-05	0.00077269
"FBgn0000018"	522.67	505.48	-0.048243	0.54834	0.73346
"FBgn0000024"	5.752	8.7189	0.60009	0.36131	0.58937
"FBgn0000028"	0	1.4893	Inf	0.2527	0.46047

Finally, by considering the variance as the sum of the shot noise term (i.e. mean) and a locally regressed non-parametric smooth function of the mean, the counts are modeled according to the distribution proposed in [3].

```
diffTableLocal = rnaseqde(geneCountTable, ["untreated3", "untreated4"], ["treated2", "treated3"], Va
```

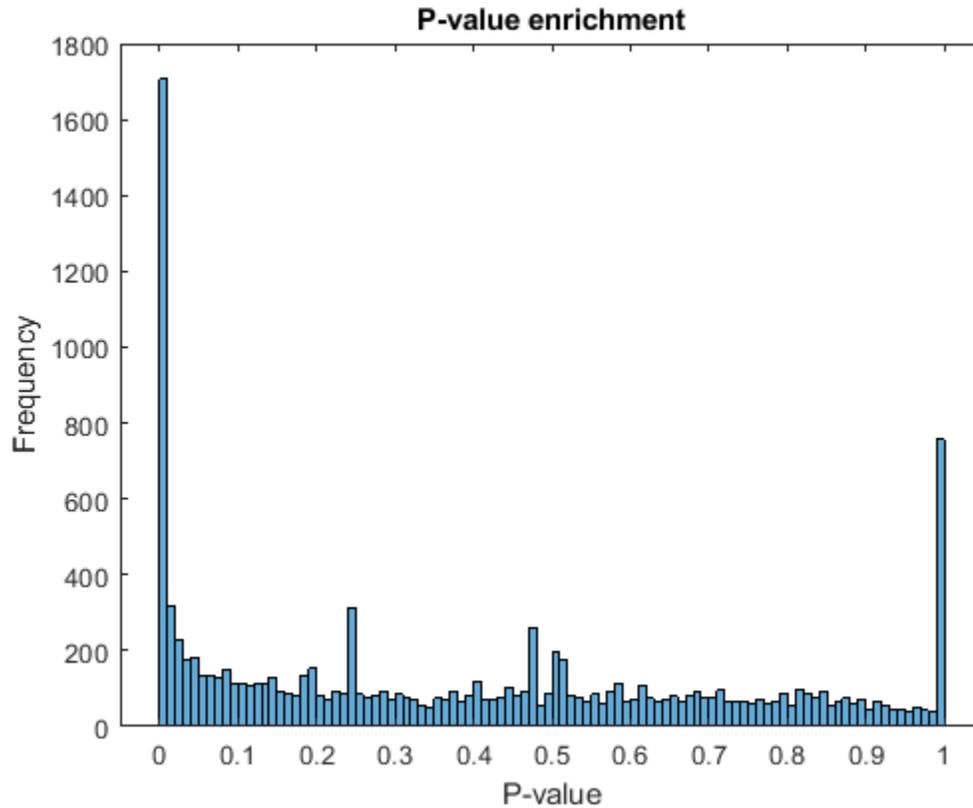
```
% Preview the results.
head(diffTableLocal)
```

ID	Mean1	Mean2	Log2FoldChange	PValue	AdjustedPValue
"FBgn0000003"	0.51415	1.3808	1.4253	1	1
"FBgn0000008"	135.9	129.48	-0.069799	0.67298	0.89231
"FBgn0000014"	16.838	13.384	-0.33119	0.6421	0.87234
"FBgn0000015"	3.1234	0.42413	-2.8806	0.22776	0.57215
"FBgn0000017"	6151.8	5117.4	-0.26559	0.0014429	0.014207
"FBgn0000018"	522.67	505.48	-0.048243	0.65307	0.88136
"FBgn0000024"	5.752	8.7189	0.60009	0.55154	0.81984
"FBgn0000028"	0	1.4893	Inf	0.42929	0.7765

The output of `rnaseqde` includes a vector of P-values. A P-value indicates the probability that a change in expression as strong as the one observed (or even stronger) would occur under the null hypothesis, i.e. the conditions have no effect on gene expression. In the histogram of the P-values we observe an enrichment of low values (due to differentially expressed genes), whereas other values are uniformly spread (due to non-differentially expressed genes). The enrichment of values equal to 1 are due to genes with very low counts.

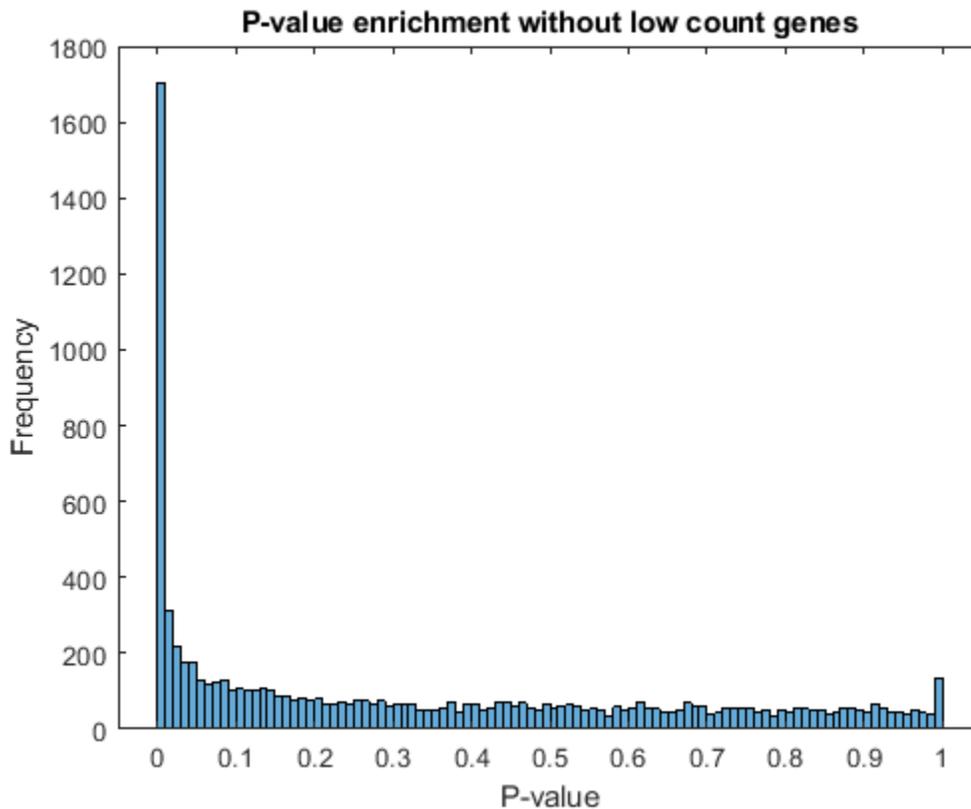
```
figure;
histogram(diffTableLocal.PValue, 100)
```

```
xlabel('P-value')
ylabel('Frequency')
title('P-value enrichment')
```



Filter out those genes with relatively low count to observe a more uniform spread of non-significant P-values across the range (0,1]. Note that this does not affect the distribution of significant P-values.

```
lowCountThreshold = 10;
lowCountGenes = all(counts < lowCountThreshold, 2);
histogram(diffTableLocal.PValue(~lowCountGenes),100)
xlabel('P-value')
ylabel('Frequency')
title('P-value enrichment without low count genes')
```



Multiple Testing and Adjusted P-values

Thresholding P-values to determine what fold changes are more significant than others is not appropriate for this type of data analysis, due to the multiple testing problem. While performing a large number of simultaneous tests, the probability of getting a significant result simply due to chance increases with the number of tests. In order to account for multiple testing, perform a correction (or adjustment) of the P-values so that the probability of observing at least one significant result due to chance remains below the desired significance level.

The Benjamini-Hochberg adjustment [6] is a statistical method that provides an adjusted P-value answering the following question: what would be the fraction of false positives if all the genes with adjusted P-values below a given threshold were considered significant?

The output of `rnaseqde` includes a vector of adjusted P-values in the "AdjustedPValue" field. By default, the P-values are adjusted using the Benjamini-Hochberg adjustment. Alternatively, the "FDRMethod" Name-Value argument in `rnaseqde` can be set to "storey" to perform Storey's procedure [7].

Set a threshold of 0.1 for the adjusted P-values, equivalent to consider a 10% false positives as acceptable, and identify the genes that are significantly expressed by considering all the genes with adjusted P-values below this threshold.

```
% create a table with significant genes
sig = diffTableLocal.AdjustedPValue < 0.1;
diffTableLocalSig = diffTableLocal(sig,:);
```

```
diffTableLocalSig = sortrows(diffTableLocalSig, 'AdjustedPValue');
numberSigGenes = size(diffTableLocalSig,1)
```

```
numberSigGenes =
    1904
```

Identifying the Most Up-regulated and Down-regulated Genes

You can now identify the most up-regulated or down-regulated genes by considering an absolute fold change above a chosen cutoff. For example, a cutoff of 1 in log₂ scale yields the list of genes that are up-regulated with a 2 fold change.

```
% find up-regulated genes
up = diffTableLocalSig.Log2FoldChange > 1;
upGenes = sortrows(diffTableLocalSig(up,:), 'Log2FoldChange', 'descend');
numberSigGenesUp = sum(up)

% display the top 10 up-regulated genes
top10GenesUp = upGenes(1:10,:)

% find down-regulated genes
down = diffTableLocalSig.Log2FoldChange < -1;
downGenes = sortrows(diffTableLocalSig(down,:), 'Log2FoldChange', 'ascend');
numberSigGenesDown = sum(down)

% find top 10 down-regulated genes
top10GenesDown = downGenes(1:10,)
```

```
numberSigGenesUp =
    129
```

```
top10GenesUp =
    10x6 table
```

ID	Mean1	Mean2	Log2FoldChange	PValue	AdjustedPValue
"FBgn0030173"	0	6.7957	Inf	0.0063115	0.047764
"FBgn0036822"	0	6.2729	Inf	0.012203	0.079274
"FBgn0052548"	1.0476	15.269	3.8654	0.00016945	0.0022662
"FBgn0050495"	1.0283	12.635	3.6191	0.0018949	0.017972
"FBgn0063667"	3.1042	38.042	3.6153	8.5037e-08	2.3845e-06
"FBgn0033764"	16.324	167.61	3.3601	1.8345e-25	2.9174e-23
"FBgn0037290"	16.228	155.46	3.26	3.5583e-23	4.6941e-21
"FBgn0033733"	1.5424	13.384	3.1172	0.0027276	0.024283
"FBgn0037191"	1.6003	12.753	2.9945	0.0047803	0.038193
"FBgn0033943"	1.581	12.319	2.962	0.0053635	0.041986

```
numberSigGenesDown =
```

181

top10GenesDown =

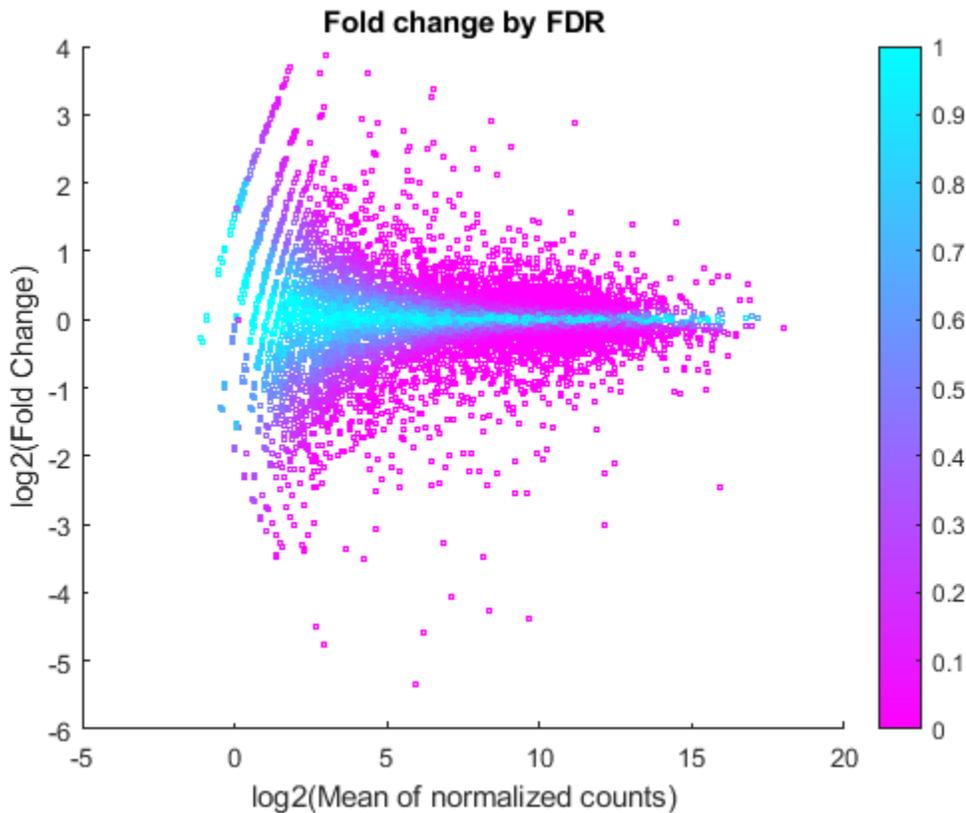
10x6 table

ID	Mean1	Mean2	Log2FoldChange	PValue	AdjustedPValue
"FBgn0053498"	30.938	0	-Inf	9.8404e-11	4.345e-09
"FBgn0259236"	13.618	0	-Inf	1.5526e-05	0.00027393
"FBgn0052500"	8.7405	0	-Inf	0.00066783	0.0075343
"FBgn0039331"	7.3908	0	-Inf	0.0019558	0.018474
"FBgn0040697"	6.8381	0	-Inf	0.0027378	0.024336
"FBgn0034972"	5.8291	0	-Inf	0.0068564	0.05073
"FBgn0040967"	5.2764	0	-Inf	0.0096039	0.065972
"FBgn0031923"	4.7429	0	-Inf	0.016164	0.098762
"FBgn0085359"	121.97	2.9786	-5.3557	5.5813e-33	1.5068e-30
"FBgn0004854"	14.402	0.53259	-4.7571	8.1587e-05	0.0012034

A good visualization of the gene expressions and their significance is given by plotting the fold change versus the mean in log scale and coloring the data points according to the adjusted P-values.

figure

```
scatter(log2(geneTable.meanBase),diffTableLocal.Log2FoldChange,3,diffTableLocal.PValue,'o')
colormap(flipud(cool(256)))
colorbar;
ylabel('log2(Fold Change)')
xlabel('log2(Mean of normalized counts)')
title('Fold change by FDR')
```



You can see here that for weakly expressed genes (i.e. those with low means), the FDR is generally high because low read counts are dominated by Poisson noise and consequently any biological variability is drowned in the uncertainties from the read counting.

References

- [1] Brooks et al. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research* 2011. 21:193-202.
- [2] Mortazavi et al. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature Methods* 2008. 5:621-628.
- [3] Anders et al. Differential expression analysis for sequence count data. *Genome Biology* 2010. 11:R106.
- [4] Marioni et al. RNA-Seq: An assessment of technical reproducibility and comparison with gene expression arrays. *Genome Research* 2008. 18:1509-1517.
- [5] Robinson et al. Moderated statistical test for assessing differences in tag abundance. *Bioinformatics* 2007. 23(21):2881-2887.
- [6] Benjamini et al. Controlling the false discovery rate: a practical and powerful approach to multiple testing. 1995. *Journal of the Royal Statistical Society, Series B* 57 (1):289-300.

[7] J.D. Storey. "A direct approach to false discovery rates", Journal of the Royal Statistical Society, B (2002), 64(3), pp.479-498.

See Also

`featurecount` | `nbintest` | `mairplot` | `plotVarianceLink`

More About

- "High-Throughput Sequencing"

Visualize NGS Data Using Genomics Viewer App

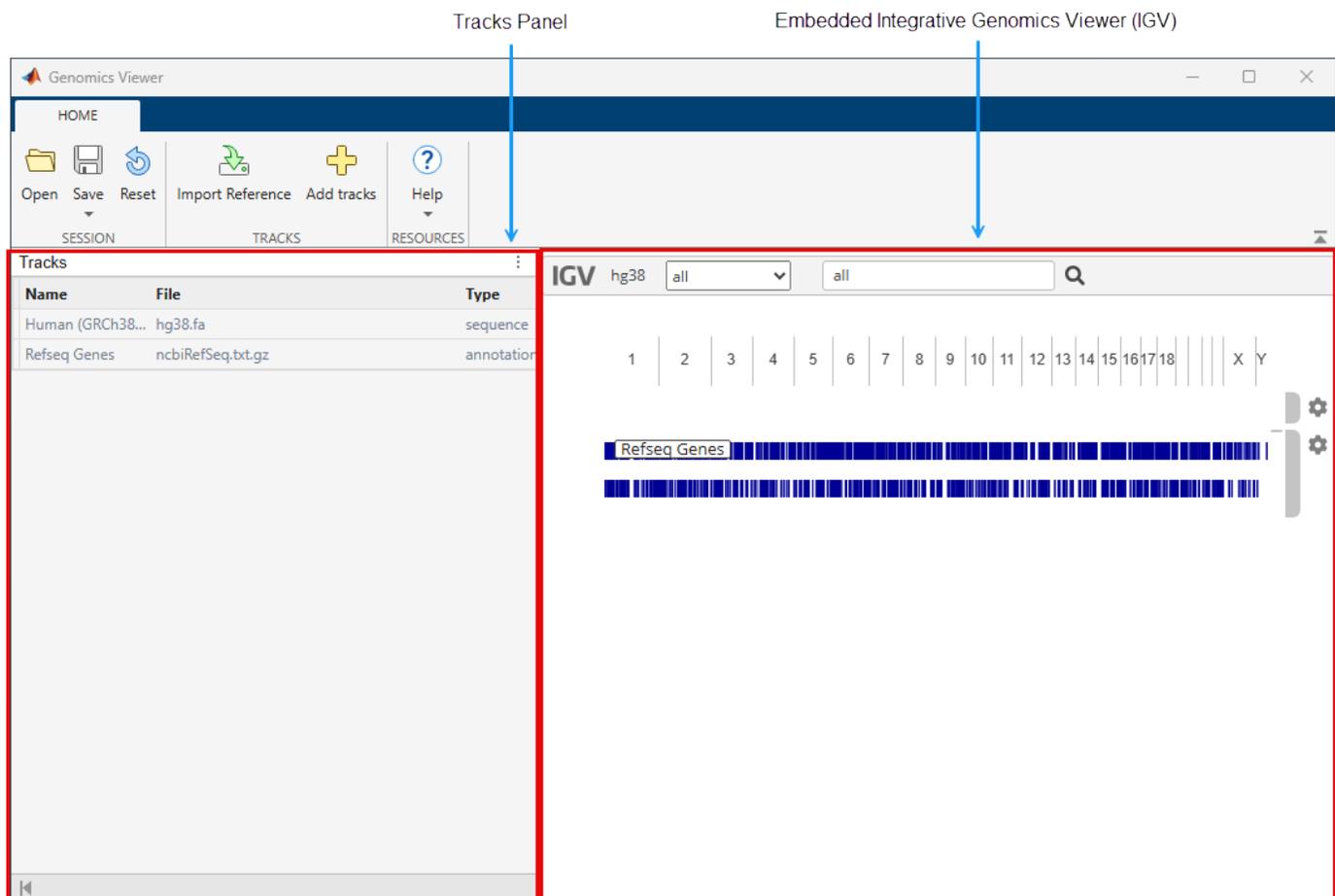
The Genomics Viewer app lets you view and explore integrated genomic data with an embedded version of the Integrative Genomics Viewer (IGV) [1][2]. The genomic data include NGS read alignments, genome variants, and segmented copy number data.

The first part of this example gives a brief overview of the app and supported file formats. The second part of the example explores a single nucleotide variation in the cytochrome p450 gene (CYP2C19).

Open the App

At the command line, type `genomicsViewer`. Alternatively, click the app icon on the **Apps** tab. The app requires an internet connection.

By default, the app loads Human (GRCh38/hg38) as the reference sequence and Refseq Genes as the annotation file. There are two main panels in the app. The left panel is the **Tracks** panel and the right panel is the embedded IGV web application. The **Tracks** panel is a *read-only* area displaying the track names, source file names, and track types. The **Tracks** panel updates accordingly as you configure the tracks in the embedded IGV app.



The **Reset** button restores the app to the default view with two tracks (HG38 with Refseq Genes) and removes any other existing tracks. Before resetting, you can save the current view as a session (.json) file and restore it later.

Add Tracks by Importing Data

You can import data from local files or specify URLs. If you are specifying a URL, the URL must start with `http`, `https`, or `gs`. Other file transfer protocols, such as `ftp`, are not supported. For a list of supported data file formats, see <https://igvteam.github.io/igv-webapp/fileFormats.html>.

Import Reference Sequence

You can import a single reference sequence. The reference sequence must be in a FASTA file. Select **Import Reference** on the **Home** tab. You can also import a corresponding cytoband file that contains cytogenetic G-banding data.

Import Sequence Read Alignment Data

You can import multiple data sets of sequence read alignment data. The alignment data must be a BAM or CRAM file. It is not required that you have the corresponding index file (.BAI or .CRAI) in the same location as your BAM or CRAM file. However, the absence of the index file will make the app slower.

You can add read alignment files using **Add tracks from file** and **Add tracks from URL** options from the **Add tracks** button.

Import Feature Annotations and Other Genomic Data

You can import multiple sets of feature annotations from several files that contain data for a single reference sequence. The supported annotation files are: .BED, .GFF, .GFF3, and .GTF.

You can also import structural variants (.VCF) and visualize genetic alterations, such as insertions and deletions.

You can view segmented copy number data (.SEG) and quantitative genomic data (.WIG, .BIGWIG, and .BEDGRAPH), such as ChIP peaks and alignment coverage.

You can add annotation and genomic data files using **Add tracks from file** and **Add tracks from URL** options from the **Add tracks** button.

Visualize Single Nucleotide Variation in Cytochrome P450

The *CYP2C19* gene is a member of the cytochrome P450 gene family. Enzymes produced from cytochrome P450 genes are involved in the metabolism of various molecules and chemicals within cells. The CYP2C19 enzyme plays a role in the metabolizing of at least 10 percent of commonly prescribed drugs [3]. Polymorphisms in the cytochrome p450 family may cause adverse drug responses in individuals. One example of single nucleotide variation is *rs4986893* at position *chr10:94,780,653* where G is replaced by A. This allelic variant is also known as *CYP2C19*3*. The following steps show how to visualize such variation in the app using both low coverage and high coverage data.

Load Session File

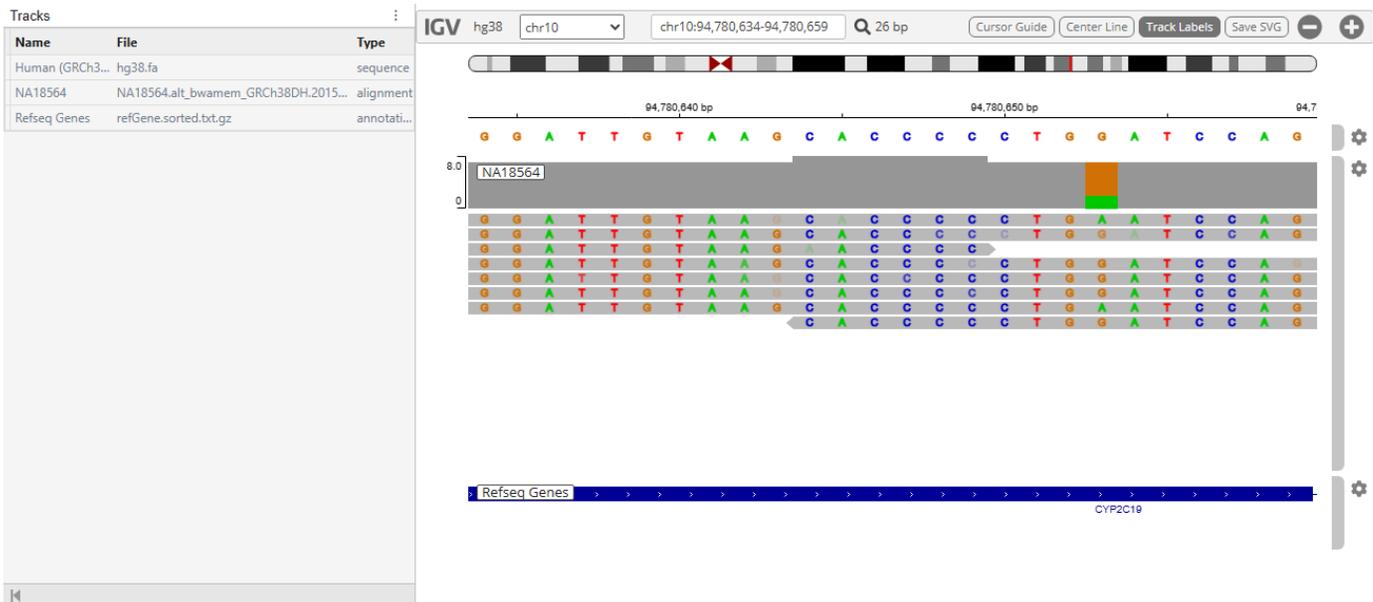
For the purposes of this example, start with a session file (`rs4986893.json`) that has some preloaded tracks. After downloading the file, load it in the app. Click **Open** and select `rs4986893.json`.

Explore Low Coverage Data

The session contains three tracks:

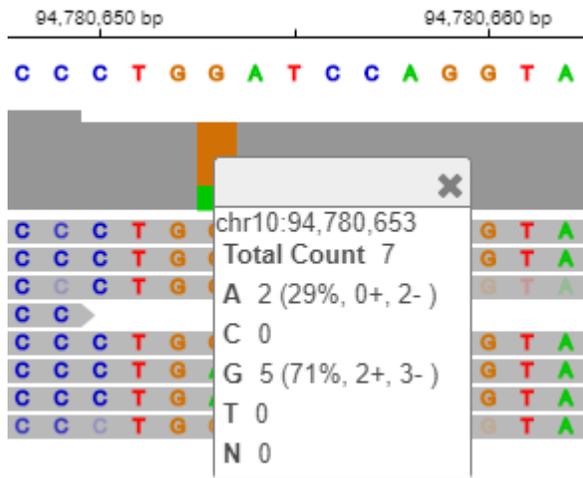
- *Human (GRCh38/hg38)* as a reference
- *NA18564* as low coverage alignment data
- Refseq Genes

The low coverage alignment data comes from a female Han Chinese from Beijing, China. The sample ID is *NA18564* and the sample has been identified with the *CYP2C19**3 mutation [4].



The alignment data has been centered around the location of the mutation on the *CYP2C19* gene.

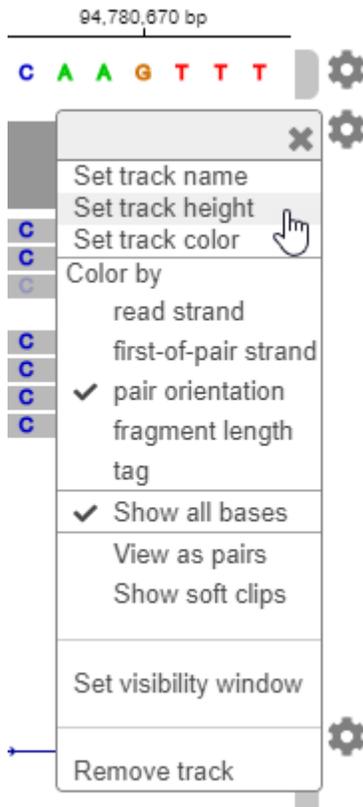
- 1 Click the orange bar in the coverage area to look at the position and allele distribution information.



It shows that 71% of the reads have G while 29% have A at the location *chr10:94,780,653*. This data is a low coverage data and may not show all the occurrences of this mutation. A high coverage data will be explored later in the example.

Close the data tip window.

- 2 You can customize the various aspects of the data display in the app. For example, you can change the track height to make more room for later tracks. Click the second gear icon. Select **Set track height**. Enter 200.

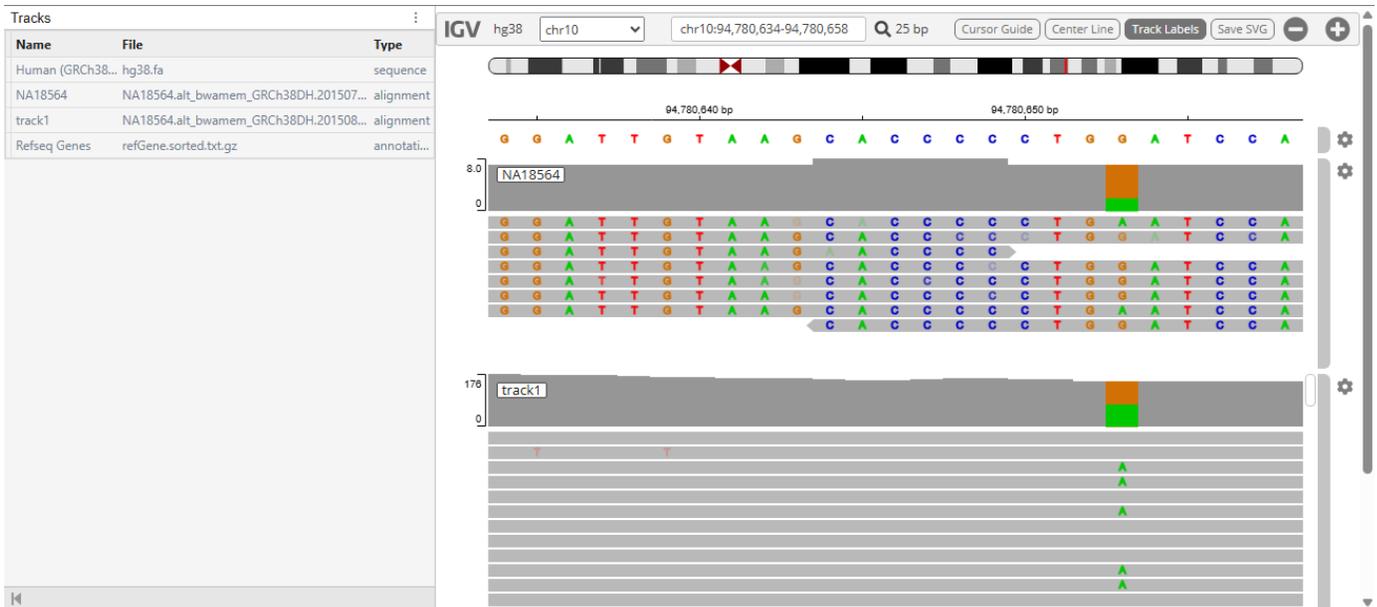


For details on the embedded IGV app and its available options, visit [here](#).

Explore High Coverage Data

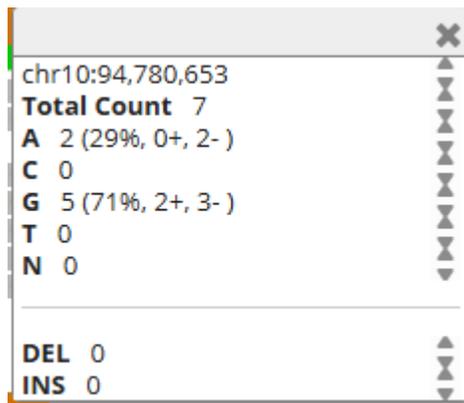
You can look at the high coverage data from the same sample to see the occurrences of this mutation.

- 1 Go to The International Genome Sample Resource website.
- 2 Search for the sample *NA18564*.
- 3 Download the *Exome* alignment file that is in the .CRAM format.
- 4 Also download the corresponding index file that is in the .CRAI format. Save the file in the same location as the source .CRAM file.
- 5 Click the (+) icon on the **Home** tab. Select the downloaded .CRAM file and click **Open**.



The high coverage data appears as a track. You can now see many occurrences of the mutation in several reads.

- 6 Click the orange bar in the coverage area to see the allele distribution.



References

- [1] Robinson, J., H. Thorvaldsdóttir, W. Winckler, M. Guttman, E. Lander, G. Getz, J. Mesirov. 2011. Integrative Genomics Viewer. *Nature Biotechnology*. 29:24-26.
- [2] Thorvaldsdóttir, H., J. Robinson, J. Mesirov. 2013. Integrative Genomics Viewer (IGV): High-performance genomics data visualization and exploration. *Briefings in Bioinformatics*. 14:178-192.
- [3] <https://medlineplus.gov/genetics/gene/cyp2c19/>
- [4] https://www.coriell.org/0/Sections/Search/Sample_Detail.aspx?Ref=NA18564&Product=DNA

See Also

Genomics Viewer | `bioinfo.pipeline.block.GenomicsViewer` | Sequence Alignment

Exploring Genome-Wide Differences in DNA Methylation Profiles

This example shows how to perform a genome-wide analysis of DNA methylation in the human by using genome sequencing.

Note: For enhanced performance, MathWorks recommends that you run this example on a 64-bit platform, because the memory footprint is close to 2 GB. On a 32-bit platform, if you receive "Out of memory" errors when running this example, try increasing the virtual memory (or swap space) of your operating system or try setting the 3GB switch (32-bit Windows® XP only). For details, see "Resolve "Out of Memory" Errors".

Introduction

DNA methylation is an epigenetic modification that modulates gene expression and the maintenance of genomic organization in normal and disease processes. DNA methylation can define different states of the cell, and it is inheritable during cell replication. Aberrant DNA methylation patterns have been associated with cancer and tumor suppressor genes.

In this example you will explore the DNA methylation profiles of two human cancer cells: parental HCT116 colon cancer cells and DICERex5 cells. DICERex5 cells are derived from HCT116 cells after the truncation of the DICER1 alleles. Serre et al. in [1] proposed to study DNA methylation profiles by using the MBD2 protein as a methyl CpG binding domain and subsequently used high-throughput sequencing (HTseq). This technique is commonly known as MBD-Seq. Short reads for two replicates of the two samples have been submitted to NCBI's SRA archive by the authors of [1]. There are other technologies available to interrogate DNA methylation status of CpG sites in combination with HTseq, for example MeDIP-seq or the use of restriction enzymes. You can also analyze this type of data sets following the approach presented in this example.

Data Sets

You can obtain the unmapped single-end reads for four sequencing experiments from NCBI. Short reads were produced using Illumina®'s Genome Analyzer II. Average insert size is 120 bp, and the length of short reads is 36 bp.

This example assumes that you:

- (1) downloaded the files `SRR030222.sra`, `SRR030223.sra`, `SRR030224.sra` and `SRR030225.sra` containing the unmapped short reads for two replicates of from the DICERex5 sample and two replicates from the HCT116 sample respectively, from NCBI SRA Run Selector and converted them to FASTQ-formatted files using the NCBI SRA Toolkit.
- (2) produced SAM-formatted files by mapping the short reads to the reference human genome (NCBI Build 37.5) using the Bowtie [2] algorithm. Only uniquely mapped reads are reported.
- (3) compressed the SAM formatted files to BAM and ordered them by reference name first, then by genomic position by using SAMtools [3].

This example also assumes that you downloaded the reference human genome (GRCh37.p5). You can use the `bowtie-inspect` command to reconstruct the human reference directly from the bowtie indices. Or you may download the reference from the NCBI repository by uncommenting the following line:

```
% getgenbank('NC_000009', 'FileFormat', 'fasta', 'tofile', 'hsch9.fasta');
```

Creating a MATLAB® Interface to the BAM-Formatted Files

To explore the signal coverage of the HCT116 samples you need to construct a BioMap. BioMap has an interface that provides direct access to the mapped short reads stored in the BAM-formatted file, thus minimizing the amount of data that is actually loaded into memory. Use the function `baminfo` to obtain a list of the existing references and the actual number of short reads mapped to each one.

```
info = baminfo('SRR030224.bam', 'ScanDictionary', true);
fprintf('%-35s\n', 'Reference', 'Number of Reads');
for i = 1:numel(info.ScannedDictionary)
    fprintf('%-35s\n', info.ScannedDictionary{i}, ...
        info.ScannedDictionaryCount(i));
end
```

Reference	Number of Reads
gi 224589800 ref NC_000001.10	205065
gi 224589811 ref NC_000002.11	187019
gi 224589815 ref NC_000003.11	73986
gi 224589816 ref NC_000004.11	84033
gi 224589817 ref NC_000005.9	96898
gi 224589818 ref NC_000006.11	87990
gi 224589819 ref NC_000007.13	120816
gi 224589820 ref NC_000008.10	111229
gi 224589821 ref NC_000009.11	106189
gi 224589801 ref NC_000010.10	112279
gi 224589802 ref NC_000011.9	104466
gi 224589803 ref NC_000012.11	87091
gi 224589804 ref NC_000013.10	53638
gi 224589805 ref NC_000014.8	64049
gi 224589806 ref NC_000015.9	60183
gi 224589807 ref NC_000016.9	146868
gi 224589808 ref NC_000017.10	195893
gi 224589809 ref NC_000018.9	60344
gi 224589810 ref NC_000019.9	166420
gi 224589812 ref NC_000020.10	148950
gi 224589813 ref NC_000021.8	310048
gi 224589814 ref NC_000022.10	76037
gi 224589822 ref NC_000023.10	32421
gi 224589823 ref NC_000024.9	18870
gi 17981852 ref NC_001807.4	1015
Unmapped	6805842

In this example you will focus on the analysis of chromosome 9. Create a BioMap for the two HCT116 sample replicates.

```
bm_hct116_1 = BioMap('SRR030224.bam', 'SelectRef', 'gi|224589821|ref|NC_000009.11|')
bm_hct116_2 = BioMap('SRR030225.bam', 'SelectRef', 'gi|224589821|ref|NC_000009.11|')
```

```
bm_hct116_1 =
```

```
BioMap with properties:
```

```
SequenceDictionary: 'gi|224589821|ref|NC_000009.11|'
Reference: [106189x1 File indexed property]
```

```

Signature: [106189x1 File indexed property]
Start: [106189x1 File indexed property]
MappingQuality: [106189x1 File indexed property]
Flag: [106189x1 File indexed property]
MatePosition: [106189x1 File indexed property]
Quality: [106189x1 File indexed property]
Sequence: [106189x1 File indexed property]
Header: [106189x1 File indexed property]
NSeqs: 106189
Name: ''

```

```
bm_hct116_2 =
```

```
BioMap with properties:
```

```

SequenceDictionary: 'gi|224589821|ref|NC_000009.11|'
Reference: [107586x1 File indexed property]
Signature: [107586x1 File indexed property]
Start: [107586x1 File indexed property]
MappingQuality: [107586x1 File indexed property]
Flag: [107586x1 File indexed property]
MatePosition: [107586x1 File indexed property]
Quality: [107586x1 File indexed property]
Sequence: [107586x1 File indexed property]
Header: [107586x1 File indexed property]
NSeqs: 107586
Name: ''

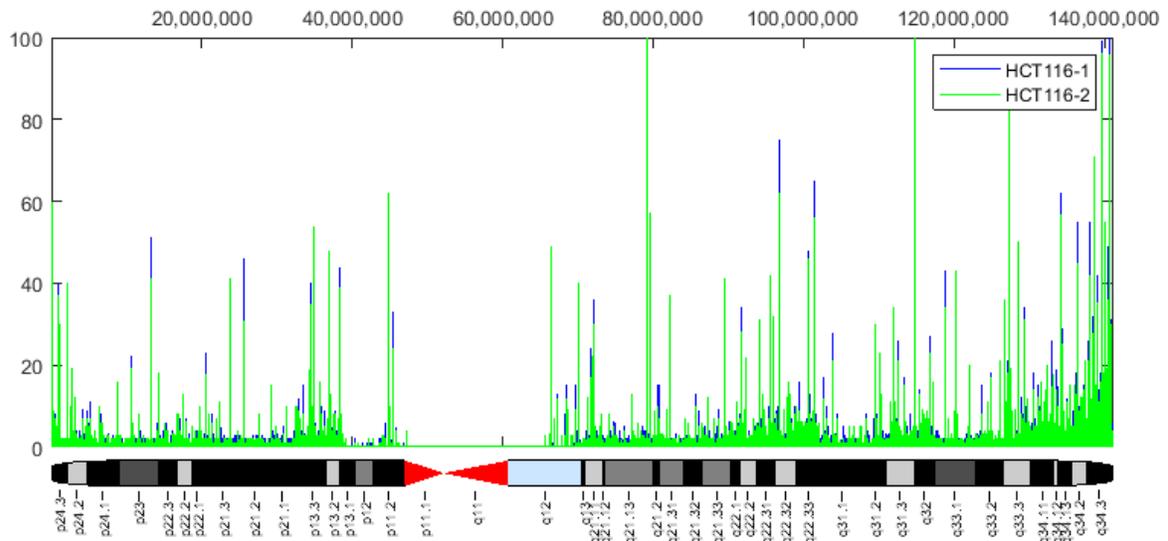
```

Using a binning algorithm provided by the `getBaseCoverage` method, you can plot the coverage of both replicates for an initial inspection. For reference, you can also add the ideogram for the human chromosome 9 to the plot using the `chromosomeplot` function.

```

figure
ha = gca;
hold on
n = 141213431; % length of chromosome 9
[cov,bin] = getBaseCoverage(bm_hct116_1,1,n,'binWidth',100);
h1 = plot(bin,cov,'b'); % plots the binned coverage of bm_hct116_1
[cov,bin] = getBaseCoverage(bm_hct116_2,1,n,'binWidth',100);
h2 = plot(bin,cov,'g'); % plots the binned coverage of bm_hct116_2
chromosomeplot('hs_cytoBand.txt', 9, 'AddToPlot', ha) % plots an ideogram along the x-axis
axis(ha,[1 n 0 100]) % zooms-in the y-axis
fixGenomicPositionLabels(ha) % formats tick labels and adds data cursors
legend([h1 h2], 'HCT116-1', 'HCT116-2', 'Location', 'NorthEast')
ylabel('Coverage')
title('Coverage for two replicates of the HCT116 sample')
fig = gcf;
fig.Position = max(fig.Position,[0 0 900 0]); % resize window

```



Because short reads represent the methylated regions of the DNA, there is a correlation between aligned coverage and DNA methylation. Observe the increased DNA methylation close to the chromosome telomeres; it is known that there is an association between DNA methylation and the role of telomeres for maintaining the integrity of the chromosomes. In the coverage plot you can also see a long gap over the chromosome centromere. This is due to the repetitive sequences present in the centromere, which prevent us from aligning short reads to a unique position in this region. In the data sets used in this example only about 30% of the short reads were uniquely mapped to the reference genome.

Correlating CpG Islands and DNA Methylation

DNA methylation normally occurs in CpG dinucleotides. Alteration of the DNA methylation patterns can lead to transcriptional silencing, especially in the gene promoter CpG islands. But, it is also known that DNA methylation can block CTCF binding and can silence miRNA transcription among other relevant functions. In general, it is expected that mapped reads should preferably align to CpG rich regions.

Load the human chromosome 9 from the reference file `hs37.fasta`. For this example, it is assumed that you recovered the reference from the Bowtie indices using the `bowtie-inspect` command; therefore `hs37.fasta` contains all the human chromosomes. To load only the chromosome 9 you can use the option `nave-value` pair `BLOCKREAD` with the `fastaread` function.

```
chr9 = fastaread('hs37.fasta', 'blockread', 9);
chr9.Header
```

```
ans =
```

```
'gi|224589821|ref|NC_000009.11| Homo sapiens chromosome 9, GRCh37 primary reference assembly
```

Use the `cpgisland` function to find the CpG clusters. Using the standard definition for CpG islands [4], 200 or more bp islands with 60% or greater CpGobserved/CpGexpected ratio, leads to 1682 CpG islands found in chromosome 9.

```
cpgi = cpgisland(chr9.Sequence)
```

```
cpgi =
```

```
struct with fields:
```

```
Starts: [10783 29188 73049 73686 113309 114488 116877 ... ] (1×1682 double)
Stops: [11319 29409 73624 73893 114336 114809 117105 ... ] (1×1682 double)
```

Use the `getCounts` method to calculate the ratio of aligned bases that are inside CpG islands. For the first replicate of the sample HCT116, the ratio is close to 45%.

```
aligned_bases_in_CpG_islands = getCounts(bm_hct116_1,cpgi.Starts,cpgi.Stops,'method','sum')
aligned_bases_total = getCounts(bm_hct116_1,1,n,'method','sum')
ratio = aligned_bases_in_CpG_islands ./ aligned_bases_total
```

```
aligned_bases_in_CpG_islands =
```

```
1724363
```

```
aligned_bases_total =
```

```
3822804
```

```
ratio =
```

```
0.4511
```

You can explore high resolution coverage plots of the two sample replicates and observe how the signal correlates with the CpG islands. For example, explore the region between 23,820,000 and 23,830,000 bp. This is the 5' region of the human gene ELAVL2.

```
r1 = 23820001; % set the region limits
r2 = 23830000;
fhELAVL2 = figure; % keep the figure handle to use it later
hold on
% plot high-resolution coverage of bm_hct116_1
h1 = plot(r1:r2,getBaseCoverage(bm_hct116_1,r1,r2,'binWidth',1),'b');
% plot high-resolution coverage of bm_hct116_2
h2 = plot(r1:r2,getBaseCoverage(bm_hct116_2,r1,r2,'binWidth',1),'g');

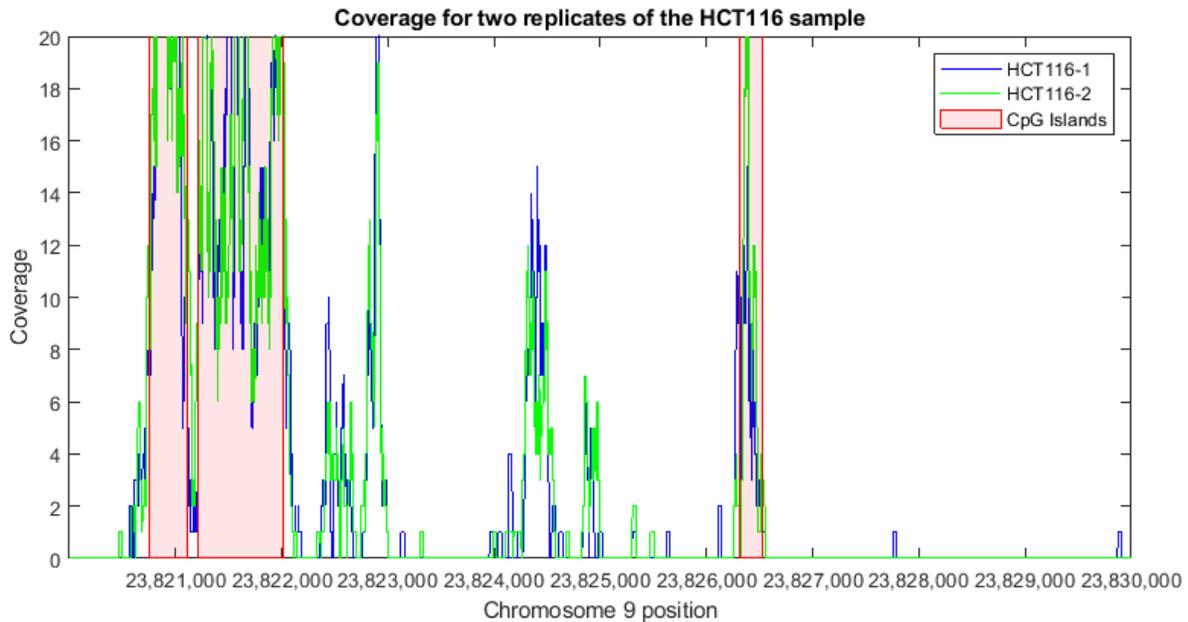
% mark the CpG islands within the [r1 r2] region
for i = 1:numel(cpgi.Starts)
    if cpgi.Starts(i)>r1 && cpgi.Stops(i)<r2 % is CpG island inside [r1 r2]?
        px = [cpgi.Starts([i i]) cpgi.Stops([i i])]; % x-coordinates for patch
        py = [0 max(ylim) max(ylim) 0]; % y-coordinates for patch
        hp = patch(px,py,'r','FaceAlpha',.1,'EdgeColor','r','Tag','cpgi');
    end
end
```

```

end

axis([r1 r2 0 20])           % zooms-in the y-axis
fixGenomicPositionLabels(gca) % formats tick labels and adds data cursors
legend([h1 h2 hp], 'HCT116-1', 'HCT116-2', 'CpG Islands')
ylabel('Coverage')
xlabel('Chromosome 9 position')
title('Coverage for two replicates of the HCT116 sample')

```



Statistical Modelling of Count Data

To find regions that contain more mapped reads than would be expected by chance, you can follow a similar approach to the one described by Serre et al. [1]. The number of counts for non-overlapping contiguous 100 bp windows is statistically modeled.

First, use the `getCounts` method to count the number of mapped reads that start at each window. In this example you use a binning approach that considers only the start position of every mapped read, following the approach of Serre et al. However, you may also use the `OVERLAP` and `METHOD` name-value pairs in `getCounts` to compute more accurate statistics. For instance, to obtain the maximum coverage for each window considering base pair resolution, set `OVERLAP` to 1 and `METHOD` to `MAX`.

```

n = numel(chr9.Sequence); % length of chromosome
w = 1:100:n; % windows of 100 bp

counts_1 = getCounts(bm_hct116_1,w,w+99,'independent',true,'overlap','start');
counts_2 = getCounts(bm_hct116_2,w,w+99,'independent',true,'overlap','start');

```

First, try to model the counts assuming that all the windows with counts are biologically significant and therefore from the same distribution. Use the negative binomial distribution to fit a model the count data.

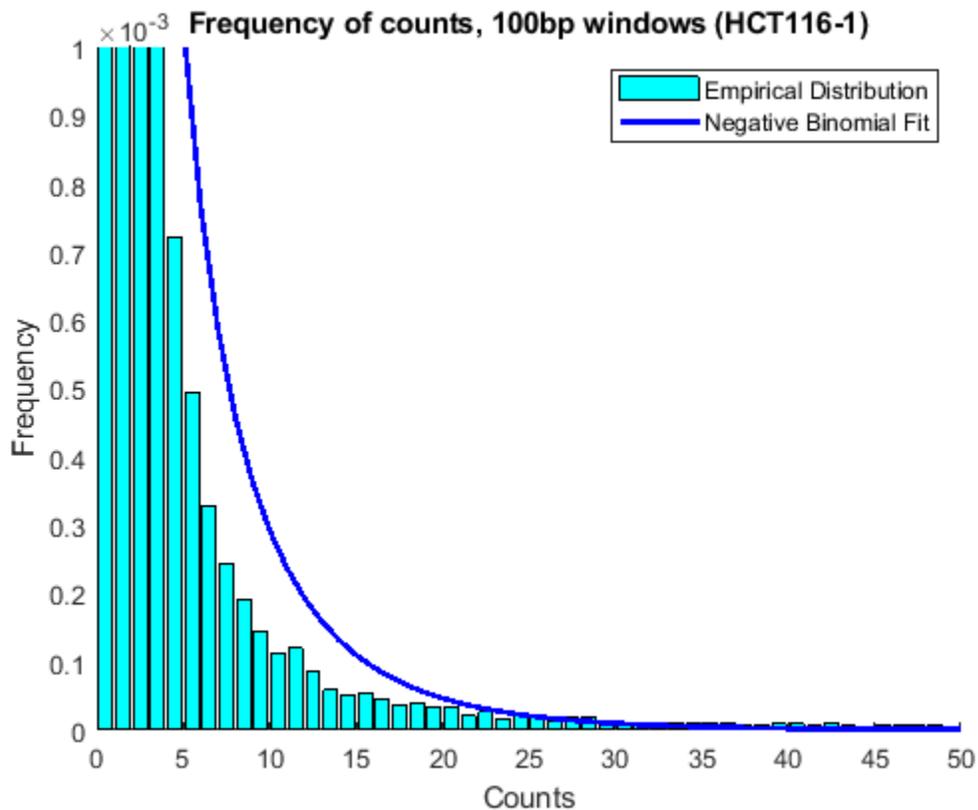
```
nbp = nbinfit(counts_1);
```

Plot the fitted model over a histogram of the empirical data.

```

figure
hold on
edges = 0:100;
emphist = histcounts(counts_1,edges); % calculate the empirical distribution
binCenters = (edges(1:end-1) + edges(2:end)) / 2;
bar(binCenters,emphist./sum(emphist),'c','grouped') % plot histogram
plot(0:100,nbinpdf(0:100,nbp(1),nbp(2)),'b','linewidth',2); % plot fitted model
axis([0 50 0 .001])
legend('Empirical Distribution','Negative Binomial Fit')
ylabel('Frequency')
xlabel('Counts')
title('Frequency of counts, 100bp windows (HCT116-1)')

```



The poor fitting indicates that the observed distribution may be due to the mixture of two models, one that represents the background and one that represents the count data in methylated DNA windows.

A more realistic scenario would be to assume that windows with a small number of mapped reads are mainly the background (or null model). Serre et al. assumed that 100-bp windows containing four or more reads are unlikely to be generated by chance. To estimate a good approximation to the null model, you can fit the left body of the empirical distribution to a truncated negative binomial distribution. To fit a truncated distribution use the `mle` function. First you need to define an anonymous function that defines the right-truncated version of `nbinpdf`.

```
rtnbinpdf = @(x,p1,p2,t) nbinpdf(x,p1,p2) ./ nbincdf(t-1,p1,p2);
```

Define the fitting function using another anonymous function.

```
rtnbinfit = @(x2,t) mle(x2,'pdf',@(x3,p1,p2) rtnbinpdf(x3,p1,p2,t),'start',nbinfit(x2),'lower',[
```

Before fitting the real data, let us assess the fitting procedure with some sampled data from a known distribution.

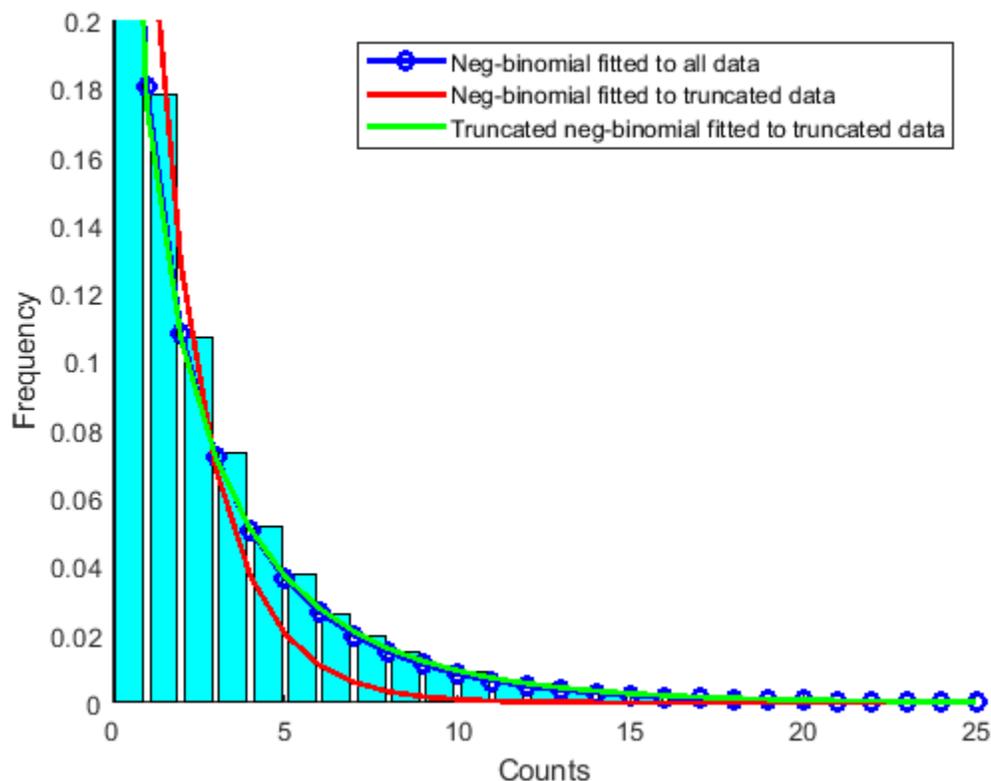
```

nbp = [0.5 0.2]; % Known coefficients
x = nbinrnd(nbp(1),nbp(2),10000,1); % Random sample
trun = 6; % Set a truncation threshold

nbphat1 = nbinfit(x); % Fit non-truncated model to all data
nbphat2 = nbinfit(x(x<trun)); % Fit non-truncated model to truncated data (wrong)
nbphat3 = rtnbinfit(x(x<trun),trun); % Fit truncated model to truncated data

figure
hold on
edges = 0:100;
emphist = histcounts(x,edges); % Calculate the empirical distribution
binCenters = (edges(1:end-1) + edges(2:end)) / 2;
bar(binCenters,emphist./sum(emphist),'c','grouped') % plot histogram
h1 = plot(0:100,nbinpdf(0:100,nbphat1(1),nbphat1(2)),'b-o','linewidth',2);
h2 = plot(0:100,nbinpdf(0:100,nbphat2(1),nbphat2(2)),'r','linewidth',2);
h3 = plot(0:100,rtnbinpdf(0:100,nbphat3(1),nbphat3(2)),'g','linewidth',2);
axis([0 25 0 .2])
legend([h1 h2 h3],'Neg-binomial fitted to all data',...
       'Neg-binomial fitted to truncated data',...
       'Truncated neg-binomial fitted to truncated data')
ylabel('Frequency')
xlabel('Counts')

```



Identifying Significant Methylated Regions

For the two replicates of the HCT116 sample, fit a right-truncated negative binomial distribution to the observed null model using the `rtnbinfit` anonymous function previously defined.

```
trun = 4; % Set a truncation threshold (as in [1])
pn1 = rtnbinfit(counts_1(counts_1<trun),trun); % Fit to HCT116-1 counts
pn2 = rtnbinfit(counts_2(counts_2<trun),trun); % Fit to HCT116-2 counts
```

Calculate the p-value for each window to the null distribution.

```
pval1 = 1 - nbincdf(counts_1,pn1(1),pn1(2));
pval2 = 1 - nbincdf(counts_2,pn2(1),pn2(2));
```

Calculate the false discovery rate using the `mafdr` function. Use the name-value pair `BHFDR` to use the linear-step up (LSU) procedure ([6]) to calculate the FDR adjusted p-values. Setting the `FDR < 0.01` permits you to identify the 100-bp windows that are significantly methylated.

```
fdr1 = mafdr(pval1,'bhfdr',true);
fdr2 = mafdr(pval2,'bhfdr',true);
```

```
w1 = fdr1<.01; % logical vector indicating significant windows in HCT116-1
w2 = fdr2<.01; % logical vector indicating significant windows in HCT116-2
w12 = w1 & w2; % logical vector indicating significant windows in both replicates
```

```
Number_of_sig_windows_HCT116_1 = sum(w1)
Number_of_sig_windows_HCT116_2 = sum(w2)
Number_of_sig_windows_HCT116 = sum(w12)
```

```
Number_of_sig_windows_HCT116_1 =
    1662
```

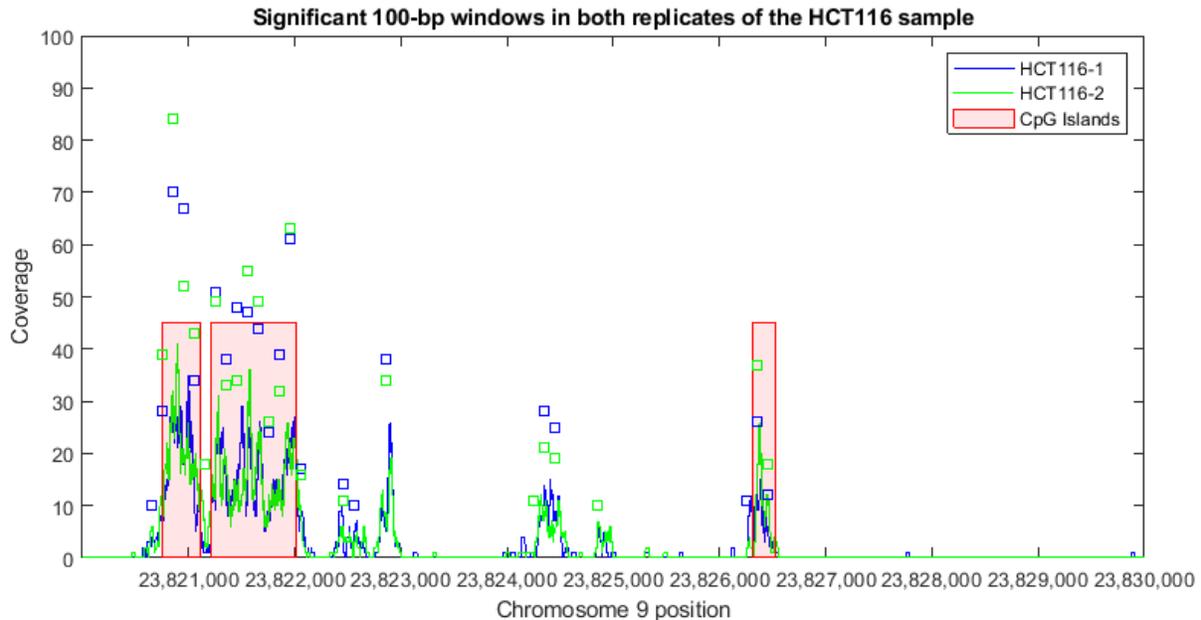
```
Number_of_sig_windows_HCT116_2 =
    1674
```

```
Number_of_sig_windows_HCT116 =
    1346
```

Overall, you identified 1662 and 1674 non-overlapping 100-bp windows in the two replicates of the HCT116 samples, which indicates there is significant evidence of DNA methylation. There are 1346 windows that are significant in both replicates.

For example, looking again in the promoter region of the `ELAVL2` human gene you can observe that in both sample replicates, multiple 100-bp windows have been marked significant.

```
figure(fhELAVL2) % bring back to focus the previously plotted figure
plot(w(w1)+50,counts_1(w1),'bs','HandleVisibility','off') % plot significant windows in HCT116-1
plot(w(w2)+50,counts_2(w2),'gs','HandleVisibility','off') % plot significant windows in HCT116-2
axis([r1 r2 0 100])
title('Significant 100-bp windows in both replicates of the HCT116 sample')
```



Finding Genes With Significant Methylated Promoter Regions

DNA methylation is involved in the modulation of gene expression. For instance, it is well known that hypermethylation is associated with the inactivation of several tumor suppressor genes. You can study in this data set the methylation of gene promoter regions.

First, download from Ensembl a tab-separated-value (TSV) table with all protein encoding genes to a text file, `ensemblmart_genes_hum37.txt`. For this example, we are using Ensembl release 64. Using Ensembl's BioMart service, you can select a table with the following attributes: chromosome name, gene biotype, gene name, gene start/end, and strand direction.

Use the provided helper function `ensemblmart2gff` to convert the downloaded TSV file to a GFF formatted file. Then use `GFFAnnotation` to load the file into MATLAB and create a subset with the genes present in chromosome 9 only. This results 800 annotated protein-coding genes in the Ensembl database.

```
GFFfilename = ensemblmart2gff('ensemblmart_genes_hum37.txt');
a = GFFAnnotation(GFFfilename)
a9 = getSubset(a, 'reference', '9')
numGenes = a9.NumEntries
```

```
a =
```

```
GFFAnnotation with properties:
```

```
FieldNames: {1x9 cell}
NumEntries: 21184
```

```
a9 =
```

```
GFFAnnotation with properties:
```

```
FieldNames: {1x9 cell}
NumEntries: 800
```

```
numGenes =
    800
```

Find the promoter regions for each gene. In this example we consider the proximal promoter as the -500/100 upstream region.

```
downstream = 500;
upstream   = 100;

geneDir = strcmp(a9.Strand, '+'); % logical vector indicating strands in the forward direction

% calculate promoter's start position for genes in the forward direction
promoterStart(geneDir) = a9.Start(geneDir) - downstream;
% calculate promoter's end position for genes in the forward direction
promoterStop(geneDir) = a9.Start(geneDir) + upstream;
% calculate promoter's start position for genes in the reverse direction
promoterStart(~geneDir) = a9.Stop(~geneDir) - upstream;
% calculate promoter's end position for genes in the reverse direction
promoterStop(~geneDir) = a9.Stop(~geneDir) + downstream;
```

Use a dataset as a container for the promoter information, as we can later add new columns to store gene counts and p-values.

```
promoters = dataset({a9.Feature, 'Gene'});
promoters.Strand = char(a9.Strand);
promoters.Start = promoterStart';
promoters.Stop = promoterStop';
```

Find genes with significant DNA methylation in the promoter region by looking at the number of mapped short reads that overlap at least one base pair in the defined promoter region.

```
promoters.Counts_1 = getCounts(bm_hct116_1, promoters.Start, promoters.Stop, ...
    'overlap', 1, 'independent', true);
promoters.Counts_2 = getCounts(bm_hct116_2, promoters.Start, promoters.Stop, ...
    'overlap', 1, 'independent', true);
```

Fit a null distribution for each sample replicate and compute the p-values:

```
trun = 5; % Set a truncation threshold
pn1 = rtnbinfit(promoters.Counts_1(promoters.Counts_1 < trun), trun); % Fit to HCT116-1 promoter counts
pn2 = rtnbinfit(promoters.Counts_2(promoters.Counts_2 < trun), trun); % Fit to HCT116-2 promoter counts
promoters.pval_1 = 1 - nbincdf(promoters.Counts_1, pn1(1), pn1(2)); % p-value for every promoter in HCT116-1
promoters.pval_2 = 1 - nbincdf(promoters.Counts_2, pn2(1), pn2(2)); % p-value for every promoter in HCT116-2
```

```
Number_of_sig_promoters = sum(promoters.pval_1 < .01 & promoters.pval_2 < .01)
```

```
Ratio_of_sig_methylated_promoters = Number_of_sig_promoters./numGenes
```

```
Number_of_sig_promoters =
```

```
74
```

```
Ratio_of_sig_methylated_promoters =
    0.0925
```

Observe that only 74 (out of 800) genes in chromosome 9 have significantly DNA methylated regions (pval<0.01 in both replicates). Display a report of the 30 genes with the most significant methylated promoter regions.

```
[~,order] = sort(promoters.pval_1.*promoters.pval_2);
promoters(order(1:30),[1 2 3 4 5 7 6 8])
```

```
ans =
```

Gene	Strand	Start	Stop	Counts_1
{ 'DMRT3' }	+	976464	977064	223
{ 'CNTFR' }	-	34590021	34590621	219
{ 'GABBR2' }	-	101471379	101471979	404
{ 'CACNA1B' }	+	140771741	140772341	454
{ 'BARX1' }	-	96717554	96718154	264
{ 'FAM78A' }	-	134151834	134152434	497
{ 'FOXB2' }	+	79634071	79634671	163
{ 'TLE4' }	+	82186188	82186788	157
{ 'ASTN2' }	-	120177248	120177848	141
{ 'FOXE1' }	+	100615036	100615636	149
{ 'MPDZ' }	-	13279489	13280089	129
{ 'PTPRD' }	-	10612623	10613223	145
{ 'PALM2-AKAP2' }	+	112542089	112542689	134
{ 'FAM69B' }	+	139606522	139607122	112
{ 'WNK2' }	+	95946698	95947298	108
{ 'IGFBPL1' }	-	38424344	38424944	110
{ 'AKAP2' }	+	112542269	112542869	107
{ 'C9orf4' }	-	111929471	111930071	102
{ 'COL5A1' }	+	137533120	137533720	84
{ 'LHX3' }	-	139096855	139097455	74
{ 'OLFM1' }	+	137966768	137967368	75
{ 'NPR2' }	+	35791651	35792251	68
{ 'DBC1' }	-	122131645	122132245	61
{ 'SOHLH1' }	-	138591274	138591874	56
{ 'PIP5K1B' }	+	71320075	71320675	59
{ 'PRDM12' }	+	133539481	133540081	53
{ 'ELAVL2' }	-	23826235	23826835	50
{ 'ZFP37' }	-	115818939	115819539	59
{ 'RP11-35N6.1' }	+	103790491	103791091	60
{ 'DMRT2' }	+	1049854	1050454	54

pval_1	Counts_2	pval_2
6.6613e-16	253	5.5511e-16
6.6613e-16	226	5.5511e-16
6.6613e-16	400	5.5511e-16
6.6613e-16	408	5.5511e-16
6.6613e-16	286	5.5511e-16
6.6613e-16	499	5.5511e-16
1.4e-13	165	6.0352e-13

3.5649e-13	151	4.7347e-12
4.3566e-12	163	8.0969e-13
1.2447e-12	133	6.7598e-11
2.8679e-11	148	7.3682e-12
2.3279e-12	127	1.6448e-10
1.3068e-11	135	5.0276e-11
4.1911e-10	144	1.3295e-11
7.897e-10	125	2.2131e-10
5.7523e-10	114	1.1364e-09
9.2538e-10	106	3.7513e-09
2.0467e-09	96	1.6795e-08
3.6266e-08	97	1.4452e-08
1.8171e-07	91	3.5644e-08
1.5457e-07	69	1.0074e-06
4.8093e-07	73	5.4629e-07
1.5082e-06	62	2.9575e-06
3.4322e-06	67	1.3692e-06
2.0943e-06	63	2.5345e-06
5.6364e-06	61	3.4518e-06
9.2778e-06	62	2.9575e-06
2.0943e-06	47	3.0746e-05
1.7771e-06	42	6.8037e-05
4.7762e-06	46	3.6016e-05

Finding Intergenic Regions that are Significantly Methylated

Serre et al. [1] reported that, in these data sets, approximately 90% of the uniquely mapped reads fall outside the 5' gene promoter regions. Using a similar approach as before, you can find genes that have intergenic methylated regions. To compensate for the varying lengths of the genes, you can use the maximum coverage, computed base-by-base, instead of the raw number of mapped short reads. Another alternative approach to normalize the counts by the gene length is to set the METHOD name-value pair to rpkm in the getCounts function.

```
intergenic = dataset({a9.Feature, 'Gene'});
intergenic.Strand = char(a9.Strand);
intergenic.Start = a9.Start;
intergenic.Stop = a9.Stop;

intergenic.Counts_1 = getCounts(bm_hct116_1,intergenic.Start,intergenic.Stop,...
    'overlap','full','method','max','independent',true);
intergenic.Counts_2 = getCounts(bm_hct116_2,intergenic.Start,intergenic.Stop,...
    'overlap','full','method','max','independent',true);
trun = 10; % Set a truncation threshold
pn1 = rtnbinfit(intergenic.Counts_1(intergenic.Counts_1<trun),trun); % Fit to HCT116-1 intergenic
pn2 = rtnbinfit(intergenic.Counts_2(intergenic.Counts_2<trun),trun); % Fit to HCT116-2 intergenic
intergenic.pval_1 = 1 - nbincdf(intergenic.Counts_1,pn1(1),pn1(2)); % p-value for every intergenic
intergenic.pval_2 = 1 - nbincdf(intergenic.Counts_2,pn2(1),pn2(2)); % p-value for every intergenic

Number_of_sig_genes = sum(intergenic.pval_1<.01 & intergenic.pval_2<.01)

Ratio_of_sig_methylated_genes = Number_of_sig_genes./numGenes

[~,order] = sort(intergenic.pval_1.*intergenic.pval_2);

intergenic(order(1:30),[1 2 3 4 5 7 6 8])
```

Number_of_sig_genes =

62

Ratio_of_sig_methylated_genes =

0.0775

ans =

Gene	Strand	Start	Stop	Counts_1
{ 'AL772363.1' }	-	140762377	140787022	106
{ 'CACNA1B' }	+	140772241	141019076	106
{ 'SUSD1' }	-	114803065	114937688	88
{ 'C9orf172' }	+	139738867	139741797	99
{ 'NR5A1' }	-	127243516	127269709	86
{ 'BARX1' }	-	96713628	96717654	77
{ 'KCNT1' }	+	138594031	138684992	58
{ 'GABBR2' }	-	101050391	101471479	65
{ 'FOXB2' }	+	79634571	79635869	51
{ 'NDOR1' }	+	140100119	140113813	54
{ 'KIAA1045' }	+	34957484	34984679	50
{ 'ADAMTSL2' }	+	136397286	136440641	55
{ 'PAX5' }	-	36833272	37034476	48
{ 'OLFM1' }	+	137967268	138013025	55
{ 'PBX3' }	+	128508551	128729656	45
{ 'FOXE1' }	+	100615536	100618986	49
{ 'MPDZ' }	-	13105703	13279589	51
{ 'ASTN2' }	-	119187504	120177348	43
{ 'ARRDC1' }	+	140500106	140509812	49
{ 'IGFBPL1' }	-	38408991	38424444	45
{ 'LHX3' }	-	139088096	139096955	44
{ 'PAPPA' }	+	118916083	119164601	44
{ 'CNTFR' }	-	34551430	34590121	41
{ 'DMRT3' }	+	976964	991731	40
{ 'TUSC1' }	-	25676396	25678856	46
{ 'ELAVL2' }	-	23690102	23826335	35
{ 'SMARCA2' }	+	2015342	2193624	36
{ 'GAS1' }	-	89559279	89562104	34
{ 'GRIN1' }	+	140032842	140063207	36
{ 'TLE4' }	+	82186688	82341658	36

pval_1	Counts_2	pval_2
8.6597e-15	98	1.8763e-14
8.6597e-15	98	1.8763e-14
2.2904e-12	112	7.7716e-16
7.4718e-14	96	3.5749e-14
4.268e-12	90	2.5457e-13
7.0112e-11	62	2.569e-09
2.5424e-08	73	6.9019e-11
2.9078e-09	58	9.5469e-09
2.2131e-07	58	9.5469e-09
8.7601e-08	55	2.5525e-08
3.0134e-07	55	2.5525e-08

6.4307e-08	45	6.7163e-07
5.585e-07	49	1.8188e-07
6.4307e-08	42	1.7861e-06
1.4079e-06	51	9.4566e-08
4.1027e-07	46	4.8461e-07
2.2131e-07	42	1.7861e-06
2.6058e-06	43	1.2894e-06
4.1027e-07	36	1.2564e-05
1.4079e-06	39	4.7417e-06
1.9155e-06	36	1.2564e-05
1.9155e-06	35	1.7377e-05
4.8199e-06	37	9.0815e-06
6.5537e-06	37	9.0815e-06
1.0346e-06	31	6.3417e-05
3.0371e-05	41	2.4736e-06
2.2358e-05	40	3.4251e-06
4.1245e-05	41	2.4736e-06
2.2358e-05	38	6.5629e-06
2.2358e-05	37	9.0815e-06

For instance, explore the methylation profile of the *BARX1* gene, the sixth significant gene with intergenic methylation in the previous list. The GTF formatted file `ensemblmart_barx1.gtf` contains structural information for this gene obtained from Ensembl using the BioMart service.

Use `GTFAnnotation` to load the structural information into MATLAB. There are two annotated transcripts for this gene.

```
barx1 = GTFAnnotation('ensemblmart_barx1.gtf')
transcripts = getTranscriptNames(barx1)
```

```
barx1 =
```

```
GTFAnnotation with properties:
```

```
FieldNames: {1×11 cell}
NumEntries: 18
```

```
transcripts =
```

```
2×1 cell array
```

```
{'ENST000000253968'}
{'ENST000000401724'}
```

Plot the DNA methylation profile for both HCT116 sample replicates with base-pair resolution. Overlay the CpG islands and plot the exons for each of the two transcripts along the bottom of the plot.

```
range = barx1.getRange;
r1 = range(1)-1000; % set the region limits
r2 = range(2)+1000;
figure
hold on
% plot high-resolution coverage of bm_hct116_1
```

```

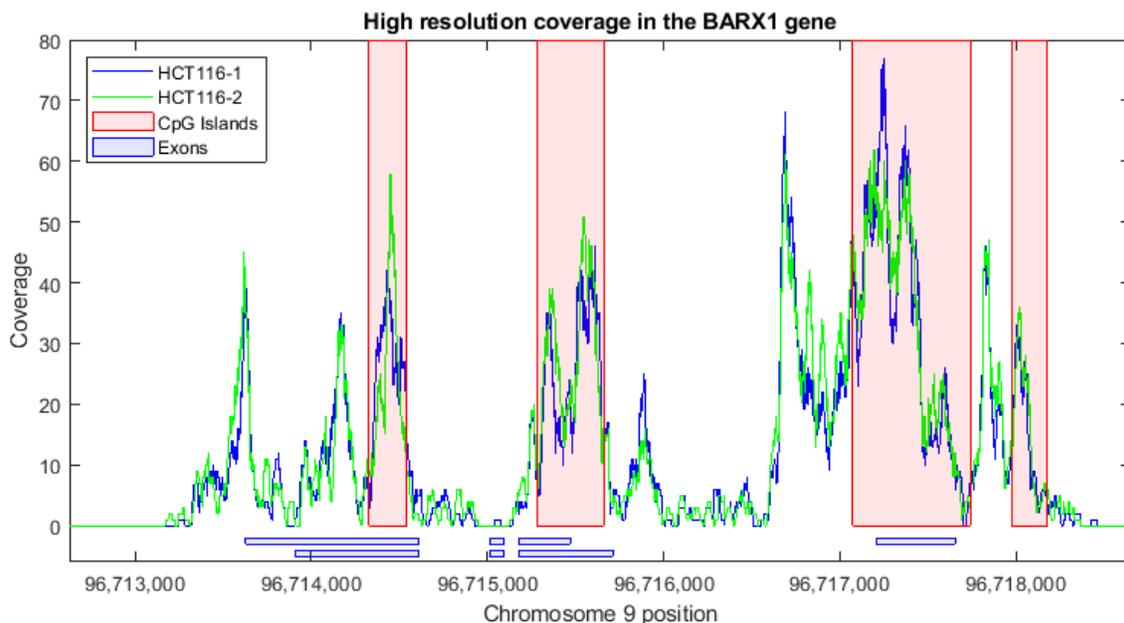
h1 = plot(r1:r2,getBaseCoverage(bm_hct116_1,r1,r2,'binWidth',1),'b');
% plot high-resolution coverage of bm_hct116_2
h2 = plot(r1:r2,getBaseCoverage(bm_hct116_2,r1,r2,'binWidth',1),'g');

% mark the CpG islands within the [r1 r2] region
for i = 1:numel(cpgi.Starts)
    if cpgi.Starts(i)>r1 && cpgi.Stops(i)<r2 % is CpG island inside [r1 r2]?
        px = [cpgi.Starts([i i]) cpgi.Stops([i i])]; % x-coordinates for patch
        py = [0 max(ylim) max(ylim) 0]; % y-coordinates for patch
        hp = patch(px,py,'r','FaceAlpha',.1,'EdgeColor','r','Tag','cpgi');
    end
end

% mark the exons at the bottom of the axes
for i = 1:numel(transcripts)
    exons = getSubset(barx1,'Transcript',transcripts{i},'Feature','exon');
    for j = 1:exons.NumEntries
        px = [exons.Start([j j]);exons.Stop([j j])]; % x-coordinates for patch
        py = [0 1 1 0]-i*2-1; % y-coordinates for patch
        hq = patch(px,py,'b','FaceAlpha',.1,'EdgeColor','b','Tag','exon');
    end
end

axis([r1 r2 -numel(transcripts)*2-2 80]) % zooms-in the y-axis
fixGenomicPositionLabels(gca) % formats tick labels and adds data cursors
ylabel('Coverage')
xlabel('Chromosome 9 position')
title('High resolution coverage in the BARX1 gene')
legend([h1 h2 hp hq], 'HCT116-1', 'HCT116-2', 'CpG Islands', 'Exons', 'Location', 'NorthWest')

```



Observe the highly methylated region in the 5' promoter region (right-most CpG island). Recall that for this gene transcription occurs in the reverse strand. More interesting, observe the highly methylated regions that overlap the initiation of each of the two annotated transcripts (two middle CpG islands).

Differential Analysis of Methylation Patterns

In the study by Serre et al. another cell line is also analyzed. New cells (DICERex5) are derived from the same HCT116 colon cancer cells after truncating the DICER1 alleles. It has been reported in literature [5] that there is a localized change of DNA methylation at small number of gene promoters. In this example, you will find significant 100-bp windows in two sample replicates of the DICERex5 cells following the same approach as the parental HCT116 cells, and then you will search statistically significant differences between the two cell lines.

The helper function `getWindowCounts` captures the similar steps to find windows with significant coverage as before. `getWindowCounts` returns vectors with counts, p-values, and false discovery rates for each new replicate.

```
bm_dicer_1 = BioMap('SRR030222.bam', 'SelectRef', 'gi|224589821|ref|NC_000009.11|');
bm_dicer_2 = BioMap('SRR030223.bam', 'SelectRef', 'gi|224589821|ref|NC_000009.11|');
[counts_3,pval3,fdr3] = getWindowCounts(bm_dicer_1,4,w,100);
[counts_4,pval4,fdr4] = getWindowCounts(bm_dicer_2,4,w,100);
w3 = fdr3<.01; % logical vector indicating significant windows in DICERex5_1
w4 = fdr4<.01; % logical vector indicating significant windows in DICERex5-2
w34 = w3 & w4; % logical vector indicating significant windows in both replicates
Number_of_sig_windows_DICERex5_1 = sum(w3)
Number_of_sig_windows_DICERex5_2 = sum(w4)
Number_of_sig_windows_DICERex5 = sum(w34)
```

```
Number_of_sig_windows_DICERex5_1 =
    908
```

```
Number_of_sig_windows_DICERex5_2 =
    1041
```

```
Number_of_sig_windows_DICERex5 =
    759
```

To perform a differential analysis you use the 100-bp windows that are significant in at least one of the samples (either HCT116 or DICERex5).

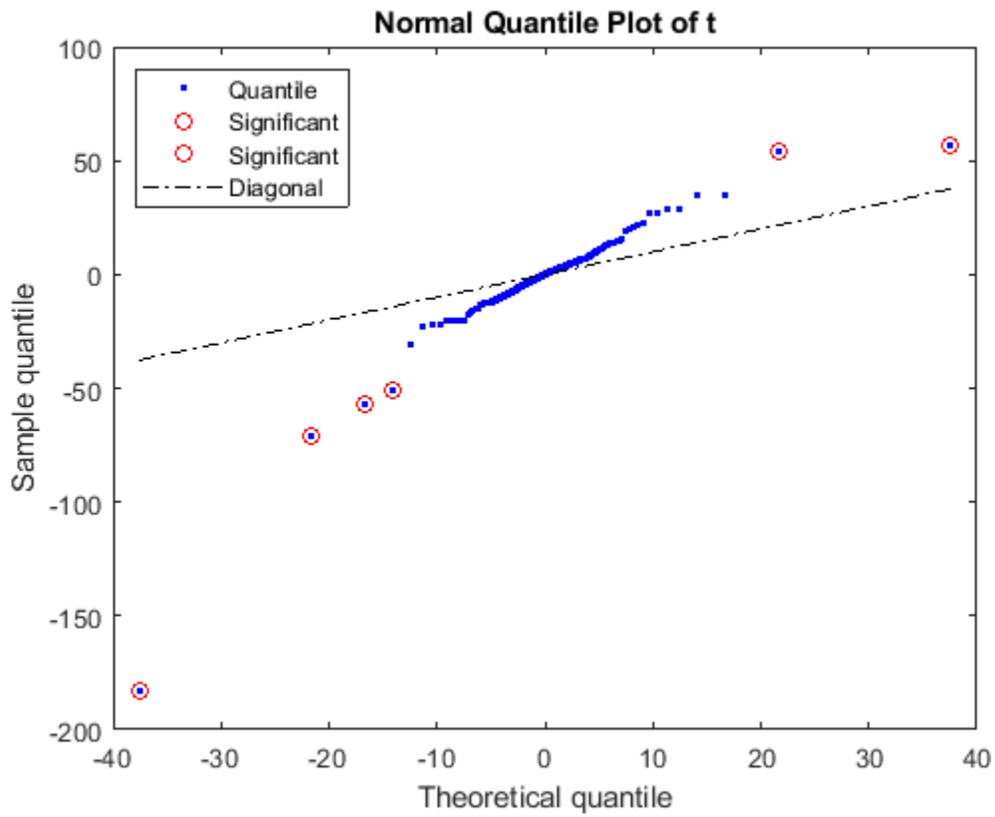
```
wd = w34 | w12; % logical vector indicating windows included in the diff. analysis
counts = [counts_1(wd) counts_2(wd) counts_3(wd) counts_4(wd)];
ws = w(wd); % window start for each row in counts
```

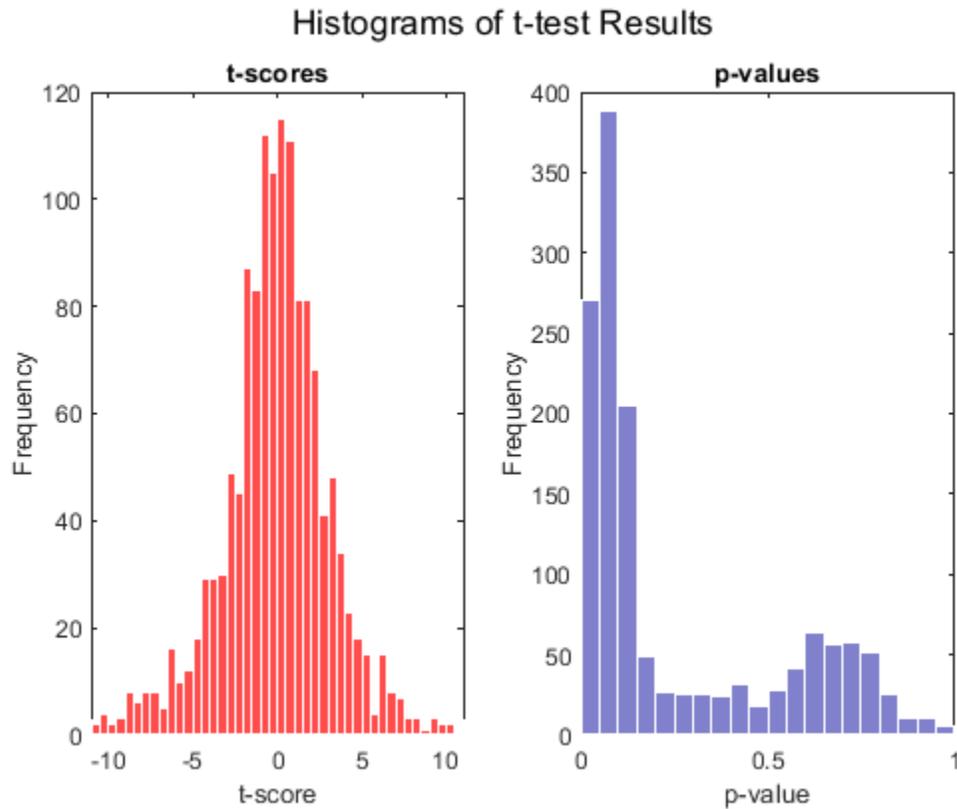
Use the function `manorm` to normalize the data. The `PERCENTILE` name-value pair lets you filter out windows with very large number of counts while normalizing, since these windows are mainly due to artifacts, such as repetitive regions in the reference chromosome.

```
counts_norm = round(manorm(counts, 'percentile', 90).*100);
```

Use the function `mattest` to perform a two-sample t-test to identify differentially covered windows from the two different cell lines.

```
pval = mattest(counts_norm(:,[1 2]),counts_norm(:,[3 4]),'bootstrap',true,...  
  'showhist',true,'showplot',true);
```





Create a report with the 25 most significant differentially covered windows. While creating the report use the helper function `findClosestGene` to determine if the window is intergenic, intragenic, or if it is in a proximal promoter region.

```
[~,ord] = sort(pval);
fprintf('Window Pos      Type                p-value   HCT116   DICERex5\n\n');
for i = 1:25
    j = ord(i);
    [~,msg] = findClosestGene(a9,[ws(j) ws(j)+99]);
    fprintf('%10d  %-25s %7.6f%5d%5d %5d%5d\n', ...
        ws(j),msg,pval(j),counts_norm(j,:));
end
```

Window Pos	Type	p-value	HCT116	DICERex5
140311701	Intergenic (EXD3)	0.000020	13 13	104 105
139546501	Intragenic	0.001525	21 21	91 93
10901	Intragenic	0.002222	258 257	434 428
120176801	Intergenic (ASTN2)	0.002270	266 270	155 155
139914801	Intergenic (ABCA2)	0.002482	64 63	26 25
126128501	Intergenic (CRB2)	0.002664	94 93	129 130
71939501	Prox. Promoter (FAM189A2)	0.004687	107 101	0 0
124461001	Intergenic (DAB2IP)	0.004747	77 76	39 37
140086501	Intergenic (TPRN)	0.005489	47 42	123 124
79637201	Intragenic	0.006323	52 51	32 31
136470801	Intragenic	0.006323	52 51	32 31
140918001	Intergenic (CACNA1B)	0.006786	176 169	71 68
100615901	Intergenic (FOX E1)	0.006969	262 253	123 118

98221901	Intergenic (PTCH1)	0.008291	26	30	104	99
138730601	Intergenic (CAMSAP1)	0.008602	26	21	97	93
89561701	Intergenic (GAS1)	0.008653	77	76	6	12
977401	Intergenic (DMRT3)	0.008678	236	245	129	124
37002601	Intergenic (PAX5)	0.008822	133	127	207	211
139744401	Intergenic (PHPT1)	0.009078	47	46	32	31
126771301	Intragenic	0.009547	43	46	97	93
93922501	Intragenic	0.009565	34	34	149	161
94187101	Intragenic	0.009579	73	80	6	6
136044401	Intragenic	0.009623	39	34	110	105
139611201	Intergenic (FAM69B)	0.009623	39	34	110	105
139716201	Intergenic (C9orf86)	0.009860	73	72	136	130

Plot the DNA methylation profile for the promoter region of gene FAM189A2, the most significant differentially covered promoter region from the previous list. Overlay the CpG islands and the FAM189A2 gene.

```

range = getRange(getSubset(a9, 'Feature', 'FAM189A2'));
r1 = range(1)-1000;
r2 = range(2)+1000;
figure
hold on

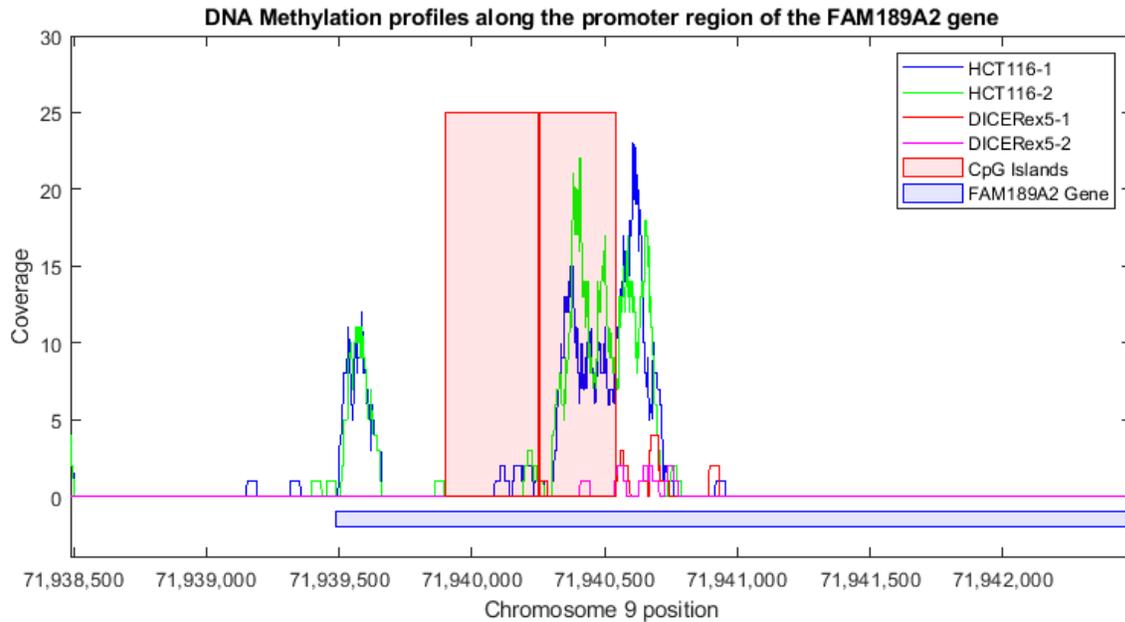
% plot high-resolution coverage of all replicates
h1 = plot(r1:r2,getBaseCoverage(bm_hct116_1,r1,r2,'binWidth',1),'b');
h2 = plot(r1:r2,getBaseCoverage(bm_hct116_2,r1,r2,'binWidth',1),'g');
h3 = plot(r1:r2,getBaseCoverage(bm_dicer_1,r1,r2,'binWidth',1),'r');
h4 = plot(r1:r2,getBaseCoverage(bm_dicer_2,r1,r2,'binWidth',1),'m');

% mark the CpG islands within the [r1 r2] region
for i = 1:numel(cpgi.Starts)
    if cpgi.Starts(i)>r1 && cpgi.Stops(i)<r2 % is CpG island inside [r1 r2]?
        px = [cpgi.Starts([i i]) cpgi.Stops([i i])]; % x-coordinates for patch
        py = [0 max(ylim) max(ylim) 0]; % y-coordinates for patch
        hp = patch(px,py,'r','FaceAlpha',.1,'EdgeColor','r','Tag','cpgi');
    end
end

% mark the gene at the bottom of the axes
px = range([1 1 2 2]);
py = [0 1 1 0]-2;
hq = patch(px,py,'b','FaceAlpha',.1,'EdgeColor','b','Tag','gene');

axis([r1 r1+4000 -4 30]) % zooms-in
fixGenomicPositionLabels(gca) % formats tick labels and adds datacursors
ylabel('Coverage')
xlabel('Chromosome 9 position')
title('DNA Methylation profiles along the promoter region of the FAM189A2 gene')
legend([h1 h2 h3 h4 hp hq],...
    'HCT116-1','HCT116-2','DICERex5-1','DICERex5-2','CpG Islands','FAM189A2 Gene',...
    'Location','NorthEast')

```



Observe that the CpG islands are clearly unmethylated for both of the DICERex5 replicates.

References

- [1] Serre, D., Lee, B.H., and Ting A.H., "MBD-isolated Genome Sequencing provides a high-throughput and comprehensive survey of DNA methylation in the human genome", *Nucleic Acids Research*, 38(2):391-9, 2010.
- [2] Langmead, B., Trapnell, C., Pop, M., and Salzberg, S.L., "Ultrafast and Memory-efficient Alignment of Short DNA Sequences to the Human Genome", *Genome Biology*, 10(3):R25, 2009.
- [3] Li, H., et al., "The Sequence Alignment/map (SAM) Format and SAMtools", *Bioinformatics*, 25(16):2078-9, 2009.
- [4] Gardiner-Garden, M. and Frommer, M., "CpG islands in vertebrate genomes", *Journal of Molecular Biology*, 196(2):261-82, 1987.
- [5] Ting, A.H., et al., "A Requirement for DICER to Maintain Full Promoter CpG Island Hypermethylation in Human Cancer Cells", *Cancer Research*, 68(8):2570-5, 2008.
- [6] Benjamini, Y. and Hochberg, Y., "Controlling the false discovery rate: a practical and powerful approach to multiple testing", *Journal of the Royal Statistical Society*, 57(1):289-300, 1995.

Exploring Protein-DNA Binding Sites from Paired-End ChIP-Seq Data

This example shows how to perform a genome-wide analysis of a transcription factor in the *Arabidopsis Thaliana* (Thale Cress) model organism.

For enhanced performance, it is recommended that you run this example on a 64-bit platform, because the memory footprint is close to 2 Gb. On a 32-bit platform, if you receive "Out of memory" errors when running this example, try increasing the virtual memory (or swap space) of your operating system or try setting the 3GB switch (32-bit Windows® XP only). For details, see "Resolve "Out of Memory" Errors".

Introduction

ChIP-Seq is a technology that is used to identify transcription factors that interact with specific DNA sites. First chromatin immunoprecipitation enriches DNA-protein complexes using an antibody that binds to a particular protein of interest. Then, all the resulting fragments are processed using high-throughput sequencing. Sequencing fragments are mapped back to the reference genome. By inspecting over-represented regions it is possible to mark the genomic location of DNA-protein interactions.

In this example, short reads are produced by the paired-end Illumina® platform. Each fragment is reconstructed from two short reads successfully mapped, with this the exact length of the fragment can be computed. Using paired-end information from sequence reads maximizes the accuracy of predicting DNA-protein binding sites.

Data Set

This example explores the paired-end ChIP-Seq data generated by Wang *et.al.* [1] using the Illumina® platform. The data set has been courteously submitted to the Gene Expression Omnibus repository with accession number GSM424618. The unmapped paired-end reads can be obtained from the NCBI FTP site.

This example assumes that you:

- (1) downloaded the data containing the unmapped short read and converted it to FASTQ formatted files using the NCBI SRA Toolkit.
- (2) produced a SAM formatted file by mapping the short reads to the Thale Cress reference genome, using a mapper such as BWA [2], Bowtie, or SSAHA2 (which is the mapper used by authors of [1]), and,
- (3) ordered the SAM formatted file by reference name first, then by genomic position.

For the published version of this example, 8,655,859 paired-end short reads are mapped using the BWA mapper [2]. BWA produced a SAM formatted file (`aratha.sam`) with 17,311,718 records (8,655,859 x 2). Repetitive hits were randomly chosen, and only one hit is reported, but with lower mapping quality. The SAM file was ordered and converted to a BAM formatted file using SAMtools [3] before being loaded into MATLAB.

The last part of the example also assumes that you downloaded the reference genome for the Thale Cress model organism (which includes five chromosomes). Uncomment the following lines of code to download the reference from the NCBI repository:

```
% getgenbank('NC_003070','FileFormat','fasta','tofile','ach1.fasta');
% getgenbank('NC_003071','FileFormat','fasta','tofile','ach2.fasta');
% getgenbank('NC_003074','FileFormat','fasta','tofile','ach3.fasta');
% getgenbank('NC_003075','FileFormat','fasta','tofile','ach4.fasta');
% getgenbank('NC_003076','FileFormat','fasta','tofile','ach5.fasta');
```

Creating a MATLAB® Interface to a BAM Formatted File

To create local alignments and look at the coverage we need to construct a **BioMap**. **BioMap** has an interface that provides direct access to the mapped short reads stored in the BAM formatted file, thus minimizing the amount of data that is actually loaded to the workspace. Create a **BioMap** to access all the short reads mapped in the BAM formatted file.

```
bm = BioMap('aratha.bam')
```

```
bm =
```

```
BioMap with properties:
```

```
SequenceDictionary: {5x1 cell}
    Reference: [14637324x1 File indexed property]
    Signature: [14637324x1 File indexed property]
    Start: [14637324x1 File indexed property]
MappingQuality: [14637324x1 File indexed property]
    Flag: [14637324x1 File indexed property]
    MatePosition: [14637324x1 File indexed property]
    Quality: [14637324x1 File indexed property]
    Sequence: [14637324x1 File indexed property]
    Header: [14637324x1 File indexed property]
    NSeqs: 14637324
    Name: ''
```

Use the `getSummary` method to obtain a list of the existing references and the actual number of short read mapped to each one.

```
getSummary(bm)
```

```
BioMap summary:
```

```

                                Name: ''
                                Container_Type: 'Data is file indexed.'
                                Total_Number_of_Sequences: 14637324
                                Number_of_References_in_Dictionary: 5

    Number_of_Sequences    Genomic_Range
Chr1    3151847            1    30427671
Chr2    3080417           1000  19698292
Chr3    3062917            94    23459782
Chr4    2218868           1029  18585050
Chr5    3123275            11    26975502
```

The remainder of this example focuses on the analysis of one of the five chromosomes, **Chr1**. Create a new **BioMap** to access the short reads mapped to the first chromosome by subsetting the first one.

```
bm1 = getSubset(bm,'SelectReference','Chr1')
```

```
bm1 =  
  
BioMap with properties:  
  
SequenceDictionary: 'Chr1'  
    Reference: [3151847x1 File indexed property]  
    Signature: [3151847x1 File indexed property]  
    Start: [3151847x1 File indexed property]  
MappingQuality: [3151847x1 File indexed property]  
    Flag: [3151847x1 File indexed property]  
MatePosition: [3151847x1 File indexed property]  
    Quality: [3151847x1 File indexed property]  
    Sequence: [3151847x1 File indexed property]  
    Header: [3151847x1 File indexed property]  
    NSeqs: 3151847  
    Name: ''
```

By accessing the Start and Stop positions of the mapped short read you can obtain the genomic range.

```
x1 = min(getStart(bm1))  
x2 = max(getStop(bm1))
```

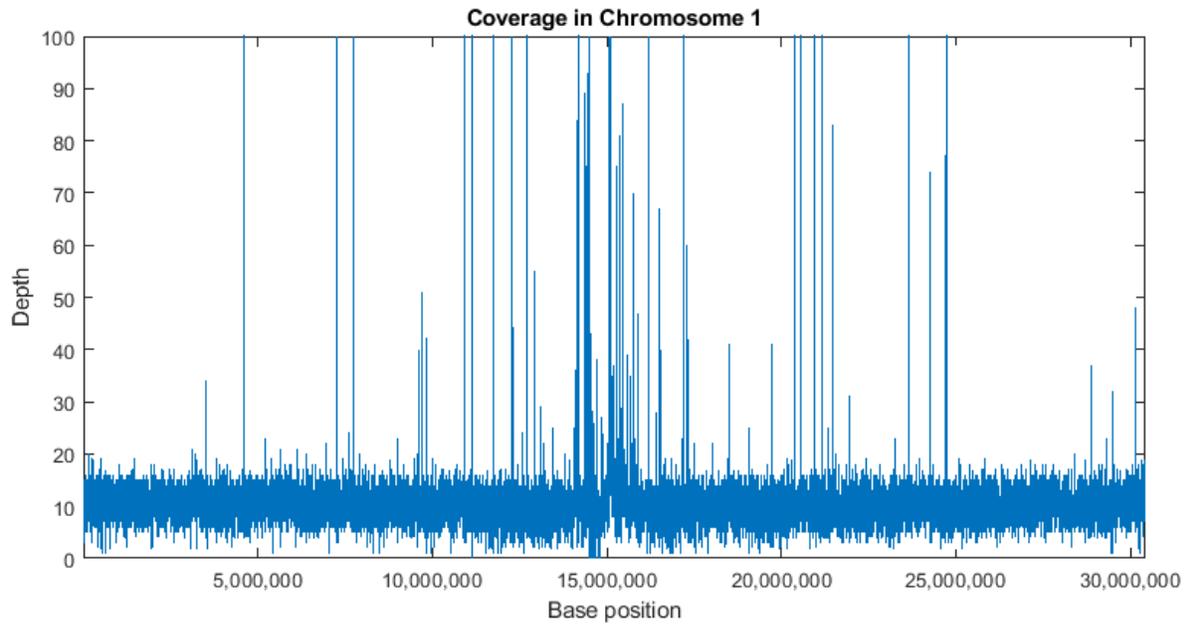
```
x1 =  
  
uint32  
  
1  
  
x2 =  
  
uint32  
  
30427671
```

Exploring the Coverage at Different Resolutions

To explore the coverage for the whole range of the chromosome, a binning algorithm is required. The `getBaseCoverage` method produces a coverage signal based on effective alignments. It also allows you to specify a bin width to control the size (or resolution) of the output signal. However internal computations are still performed at the base pair (bp) resolution. This means that despite setting a large bin size, narrow peaks in the coverage signal can still be observed. Once the coverage signal is plotted you can program the figure's data cursor to display the genomic position when using the tooltip. You can zoom and pan the figure to determine the position and height of the ChIP-Seq peaks.

```
[cov,bin] = getBaseCoverage(bm1,x1,x2,'binWidth',1000,'binType','max');  
figure  
plot(bin,cov)  
axis([x1,x2,0,100])           % sets the axis limits  
fixGenomicPositionLabels     % formats tick labels and adds data cursors  
xlabel('Base position')
```

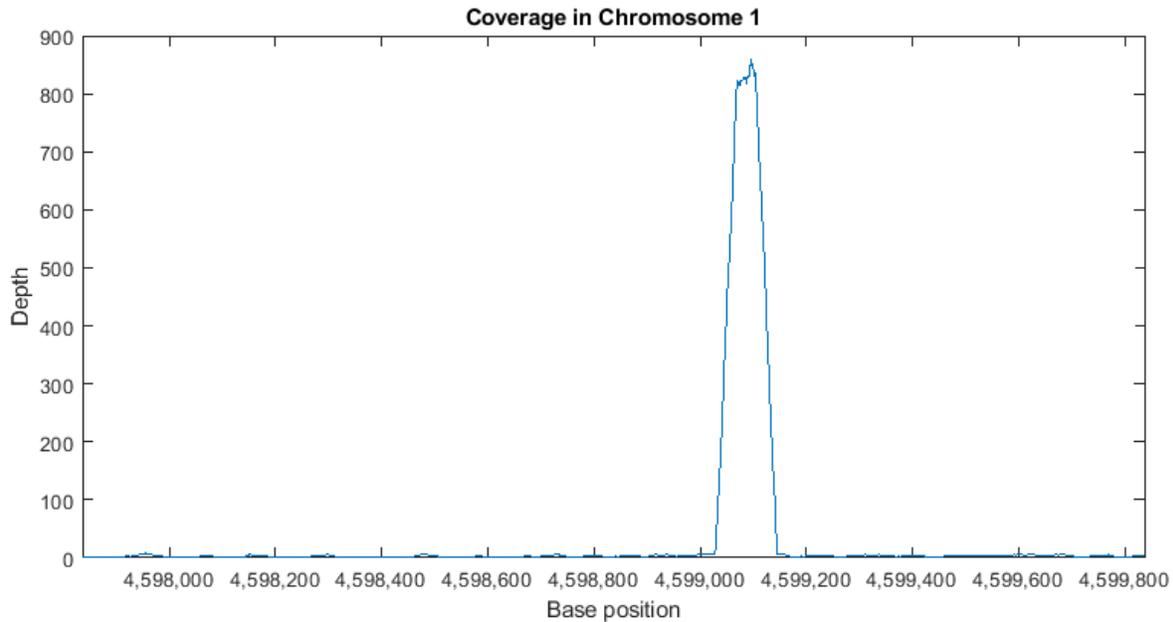
```
ylabel('Depth')
title('Coverage in Chromosome 1')
```



It is also possible to explore the coverage signal at the bp resolution (also referred to as the *pile-up* profile). Explore one of the large peaks observed in the data at position 4598837.

```
p1 = 4598837-1000;
p2 = 4598837+1000;
```

```
figure
plot(p1:p2,getBaseCoverage(bm1,p1,p2))
xlim([p1,p2])           % sets the x-axis limits
fixGenomicPositionLabels % formats tick labels and adds data cursors
xlabel('Base position')
ylabel('Depth')
title('Coverage in Chromosome 1')
```



Identifying and Filtering Regions with Artifacts

Observe the large peak with coverage depth of 800+ between positions 4599029 and 4599145. Investigate how these reads are aligning to the reference chromosome. You can retrieve a subset of these reads enough to satisfy a coverage depth of 25, since this is sufficient to understand what is happening in this region. Use `getIndex` to obtain indices to this subset. Then use `getCompactAlignment` to display the corresponding multiple alignment of the short-reads.

```
i = getIndex(bm1,4599029,4599145,'depth',25);
bmX = getSubset(bm1,i,'inmemory',false)
getCompactAlignment(bmX,4599029,4599145)
```

```
bmX =
```

```
BioMap with properties:
```

```
SequenceDictionary: 'Chr1'
Reference: [62x1 File indexed property]
Signature: [62x1 File indexed property]
Start: [62x1 File indexed property]
MappingQuality: [62x1 File indexed property]
Flag: [62x1 File indexed property]
MatePosition: [62x1 File indexed property]
Quality: [62x1 File indexed property]
Sequence: [62x1 File indexed property]
Header: [62x1 File indexed property]
NSeqs: 62
Name: ''
```

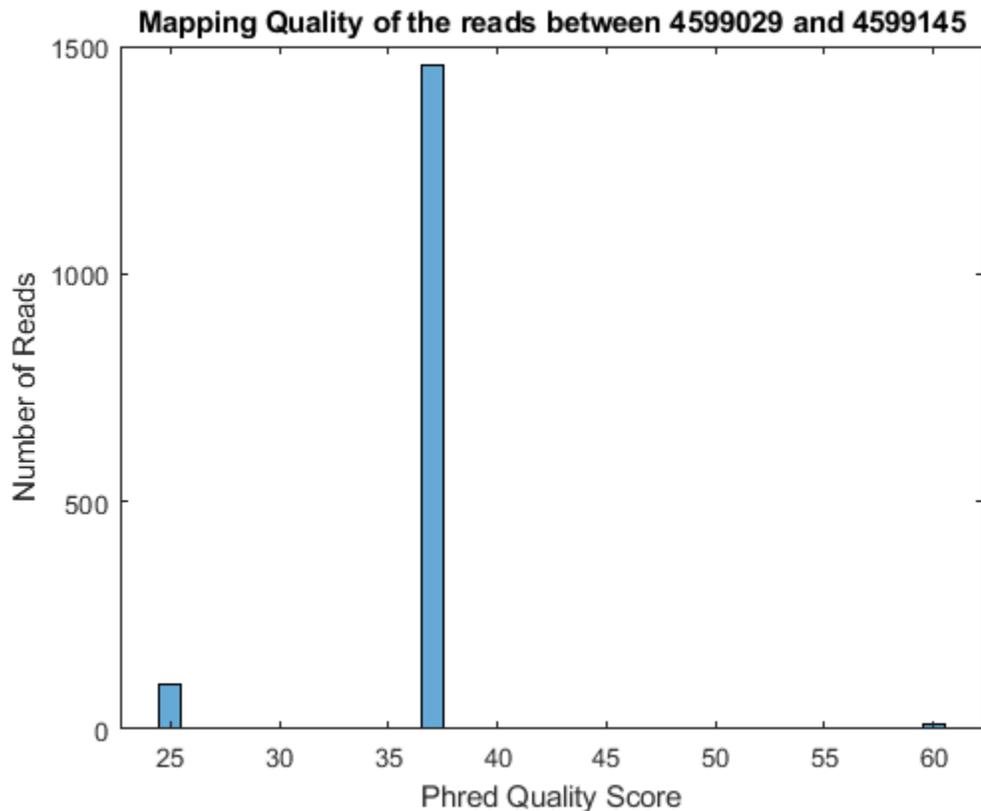
```
ans =
```

35×117 char array

```
'AGTT AATCAAATAGAAAGCCCCGAGGGCGCCATATCTAGGCGC  AACTATGTGATTGAATAAAATCCTCCTCTATCTGTTGCGG  GA
'AGTGC TCAAATAGAAAGCCCCGAGGGCGCCATATTCTAGGAGCCC  GAATAAAATCCTCCTCTATCTGTTGCGGGTCA
'AGTTCAA  CCGAGGGCGCCATATTCTAGGAGCCCAAATATGTGATT  TATCTGTTGCGGGTCA
'AGTTCAATCAAATAGAAAGC  TTCTAGGAGCCCAAATATGTGATTGAATAAAATCCTCCTC
'AGTT  AAGGAGCCCAAATATGTGATTGAATAAAATCCACCTCTAT
'AGTACAATCAAATAGAAAGCCCCGAGGGCGCCATA  TAGGAGCCCAAATATGTGATTGAATAAAATCCTCCTCTAT
'CGTACAATCAAATAGAAAGCCCCGAGGGCGCCATATTC  GGAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCT
'CGTACAATCAAATAGAAAGCCCCGAGGGCGCCATATTC  GGAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCT
'CGTACAATCAAATAGAAAGCCCCGAGGGCGCCATATTC  GGAGCCCAAAGCTATGTGATTGAATAAAATCCTCCTCTATCT
'CGTACAATCAAATAGAAAGCCCCGAGGGCGCCATATTC  GGAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCT
'AGTTCAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTA  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'GATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTA  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'GATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTA  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'GATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTA  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG  CCAAATATGTGATTGAATAAAATCCTCCTCTATCTGTTG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG  CACAAATATGTGATTGAATAAAATCCTCCTCTATCTGTTG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG  CCAAATATGTGATTGAATAAAATCCTCCTCTATCTGTTG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTCG
'  ATACAATCAAATAGAAAGCCCCGGGGGCGCCATATTCTAG
'  ATTGAGTCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG
'  ATACAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAG
'  CAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAGGAG
'  CAATCAAATAGAAAGCCCCGAGGGCGCCATATTCTAGGAG
'  TAGGAGCCCAAATATGTGATTGAATAAAATCCTCCTCTAT
'  TAGGAGCCCAAATATGCCATTGAATAAAATCCTCCGCTAT
'  GGAGCCCAAAGCTATGTGATTGAATAAAATCCTCCTCTATCT
'  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
'  GAGCCCAAATATGTGATTGAATAAAATCCTCCTCTATCTG
```

In addition to visually confirming the alignment, you can also explore the mapping quality for all the short reads in this region, as this may hint to a potential problem. In this case, less than one percent of the short reads have a Phred quality of 60, indicating that the mapper most likely found multiple hits within the reference genome, hence assigning a lower mapping quality.

```
figure
i = getIndex(bm1,4599029,4599145);
histogram(double(getMappingQuality(bm1,i)))
title('Mapping Quality of the reads between 4599029 and 4599145')
xlabel('Phred Quality Score')
ylabel('Number of Reads')
```



Most of the large peaks in this data set occur due to satellite repeat regions or due to its closeness to the centromere [4], and show characteristics similar to the example just explored. You may explore other regions with large peaks using the same procedure.

To prevent these problematic regions, two techniques are used. First, given that the provided data set uses paired-end sequencing, by removing the reads that are not aligned in a proper pair reduces the number of potential aligner errors or ambiguities. You can achieve this by exploring the `flag` field of the SAM formatted file, in which the second less significant bit is used to indicate if the short read is mapped in a proper pair.

```
i = find(bitget(getFlag(bm1),2));
bm1_filtered = getSubset(bm1,i)
```

```
bm1_filtered =
```

```
BioMap with properties:
```

```
SequenceDictionary: 'Chr1'
  Reference: [3040724x1 File indexed property]
  Signature: [3040724x1 File indexed property]
  Start: [3040724x1 File indexed property]
MappingQuality: [3040724x1 File indexed property]
  Flag: [3040724x1 File indexed property]
MatePosition: [3040724x1 File indexed property]
  Quality: [3040724x1 File indexed property]
  Sequence: [3040724x1 File indexed property]
```

```
Header: [3040724x1 File indexed property]
NSeqs: 3040724
Name: ''
```

Second, consider only uniquely mapped reads. You can detect reads that are equally mapped to different regions of the reference sequence by looking at the mapping quality, because BWA assigns a lower mapping quality (less than 60) to this type of short read.

```
i = find(getMappingQuality(bm1_filtered)==60);
bm1_filtered = getSubset(bm1_filtered,i)
```

```
bm1_filtered =
```

```
BioMap with properties:
```

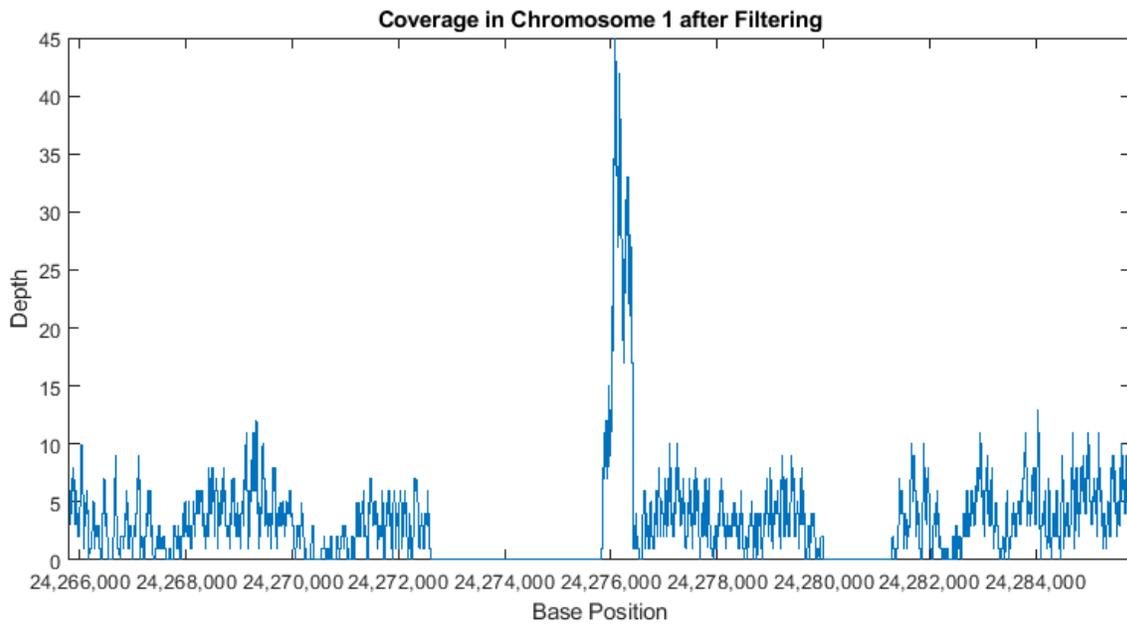
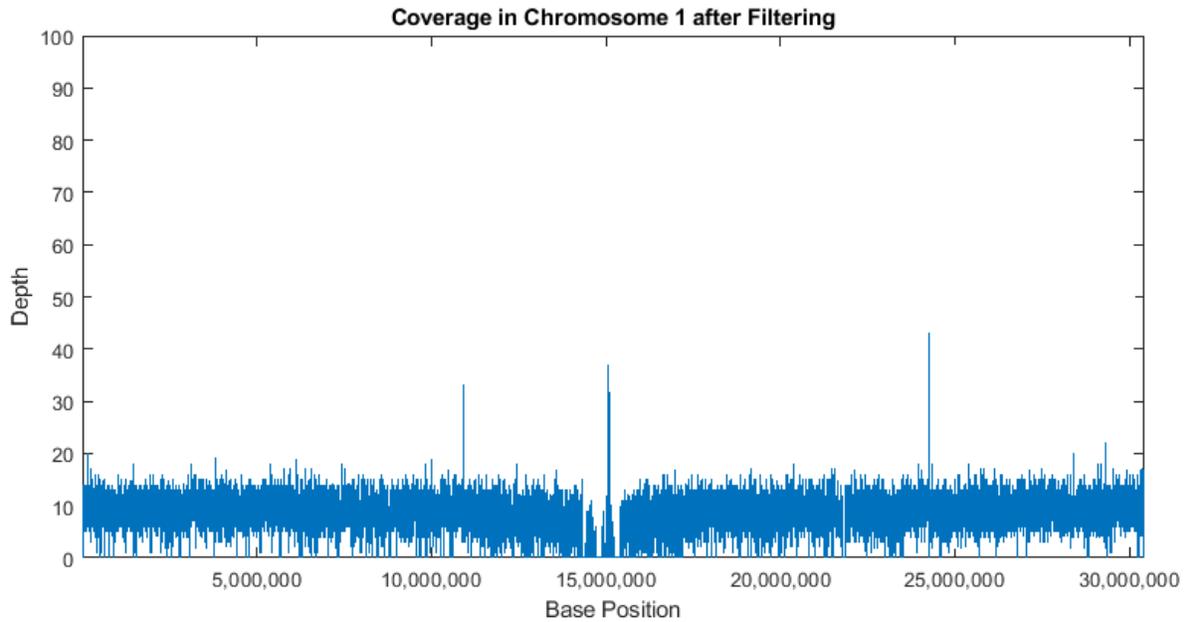
```
SequenceDictionary: 'Chr1'
Reference: [2313252x1 File indexed property]
Signature: [2313252x1 File indexed property]
Start: [2313252x1 File indexed property]
MappingQuality: [2313252x1 File indexed property]
Flag: [2313252x1 File indexed property]
MatePosition: [2313252x1 File indexed property]
Quality: [2313252x1 File indexed property]
Sequence: [2313252x1 File indexed property]
Header: [2313252x1 File indexed property]
NSeqs: 2313252
Name: ''
```

Visualize again the filtered data set using both, a coarse resolution with 1000 bp bins for the whole chromosome, and a fine resolution for a small region of 20,000 bp. Most of the large peaks due to artifacts have been removed.

```
[cov,bin] = getBaseCoverage(bm1_filtered,x1,x2,'binWidth',1000,'binType','max');
figure
plot(bin,cov)
axis([x1,x2,0,100]) % sets the axis limits
fixGenomicPositionLabels % formats tick labels and adds datacursors
xlabel('Base Position')
ylabel('Depth')
title('Coverage in Chromosome 1 after Filtering')
```

```
p1 = 24275801-10000;
p2 = 24275801+10000;
```

```
figure
plot(p1:p2,getBaseCoverage(bm1_filtered,p1,p2))
xlim([p1,p2]) % sets the x-axis limits
fixGenomicPositionLabels % formats tick labels and adds datacursors
xlabel('Base Position')
ylabel('Depth')
title('Coverage in Chromosome 1 after Filtering')
```



Recovering Sequencing Fragments from the Paired-End Reads

In Wang's paper [1] it is hypothesized that paired-end sequencing data has the potential to increase the accuracy of the identification of chromosome binding sites of DNA associated proteins because the fragment length can be derived accurately, while when using single-end sequencing it is necessary to resort to a statistical approximation of the fragment length, and use it indistinctly for all putative binding sites.

Use the paired-end reads to reconstruct the sequencing fragments. First, get the indices for the forward and the reverse reads in each pair. This information is captured in the fifth bit of the `flag` field, according to the SAM file format.

```
fow_idx = find(~bitget(getFlag(bm1_filtered),5));
rev_idx = find(bitget(getFlag(bm1_filtered),5));
```

SAM-formatted files use the same header strings to identify pair mates. By pairing the header strings you can determine how the short reads in `BioMap` are paired. To pair the header strings, simply order them in ascending order and use the sorting indices (`hf` and `hr`) to link the unsorted header strings.

```
[~,hf] = sort(getHeader(bm1_filtered,fow_idx));
[~,hr] = sort(getHeader(bm1_filtered,rev_idx));
mate_idx = zeros(numel(fow_idx),1);
mate_idx(hf) = rev_idx(hr);
```

Use the resulting `fow_idx` and `mate_idx` variables to retrieve pair mates. For example, retrieve the paired-end reads for the first 10 fragments.

```
for j = 1:10
    disp(getInfo(bm1_filtered, fow_idx(j)))
    disp(getInfo(bm1_filtered, mate_idx(j)))
end
```

SRR054715.sra.6849385	163	20	60	40M	AACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAA	BF
SRR054715.sra.6849385	83	229	60	40M	CCTATTTCTTGTTTCTTTCTTCACTTAGCTATGGA	00
SRR054715.sra.6992346	99	20	60	40M	AACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAA	=B
SRR054715.sra.6992346	147	239	60	40M	GTGGTTTTCTTTCTTCACTTAGCTATGGATGGTTTATCT	F
SRR054715.sra.8438570	163	47	60	40M	CTAAATCCCTAAATCTTAAATCCTACATCCATGAATCCC	BF
SRR054715.sra.8438570	83	274	60	40M	TATCTTCATTTGTTATATTGGATACAAGCTTTGCTACGAT	BF
SRR054715.sra.1676744	163	67	60	40M	ATCCTACATCCATGAATCCCTAAATACCTAATCCCCTAAA	BF
SRR054715.sra.1676744	83	283	60	40M	TTGTTATATTGGATACAAGCTTTGCTACGATCTACATTTG	CO
SRR054715.sra.6820328	163	73	60	40M	CATCCATGAATCCCTAAATACCTAATCCCTAAACCCGAA	BF
SRR054715.sra.6820328	83	267	60	40M	GTTGGTGTATCTTCATTTGTTATATTGGATACGAGCTTTG	BF
SRR054715.sra.1559757	163	103	60	40M	TAAACCCGAAACCGGTTTCTCTGGTTGAAACTCATTGTGT	F
SRR054715.sra.1559757	83	311	60	40M	GATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTA	<
SRR054715.sra.5658991	163	103	60	40M	CAAACCCGAAACCGGTTTCTCTGGTTGAAACTCATTGTGT	7
SRR054715.sra.5658991	83	311	60	40M	GATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTA	3
SRR054715.sra.4625439	163	143	60	40M	ATATAATGATAATTTTAGCGTTTTTATGCAATTGCTTATT	F
SRR054715.sra.4625439	83	347	60	40M	CTTAGTGTTGGTTTATCTCAAGAATCTTATTAATTGTTTG	+L
SRR054715.sra.1007474	163	210	60	40M	ATTTGAGGTCAATACAAATCCTATTTCTTGTTGGTTTGCTT	F
SRR054715.sra.1007474	83	408	60	40M	TATTGTCATTCTTACTCCTTTGTGGAAATGTTTGTCTAT	BF
SRR054715.sra.7345693	99	213	60	40M	TGAGGTCAATACAAATCCTATTTCTTGTTGGTTTCTTTCT	B:
SRR054715.sra.7345693	147	393	60	40M	TTATTTTTGGACATTTATTGTCACTTACTCCTTTGGGG	F

Use the paired-end indices to construct a new `BioMap` with the minimal information needed to represent the sequencing fragments. First, calculate the insert sizes.

```
J = getStop(bm1_filtered, fow_idx);
K = getStart(bm1_filtered, mate_idx);
L = K - J - 1;
```

Obtain the new signature (or CIGAR string) for each fragment by using the short read original signatures separated by the appropriate number of skip CIGAR symbols (`N`).

```
n = numel(L);
cigars = cell(n,1);
```

```

for i = 1:n
    cigars{i} = sprintf('%dN' ,L(i));
end
cigars = strcat( getSignature(bm1_filtered, fow_idx),...
                cigars,...
                getSignature(bm1_filtered, mate_idx));

```

Reconstruct the sequences for the fragments by concatenating the respective sequences of the paired-end short reads.

```

seqs = strcat( getSequence(bm1_filtered, fow_idx),...
               getSequence(bm1_filtered, mate_idx));

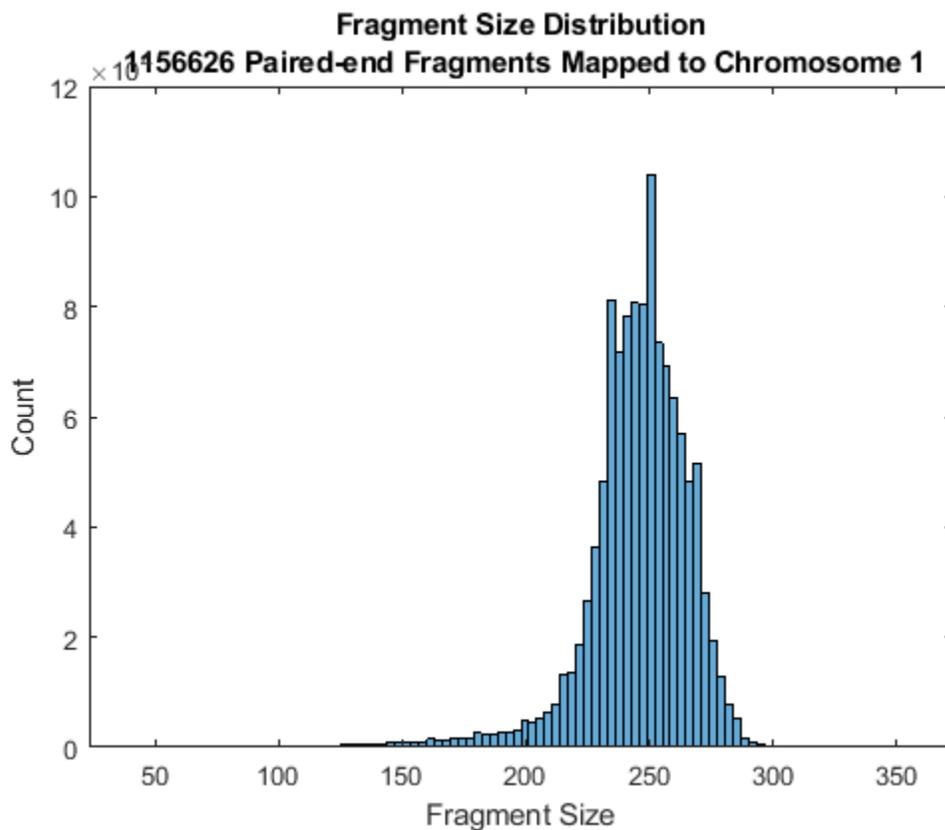
```

Calculate and plot the fragment size distribution.

```

J = getStart(bm1_filtered,fow_idx);
K = getStop(bm1_filtered,mate_idx);
L = K - J + 1;
figure
histogram(double(L),100)
title(sprintf('Fragment Size Distribution\n %d Paired-end Fragments Mapped to Chromosome 1',n))
xlabel('Fragment Size')
ylabel('Count')

```



Construct a new BioMap to represent the sequencing fragments. With this, you will be able explore the coverage signals as well as local alignments of the fragments.

```

bm1_fragments = BioMap('Sequence',seqs,'Signature',cigars,'Start',J)

```

```

bm1_fragments =
  BioMap with properties:
    SequenceDictionary: {0x1 cell}
      Reference: {0x1 cell}
      Signature: {1156626x1 cell}
      Start: [1156626x1 uint32]
    MappingQuality: [0x1 uint8]
      Flag: [0x1 uint16]
    MatePosition: [0x1 uint32]
      Quality: {0x1 cell}
      Sequence: {1156626x1 cell}
      Header: {0x1 cell}
      NSeqs: 1156626
      Name: ''

```

Exploring the Coverage Using Fragment Alignments

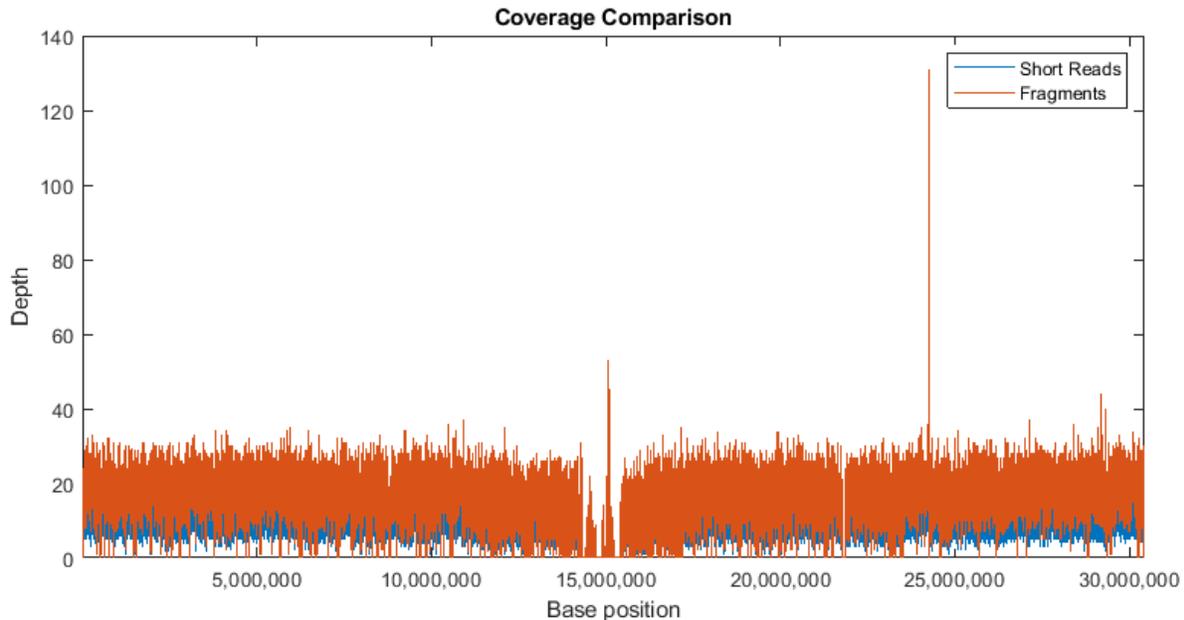
Compare the coverage signal obtained by using the reconstructed fragments with the coverage signal obtained by using individual paired-end reads. Notice that enriched binding sites, represented by peaks, can be better discriminated from the background signal.

```

cov_reads = getBaseCoverage(bm1_filtered,x1,x2,'binWidth',1000,'binType','max');
[cov_fragments,bin] = getBaseCoverage(bm1_fragments,x1,x2,'binWidth',1000,'binType','max');

figure
plot(bin,cov_reads,bin,cov_fragments)
xlim([x1,x2]) % sets the x-axis limits
fixGenomicPositionLabels % formats tick labels and adds data cursors
xlabel('Base position')
ylabel('Depth')
title('Coverage Comparison')
legend('Short Reads','Fragments')

```



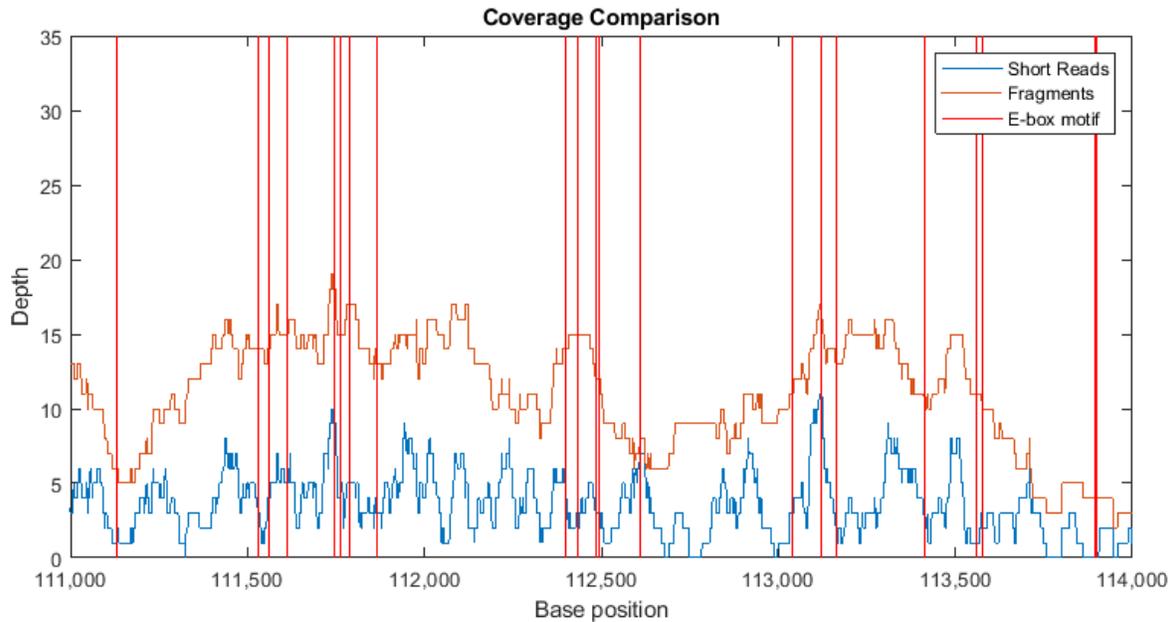
Perform the same comparison at the bp resolution. In this dataset, Wang et.al. [1] investigated a basic helix-loop-helix (*bHLH*) transcription factor. *bHLH* proteins typically bind to a consensus sequence called an *E-box* (with a *CANNTG* motif). Use `fastaread` to load the reference chromosome, search for the *E-box* motif in the 3' and 5' directions, and then overlay the motif positions on the coverage signals. This example works over the region 1-200,000, however the figure limits are narrowed to a 3000 bp region in order to better depict the details.

```
p1 = 1;
p2 = 200000;

cov_reads = getBaseCoverage(bm1_filtered,p1,p2);
[cov_fragments,bin] = getBaseCoverage(bm1_fragments,p1,p2);

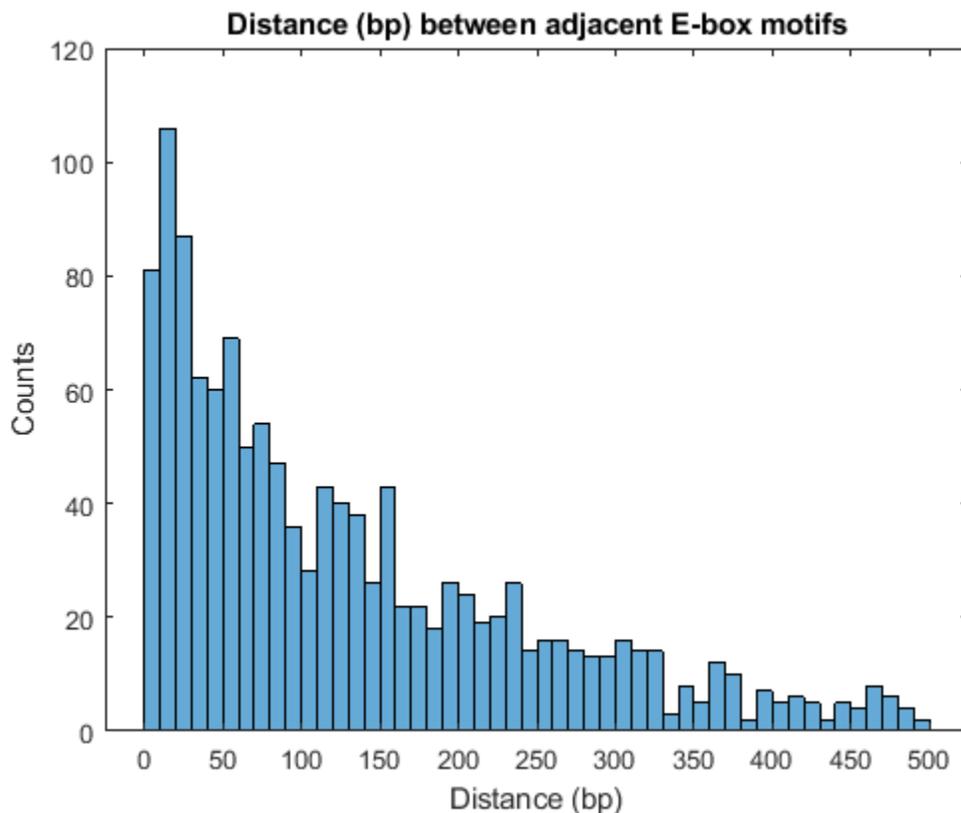
chr1 = fastaread('ach1.fasta');
mp1 = regexp(chr1.Sequence(p1:p2),'CA..TG')+3+p1;
mp2 = regexp(chr1.Sequence(p1:p2),'GT..AC')+3+p1;
motifs = [mp1 mp2];

figure
plot(bin,cov_reads,bin,cov_fragments)
hold on
plot([1;1;1]*motifs,[0;max(ylim);NaN],'r')
xlim([111000 114000]) % sets the x-axis limits
fixGenomicPositionLabels % formats tick labels and adds datacursors
xlabel('Base position')
ylabel('Depth')
title('Coverage Comparison')
legend('Short Reads','Fragments','E-box motif')
```



Observe that it is not possible to associate each peak in the coverage signals with an *E-box* motif. This is because the length of the sequencing fragments is comparable to the average motif distance, blurring peaks that are close. Plot the distribution of the distances between the *E-box* motif sites.

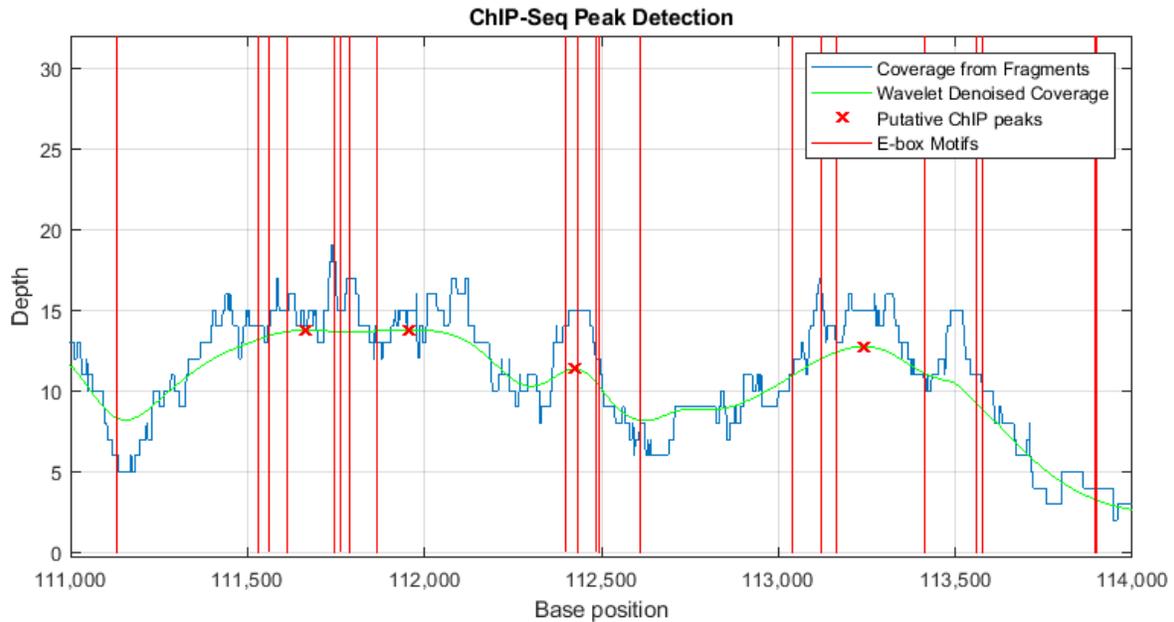
```
motif_sep = diff(sort(motifs));
figure
histogram(motif_sep(motif_sep<500),50)
title('Distance (bp) between adjacent E-box motifs')
xlabel('Distance (bp)')
ylabel('Counts')
```



Finding Significant Peaks in the Coverage Signal

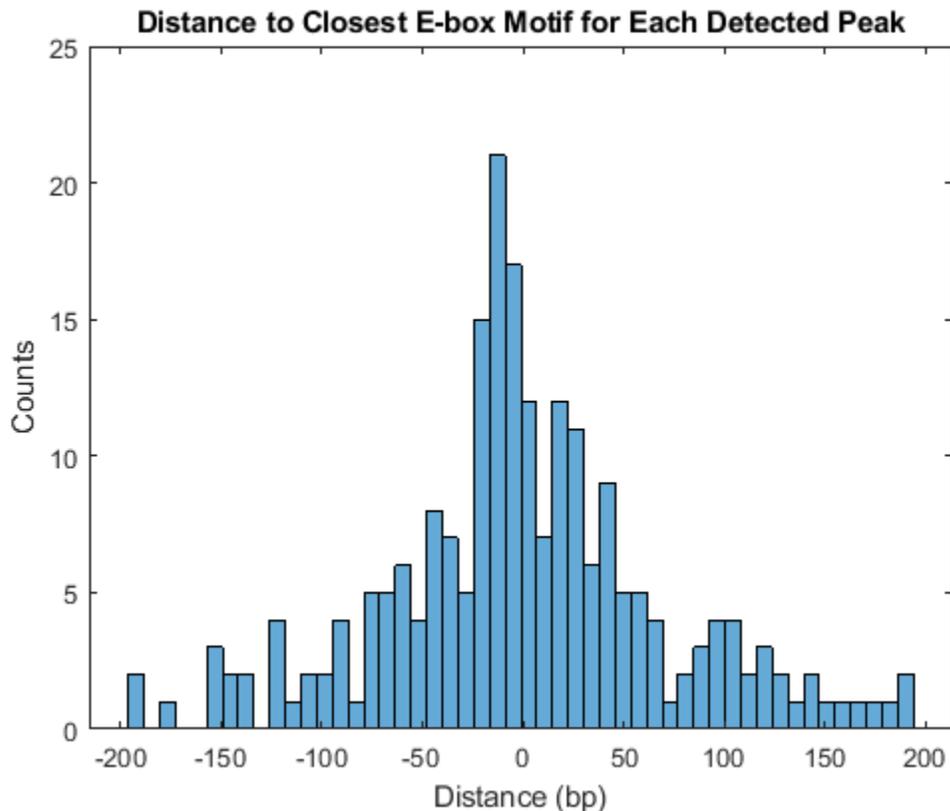
Use the function `mspeaks` to perform peak detection with Wavelets denoising on the coverage signal of the fragment alignments. Filter putative ChIP peaks using a height filter to remove peaks that are not enriched by the binding process under consideration.

```
putative_peaks = mspeaks(bin,cov_fragments,'noiseestimator',20,...
                        'heightfilter',10,'showplot',true);
hold on
legend('off')
plot([1;1;1]*motifs(motifs>p1 & motifs<p2),[0;max(ylim);NaN],'r')
xlim([111000 114000])      % sets the x-axis limits
fixGenomicPositionLabels  % formats tick labels and adds datacursors
legend('Coverage from Fragments','Wavelet Denoised Coverage','Putative ChIP peaks','E-box Motifs')
xlabel('Base position')
ylabel('Depth')
title('ChIP-Seq Peak Detection')
```



Use the `knnsearch` function to find the closest motif to each one of the putative peaks. As expected, most of the enriched ChIP peaks are close to an *E-box* motif [1]. This reinforces the importance of performing peak detection at the finest resolution possible (bp resolution) when the expected density of binding sites is high, as it is in the case of the *E-box* motif. This example also illustrates that for this type of analysis, paired-end sequencing should be considered over single-end sequencing [1].

```
h = knnsearch(motifs',putative_peaks(:,1));
distance = putative_peaks(:,1)-motifs(h(:))';
figure
histogram(distance(abs(distance)<200),50)
title('Distance to Closest E-box Motif for Each Detected Peak')
xlabel('Distance (bp)')
ylabel('Counts')
```



References

- [1] Wang, Congmao, Jie Xu, Dasheng Zhang, Zoe A Wilson, and Dabing Zhang. "An Effective Approach for Identification of in Vivo Protein-DNA Binding Sites from Paired-End ChIP-Seq Data." *BMC Bioinformatics* 11, no. 1 (2010): 81.
- [2] Li, H., and R. Durbin. "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform." *Bioinformatics* 25, no. 14 (July 15, 2009): 1754-60.
- [3] Li, H., B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. "The Sequence Alignment/Map Format and SAMtools." *Bioinformatics* 25, no. 16 (August 15, 2009): 2078-79.
- [4] Jothi, R., S. Cuddapah, A. Barski, K. Cui, and K. Zhao. "Genome-Wide Identification of in Vivo Protein-DNA Binding Sites from ChIP-Seq Data." *Nucleic Acids Research* 36, no. 16 (August 1, 2008): 5221-31.
- [5] Hoffman, Brad G, and Steven J M Jones. "Genome-Wide Identification of DNA-Protein Interactions Using Chromatin Immunoprecipitation Coupled with Flow Cell Sequencing." *Journal of Endocrinology* 201, no. 1 (April 2009): 1-13.
- [6] Ramsey, Stephen A., Theo A. Knijnenburg, Kathleen A. Kennedy, Daniel E. Zak, Mark Gilchrist, Elizabeth S. Gold, Carrie D. Johnson, et al. "Genome-Wide Histone Acetylation Data Improve Prediction of Mammalian Transcription Factor Binding Sites." *Bioinformatics* 26, no. 17 (September 1, 2010): 2071-75.

See Also

BioMap | [getBaseCoverage](#) | [getgenbank](#) | [getSummary](#)

Related Examples

- “Identifying Differentially Expressed Genes from RNA-Seq Data” on page 2-32
- “Count Features from NGS Reads” on page 2-23
- “Exploring Genome-Wide Differences in DNA Methylation Profiles” on page 2-59

Working with Illumina/Solexa Next-Generation Sequencing Data

This example shows how to read and perform basic operations with data produced by the Illumina®/Solexa Genome Analyzer®.

Introduction

During an analysis run with the Genome Analyzer Pipeline software, several intermediate files are produced. This example shows how to read and manipulate the information contained in sequence files (`_sequence.txt`).

Reading `_sequence.txt` (FASTQ) Files

The `_sequence.txt` files are FASTQ-formatted files that contain the sequence reads and their quality scores, after quality trimming and filtering. You can use the `fastqinfo` function to display a summary of the contents of a `_sequence.txt` file, and the `fastqread` function to read the contents of the file. The output, `reads`, is a cell array of structures containing the Header, Sequence and Quality fields.

```
filename = 'ilmnsolexa_sequence.txt';
info = fastqinfo(filename)
reads = fastqread(filename)

info =

    struct with fields:

        Filename: 'ilmnsolexa_sequence.txt'
        FilePath: 'C:\TEMP\tpadbe7f41\bioinfo-ex25447385'
        FileModDate: '06-May-2009 16:02:48'
        FileSize: 30124
        NumberOfEntries: 260

reads =

    1×260 struct array with fields:

        Header
        Sequence
        Quality
```

Because there is one sequence file per tile, it is not uncommon to have a collection of over 1,000 files in total. You can read the entire collection of files associated with a given analysis run by concatenating the `_sequence.txt` files into a single file. However, because this operation usually produces a large file that requires ample memory to be stored and processed, it is advisable to read the content in chunks using the `blockread` option of the `fastqread` function. For example, you can read the first `M` sequences, or the last `M` sequences, or any `M` sequences in the file.

```
M = 150;
N = info.NumberOfEntries;
```

```
readsFirst = fastqread(filename, 'blockread', [1 M])
readsLast = fastqread(filename, 'blockread', [N-M+1, N])
```

```
readsFirst =
```

```
1×150 struct array with fields:
```

```
Header
Sequence
Quality
```

```
readsLast =
```

```
1×150 struct array with fields:
```

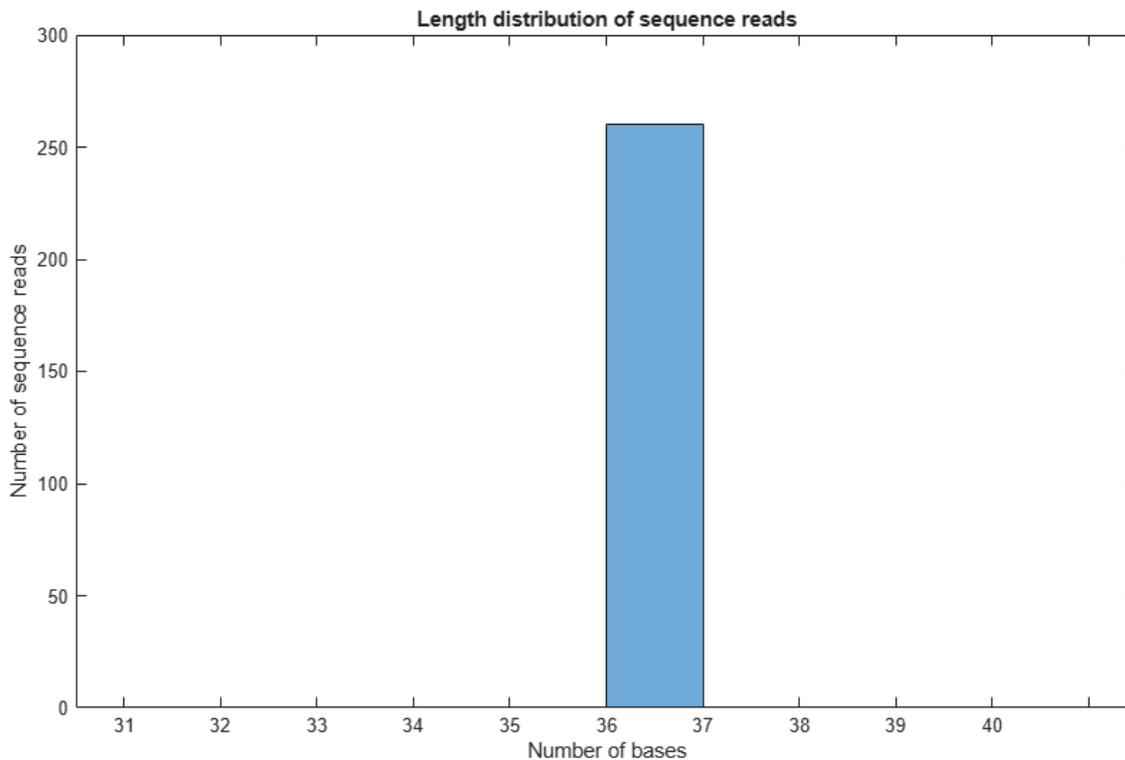
```
Header
Sequence
Quality
```

Surveying the Length Distribution of Sequence Reads

Once you load the sequence information into your workspace, you can determine the number and length of the sequence reads and plot their distribution as follows:

```
seqs = {reads.Sequence};
readsLen = cellfun(@length, seqs);

figure(); histogram(readsLen,10);
xlabel('Number of bases'); ylabel('Number of sequence reads');
xticklabels([31:1:40]);
title('Length distribution of sequence reads')
```



As expected, in this example all sequence reads are 36 bp long.

Surveying the Base Composition of the Sequence Reads

You can also examine the nucleotide composition by surveying the number of occurrences of each base type in each sequence read, as shown below:

```
nt = {'A', 'C', 'G', 'T'};
N = size(seqs, 2);
pos = cell(4, N);

for i = 1:4
    pos(i,:) = cellfun(@(s) strfind(s, nt{i}), seqs, 'UniformOutput', false);
end

count = zeros(4, N);
for i = 1:4
    count(i,:) = cellfun(@length, pos(i,:));
end

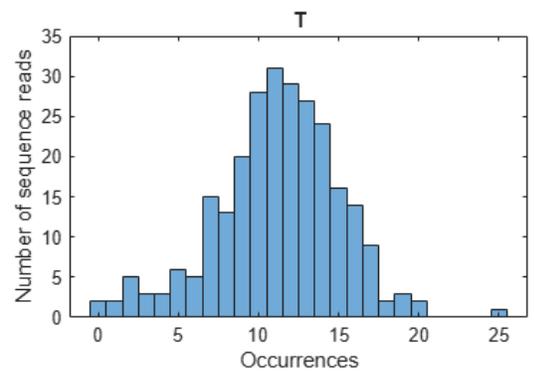
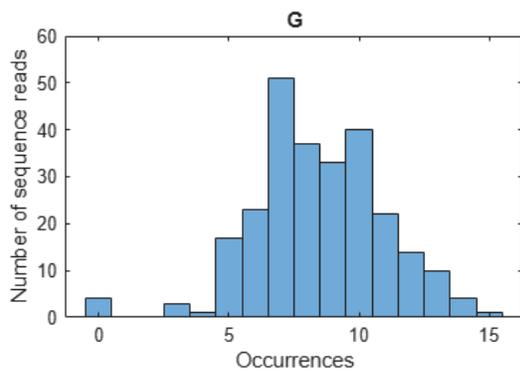
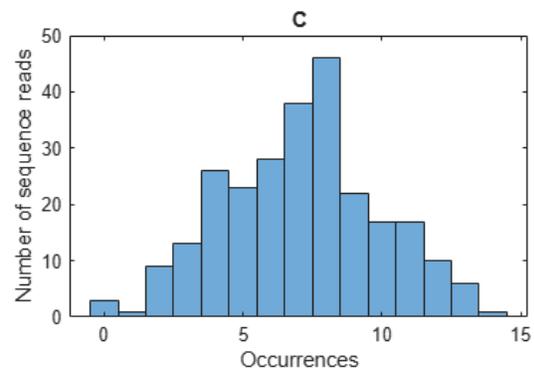
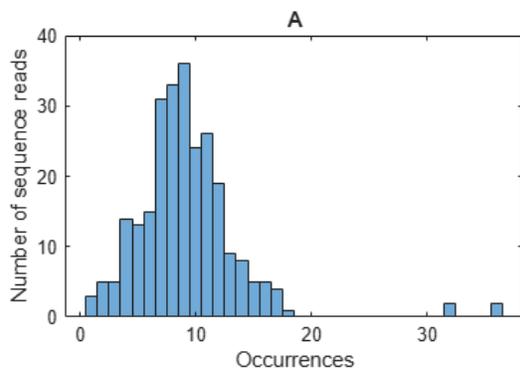
% Plot nucleotide distribution in subplots
figure();
for i = 1:4
    subplot(2, 2, i);
    histogram(count(i, :));
    title(nt{i});
    xlabel('Occurrences');
    ylabel('Number of sequence reads');
```

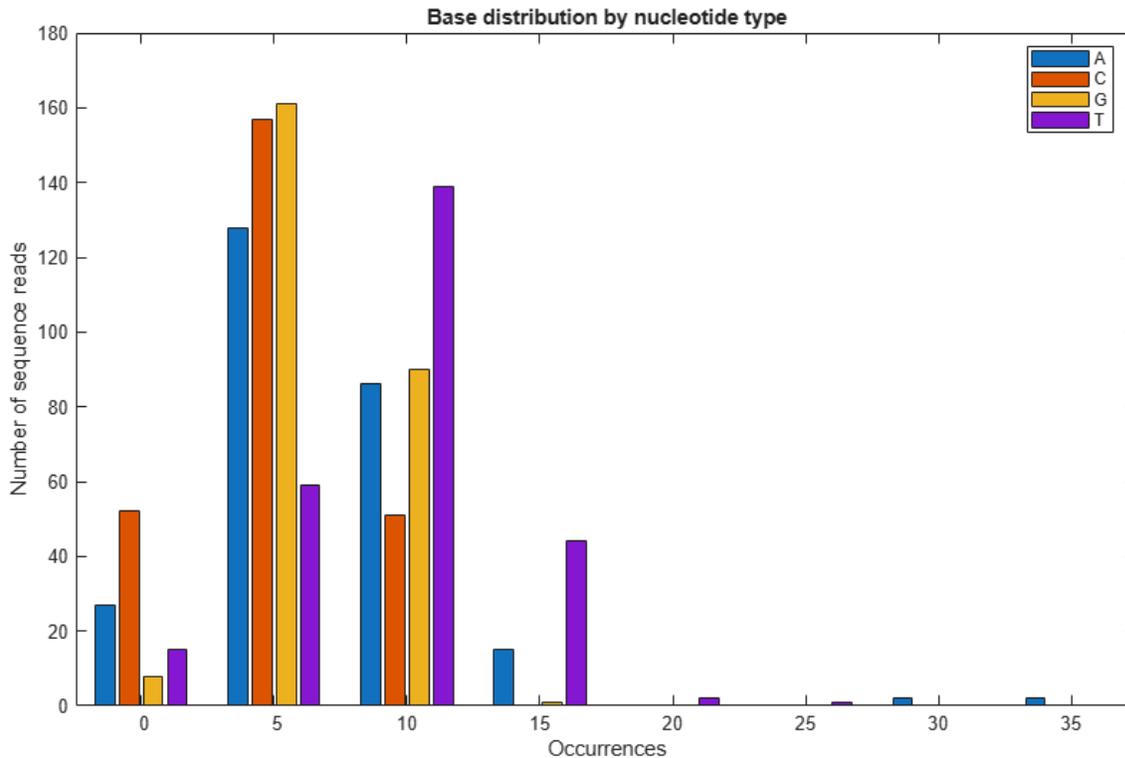
```
end
```

```
binWidth = 5;
maxCount = max(count(:));
edges = 0:binWidth:maxCount+binWidth;

histData = zeros(length(edges)-1, 4);
for i = 1:4
    histData(:, i) = histcounts(count(i,:), edges);
end
```

```
% Plot grouped bars
figure;
bar(histData, 'grouped');
xticks(1:length(edges)-1);
xticklabels(cellstr(num2str((edges(1:end-1)))));
xlabel('Occurrences');
ylabel('Number of sequence reads');
legend('A', 'C', 'G', 'T', 'Location', 'Best');
title('Base distribution by nucleotide type');
```





Surveying the Quality Score Distribution

Each sequence read in the `_sequence.txt` file is associated with a score. The score is defined as $SQ = -10 * \log_{10} (p / (1-p))$, where p is the probability error of a base. You can examine the quality scores associated with the base calls by converting the ASCII format into a numeric representation, and then plotting their distribution, as shown below:

```
sq = {reads.Quality}; % in ASCII format
SQ = cellfun(@(x) double(x)-64, {reads.Quality}, 'UniformOutput', false); % in integer format

%=== average, median and standard deviation
avgSQ = cellfun(@mean, SQ);
medSQ = cellfun(@median, SQ);
stdSQ = cellfun(@std, SQ);

%=== plot distribution of median and average quality
figure();
subplot(1,2,1); histogram(medSQ);
xlabel('Median Score SQ'); ylabel('Number of sequence reads');
subplot(1,2,2); boxplot(avgSQ); ylabel('Average Score SQ');
```



```
'GGACTTTGTAGGATACCCTCGCTTCCTtcTCCTgT'
```

Summarizing Read Occurrences

To summarize read occurrences, you can determine the number of unique read sequences and their distribution across the data set. You can also identify those sequence reads that occur multiple times, often because they correspond to adapters or primers used in the sequencing process.

```
%=== determine read frequency
[uReads,~,n] = unique({reads.Sequence});
numUnique = numel(uReads)
readFreq = accumarray(n(:,1),1);
figure(); histogram(readFreq, unique(readFreq));
xlabel('Occurrences'); ylabel('Number of sequence reads');
xticks([1 2 3])
title('Read occurrences');

%=== identify multiply-occurring sequence reads
d = readFreq > 1;
dupReads = uReads(d)
dupFreq = readFreq(d)
```

```
numUnique =
```

```
250
```

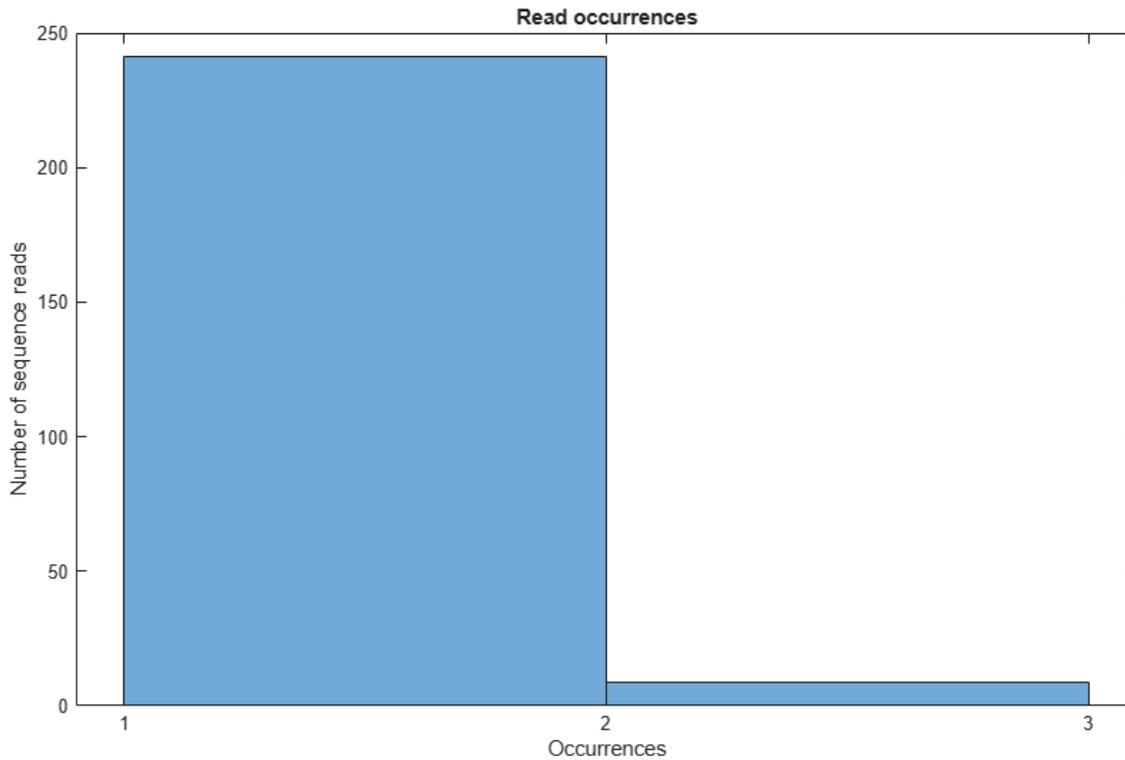
```
dupReads =
```

```
9×1 cell array
```

```
{'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'}
{'GATTTTATTGGTATCAGGGTTAATCGTGCCAAGAAA'}
{'GCATGGGTGATGCTGGTATTAATCTGCCATTCAAG'}
{'GGGATGAACATAATAAGCAATGACGGCAGCAATAAA'}
{'GGGGGAGCACATTGTAGCATTGTGCCAATTCATCCA'}
{'GGTTATTAAGAGATTATTTGTCTCCAGCCACTTAA'}
{'GTTCTCACTTCTGTTACTCCAGCTTCTTCGGCACCT'}
{'GTTGCTGCCATCTCAAAAACATTTGGACTGCTCCGC'}
{'GTTGGTTTCTATGTGGCTAAATACGTAAACAAAAG'}
```

```
dupFreq =
```

```
2 2 2 2 2 2 3 2 2
```



Identifying Homopolymers Artifacts

Illumina/Solexa sequencing may produce false polyA at the edges of a tile. To identify these artifacts, you need to identify homopolymers, that is, sequence reads composed of one type of nucleotide only. In the data set under consideration, there are two homopolymers, both of which are polyA.

```
%=== find homopolymers
pc = (count ./ len) * 100;
[homopolType,homopolIndex] = find(pc == 100);

homopolIndex
homopol = {reads(homopolIndex).Sequence}'

homopolIndex =

    251
    257

homopol =

    2x1 cell array

    {'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'}
    {'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'}
```

Similarly, you can identify sequence reads that are near-matches to homopolymers, that is, sequence reads that are composed almost exclusively of one nucleotide type.

```
%=== find near-homopolymers
[nearhomopolType, nearhomopolIndex] = find(pc < 100 & pc > 85); % more than 85% same base
nearhomopolIndex
nearHomopol = {reads(nearhomopolIndex).Sequence}'
```

```
nearhomopolIndex =
```

```
    4
   243
```

```
nearHomopol =
```

```
    2×1 cell array
```

```
    {'AAAAACATAAAAAAAAAAATAAAAAACAAAAAAAA'}
    {'AAAAAATAAAAAAAAAAATAAAAAATTAAAAA'}
```

Writing Data to FASTQ Format

Once you have processed and analyzed your data, it might be convenient to save a subset of sequences in a separate FASTQ file for future consideration. For this purpose you can use the `fastqwrite` function.

Bioinformatics Pipeline SplitDimension

Some of the blocks in a bioinformatics pipeline operate on their input data arrays as one single input while other blocks can operate on individual elements or slices of the input data array independently. The `SplitDimension` property of a block input controls how to split the block input data (or input array) across multiple runs of the same block in a pipeline. In other words, `SplitDimension` allows you to control how to parallelize independent runs of the same block (with a different input for each run).

Specify SplitDimension to Select Which Input Array Dimensions to Split

You can specify a vector of integers to indicate which dimensions (such the row or column dimension) of the input array to split and pass to the block run method. By splitting the input data, you are specifying how many times you want to run the same block with different inputs.

For example, the `bioinfo.pipeline.block.SeqSplit` block can apply the same trimming operation on an array of input FASTQ files. To specify that `SeqTrim` runs on each input file in the array independently, set the `SplitDimension` property of the block input to a specific dimension (such as 1 for the row dimension or 2 for the column dimension of the array).

You can also specify an empty array `[]` as the value to perform no dimension splitting of input data, that is, the block runs one time for all of input data. Alternatively, specify "all" to pass all elements of the input array to the run method of the block independently. For instance, if there are n elements, the block runs n times independently.

For an example of how to use `SplitDimension`, see "Split Input SAM Files and Assemble Transcriptomes Using Bioinformatics Pipeline" on page 2-112.

Note If you are running the "Bioinformatics Toolbox Software Support Packages" on page 2-21 (such as `Bowtie2`, `BWA`, or `Cufflinks`) remotely, ensure that these support packages are installed in the remote clusters that you are running the pipeline.

Provide Compatible Array sizes

A block can have different split dimensions for each input (port), but inputs that share split dimensions must have compatible sizes. As with binary operations on MATLAB arrays, two inputs have a compatible size for a dimension if the size of the inputs is the same or one of the dimension sizes is 1. For an input whose size is 1 (or scalar) in a split dimension, the value in that dimension is implicitly expanded to match the same size as the other dimensions. For MATLAB arrays, dimension one refers to the number of rows and dimension two refers to the number of columns.

The total number of times the block runs within a pipeline is the product of the sizes of the input value in the split dimensions. For example, consider a block with two input ports X and Y . The following table shows the total number of runs (or processes) for various values of `SplitDimension`.

X array size	Y array size	X.SplitDimension	Y.SplitDimension	Total number of runs
1-by-1	2-by-2	[]	[]	$1 \times 1 = 1$. This is the default (no dimensional splitting).
1-by-1	2-by-3	[]	1	$2 \times 1 = 2$
5-by-1	1-by-3	1	2	$5 \times 3 = 15$
2-by-2	3-by-3	2	2	0 because of dimension mismatch
2-by-3	2-by-4	2	"all"	0 because of dimension mismatch
3-by-1-by-4	1-by-3	"all"	2	$3 \times 3 \times 4 = 36$
0-by-1	1-by-1	[]	[]	$1 \times 1 = 1$
0-by-1	1-by-1	1	[]	0 because of size 0 in dimension 1

Empty sizes are allowed only in non-SplitDimension. If no inputs specify a SplitDimension, there will always be exactly one run, regardless of the input array sizes. You can merge the output results from multiple block runs with cell arrays. For details, see UniformOutput.

Default Value of SplitDimension for Built-In Pipeline Blocks

The default value of the SplitDimension property is "all", instead of being empty, for some input ports of built-in pipeline blocks when the expected use case for the blocks is to parallelize across all input data for those input ports.

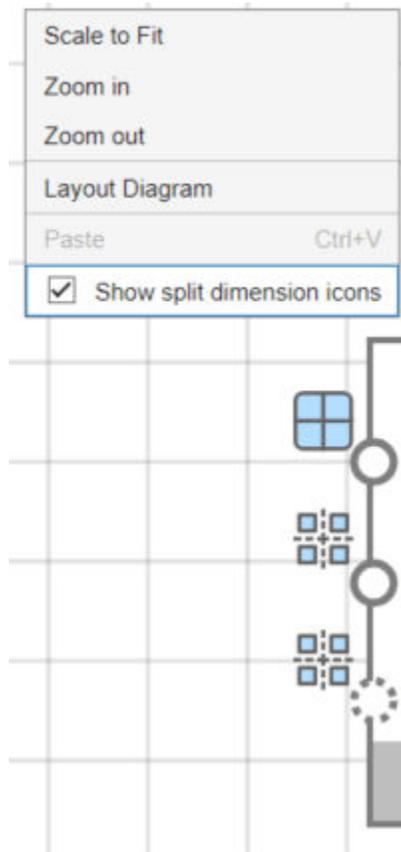
The table below lists all the built-in blocks with their corresponding SplitDimension values. (The UserFunction and FileChooser blocks have no input ports.)

Built-in Block	Input Port	SplitDimension Default Value
BLASTN	QueryFile	"all"
	BlastDatabase	[]
BLASTP	QueryFile	"all"
	BlastDatabase	[]
BLASTX	QueryFile	"all"
	BlastDatabase	[]
BamSort	BAMFile	"all"
Bowtie2	IndexBaseName	[]
	Reads1Files	"all"
	Reads2Files	"all"

Bowtie2Build	ReferenceFASTAFiles	[]
	IndexBaseName	[]
BwaIndex	ReferenceFASTAFile	"all"
BwaMEM	IndexBaseName	[]
	Reads1File	"all"
	Reads2File	"all"
CuffCompare	GenomicAnnotationFiles	[]
CuffDiff	GenomicAnnotationFile	[]
	GenomicAlignmentFiles	[]
CuffMerge	GenomicAnnotationFiles	[]
CuffNorm	GenomicAnnotationFile	[]
	GenomicAlignmentFiles	[]
CuffQuant	GenomicAnnotationFile	[]
	GenomicAlignmentFiles	[]
Cufflinks	GenomicAlignmentFiles	"all"
FeatureCount	GTFFile	[]
	GenomicAlignmentFiles	[]
GenomicsViewer	Reference	[]
	Cytoband	[]
	Tracks	[]
Load	MatFile	[]
MakeBlastDatabase	InputFile	[]
SRAFasterqDump	SRRID	"all"
SRASAMDump	SRRID	"all"
SamSort	SAMFile	"all"
Save	Var1	[]
SeqFilter	FASTQFiles	"all"
SeqSplit	FASTQFiles	"all"
	BarcodeFile	"all"
SeqTrim	FASTQFiles	"all"
TBLASTN	QueryFile	"all"
	BlastDatabase	[]
TBLASTX	QueryFile	"all"
TBLASTX	BlastDatabase	[]

Show split dimensions in Biopipeline Designer

In Biopipeline Designer, you can see dedicated icons for the split dimension settings of the input ports of your pipeline blocks. To show or hide the icons, open the diagram context menu and select **Show split dimension icons**.



The three icons indicate the following:

-  — Inputs to this port are not split along any dimension.
-  — Inputs to this port are split along one dimension.
-  — Inputs to this port are split along more than one dimension.

See Also

SplitDimension | `bioinfo.pipeline.Input` | `bioinfo.pipeline.Pipeline` | Biopipeline Designer

Related Examples

- “Split Input SAM Files and Assemble Transcriptomes Using Bioinformatics Pipeline” on page 2-112

Split Input SAM Files and Assemble Transcriptomes Using Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.block.*
```

Create a pipeline.

```
P = Pipeline
P =
  Pipeline with properties:
    Blocks: [0x1 bioinfo.pipeline.Block]
    BlockNames: [0x1 string]
```

Use a FileChooser block to select the provided SAM files. The files contain aligned reads for *Mycoplasma pneumoniae* from two samples.

```
fileChooserBlock = FileChooser([which("Myco_1_1.sam"); which("Myco_1_2.sam")]);
```

Create a Cufflinks block.

```
cufflinksBlock = Cufflinks;
```

Add the blocks to the pipeline.

```
addBlock(P,[fileChooserBlock,cufflinksBlock]);
```

Connect the blocks.

```
connect(P,fileChooserBlock,cufflinksBlock,["Files","GenomicAlignmentFiles"]);
```

Set SplitDimension to 1 for the GenomicAlignmentFiles input port. The value of 1 corresponds to the row dimension of the input, which means that the Cufflinks block will run on each individual SAM files (Myco_1_1.sam and Myco_1_2.sam).

```
cufflinksBlock.Inputs.GenomicAlignmentFiles.SplitDimension = 1;
```

Run the pipeline. The pipeline runs Cufflinks block two times independently and generates a set of four files for each SAM file.

```
run(P);
```

Get the block results.

```
cufflinksResults = results(P,cufflinksBlock)
cufflinksResults = struct with fields:
  TranscriptsGTFFile: [2x1 bioinfo.pipeline.datatype.File]
  IsoformsFPKMFile: [2x1 bioinfo.pipeline.datatype.File]
  GenesFPKMFile: [2x1 bioinfo.pipeline.datatype.File]
  SkippedTranscriptsGTFFile: [2x1 bioinfo.pipeline.datatype.File]
```

Use the process table to check the total number of runs for each block. Cufflinks ran two times independently.

```
t = processTable(P,Expanded=true);
```

Set `SplitDimension` to empty `[]` (which is the default). In this case, the pipeline does split the input files and runs Cufflinks just once for both SAM files, processing each SAM file one after another.

```
cufflinksBlock.Inputs.GenomicAlignmentFiles.SplitDimension = [];
deleteResults(P,IncludeFiles=true);
run(P);
cufflinksResults = results(P,cufflinksBlock)
```

```
cufflinksResults = struct with fields:
    TranscriptsGTFFile: [2x1 bioinfo.pipeline.datatype.File]
    IsoformsFPKMFile: [2x1 bioinfo.pipeline.datatype.File]
    GenesFPKMFile: [2x1 bioinfo.pipeline.datatype.File]
    SkippedTranscriptsGTFFile: [2x1 bioinfo.pipeline.datatype.File]
```

Check the process table, which confirms that Cufflinks ran just once.

```
t2 = processTable(P,Expanded=true);
```

Tip: you can speed up the pipeline run by setting `UseParallel=true` if you have Parallel Computing Toolbox™. The pipeline can schedule independent executions of blocks on parallel pool workers.

```
run(P,UseParallel=true)
```

See Also

[bioinfo.pipeline.Pipeline](#) | [bioinfo.pipeline.block.Cufflinks](#) | [SplitDimension](#)

Related Examples

- “Bioinformatics Pipeline SplitDimension” on page 2-108

Bioinformatics Pipeline Run Mode

When you rerun a pipeline after making some changes to it, the pipeline detects these changes and reruns only those blocks that are affected by these changes. This automatic change detection enables quick and efficient iterative workflows where you can tweak some parameters of a block or change the input values or data for your analysis.

By default, the pipeline uses the `Minimal` run mode. In this mode, the pipeline runs only the blocks for which one of the following statements is true:

- The block has not been run before or its results have been deleted.
- You have modified the block since the last time it ran.
- Input data, including new runtime inputs, to the block has changed since the last run.
- The block has one or more upstream blocks which have run since the last time the block was run.

If you are running only a subset of blocks within a pipeline, these rules are applied only to those selected blocks.

The other run mode is the `Full` mode. The pipeline runs all blocks even if they have previously computed results and there have been no changes affecting the block results.

Tip Use the default `Minimal` run mode whenever possible because skipping up-to-date blocks can save significant time running the pipeline, especially when the pipeline has long-running blocks that do not need to rerun.

See Also

`bioinfo.pipeline.Pipeline` | Biopipeline Designer

Create Simple Pipeline to Plot Sequence Quality Data Using Biopipeline Designer

This example shows how to create a bioinformatics pipeline in the **Biopipeline Designer** app that loads sequence read data, filters some sequences based on quality, and displays the quality statistics of the filtered data.

The example provides individual steps to build the pipeline from the beginning. To open the same completed pipeline, open the app. Then select **Open > Open Example Pipeline > Plot Sequence Quality Data**.

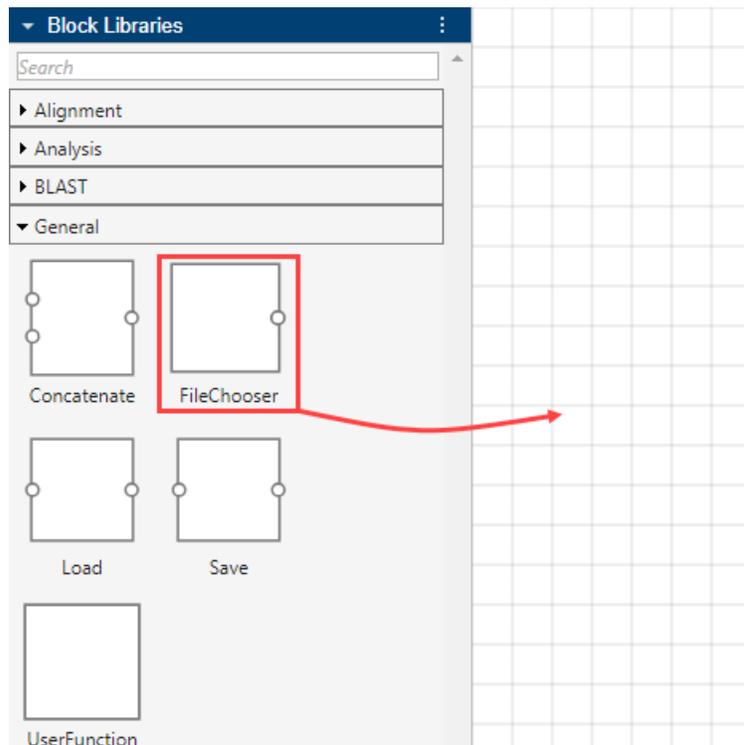
Open Biopipeline Designer App

Enter the following at the MATLAB® command line.

```
biopipelineDesigner
```

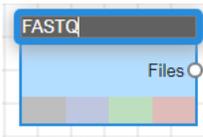
Select Input File Using FileChooser Block

In the **Block Libraries** panel of the app, scroll down to the **General** section. Drag the **FileChooser** block onto the diagram.



You can also use the **Search** box to look for specific built-in blocks in the **Block Libraries**.

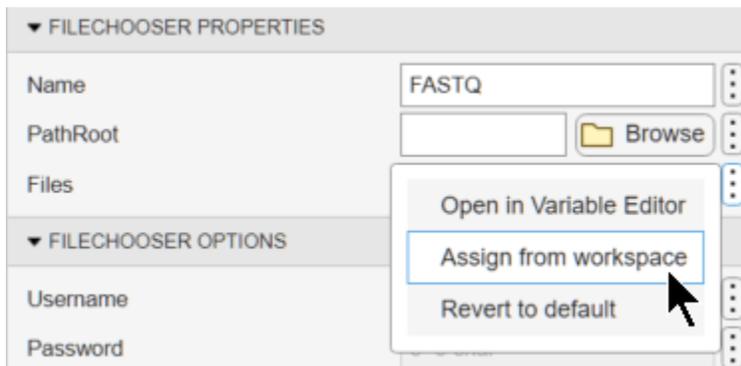
Double-click the block name `FileChooser_1` and rename as `FASTQ`.



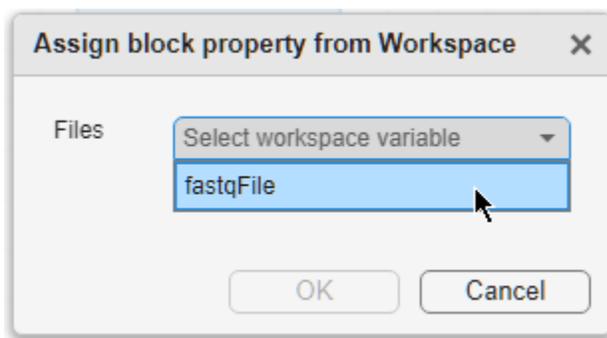
Run the following command at the MATLAB command line to create a variable that contains the full file path to the provided sequence read data.

```
fastqFile = which("SRR005164_1_50.fastq");
```

In the app, click the **FASTQ** block. In the **Pipeline Inspector** pane, under **FileChooser Properties**, click the vertical three-dot menu next to the **Files** property. Select **Assign from workspace**.



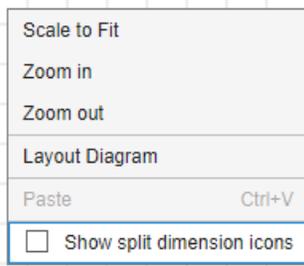
Select `fastqFile` from the list. Click **OK**.



Disable SplitDimension Icons Display

By default, the app displays dedicated icons for the split dimension settings of the input ports of your pipeline blocks. For the purposes of the example, disable the icon display. For more details about the icons and their meanings, see “Show split dimensions in Biopipeline Designer” on page 2-111.

Right-click in an empty area of the diagram and clear **Show split dimensions icons**. It would look as follows afterwards.



Filter Sequences Based on Quality

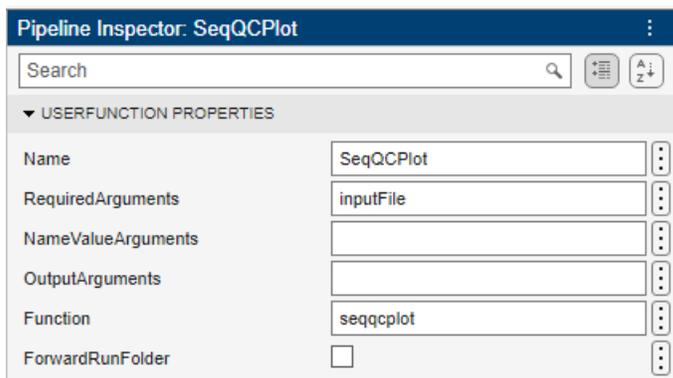
In the **Block Libraries** panel, under the **Sequence Utilities** section, drag the **SeqFilter** block onto the diagram. This block can filter sequences based on some specifications. The **Pipeline Inspector** panel shows the default values of the block properties and filtering options. In the **SeqFilter Options** section, change **Threshold** to **10,20**. Keep the other options as default. This **10,20** threshold value filters out any sequences with more than 10 low quality bases, where a base is considered low quality when its quality score is less than 20. For details, see `SeqFilterOptions`.

▼ SEQFILTER PROPERTIES	
Name	SeqFilter_1
▼ SEQFILTER OPTIONS	
Method	MaxNumberLowQualityBases
Threshold	10,20
WindowSize	Inf
Encoding	Illumina18
OutputSuffix	_filtered
PairedFiles	<input type="checkbox"/>
WriteSingleton	<input type="checkbox"/>
▼ INPUT PORT PROPERTIES	
FASTQFiles.Value	
FASTQFiles.SplitDimension	[]
▼ OUTPUT PORT PROPERTIES	
FilteredFASTQFiles.UniformOutput	<input checked="" type="checkbox"/>
NumFilteredIn.UniformOutput	<input checked="" type="checkbox"/>
NumFilteredOut.UniformOutput	<input checked="" type="checkbox"/>

Plot Sequence Quality Data

Create a custom (`bioinfo.pipeline.block.UserFunction`) block that calls an existing MATLAB function `seqqcplot` to plot the quality statistics of the filtered data.

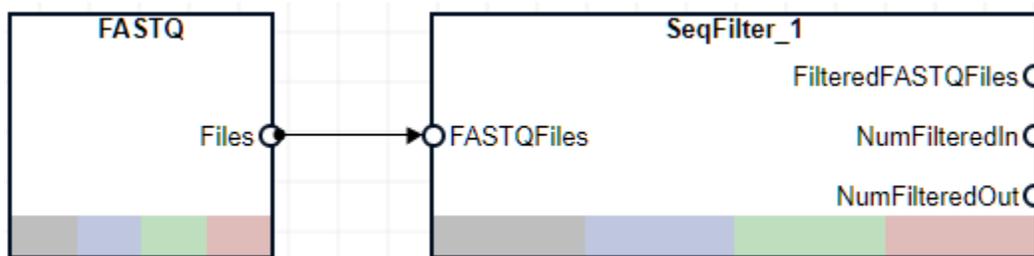
- 1 In the **Block Libraries** panel, under the **General** section, drag and drop the **UserFunction** block onto the diagram.
- 2 Rename the block to **SeqQCPlot**.
- 3 In the **Pipeline Inspector** pane, under **UserFunction Properties**, set the **RequiredArguments** to `inputFile` and **Function** to `seqqcplot`.



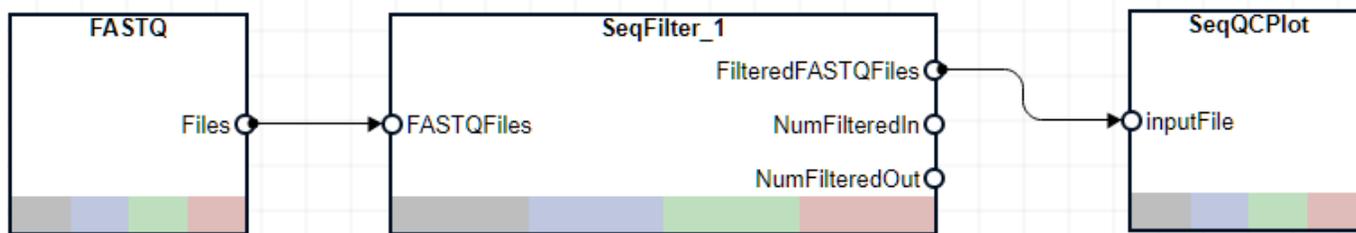
Connect Blocks and Run Pipeline

After setting up the blocks, you can now connect them to complete the pipeline.

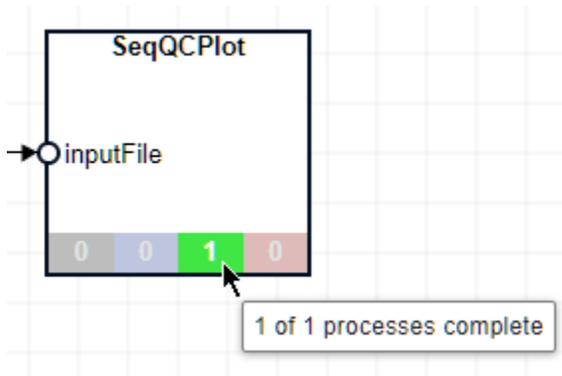
Drag an arrow from the **Files** output port of **FASTQ** to the **FASTQFiles** port of **SeqFilter_1**.



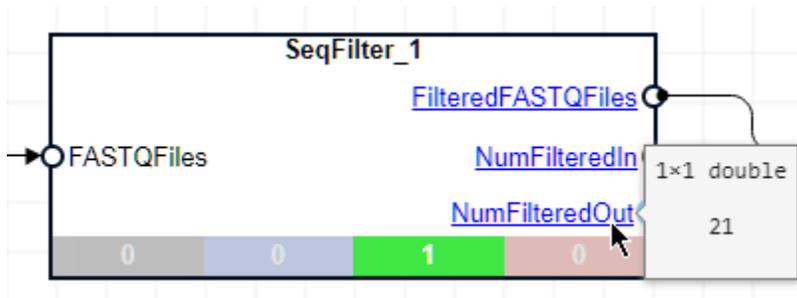
Next connect the **FilteredFASTQFiles** port to **inputFile** port.



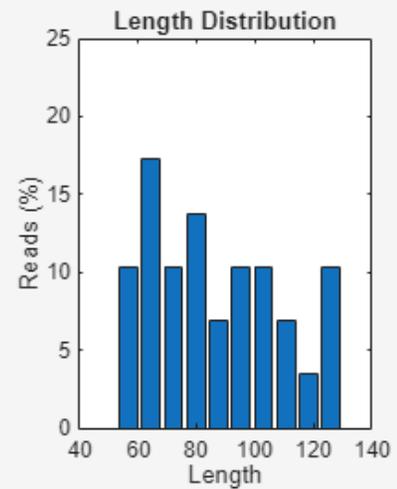
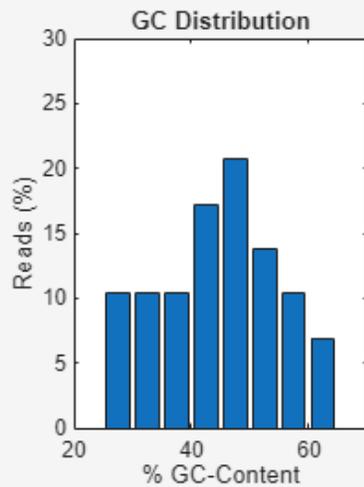
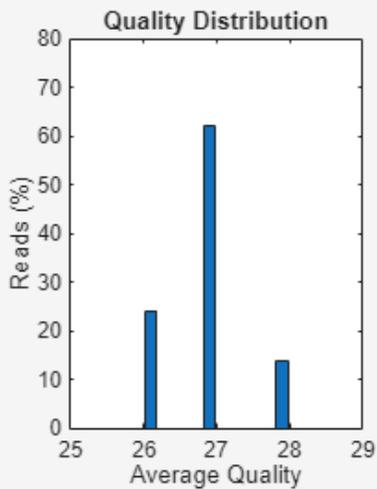
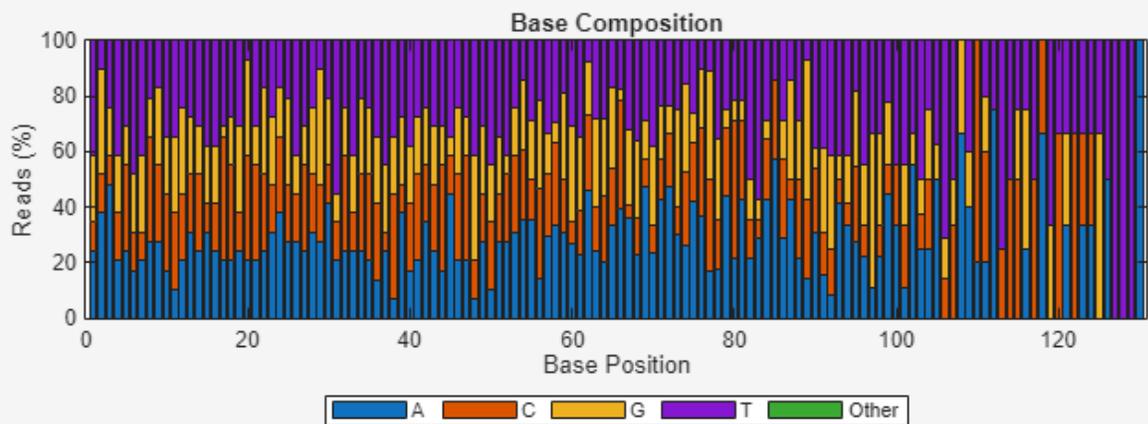
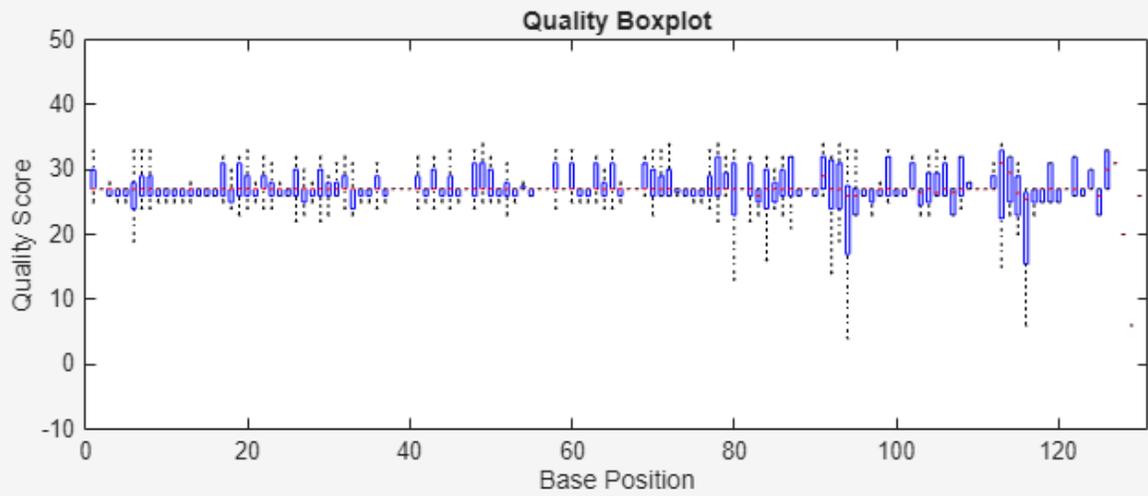
On the toolstrip of the app, click **Run**. During the run, you can see the progress of each block at its status bar. Point to a color-coded section with a number to see its meaning.



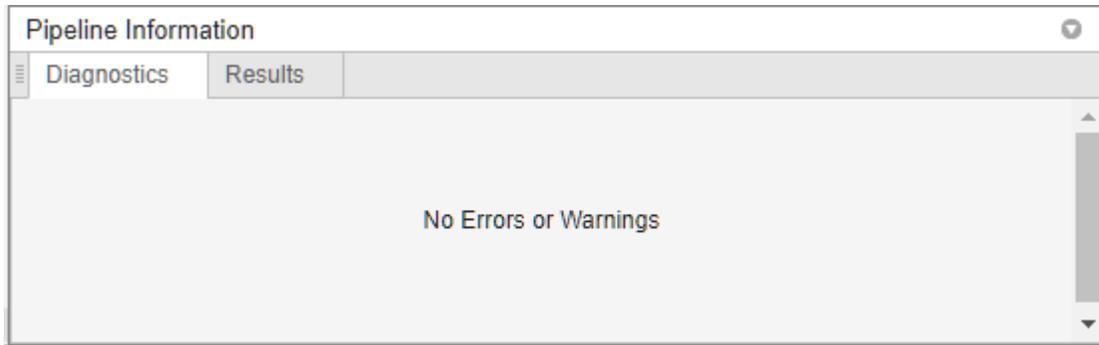
After the run, you can click each output port name of a block to see the output value. For example, click **NumFilteredOut** to see the total number of reads that were filtered out by the block.



The app generates the following figure, which contains quality statistics plots of the filtered data.



If there are any errors or warnings, the app shows them in the **Diagnostics** tab of the **Pipeline Information** panel, which is at the bottom of the diagram.



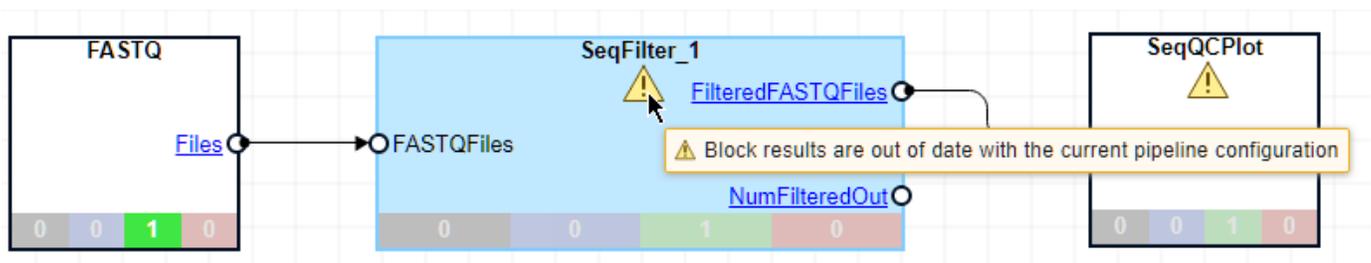
Click the **Results** tab. In the **Source** column, expand **SeqFilter_1** to see the block results, such as the filtered FASTQ file and the number of sequences that are selected and filtered out.

Pipeline Information	
Diagnostics	
Source	Value
▶ FASTQ	1×1 struct
▼ SeqFilter_1	1×1 struct
▶ FilteredFASTQFiles	1×1 File
▶ NumFilteredIn	29
▶ NumFilteredOut	21
SeqQCPlot	1×1 struct

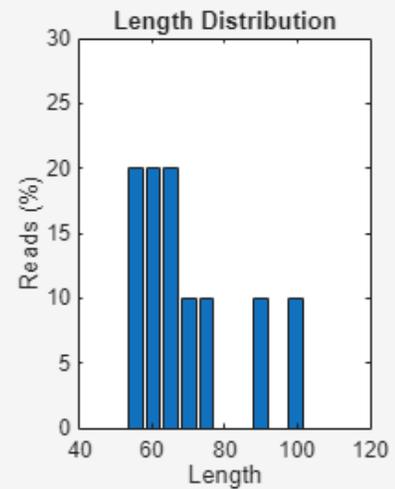
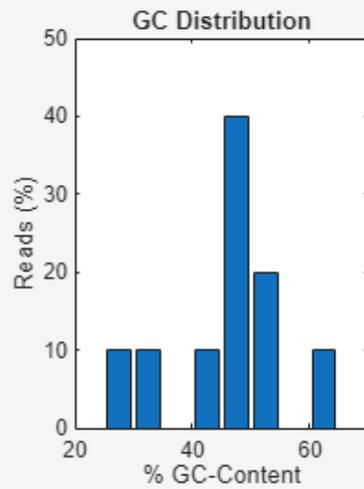
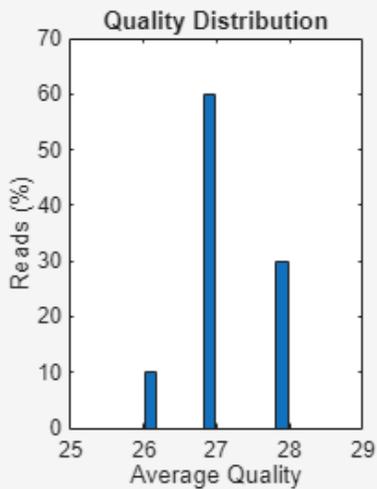
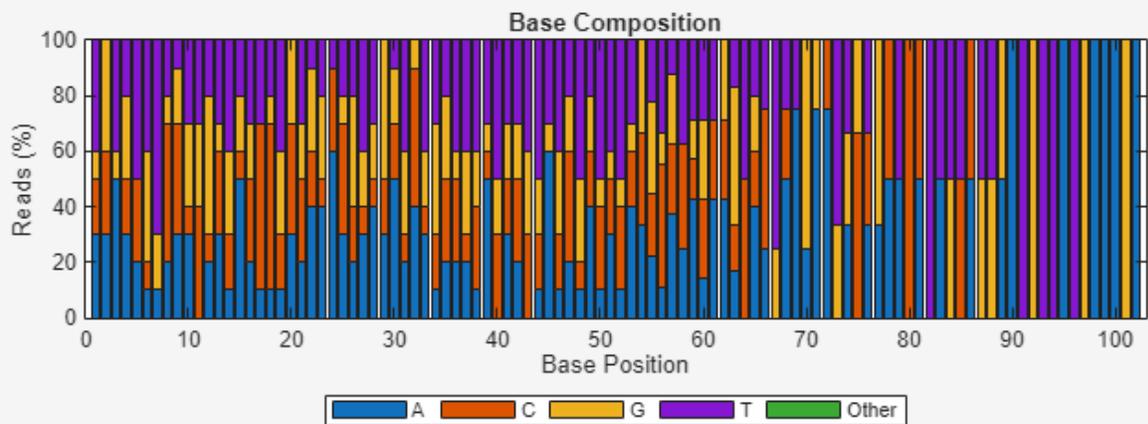
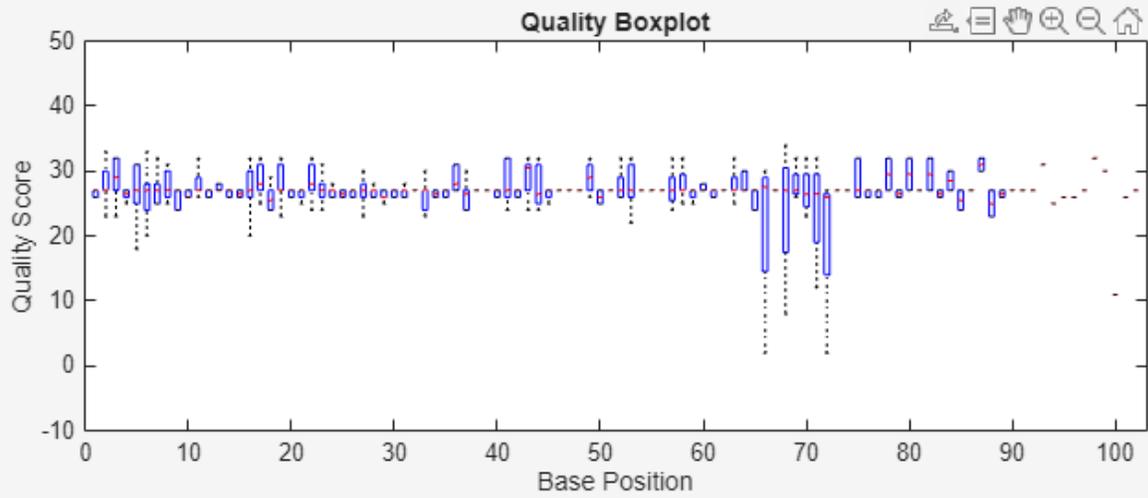
Rerun Pipeline with Different Filtering Threshold

You can specify a different threshold to filter sequences and rerun the pipeline. The app is aware of which blocks in the pipeline have changed and which other blocks, such as downstream blocks, are affected as a result. Hence, on subsequent runs, it reruns only those blocks that are needed, instead of every block in the pipeline. For details, see “Bioinformatics Pipeline Run Mode” on page 2-114.

Click **SeqFilter_1**. In the **Pipeline Inspector** panel, change its **Threshold** option to 5, 20. This setting now filters out any sequence with more than 5 low quality bases, where a base is considered low quality when its score is less than 20. Both **SeqFilter** and **SeqQCPlot** blocks now have a warning icon to indicate that the results are now out of date due to the change to the **SeqFilter** block.



Click **Run**. The app generates the following figure. During this run, the app does not rerun the **FASTQ** block because it is not needed. It only reruns the other two blocks.



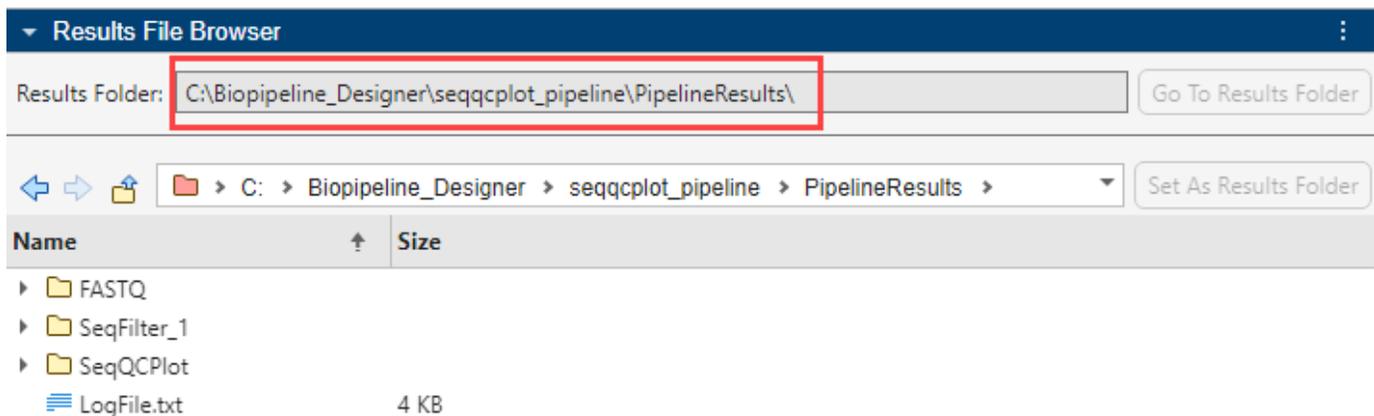
Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

Go to the **Results** tab of the **Pipeline Information** to check the new results.

Pipeline Information	
Diagnostics Results	
Source	Value
▶ FASTQ	1×1 struct
▼ SeqFilter_1	1×1 struct
▶ FilteredFASTQFiles	1×1 File
▶ NumFilteredIn	10
▶ NumFilteredOut	40
SeqQCPlot	1×1 struct

Navigate, View, and Access Output Files in Results File Browser

By default, the app saves the pipeline results in the PipelineResults folder in the MATLAB current folder. In this example, the current folder is C:\Biopipeline_Designer\seqqcplot_pipeline\ and the pipeline results folder is C:\Biopipeline_Designer\seqqcplot_pipeline\PipelineResults\ as shown in the **Results File Browser** pane.

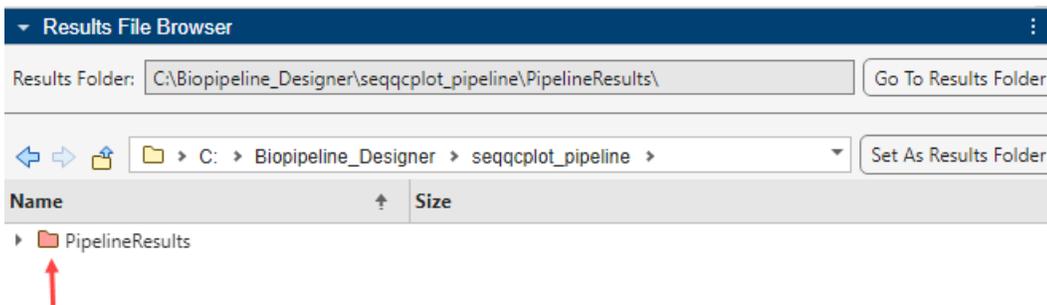


The **Results File Browser** pane allows you to navigate, view, and access the output files generated by the pipeline. You can expand the results folder for each block to view the corresponding output files.

Expand **SeqFilter_1 > 1** to see the FASTQ file containing the output sequences of the **SeqFilter_1** block.

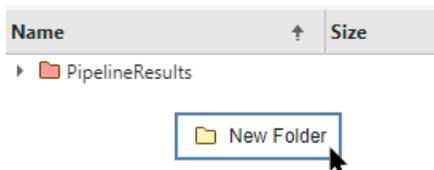
Name	↑	Size
▼ PipelineResults		
▶ FASTQ		
▼ SeqFilter_1		
▼ 1		
blockResults.mat		1 KB
SRR005164_1_50_filtered.fastq		8 KB
.block_run.ts		1 KB
runData.mat		6 KB
▶ SeqQCPlot		
LogFile.txt		4 KB

Click the up arrow  to go up one folder. The app uses a red folder to indicate the current results folder.

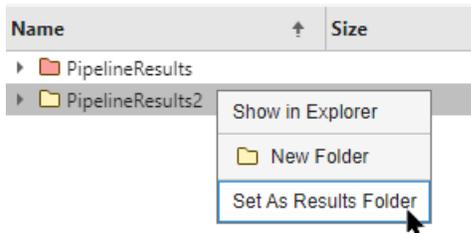


Clicking the **Go To Results Folder** button takes you directly to the current results folder if you are not in it.

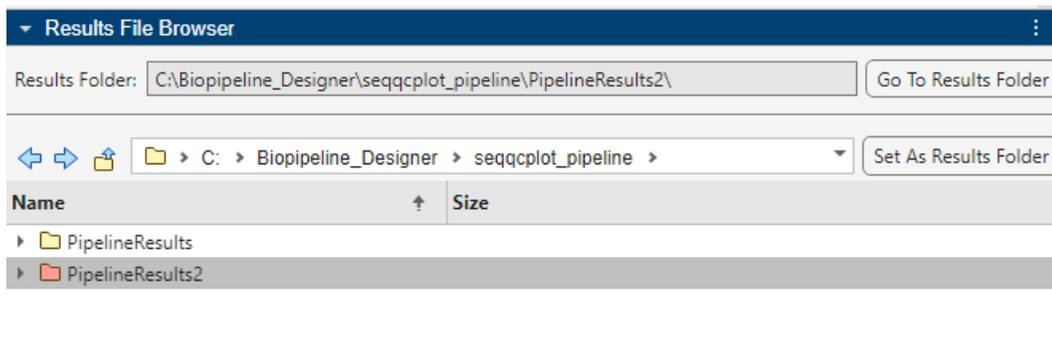
You can use the context (right-click) menu for additional options. For instance, create a new folder and set it as the new results folder. Right-click within the **Results File Browser** and select **New Folder**.



Click the folder once to rename it as **PipelineResults2**. Right-click the folder and select **Set As Results Folder**.

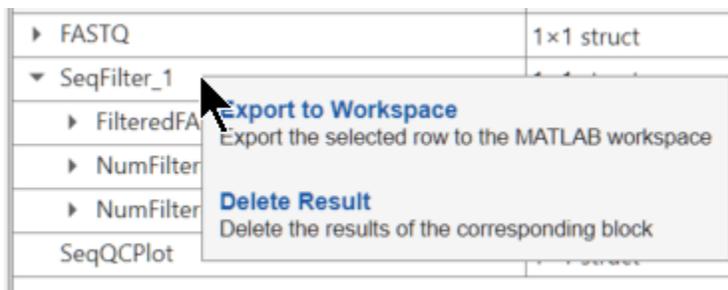


The color of **PipelineResults2** is changed to red to indicate that **PipelineResults2** is the current results folder, and the **Results Folder** row shows the newly-set results folder. The app saves any subsequent pipeline results in this folder.



Export Results

You can export each output of a block or every output of a block to the MATLAB workspace by selecting **Export to Workspace** from the context (right-click) menu of the corresponding row in the **Results** table. To export all outputs of a block, right-click at the block level.



See Also

Biopipeline Designer | “Bioinformatics Pipeline Run Mode” on page 2-114 | SeqFilterOptions

Related Examples

- “Count RNA-Seq Reads Using Biopipeline Designer” on page 2-127

Count RNA-Seq Reads Using Biopipeline Designer

This example shows how to build a bioinformatics pipeline to count the number of reads mapped to genes identified by Cufflinks using a sample paired-end RNA-Seq data for chromosome 4 of the *Drosophila* genome.

The example provides individual steps to build the pipeline from the beginning. To open the same completed pipeline, open the app. Then select **Open > Open Example Pipeline > Count RNA-Seq Reads**.

Open Biopipeline Designer App

At the MATLAB® command line, enter:

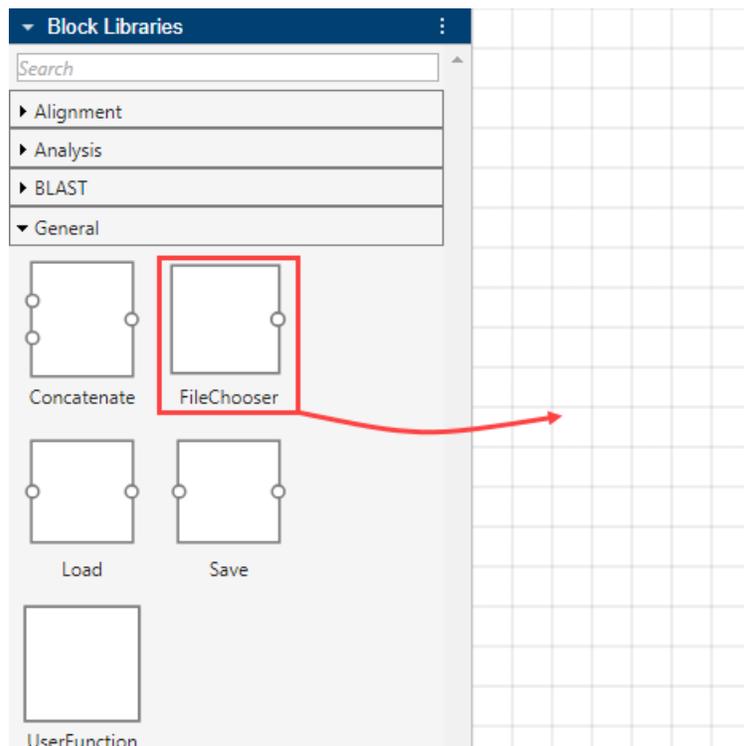
```
biopipelineDesigner
```

Select Data Files

The example uses chromosome 4 of the *Drosophila* genome as a reference (`Dmel_chr4.fa`). It also uses a sample paired-end data provided with the toolbox (`SRR6008575_10k_1.fq` and `SRR6008575_10k_2.fq`). Use a FileChooser block for each of these files to load the data into the app.

Create FileChooser Block for Reference Sequence

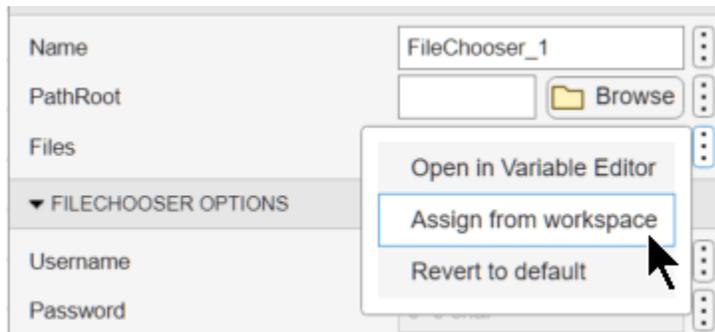
In the **Block Library** pane of the app, scroll down to the **General** section. Drag and drop a **FileChooser** block onto the diagram.



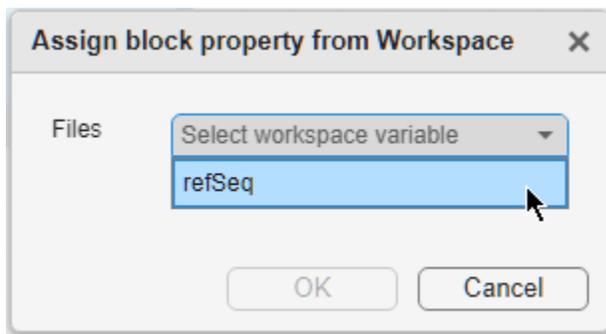
Run the following command at the MATLAB command line to create a variable that contains the full file path to the provided reference sequence.

```
refSeq = which("Dmel_chr4.fa");
```

In the app, click the **FileChooser** block. In the **Pipeline Inspector** pane, under **FileChooser Properties**, click the vertical three-dot menu next to the **Files** property. Select **Assign from workspace**.



Select **refSeq** from the list. Click **OK**.



Create FileChooser Blocks for Paired-End Data

There are two sample files (SRR6008575_10k_1.fq and SRR6008575_10k_2.fq) provided with the toolbox that contain RNA-Seq data for pair-end reads. You need to create a **FileChooser** block for each file.

First, run the following commands at the MATLAB command line to create two variables that contain the full file path to the provided files.

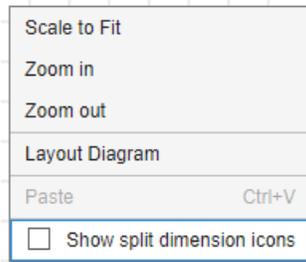
```
reads1 = which("SRR6008575_10k_1.fq");
reads2 = which("SRR6008575_10k_2.fq");
```

In the app, drag and drop two **FileChooser** blocks. Click **FileChooser_2** and set its **Files** property to the reads1 variable and reads2 for **FileChooser_3** (following the similar steps you did for the reference sequence refSeq previously).

Disable SplitDimension Icons Display

By default, the app displays dedicated icons for the split dimension settings of the input ports of your pipeline blocks. For the purposes of the example, disable the icon display. For more details about the icons and their meanings, see "Show split dimensions in Biopipeline Designer" on page 2-111.

Right-click in an empty area of the diagram and clear **Show split dimensions icons**. It would look as follows afterwards.

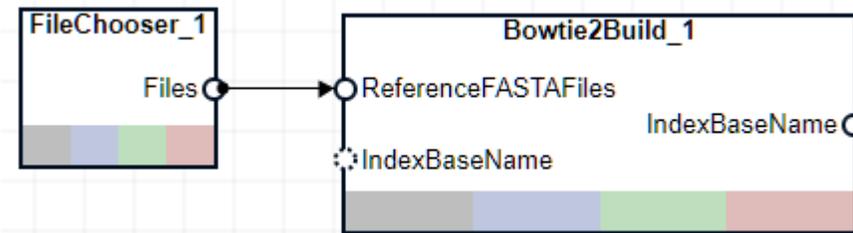


Build Reference Genome Indices

Generate indices for the reference genome files before aligning the reads to it. Use a **Bowtie2Build** block to build such indices.

From the **Block Library** pane, under the **Alignment** section, drag and drop the **Bowtie2Build** block onto the diagram.

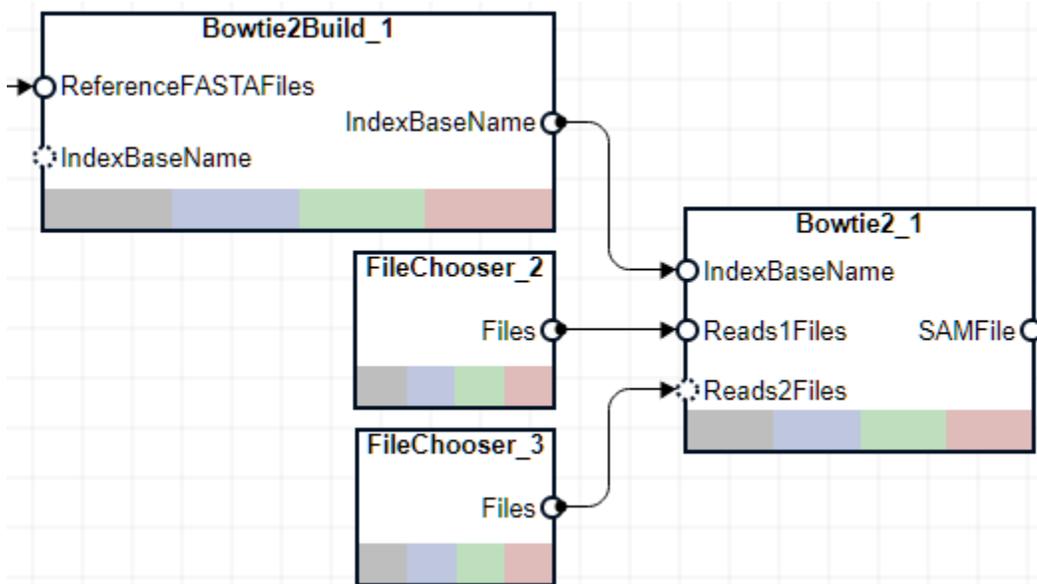
Connect **FileChooser_1** to **Bowtie2Build_1** blocks as shown next. To connect, place your pointer at the output port of the first block and drag (an arrow) towards the input port of the second block.



Align Reads Using Bowtie2

Use a **Bowtie2** block as an aligner to map reads from two sample files (**FileChooser_2** and **FileChooser_3**) against the reference sequence.

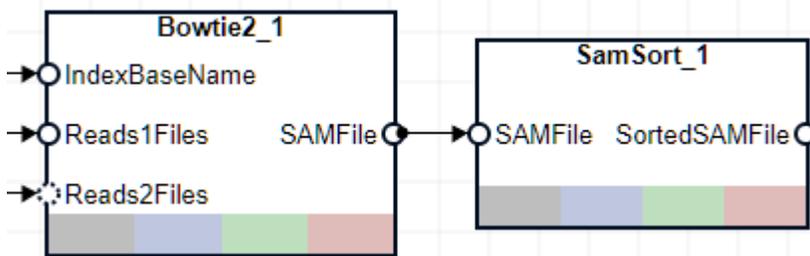
Drag and drop a **Bowtie2** block from the **Block Library** pane. The **IndexBaseName** input port of the block takes in the base name of the index files, which is the output of the previous (or upstream) **Bowtie2Build_1** block. The **Reads1Files** and **Reads2Files** input ports takes in the first mate and second mate reads, respectively. The **IndexBaseName** and **Reads1Files** input ports are required and must be connected, as indicated by solid circles. The **Reads2File** port is an optional port, indicated by a dotted circle, and you use it only when you have paired-end data, such as in this example. Connect these three blocks as shown next. The **Bowtie2 Options** section of the **Pipeline Inspector** pane lists all the available alignment options. For details on each options, see [Bowtie2AlignOptions](#).



Sort SAM Files

The next step is to use the **SamSort** block, which sorts the alignment records by the reference sequence name first and then by position within the reference. Sorting is needed because you will use the **Cufflinks** block next to assemble transcriptomes based on the aligned reads, and the block requires sorted SAM files as inputs.

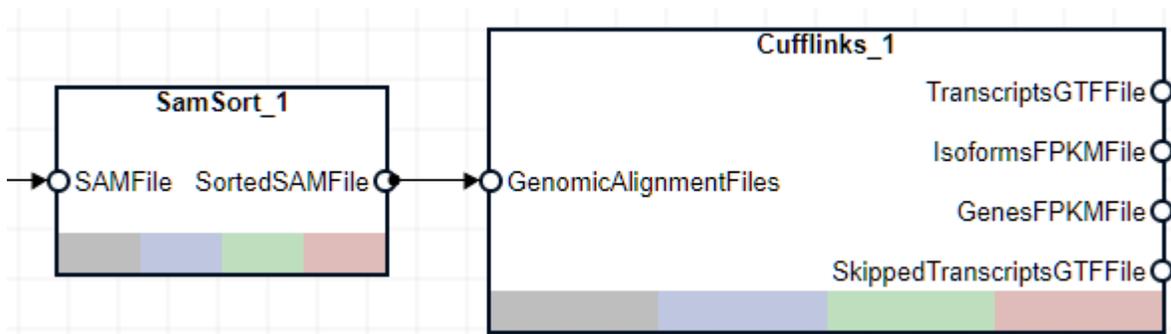
Drag and drop a **SamSort** block from the **Sequence Utilities** section of **Block Library** onto the diagram. Then connect the **Bowtie2_1** and **SamSort_1** blocks.



Assemble Reads into Transcriptomes

Create a GTF (Gene Transfer Format) file from the aligned data (SAM files) to quantify transcript expression. Use the **Cufflinks** block to assemble the sorted SAM files into GTF files, which contains information on gene features, including the start and end positions of transcripts.

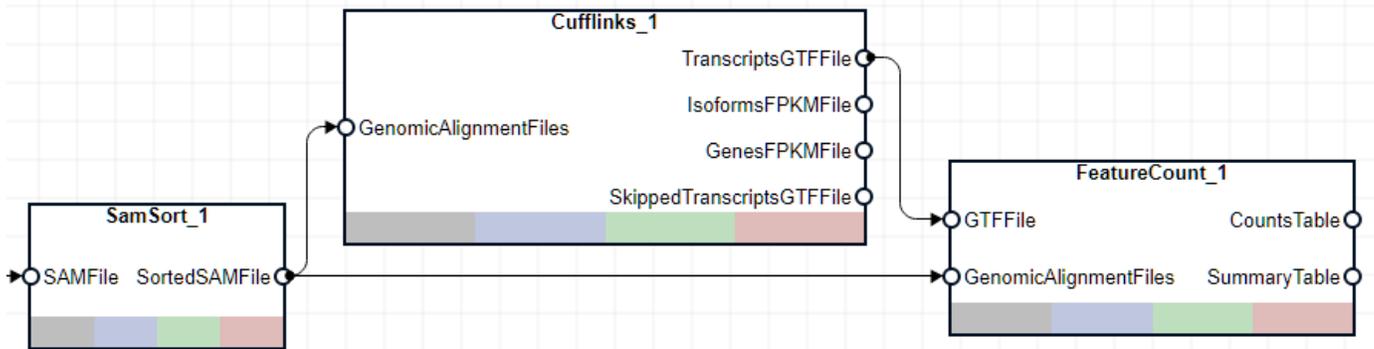
Drag and drop a **Cufflinks** block from the **Analysis** section of **Block Library** onto the diagram. Then connect the **SamSort_1** and **Cufflinks_1** blocks.



Count Reads from Paired-End Data

Use the **FeatureCount** block to count the number of reads in the sorted SAM file that map onto genomic features in the GTF file (**TranscriptsGTFFile**) generated by the **Cufflinks** block. Specifically, you will count the number of reads mapped to genes identified by **Cufflinks**.

Drag and drop a **FeatureCount** block from the **Analysis** section of **Block Library**. Then connect the three blocks (**SamSort_1**, **Cufflinks_1**, and **FeatureCount_1**) as shown next:



Plot Read Counts

As the last step in this pipeline, create a custom function that plots read count results for cufflinks-identified genes.

Go back to the MATLAB desktop. On the **Home** tab, click **New Script**. An untitled file opens in the **Editor**. Copy and paste the following code in the file that defines a custom function called `plotCounts`. The function generates two plots. The first plot contains the read counts of each gene identified by **Cufflinks**. The second plot shows the genomic locations of these counts.

```
function plotCounts(fcCountsTable,cufflinksGenesFPKMFile)
    genesFPKMTable = readtable(cufflinksGenesFPKMFile,FileType="text");
    % Plot counts of genes identified by Cufflinks.
    figure
    geneNames = categorical(fcCountsTable.ID,fcCountsTable.ID);
    stem(geneNames, log2(fcCountsTable.Aligned_sorted))
    xlabel("Cufflinks-identified genes")
    ylabel("log2 counts")

    % Plot counts along their respective genomic positions.
    geneStart = str2double(extractBetween(genesFPKMTable.locus,":","-"));
```

```

figure
stem(geneStart,log2(fcCountsTable.Aligned_sorted))
xlabel("Drosophila Chromosome 4 Genomic Position")
ylabel("log2 counts")
end

```

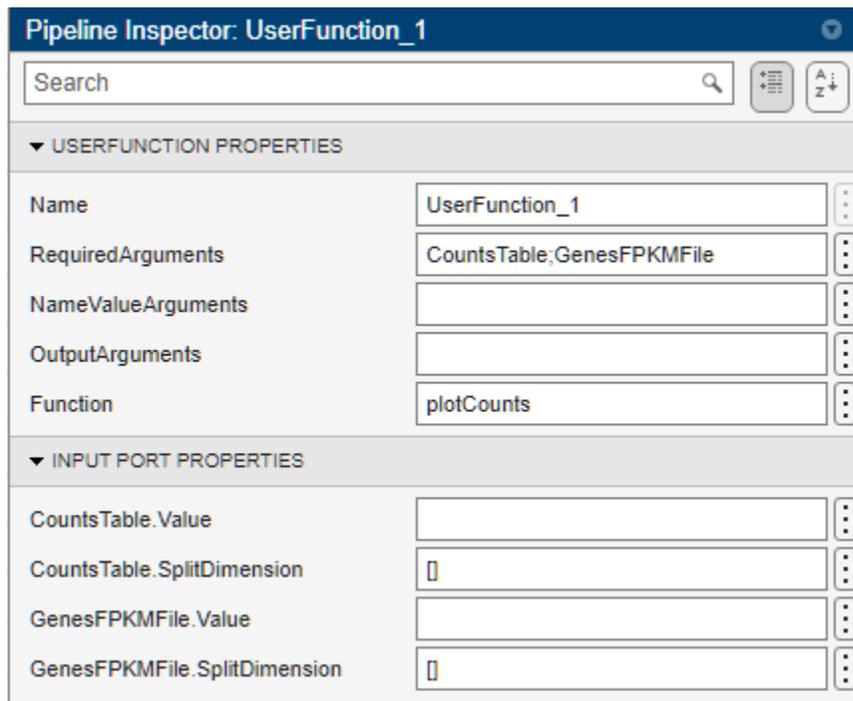
Save the file as `plotCounts.m` in the current directory.

Create UserFunction Block to Represent Custom Function

A **UserFunction** block can represent any existing or custom function and can be used as a block in your pipeline.

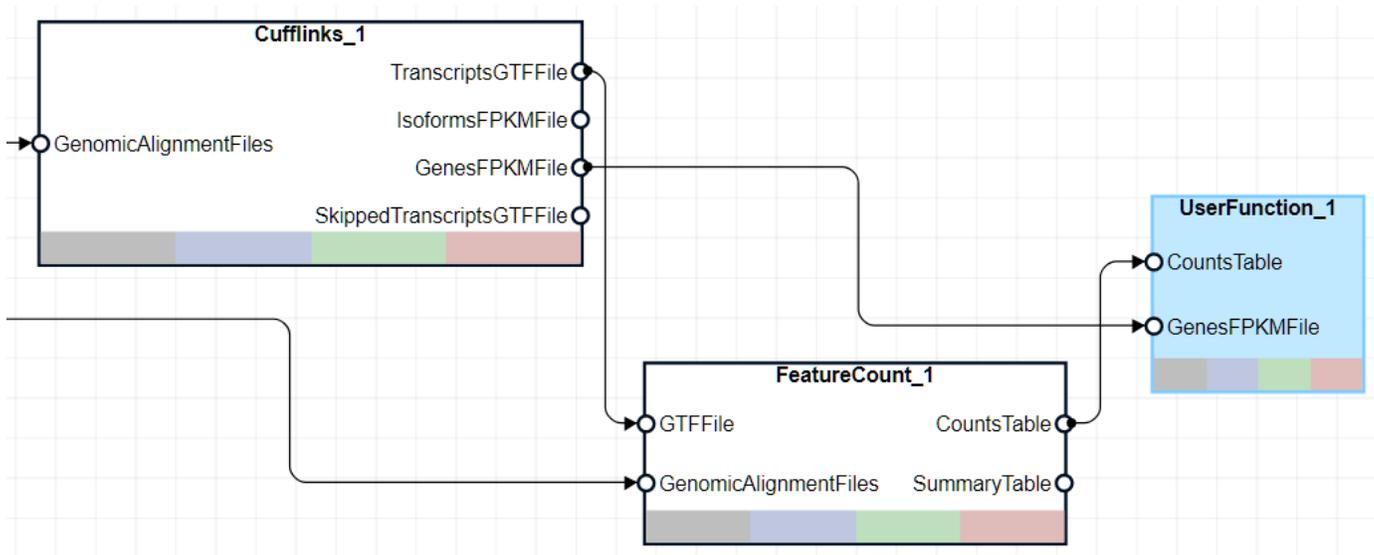
Drag and drop a **UserFunction** block from **Block Library**. In the **Pipeline Inspector** pane, under **UserFunction Properties**, update:

- **RequiredArguments** to `CountsTable, GenesFPKMFile`
- **Function** to `plotCounts`



The **UserFunction_1** block is then updated with two input ports named after the values of **RequiredArguments**.

Connect three blocks (**Cufflinks_1**, **FeatureCount_1**, and **UserFunction_1**) as shown next.



Tip

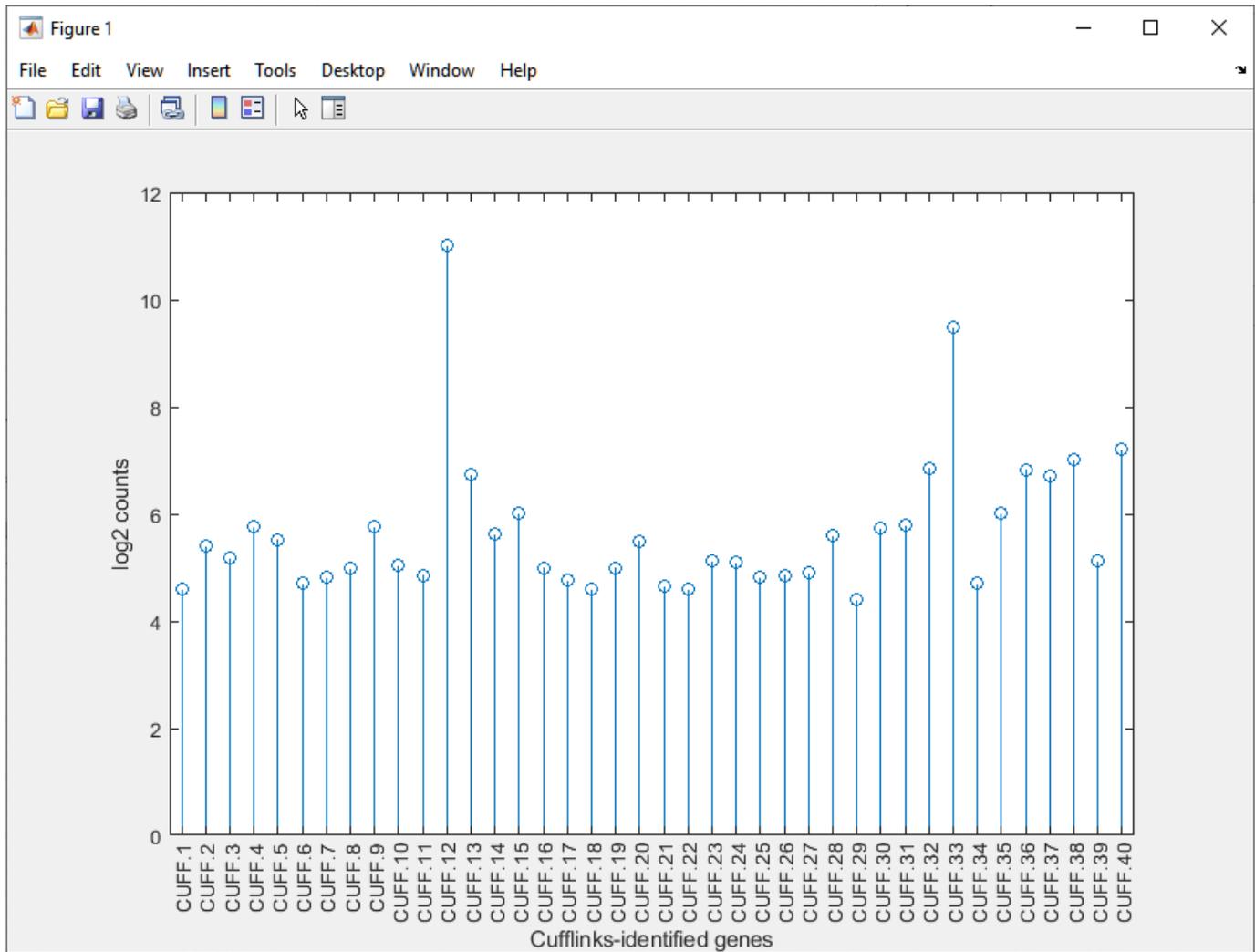
You can edit the underlying function definition of your **UserFunction** block in the MATLAB® editor. There are two ways.

- Right-click the block and select **Edit Function** from the context menu.
- From the **Pipeline Inspector** pane, click the three-dot menu next to the **Function** property. Select **Edit Function**.

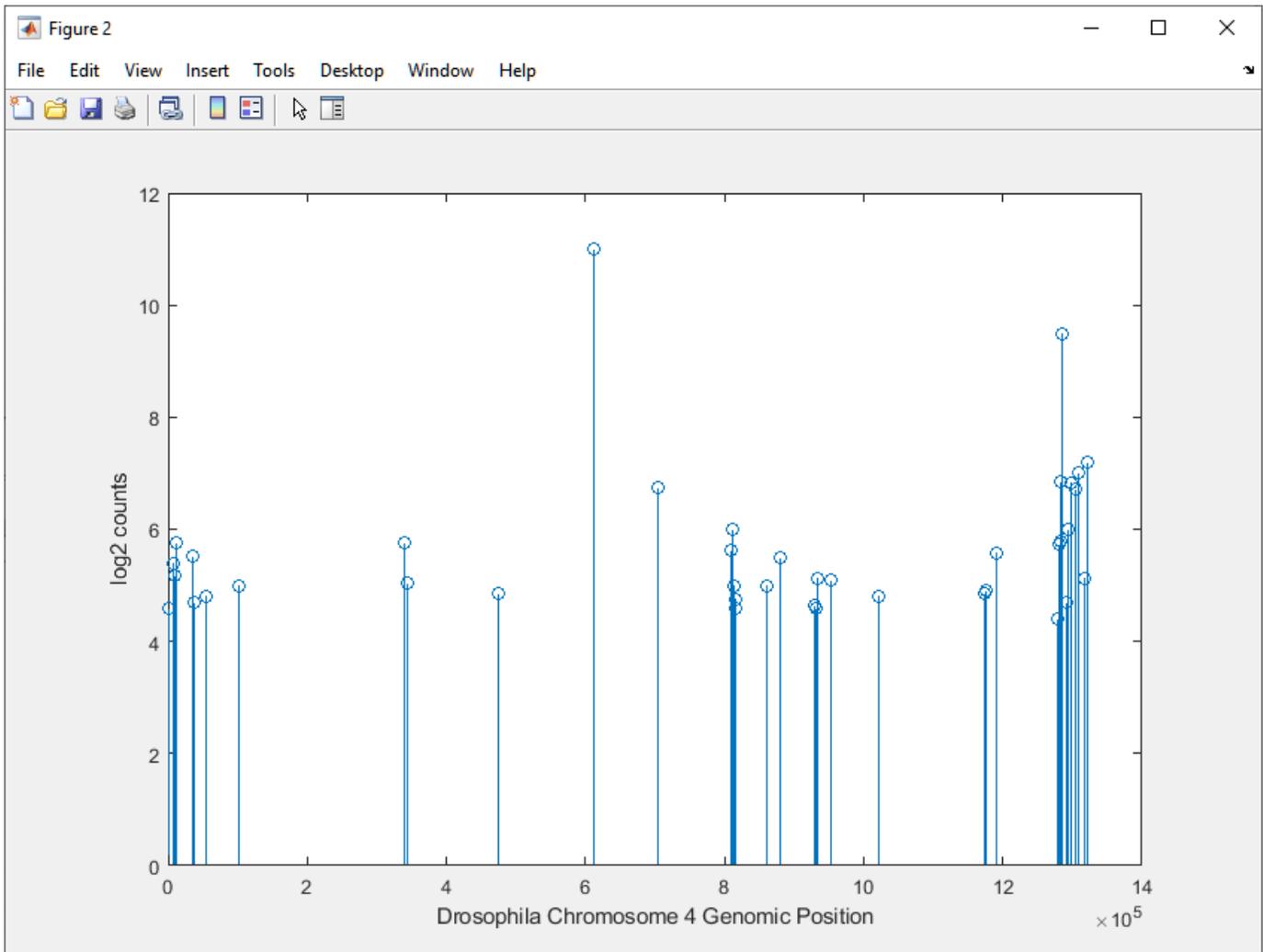
Run Pipeline

Click **Run** on the **Home** tab of the app. The app generates the following two figures.

The first figure shows the log₂ counts of each gene identified by **Cufflinks**.



The second figure shows the individual genomic locations of these counts.



See Also

[bioinfo.pipeline.Pipeline](#) | [bioinfo.pipeline.block.Cufflinks](#) |
[bioinfo.pipeline.block.Bowtie2](#) | [bioinfo.pipeline.block.Bowtie2Build](#) |
[bioinfo.pipeline.block.FeatureCount](#) | [bioinfo.pipeline.block.SamSort](#) |
[bioinfo.pipeline.block.FileChooser](#) | [bioinfo.pipeline.block.UserFunction](#)

Related Examples

- “Create Simple Pipeline to Plot Sequence Quality Data Using Biopipeline Designer” on page 2-115

Bioinformatics Pipeline Block Libraries

Block libraries are collections of built-in and custom blocks that you can use in your analysis pipelines.

The Biopipeline Designer app categorizes the built-in blocks into different libraries. You can see these libraries and the corresponding blocks in the **Block Libraries** pane of the app.

HOME

New Open Save Compile Export Layout Undo Redo Run

PROJECT PIPELINE UNDO

Block Libraries

Search

Alignment

Bowtie2 Bowtie2Build BwaIndex BwaMEM

Analysis

CuffCompare CuffDiff CuffMerge CuffNorm

CuffQuant Cufflinks FeatureCount GenomicsVie...

BLAST

General

SRA

Sequence_Uilities

Results File Browser

You can also create custom blocks or configure existing blocks and save them in custom libraries to reuse them in your analysis pipelines. For details, see “Create and Save Blocks in Block Libraries Using Biopipeline Designer”.

If you are using the command line interface, you can create a built-in block using `bioinfo.pipeline.block.blockName`, where *blockName* is the name of a built-in block. To create a custom library at the command line, use `bioinfo.pipeline.library.Library`.

Modifications to block libraries made in the app are reflected in the command line or vice versa.

See Also

`bioinfo.pipeline.library.Library` | Biopipeline Designer

Analyze Gene Expression Profiles Using Biopipeline Designer

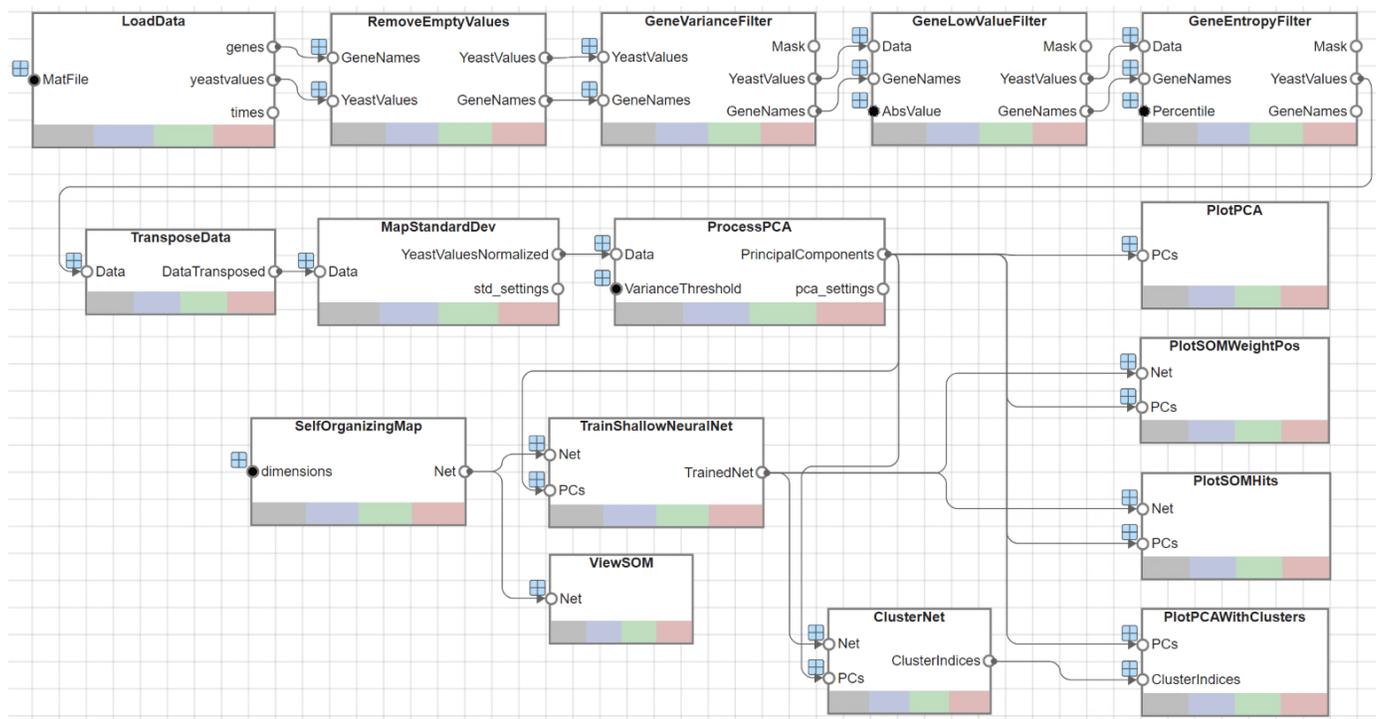
This example shows how to look for patterns in gene expression profiles using neural networks, facilitated with the Biopipeline Designer app. The example requires Deep Learning Toolbox™ and Bioinformatics Toolbox™.

Specifically, the example analyzes gene expression changes in *Saccharomyces cerevisiae* (baker's yeast) during the diauxic shift when yeast switches from fermenting glucose to respiring ethanol after glucose is depleted. DNA microarray data is used to study how nearly all yeast genes change their expression over time during this metabolic transition. For additional details on this process, see “Gene Expression Profile Analysis” on page 4-95.

Enter the following command to open the prebuilt pipeline in Biopipeline Designer.

```
openExample('bioinfo/GeneExpressionAnalysisExample.m')
```

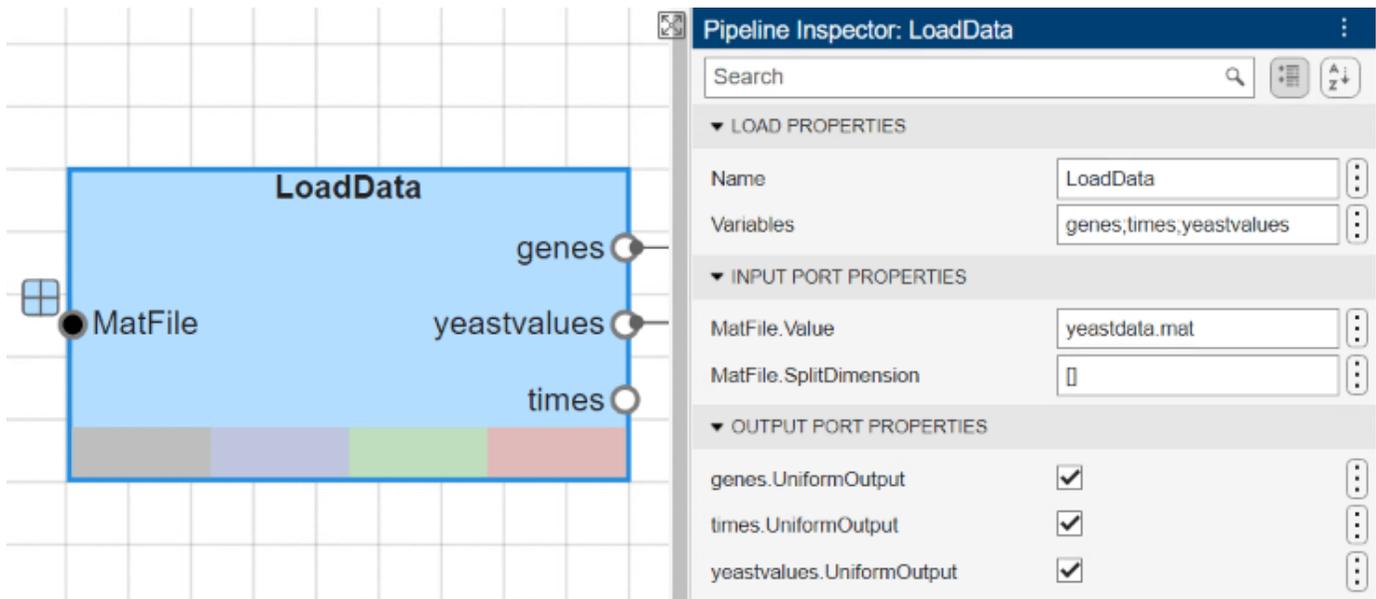
The app opens as shown next.



Load Data

This example uses data from [1]. The full data set can be downloaded from the Gene Expression Omnibus website: <https://www.yeastgenome.org>

The pipeline starts by first loading data into MATLAB® using a built-in **Load** block named *LoadData*. The data file yeastdata.mat was set by entering the appropriate filename in the MAT-file input port value within the **Pipeline Inspector** pane.



Gene expression levels were measured at seven time points during the diauxic shift. The variable *times* contains the times at which the expression levels were measured in the experiment. The variable *genes* contains the names of the genes whose expression levels were measured. The variable *yeastvalues* contains the "VALUE" data or LOG₂ of ratio of CH2DN_MEAN and CH1DN_MEAN from the seven time steps in the experiment.

Filter Genes

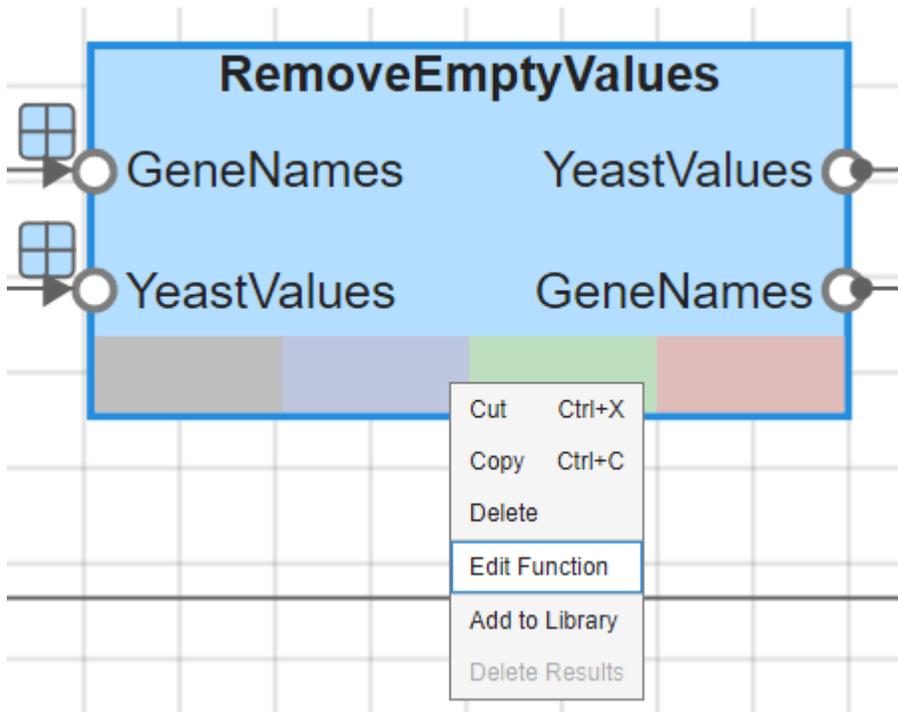
The data set is quite large and a lot of the information corresponds to genes that do not show any interesting changes during the experiment. To find the interesting genes, first, reduce the size of the data set by removing genes with expression profiles that do not show anything of interest. There are 6400 expression profiles. You can use a number of techniques to reduce this to some subset that contains the most significant genes.

The gene list contains several spots marked as 'EMPTY'. These are empty spots on the array, and while they might have data associated with them, for the purposes of this example, you can consider these points to be noise.

The *yeastvalues* variable also contains several expression level is marked as NaN. This indicates that no data was collected for this spot at the particular time step. One approach to dealing with these missing values would be to impute them using the mean or median of data for the particular gene over time. This example uses a less rigorous approach of simply throwing away the data for any genes where one or more expression level was not measured.

The **UserFunction** block *RemoveEmptyValues* removes all genes that are marked as 'EMPTY' or have NaN expression levels.

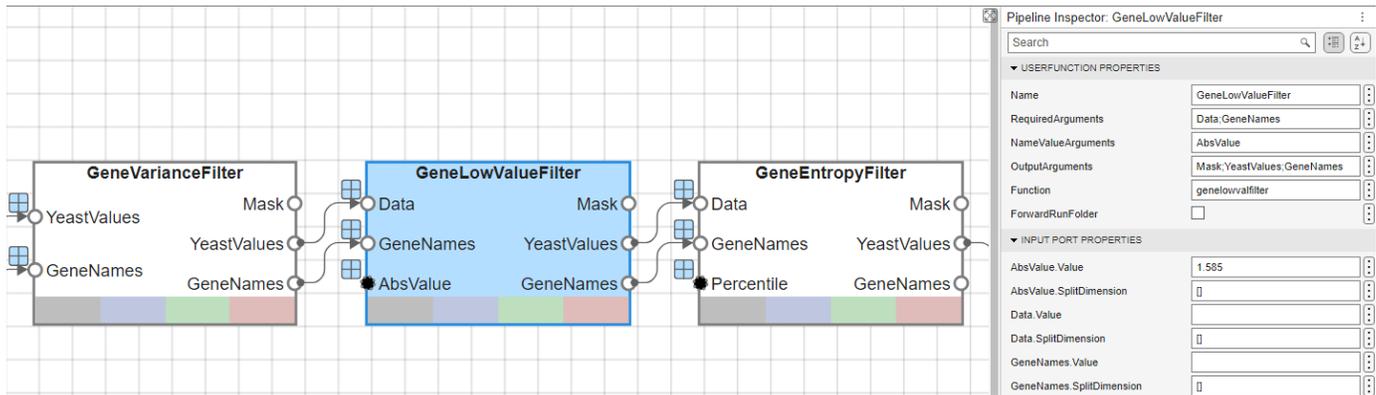
To view the custom function that this block uses, right-click the block and select **Edit Function**.



If you plot the expression profiles of all the remaining genes, you would see that most profiles are flat and not significantly different from the others. This flat data could still be useful because it indicates that these genes are not significantly affected by the diauxic shift. However, this example focuses on the genes with large changes in expression accompanying the diauxic shift. You can use filtering functions in Bioinformatics Toolbox to remove genes with various types of profiles that do not provide relevant information about genes affected by the metabolic change.

Use the `genevarfilter` function to filter out genes with small variance over time, the function `genelowvalfilter` to remove genes that have very low absolute value expression levels, and the `geneentropyfilter` to remove genes whose profiles have low entropy. These functions are each utilized within the **UserFunction** blocks named *GeneVarianceFilter*, *GeneLowValueFilter*, and *GeneEntropyFilter*, respectively.

The function that is used for each of these blocks can be accessed through the **Function** property of the corresponding **UserFunction** blocks. You can specify the values for *AbsVal* and *Percentile* in the **Pipeline Inspector** pane for the *GeneLowValueFilter* block and *GeneEntropyFilter* block, respectively. The next figure shows the *GeneLowValueFilter* block, where the threshold is defined as 1.585 within the **AbsValue.Value** text editor.



Perform Principal Component Analysis

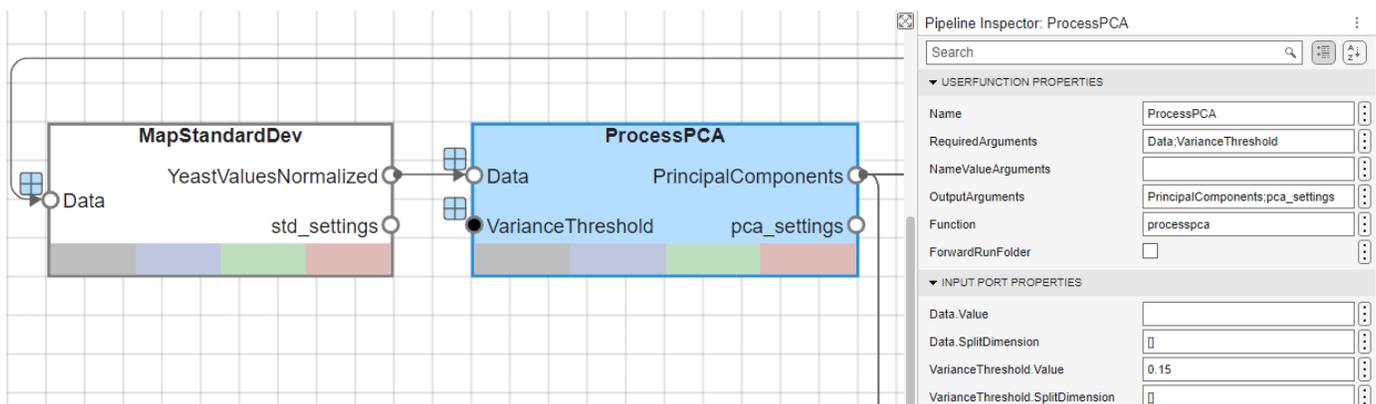
Now that you have a manageable list of genes, you can look for relationships between the profiles.

Normalizing the standard deviation and mean of data allows the network to treat each input as equally important over its range of values.

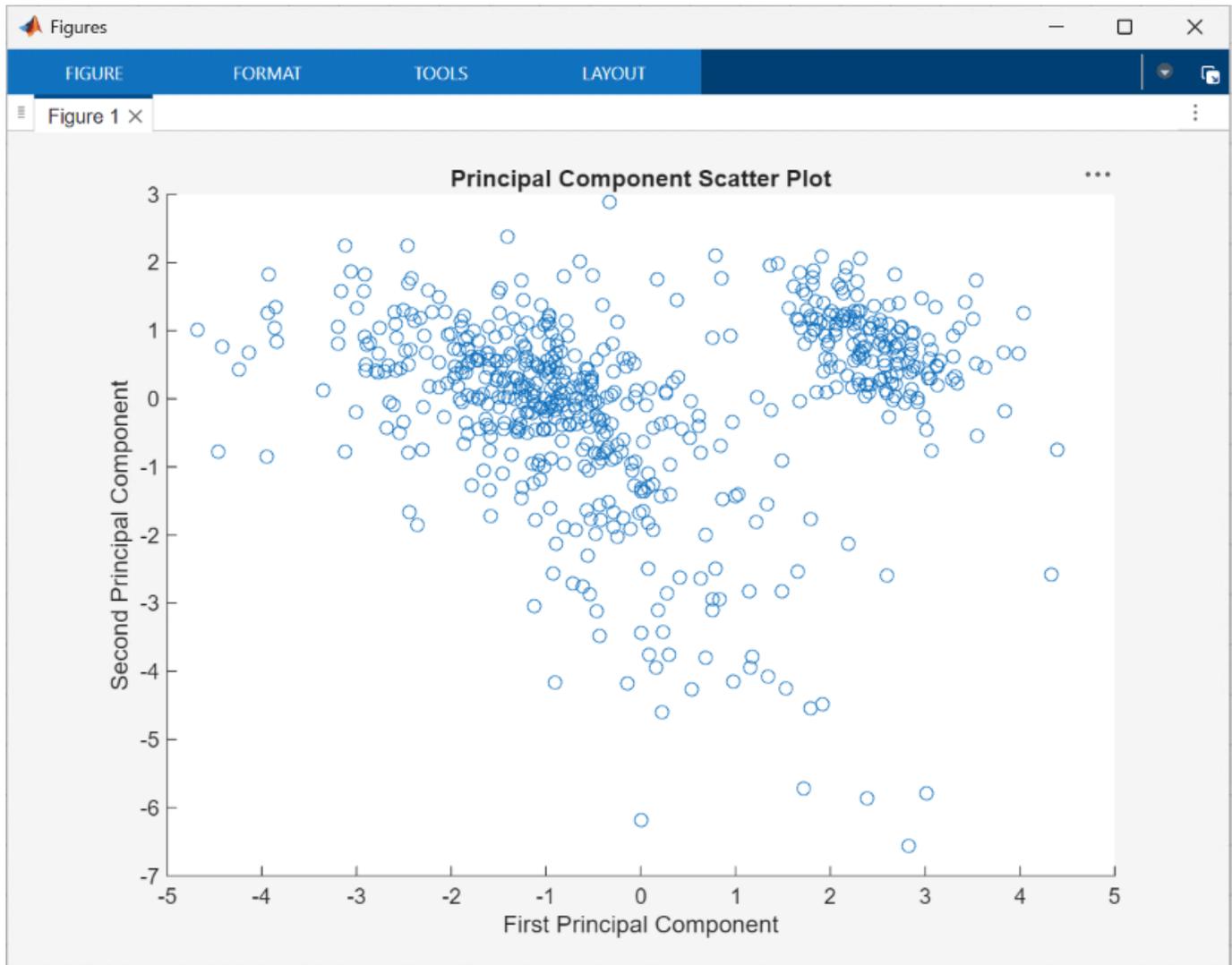
Principal-component analysis (PCA) is a useful technique that can be used to reduce the dimensionality of large data sets, such as those from microarray analysis. This technique isolates the principal components of the dataset eliminating those components that contribute the least to the variation in the data set.

The input vectors are first normalized, using `mapstd`, so that they have zero mean and unity variance. `processpca` is the function that implements the PCA algorithm. A variance threshold of 0.15 is provided to the `processpca` function, which means that `processpca` eliminates those principal components that contribute less than 15% to the total variation in the data set. The output of this block now contains the principal components of the *yeastvalues* data.

Similar to the filtering blocks above, these two blocks, *MapStandardDev* and *ProcessPCA*, are UserFunction blocks that call `mapstd` and `processpca`.



Visualize the principal components using the `scatter` function, specified in the **UserFunction** block *PlotPCA*, which generates the following figure:

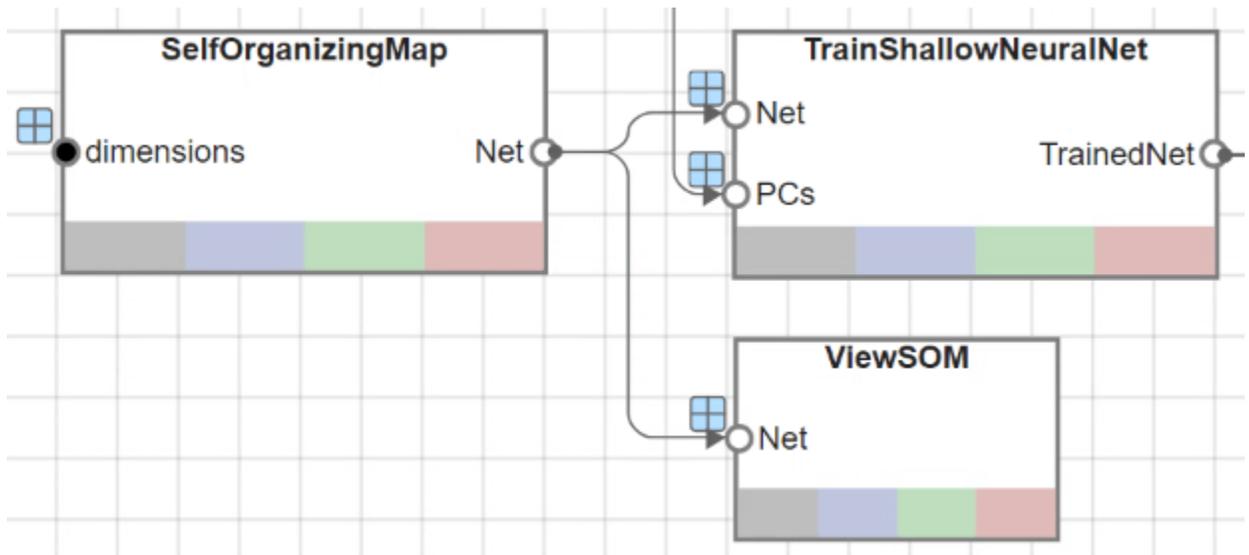


Use Self-Organizing Maps for Cluster Analysis

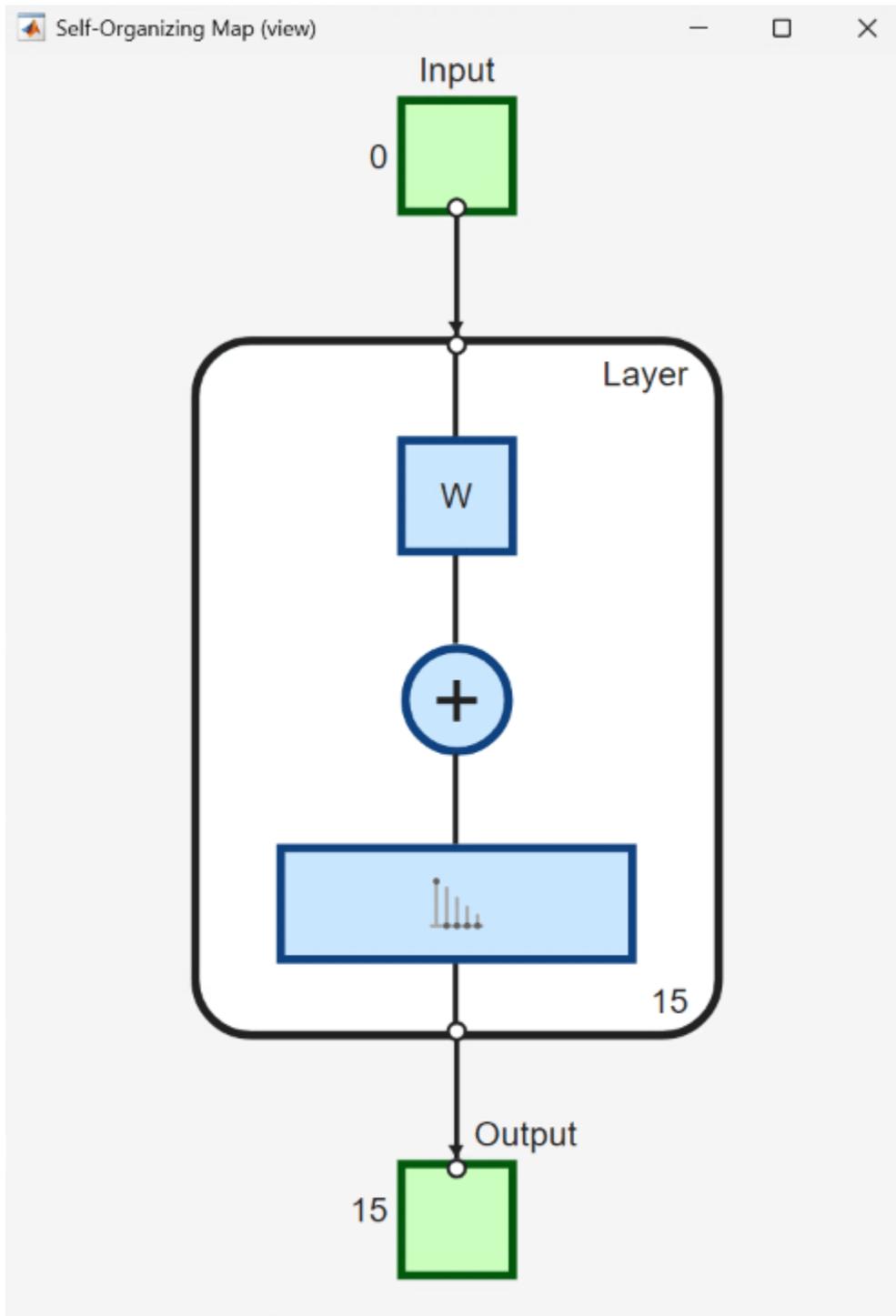
You can cluster the principal components using the Self-Organizing map (SOM) clustering algorithm.

The `selforgmap` function creates a Self-Organizing map network which can then be trained with the `train` function.

These functions are utilized through the **UserFunction** blocks named *SelfOrganizingMap* and *TrainShallowNeuralNet*, respectively.



The **UserFunction** block *ViewSOM* calls the `view` function on the generated SOM for an overview of the network's architecture.



To train the network, use the Neural Network Training tool, which shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training are highlighted in green.

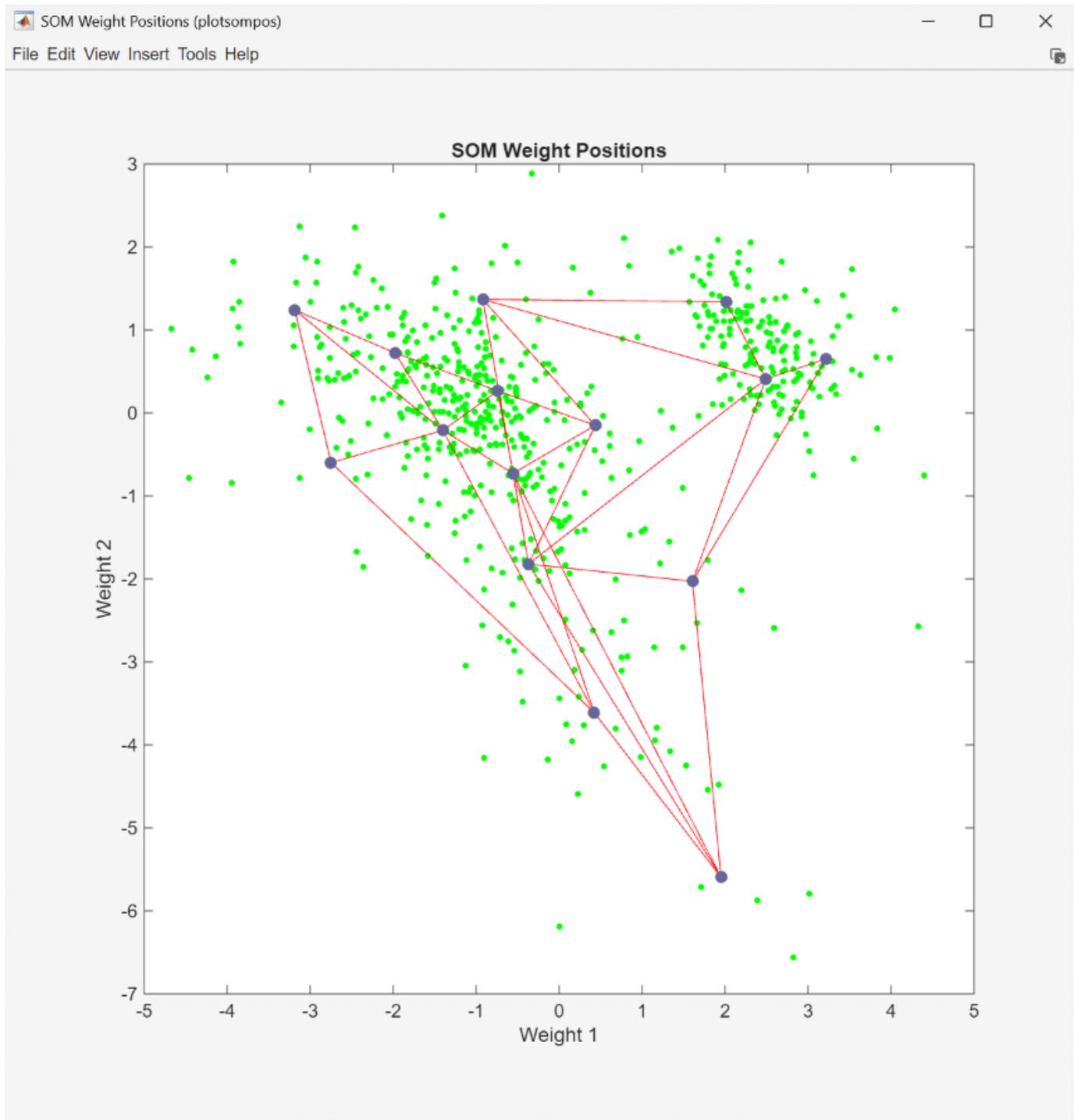
The **Training Plots** section of the tool provides various visualization plots that you can open during and after training.

The screenshot shows a window titled "Neural Network Training (11-Dec-2025 16:46:38)". At the top, there is a "Network Diagram" button. Below that is the "Training Results" section, which states "Training finished: Reached maximum number of epochs" with a green checkmark icon. The "Training Progress" section contains a table with the following data:

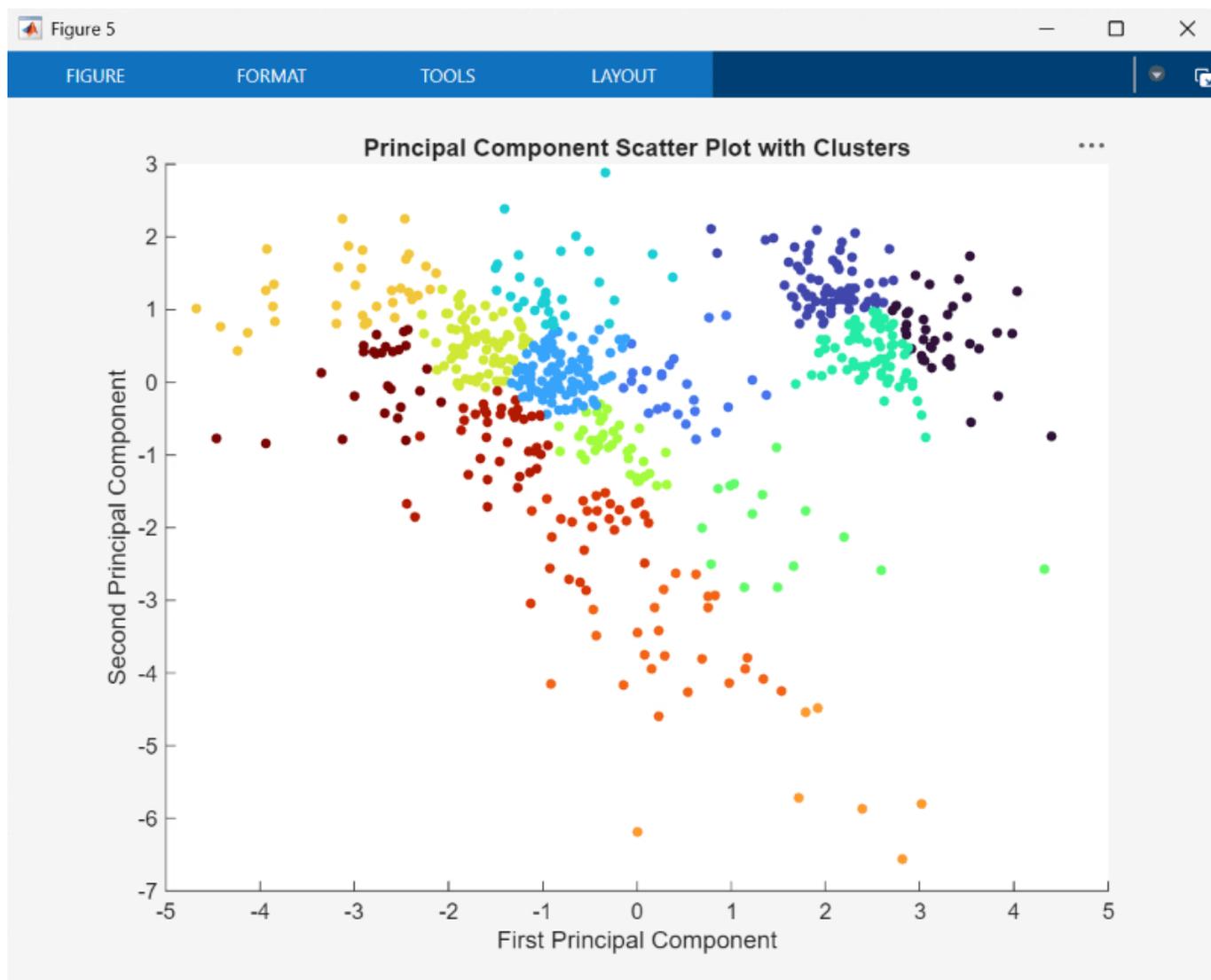
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200
Elapsed Time	-	00:00:04	-

Below the table is the "Training Algorithms" section, which lists: "Data Division: Batch Weight/Bias Rule trainbu", "Performance: Mean Squared Error mse", and "Calculations: MATLAB". The "Training Plots" section at the bottom contains six buttons: "SOM Topology", "SOM Neighbor Connections", "SOM Neighbor Distances", "SOM Input Planes", "SOM Sample Hits", and "SOM Weight Positions".

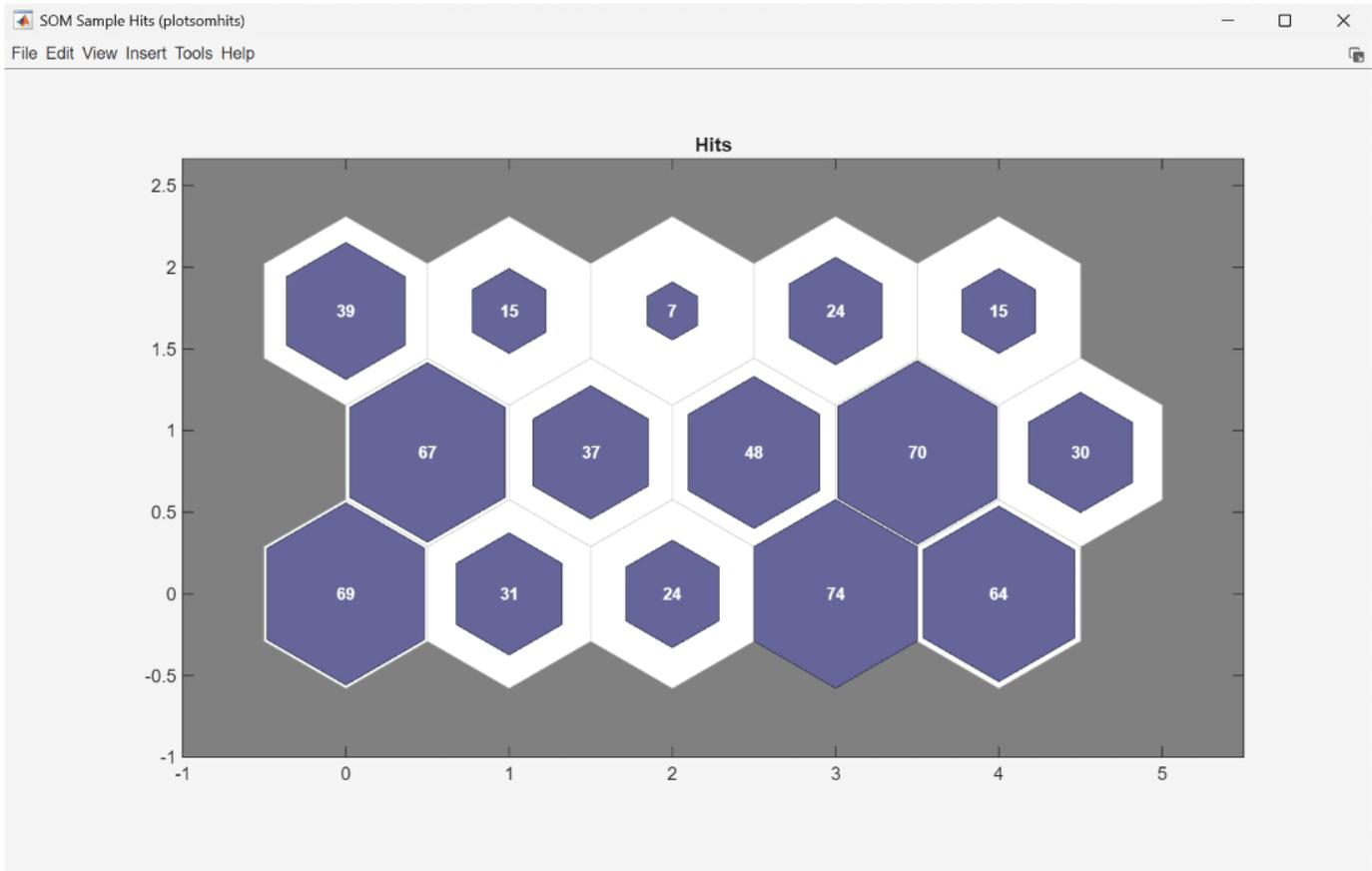
The **UserFunction** block *PlotSOMWeightPos* uses the `plotsompos` function to display the derived network over a scatter plot of the first two dimensions of the data.



You can also assign clusters using the SOM by finding the nearest node to each point in the data set. This is performed within the **UserFunction** block *ClusterNet* and resulting clusters are plotted by the block *PlotPCAWithClusters*.



Finally, use `plotsomhits` to see how many vectors are assigned to each of the neurons in the map, within the **UserFunction** block `PlotSOMHits`.



You can also use other clustering algorithms, such as Hierarchical clustering and K-means, available in the Statistics and Machine Learning Toolbox™ for cluster analysis.

References

[1] DeRisi, Joseph L., Vishwanath R. Iyer, and Patrick O. Brown. "Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale." *Science* 278, no. 5338 (1997): 680-86.

See Also

Biopipeline Designer | `bioinfo.pipeline.block.Load` |
`bioinfo.pipeline.block.UserFunction`

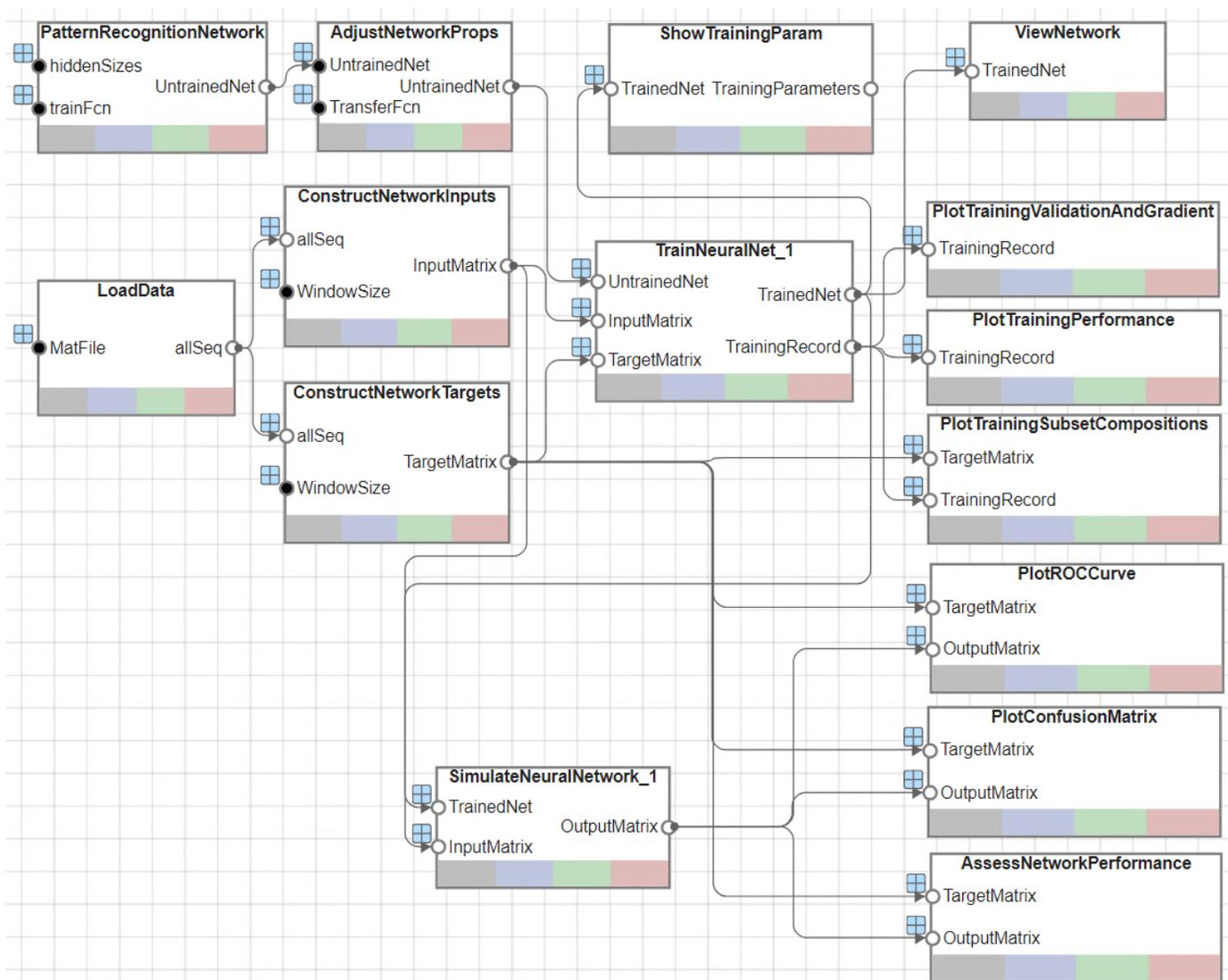
Predict Protein Secondary Structure Using Biopipeline Designer

This example uses a feed-forward neural network to predict protein secondary structures in Biopipeline Designer. Neural network models attempt to simulate the information processing that occurs in the brain and are widely used in a variety of applications, including automated pattern recognition. The example requires Deep Learning Toolbox™ and Bioinformatics Toolbox™.

Enter the following command to open the prebuilt pipeline in Biopipeline Designer.

```
openExample('bioinfo/ProteinStructurePredictionExample.m')
```

The app opens the pipeline as shown next.



Due to the random nature of some steps in the following approach, numeric results might be slightly different every time the network is trained or a prediction is simulated. For reproducible results, set the global random generator using the `rng` function before running the pipeline.

```
rng(0)
```

Load Data and Set Random Seed

The example uses the Rost-Sander dataset [1] that consists of proteins whose structures span a relatively wide range of domain types, composition, and length. The file `RostSanderDataset.mat` contains a subset of this data, where the structural assignment of every residue is reported for each protein sequence.

The block `LoadData` loads the data and specifically extracts the variable `allSeq` from the `RostSanderDataset.mat` file. This block is a built-in **Load** Block, and you can customize the block properties in the **Pipeline Inspector** pane.

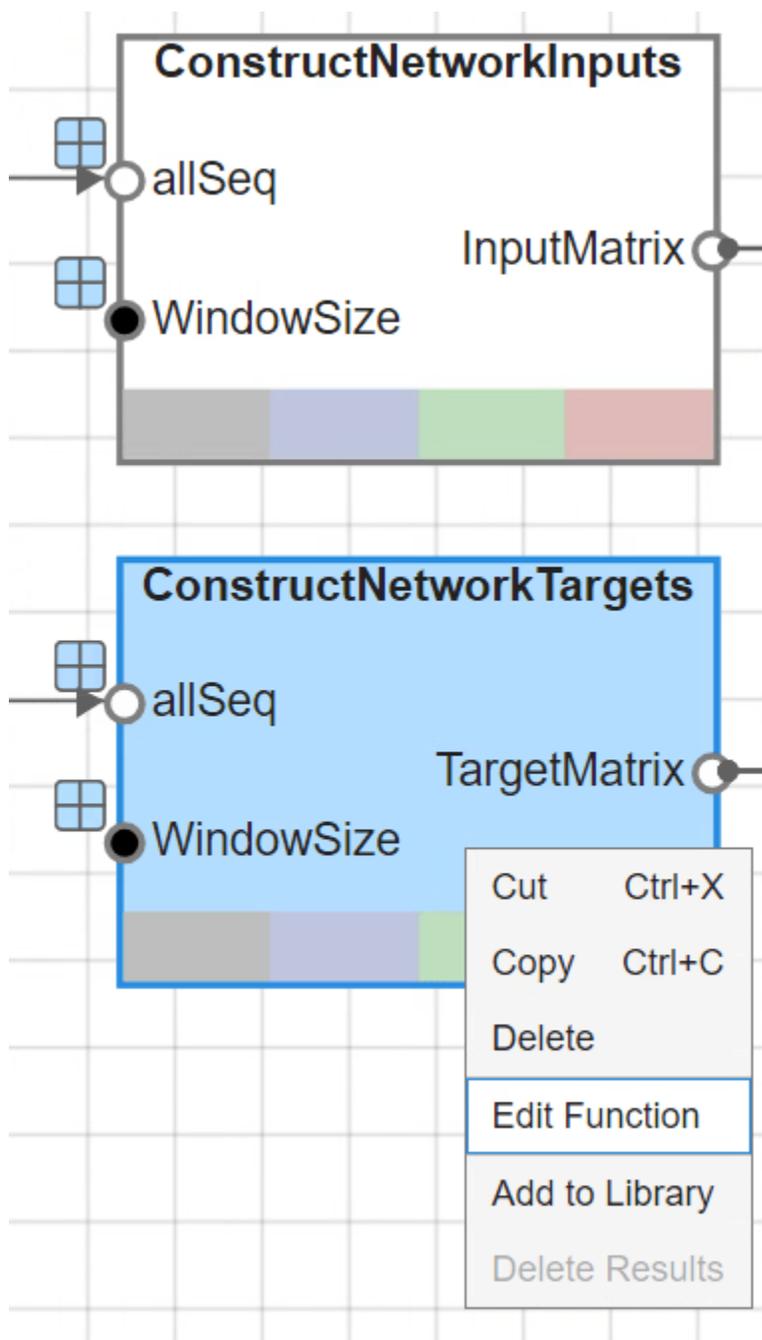
The image shows the Pipeline Inspector for the `LoadData` block. The left pane displays the block with an input port labeled `MatFile` and an output port labeled `allSeq`. The right pane, titled "Pipeline Inspector: LoadData", contains the following configuration:

LOAD PROPERTIES	
Name	LoadData
Variables	allSeq
INPUT PORT PROPERTIES	
MatFile.Value	RostSanderDataset.mat
MatFile.SplitDimension	[]
OUTPUT PORT PROPERTIES	
allSeq.UniformOutput	<input checked="" type="checkbox"/>

Define Network Architecture

This example builds a neural network to learn the structural state (helix, sheet, or coil) of each residue in a given protein, based on the structural patterns observed during a training phase. In this pipeline, the input and output layers, that is, the input and target matrices, are defined within the **UserFunction** blocks named `ConstructNetworkInputs` and `ConstructNetworkTargets`, respectively.

To view the underlying custom functions of these blocks, right-click either block and select **Edit Function**. The function definition is then shown in the MATLAB editor.



Build Neural Network

The problem of secondary structure prediction is similar to a pattern recognition problem, where you can train the network to recognize the structural state of the central residue most likely to occur when specific residues in the given sliding window are observed.

Create a pattern recognition neural network using the input and target matrices defined above and specify a hidden layer of particular size. The pipeline uses the size of 20, which you can modify in the **Pipeline Inspector** pane after selecting the *PatternRecognitionNetwork* block.

The image shows a screenshot of the Biopipeline Designer interface. On the left, a block named **PatternRecognitionNetwork** is displayed on a grid. It has two input ports on the left labeled **hiddenSizes** and **trainFcn**, and one output port on the right labeled **UntrainedNet**. The block is divided into four colored segments: grey, purple, green, and red. On the right, the **Pipeline Inspector: PatternRecognitionNetwork** panel is open, showing the following properties:

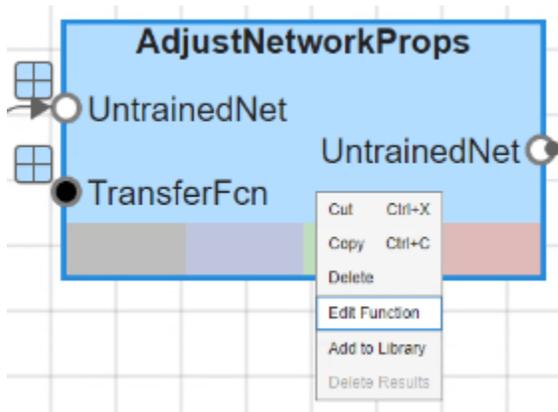
USERFUNCTION PROPERTIES	
Name	PatternRecognitionNetwork
RequiredArguments	hiddenSizes;trainFcn
NameValueArguments	
OutputArguments	UntrainedNet
Function	patternnet
ForwardRunFolder	<input type="checkbox"/>
INPUT PORT PROPERTIES	
hiddenSizes.Value	20
hiddenSizes.SplitDimension	[]
trainFcn.Value	traincgb
trainFcn.SplitDimension	[]
OUTPUT PORT PROPERTIES	
UntrainedNet.UniformOutput	<input checked="" type="checkbox"/>

This block uses the built-in MATLAB function `patternnet` to build the neural network as defined in the Function property shown in the image above.

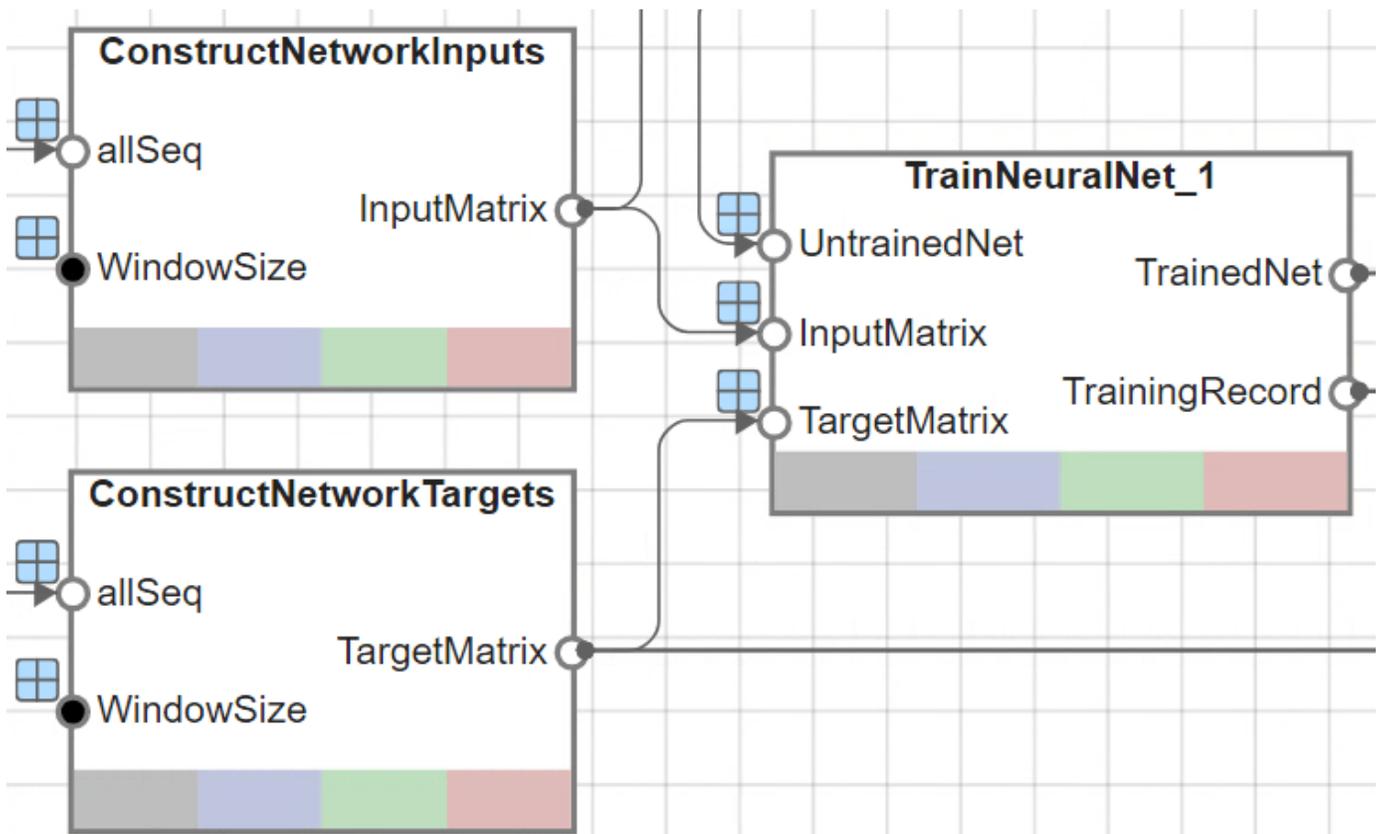
Train Neural Network

The pattern recognition network uses the default scaled conjugate gradient algorithm for training, but other algorithms are available in Deep Learning Toolbox. You can modify the training function by adjusting the property value for the `trainFcn` input port seen in the **Pipeline Inspector** pane of the *PatternRecognitionNetwork* block.

This network uses the `logsig` transfer function between the input and hidden layers to produce an output signal that is between and close to either 0 or 1, simulating the firing of a neuron [2]. For details on the network training, see **Training the Neural Network** in “Predicting Protein Secondary Structure Using a Neural Network” on page 3-107. You can modify which transfer function is used by updating the code within the **UserFunction** block named *AdjustNetworkProps*.



The **UserFunction** block named *TrainNeuralNet* leverages the built-in *train* function to train the neural network on the input data. The input port *Net* is the *patternnet* created earlier by the block *PatternRecognitionNetwork*, after the transfer function has been defined in *AdjustNetworkProps*.



You can also modify the number of hidden layers or the size of the hidden layer could be modified in the **Pipeline Inspector** pane for the *PatternRecognitionNetwork* block.

During training, the training tool window opens and displays the progress.

Neural Network Training (11-Dec-2025 15:58:46)

Network Diagram

Training Results

Training finished: Met validation criterion 

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	52	1000
Elapsed Time	-	00:00:11	-
Performance	0.46	0.268	0
Gradient	0.518	0.00729	1e-10
Validation Checks	0	6	6
Step Size	100	0.215	1e-06

Training Algorithms

Data Division: Random dividerand

Training: Conjugate Gradient with Beale-Powell Restarts trai...

Performance: Cross Entropy crossentropy

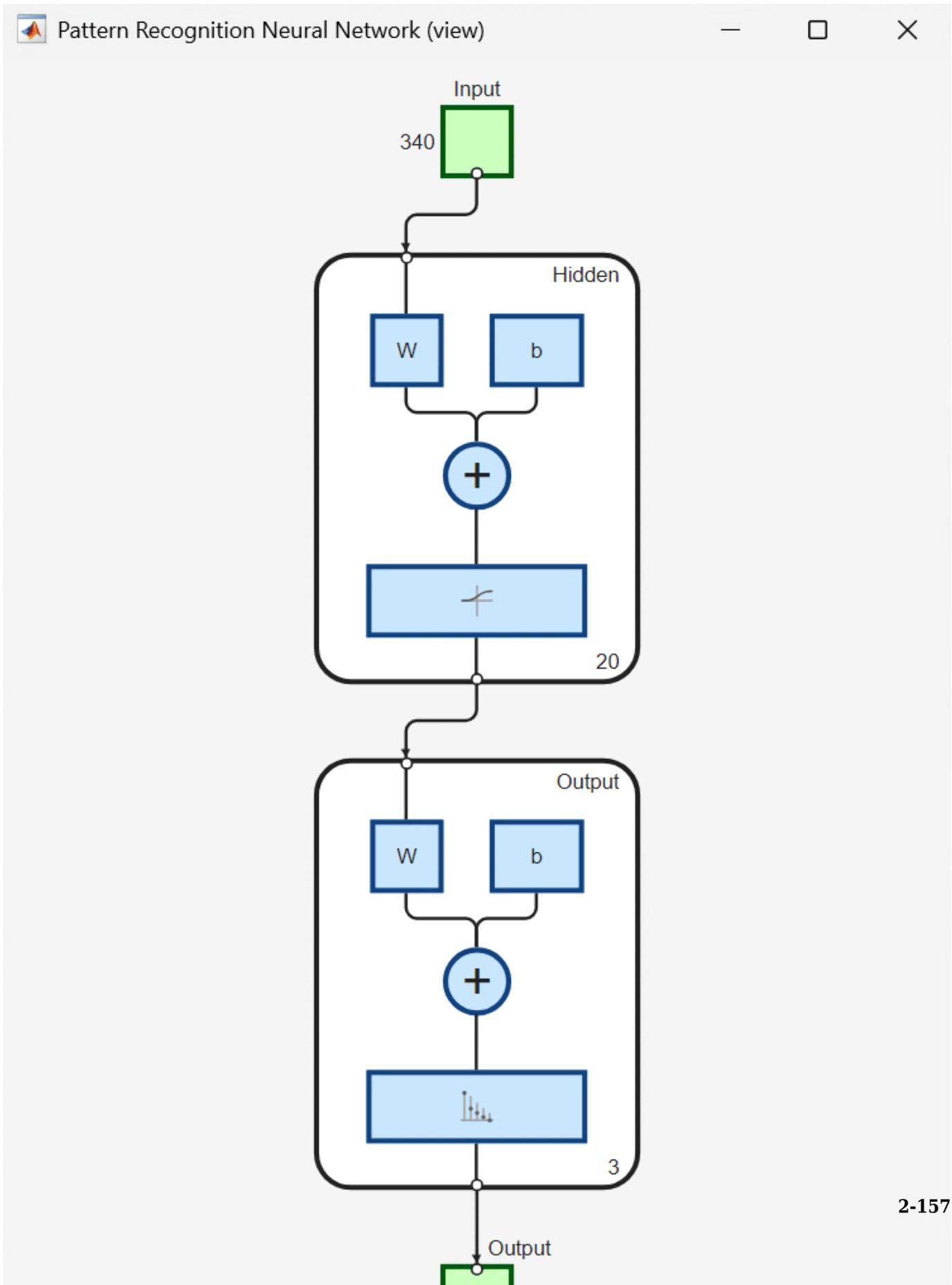
Calculations: MEX

Training Plots

Performance

Training State

You can see the training details such as the algorithm, the performance criteria, the type of error considered, and so on. The *ViewNetwork* block also generates a graphical view of the neural network.



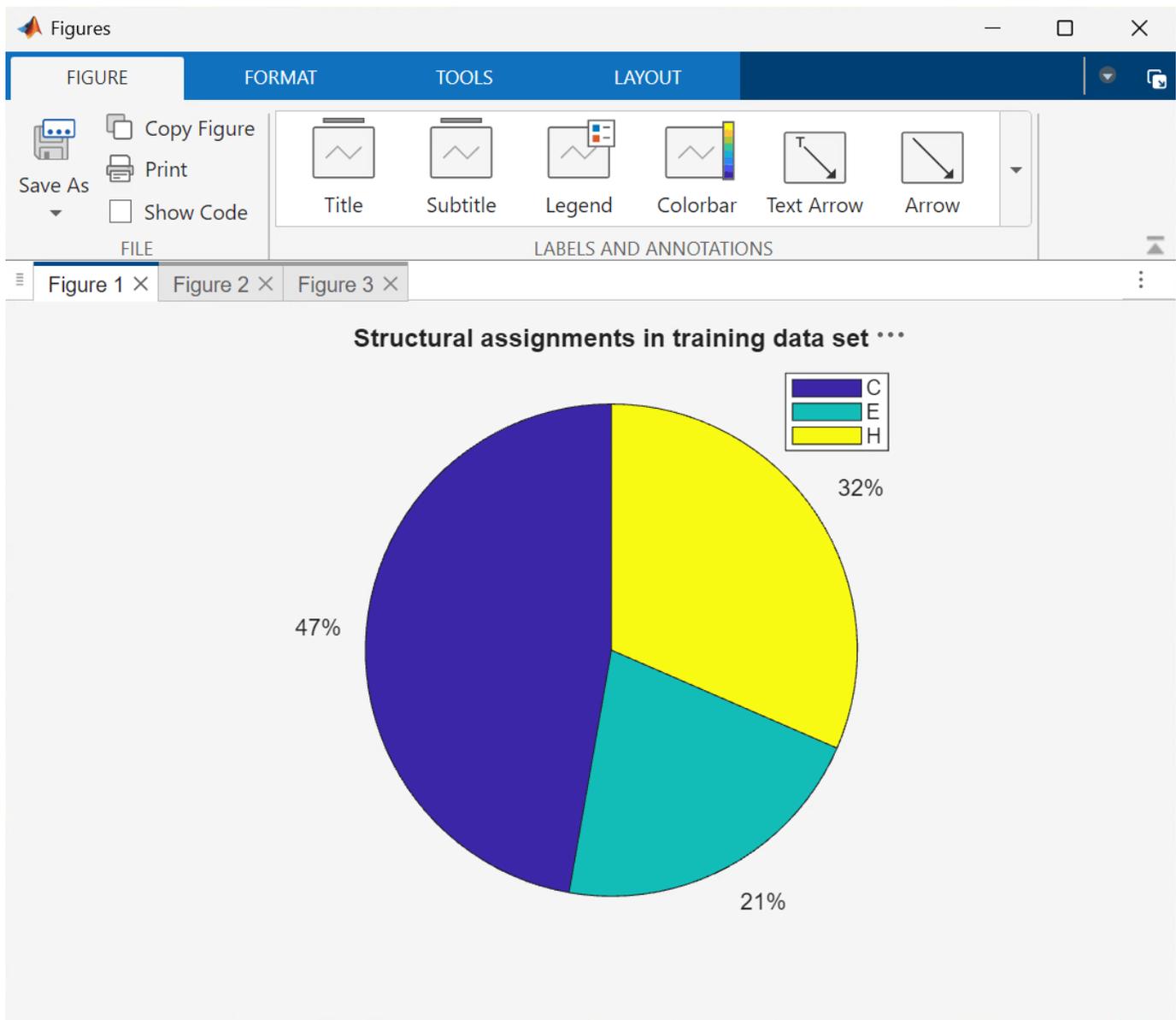
This example uses the early stopping criterion to avoid data overfitting. For details, see **Training the Neural Network** in “Predicting Protein Secondary Structure Using a Neural Network” on page 3-107.

The `train` function, by default, divides the available training data set into three subsets. The function assigns:

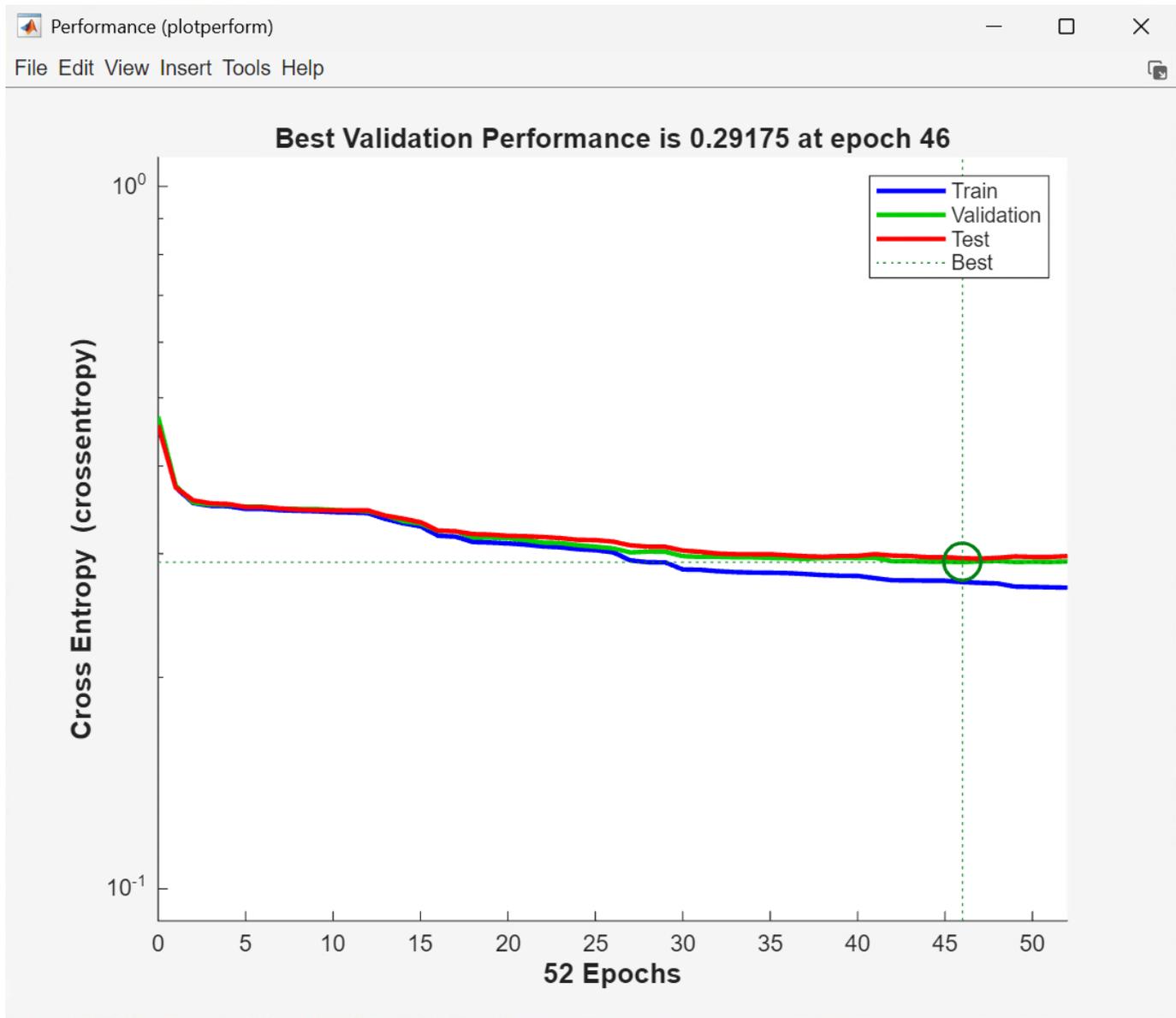
- 60% of the samples to the training set, which is used for computing the gradient and updating the network weights and biases
- 20% to the validation set, which is used to monitor the cross-entropy error during the training process because it tends to increase when data is overfitted
- 20% to the test set, which is used to assess the quality of the division of the data set

You can apply other types of partitioning by specifying the property `net.divideFcn` (default `dividerand`). This property can be easily modified within the *AdjustNetworkProps* block, similar to the transfer function described previously.

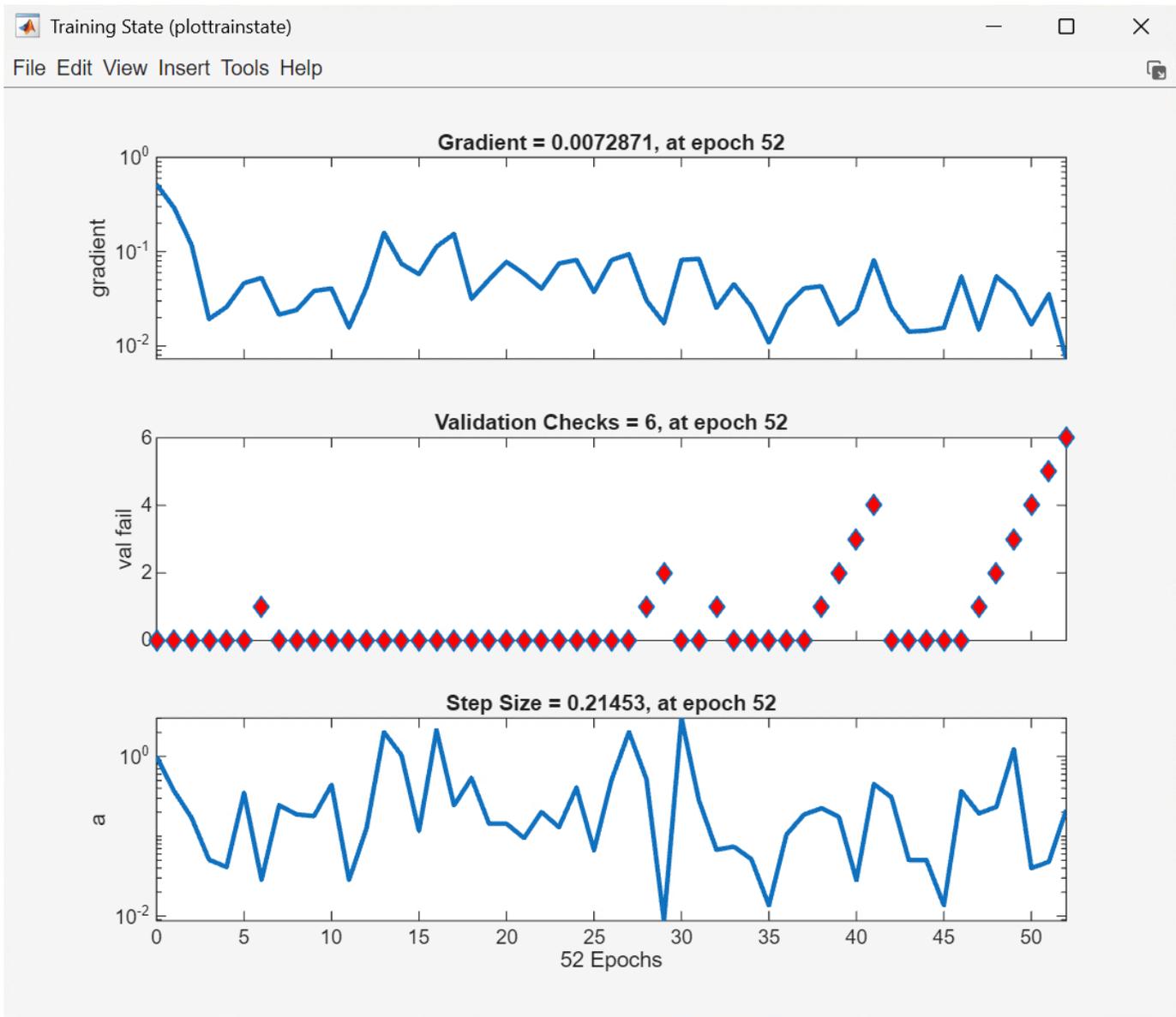
The structural composition of the residues in the three subsets is comparable, as seen from the following three figures generated by the **UserFunction** block named *PlotTrainingSubsetCompositions*.



The block *PlotTrainingPerformance* uses the `plotperform` function to display the trends of the training, validation, and test errors as training iterations pass.

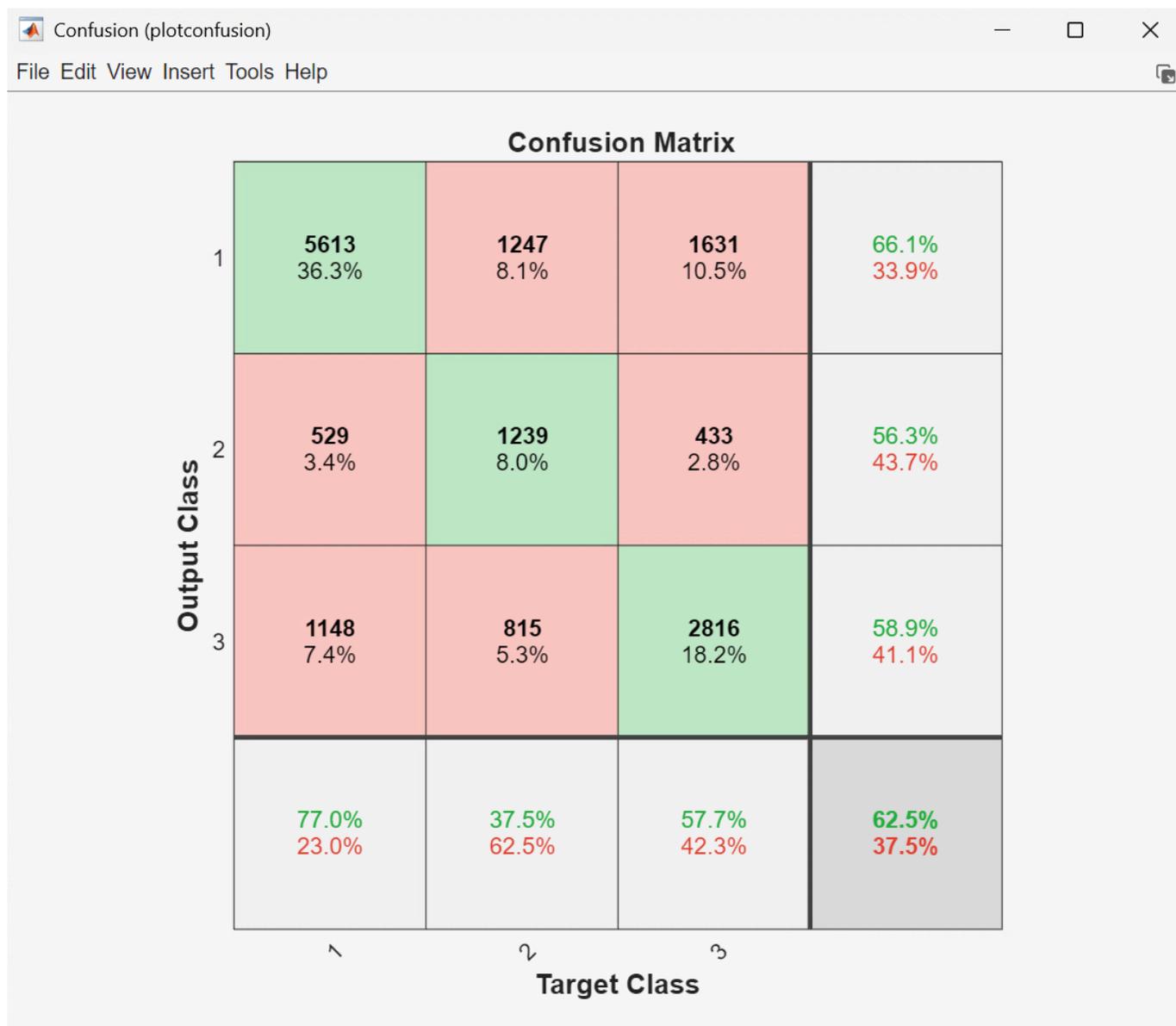


The training process stops when one of several conditions (see `net.trainParam` which can also be viewed within the `AdjustNetworkProps` block) is met. For example, in the training considered, the training process stops when the validation error increases for a specified number of iterations (6) or the maximum number of allowed iterations is reached (1000).

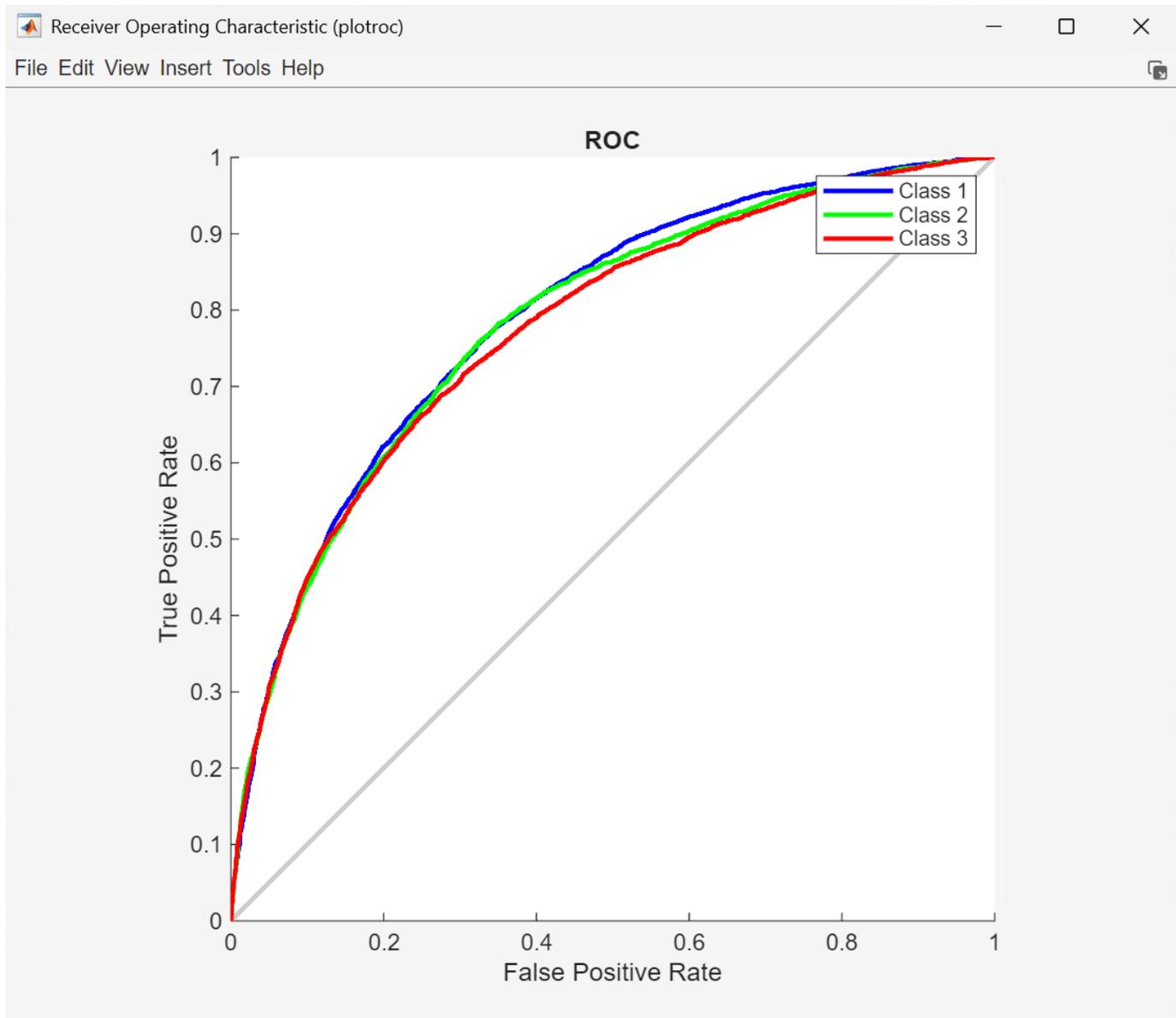


Analyze the Network Response

To analyze the network response, examine the confusion matrix by considering the outputs of the trained network and comparing them to the expected results (targets). The *PlotConfusionMatrix* block leverages the MATLAB function `plotconfusion` to generate the following figure.



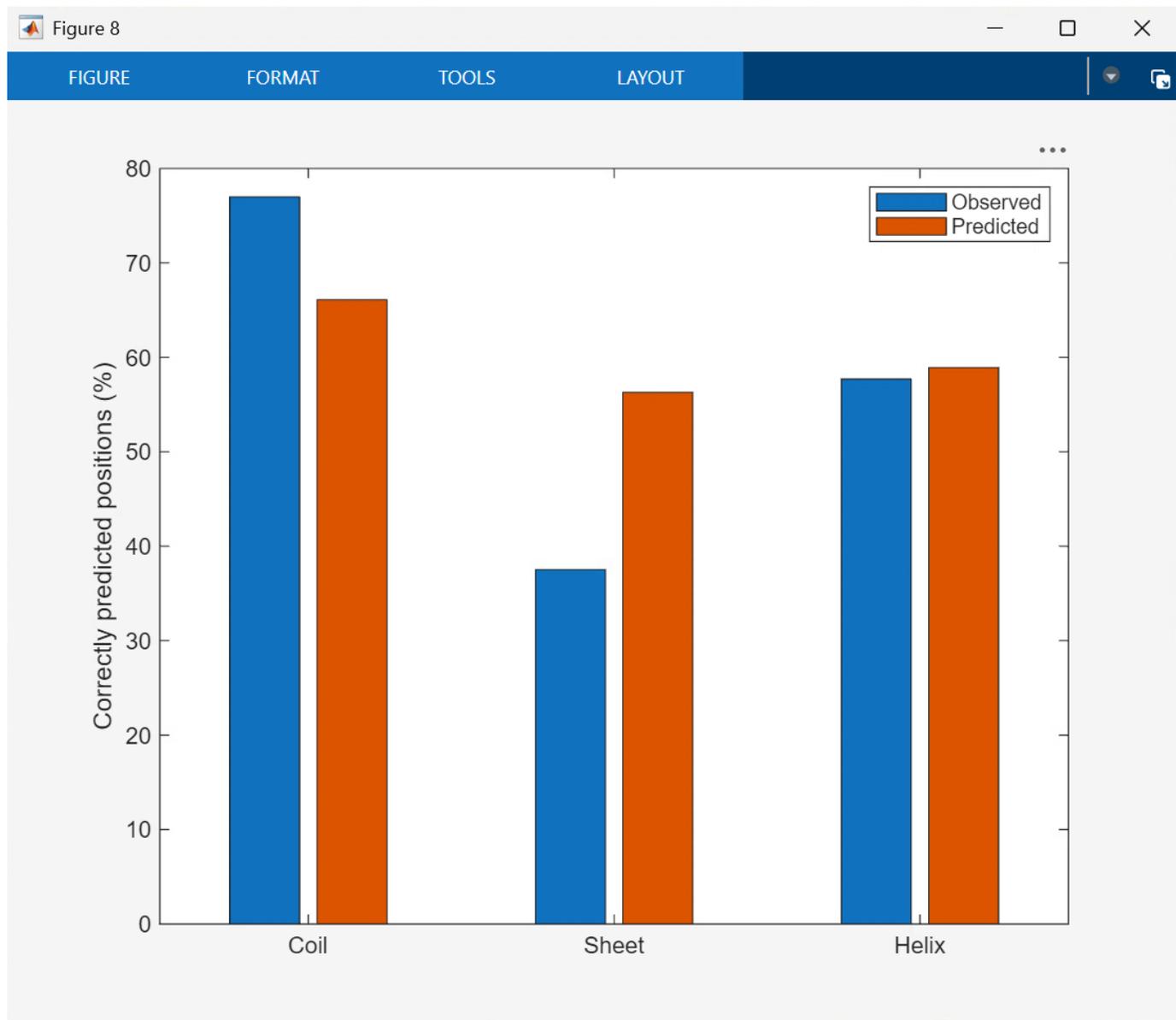
Plot the Receiver Operating Characteristic (ROC) curve, which is a plot of the true positive rate (sensitivity) versus the false positive rate (1 - specificity), using the *PlotROCCurve* block.



For more details on these plots and the information that they explain, see **Analyzing the Network Response** in "Predicting Protein Secondary Structure Using a Neural Network" on page 3-107.

Assess Network Performance

You can evaluate structure predictions in detail by calculating prediction quality indices [3], which indicate how well a particular state is predicted and whether overprediction or underprediction has occurred. Calculate the network performance in the *AssessNetworkPerformance* block, whose implementation can be viewed by opening the block's function using the same Edit Function workflow. More details on this calculation can also be found here **Assessing Network Performance** in "Predicting Protein Secondary Structure Using a Neural Network" on page 3-107.



References

- [1] Rost, B., and Sander, C., "Prediction of protein secondary structure at better than 70% accuracy", *Journal of Molecular Biology*, 232(2):584-99, 1993.
- [2] Holley, L H, and M Karplus. "Protein Secondary Structure Prediction with a Neural Network." *Proceedings of the National Academy of Sciences* 86, no. 1 (1989): 152-56. <https://doi.org/10.1073/pnas.86.1.152>.

[3] Kabsch, Wolfgang, and Christian Sander. "How Good Are Predictions of Protein Secondary Structure?" *FEBS Letters* 155, no. 2 (1983): 179-82. [https://doi.org/10.1016/0014-5793\(82\)80597-8](https://doi.org/10.1016/0014-5793(82)80597-8).

See Also

Biopipeline Designer | `bioinfo.pipeline.block.Load` |
`bioinfo.pipeline.block.UserFunction`

Sequence Analysis

Sequence analysis is the process you use to find information about a nucleotide or amino acid sequence using computational methods. Common tasks in sequence analysis are identifying genes, determining the similarity of two genes, determining the protein coded by a gene, and determining the function of a gene by finding a similar gene in another organism with a known function.

- “Exploring a Nucleotide Sequence Using Command Line” on page 3-2
- “Compare Sequences Using Sequence Alignment Algorithms” on page 3-15
- “View and Align Multiple Sequences” on page 3-22
- “Analyzing Synonymous and Nonsynonymous Substitution Rates” on page 3-36
- “Investigating the Bird Flu Virus” on page 3-46
- “Exploring Primer Design” on page 3-57
- “Identifying Over-Represented Regulatory Motifs” on page 3-67
- “Predicting and Visualizing the Secondary Structure of RNA Sequences” on page 3-78
- “Using HMMs for Profile Analysis of a Protein Family” on page 3-90
- “Predicting Protein Secondary Structure Using a Neural Network” on page 3-107
- “Calculating and Visualizing Sequence Statistics” on page 3-124
- “Aligning Pairs of Sequences” on page 3-138
- “Assessing the Significance of an Alignment” on page 3-144
- “Using Scoring Matrices to Measure Evolutionary Distance” on page 3-153
- “Calling Bioperl Functions from MATLAB” on page 3-157
- “Accessing NCBI Entrez Databases with E-Utilities” on page 3-169

Exploring a Nucleotide Sequence Using Command Line

In this section...

“Overview of Example” on page 3-2
 “Searching the Web for Sequence Information” on page 3-2
 “Reading Sequence Information from the Web” on page 3-4
 “Determining Nucleotide Composition” on page 3-5
 “Determining Codon Composition” on page 3-8
 “Open Reading Frames” on page 3-11
 “Amino Acid Conversion and Composition” on page 3-13

Overview of Example

After sequencing a piece of DNA, one of the first tasks is to investigate the nucleotide content in the sequence. Starting with a DNA sequence, this example uses sequence statistics functions to determine mono-, di-, and trinucleotide content, and to locate open reading frames.

Searching the Web for Sequence Information

The following procedure illustrates how to use the system web browser to search the Web for information. In this example you are interested in studying the human mitochondrial genome. While many genes that code for mitochondrial proteins are found in the cell nucleus, the mitochondrial has genes that code for proteins used to produce energy.

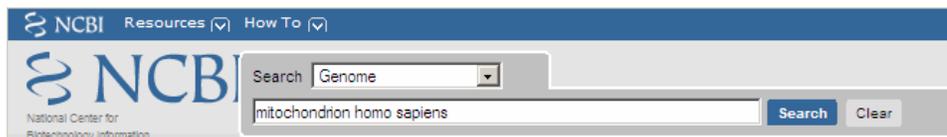
First research information about the human mitochondria and find the nucleotide sequence for the genome. Next, look at the nucleotide content for the entire sequence. And finally, determine open reading frames and extract specific gene sequences.

- 1 In the MATLAB Command Window, type

```
web('http://www.ncbi.nlm.nih.gov/')
```

A separate browser window opens with the home page for the NCBI Web site.

- 2 Search the NCBI Web site for information. For example, to search for the human mitochondrion genome, from the **Search** list, select Genome , and in the **Search** list, enter mitochondrion homo sapiens.



The NCBI Web search returns a list of links to relevant pages.

NCBI Entrez Genome

Search Genome for mitochondrion homo sapiens Go Clear Save Search

Limits Preview/Index History Clipboard Details

Display Summary Show 20 Send to

All: 49

Items 1 - 20 of 49 Page 1 of 3 Next

1: [NC_003415](#) Links

Ancylostoma duodenale mitochondrion, complete genome
DNA; circular; Length: 13,721 nt
Organelle: mitochondrion
Created: 2002/02/21

- 3 Select a result page. For example, click the link labeled **NC_012920**.

The web browser displays the NCBI page for the human mitochondrial genome.

The image shows the top portion of the NCBI Genome browser. At the top left is the NCBI logo. To its right is a circular map of the human mitochondrial genome with various genes labeled: *g3fc*, *thrL*, *cybR*, *ushR*, *fdoL*, and *g3fc*. Below the map are navigation tabs: PubMed, Nucleotide, Protein, Genome, Structure, OMIM, PMC, Journals, and Books. A search bar contains the text "Genome" and "for". Below the search bar are buttons for "Limits", "Preview/Index", "History", "Clipboard", and "Details". At the bottom of this section are "Display" and "Show" dropdown menus, and a "Send to" dropdown menu.

[Genome](#) > [Eukaryota](#) > [Homo sapiens mitochondrion, complete genome](#)

Links

Lineage: [Eukaryota](#); [Fungi/Metazoa group](#); [Metazoa](#); [Eumetazoa](#); [Bilateria](#); [Coelomata](#); [Deuterostomia](#); [Chordata](#); [Craniata](#); [Vertebrata](#); [Gnathostomata](#); [Teleostomi](#); [Euteleostomi](#); [Sarcopterygii](#); [Tetrapoda](#); [Amniota](#); [Mammalia](#); [Theria](#); [Eutheria](#); [Euarchontoglires](#); [Primates](#); [Haplorrhini](#); [Simiiformes](#); [Catarrhini](#); [Hominoidea](#); [Hominidae](#); [Homininae](#); [Homo](#); [Homo sapiens](#)

Genome Info:	Features:	BLAST homologs:	Links:	Review Info:
Refseq: NC_012920	Genes: 37	COG	Genome Project	Publications: [2]
GenBank: J01415	Protein coding: 13	TaxMap	Refseq FTP	Refseq Status: PROVISIONAL
Length: 16,569 nt	Structural RNAs: 24	TaxPlot	GenBank FTP	Seq. Status: Completed
GC Content: 44%	Pseudo genes: None	GenePlot	BLAST	Sequencing center: Center for Molecular and Mitochondrial Medicine and Genetics (MAMMAG) University of California, University of California, Irvine, Mitomap.org, USA, Irvine
% Coding: 68%	Others: 30	gMap	TraceAssembly	Completed: 2009/07/08
Topology: circular	Contigs: None		CDD	Organism Group
Molecule: dsDNA			Other genomes for species: 5683	

Gene Classification based on [COG functional categories](#)

Search gene, GeneID or locus_tag:



Display [Overview](#) Show [20](#) Send to

Reading Sequence Information from the Web

The following procedure illustrates how to find a nucleotide sequence in a public database and read the sequence information into the MATLAB environment. Many public databases for nucleotide sequences are accessible from the Web. The MATLAB Command Window provides an integrated environment for bringing sequence information into the MATLAB environment.

The consensus sequence for the human mitochondrial genome has the GenBank accession number NC_012920. Since the whole GenBank entry is quite large and you might only be interested in the sequence, you can get just the sequence information.

- 1 Get sequence information from a Web database. For example, to retrieve sequence information for the human mitochondrial genome, in the MATLAB Command Window, type

```
mitochondria = getgenbank('NC_012920', 'SequenceOnly', true)
```

The `getgenbank` function retrieves the nucleotide sequence from the GenBank database and creates a character array.

```
mitochondria =
GATCACAGGTCTATCACCTATTAACCACTCACGGGAGCTCTCCATGCAT
TTGGTATTTTCGTCTGGGGGTGTGCACGCGATAGCATTGCGAGACGCTG
GAGCCGGAGCACCTATGTCGAGTATCTGTCTTTGATTCCTGCCTCATT
CTATTATTATCGCACCTACGTTCAATATTACAGGCGAACATACCTACTA
AAGT . . .
```

- 2 If you don't have a Web connection, you can load the data from a MAT file included with the Bioinformatics Toolbox software, using the command

```
load mitochondria
```

The `load` function loads the sequence `mitochondria` into the MATLAB Workspace.

- 3 Get information about the sequence. Type

```
whos mitochondria
```

Information about the size of the sequence displays in the MATLAB Command Window.

Name	Size	Bytes	Class	Attributes
mitochondria	1x16569	33138	char	

Determining Nucleotide Composition

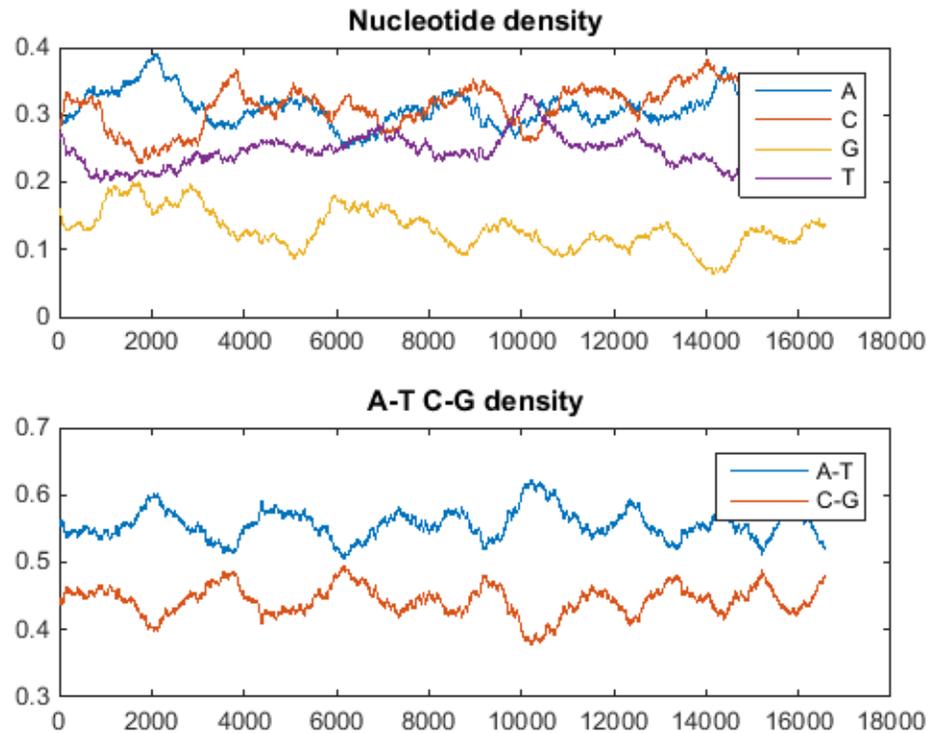
The following procedure illustrates how to determine the monomers and dimers, and then visualize data in graphs and bar plots. Sections of a DNA sequence with a high percent of A+T nucleotides usually indicate intergenic parts of the sequence, while low A+T and higher G+C nucleotide percentages indicate possible genes. Many times high CG dinucleotide content is located before a gene.

After you read a sequence into the MATLAB environment, you can use the sequence statistics functions to determine if your sequence has the characteristics of a protein-coding region. This procedure uses the human mitochondrial genome as an example. See “Reading Sequence Information from the Web” on page 3-4.

- 1 Plot monomer densities and combined monomer densities in a graph. In the MATLAB Command Window, type

```
ntdensity(mitochondria)
```

This graph shows that the genome is A+T rich.



- 2 Count the nucleotides using the `basecount` function.

```
basecount(mitochondria)
```

A list of nucleotide counts is shown for the 5'-3' strand.

```
ans =
  A: 5124
  C: 5181
  G: 2169
  T: 4094
```

- 3 Count the nucleotides in the reverse complement of a sequence using the `seqrcomplement` function.

```
basecount(seqrcomplement(mitochondria))
```

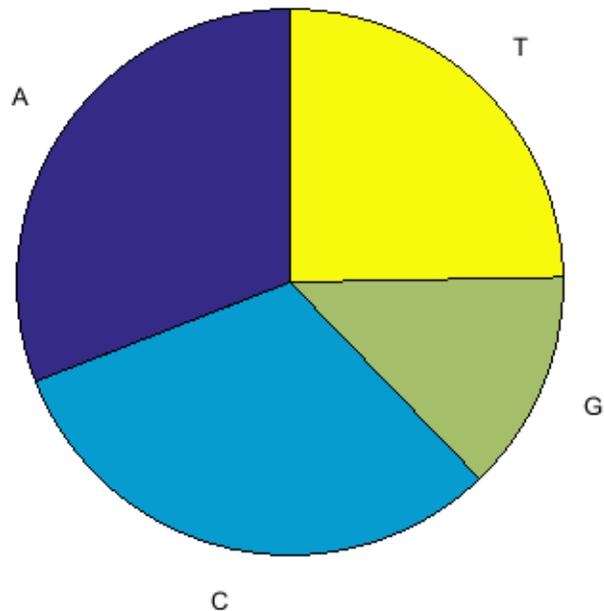
As expected, the nucleotide counts on the reverse complement strand are complementary to the 5'-3' strand.

```
ans =
  A: 4094
  C: 2169
  G: 5181
  T: 5124
```

- 4 Use the function `basecount` with the `chart` option to visualize the nucleotide distribution.

```
figure
basecount(mitochondria,'chart','pie');
```

A pie chart displays in the MATLAB Figure window.

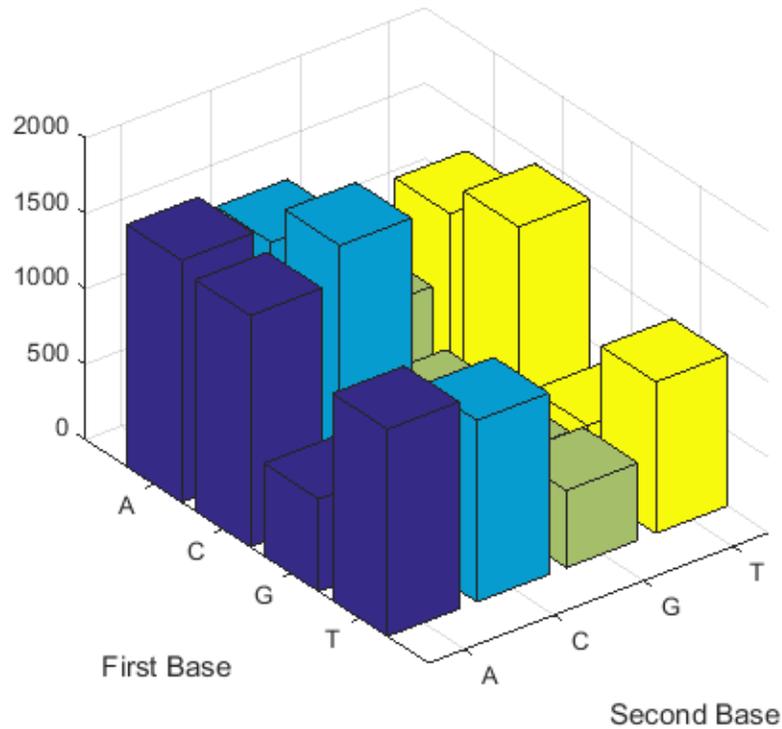


- 5 Count the dimers in a sequence and display the information in a bar chart.

```
figure  
dimercount(mitochondria,'chart','bar')
```

```
ans =
```

```
AA: 1604  
AC: 1495  
AG: 795  
AT: 1230  
CA: 1534  
CC: 1771  
CG: 435  
CT: 1440  
GA: 613  
GC: 711  
GG: 425  
GT: 419  
TA: 1373  
TC: 1204  
TG: 513  
TT: 1004
```



Determining Codon Composition

The following procedure illustrates how to look at codons for the six reading frames. Trinucleotides (codon) code for an amino acid, and there are 64 possible codons in a nucleotide sequence. Knowing the percent of codons in your sequence can be helpful when you are comparing with tables for expected codon usage.

After you read a sequence into the MATLAB environment, you can analyze the sequence for codon composition. This procedure uses the human mitochondria genome as an example. See “Reading Sequence Information from the Web” on page 3-4.

- 1 Count codons in a nucleotide sequence. In the MATLAB Command Window, type

```
codoncount(mitochondria)
```

The codon counts for the first reading frame displays.

AAA - 167	AAC - 171	AAG - 71	AAT - 130
ACA - 137	ACC - 191	ACG - 42	ACT - 153
AGA - 59	AGC - 87	AGG - 51	AGT - 54
ATA - 126	ATC - 131	ATG - 55	ATT - 113
CAA - 146	CAC - 145	CAG - 68	CAT - 148
CCA - 141	CCC - 205	CCG - 49	CCT - 173
CGA - 40	CGC - 54	CGG - 29	CGT - 27
CTA - 175	CTC - 142	CTG - 74	CTT - 101
GAA - 67	GAC - 53	GAG - 49	GAT - 35
GCA - 81	GCC - 101	GCG - 16	GCT - 59

```

GGA - 36      GGC - 47      GGG - 23      GGT - 28
GTA - 43      GTC - 26      GTG - 18      GTT - 41
TAA - 157     TAC - 118     TAG - 94      TAT - 107
TCA - 125     TCC - 116     TCG - 37     TCT - 103
TGA - 64      TGC - 40     TGG - 29     TGT - 26
TTA - 96      TTC - 107     TTG - 47     TTT - 78

```

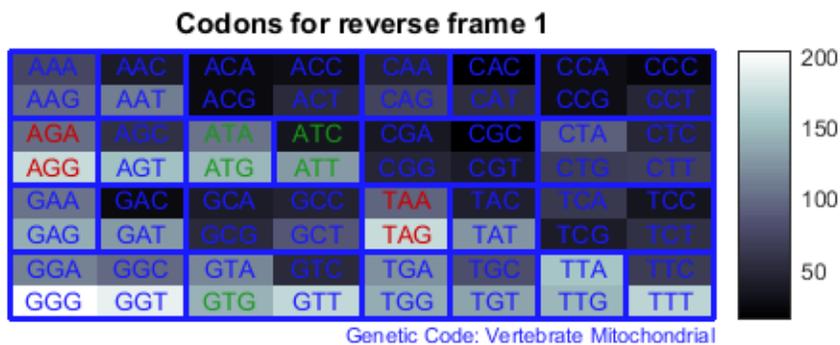
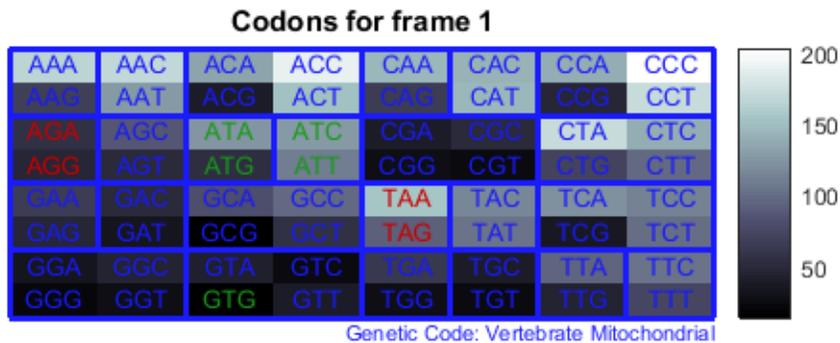
2 Count the codons in all six reading frames and plot the results in heat maps.

```

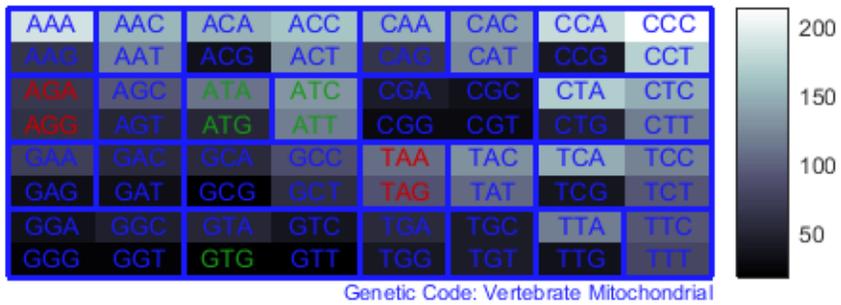
for frame = 1:3
    figure
    subplot(2,1,1);
    codoncount(mitochondria,'frame',frame,'figure',true,...
        'geneticcode','Vertebrate Mitochondrial');
    title(sprintf('Codons for frame %d',frame));
    subplot(2,1,2);
    codoncount(mitochondria,'reverse',true,'frame',frame,...
        'figure',true,'geneticcode','Vertebrate Mitochondrial');
    title(sprintf('Codons for reverse frame %d',frame));
end

```

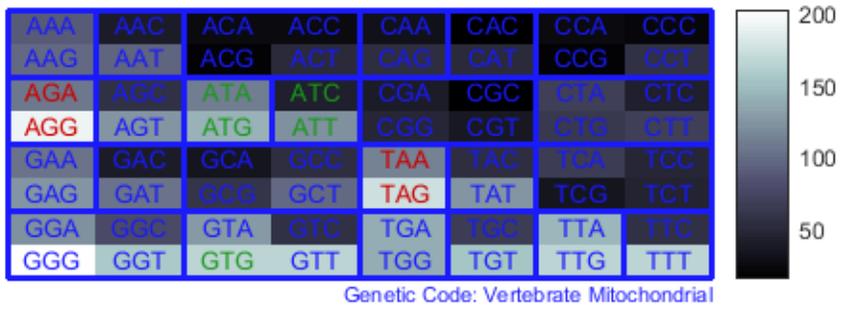
Heat maps display all 64 codons in the 6 reading frames.



Codons for frame 2

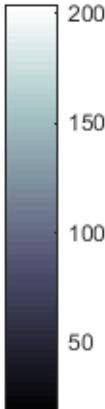


Codons for reverse frame 2



Codons for frame 3

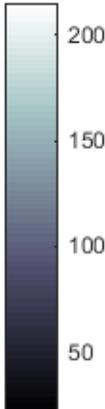
AAA	AAC	ACA	ACC	CAA	CAC	CCA	CCC
AAG	AAT	ACG	ACT	CAG	CAT	CCG	CCT
AGA	AGC	ATA	ATC	CGA	CGC	CTA	CTC
AGG	AGT	ATG	ATT	CGG	CGT	CTG	CTT
GAA	GAC	GCA	GCC	TAA	TAC	TCA	TCC
GAG	GAT	GCG	GCT	TAG	TAT	TCG	TCT
GGA	GGC	GTA	GTC	TGA	TGC	TTA	TTC
GGG	GGT	GTG	GTT	TGG	TGT	TTG	TTT



Genetic Code: Vertebrate Mitochondrial

Codons for reverse frame 3

AAA	AAC	ACA	ACC	CAA	CAC	CCA	CCC
AAG	AAT	ACG	ACT	CAG	CAT	CCG	CCT
AGA	AGC	ATA	ATC	CGA	CGC	CTA	CTC
AGG	AGT	ATG	ATT	CGG	CGT	CTG	CTT
GAA	GAC	GCA	GCC	TAA	TAC	TCA	TCC
GAG	GAT	GCG	GCT	TAG	TAT	TCG	TCT
GGA	GGC	GTA	GTC	TGA	TGC	TTA	TTC
GGG	GGT	GTG	GTT	TGG	TGT	TTG	TTT



Genetic Code: Vertebrate Mitochondrial

Open Reading Frames

The following procedure illustrates how to locate the open reading frames using a specific genetic code. Determining the protein-coding sequence for a eukaryotic gene can be a difficult task because introns (noncoding sections) are mixed with exons. However, prokaryotic genes generally do not have introns and mRNA sequences have the introns removed. Identifying the start and stop codons for translation determines the protein-coding section, or open reading frame (ORF), in a sequence. Once you know the ORF for a gene or mRNA, you can translate a nucleotide sequence to its corresponding amino acid sequence.

After you read a sequence into the MATLAB environment, you can analyze the sequence for open reading frames. This procedure uses the human mitochondria genome as an example. See “Reading Sequence Information from the Web” on page 3-4.

- 1 Display open reading frames (ORFs) in a nucleotide sequence. In the MATLAB Command Window, type:

```
seqshoworfs(mitochondria);
```

If you compare this output to the genes shown on the NCBI page for NC_012920, there are fewer genes than expected. This is because vertebrate mitochondria use a genetic code slightly different from the standard genetic code. For a list of genetic codes, see the *Genetic Code* table in the aa2nt reference page.

- 2 Display ORFs using the Vertebrate Mitochondrial code.

```
orfs= seqshoworfs(mitochondria,...
                  'GeneticCode','Vertebrate Mitochondrial',...
                  'alternativestart',true);
```

Notice that there are now two large ORFs on the third reading frame. One starts at position 4470 and the other starts at 5904. These correspond to the genes ND2 (NADH dehydrogenase subunit 2 [Homo sapiens]) and COX1 (cytochrome c oxidase subunit I) genes.

- 3 Find the corresponding stop codon. The start and stop positions for ORFs have the same indices as the start positions in the fields Start and Stop.

```
ND2Start = 4470;
StartIndex = find(orfs(3).Start == ND2Start)
ND2Stop = orfs(3).Stop(StartIndex)
```

The stop position displays.

```
ND2Stop =
        5511
```

- 4 Using the sequence indices for the start and stop of the gene, extract the subsequence from the sequence.

```
ND2Seq = mitochondria(ND2Start:ND2Stop)
```

The subsequence (protein-coding region) is stored in ND2Seq and displayed on the screen.

```
attaatcccctggcccaaccgctcatctactctaccatctttgcaggcac
actcatcacagcgctaagctcgactgattttttacctgagtaggcctag
aaataaacatgctagcttttattccagttctaaccaaaaaataaacct
cgttccacagaagctgccatcaagtatttctcagcaagcaaccgcatc
cataatccttc . . .
```

- 5 Determine the codon distribution.

```
codoncount (ND2Seq)
```

The codon count shows a high amount of ACC, ATA, CTA, and ATC.

AAA - 10	AAC - 14	AAG - 2	AAT - 6
ACA - 11	ACC - 24	ACG - 3	ACT - 5
AGA - 0	AGC - 4	AGG - 0	AGT - 1
ATA - 23	ATC - 24	ATG - 1	ATT - 8
CAA - 8	CAC - 3	CAG - 2	CAT - 1
CCA - 4	CCC - 12	CCG - 2	CCT - 5
CGA - 0	CGC - 3	CGG - 0	CGT - 1
CTA - 26	CTC - 18	CTG - 4	CTT - 7

```

GAA - 5      GAC - 0      GAG - 1      GAT - 0
GCA - 8      GCC - 7      GCG - 1      GCT - 4
GGA - 5      GGC - 7      GGG - 0      GGT - 1
GTA - 3      GTC - 2      GTG - 0      GTT - 3
TAA - 0      TAC - 8      TAG - 0      TAT - 2
TCA - 7      TCC - 11     TCG - 1      TCT - 4
TGA - 10     TGC - 0      TGG - 1      TGT - 0
TTA - 8      TTC - 7      TTG - 1      TTT - 8

```

- 6 Look up the amino acids for codons ATA, CTA, ACC, and ATC.

```

aminolookup('code',nt2aa('ATA'))
aminolookup('code',nt2aa('CTA'))
aminolookup('code',nt2aa('ACC'))
aminolookup('code',nt2aa('ATC'))

```

The following displays:

```

Ile   isoleucine
Leu   leucine
Thr   threonine
Ile   isoleucine

```

Amino Acid Conversion and Composition

The following procedure illustrates how to extract the protein-coding sequence from a gene sequence and convert it to the amino acid sequence for the protein. Determining the relative amino acid composition of a protein will give you a characteristic profile for the protein. Often, this profile is enough information to identify a protein. Using the amino acid composition, atomic composition, and molecular weight, you can also search public databases for similar proteins.

After you locate an open reading frame (ORF) in a gene, you can convert it to an amino sequence and determine its amino acid composition. This procedure uses the human mitochondria genome as an example. See “Open Reading Frames” on page 3-11.

- 1 Convert a nucleotide sequence to an amino acid sequence. In this example, only the protein-coding sequence between the start and stop codons is converted.

```

ND2AASeq = nt2aa(ND2Seq, 'geneticcode', ...
                 'Vertebrate Mitochondrial')

```

The sequence is converted using the Vertebrate Mitochondrial genetic code. Because the property `AlternativeStartCodons` is set to `'true'` by default, the first codon `att` is converted to M instead of I.

```

MNPLAQPVIYSTIFAGTLITALSSHWFFTWGLEMNMLAFIPVLTKKMN
RSTEAAIKYFLTQATASMIILLMAILFNNMLSGQWTMTNTTNOYSSLM
AMAMKLGMAPFHFVWPEVTQGTPLTSGLLLLTWQKLAPISIMYQISPS
VLLLLTSLSIMAGSWGGLNQTQLRKILAYSSITHMGWMMAVLPYNPNM
TILNLTIIYIILTTTAFLLLNLSSTTTLLSRTWNKLTWLTPLIPSTLL
LGGLPPLTGFLPKWAIIEEFTKNNSLIPTIMATITLLNLYFYLRRIYST
SITLLPMSNNVKMKWQFEHTKPTPFLPTLIALTTLLLPISPFMLMIL

```

- 2 Compare your conversion with the published conversion in the GenPept database.

```

ND2protein = getgenpept('YP_003024027', 'sequenceonly', true)

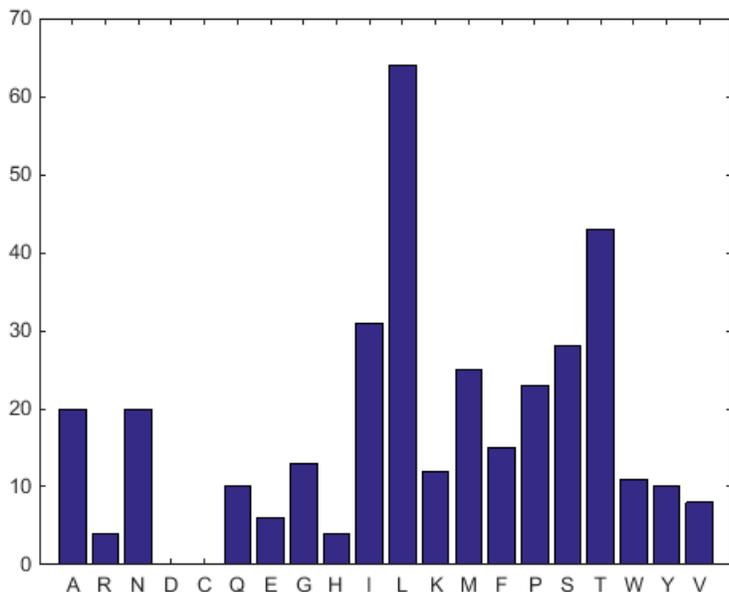
```

The `getgenpept` function retrieves the published conversion from the NCBI database and reads it into the MATLAB Workspace.

- 3 Count the amino acids in the protein sequence.

```
aaccount(ND2AASeq, 'chart','bar')
```

A bar graph displays. Notice the high content for leucine, threonine and isoleucine, and also notice the lack of cysteine and aspartic acid.



- 4 Determine the atomic composition and molecular weight of the protein.

```
atomiccomp(ND2AASeq)
molweight (ND2AASeq)
```

The following displays in the MATLAB Workspace:

```
ans =
```

```
C: 1818
H: 2882
N: 420
O: 471
S: 25
```

```
ans =
```

```
3.8960e+004
```

If this sequence was unknown, you could use this information to identify the protein by comparing it with the atomic composition of other proteins in a database.

Compare Sequences Using Sequence Alignment Algorithms

Determining the similarity between two sequences is a common task in computational biology. Starting with a nucleotide sequence for a human gene, this example uses alignment algorithms to locate and verify a corresponding gene in a model organism.

In this example, you are interested in studying Tay-Sachs disease. Tay-Sachs is an autosomal recessive disease caused by the absence of the enzyme beta-hexosaminidase A (Hex A). This enzyme is responsible for the breakdown of gangliosides (GM2) in brain and nerve cells.

First, research information about Tay-Sachs and the enzyme that is associated with this disease, then find the nucleotide sequence for the human gene that codes for the enzyme, and finally find a corresponding gene in another organism to use as a model for study.

In the MATLAB Command window, enter:

```
web('https://www.ncbi.nlm.nih.gov/books/NBK22250/')
```

The system web browser opens with the Tay-Sachs disease page in the Genes and Diseases section of the NCBI web site. This section provides a comprehensive introduction to medical genetics. In particular, this page contains an introduction and pictorial representation of the enzyme Hex A and its role in the metabolism of the lipid GM2 ganglioside.

After completing your research, you have concluded the following:

The gene HEXA codes for the alpha subunit of the dimer enzyme hexosaminidase A (Hex A), while the gene HEXB codes for the beta subunit of the enzyme. A third gene, GM2A, codes for the activator protein GM2. However, it is a mutation in the gene HEXA that causes Tay-Sachs.

Retrieve Sequence Information from a Public Database

The following procedure illustrates how to find the nucleotide sequence for a human gene in a public database and read the sequence information into the MATLAB environment. Many public databases for nucleotide sequences (for example, GenBank®, EMBL-EBI) are accessible from the Web.

After you locate a sequence, you need to move the sequence data into the MATLAB Workspace.

In the MATLAB Command Window, enter:

```
web('https://www.ncbi.nlm.nih.gov/')
```

Search for the gene you are interested in studying. For example, from the **Search** list, select **Nucleotide**, and in the **for** box enter Tay-Sachs. Look for the genes that code the alpha and beta subunits of the enzyme hexosaminidase A (Hex A), and the gene that codes the activator enzyme. The NCBI reference for the human gene HEXA has accession number NM_000520.

Get sequence data into the MATLAB environment. For example, to get sequence information for the human gene HEXA, enter:

```
humanHEXA = getgenbank('NM_000520')

humanHEXA = struct with fields:
    LocusName: 'NM_000520'
    LocusSequenceLength: '4785'
    LocusNumberofStrands: ''
```

```
LocusTopology: 'linear'
LocusMoleculeType: 'mRNA'
LocusGenBankDivision: 'PRI'
LocusModificationDate: '05-OCT-2024'
Definition: 'Homo sapiens hexosaminidase subunit alpha (HEXA), transcript variant
Accession: 'NM_000520'
Version: 'NM_000520.6'
GI: ''
Project: []
DBLink: []
Keywords: 'RefSeq; MANE Select.'
Segment: []
Source: 'Homo sapiens (human)'
SourceOrganism: [4x65 char]
Reference: {[1x1 struct] [1x1 struct] [1x1 struct] [1x1 struct] [1x1 struct]}
Comment: [46x66 char]
Features: [160x74 char]
CDS: [1x1 struct]
Sequence: 'ctcacgtggccagccccctccgagaggggagaccagcgggcatgacaagctccaggctttggttttc
SearchURL: 'https://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=nucore&id=NM_000520.6'
RetrieveURL: 'https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucore&id=NM_000520.6'
```

Search a Public Database for Related Genes

The following procedure illustrates how to find the nucleotide sequence for a mouse gene related to a human gene, and read the sequence information into the MATLAB environment. The sequence and function of many genes is conserved during the evolution of species through homologous genes. Homologous genes are genes that have a common ancestor and similar sequences. One goal of searching a public database is to find similar genes. If you are able to locate a sequence in a database that is similar to your unknown gene or protein, it is likely that the function and characteristics of the known and unknown genes are the same.

After finding the nucleotide sequence for a human gene, you can do a BLAST search or search in the genome of another organism for the corresponding gene. This procedure uses the mouse genome as an example.

In the MATLAB Command window, enter:

```
web('http://www.ncbi.nlm.nih.gov')
```

Search the nucleotide database for the gene or protein you are interested in studying. For example, from the **Search** list, select **Nucleotide**, and in the **for** box enter **hexosaminidase A**.

The search returns entries for the mouse and human genomes. For the purposes of this example, use the accession number **AK080777** for the mouse gene **HEXA**.

Get sequence information for the mouse gene into the MATLAB environment.

```
mouseHEXA = getgenbank('AK080777')
```

Compare Amino Acid Sequences

The following procedure illustrates how to use global and local alignment functions to compare two amino acid sequences. You could use alignment functions to look for similarities between two nucleotide sequences, but alignment functions return more biologically meaningful results when you are using amino acid sequences.

If you did not retrieve gene data from the Web, you can load example data from a MAT-file included with the Bioinformatics Toolbox™ software. In the MATLAB Command window, enter:

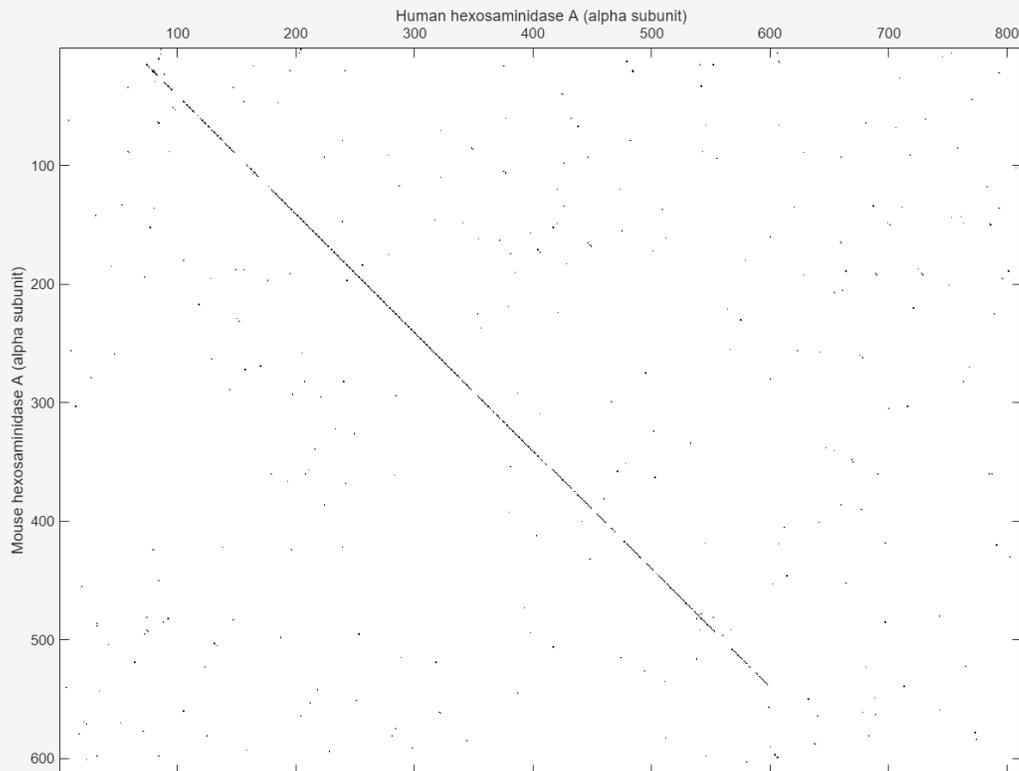
```
load hexosaminidase
```

Convert the human and mouse DNA sequences to the amino acid sequences.

```
humanProtein = nt2aa(humanHEXA.Sequence);
mouseProtein = nt2aa(mouseHEXA.Sequence);
```

Draw a dot plot comparing the human and mouse amino acid sequences. Dot plots are one of the easiest ways to look for similarity between sequences. The diagonal line shown below indicates that there may be a good alignment between the two sequences.

```
warning('off','bioinfo:seqdotplot:imageTooBigForScreen');
seqdotplot(mouseProtein,humanProtein,4,3);
ylabel('Mouse hexosaminidase A (alpha subunit)')
xlabel('Human hexosaminidase A (alpha subunit)')
uif =(gcf);
uif.Position(:) = [100 100 1280 800]; % Resize the figure.
```



```
warning('on','bioinfo:seqdotplot:imageTooBigForScreen');
```

Globally align the two amino acid sequences, using the Needleman-Wunsch algorithm.

```
[GlobalScore, GlobalAlignment] = nwalign(humanProtein,mouseProtein)
```



```

mouseStops = find(mouseProtein == '*')
mouseStops = 1x4
    539    557    574    606

```

Looking at the amino acid sequence for `humanProtein`, the first M is at position 70, and the first stop after that position is actually the second stop in the sequence (position 599). Looking at the amino acid sequence for `mouseProtein`, the first M is at position 11, and the first stop after that position is the first stop in the sequence (position 557).

Truncate the sequences to include only amino acids in the protein and the stop.

```

humanProteinORF = humanProtein(70:humanStops(2))
humanProteinORF =
'MTSSRLWFSLLLLAAAFAGRATALWPWPQNFQTSQRYVLYPNNFQFYDVSSAAQPGCSVLDEAFQRYRDLLFGSGSWPRPYLTGKRHTLEKNVL
mouseProteinORF = mouseProtein(11:mouseStops(1))
mouseProteinORF =
'MAGCRLWVSLLLLLAAALACLATALWPWPQYIQTYHRRYTLYPNNFQFRYHVSSAAQAGCVVLDEAFRRYRNLLFGSGSWPRPSFSNKQQTGKNIL

```

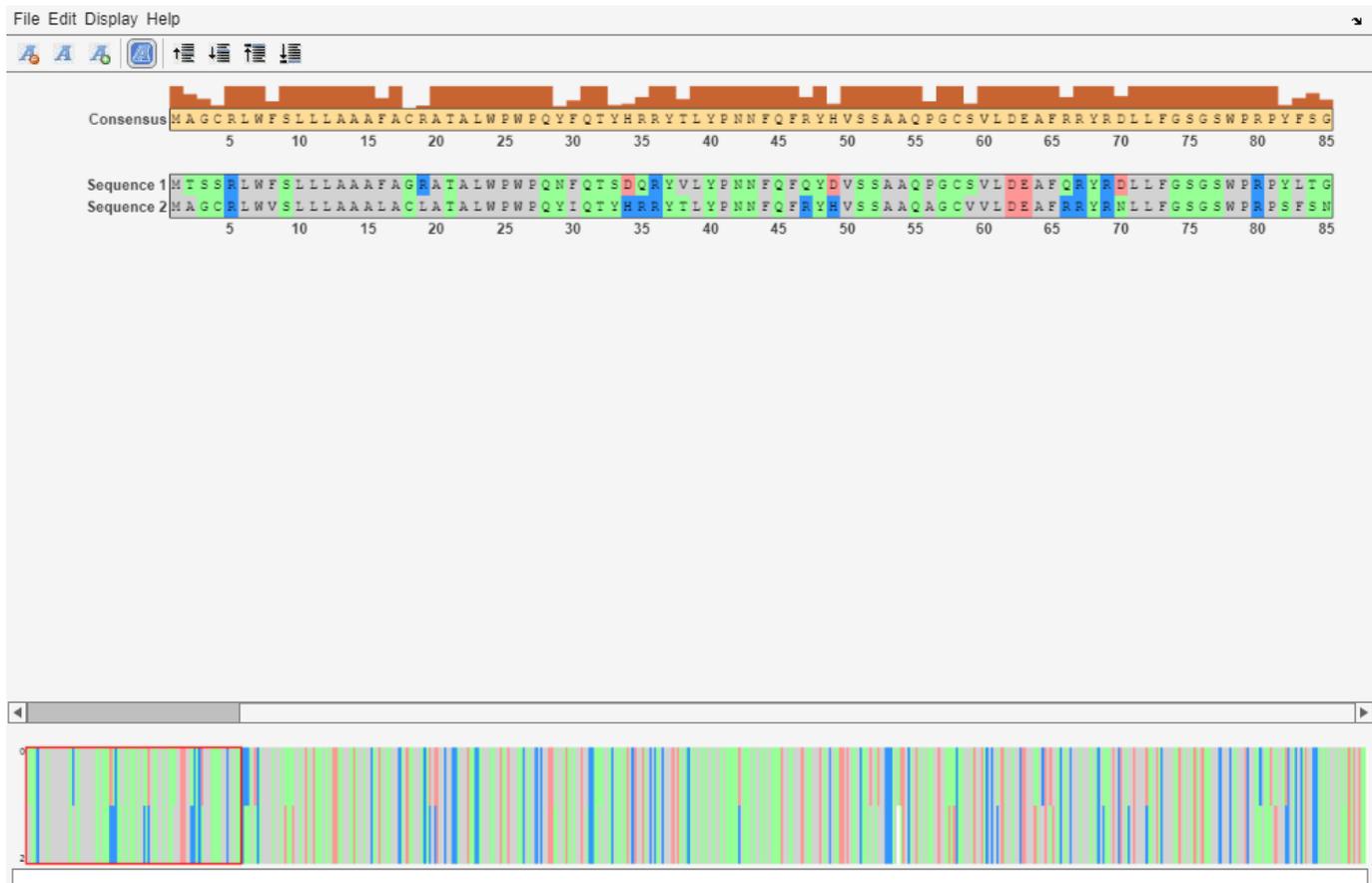
Globally align the trimmed amino acid sequences.

```

[GlobalScore_trim, GlobalAlignment_trim] = nwalignment(humanProteinORF,mouseProteinORF);
seqalignviewer(GlobalAlignment_trim);

```

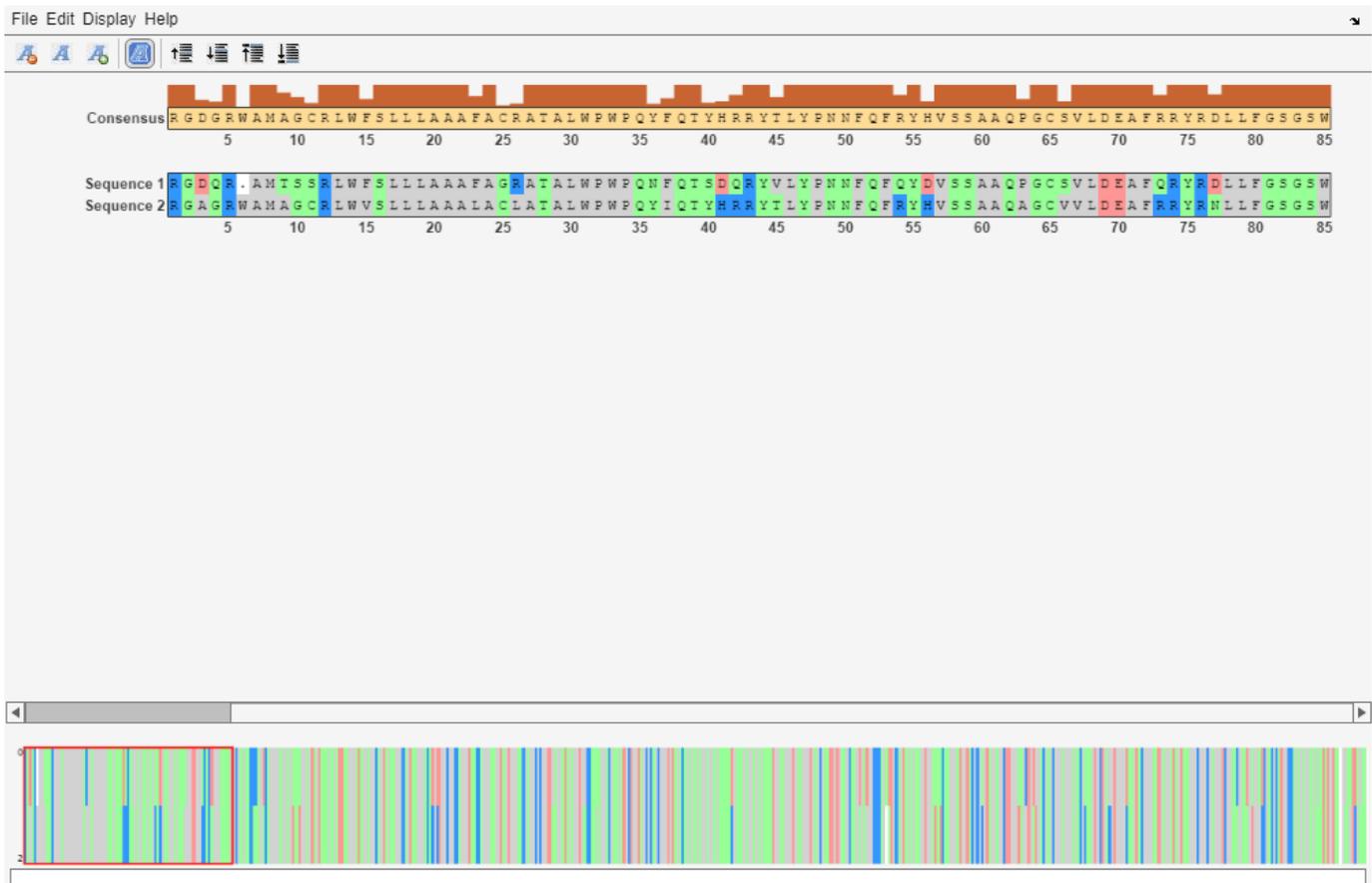
3 Sequence Analysis



An alternative method to working with subsequences is to use a local alignment function with the nontruncated sequences.

Locally align the two amino acid sequences using a Smith-Waterman algorithm.

```
[LocalScore, LocalAlignment] = swalign(humanProtein,mouseProtein);  
seqalignviewer(LocalAlignment);
```



close all;

See Also

swalign | nwalignment

View and Align Multiple Sequences

In this section...
“Overview of the Sequence Alignment App” on page 3-22
“Visualize Multiple Sequence Alignment” on page 3-22
“Adjust Sequence Alignments Manually” on page 3-23
“Rearrange Rows” on page 3-31
“Generate Phylogenetic Tree from Aligned Sequences” on page 3-33

Overview of the Sequence Alignment App

The Sequence Alignment app integrates many sequence and multiple alignment functions in the toolbox. Instead of entering commands in the MATLAB Command Window, you can use this app to visually inspect a multiple alignment and make manual adjustments.

Visualize Multiple Sequence Alignment

- 1 Read a multiple sequence alignment file of the gag polyprotein for several HIV strains.

```
gagaa = multialignread('aagag.aln')
```

- 2 View the aligned sequences in the **Sequence Alignment** app.

```
seqalignviewer(gagaa);
```

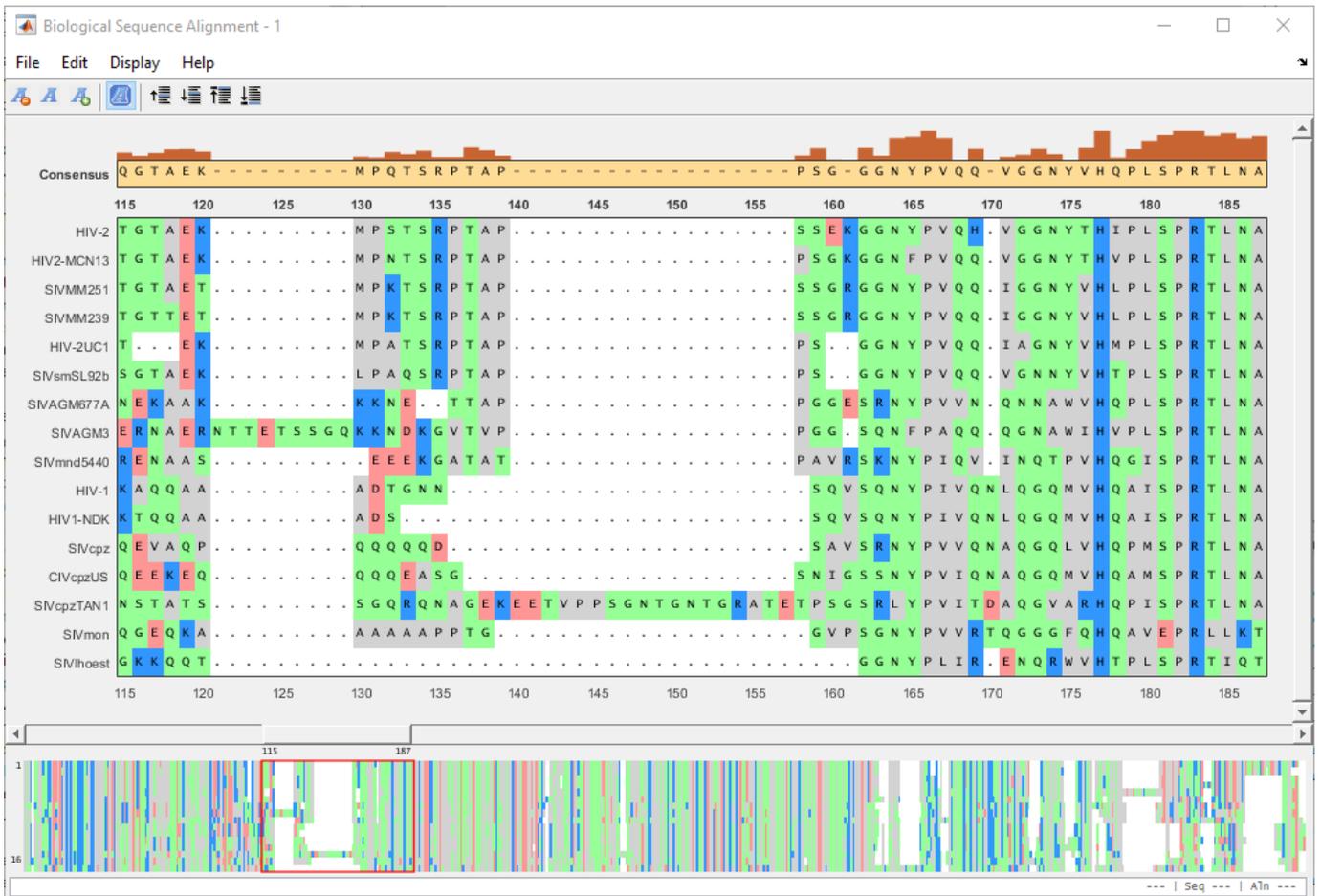


Adjust Sequence Alignments Manually

Algorithms for aligning multiple sequences do not always produce an optimal result. By visually inspecting the alignment, you can identify areas whose alignment can be improved by a manual adjustment.

- 1 To better visualize the sequence alignments, you can zoom in by selecting **Display > Zoom in**. Select this option multiple times until you achieve the zoom level you want.
- 2 Identify an area where you could improve the alignment.

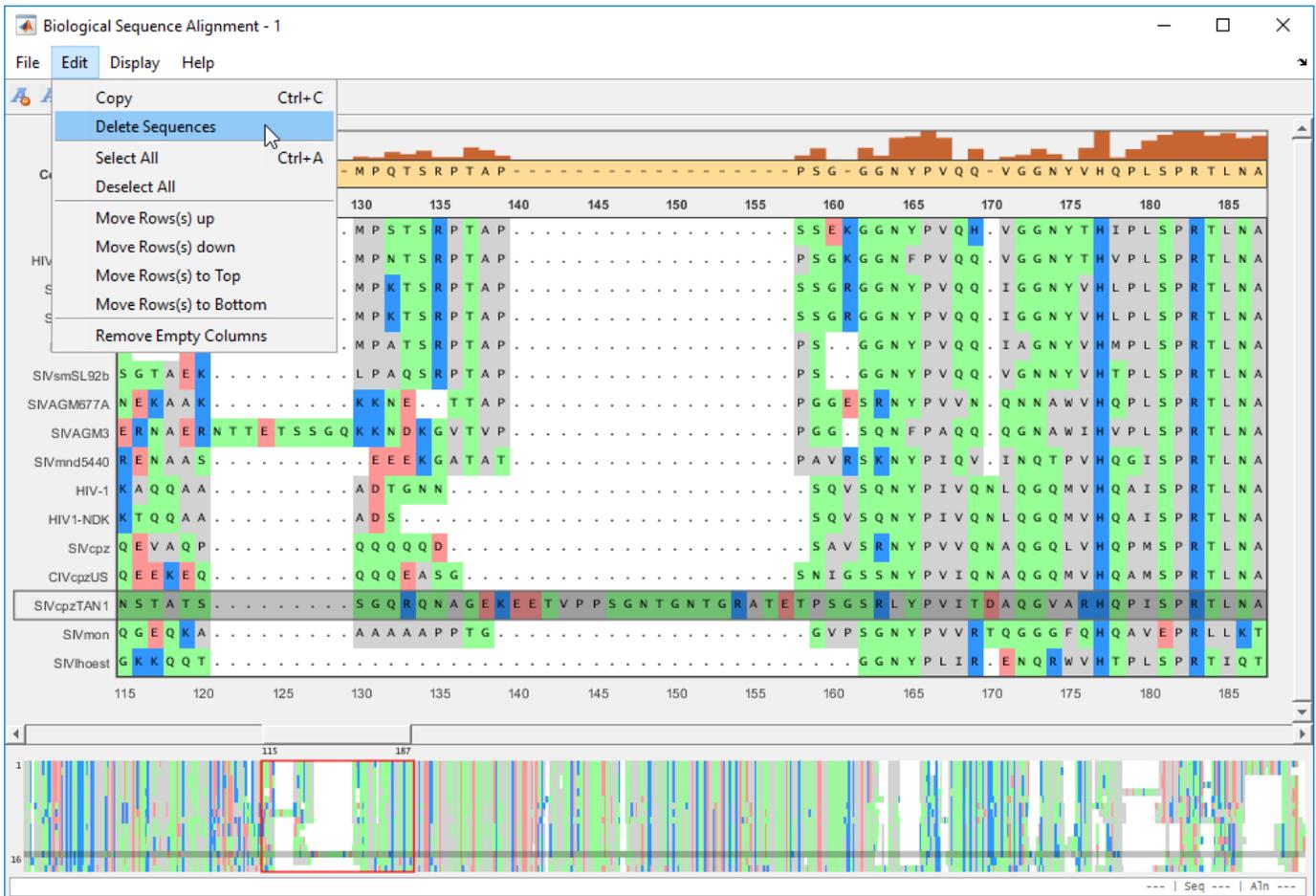
3 Sequence Analysis



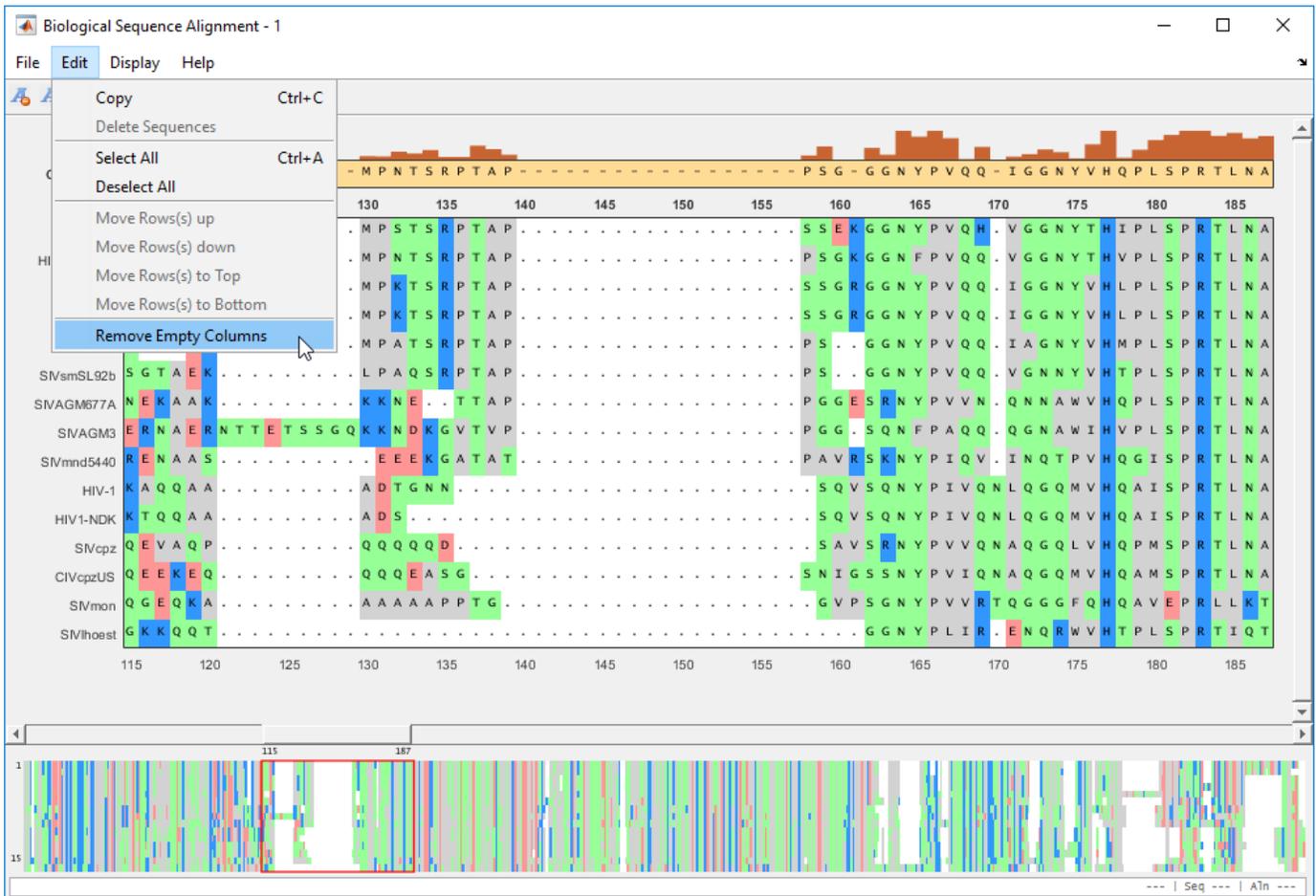
- 3 Click a letter or a region. The selected region is the center block. You can then drag the sequence(s) to the left or right of the center block.

The screenshot shows the 'Biological Sequence Alignment - 1' window. At the top, there is a menu bar with 'File', 'Edit', 'Display', and 'Help'. Below the menu bar is a toolbar with various icons. The main area displays a multiple sequence alignment. At the top, a 'Consensus' sequence is shown: A E Q G T A E K - - - - - M P Q T S R P T A P - - - - - P S G - G G N Y P V Q Q - V G G N Y V H Q P L S P R T L. Below this, individual sequences are listed, including HIV-2, HIV2-MCN13, SIVMM251, SIVMM239, HIV-2UC1, SIVsmSL92b, SIVAGM677A, SIVAGM3, SIVmnd5440, HIV-1, HIV1-NDK, SIVcpz, CIVcpzUS, SIVcpzTAN1, SIVmon, and SIVlhoest. The alignment is color-coded by amino acid type. A red box highlights a region of the alignment between positions 113 and 185. At the bottom, a vertical scale on the left indicates sequence positions from 1 to 16. The status bar at the bottom right shows 'SIVmnd5440 | Seq 9 | Aln 155'.

- 8 Suppose you want to remove one or more of the aligned sequences. First select the sequence(s) to be removed. Then select **Edit > Delete Sequences**.



9 Remove empty columns by selecting **Edit > Remove Empty Columns**.



10 After the edit, you can export the aligned sequences or consensus sequence to a FASTA file or MATLAB Workspace from the **File** menu.

Rearrange Rows

You can move the rows (sequences) up or down by one row. You can also move selected rows to the top or bottom of the list.



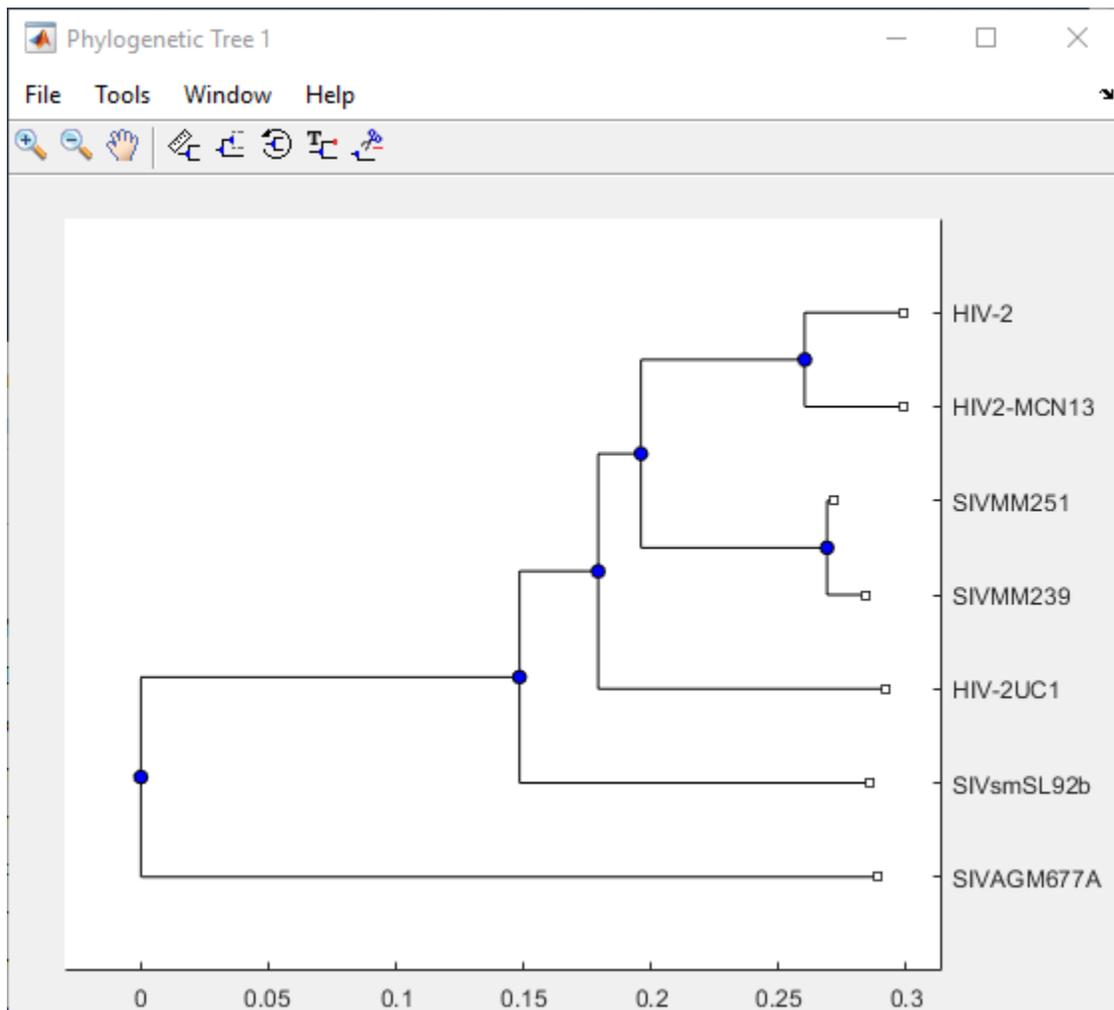
Generate Phylogenetic Tree from Aligned Sequences

You can generate a phylogenetic tree using the aligned sequences from within the app. You can select a subset of sequences or use all the sequences to generate a tree.

Select **Display > View Tree > Selected...** to generate a tree from selected sequences.



A phylogenetic tree for the sequences is displayed in the Phylogenetic Tree app. For details on the app, see "Using the Phylogenetic Tree App" on page 5-2.



See Also

seqalignviewer | Sequence Alignment | Genomics Viewer

More About

- "Sequence Alignments" on page 1-7
- "Aligning Pairs of Sequences" on page 3-138

Analyzing Synonymous and Nonsynonymous Substitution Rates

This example shows how the analysis of synonymous and nonsynonymous mutations at the nucleotide level can suggest patterns of molecular adaptation in the genome of HIV-1. This example is based on the discussion of natural selection at the molecular level presented in Chapter 6 of "Introduction to Computational Genomics. A Case Studies Approach" [1].

Introduction

The human immunodeficiency virus 1 (HIV-1) is the more geographically widespread of the two viral strains that cause Acquired Immunodeficiency Syndrome (AIDS) in humans. Because the virus rapidly and constantly evolves, at the moment there is no cure nor vaccine against HIV infection. The HIV virus presents a very high mutation rate that allows it to evade the response of our immune system as well as the action of specific drugs. At the same time, however, the rapid evolution of HIV provides a powerful mechanism that reveals important insights into its function and resistance to drugs. By estimating the force of selective pressures (positive and purifying selections) across various regions of the viral genome, we can gain a general understanding of how the virus evolves. In particular, we can determine which genes evolve in response to the action of the targeted immune system and which genes are conserved because they are involved in some of the virus essential functions.

Nonsynonymous mutations to a DNA sequence cause a change in the translated amino acid sequence, whereas synonymous mutations do not. The comparison between the number of nonsynonymous mutations (d_n or K_a), and the number of synonymous mutations (d_s or K_s), can suggest whether, at the molecular level, natural selection is acting to promote the fixation of advantageous mutations (positive selection) or to remove deleterious mutations (purifying selection). In general, when positive selection dominates, the K_a/K_s ratio is greater than 1; in this case, diversity at the amino acid level is favored, likely due to the fitness advantage provided by the mutations. Conversely, when negative selection dominates, the K_a/K_s ratio is less than 1; in this case, most amino acid changes are deleterious and, therefore, are selected against. When the positive and negative selection forces balance each other, the K_a/K_s ratio is close to 1.

Extracting Sequence Information for Two HIV-1 Genomes

Download two genomic sequences of HIV-1 (GenBank® accession numbers AF033819 and M27323). For each encoded gene we extract relevant information, such as nucleotide sequence, translated sequence and gene product name.

```
hiv1(1) = getgenbank('AF033819');  
hiv1(2) = getgenbank('M27323');
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load hiv1.mat
```

Extract the gene sequence information using the `featureparse` function.

```
genes1 = featureparse(hiv1(1), 'feature', 'CDS', 'Sequence', 'true');  
genes2 = featureparse(hiv1(2), 'feature', 'CDS', 'Sequence', 'true');
```

Calculating the Ka/Ks Ratio for HIV-1 Genes

Align the corresponding protein sequences in the two HIV-1 genomes and use the resulting alignment as a guide to insert the appropriate gaps in the nucleotide sequences. Then calculate the Ka/Ks ratio for each individual gene and plot the results.

```
KaKs = zeros(1,numel(genes1));
for iCDS = 1:numel(genes1)
    % align aa sequences of corresponding genes
    [score,alignment] = nwalignment(genes1(iCDS).translation,genes2(iCDS).translation);
    seq1 = seqinsertgaps(genes1(iCDS).Sequence,alignment(1,:));
    seq2 = seqinsertgaps(genes2(iCDS).Sequence,alignment(3,:));

    % Calculate synonymous and nonsynonymous substitution rates
    [dn,ds] = dnds(seq1,seq2);
    KaKs(iCDS) = dn/ds;
end

% plot Ka/Ks ratio for each gene
bar(KaKs);
ylabel('Ka / Ks')
xlabel('genes')
ax = gca;
ax.XTickLabel = {genes1.product};
% plot dotted line at threshold 1
hold on
line([0 numel(KaKs)+1],[1 1],'LineStyle',':');
KaKs
```

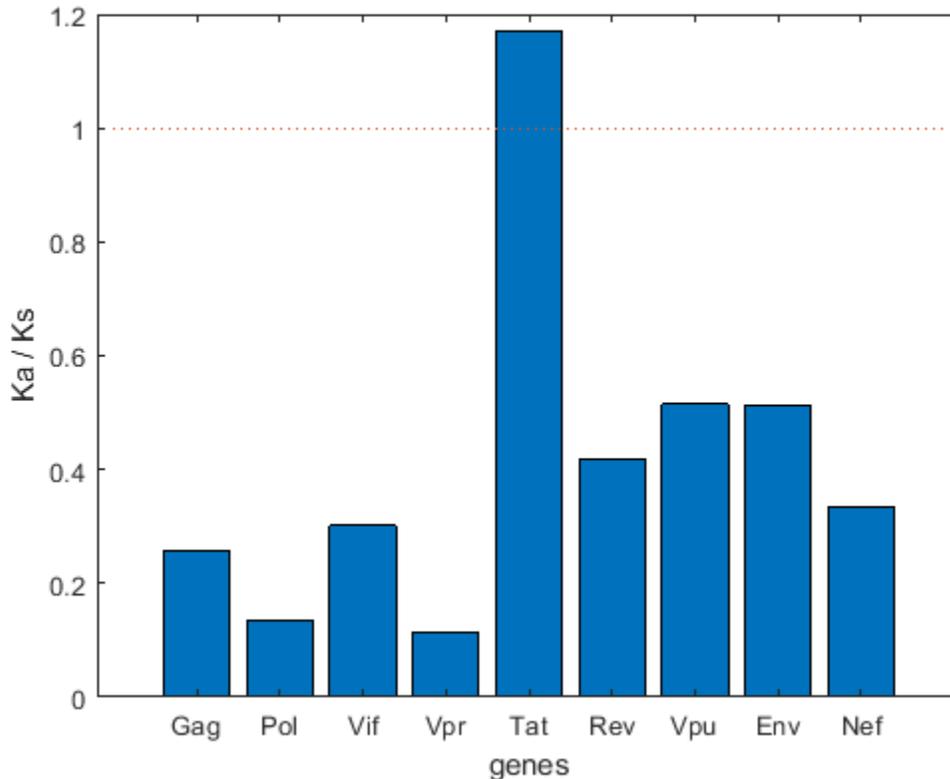
KaKs =

Columns 1 through 7

0.2560 0.1359 0.3013 0.1128 1.1686 0.4179 0.5150

Columns 8 through 9

0.5115 0.3338



All the considered genes, with the exception of TAT, have a total Ka/Ks less than 1. This is in accordance with the fact that most protein-coding genes are considered to be under the effect of purifying selection. Indeed, the majority of observed mutations are synonymous and do not affect the integrity of the encoded proteins. As a result, the number of synonymous mutations generally exceeds the number of nonsynonymous mutations. The case of TAT represents a well known exception; at the amino acid level, the TAT protein is one of the least conserved among the viral proteins.

Calculating the Ka/Ks Ratio Using Sliding Windows

Oftentimes, different regions of a single gene can be exposed to different selective pressures. In these cases, calculating Ka/Ks over the entire length of the gene does not provide a detailed picture of the evolutionary constraints associated with the gene. For example, the total Ka/Ks associated with the ENV gene is 0.5155. However, the ENV gene encodes for the envelope glycoprotein GP160, which in turn is the precursor of two proteins: GP120 (residues 31-511 in AF033819) and GP41 (residues 512-856 in AF033819). GP120 is exposed on the surface of the viral envelope and performs the first step of HIV infection; GP41 is non-covalently bonded to GP120 and is involved in the second step of HIV infection. Thus, we can expect these two proteins to respond to different selective pressures, and a global analysis on the entire ENV gene can obscure diversified behavior. For this reason, we conduct a finer analysis by using sliding windows of different sizes.

Align ENV genes of the two genomes and measure the Ka/Ks ratio over sliding windows of size equal to 5, 45, and 200 codons.

```
env = 8; % ORF number corresponding to gene ENV
```

```
% align the two ENV genes
```

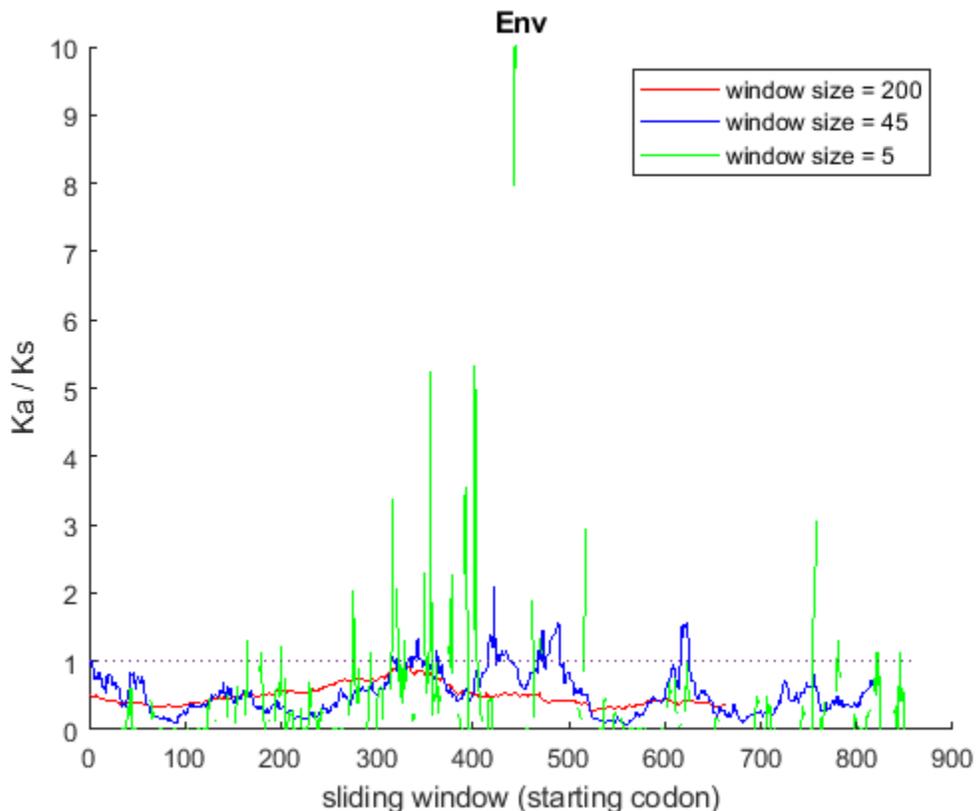
```

[score,alignment] = nwalignment(genes1(env).translation,genes2(env).translation);
env_1 = seqinsertgaps(genes1(env).Sequence,alignment(1,:));
env_2 = seqinsertgaps(genes2(env).Sequence,alignment(3,:));

% compute Ka/Ks using sliding windows of different sizes
[dn1, ds1, vardn1, vards1] = dnds(env_1, env_2, 'window', 200);
[dn2, ds2, vardn2, vards2] = dnds(env_1, env_2, 'window', 45);
[dn3, ds3, vardn3, vards3] = dnds(env_1, env_2, 'window', 5);

% plot the Ka/Ks trends for the different window sizes
figure()
hold on
plot(dn1./ds1, 'r');
plot(dn2./ds2, 'b');
plot(dn3./ds3, 'g');
line([0 numel(dn3)], [1 1], 'LineStyle', ':');
legend('window size = 200', 'window size = 45', 'window size = 5');
ylim([0 10])
ylabel('Ka / Ks')
xlabel('sliding window (starting codon)')
title 'Env';

```



The choice of the sliding window size can be problematic: windows that are too long (in this example, 200 codons) average across long regions of a single gene, thus hiding segments where Ka/Ks is potentially behaving in a peculiar manner. Too short windows (in this example, 5 codons) are likely to produce results that are very noisy and therefore not very meaningful. In the case of the ENV gene, a sliding window of 45 codons seems to be appropriate. In the plot, although the general trend is below

the threshold of 1, we observe several peaks over the threshold of 1. These regions appear to undergo positive selection that favors amino acid diversity, as it provides some fitness advantage.

Using Sliding Window Analyses for GAG, POL and ENV Genes

You can perform similar analyses on other genes that display a global Ka/Ks ratio less than 1. Compute the global Ka/Ks ratio for the GAG, POL and ENV genes. Then repeat the calculation using a sliding window.

```
gene_index = [1;2;8]; % ORF corresponding to the GAG, POL, ENV genes
windowSize = 45;

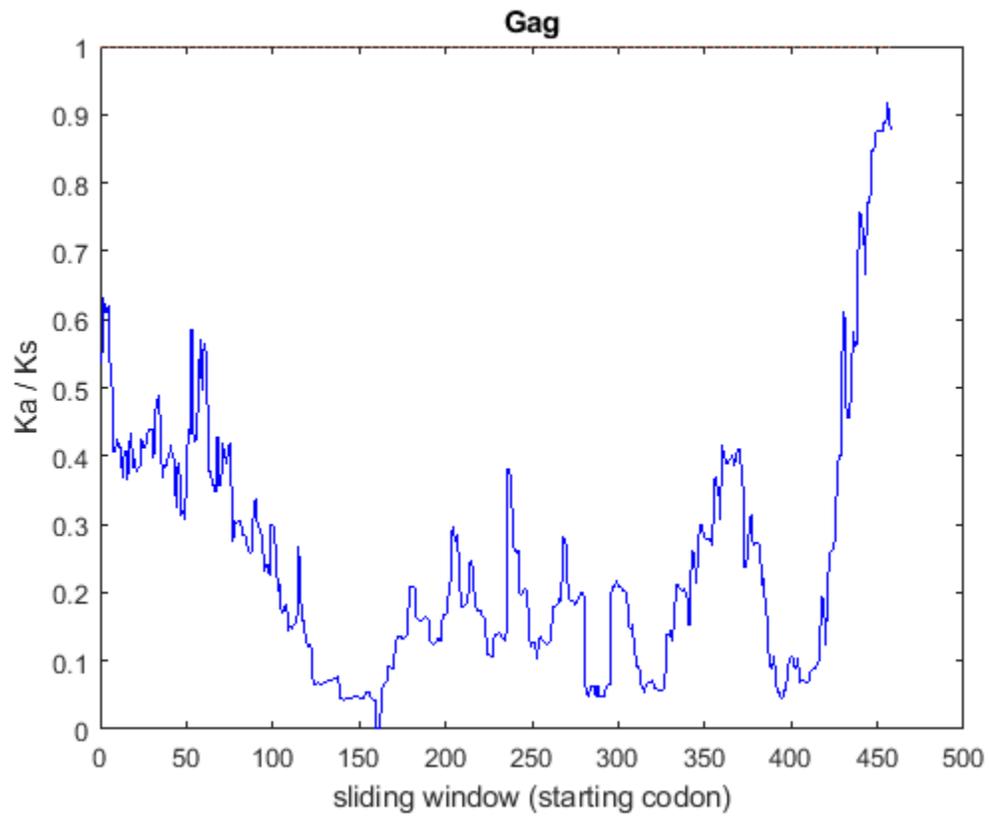
% display the global Ka/Ks for the GAG, POL and ENV genes
KaKs(gene_index)

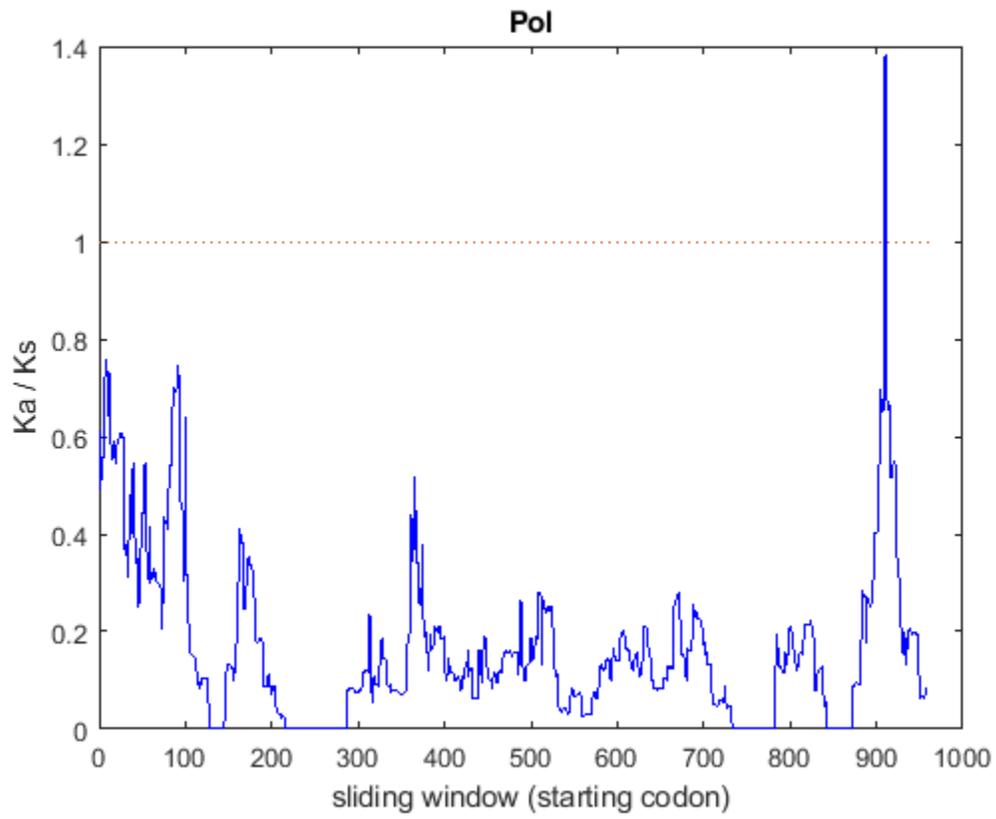
for i = 1:numel(gene_index)
    ID = gene_index(i);
    [score,alignment] = nwalign(genes1(ID).translation,genes2(ID).translation);
    s1 = seqinsertgaps(genes1(ID).Sequence,alignment(1,:));
    s2 = seqinsertgaps(genes2(ID).Sequence,alignment(3,:));

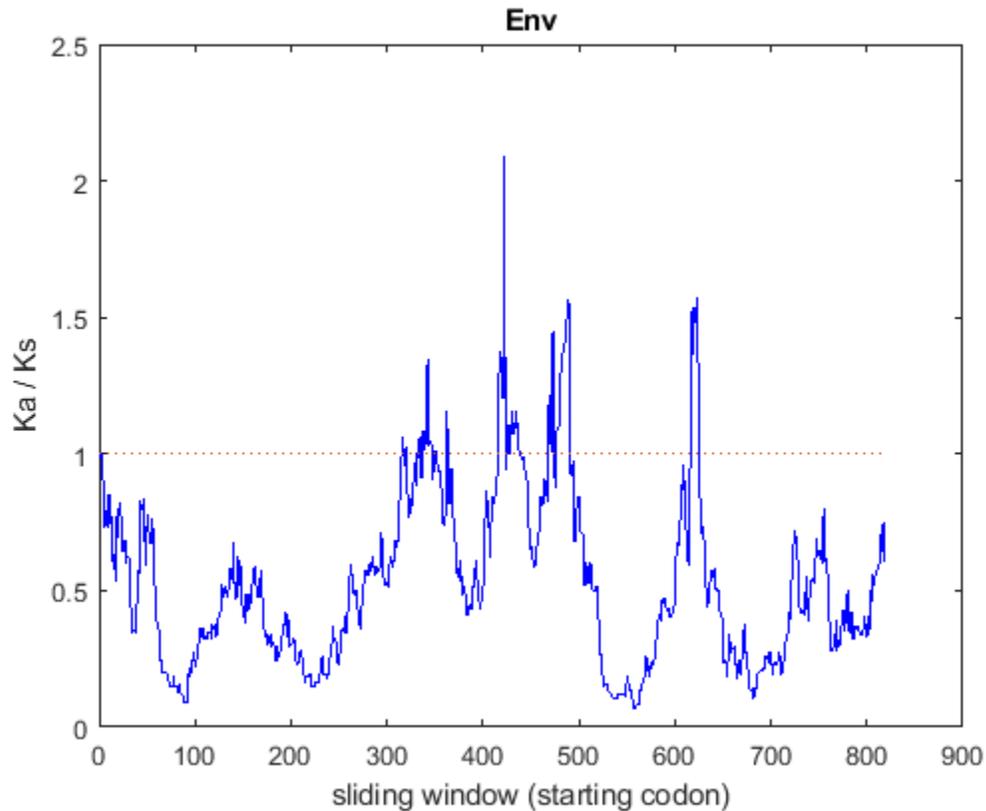
    % plot Ka/Ks ratio obtained with the sliding window
    [dn, ds, vardn, vards] = dnds(s1, s2, 'window', windowSize);
    figure()
    plot(dn./ds, 'b')
    line([0 numel(dn)], [1 1], 'LineStyle', ':')
    ylabel('Ka / Ks')
    xlabel('sliding window (starting codon)')
    title(genes1(ID).product);
end

ans =

    0.2560    0.1359    0.5115
```







The GAG (Group-specific Antigen) gene provides the basic physical infrastructure of the virus. It codes for p24 (the viral capsid), p6 and p7 (the nucleocapsid proteins), and p17 (a matrix protein). Since this gene encodes for many fundamental proteins that are structurally important for the survival of the virus, the number of synonymous mutations exceeds the number of nonsynonymous mutations (i.e., $Ka/Ks < 1$). Thus, this protein is expected to be constrained by purifying selection to maintain viral infectivity.

The POL gene codes for viral enzymes, such as reverse transcriptase, integrase, and protease. These enzymes are essential to the virus survival and, therefore, the selective pressure to preserve their function and structural integrity is quite high. Consequently, this gene appears to be under purifying selection and we observe Ka/Ks ratio values less than 1 for the majority of the gene length.

The ENV gene codes for the precursor to GP120 and GP41, proteins embedded in the viral envelope, which enable the virus to attach to and fuse with target cells. GP120 infects any target cell by binding to the CD4 receptor. As a consequence, GP120 has to maintain the mechanism of recognition of the host cell and at the same time avoid the detection by the immune system. These two roles are carried out by different parts of the protein, as shown by the trend in the Ka/Ks ratio. This viral protein is undergoing purifying ($Ka/Ks < 1$) and positive selection ($Ka/Ks > 1$) in different regions. A similar trend is observed in GP41.

Analyzing the Ka/Ks Ratio and Epitopes in GP120

The glycoprotein GP120 binds to the CD4 receptor of any target cell, particularly the helper T-cell. This represents the first step of HIV infection and, therefore, GP120 was among the first proteins studied with the intent of finding a HIV vaccine. It is interesting to determine which regions of GP120

appear to undergo purifying selection, as indicators of protein regions that are functionally or structurally important for the virus survival, and could potentially represent drug targets.

From ENV genes, extract the sequences coding for GP120. Compute the Ka/Ks over sliding window of size equal to 45 codons. Plot and overlap the trend of Ka/Ks with the location of four T cell epitopes for GP120.

```
% GP120 protein boundaries in genome1 and genome2 respectively
gp120_start = [31; 30]; % protein boundaries
gp120_stop = [511; 501];
gp120_startnt = gp120_start*3-2; % nt boundaries
gp120_stopnt = gp120_stop*3;

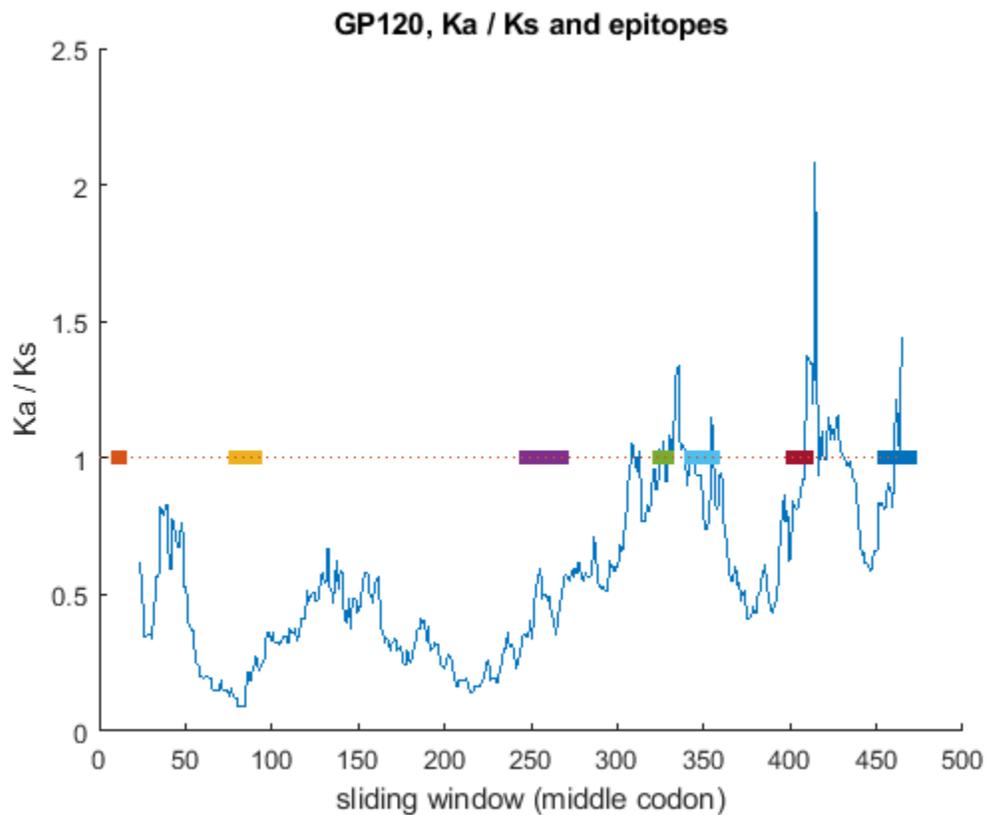
% align GP120 proteins and insert appropriate gaps in nt sequence
[score,alignment] = nwalignment(genes1(env).translation(gp120_start(1):gp120_stop(1)), ...
                               genes2(env).translation(gp120_start(2):gp120_stop(2)));
gp120_1 = seqinsertgaps(genes1(env).Sequence(gp120_startnt(1):gp120_stopnt(1)),alignment(1,:));
gp120_2 = seqinsertgaps(genes2(env).Sequence(gp120_startnt(2):gp120_stopnt(2)),alignment(3,:));

% Compute and plot Ka/Ks ratio using the sliding window
[dn120, ds120, vardn120, vards120] = dnds(gp120_1, gp120_2, 'window', windowSize);

% Epitopes for GP120 identified by cellular methods (see reference [2])
epitopes = {'TVYYGVPVWK', 'HEDIISLWQSLKPCVKLTPL', ...
            'EVVIRSANFTNDAKATIIVQLNQSVEINCT', 'QIASKLREQFGNNK', ...
            'QSSGGDPEIVTHSFNCGGEFF', 'KQFINMWQEVGKAMYAPP', ...
            'DMRDNWRSELYKYKVVKIEPLGVAP'};

% Find location of the epitopes in the aligned sequences:
epiLoc = zeros(numel(epitopes),2);
for i = 1:numel(epitopes)
    [sco,ali,ind] = swalign(alignment(1,:),epitopes{i});
    epiLoc(i,:) = ind(1) + [0 length(ali)-1];
end

figure
hold on
% plot Ka/Ks relatively to the middle codon of the sliding window
plot(windowSize/2+(1:numel(dn120)),dn120./ds120)
plot(epiLoc,[1 1],'linewidth',5)
line([0 numel(dn120)+windowSize/2],[1 1],'LineStyle',':')
title('GP120, Ka / Ks and epitopes');
ylabel('Ka / Ks');
xlabel('sliding window (middle codon)');
```



Although the general trend of the Ka/Ks ratio is less than 1, there are some regions where the ratio is greater than one, indicating that these regions are likely to be under positive selection. Interestingly, the location of some of these regions corresponds to the presence of T cell epitopes, identified by cellular methods. These segments display high amino acid variability because amino acid diversity in these regions allows the virus to evade the host immune system recognition. Thus, we can conclude that the source of variability in this regions is likely to be the host immune response.

References

- [1] Cristianini, N. and Hahn, M.W., "Introduction to Computational Genomics: A Case Studies Approach", Cambridge University Press, 2007.
- [2] Siebert, S.A., et al., "Natural Selection on the gag, pol, and env Genes of Human Immunodeficiency Virus 1 (HIV-1)", *Molecular Biology and Evolution*, 12(5):803-813, 1995.

Investigating the Bird Flu Virus

This example shows how to calculate Ka/Ks ratios for eight genes in the H5N1 and H2N3 virus genomes, and perform a phylogenetic analysis on the HA gene from H5N1 virus isolated from chickens across Africa and Asia. For the phylogenetic analysis, you will reconstruct a neighbor-joining tree and create a 3-D plot of sequence distances using multidimensional scaling. Finally, you will map the geographic locations where each HA sequence was found on a regional map. Sequences used in this example were selected from the bird flu case study on the Computational Genomics Website [1]. Note: The final section in this example requires the Mapping Toolbox™.

Introduction

There are three types of influenza virus: Type A, B and C. All influenza genomes are comprised of eight segments or genes that code for polymerase B2 (PB2), polymerase B1 (PB1), polymerase A (PA), hemagglutinin (HA), nucleoprotein (NP), neuraminidase (NA), matrix (M1), and non-structural (NS1) proteins. Note: Type C virus has hemagglutinin-esterase (HE), a homolog to HA.

Of the three types of influenza, Type A has the potential to be the most devastating. It affects birds (its natural reservoir), humans and other mammals and has been the major cause of global influenza epidemics. Type B affects only humans causing local epidemics, and Type C does not tend to cause serious illness.

Type A influenzas are further classified into different subtypes according to variations in the amino acid sequences of HA (H1-16) and NA (N1-9) proteins. Both proteins are located on the outside of the virus. HA attaches the virus to the host cell then aids in the process of the virus being fused in to the cell. NA clips the newly created virus from the host cell so it can move on to a healthy new cell. Difference in amino acid composition within a protein and recombination of the various HA and NA proteins contribute to Type A influenzas' ability to jump host species (i.e. bird to humans) and wide range of severity. Many new drugs are being designed to target HA and NA proteins [2,3,4].

In 1997, H5N1 subtype of the avian influenza virus, a Type A influenza virus, made an unexpected jump to humans in Hong Kong causing the deaths of six people. To control the rapidly spreading disease, all poultry in Hong Kong was destroyed. Sequence analysis of the H5N1 virus is shown here [2,4].

Calculate Ka/Ks Ratio For Each H5N1 Gene

An investigation of the Ka/Ks ratios for each gene segment of the H5N1 virus will provide some insight into how each is changing over time. Ka/Ks is the ratio of non-synonymous changes to synonymous in a sequence. For a more detailed explanation of Ka/Ks ratios, see “Analyzing Synonymous and Nonsynonymous Substitution Rates” on page 3-36. To calculate Ka/Ks, you need a copy of the gene from two time points. You can use H5N1 virus isolated from chickens in Hong Kong in 1997 and 2001. For comparison, you can include H2N3 virus isolated from mallard ducks in Alberta in 1977 and 1985 [1].

For the purpose of this example, sequence data is provided in four MATLAB® structures that were created by `genbankread`.

Load H5N1 and H2N3 sequence data.

```
load('birdflu.mat','chicken1997','chicken2001','mallard1977','mallard1985')
```

Data in public repositories is frequently curated and updated. You can retrieve the up-to-date datasets by using the `getgenbank` function. Note that if data has indeed changed, the results of this example might be slightly different when you use up-to-date datasets.

```
chicken1997 = arrayfun(@(x)getgenbank(x{:}),{chicken1997.Accession});
chicken2001 = arrayfun(@(x)getgenbank(x{:}),{chicken2001.Accession});
mallard1977 = arrayfun(@(x)getgenbank(x{:}),{mallard1977.Accession});
mallard1985 = arrayfun(@(x)getgenbank(x{:}),{mallard1985.Accession});
```

You can extract just the coding portion of the nucleotide sequences using the `featureparse` function. The `featureparse` function returns a structure with fields containing information from the Features section in a GenBank file including with a Sequence field that contains just the coding sequence.

```
for ii = 1:numel(chicken1997)
    ntSeq97{ii} = featureparse(chicken1997(ii), 'feature', 'cds', 'sequence', true);
    ntSeq01{ii} = featureparse(chicken2001(ii), 'feature', 'cds', 'sequence', true);
    ntSeq77{ii} = featureparse(mallard1977(ii), 'feature', 'cds', 'sequence', true);
    ntSeq85{ii} = featureparse(mallard1985(ii), 'feature', 'cds', 'sequence', true);
end

ntSeq97{1}

ans =

    struct with fields:
        Location: '<1..>2273'
        Indices: [1 2273]
        UnknownFeatureBoundaries: 1
        gene: 'PB2'
        codon_start: '1'
        product: 'PB2 protein'
        protein_id: 'AAF02361.1'
        db_xref: 'GI:6048850'
        translation: 'RIKELRDLMSQSR TREILTKTTVDHMAIIKKYTSGRQEKNPALRMKWMAMKYPITADKRIMEMII
        Sequence: 'agaataaaagaactaagagatttgatgctgcaatctcgcacacgcgagatactgacaaaaccac'
```

Visual inspection of the sequence structures revealed some of the genes have splice variants represented in the GenBank files. Because this analysis is only on PB2, PB1, PA, HA, NP, NA, M1, and NS1 genes, you need to remove any splice variants.

Remove splice variants from 1997 H5N1

```
ntSeq97{7}(1) = [];% M2
ntSeq97{8}(1) = [];% NS2
```

Remove splice variants from 1977 H2N3

```
ntSeq77{2}(2) = [];% PB1-F2
ntSeq77{7}(1) = [];% M2
ntSeq77{8}(1) = [];% NS2
```

Remove splice variants from 1985 H2N3

```
ntSeq85{2}(2) = [];% PB1-F2
ntSeq85{7}(1) = [];% M2
ntSeq85{8}(1) = [];% NS2
```

You need to align the nucleotide sequences to calculate the Ka/Ks ratio. Align protein sequences for each gene (available in the 'translation' field) using `nwalign` function, then insert gaps into nucleotide sequence using `seqinsertgaps`. Use the function `dnds` to calculate non-synonymous and synonymous substitution rates for each of the eight genes in the virus genomes. If you are interested in seeing the sequence alignments, set the 'verbose' option to true when using `dnds`.

Influenza gene names

```
proteins = {'PB2', 'PB1', 'PA', 'HA', 'NP', 'NA', 'M1', 'NS1'};
```

H5N1 Virus

```
for ii = 1:numel(ntSeq97)
    [sc,align] = nwalign(ntSeq97{ii}.translation,ntSeq01{ii}.translation,'alpha','aa');
    ch97seq = seqinsertgaps(ntSeq97{ii}.Sequence,align(1,:));
    ch01seq = seqinsertgaps(ntSeq01{ii}.Sequence,align(3,:));
    [dn,ds] = dnds(ch97seq,ch01seq);
    H5N1.(proteins{ii}) = dn/ds;
end
```

H2N3 Virus

```
for ii = 1:numel(ntSeq77)
    [sc,align] = nwalign(ntSeq77{ii}.translation,ntSeq85{ii}.translation,'alpha','aa');
    ch77seq = seqinsertgaps(ntSeq77{ii}.Sequence,align(1,:));
    ch85seq = seqinsertgaps(ntSeq85{ii}.Sequence,align(3,:));
    [dn,ds] = dnds(ch77seq,ch85seq);
    H2N3.(proteins{ii}) = dn/ds;
end
```

H5N1
H2N3

H5N1 =

struct with fields:

```
PB2: 0.0226
PB1: 0.0240
PA: 0.0307
HA: 0.0943
NP: 0.0517
NA: 0.1015
M1: 0.0460
NS1: 0.3010
```

H2N3 =

struct with fields:

```
PB2: 0.0048
```

```

PB1: 0.0021
PA: 0.0089
HA: 0.0395
NP: 0.0071
NA: 0.0559
M1: 0
NS1: 0.1954

```

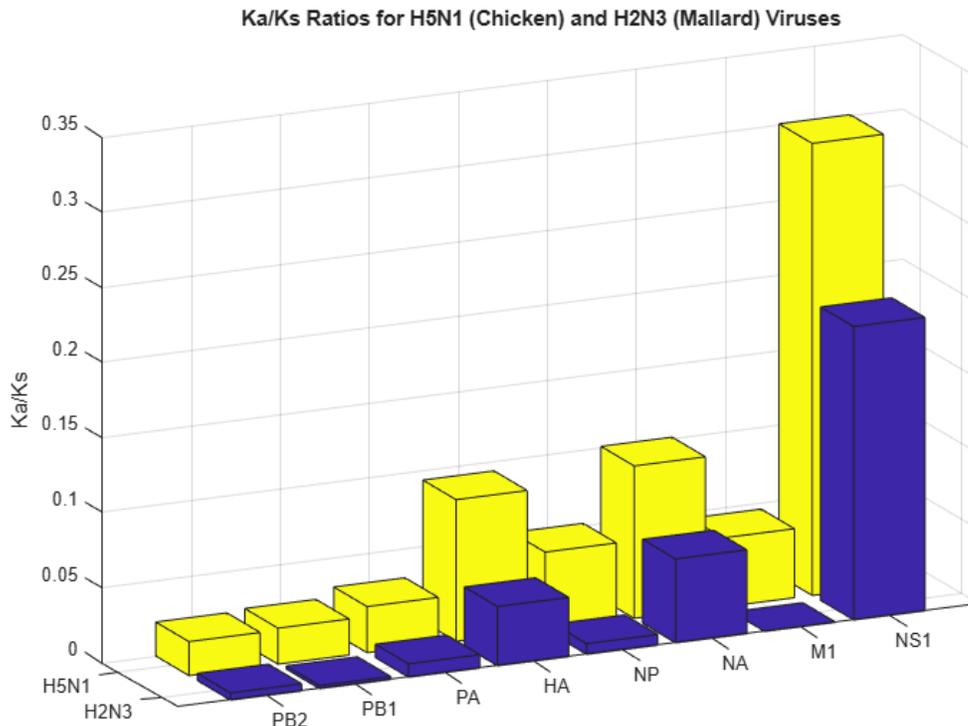
Note: Ka/Ks ratio results may vary from those shown on [1] due to sequence splice variants.

Visualize Ka/Ks ratios in 3-D bar graph.

```

H5N1rates = cellfun(@(x)(H5N1.(x)),proteins);
H2N3rates = cellfun(@(x)(H2N3.(x)),proteins);
bar3([H2N3rates' H5N1rates']);
ax = gca;
ax.XTickLabel = {'H2N3','H5N1'};
ax.YTickLabel = proteins;
zlabel('Ka/Ks');
view(-115,16);
title('Ka/Ks Ratios for H5N1 (Chicken) and H2N3 (Mallard) Viruses');

```



NS1, HA and NA have larger non-synonymous to synonymous ratios compared to the other genes in both H5N1 and H2N3. Protein sequence changes to these genes have been attributed to an increase in H5N1 pathogenicity. In particular, changes to the HA gene may provide the virus the ability to transfer into others species beside birds [2,3].

Perform a Phylogenetic Analysis of the HA Protein

The H5N1 virus attaches to cells in the gastrointestinal tract of birds and the respiratory tract of humans. Changes to the HA protein, which helps bind the virus to a healthy cell and facilitates its incorporation into the cell, are what allow the virus to affect different organs in the same and different species. This may provide it the ability to jump from birds to humans [2,3]. You can perform a phylogenetic analysis of the HA protein from H5N1 virus isolated from chickens at different times (years) in different regions of Asia and Africa to investigate their relationship to each other.

Load HA amino acid sequence data from 16 regions/times from the MAT-file provided `birdflu.mat` or retrieve the up-to-date sequence data from the NCBI repository using the `getgenpept` function.

```
load('birdflu.mat', 'HA')
```

```
HA = arrayfun(@(x) getgenpept(x{:}), {HA.Accession});
```

Create a new structure array containing fields corresponding to amino acid sequence (Sequence) and source information (Header). You can extract source information from the HA using `featureparse` then parse with `regexp`.

```
for ii = 1:numel(HA)
    source = featureparse(HA(ii), 'feature', 'source');
    strain = regexp(source.strain, 'A/[Cc]hicken/(\w+\s*\w*).*?/(\d+)', 'tokens');
    proteinHA(ii).Header = sprintf('%s_%s', char(strain{1}(1)), char(strain{1}(2)));
    proteinHA(ii).Sequence = HA(ii).Sequence;
end
```

```
proteinHA(1)
```

```
ans =
```

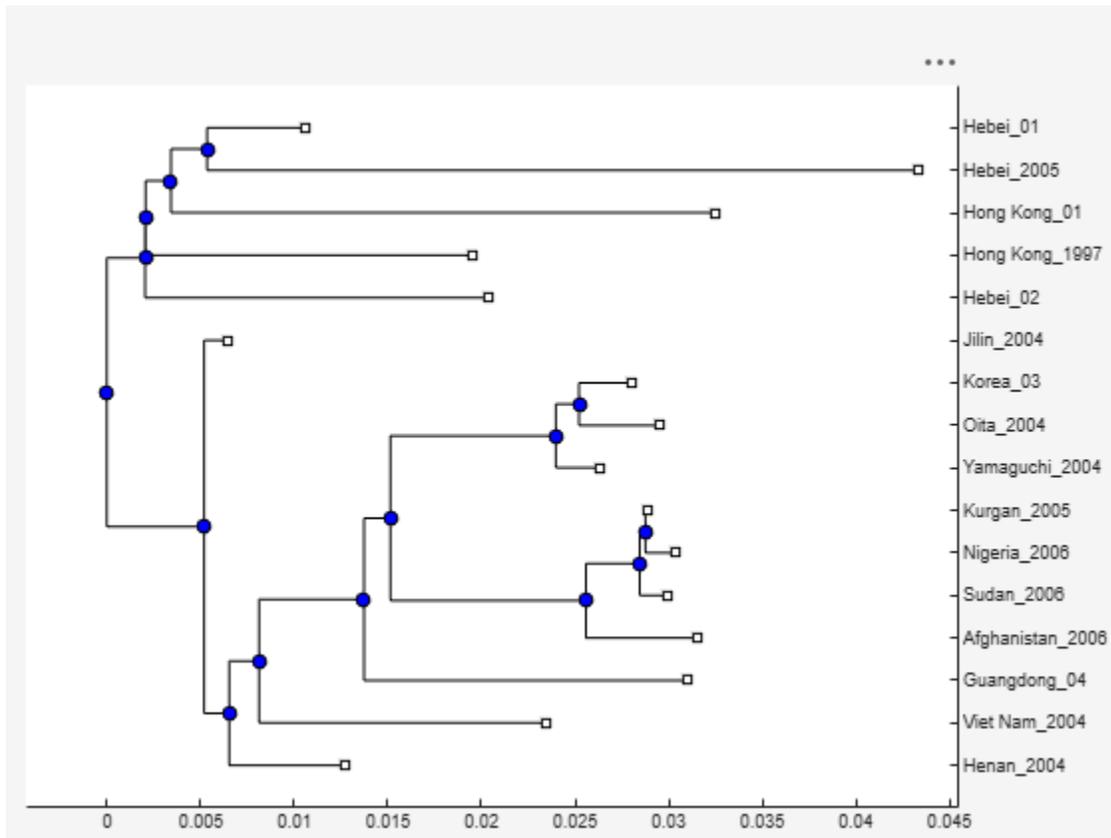
```
    struct with fields:
```

```
        Header: 'Nigeria_2006'
```

```
        Sequence: 'mekivllfaivglvksdqicigyhannsteqvdtimknavtvthaqdklekthngklcdldgvykplilrdcsvagwllgnpr'
```

Align the HA amino acid sequences using `multialign` and visualize the alignment with `seqalignviewer`.

```
alignHA = multialign(proteinHA);
seqalignviewer(alignHA);
```

Visualize Sequence Distances with Multidimensional Scaling (MDS)

Another way to visualize the relationship between sequences is to use multidimensional scaling (MDS) with the distances calculated for the phylogenetic tree. This functionality is provided by the `cmdscale` function in Statistics and Machine Learning Toolbox™.

```
[Y,eigvals] = cmdscale(distHA);
```

You can use the eigenvalues returned by `cmdscale` to help guide your decision of whether to use the first two or three dimensions in your plot.

```
sigVecs = [1:3;eigvals(1:3)';eigvals(1:3)'/max(abs(eigvals))];
report = ['Dimension Eigenvalues Normalized' ...
         sprintf('\n %d\t %1.4f %1.4f',sigVecs)];
display(report);
```

```
report =
```

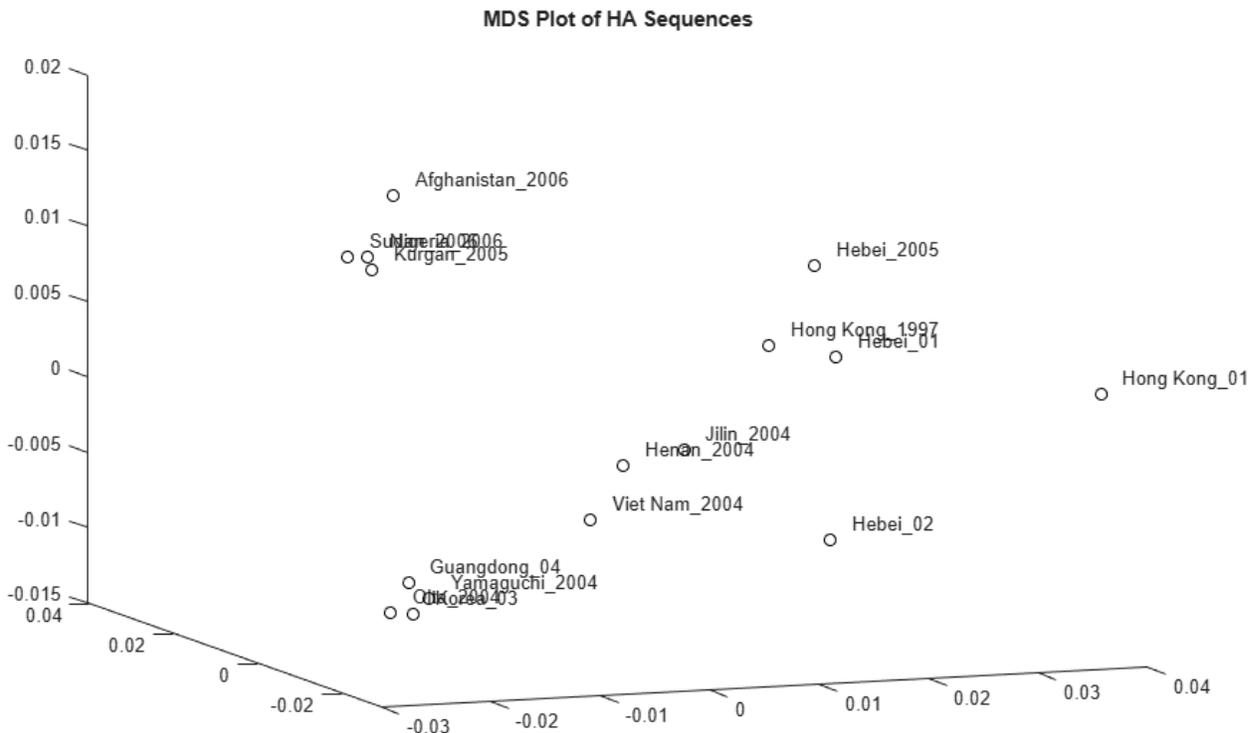
'Dimension	Eigenvalues	Normalized
1	0.0062	1.0000
2	0.0028	0.4462
3	0.0014	0.2209'

The first two dimensions represent a large portion of the data, but the third still contains information that might help resolve clusters in the sequence data. You can create a three dimensional scatter plot using `plot3` function.

```

locations = {proteinHA(:).Header};
figure
plot3(Y(:,1),Y(:,2),Y(:,3), 'ok');
text(Y(:,1)+0.002,Y(:,2),Y(:,3)+0.001,locations, 'interpreter', 'no');
title('MDS Plot of HA Sequences');
view(-21,12);

```



Clusters appear to correspond to groupings in the phylogenetic tree. Find the sequences belonging to each cluster using the `subtree` method of `phytree`. One of `subtree`'s required inputs is the node number (number of leaves + number of branches), which will be the new subtree's root node. For your example, the cluster containing Hebei and Hong Kong in the MDS plot is equivalent to the subtree whose root node is Branch 14, which is Node 30 (16 leaves + 14 branches).

```

cluster1 = get(subtree(HA_NJtree,30), 'LeafNames');
cluster2 = get(subtree(HA_NJtree,21), 'LeafNames');
cluster3 = get(subtree(HA_NJtree,19), 'LeafNames');

```

Get an index for the sequences belonging to each cluster.

```

[cl1,cl1_ind] = intersect(locations,cluster1);
[cl2,cl2_ind] = intersect(locations,cluster2);
[cl3,cl3_ind] = intersect(locations,cluster3);
[cl4,cl4_ind] = setdiff(locations,{cl1{:} cl2{:} cl3{:}});

```

Change the color and marker symbols on the MDS plot to correspond to each cluster.

```

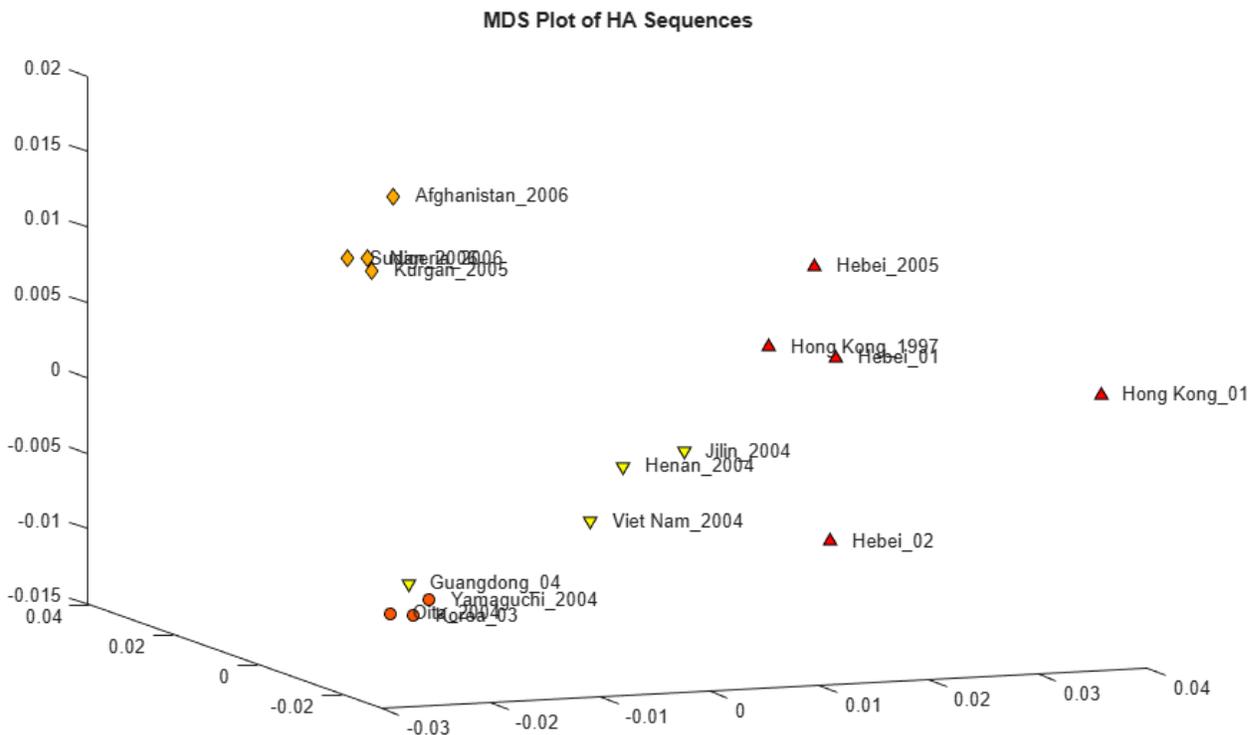
h = plot3(Y(cl1_ind,1),Y(cl1_ind,2),Y(cl1_ind,3), '^', ...
          Y(cl2_ind,1),Y(cl2_ind,2),Y(cl2_ind,3), 'o', ...

```

```

        Y(cl3_ind,1),Y(cl3_ind,2),Y(cl3_ind,3),'d',...
        Y(cl4_ind,1),Y(cl4_ind,2),Y(cl4_ind,3),'v');
numClusters = 4;
col = autumn(numClusters);
for i = 1:numClusters
    h(i).MarkerFaceColor = col(i,:);
end
set(h(:),'MarkerEdgeColor','k');
text(Y(:,1)+0.002,Y(:,2),Y(:,3),locations,'interpreter','no');
title('MDS Plot of HA Sequences');
view(-21,12);

```



For more detailed information on using Ka/Ks ratios, phylogenetics and MDS for sequence analysis, see Cristianini and Hahn [5].

Optional: Display Geographic Regions of the H5N1 Virus on a Map of Africa and Asia

If you have Mapping toolbox, you can run the following code to generate a figure that shows the geographic locations of the virus.

Create a geostruct structure, `regionHA`, that contains the geographic information for each feature, or sequence, to be displayed. A geostruct is required to have `Geometry`, `Lat`, and `Lon` fields that specify the feature type, latitude and longitude. This information is used by mapping functions in Mapping Toolbox to display geospatial data.

```

[regionHA(1:16).Geometry] = deal('Point');
[regionHA(:).Lat] = deal(9.10, 34.31, 15.31, 39.00, 39.00, 39.00, 55.26,...
    15.56, 34.00, 33.14, 34.20, 23.00, 37.35, 44.00,...

```

```

                22.11, 22.11);
[regionHA(:).Lon] = deal(7.10, 69.08, 32.35, 116.00, 116.00, 116.00,...
                        65.18, 105.48, 114.00, 131.36, 131.40, 113.00,...
                        127.00, 127.00, 114.14, 114.14);

```

A geostruct can also have attribute fields that contain additional information about each feature. Add attribute fields Name and Cluster to the regionHA structure. The Cluster field contains the sequence's cluster number, which you will use to identify the sequences' cluster membership.

```

[regionHA(:).Name] = deal(proteinHA.Header);
[regionHA(cl1_ind).Cluster] = deal(1);
[regionHA(cl2_ind).Cluster] = deal(2);
[regionHA(cl3_ind).Cluster] = deal(3);
[regionHA(cl4_ind).Cluster] = deal(4);

```

Create a structure using the makesymbolspec function, which will contain marker and color specifications for each marker to be displayed on the map. You will pass this structure to the geoshow function. Symbol markers and colors are set to correspond with the clusters in MDS plot.

```

clusterSymbols = makesymbolspec('Point',...
    {'Cluster',1,'Marker', '^'},...
    {'Cluster',2,'Marker', 'o'},...
    {'Cluster',3,'Marker', 'd'},...
    {'Cluster',4,'Marker', 'v'},...
    {'Cluster',[1 4],'MarkerFaceColor',autumn(4)},...
    {'Default','MarkerSize', 6},...
    {'Default','MarkerEdgeColor','k'});

```

Load the mapping information and use the geoshow function to plot virus locations on a map.

```

load coastlines
load topo
figure
fig = gcf;
worldmap([-45 85],[0 160])
setm(gca,'mapprojection','robinson',...
    'plabellocation',30,'mlabelparallel',-45,'mlabellocation',30)
geoshow(coastlat,coastlon)
geoshow(topo, topolegend, 'DisplayType', 'texturemap')
demcmap(topo)
brighten(.60)
geoshow(regionHA,'SymbolSpec',clusterSymbols);
title('Geographic Locations of HA Sequence in Africa and Asia')

close all

```

References

- [1] https://computationalgenomics.blogs.bristol.ac.uk/case_studies/birdflu_demo
- [2] Laver, W.G., Bischofberger, N. and Webster, R.G., "Disarming Flu Viruses", Scientific American, 280(1):78-87, 1999.
- [3] Suzuki, Y. and Masatoshi, N., "Origin and Evolution of Influenza Virus Hemagglutinin Genes", Molecular Biology and Evolution, 19(4):501-9, 2002.
- [4] Gambaryan, A., et al., "Evolution of the receptor binding phenotype of influenza A(H5) viruses", Virology, 344(2):432-8, 2006.

[5] Cristianini, N. and Hahn, M.W., "Introduction to Computational Genomics: A Case Studies Approach", Cambridge University Press, 2007.

[6] Google Earth images were acquired using Google Earth Pro. For more information about Google Earth and Google Earth Pro, visit <http://earth.google.com/>

Exploring Primer Design

This example shows how to use the Bioinformatics Toolbox™ to find potential primers that can be used for automated DNA sequencing.

Introduction

Primer design for PCR can be a daunting task. A good primer should usually meet the following criteria:

- Length is 18-30 bases.
- Melting temperature is 50-60 degrees Celsius.
- GC content is between 45% and 55%.
- Does not form stable hairpins.
- Does not self dimerize.
- Does not cross dimerize with other primers in the reaction.
- Has a GC clamp at the 3' end of the primer.

This example uses MATLAB® and Bioinformatics Toolbox to find PCR primers for the human hexosaminidase gene.

First load the hexosaminidase nucleotide sequence from the provided MAT-file `hexosaminidase.mat`. The DNA sequence that you want to find primers for is in the `Sequence` field of the loaded structure.

```
load('hexosaminidase.mat','humanHEXA')
sequence = humanHEXA.Sequence;
```

You can also use the `getgenbank` function to retrieve the sequence information from the NCBI data repository and load it into MATLAB. The NCBI reference sequence for HEXA has accession number `NM_000520`.

```
humanHEXA = getgenbank('NM_000520');
```

Calculating Properties of an Oligonucleotide

The `oligoprop` function is a useful tool to get properties of oligonucleotide DNA sequences. This function calculates the GC content, molecular weight, melting temperature, and secondary structure information. `oligoprop` returns a structure that has fields with the associated information. Use the `help` command to see what other properties `oligoprop` returns.

```
nt = oligoprop('AAGCTCAAAAACGCGCGGTATTCGACTGGCGTGATCTATTTTATGCT')
```

```
nt =
```

```
struct with fields:
```

```
GC: 44.6809
GCdelta: 0
Hairpins: [3×47 char]
Dimers: [9×47 char]
MolWeight: 1.4468e+04
```

```
MolWeightdelta: 0
      Tm: [68.9128 79.7752 85.3393 69.6497 68.2474 75.8931]
      Tmdelta: [0 0 0 0 0 0]
      Thermo: [4×3 double]
      Thermodelta: [4×3 double]
```

Finding All Potential Forward Primers

The goal is to create a list of all potential forward primers of length 20. You can do this either by iterating over the entire sequence and taking subsequences at every possible position or by using a matrix of indices. The following example shows how you can set a matrix of indices and then use it to create all possible forward subsequences of length 20, in this case N-19 subsequences where N is the length of the target hexosaminidase sequence. Then you can use the `oligoprop` function to get properties for each of the potential primers in the list.

```
N = length(sequence) % length of the target sequence
M = 20 % desired primer length
index = repmat((0:N-M)',1,M)+repmat(1:M,N-M+1,1);
fwdprimerlist = sequence(index);

for i = N-19:-1:1 % reverse order to pre-allocate structure
    fwdprimerprops(i) = oligoprop(fwdprimerlist(i,:));
end
```

```
N =
    2437
```

```
M =
    20
```

Finding All Potential Reverse Primers

After you have all potential forward primers, you can search for reverse primers in a similar fashion. Reverse primers are found on the complementary strand. To obtain the complementary strand use the `seqcomplement` function.

```
comp_sequence = seqcomplement(sequence);
revprimerlist = seqreverse(comp_sequence(index));

for i = N-19:-1:1 % reverse order to preallocate structure
    revprimerprops(i) = oligoprop(revprimerlist(i,:));
end
```

Filtering Primers Based on GC Content

The GC content information for the primers is in a structure with the field `GC`. To eliminate all potential primers that do not meet the criteria stated above (a GC content of 45% to 55%), you can make a logical indexing vector that indicates which primers have GC content outside the acceptable range. Extract the `GC` field from the structure and convert it to a numeric vector.

```
fwdgc = [fwdprimerprops.GC]';
revgc = [revprimerprops.GC]';
```

```
bad_fwdprimers_gc = fwdgc < 45 | fwdgc > 55;
bad_revprimers_gc = revgc < 45 | revgc > 55;
```

Filtering Primers Based on Their Melting Temperature

The melting temperature is significant when you are designing PCR protocols. Create another logical indexing vector to keep track of primers with bad melting temperatures. The melting temperatures from `oligoprop` are estimated in a variety of ways (basic, salt-adjusted, nearest-neighbor). The following example uses the nearest-neighbor estimates for melting temperatures with parameters established by SantaLucia, Jr. [1]. These are stored in the fifth element of the field `Tm` returned by `oligoprop`. The other elements of this field represent other methods to estimate the melting temperature. You can also use the `mean` function to compute an average over all the estimates.

```
fwdtm = cell2mat({fwdprimerprops.Tm});
revtm = cell2mat({revprimerprops.Tm});
bad_fwdprimers_tm = fwdtm(:,5) < 50 | fwdtm(:,5) > 60;
bad_revprimers_tm = revtm(:,5) < 50 | revtm(:,5) > 60;
```

Finding Primers With Self-Dimerization and Hairpin Formation

Self-dimerization and hairpin formation can prevent the primer from binding to the target sequence. As above, you can create logical indexing vectors to indicate whether the potential primers do or do not form self-dimers or hairpins.

```
bad_fwdprimers_dimers = ~cellfun('isempty',{fwdprimerprops.Dimers});
bad_fwdprimers_hairpin = ~cellfun('isempty',{fwdprimerprops.Hairpins});
bad_revprimers_dimers = ~cellfun('isempty',{revprimerprops.Dimers});
bad_revprimers_hairpin = ~cellfun('isempty',{revprimerprops.Hairpins});
```

Finding Primers Without a GC Clamp

A strong base pairing at the 3' end of the primer is helpful. Find all the primers that do not end in a G or C. Remember that all the sequences in the lists are 5'→3'.

```
bad_fwdprimers_clamp = lower(fwdprimerlist(:,end)) == 'a' | lower(fwdprimerlist(:,end)) == 't';
bad_revprimers_clamp = lower(revprimerlist(:,end)) == 'a' | lower(revprimerlist(:,end)) == 't';
```

Finding Primers With Nucleotide Repeats

Primers that have stretches of repeated nucleotides can give poor PCR results. These are sequences with low complexity. To eliminate primers with stretches of four or more repeated bases, use the function `regexpi`.

```
fwdrepeats = regexpi(cellstr(fwdprimerlist), 'a{4,}|c{4,}|g{4,}|t{4,}', 'ONCE');
revrepeats = regexpi(cellstr(revprimerlist), 'a{4,}|c{4,}|g{4,}|t{4,}', 'ONCE');
bad_fwdprimers_repeats = ~cellfun('isempty',fwdrepeats);
bad_revprimers_repeats = ~cellfun('isempty',revrepeats);
```

Find the Primers That Satisfy All the Criteria

The rows of the original list of subsequences correspond to the base number where each subsequence starts. You can use the logical indexing vectors collected so far and create a new list of primers that satisfy all the criteria discussed above. The figure shows how the forward primers have been filtered, where values equal to 1 indicates bad primers and values equal to 0 indicates good primers.

```
bad_fwdprimers = [bad_fwdprimers_gc, bad_fwdprimers_tm,...
                 bad_fwdprimers_dimers, bad_fwdprimers_hairpin,...
                 bad_fwdprimers_clamp, bad_fwdprimers_repeats];
bad_revprimers = [bad_revprimers_gc, bad_revprimers_tm,...
                 bad_revprimers_dimers, bad_revprimers_hairpin,...
                 bad_revprimers_clamp, bad_revprimers_repeats];

good_fwdpos = find(all(~bad_fwdprimers,2));
good_fwdprimers = fwdprimerlist(good_fwdpos,:);
good_fwdprop = fwdprimerprops(good_fwdpos);
N_good_fwdprimers = numel(good_fwdprop)

good_revpos = find(all(~bad_revprimers,2));
good_revprimers = revprimerlist(good_revpos,:);
good_revprop = revprimerprops(good_revpos);
N_good_revprimers = numel(good_revprop)

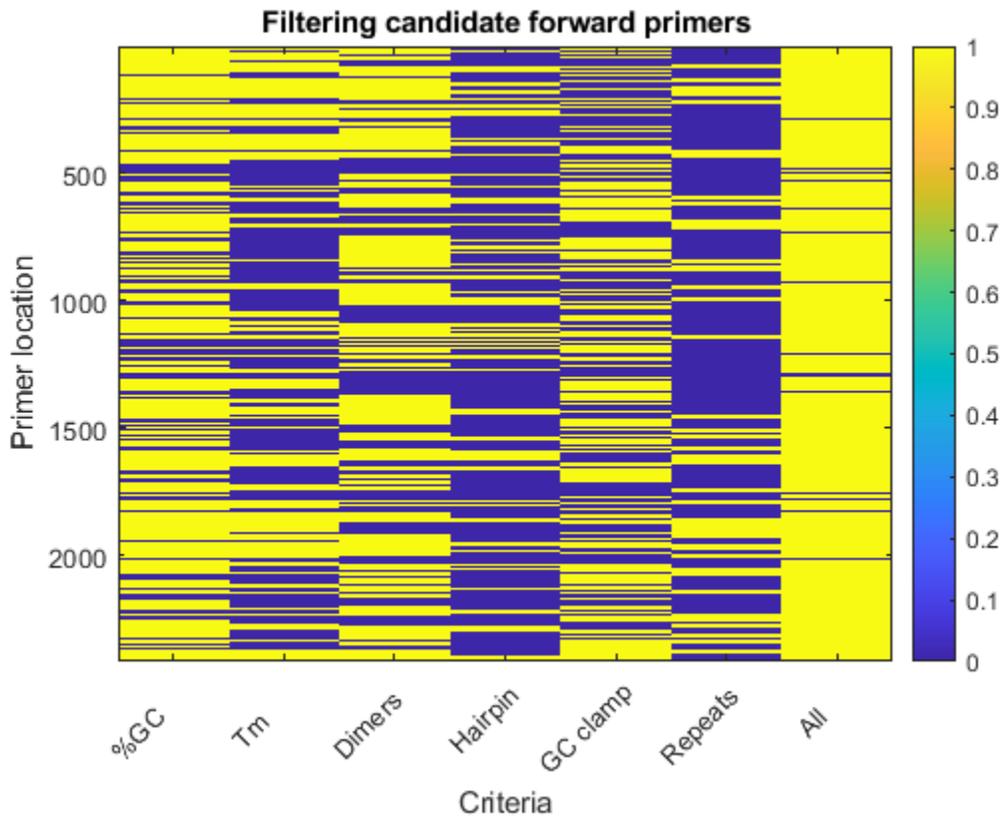
figure
imagesc([bad_fwdprimers any(bad_fwdprimers,2)]);
title('Filtering candidate forward primers');
ylabel('Primer location');
xlabel('Criteria');
ax = gca;
ax.XTickLabel = char({'%GC', 'Tm', 'Dimers', 'Hairpin', 'GC clamp', 'Repeats', 'All'});
ax.XTickLabelRotation = 45;
colorbar

N_good_fwdprimers =

    140

N_good_revprimers =

    147
```



Checking For Cross Dimerization

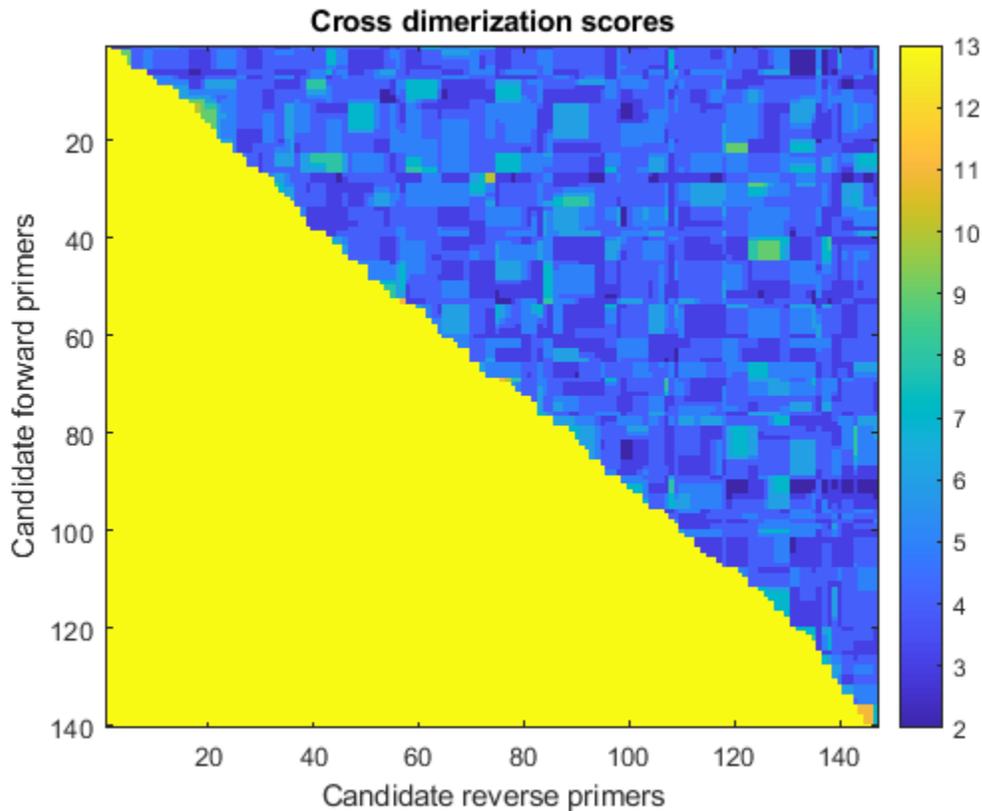
Cross dimerization can occur between the forward and reverse primer if they have a significant amount of complementarity. The primers will not function properly if they dimerize with each other. To check for dimerization, align every forward primer against every reverse primer, using the `swalign` function, and keep the low-scoring pairs of primers. This information can be stored in a matrix with rows representing forward primers and columns representing reverse primers. This exhaustive calculation can be quite time-consuming. However, there is no point in performing this calculation on primer pairs where the reverse primer is upstream of the forward primer. Therefore, these primer pairs can be ignored. The image in the figure shows the pairwise scores before being thresholded, low scores (dark blue) represent primer pairs that do not dimerize.

```
scr_mat = [-1,-1,-1,1;-1,-1,1,-1;-1,1,-1,-1;1,-1,-1,-1];
scr = zeros(N_good_fwdprimers,N_good_revprimers);
for i = 1:N_good_fwdprimers
    for j = 1:N_good_revprimers
        if good_fwdpos(i) < good_revpos(j)
            scr(i,j) = swalign(good_fwdprimers(i,:), good_revprimers(j,:), ...
                'SCORINGMATRIX',scr_mat,'GAOPEN',5,'ALPHA','NT');
        else
            scr(i,j) = 13; % give a high score to ignore forward primers
                % that are green after reverse primers
        end
    end
end
end
figure
```

```

imagesc(scr)
title('Cross dimerization scores')
xlabel('Candidate reverse primers')
ylabel('Candidate forward primers')
colorbar

```



Low scoring primer pairs are identified as logical one in an indicator matrix.

```
pairedprimers = scr<=3;
```

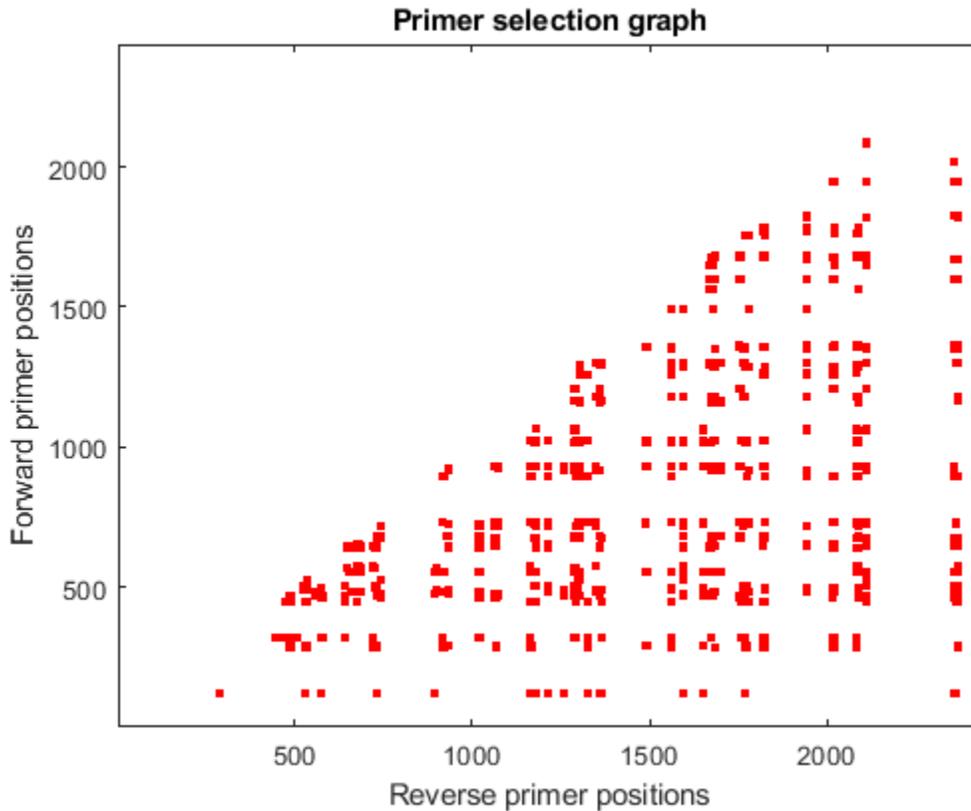
Visualizing Potential Pairs of Primers in the Sequence Domain

An alternative way to present this information is to look at all potential combinations of primers in the sequence domain. Each dot in the plot represents a possible combination between the forward and reverse primers after filtering out all those cases with potential cross dimerization.

```

[f,r] = find(pairedprimers);
figure
plot(good_revpos(r),good_fwdpos(f),'r.','markersize',10)
axis([1 N 1 N])
title('Primer selection graph')
xlabel('Reverse primer positions')
ylabel('Forward primer positions')

```

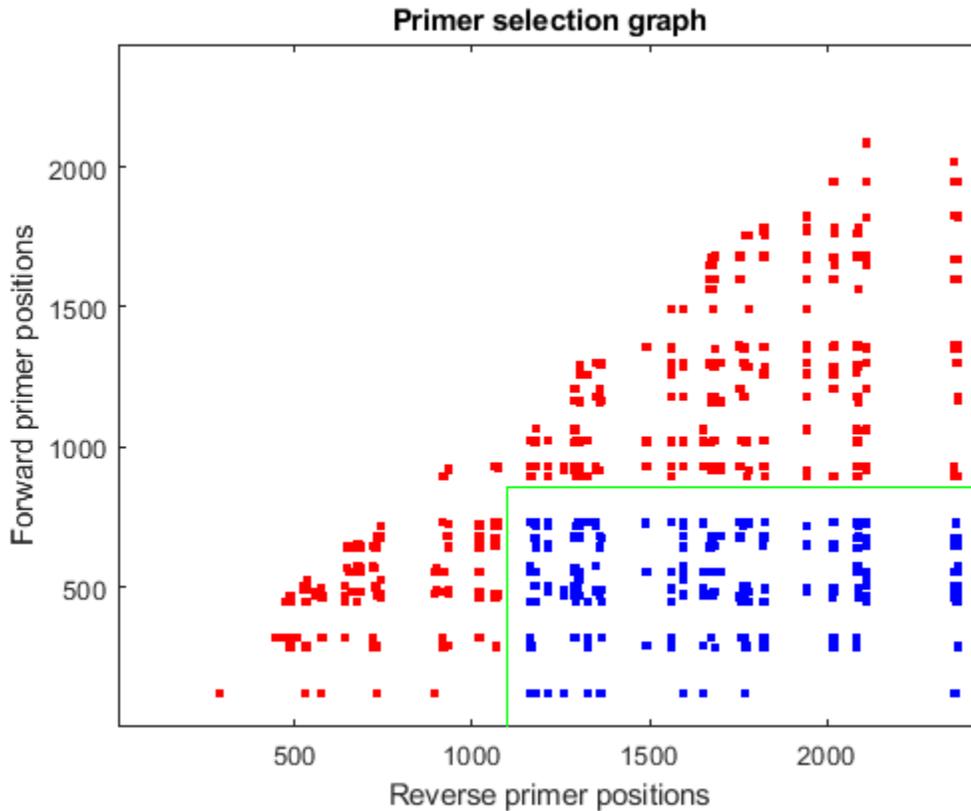


Selecting a Primer Pair to Amplify a Specific Region

You can use the information calculated so far to find the best primer pairs that allow amplification of the 220bp region from position 880 to 1100. First, you find all pairs that can cover the required region, taking into account the length of the primer. Then, you calculate the Euclidean distance of the actual positions to the desired ones, and re-order the list starting with the closest distance.

```
pairs = find(good_fwdpos(f)<(880-M) & good_revpos(r)>1100);
dist = (good_fwdpos(f(pairs))-(880-M)).^2 + (good_revpos(r(pairs))-(1100)).^2;
[dist,h] = sort(dist);
pairs = pairs(h);
```

```
hold on
plot(good_revpos(r(pairs)),good_fwdpos(f(pairs)),'b.','markersize',10)
plot([1100 1100],[1 880-M],'g')
plot([1100 N],[880-M 880-M],'g')
```



Retrieve Primer Pairs

Use the `sprintf` function to generate a report with the ten best pairs and associated information. These primer pairs can then be verified experimentally. These primers can also be 'BLASTed' using the `blastncbi` function to check specificity.

```
Primers = sprintf('Fwd/Rev Primers      Start End   %%GC   mT   Length\n\n');
for i = 1:10
    fwd = f(pairs(i));
    rev = r(pairs(i));
    Primers = sprintf('%s%-21s%-6d%-6d%-4.4g%-4.4g\n%-21s%-6d%-6d%-4.4g%-7.4g%-6d\n\n', ...
        Primers, good_fwdprimers(fwd,:),good_fwdpos(fwd),good_fwdpos(fwd)+M-1,good_fwdprop(fwd).GC,good_revprimers(rev,:),good_revpos(rev)+M-1,good_revpos(rev),good_revprop(rev).GC,good_revpos(rev) - good_fwdpos(fwd) );
end
disp(Primers)
```

Fwd/Rev Primers	Start	End	%GC	mT	Length
tacatctcgccattacctgc	732	751	50	55.61	
tcaacctcatctcctccaag	1181	1162	50	54.8	430
atacatctcgccattacctg	731	750	45	52.87	
tcaacctcatctcctccaag	1181	1162	50	54.8	431
tacatctcgccattacctgc	732	751	50	55.61	
aatcaacctcatctcctcc	1184	1165	45	52.9	433

```

tacatctcgccattacctgc 732 751 50 55.61
gaaatcaacctcatctcctc 1185 1166 45 51.08 434

atacatctcgccattacctg 731 750 45 52.87
aaatcaacctcatctcctcc 1184 1165 45 52.9 434

atacatctcgccattacctg 731 750 45 52.87
gaaatcaacctcatctcctc 1185 1166 45 51.08 435

ggatacatctcgccattacc 729 748 50 53.45
tcaacctcatctcctccaag 1181 1162 50 54.8 433

tacatctcgccattacctgc 732 751 50 55.61
gtgaaatcaacctcatctcc 1187 1168 45 51.63 436

tacatctcgccattacctgc 732 751 50 55.61
ggtgaaatcaacctcatctc 1188 1169 45 51.63 437

atacatctcgccattacctg 731 750 45 52.87
gtgaaatcaacctcatctcc 1187 1168 45 51.63 437

```

Find Restriction Enzymes That Cut Inside the Primer

Use the `rebasecuts` function to list all the restriction enzymes from the REBASE® database [2] that will cut a primer. These restriction enzymes can be used in the design of cloning experiments. For example, you can use this on the first pair of primers from the list of possible primers that you just calculated.

```

fwdprimer = good_fwdprimers(f(pairs(1)), :)
fwdcutter = unique(rebasecuts(fwdprimer))

```

```

revprimer = good_revprimers(r(pairs(1)), :)
revcutter = unique(rebasecuts(revprimer))

```

```
fwdprimer =
```

```
    'tacatctcgccattacctgc'
```

```
fwdcutter =
```

```
14×1 cell array
```

```

{'AbaSI' }
{'Acc36I'}
{'BfuAI' }
{'BmeDI' }
{'BspMI' }
{'BveI'  }
{'FspEI' }
{'LpnPI' }
{'MspJI' }
{'RlaI'  }
{'SetI'  }
{'SgeI'  }

```

```
{'SgrTI' }  
{'YkrI' }
```

```
revprimer =
```

```
'tcaacctcatctcctccaag'
```

```
revcutter =
```

```
12x1 cell array
```

```
{'AbaSI' }  
{'AspBHI'}  
{'BmeDI' }  
{'BsaXI' }  
{'FspEI' }  
{'MnII' }  
{'MspJI' }  
{'RlaI' }  
{'SetI' }  
{'SgeI' }  
{'SgrTI' }  
{'YkrI' }
```

References

- [1] SantaLucia, J. Jr., "A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics", PNAS, 95(4):1460-5, 1998.
- [2] Roberts, R.J., et al., "REBASE--restriction enzymes and DNA methyltransferases", Nucleic Acids Research, 33:D230-2, 2005.

Identifying Over-Represented Regulatory Motifs

This example illustrates a simple approach to searching for potential regulatory motifs in a set of co-expressed genomic sequences by identifying significantly over-represented ungapped words of fixed length. The discussion is based on the case study presented in Chapter 10 of "Introduction to Computational Genomics. A Case Studies Approach" [1].

Introduction

The circadian clock is the 24 hour cycle of the physiological processes that synchronize with the external day-night cycle. Most of the work on the circadian oscillator in plants has been carried out using the model plant *Arabidopsis thaliana*. In this organism, the regulation of a series of genes that need to be turned on or off at specific time of the day and night, is accomplished by small regulatory sequences found upstream the genes in question. One such regulatory motif, AAAATATCT, also known as the Evening Element (EE), has been identified in the promoter regions of circadian clock-regulated genes that show peak expression in the evening [2].

Loading Upstream Regions of Clock-Regulated Genes

We consider three sets of clock-regulated genes, clustered according to the time of the day when they are maximally expressed: set 1 corresponds to 1 KB-long upstream regions of genes whose expression peak in the morning (8am-4pm); set 2 corresponds to 1 KB-long upstream regions of genes whose expression peak in the evening (4pm-12pm); set 3 corresponds to 1 KB-long upstream regions of genes whose expression peak in the night (12pm-8am). Because we are interested in a regulatory motif in evening genes, set 2 represents our target set, while set 1 and set 3 are used as background. In each set, the sequences and their respective reverse complements are concatenated to each other, with individual sequences separated by a gap symbol (-).

```
load evemotifdemodata.mat;

% === concatenate both strands
s1 = [[set1.Sequence] seqrcomplement([set1.Sequence])];
s2 = [[set2.Sequence] seqrcomplement([set2.Sequence])];
s3 = [[set3.Sequence] seqrcomplement([set3.Sequence])];

% === compute length and number of sequences in each set
L1 = length(set1(1).Sequence);
L2 = length(set2(1).Sequence);
L3 = length(set3(1).Sequence);

N1 = numel(set1) * 2;
N2 = numel(set2) * 2;
N3 = numel(set3) * 2;

% === add separator between sequences
seq1 = seqinsertgaps(s1, 1:L1:(L1*N1)+N1, 1);
seq2 = seqinsertgaps(s2, 1:L2:(L2*N2)+N2, 1);
seq3 = seqinsertgaps(s3, 1:L3:(L3*N3)+N3, 1);
```

Identifying Over-Represented Words

To determine which candidate motif is over-represented in a given target set with respect to the background set, we identify all possible W-mers (words of length W) in both sets and compute their frequency. A word is considered over-represented if its frequency in the target set is significantly higher than the frequency in the background set. This difference is also called "margin".

```
type findOverrepresentedWords
```

```
function [nmersSorted, freqDiffSorted] = findOverrepresentedWords(seq, seq0, W)
% FINDOVERREPRESENTEDWORDS helper for evemotifdemo

% Copyright 2007 The MathWorks, Inc.

%=== find and count words of length W
nmers0 = nmercount(seq0, W);
nmers = nmercount(seq, W);

%=== compute frequency of words
f = [nmers{: ,2}]/(length(seq) - W + 1);
f0 = [nmers0{: ,2}]/(length(seq0) - W + 1);

%=== determine words common to both set
[nmersInt, i1, i2] = intersect(nmers(:,1),nmers0(:,1));
freqDiffInt = (f(i1) - f0(i2))';

%=== determine words specific to one set only
[nmersXOr, i3, i4] = setxor(nmers(:,1),nmers0(:,1));
c0 = nmers(i3,1);
d0 = nmers0(i4,1);
nmersXOr = [c0; d0];
freqDiffXOr = [f(i3) - f0(i4)]';

%=== define all words and their difference in frequency (margin)
nmersAll = [nmersInt; nmersXOr];
freqDiff = [freqDiffInt; freqDiffXOr];

%=== sort according to descending difference in frequency
[freqDiffSorted, freqDiffSortedIndex] = sort(freqDiff, 'descend');
nmersSorted = nmersAll(freqDiffSortedIndex);
```

The Evening Element Motif

If we consider all words of length $W = 9$ that appear more frequently in the target set (upstream region of genes highly expressed in the evening) with respect to the background set (upstream region of genes highly expressed in the morning and night), we notice that the most over-represented word is 'AAAATATCT', also known as the Evening Element (EE) motif.

```
W = 9;
```

```
[words, freqDiff] = findOverrepresentedWords(seq2, [seq1 seq3],W);
words(1:10)
freqDiff(1:10)
```

```
ans =
```

```
10×1 cell array
```

```
 {'AAAATATCT'}
 {'AGATATTTT'}
 {'CTCTCTCTC'}
 {'GAGAGAGAG'}
```

```
{ 'AGAGAGAGA' }
{ 'TCTCTCTCT' }
{ 'AAATATCTT' }
{ 'AAGATATTT' }
{ 'AAAAATATC' }
{ 'GATATTTTT' }
```

```
ans =

1.0e-03 *

0.1439
0.1439
0.1140
0.1140
0.1074
0.1074
0.0713
0.0713
0.0695
0.0695
```

Filtering out Repeats

Besides the EE motif, other words of length $W = 9$ appear to be over-represented in the target set. In particular, we notice the presence of repeats, i.e., words consisting of a single nucleotide or dimer repeated for the entire word length, such as 'CTCTCTCTC'. This phenomenon is quite common in genomic sequences and generally is associated with non-functional components. Because in this context the repeats are unlikely to be biologically significant, we filter them out.

```
% === determine repeats
wordsN = numel(words);
r = zeros(wordsN,1);

for i = 1:wordsN
    if (all(words{i}(1:2:end) == words{i}(1)) && ... % odd positions are the same
        all(words{i}(2:2:end) == words{i}(2))) % even positions are the same
        r(i) = 1;
    end
end
r = logical(r);

% === filter out repeats
words = words(~r);
freqDiff = freqDiff(~r);

% === consider the top 10 motifs
motif = words(1:10)
margin = freqDiff(1:10)

EEMotif = motif{1}
EEMargin = margin(1)

motif =
```

```
10x1 cell array

    {'AAAATATCT'}
    {'AGATATTTT'}
    {'AAATATCTT'}
    {'AAGATATTT'}
    {'AAAAATATC'}
    {'GATATTTTT'}
    {'AAATAAAAT'}
    {'ATTTTATTT'}
    {'TAAATAAAA'}
    {'TTTTATTTA'}

margin =

    1.0e-03 *

    0.1439
    0.1439
    0.0713
    0.0713
    0.0695
    0.0695
    0.0656
    0.0656
    0.0600
    0.0600

EEMotif =

    'AAAATATCT'

EEMargin =

    1.4393e-04
```

After removing the repeats, we observe that the EE motif ('AAAATATCT') and its reverse complement ('AGATATTTT') are at the top of the list. The other over-represented words are either simple variants of the EE motif, such as 'AAATATCTT', 'AAAAATATC', 'AAATATCTC', or their reverse complements, such as 'AAGATATTT', 'GATATTTTT', 'GAGATATTT'.

Assessing the Statistical Significance of Margins

Various techniques can be used to assess the statistical significance of the margin computed for the EE motif. For example, we can repeat the analysis using some control sequences and evaluate the resulting margins with respect to the EE margin. Genomic regions of *Arabidopsis thaliana* that are further away from the transcription start site are good candidates for this purpose. Alternatively, we could randomly split and shuffle the sequences under consideration and use these as controls. Another simple solution is to generate random sequences according to the nucleotide composition of the three original sets of sequences, as shown below.

```

% === find base composition of each set
bases1 = basecount(s1);
bases2 = basecount(s2);
bases3 = basecount(s3);

% === generate random sequences according to base composition
rs1 = randseq(length(s1), 'fromstructure', bases1);
rs2 = randseq(length(s2), 'fromstructure', bases2);
rs3 = randseq(length(s3), 'fromstructure', bases3);

% === add separator between sequences
rseq1 = seqinsertgaps(rs1, 1:L1:(L1*N1)+N1, 1);
rseq2 = seqinsertgaps(rs2, 1:L2:(L2*N2)+N2, 1);
rseq3 = seqinsertgaps(rs3, 1:L3:(L3*N3)+N3, 1);

% === compute margins for control set
[words, freqDiff] = findOverrepresentedWords(rseq2, [rseq1 rseq3], W);

```

The variable `ctrlMargin` holds the estimated margins of the top motifs for each of the 100 control sequences generated as described above. The distribution of these margins can be approximated by the extreme value distribution. We use the function `gevfit` from the Statistics and Machine Learning Toolbox™ to estimate the parameters (shape, scale, and location) of the extreme value distribution and we overlay a scaled version of its probability density function, computed using `gevpdf`, with the histogram of the margins of the control sequences.

```

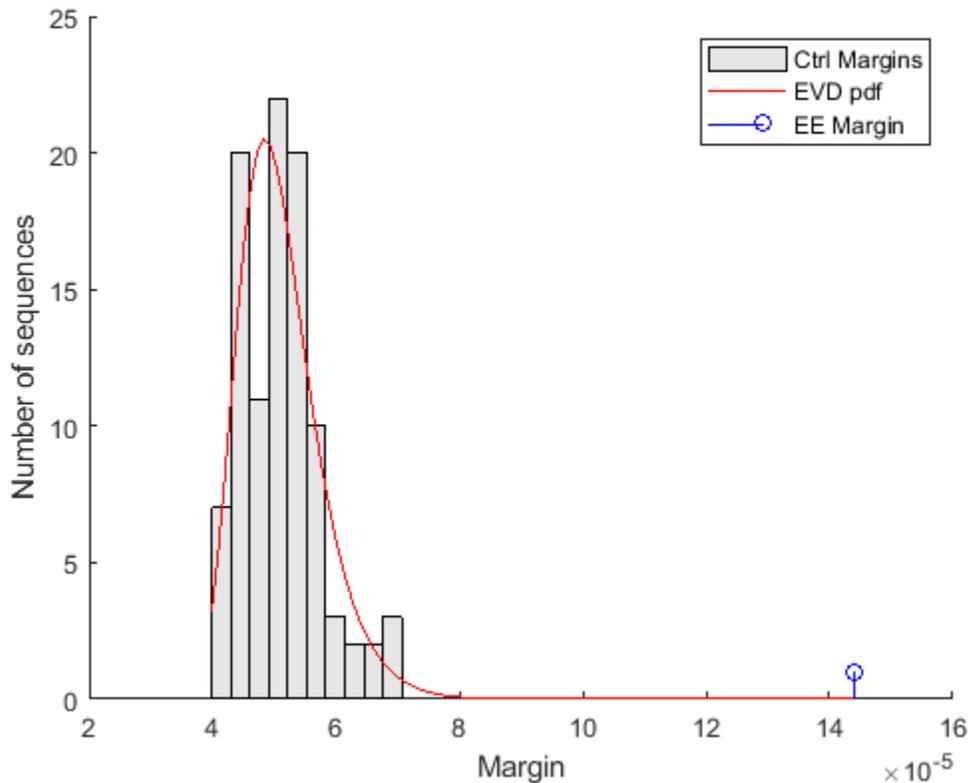
% === estimate parameters of distribution
nCtrl = length(ctrlMargin);
buckets = ceil(nCtrl/10);
parmhat = gevfit(ctrlMargin);
k = parmhat(1); % shape parameter
sigma = parmhat(2); % scale parameter
mu = parmhat(3); % location parameter

% === compute probability density function
x = linspace(min(ctrlMargin), max([ctrlMargin EEMargin]));
y = gevpdf(x, k, sigma, mu);

% === scale probability density function
[v, c] = hist(ctrlMargin, buckets);
binWidth = c(2) - c(1);
scaleFactor = nCtrl * binWidth;

% === overlay
figure()
hold on;
hist(ctrlMargin, buckets);
h = findobj(gca, 'Type', 'patch');
h.FaceColor = [.9 .9 .9];
plot(x, scaleFactor * y, 'r');
stem(EEMargin, 1, 'b');
xlabel('Margin');
ylabel('Number of sequences');
legend('Ctrl Margins', 'EVD pdf', 'EE Margin');

```



The control margins are the differences in frequency that we would expect to find when a word is over-represented by chance alone. The margin relative to the EE motif is clearly significantly larger than the control margins, and does not fit within the probability density curve of the random controls. Because the EE margin is larger than all 100 control margins, we can conclude that the over-representation of the EE motif in the target set is statistically significant and the p-value estimate is less than 0.01.

Selecting Motif Length

If we repeat the search for over-represented words of length $W = 6 \dots 11$, we observe that all the top motifs are either substrings (if $W < 9$) or superstrings (if $W > 9$) of the EE motif. Thus, how do we decide what is the correct length of this motif? We can expect that the optimal length maximizes the difference in frequency between the motif in the target set and the same motif in the background set. However, in order to compare the margin across different lengths, the margin must be normalized to account for the natural tendency of shorter words to occur more frequently. We perform this normalization by dividing each margin by the margin corresponding to the most over-represented word of identical length in a random set of sequences with a nucleotide composition similar to the target set. For convenience, the top over-represented words for length $W = 6 \dots 11$ and their margins are stored in the variables `topMotif` and `topMargin`. Similarly, the top over-represented words for length $W = 6 \dots 11$ and their margins in a random set are stored in the variables `rTopMotif` and `rTopMargin`.

```
% === top over-represented words, W = 6...11 in set 2 (evening)
topMotif
topMargin
```

```

% === top over-represented words, W = 6...11 in random set
rTopMotif
rTopMargin

% === compute score
score = topMargin ./ rTopMargin;
[bestScore, bestLength] = max(score);

% === plot
figure()
plot(6:11, score(6:11));
xlabel('Motif length');
ylabel('Normalized margin');
title('Optimal motif length');
hold on
line([bestLength bestLength], [0 bestScore], 'LineStyle', '-.')

```

```
topMotif =
```

```

11x1 cell array

    {0x0 double }
    {'AATATC' }
    {'AATATCT' }
    {'AAATATCT' }
    {'AAAATATCT' }
    {'AAAAATATCT' }
    {'AAAAAATATCT' }

```

```
topMargin =
```

```

1.0e-03 *

    NaN
    NaN
    NaN
    NaN
    NaN
    0.3007
    0.2607
    0.2074
    0.1439
    0.0648
    0.0424

```

```
rTopMotif =
```

```

11x1 cell array

    {0x0 double }

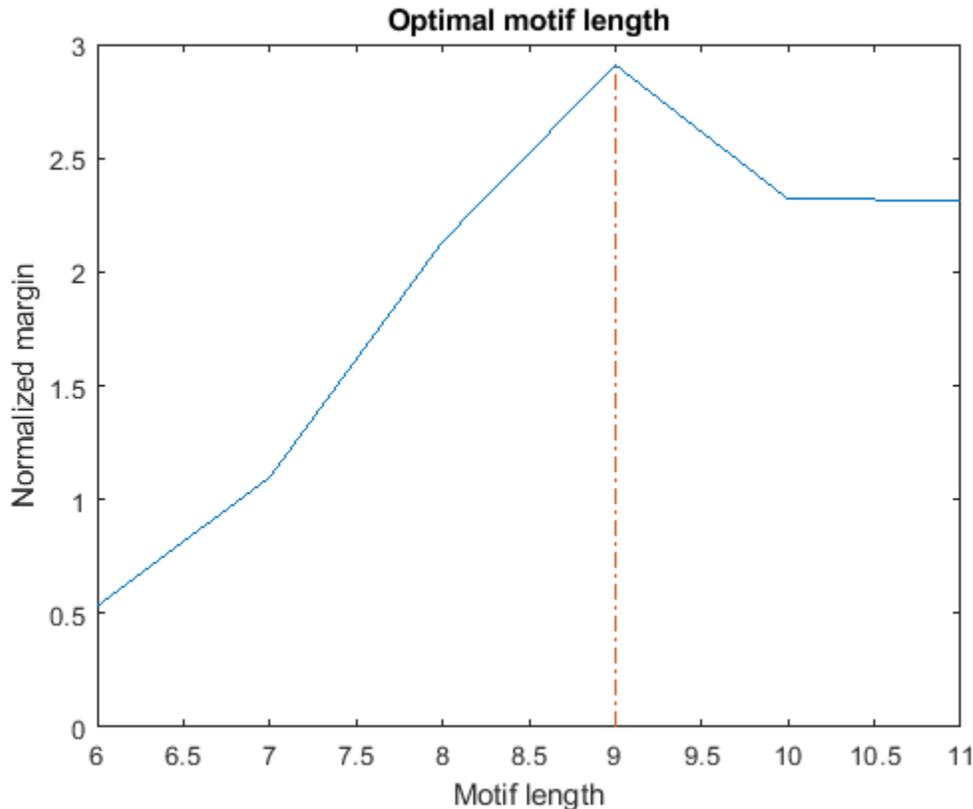
```

```
{0x0 double }  
{0x0 double }  
{0x0 double }  
{0x0 double }  
{'ATTATA' }  
{'TATAATA' }  
{'TTATTAAA' }  
{'GTTATTAAA' }  
{'ATTATATATC' }  
{'ATGTTATTATT' }
```

rTopMargin =

1.0e-03 *

```
NaN  
NaN  
NaN  
NaN  
NaN  
0.5650  
0.2374  
0.0972  
0.0495  
0.0279  
0.0183
```



By plotting the normalized margin versus the motif length, we find that length $W = 9$ is the most informative in discriminating over-represented motifs in the target sequence (evening set) against the background set (morning and night sets).

Determining the Evening Element Motif Presence Among Clock-Regulated Genes

Although the EE Motif has been identified and experimentally validated as a regulatory motif for genes whose expression peaks in the evening hours, it is not shared by all evening genes, nor is it exclusive of these genes. We count the occurrences of the EE motif in the three sequence sets and determine what proportion of genes in each set contain the motif.

```
EECount = zeros(3,1);

% === determine positions where EE motif occurs
loc1 = strfind(seq1, EEMotif);
loc2 = strfind(seq2, EEMotif);
loc3 = strfind(seq3, EEMotif);

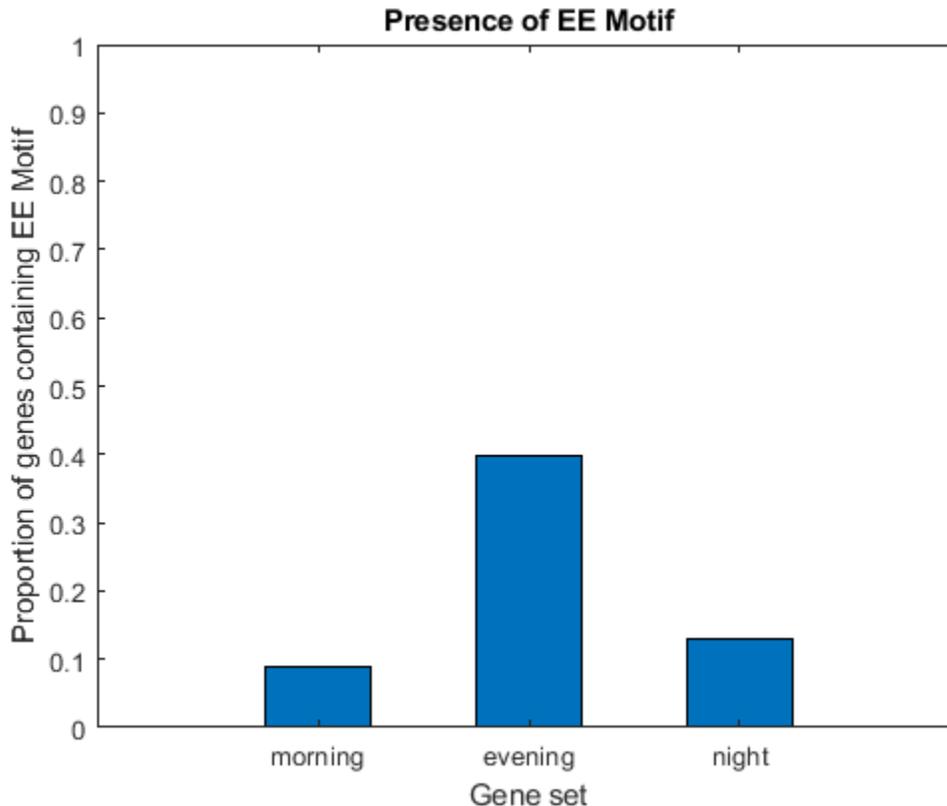
% === count occurrences
EECount(1) = length(loc1);
EECount(2) = length(loc2);
EECount(3) = length(loc3);

% === find proportions of genes with EE Motif
NumGenes = [N1; N2; N3] / 2;
EEProp = EECount ./ NumGenes;
```

```

% === plot
figure()
bar(EETProp, 0.5);
ylabel('Proportion of genes containing EE Motif');
xlabel('Gene set');
title('Presence of EE Motif');
ylim([0 1])
ax = gca;
ax.XTickLabel = {'morning', 'evening', 'night'};

```



It appears as though about 9% of genes in set 1, 40% of genes in set 2, and 13% of genes in set 3 have the EE motif. Thus, not all genes in set 2 have the motif, but it is clearly enriched in this group.

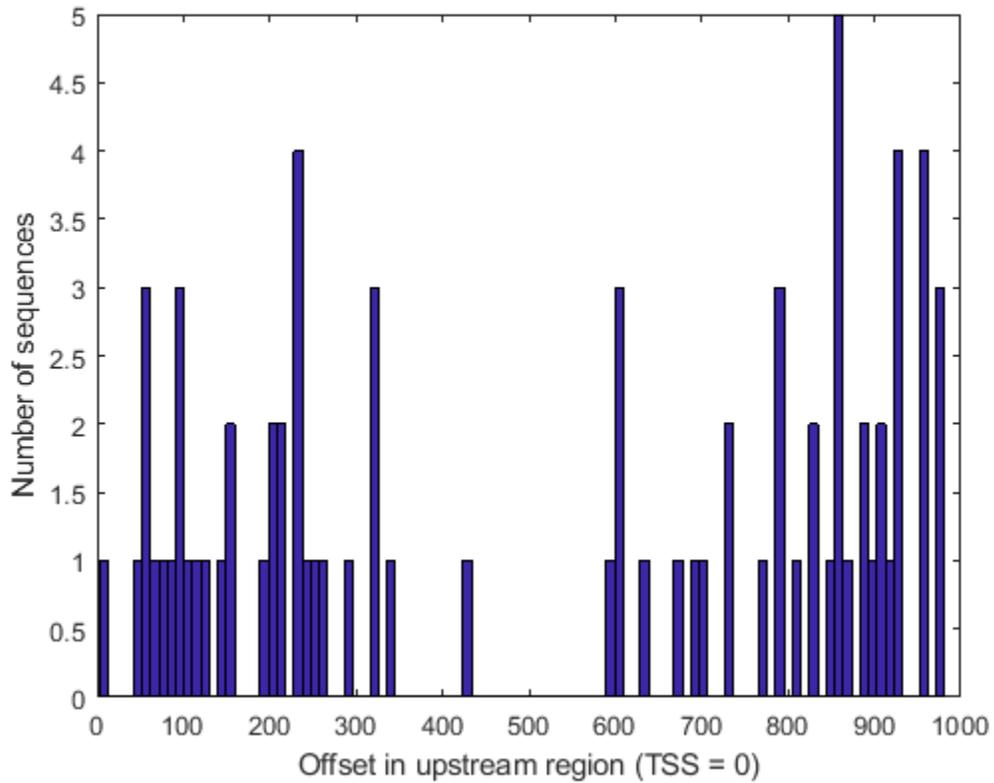
Analyzing the Evening Element Motif Location

Unlike many other functional motifs, the EE motif does not appear to accumulate at specific gene locations in the set of sequences analyzed. After determining the location of each occurrence with respect to the transcription start site (TSS), we observe a relatively uniform distribution of occurrences across the upstream region of the genes considered, with the possible exception of the middle region (between 400 and 500 bases upstream of the TSS).

```

offset = rem(loc2, 1001);
figure();
hist(offset, 100);
xlabel('Offset in upstream region (TSS = 0)');
ylabel('Number of sequences');

```



References

[1] Cristianini, N. and Hahn, M.W., "Introduction to Computational Genomics: A Case Studies Approach", Cambridge University Press, 2007.

[2] Harmer, S.L., et al., "Orchestrated Transcription of Key Pathways in Arabidopsis by the Circadian Clock", *Science*, 290(5499):2110-3, 2000.

Predicting and Visualizing the Secondary Structure of RNA Sequences

This example illustrates how to use the `rnafold` and `rnaplot` functions to predict and plot the secondary structure of an RNA sequence.

Introduction

RNA plays an important role in the cell, both as genetic information carrier (mRNA) and as functional element (tRNA, rRNA). Because the function of an RNA sequence is largely associated with its structure, predicting the RNA structure from its sequence has become increasingly important. Because base pairing and base stacking represent the majority of the free energy contribution to folding, a good estimation of secondary structure can be very helpful not only in the interpretation of the function and reactivity, but also in the analysis of the tertiary structure of the RNA molecule.

RNA Secondary Structure Prediction Using Nearest-Neighbor Thermodynamic Model

The secondary structure of an RNA sequence is determined by the interaction between its bases, including hydrogen bonding and base stacking. One of the many methods for RNA secondary structure prediction uses the nearest-neighbor model and minimizes the total free energy associated with an RNA structure. The minimum free energy is estimated by summing individual energy contributions from base pair stacking, hairpins, bulges, internal loops and multi-branch loops. The energy contributions of these elements are sequence- and length-dependent and have been experimentally determined [1]. The `rnafold` function uses the nearest-neighbor thermodynamic model to predict the minimum free-energy secondary structure of an RNA sequence. More specifically, the algorithm implemented in `rnafold` uses dynamic programming to compute the energy contributions of all possible elementary substructures and then predicts the secondary structure by considering the combination of elementary substructures whose total free energy is minimum. In this computation, the contribution of coaxially stacked helices is not accounted for, and the formation of pseudoknots (non-nested structural elements) is forbidden.

Secondary Structure of Transfer RNA Phenylalanine

tRNAs are small molecules (73-93 nucleotides) that during translation transfer specific amino acids to the growing polypeptide chain at the ribosomal site. Although at least one tRNA molecule exists for each amino acid type, both secondary and tertiary structures are well conserved among the various tRNA types, most likely because of the necessity of maintaining reliable interaction with the ribosome. We consider the following tRNA-Phe sequence from *Saccharomyces cerevisiae* and predict the minimum free-energy secondary structure using the function `rnafold`.

```
% === Predict secondary structure in bracket notation
phe_seq = 'GCGGAUUUAGCUCAGUUGGGAGAGCGCCAGACUGAAGAUUGGAGGUCCUGUGUUCGAUCCACAGAAUUCGCACCA';
phe_str = rnafold(phe_seq)

phe_str =

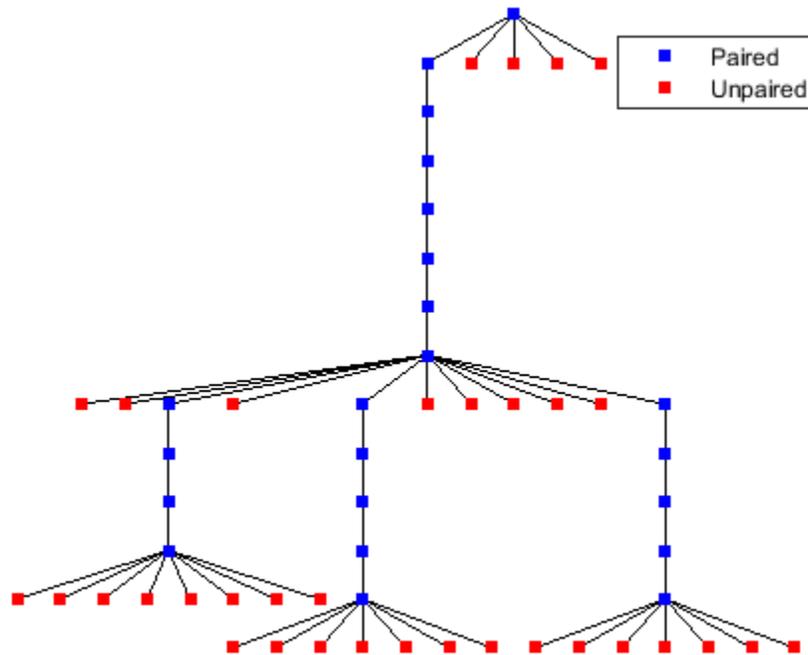
      '(((((((..((((.....))))).((((.....))))).(((.....)))))).....'

```

In the bracket notation, each dot represents an unpaired base, while a pair of equally nested, opening and closing brackets represents a base pair. Alternative representations of RNA secondary structures

can be drawn using the function `rnaplot`. For example, the structure predicted above can be displayed as a rooted tree, where leaf nodes correspond to unpaired residues and internal nodes (except the root) correspond to base pairs. You can display the position and type of each residue by clicking on the corresponding node.

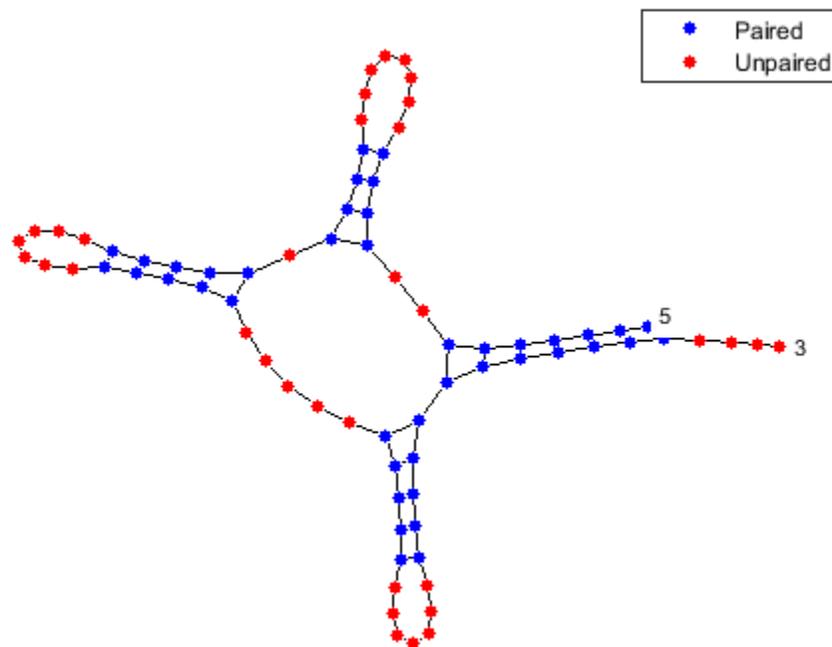
```
% === Plot RNA secondary structure as tree
rnaplot(phe_str, 'seq', phe_seq, 'format', 'tree');
```



The tRNA secondary structure is commonly represented in a diagram plot and resembles a clover leaf. It displays four base-paired stems (or "arms") and three loops. Each of the four stems has been extensively studied and characterized: acceptor stem (positions 1-7 and 66-72), D-stem (positions 10-13 and 22-25), anticodon stem (positions 27-31 and 39-43) and T-stem (positions 49-53 and 61-65). We can draw the tRNA secondary structure as a two-dimensional plot where each residue is identified by a dot and the backbone and the hydrogen bonds are represented as lines between the dots. The stems consist of consecutive stretches of base paired residues (blue dots), while the loops are formed by unpaired residues (red dots).

```
% === Plot the secondary structure using the dot diagram representation
rnaplot(phe_str, 'seq', phe_seq, 'format', 'dot');

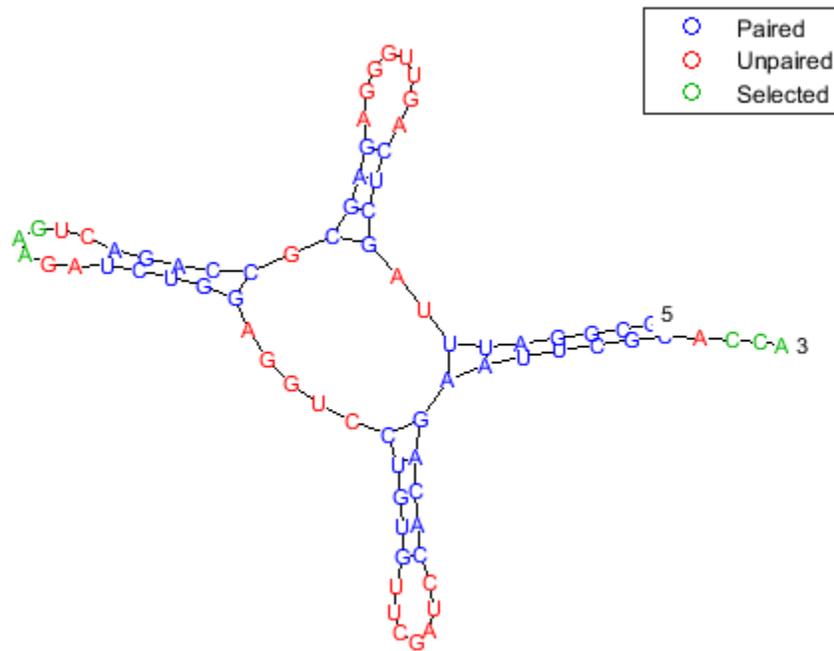
text(500, 200, 'T-stem');
text(100, 600, 'Anticodon stem');
text(550, 650, 'D-stem stem');
text(700, 400, 'Acceptor stem');
```



While all the stems are important for a proper three-dimensional folding of the molecule and successful interplay with ribosome and tRNA synthetases, the acceptor stem and the anticodon stem are particularly interesting because they include the attachment site and the anticodon triplet. The attachment site (positions 74-76) occurs at the 3' end of the RNA chains and consists of the sequence C-C-A in all amino acid acceptor stems. The anticodon triplet consists of 3 bases that pair with a complementary codon in the messenger RNA. In the case of Phe-tRNA, the anticodon sequence A-A-G (positions 34-36) pairs with the mRNA codon U-U-C, encoding the amino acid phenylalanine. We can redraw the structure and highlight these regions in the acceptor stem and anticodon stem by using the `selection` property:

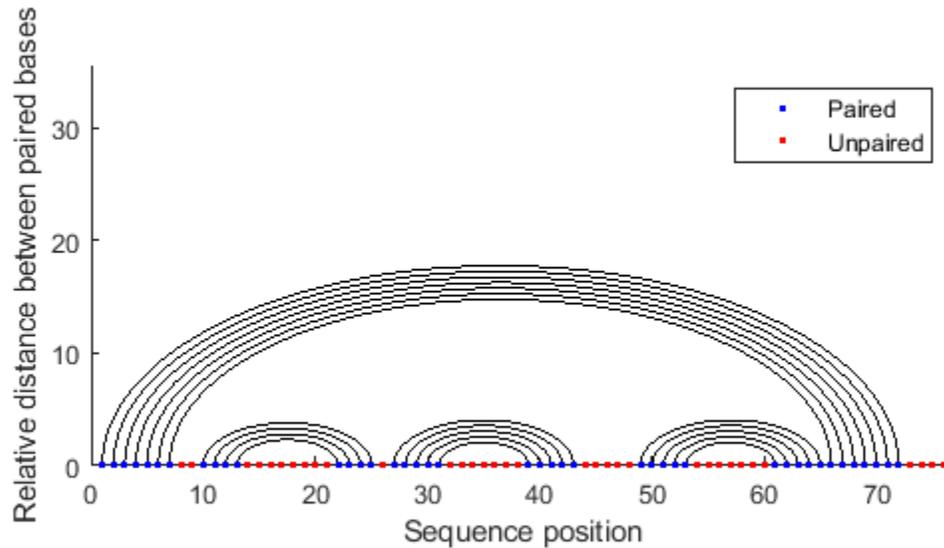
```
aag_pos = 34:36;  
cca_pos = 74:76;
```

```
rnaplot(phe_str, 'sequence', phe_seq, 'format', 'diagram', ...  
        'selection', [aag_pos, cca_pos]);
```



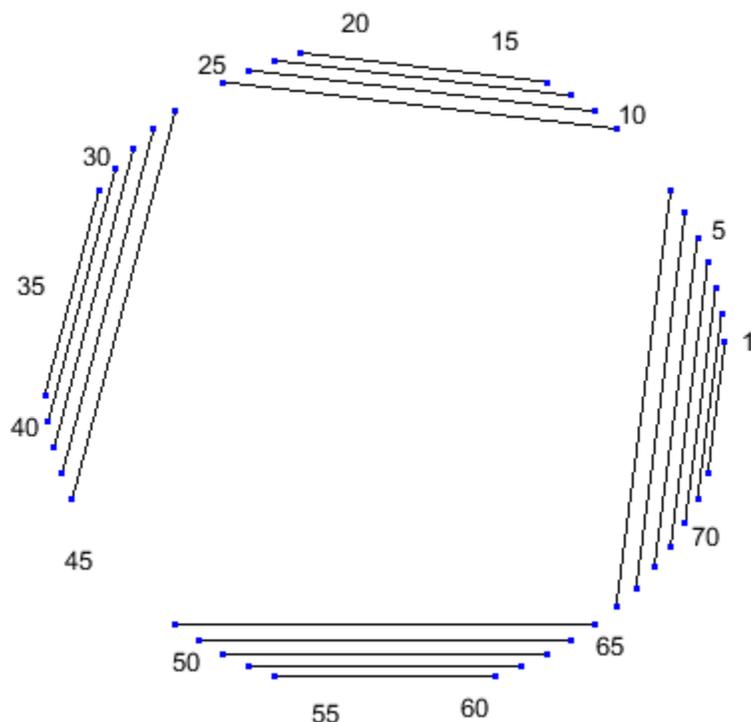
The segregation of the sequence into four separate stems is better appreciated by displaying the structure as graph plot. Each residue is represented on the abscissa and semi-elliptical lines connect bases that pair with each other. The lack of pseudoknots in the secondary structure is reflected by the absence of intersecting lines. This is expected in tRNA secondary structures and anticipated because the dynamic programming method used does not allow pseudoknots.

```
rnaplot(phe_str, 'sequence', phe_seq, 'format', 'graph');
```



Similar observations can be drawn by displaying the secondary structure as a circle, where each base is represented by a dot on the circumference of a circle of arbitrary size, and bases that pair with each other are connected by lines. The lines are visually clustered into four distinct groups, separated by stretches of unpaired residues. We can hide the unpaired residues by using `H.Unpaired`, the handle returned with the `colorby` property set to `state`.

```
[ha, H] = rnaplot(phe_str, 'sequence', phe_seq, 'format', 'circle', ...
    'colorby', 'state');
H.Unpaired.Visible = 'off';
legend off;
```



As you can see, the outputs of the `rnaplot` function include a MATLAB® structure `H` consisting of handles that can be used to change the aspect properties of various residue subsets. For example, if you set the color scheme using the `colorby` property set to `residue`, the dots are colored according to the residue type, and you can change their property using the appropriate handle.

```
[ha, H] = rnaplot(phe_str, 'sequence', phe_seq, 'format', 'circle', 'colorby', 'residue')
```

```
ha =
```

```
  Axes (Bioinfo:rnaplot:circle) with properties:
```

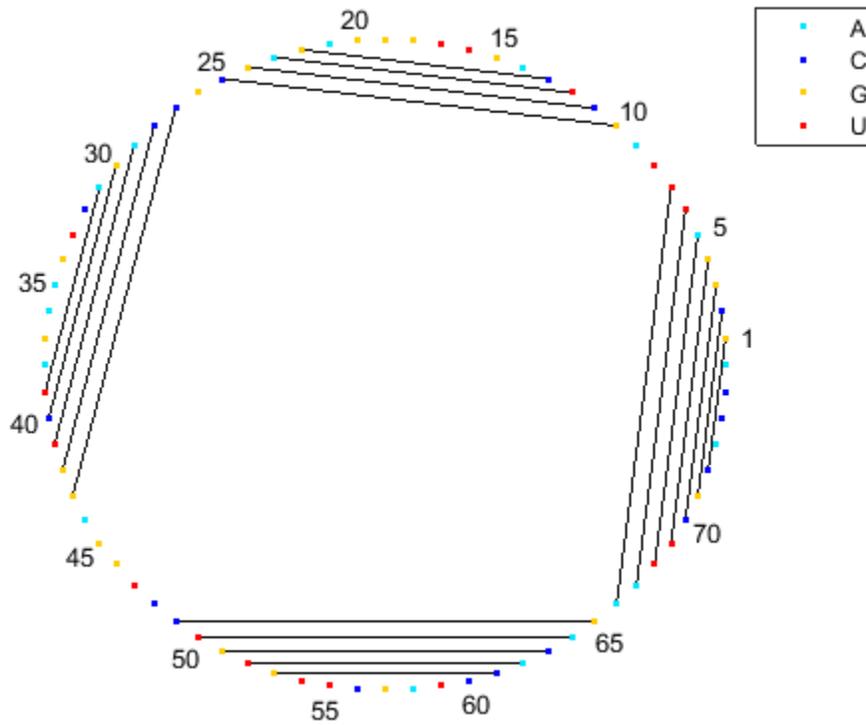
```
      XLim: [-1 1]
      YLim: [-1 1.1000]
      XScale: 'linear'
      YScale: 'linear'
  GridLineStyle: '-'
      Position: [0.1124 0.1100 0.6703 0.8150]
      Units: 'normalized'
```

```
Use GET to show all properties
```

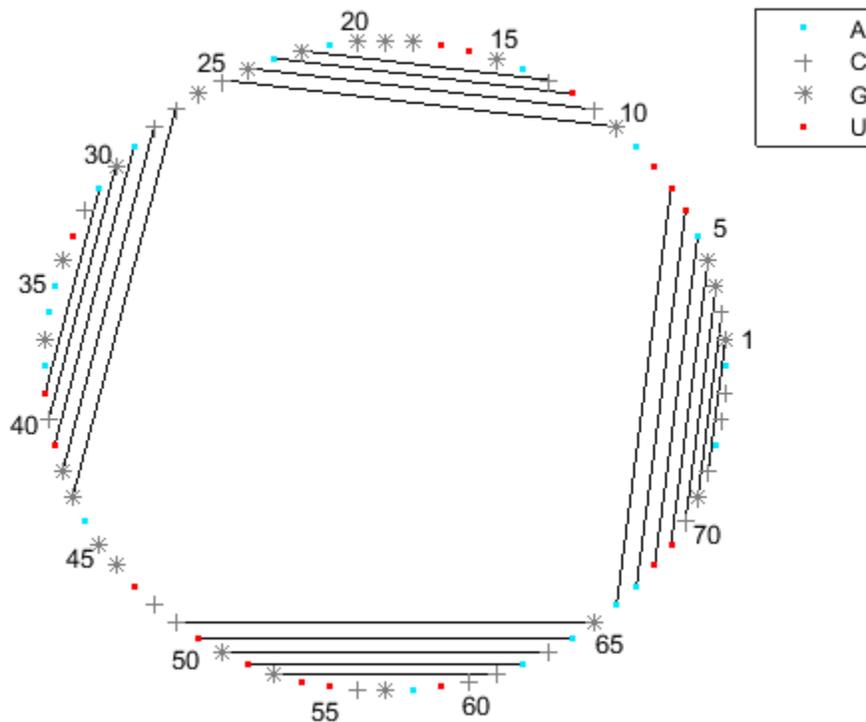
```
H =
```

```
  struct with fields:
```

```
A: [1x1 Line]
C: [1x1 Line]
G: [1x1 Line]
U: [1x1 Line]
Selected: [0x1 Line]
```



```
H.G.Color = [0.5 0.5 0.5];
H.G.Marker = '*';
H.C.Color = [0.5 0.5 0.5];
H.C.Marker = '+';
```



Conservation of Transfer RNA Phenylalanine

Despite some differences in their primary sequences, tRNAs molecules present a secondary structure pattern that is well conserved across the three phylogenetic domains. Consider the structure of the tRNA-Phe of one representative organism for each phylogenetic domain: *Saccharomyces cerevisiae* for the Eukaryotes, *Haloarcula marismortui* for the Archaea, and *Thermus thermophilus* for the Bacteria. Then predict and plot their secondary structures using the mountain plot representation.

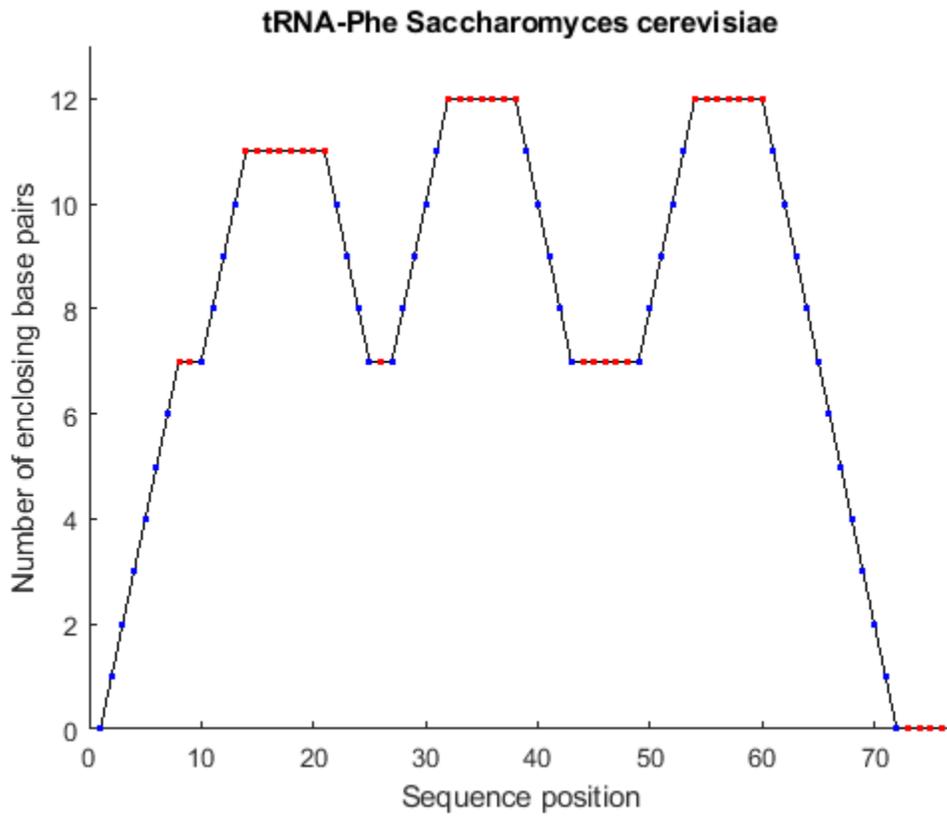
```
yeast = 'GCGGACUAGCUCAGUUGGGAGAGCGCCAGACUGAAGAUCUGGAGGUCCUGUGUJCGAUCCACAGAGUUCGCACCA';
halma = 'GCCGCCUAGCUCAGACUGGGAGAGCACUCGACUGAAGAUCGAGCUGUCCCGGUUCAAUCCGGGAGGCGGCACCA';
theth = 'GCCGAGGUAGCUCAGUUGGUAGAGCAUGCACUGAAAUCGCAGUGUCGGCGGUUCGAUUCGCCCCUCGGCACCA';
```

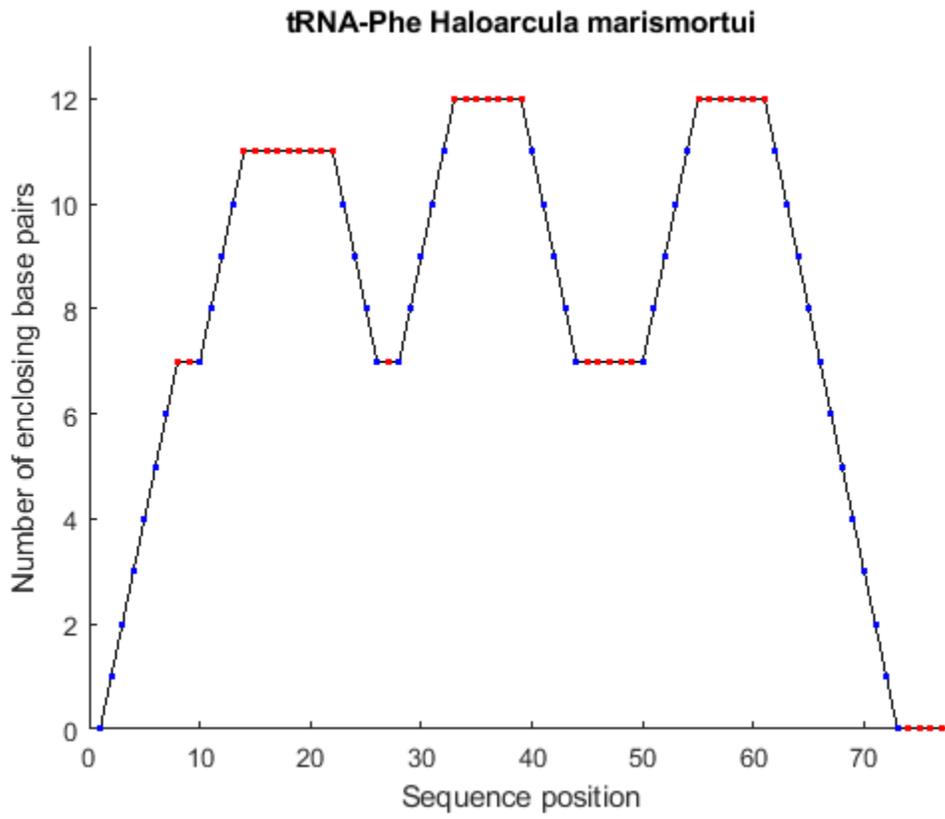
```
yeast_str = rnafold(yeast);
theth_str = rnafold(theth);
halma_str = rnafold(halma);
```

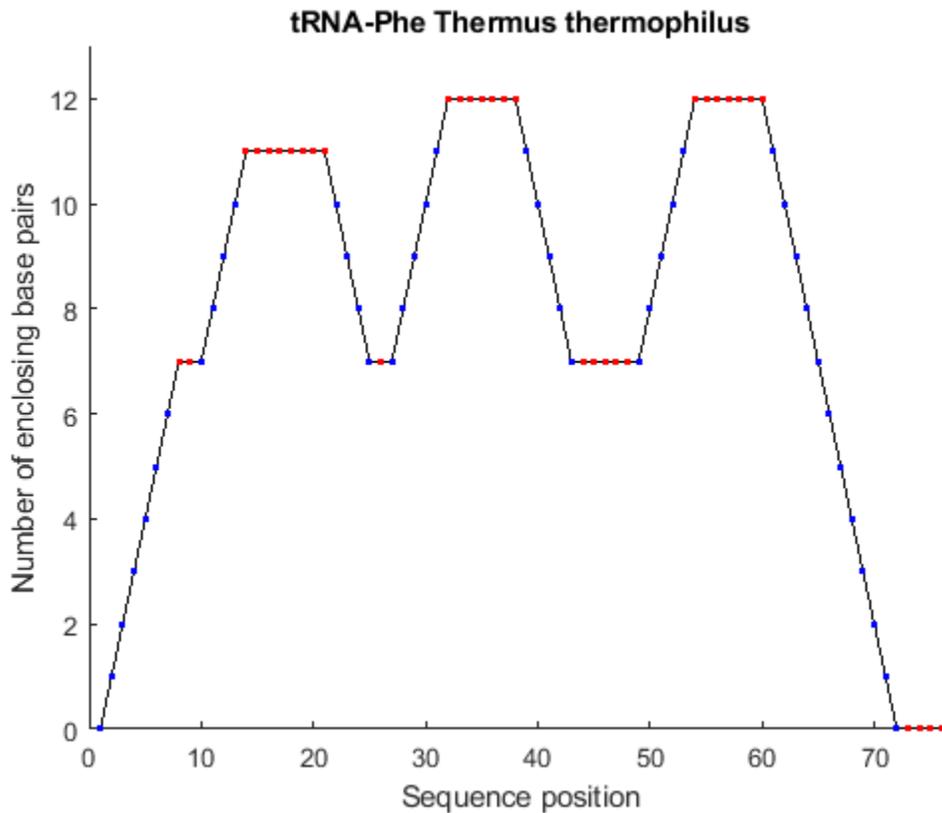
```
h1 = rnaplot(yeast_str, 'sequence', yeast, 'format', 'mountain');
title(h1, 'tRNA-Phe Saccharomyces cerevisiae');
legend hide;
```

```
h2 = rnaplot(halma_str, 'sequence', halma, 'format', 'mountain');
title(h2, 'tRNA-Phe Haloarcula marismortui');
legend hide;
```

```
h3 = rnaplot(theth_str, 'sequence', theth, 'format', 'mountain');
title(h3, 'tRNA-Phe Thermus thermophilus');
legend hide;
```





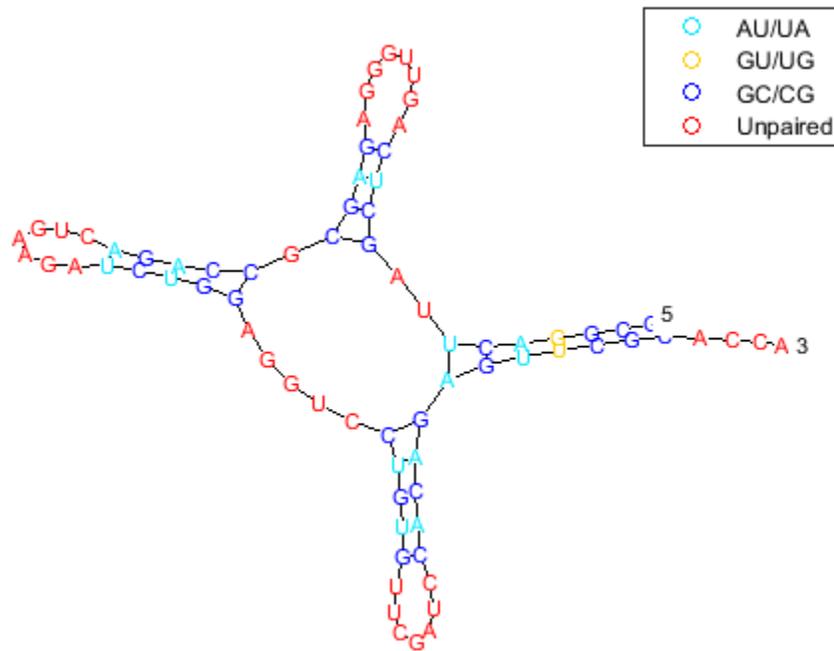


The similarity among the resulting structures is striking, the only difference being one extra residue in the D-loop of *Haloarcula marismortui*, displayed in the first flat slope in the mountain plot.

The G-U Wobble Base Pair

Besides the Watson-Crick base pairs (A-U, G-C), virtually every class of functional RNA presents G-U wobble base pairs. G-U pairs have an array of distinctive chemical, structural and conformational properties: they have high affinity for metal ions, they are almost thermodynamically as stable as Watson-Crick base pairs, and they present conformational flexibility to different environments. The wobble pair at the third position of the acceptor helix of tRNA is very highly conserved in almost all organisms. This conservation suggests that the G-U pair possesses unique features that can hardly be duplicated by other pairs. You can observe the base pair type distribution on the secondary structure diagram by coloring the base pairs according to their type.

```
rnplot(yeast_str, 'sequence', yeast, 'format', 'diagram', 'colorby', 'pair');
```



References

- [1] Matthews, D., Sabina, J., Zuker, M., and Turner, D. "Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure", *Journal of Molecular Biology*, 288(5):911-40, 1999.

Using HMMs for Profile Analysis of a Protein Family

This example shows how HMM profiles are used to characterize protein families. Profile analysis is a key tool in bioinformatics. The common pairwise comparison methods are usually not sensitive and specific enough for analyzing distantly related sequences. In contrast, Hidden Markov Model (HMM) profiles provide a better alternative to relate a query sequence to a statistical description of a family of sequences. HMM profiles use a position-specific scoring system to capture information about the degree of conservation at various positions in the multiple alignment of these sequences. HMM profile analysis can be used for multiple sequence alignment, for database searching, to analyze sequence composition and pattern segmentation, and to predict protein structure and locate genes by predicting open reading frames.

Accessing PFAM Databases

Start this example with an already built HMM of a protein family. Retrieve the model for the well-known 7-fold transmembrane receptor from the Sanger Institute database. The PFAM key number is PF00002. Also retrieve the pre-aligned sequences used to train this model. More information about the PFAM database can be found at <http://pfam.xfam.org/>.

```
hmm_7tm = gethmmprof(2);
seed_seqs = gethmmalignment(2, 'type', 'seed');
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load('gpcrfam.mat', 'hmm_7tm', 'seed_seqs')
```

Models and alignments can also be stored and parsed in later directly from the files using the `pfamhmmread`, `fastaread` and `multialignread` functions.

Display the names and contents of the first three loaded sequences using the `seqdisp` command.

```
seqdisp(seed_seqs([1 2 3]), 'row', 70)
```

```
ans =
```

```
23x81 char array
```

```
'>VIPR2_HUMAN/123-371
' 1 YILVKAIYTL GYSVS.LMSL ATGSIIILCLF .RKLHCTR.N YIHLNLFSLF ILRAISVLVK .DDVLYSSS.'
' 71 GTLHCPD... ..QPSSW. ..V.GCKLSL VFLQYCI MAN FFLLVEGLY'
'141 LHTLLVA... ..MLPP.RR CFLAYLLIGW GLPTVCIGAW TAAR..... ..L YLED.....'
'211 .....TGC. WDTN.DHSVP W....WVIRI PILISIIVNF VLFISIIRIL LQKLT..... .SPDVGGNDQ'
'281 SQY..... ..KRLAKS TLLLIPLFGV HYMV..FAVF PISI...S.S'
'351 KYQILFELCL GSF....QGL VV
'
'>VIPR_CARAU/100-348
' 1 FRSVKIGYTI GHSVS.LISL TTAIVILCMS .RKLHCTR.N YIHMHLFVSF ILKAIAVFVK .DAVLYDVIQ'
' 71 ESDNCS... ..TASV. ....GCKAVI VFFQYCI MAS FFLLVEGLY'
'141 LHALLAVS... ..FFSE.RK YFWWYILIGW GGPTIFIMAW SFAK..... ..A YFND.....'
'211 .....VGC. WDIENSDF W....WIIKT PILASILMNF ILFICIIRIL RQKIN..... .CPDIGRNE'
'281 NQY..... ..SRLAKS TLLLIPLFGI NFII..FAFI PENI...K.T'
'351 ELRLVFDLIL GSF....QGF VV
```

```
'
'>VIPR1_RAT/140-386
' 1  YNTVKTGYTI GYSLS.LASL LVAMAILSLF .RKLHCTR.N YIHMHLFMSF ILRATAVFIK .DMALFNSG.'
' 71 EIDHCS.... .EASV. ....GCKAAV VFFQYCVMAN FFVLLVEGLY'
'141 LYTLAVS.. ...FFSE.RK YFWGYILIGW GVPSVFITIW TVVR.....I YFED.....'
'211 .....FGC. WDTI.INSSL W....WIIKA PILLSILVNF VLFICIIRIL VQKLR..... .PPDIGKNS'
'281 SPY..... .SRLAKS TLLLIPLFGI HYVM..FAFF PDNF...K.A'
'351 QVKMVFELVV GSF....QGF VV
```

More information regarding how to store the profile HMM information in a MATLAB® structure is found in the help for `hmmprofstruct`.

Profile HMM Alignment

To test the profile HMM alignment tool you can re-align the sequences from the multiple alignment to the HMM model. First erase the periods in sequences used to format the downloaded aligned sequences. Doing this removes the alignment information from the sequences.

```
seqs = strrep({seed_seqs.Sequence}, '.', '');
names = {seed_seqs.Header};
```

Now align all the proteins to the HMM profile.

```
fprintf('Aligning sequences ')
scores = zeros(numel(seqs),1);
aligned_seqs = cell(numel(seqs),1);
for sn=1:numel(seqs)
    fprintf('.')
    [scores(sn),aligned_seqs{sn}]=hmmprofalign(hmm_7tm,seqs{sn});
end
fprintf('\n')
```

```
Aligning sequences .....
```

Next, send the results to the system web browser to better explore the new multiple alignment. Columns marked with * at the bottom indicate when the model was in a "match" or "delete" state.

```
hmmprofmerge(aligned_seqs,names,scores)
```

You can also explore the alignment from the command window; the `hmmprofmerge` function with one output argument places the aligned sequences into a char array.

```
str = hmmprofmerge(aligned_seqs);
str(1:10,1:80)
```

```
ans =
```

```
10x80 char array
```

```
'YILVKAIYTLGYSVS.LMSLATGSIILCLF.RKLHCTR.NYIHLNLFSLFILRAISVLVK.DDVLYSSSG-TLH.....'
'FRSVKIGYTIHGSVS.LISLTTAIVILCMS.RKLHCTR.NYIHMHLFVSVFILKAIIVFVK.DAVLYDVIQESDN.....'
'YNTVKTGYTIHGSLS.LASLLVAMAILSLF.RKLHCTR.NYIHMHLFMSFILRATAVFIK.DMALFNSG-EIDH.....'
'FGAIKTGYTIHGSLS.LISLTAAMIILCIF.RKLHCTR.NYIHMHLFMSFIMRAIAVFIK.DIVLFESG-ESDH.....'
'YLSVKALYTVGYSTS.LVTLTTAMVILCRF.RKLHCTR.NFIHMNLFVSMFLRAISVFIK.DWILYAEQD-SSH.....'
'FSTVKIIYTTGHSIS.IVALCVAIAILVAL.RRLHCPR.NYIHTQLFATFILKASAVFLK.DAAIFQGDS-TDH.....'
'LSTLKQLYTAGYATS.LISLITAVIIFTCF.RKFHCTR.NYIHINLFVSVFILRATAVFIK.DAVLFSDET-QNH.....'
```

```
'FDRLGMIYTVGYSVS.LASLTVAVLILAYF.RRLHCTR.NYIHMHLFLSFMLRAVSIFVK.DAVLYSGATLDEA.....'
'FERLYVMYTVGYSIS.FGSLAVAILIIGYF.RRLHCTR.NYIHMHLFVSFMLRATSIFVK.DRVVHAHIGVKEL.....'
'ALNFLYLTIIHGHS.IASLLISLGIFFYF.KSLSCQR.ITLHKNLFFSFVCSNVVTIIH.LTAVANNQALVAT.....'
```

Looking for Similarity with Sequence Comparison

Having a profile HHM which describes this family has several advantages over plain sequence comparison. Suppose that you have a new oligonucleotide that you want to relate to the 7-transmembrane receptor family. For this example, get a protein sequence from NCBI and extract the aminoacid sequence.

```
mousegpcr = getgenpept('NP_783573');
Bai3 = mousegpcr.Sequence;
```

This sequence is also provided in the MAT-file `gpcrfam.mat`.

```
load('gpcrfam.mat','mousegpcr')
Bai3 = mousegpcr.Sequence;
```

```
seqdisp(Bai3,'row',70)
```

```
ans =
```

```
22x82 char array
```

```
'  1  MKAVRNLLIY  IFSTYLLVMF  GFNAAQDFWC  STLVKGVIIYG  SYSVSEMPFK  NFTNCTWTLE  NPDPTKYSIY'
'  71  LKFSKKDLSC  SNFSLLAYQF  DHFSHEKIKD  LLRKNHSIMQ  LCSSKNAFV  LQYDKNFIQI  RRVFPTDFPG'
' 141  LQKKVEEDQK  SFFEFLVLNK  VSPSQFGCHV  LCTWLESCLK  SENGRTESCG  IMYTKCTCPQ  HLGEGWIDDQ'
' 211  SLVLLNNVVL  PLNEQTEGCL  TQELQTTQVC  NLTREAKRPP  KEEFGMMGDH  TIKSQRPRSV  HEKRVPQEQA'
' 281  DAAKFMAQTG  ESGVEEWSQW  SACSVCQGQ  SQVRTRTCVS  PYGTHCSGPL  RESRVCNNTA  LCPVHGVWEE'
' 351  WPSWLSCSFT  CGRGQRTRTR  SCTPPQYGGR  PCEGPETHHK  PCNIALCPVD  GQWQEWSSWS  HCSVTCNGT'
' 421  QQRSRQCTAA  AHGGSECRGP  WAESRECYNP  ECTANGQWNQ  WGHWSGCSKS  CDGGWERRMR  TCQGAAVTGG'
' 491  QCEGTGEEVR  RCSEQRCAP  YEICPEDYLI  SMVWKRTPAG  DLAFNQCP  ATGTTSTRCS  LSLHGVASWE'
' 561  QPSFARCISN  EYRHLQHSIK  EHLAKGQRM  AGDGMSQVTK  TLLDLTQRKN  FYAGDLLVSV  EILRNVTDTF'
' 631  KRASYIPASD  GVQNFFQIVS  NLLDEENKEK  WEDAQQIYPG  SIELMQVIED  FIIHIVGMGM  DFQNSYLMTG'
' 701  NVVASIQKLP  AASVLTDFIN  PMKGRKGMVD  WARNSEDRVV  IPKSIFTPVS  SKELDESSVF  VLGAVLYKNL'
' 771  DLILPTLRNY  TVVNSKVIVV  TIRPEPKTTD  SFLEIELAHL  ANGTLNYPYCV  LWDDSKSNES  LGTWSTQGCK'
' 841  TVLTDASHTK  CLCDRLSTFA  ILAQPREIV  MESSGTPSVT  LIVGSGLSCL  ALITLAVVYA  ALWRYIRSER'
' 911  SIILINFCLS  IISSNILILV  GQTQTHNSI  CTTTAF  FFLASF  TEAWQSYMAV  TGKIRTRLIR'
' 981  KRFLCLGWGL  PALVVATSVG  FTRTKGYGTD  HYCWLSLEGG  LLYAFVGPAA  AVVLVMMVIG  ILVFNKLVSR'
'1051  DGILDKKLKH  RAGQMSEPHS  GLTLKCAKCG  VVSTTALSAT  TASNAMASLW  SSCVVLPLLA  LTWMSAVLAM'
'1121  TDKRSILFQI  LFAVFDSLQ  FVIVMVHCIL  RREVQDAFC  RLRNCQDPIN  ADSSSSFPNG  HAQIMTDFEK'
'1191  DVDIACRSVL  HKDIGPCRAA  TITGTL  LNDDEEEKGT  NPEGLSYSTL  PGNVISKVII  QOPTGLHMPM'
'1261  SMNELSNPCL  KKENTELRRT  VYLCTDDNLR  GADMIVHPQ  ERMESDYIV  MPRSSVSTQP  SMKEESKMNI'
'1331  GMETLPHERL  LHYKVNPEFN  MNPPVMDQFN  MNLDQHLAPQ  EHMQNLPFEP  RTAVKNFMAS  ELDDNVGLSR'
'1401  SETGSTISMS  SLERRKSRY  DLDFEKVMHT  RKRHMELFQE  LNQKFQTLDR  FRDIPNTSSM  ENPAPNKNPW'
'1471  DTFKPPSEYQ  HYTTINVLDT  EAKDTLELRP  AEWKCLNLP  LDVQEGDFQT  EV'
```

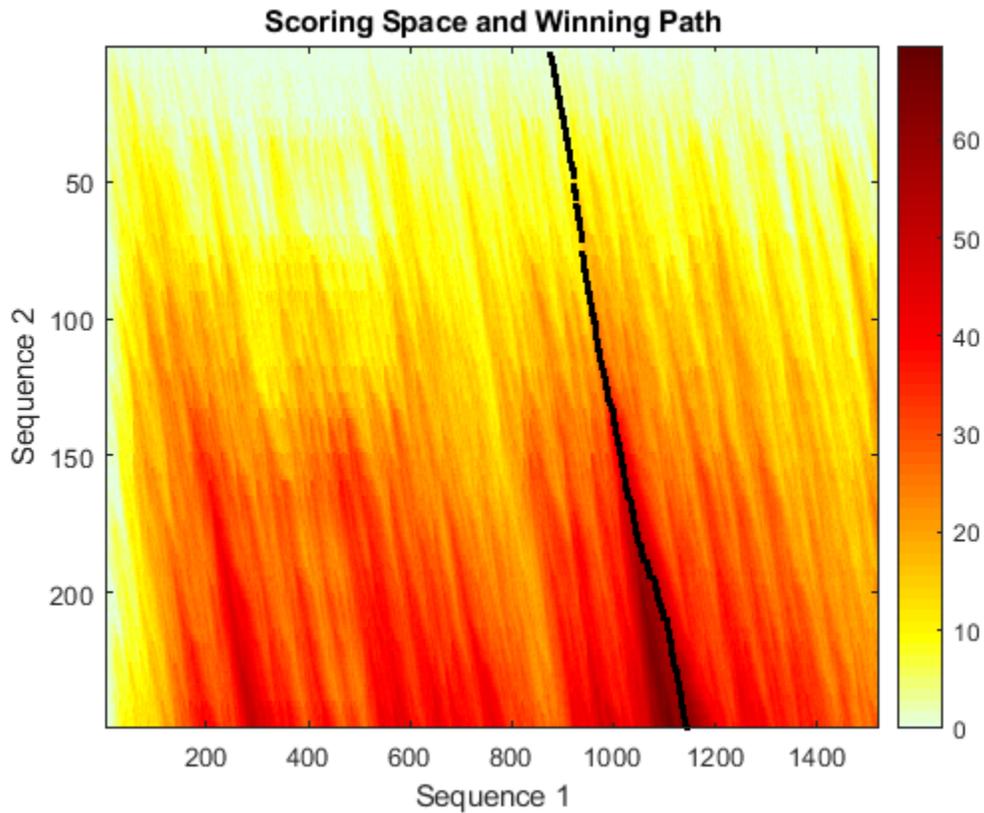
First, using local alignment compare the new sequence to one of the sequences in the multiple alignment. For instance use the first sequence, in this case the human protein 'VIPR2'. The Smith-Waterman algorithm (`swalign`) can make use of scoring matrices. Scoring matrices can capture the probability of substitution of symbols. The sequences in this example are known to be only distantly related, so BLOSUM30 is a good choice for the scoring matrix.

```
VIPR2 = seqs{1};
[sc_aa_affine, alignment] = swalign(Bai3,VIPR2,'ScoringMatrix',...
```

```

        'blosum30', 'gapopen', 5, 'extendgap', 3, 'showscore', true);
sc_aa_affine
sc_aa_affine =
    69.6000

```

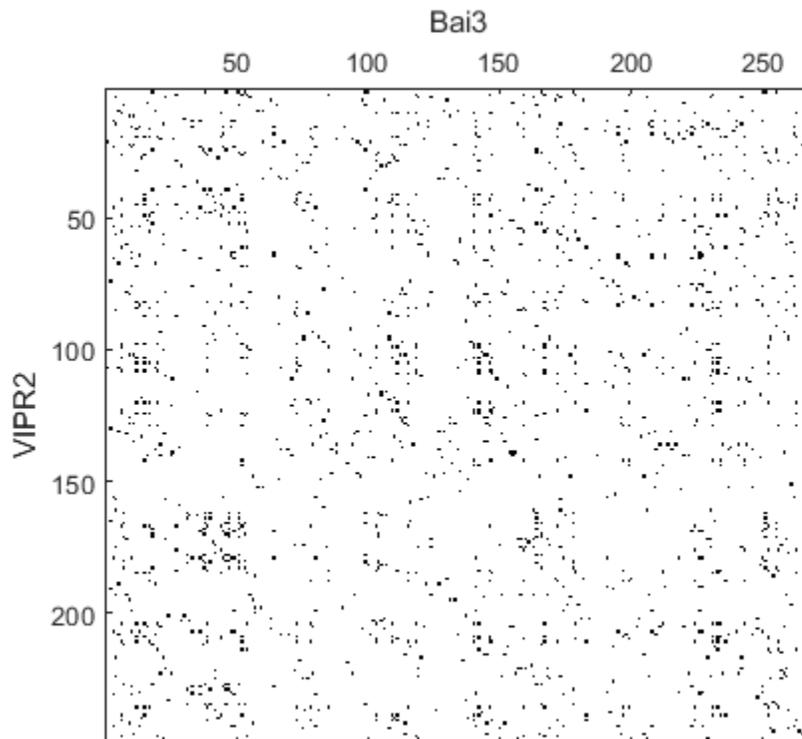


By looking at the scoring space, apparently, both sequences are related. However, this relationship could not be inferred from a dot plot.

```

Bai3_aligned_region = strep(alignment(1,:), '- ', '');
seqdotplot(VIPR2, Bai3_aligned_region, 7, 2)
ylabel('VIPR2'); xlabel('Bai3');

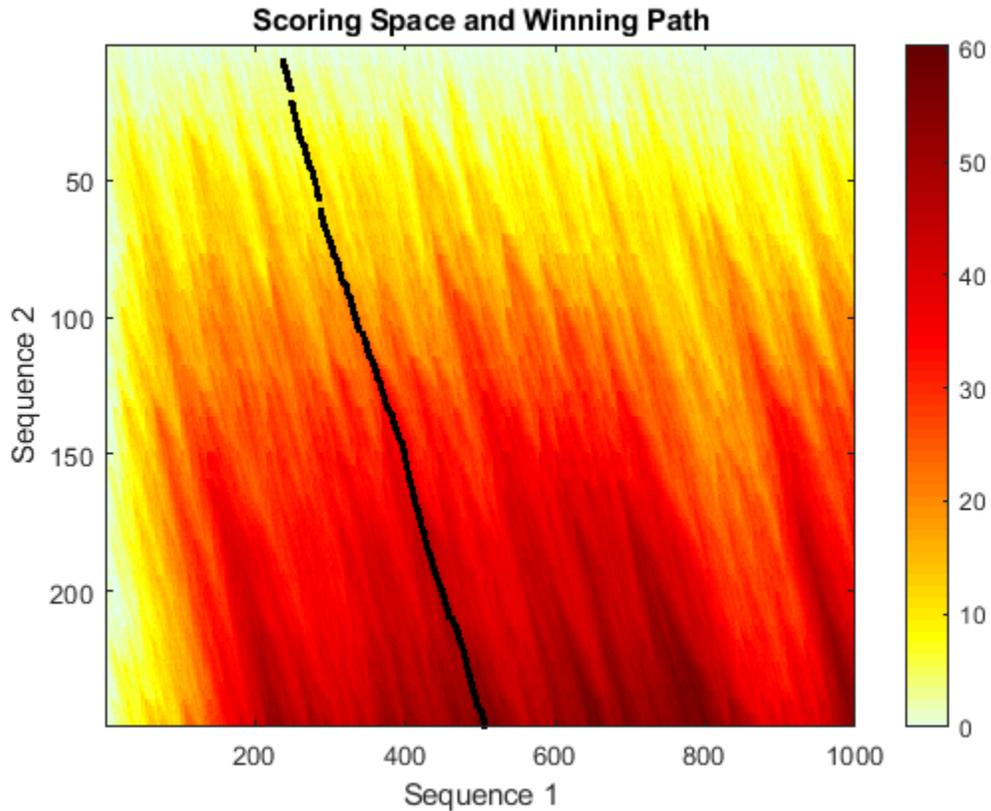
```



Is either of these two examples enough evidence to affirm that these sequences are related? One way to test this is to randomly create a fake sequence with the same distribution of amino acids and see how it aligns to the family. Notice that the score of the local alignment between the fake sequence and the VIPR2 protein is not significantly lower than the score of the alignment between the Bai3 and VIPR2 proteins. To ensure reproducibility of the results of this example, we reset the global random generator.

```
rng(0, 'twister');  
fakeSeq = randseq(1000, 'FROMSTRUCTURE', aacount(VIPR2));  
sc_fk_affine = swalign(fakeSeq, VIPR2, 'ScoringMatrix', 'blosum30', ...  
                       'gapopen', 5, 'extendgap', 3, 'showscore', true)
```

```
sc_fk_affine =  
    60.4000
```



In contrast, when you align both sequences to the family using the trained profile HMM, the score of aligning the target sequence to the family profile is significantly larger than the score of aligning the fake sequence.

```
sc_aa_hmm = hmmprofalign(hmm_7tm,Bai3)
sc_fk_hmm = hmmprofalign(hmm_7tm,fakeSeq)
```

```
sc_aa_hmm =
    214.5286
```

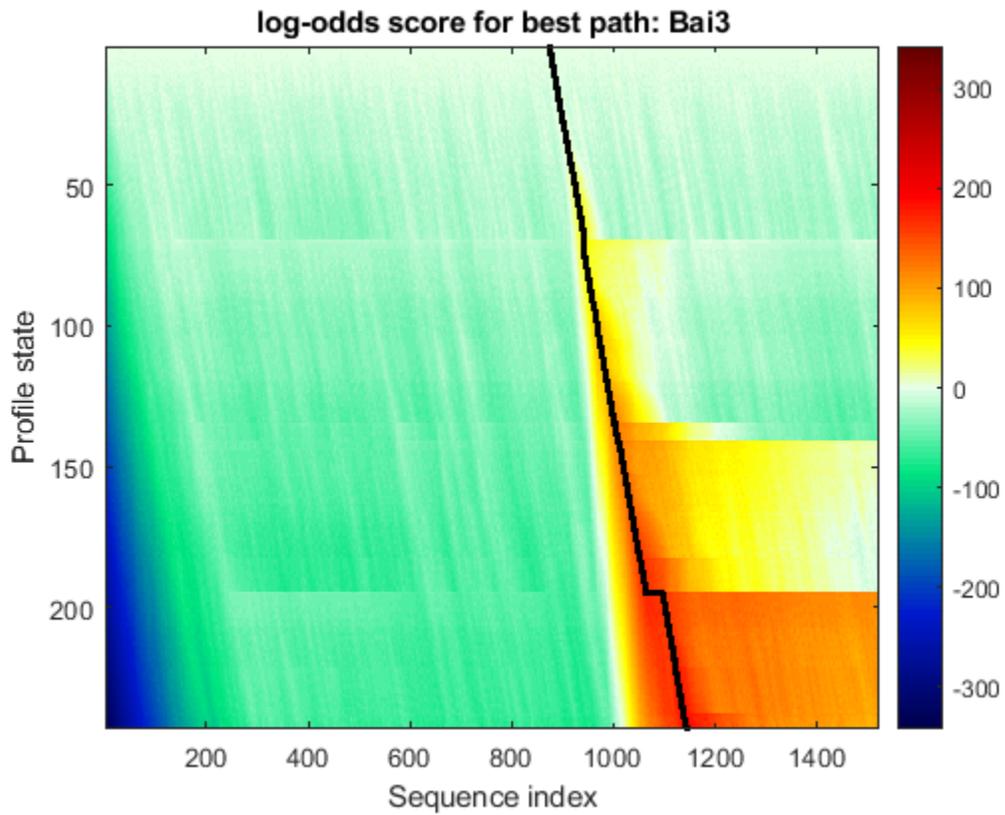
```
sc_fk_hmm =
   -49.1624
```

Exploring Profile HMM Alignment Options

Similarly to the `swalign` alignment function, when you use profile alignments you can visualize the scoring space using the `showscore` option to the `hmmprofalign` function.

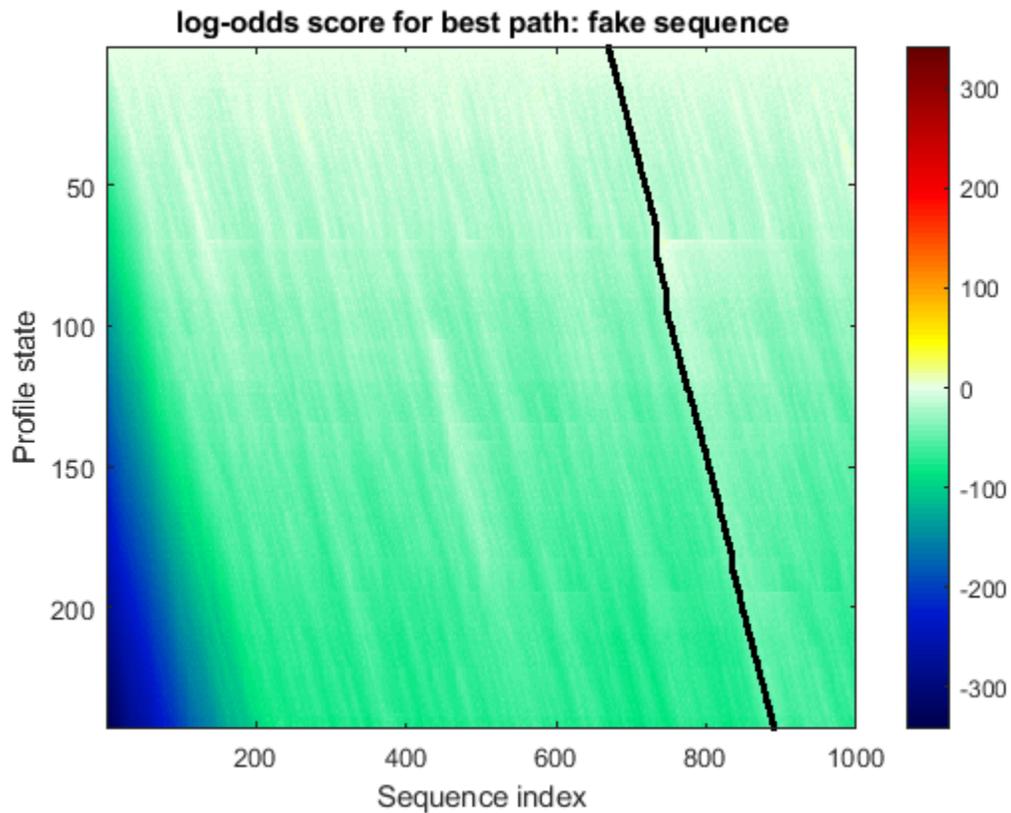
Display Bai3 aligned to the 7tm_2 family.

```
hmmprofalign(hmm_7tm,Bai3,'showscore',true);
title('log-odds score for best path: Bai3');
```



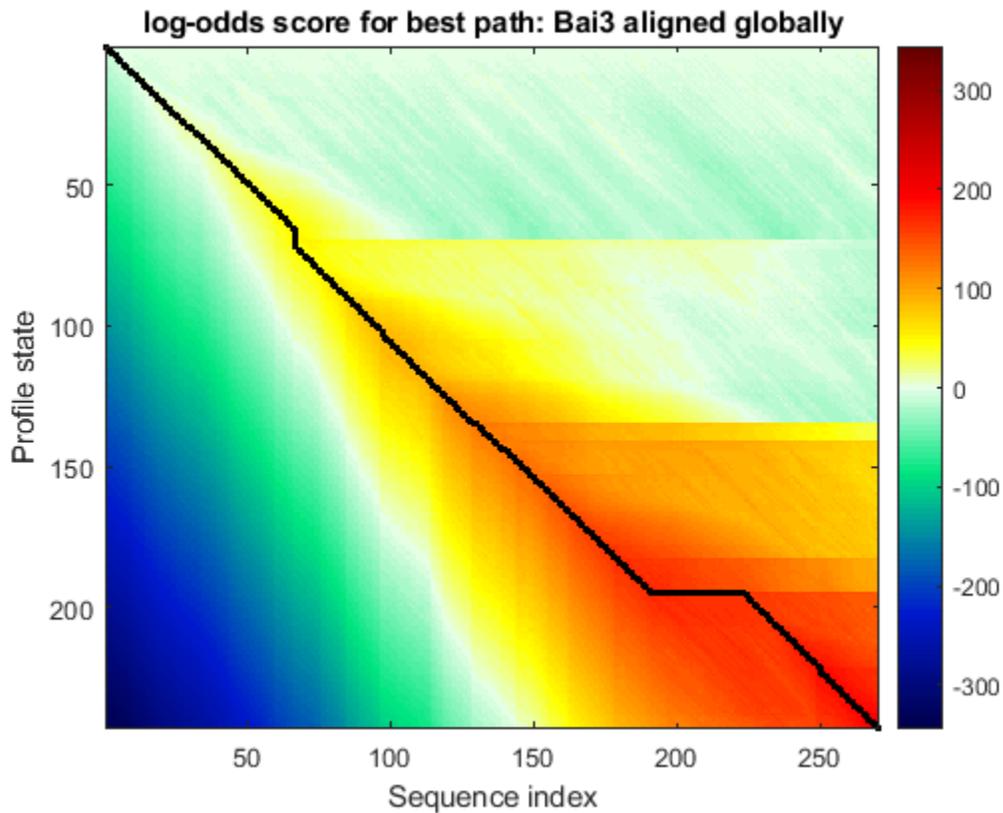
Display the "fake" sequence aligned to the 7tm_2 family.

```
hmmproalign(hmm_7tm,fakeSeq,'showscore',true);  
title('log-odds score for best path: fake sequence');
```



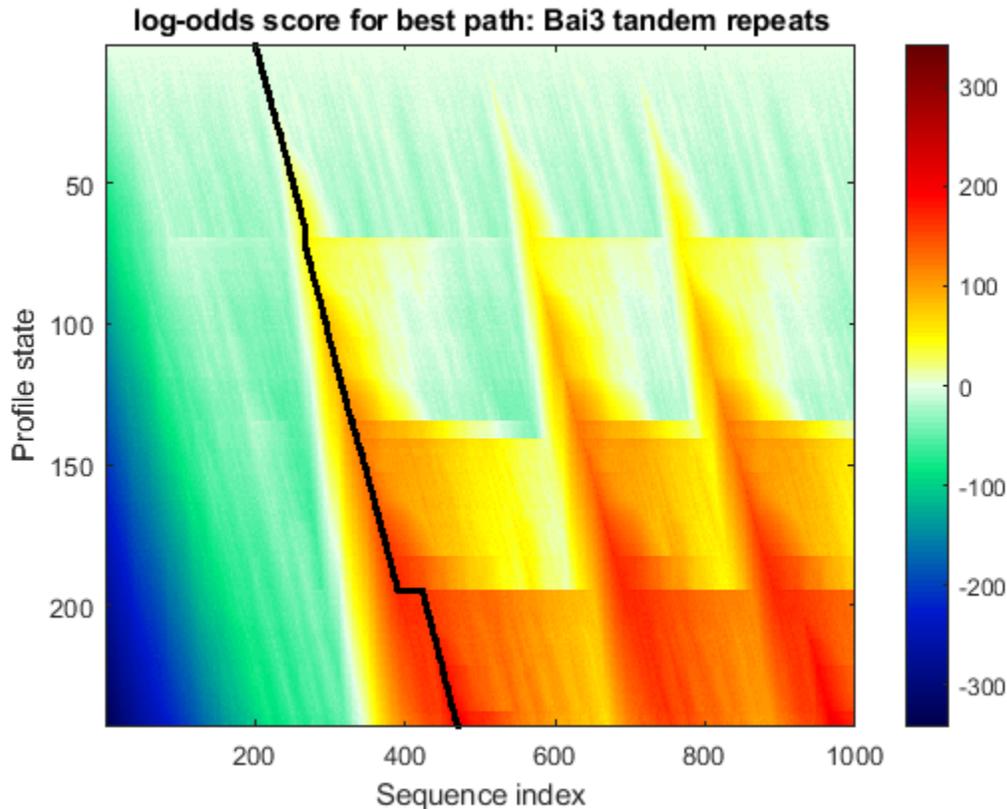
Display Bai3 globally aligned to the 7tm_2 family.

```
[sc_aa_hmm,align,ptrs] = hmmpfalign(hmm_7tm,Bai3);  
Bai3_hmmaligned_region = Bai3(min(ptrs):max(ptrs));  
hmmpfalign(hmm_7tm,Bai3_hmmaligned_region,'showscore',true);  
title('log-odds score for best path: Bai3 aligned globally');
```



Align tandemly repeated domains.

```
naa = numel(Bai3_hmmaligned_region);
repeats = randseq(1000, 'FROMSTRUCTURE', aaccount(Bai3)); %artificial example
repeats(200+(1:naa)) = Bai3_hmmaligned_region;
repeats(500+(1:naa)) = Bai3_hmmaligned_region;
repeats(700+(1:naa)) = Bai3_hmmaligned_region;
hmmprofalign(hmm_7tm, repeats, 'showscore', true);
title('log-odds score for best path: Bai3 tandem repeats');
```



Searching for Fragment Domains

In MATLAB®, you can search for fragment domains by manually activating the B->M and M->E transition probabilities of the HMM model.

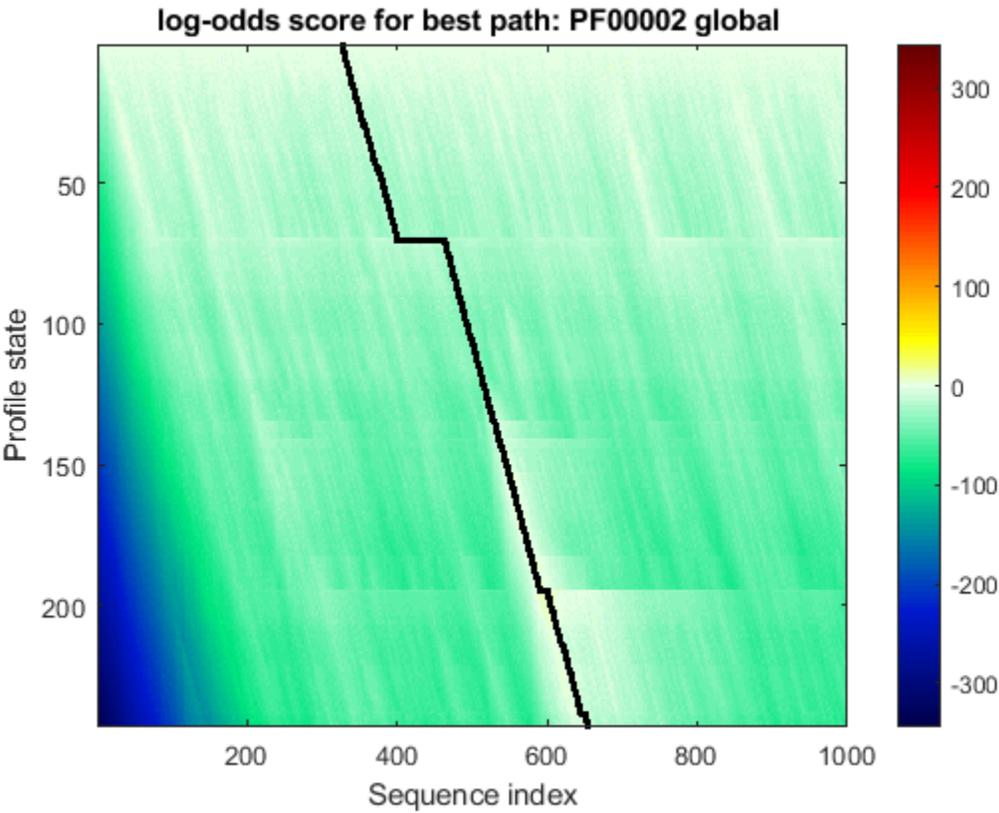
```
hmm_7tm_f = hmm_7tm;
hmm_7tm_f.BeginX(3:end)=.002;
hmm_7tm_f.MatchX(1:end-1,4)=.002;
```

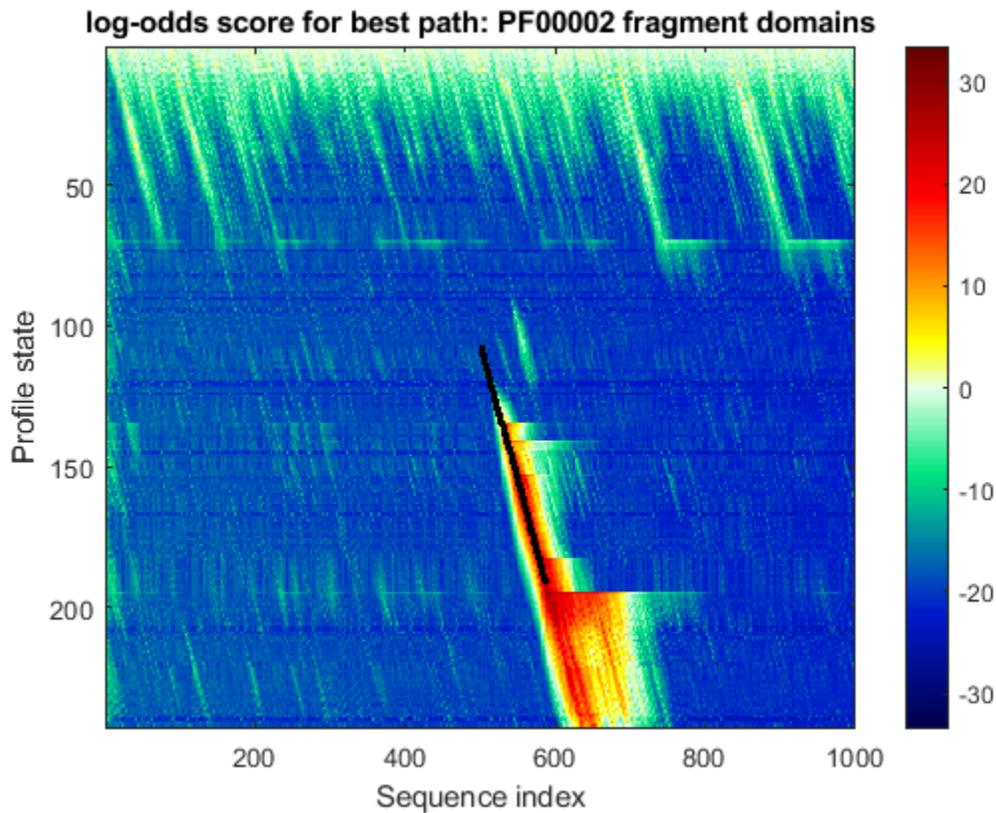
Create a random sequence, or fragment model, with a small insertion of the Bai3 protein:

```
fragment = randseq(1000, 'FROMSTRUCTURE', aaccount(Bai3));
fragment(501:550) = Bai3_hmmaligned_region(101:150);
```

Try aligning the random sequence with the inserted peptide to both models, the global and fragment model:

```
hmmprofalign(hmm_7tm, fragment, 'showscore', true);
title('log-odds score for best path: PF00002 global ');
hmmprofalign(hmm_7tm_f, fragment, 'showscore', true);
title('log-odds score for best path: PF00002 fragment domains');
```

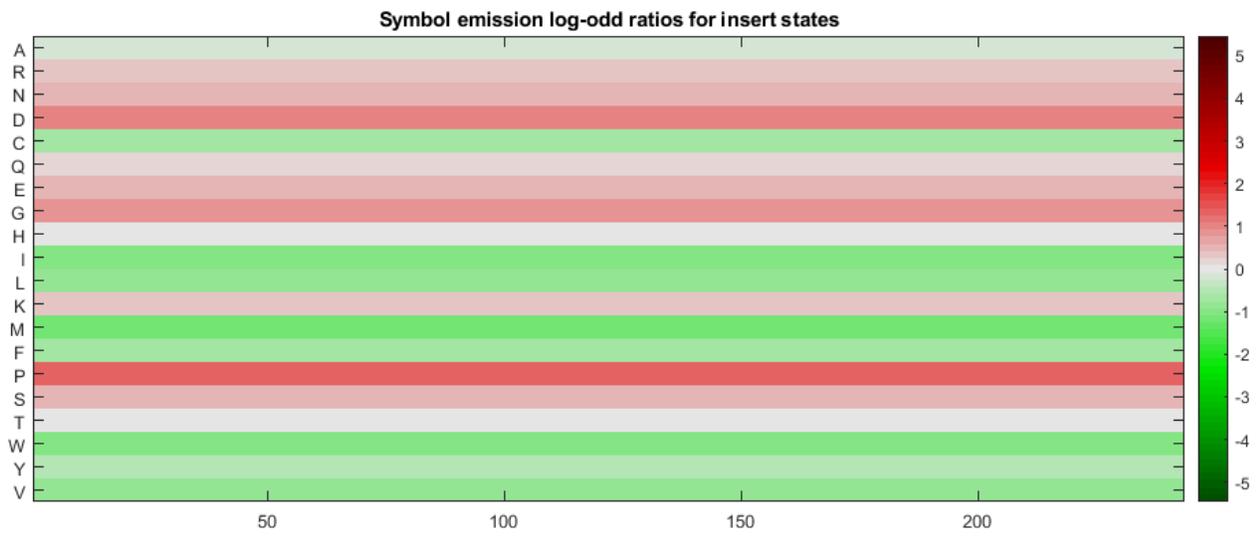
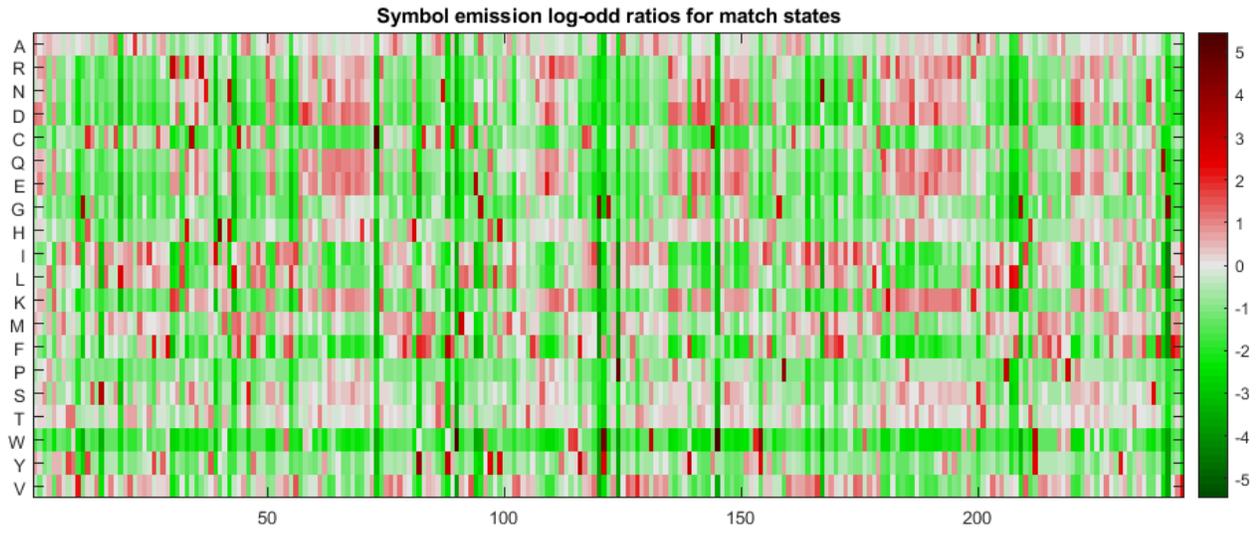


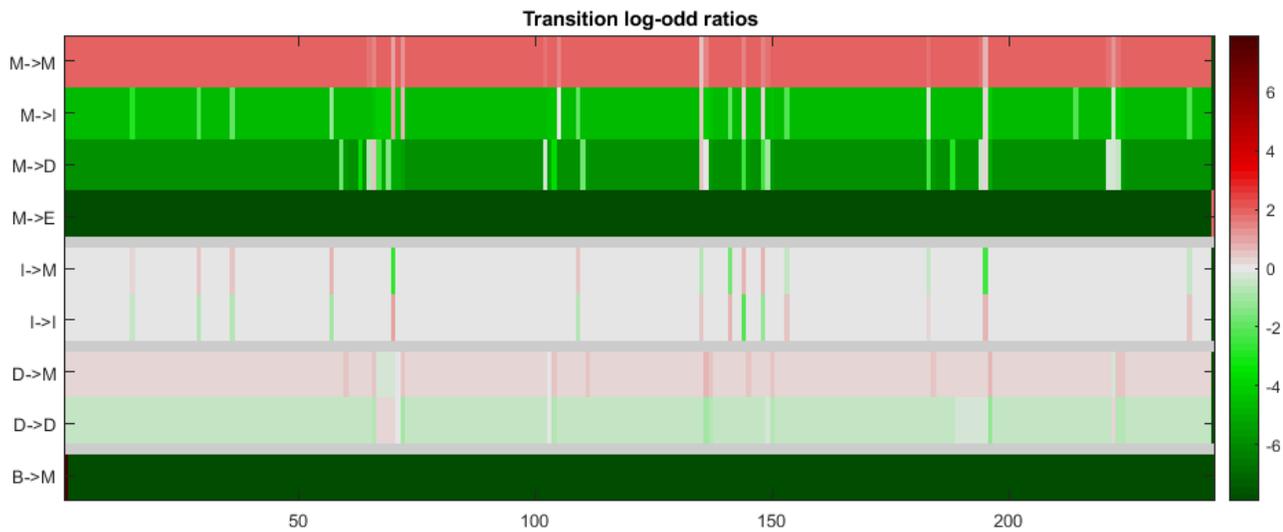


Exploring the Profile HMMs

The function `showhmmprof` is an interactive tool to explore the profile HMM. Try right and left mouse clicks over the model figures. There are three plots for each model: (1) the symbol emission probabilities in the Match states, (2) the symbol emission probabilities in the Insert states, and (3) the Transition probabilities.

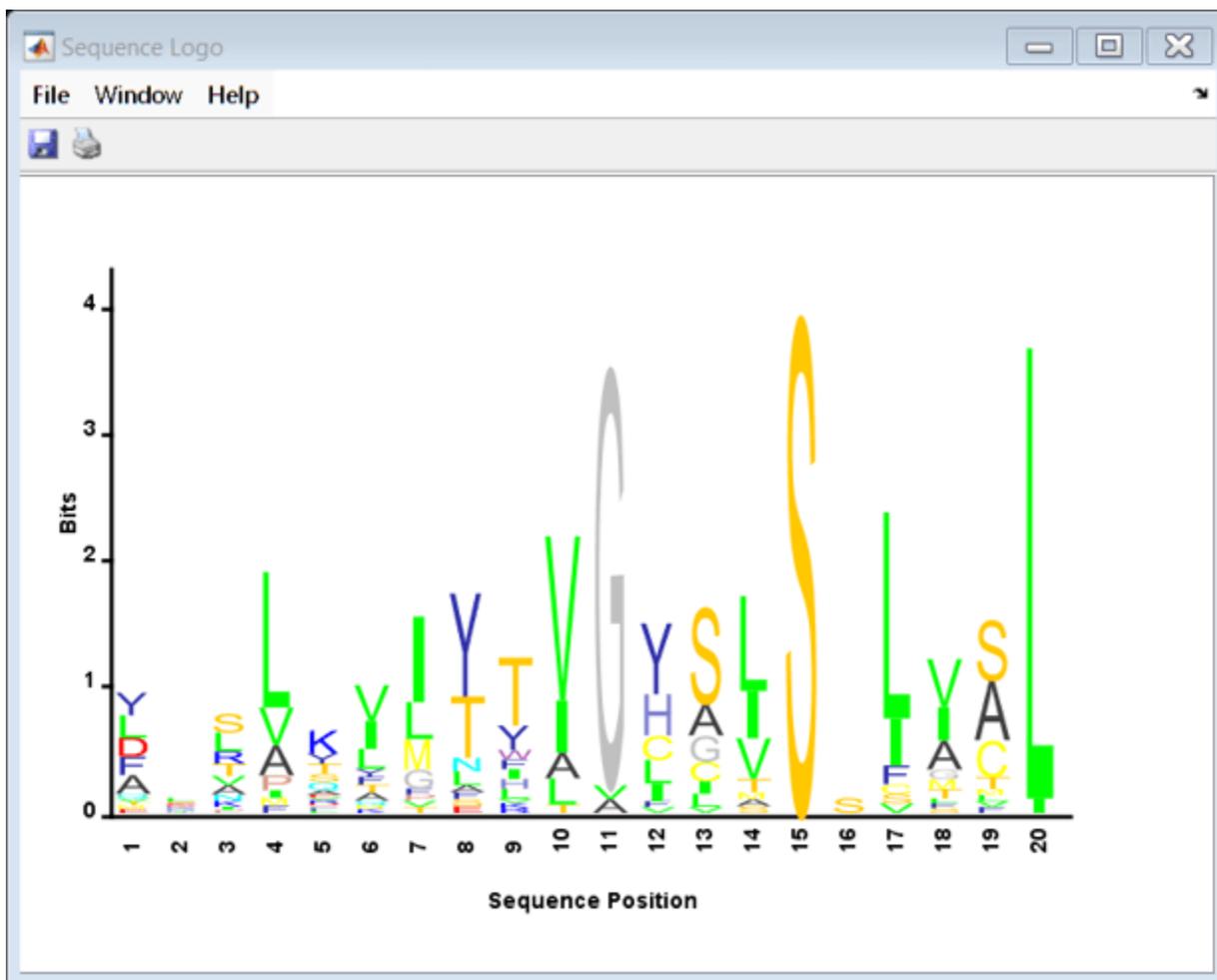
```
showhmmprof(hmm_7tm, 'scale', 'logodds')
```





An alternative method to explore a profile HMM is by creating a sequence logo from the multiple alignment. A sequence logo displays the frequency of bases found at each position within a given region, usually for a binding site. Using the `hmm_7tm` sequences, consider the portion of the Parathyroid hormone-related peptide receptor (precursor) found at the n-terminus of the PTRR_Human sequence. The `seqlogo` allows a quick visual comparison of how well this region is conserved across the 7tm family.

```
seqlogo(str, 'startat', 1, 'endat', 20, 'alphabet', 'AA')
```



Profile Estimation

Profile HMMs can also be estimated from a multiple alignment. As new sequences related to the family are found, it is possible to re-estimate the model parameters.

```
hmm_7tm_new = hmmprofileestimate(hmm_7tm, str)
```

```
hmm_7tm_new =
```

```
struct with fields:
```

```

    Name: '7tm_2'
    PfamAccessionNumber: 'PF00002.19'
    ModelDescription: '7 transmembrane receptor (Secretin family)'
    ModelLength: 243
    Alphabet: 'AA'
    MatchEmission: [243x20 double]
    InsertEmission: [243x20 double]
    NullEmission: [0.0768 0.0418 0.0396 0.0305 0.0201 ... ] (1x20 double)
    BeginX: [244x1 double]
    MatchX: [242x4 double]
    InsertX: [242x2 double]

```

```

DeleteX: [242x2 double]
FlankingInsertX: [2x2 double]
LoopX: [2x2 double]
NullX: [2x1 double]

```

In case your sequences are not pre-aligned, you can also utilize the `multialign` function before estimating a new HMM profile. It is possible to refine the HMM profile by re-aligning the sequences to the model and re-estimating the model iteratively until you converge to a locally optimal model.

```

aligned_seqs = multialign(seqs);
hmm_7tm_ma = hmmprofileestimate(hmmprofilestruct(270),aligned_seqs)
showhmmprofile(hmm_7tm_ma,'scale','logodds')
close; close; % close insertion emission prob. and transition prob.

```

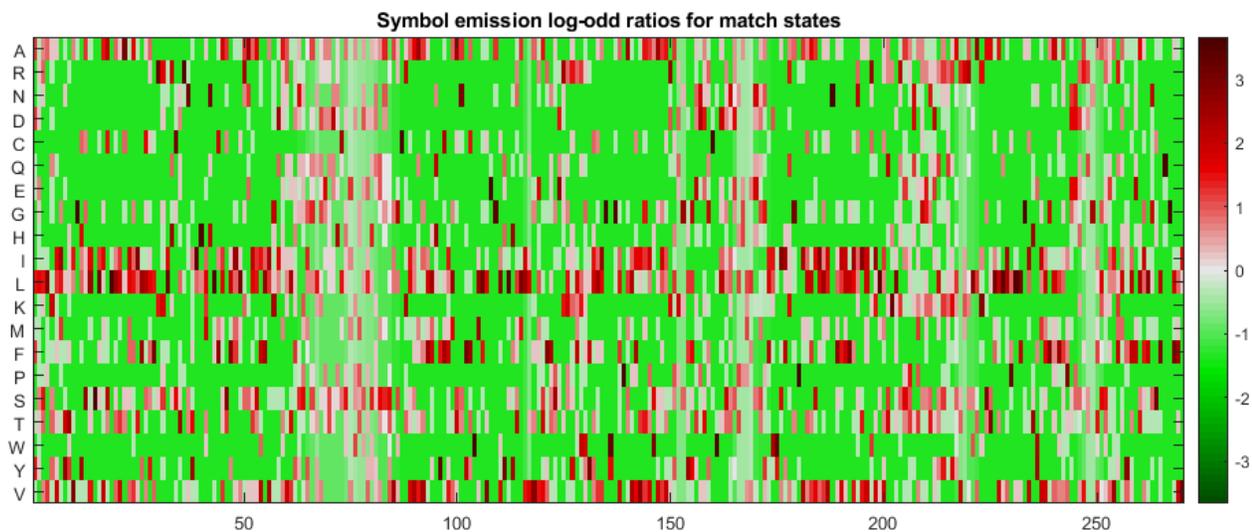
```
hmm_7tm_ma =
```

```
struct with fields:
```

```

ModelLength: 270
Alphabet: 'AA'
MatchEmission: [270x20 double]
InsertEmission: [270x20 double]
NullEmission: [0.0500 0.0500 0.0500 0.0500 0.0500 ... ] (1x20 double)
BeginX: [271x1 double]
MatchX: [269x4 double]
InsertX: [269x2 double]
DeleteX: [269x2 double]
FlankingInsertX: [2x2 double]
LoopX: [2x2 double]
NullX: [2x1 double]

```



Align all sequences to the new model.

```

fprintf('Aligning sequences ')
scores = zeros(numel(seqs),1);

```

```

aligned_seqs = cell(numel(seqs),1);
for sn=1:numel(seqs)
    fprintf('.')
    [scores(sn),aligned_seqs{sn}]=hmmprofalign(hmm_7tm_ma,seqs{sn});
end
fprintf('\n')

str = hmmprofmerge(aligned_seqs);
str(1:10,1:80)

Aligning sequences .....

ans =

    10x80 char array

'YILVKAIYTLGYSVSLMSLATGSIIILCLF.RKLHCTRNYIHLNLFILRAISVLVKDDVLYSS---SGTLHCP-....'
'FRSVKIGYTIHGSVSLISLTTAIVILCMS.RKLHCTRNYIHMHLFVSFILKAIAVFVKDAVLYDVIQ--ESDNCS-....'
'YNTVKGTGYTIGYSLASLLVAMAILSLF.RKLHCTRNYIHMHLFMSFILRATAVFIKDMALFNS---GEIDHCS-....'
'FGAIKTGYTIHGSLSLISLTAAMIILCIF.RKLHCTRNYIHMHLFMSFIMRAIAVFIKDIVLFES---GESDHCH-....'
'YLSVKALYTVGYSTSLVTLTTAMVILCRF.RKLHCTRNYIHMHLFVSFMLRAISVFIKDWILYAE---QDSSHCF-....'
'FSTVKIIYTTGHSISIVALCVAIAILVAL.RRLHCPRNYIHTQLFATFILKASAVFLKDAEIFQG---DSTDHCS-....'
'LSTLKQLYTAGYATSLISLITAVIIFTCF.RKFHCTRNYIHINLFVSFILRATAVFIKDAVLFSD---ETQNHCL-....'
'FDRLGMIYTVGYSVSLASLTVAVLILAYF.RRLHCTRNYIHMHLFSLFMLRAVSIFVKDAVLYSGATLDEAERLTE....'
'FERLYVMYTVGYSISFGSLAVAILIIGYF.RRLHCTRNYIHMHLFVSFMLRATSIFVKDRVVHAHIGVKELESIM....'
'ALNFLYLTIIHGHSIASLLISLGIFFYF.KSLSCQRITLHKNLFFSVCNSVVTIIHLTAVANNQALVATNP---....'

```

Show the aligned sequences in the browser.

```
hmmprofmerge(aligned_seqs,names,scores)
```

Predicting Protein Secondary Structure Using a Neural Network

This example shows a secondary structure prediction method that uses a feed-forward neural network and the functionality available with the Deep Learning Toolbox™.

It is a simplified example intended to illustrate the steps for setting up a neural network with the purpose of predicting secondary structure of proteins. Its configuration and training methods are not meant to be necessarily the best solution for the problem at hand.

Introduction

Neural network models attempt to simulate the information processing that occurs in the brain and are widely used in a variety of applications, including automated pattern recognition.

The Rost-Sander data set [1] consists of proteins whose structures span a relatively wide range of domain types, composition and length. The file `RostSanderDataset.mat` contains a subset of this data set, where the structural assignment of every residue is reported for each protein sequence.

```
load RostSanderDataset.mat

N = numel(allSeq);

id = allSeq(7).Header           % annotation of a given protein sequence
seq = int2aa(allSeq(7).Sequence) % protein sequence
str = allSeq(7).Structure       % structural assignment

id =

    '1CSE-ICOMPLEX(SERINEPROTEINASE-INHIBITOR)03-JU'

seq =

    'KSFPEVVGKTVDQAREYFTLHYPQYNVYFLPEGSPVTLDLRYNRVRFYNPGTNVVNHVPHVG'

str =

    'CCCHHHCCCCHHHHHHHHHHCCCCEEEEECCCECCCECCCCCEEEEECCCECCCECCCEEC'
```

In this example, you will build a neural network to learn the structural state (helix, sheet or coil) of each residue in a given protein, based on the structural patterns observed during a training phase. Due to the random nature of some steps in the following approach, numeric results might be slightly different every time the network is trained or a prediction is simulated. To ensure reproducibility of the results, we reset the global random generator to a saved state included in the loaded file, as shown below:

```
rng(savedState);
```

Define the Network Architecture

For the current problem we define a neural network with one input layer, one hidden layer and one output layer. The input layer encodes a sliding window in each input amino acid sequence, and a

prediction is made on the structural state of the central residue in the window. We choose a window of size 17 based on the statistical correlation found between the secondary structure of a given residue position and the eight residues on either side of the prediction point [2]. Each window position is encoded using a binary array of size 20, having one element for each amino acid type. In each group of 20 inputs, the element corresponding to the amino acid type in the given position is set to 1, while all other inputs are set to 0. Thus, the input layer consists of $R = 17 \times 20$ input units, i.e. 17 groups of 20 inputs each.

In the following code, we first determine for each protein sequence all the possible subsequences corresponding to a sliding window of size W by creating a Hankel matrix, where the i th column represents the subsequence starting at the i th position in the original sequence. Then for each position in the window, we create an array of size 20, and we set the j th element to 1 if the residue in the given position has a numeric representation equal to j .

```
W = 17; % sliding window size

% === binarization of the inputs
for i = 1:N
    seq = double(allSeq(i).Sequence); % current sequence
    win = hankel(seq(1:W),seq(W:end)); % all possible sliding windows
    myP = zeros(20*W,size(win,2)); % input matrix for current sequence
    for k = 1:size(win, 2)
        index = 20*(0:W-1)' + win(:,k); % input array for each position k
        myP(index,k) = 1;
    end
    allSeq(i).P = myP;
end
```

The output layer of our neural network consists of three units, one for each of the considered structural states (or classes), which are encoded using a binary scheme. To create the target matrix for the neural network, we first obtain, from the data, the structural assignments of all possible subsequences corresponding to the sliding window. Then we consider the central position in each window and transform the corresponding structural assignment using the following binary encoding: 1 0 0 for coil, 0 1 0 for sheet, 0 0 1 for helix.

```
cr = ceil(W/2); % central residue position

% === binarization of the targets
for i = 1:N
    str = double(allSeq(i).Structure); % current structural assignment
    win = hankel(str(1:W),str(W:end)); % all possible sliding windows
    myT = false(3,size(win,2));
    myT(1,:) = win(cr,:) == double('C');
    myT(2,:) = win(cr,:) == double('E');
    myT(3,:) = win(cr,:) == double('H');
    allSeq(i).T = myT;
end
```

You can perform the binarization of the input and target matrix described in the two steps above in a more concise way by executing the following equivalent code:

```
% === concise binarization of the inputs and targets
for i = 1:N
    seq = double(allSeq(i).Sequence);
    win = hankel(seq(1:W),seq(W:end)); % concurrent inputs (sliding windows)
```

```

% === binarization of the input matrix
allSeq(i).P = kron(win,ones(20,1)) == kron(ones(size(win)),(1:20)');

% === binarization of the target matrix
allSeq(i).T = allSeq(i).Structure(repmat((W+1)/2:end-(W-1)/2,3,1)) == ...
    repmat('CEH',1,length(allSeq(i).Structure)-W+1);
end

```

Once we define the input and target matrices for each sequence, we create an input matrix, P, and target matrix, T, representing the encoding for all the sequences fed into the network.

```

% === construct input and target matrices
P = double([allSeq.P]); % input matrix
T = double([allSeq.T]); % target matrix

```

Create the Neural Network

The problem of secondary structure prediction can be thought of as a pattern recognition problem, where the network is trained to recognize the structural state of the central residue most likely to occur when specific residues in the given sliding window are observed. We create a pattern recognition neural network using the input and target matrices defined above and specifying a hidden layer of size 3.

```

hsize = 3;
net = patternnet(hsize);
net.layers{1} % hidden layer
net.layers{2} % output layer

ans =

    Neural Network Layer

        name: 'Hidden'
    dimensions: 3
    distanceFcn: (none)
    distanceParam: (none)
        distances: []
        initFcn: 'initnw'
    netInputFcn: 'netsum'
    netInputParam: (none)
        positions: []
            range: [3x2 double]
            size: 3
    topologyFcn: (none)
    transferFcn: 'tansig'
    transferParam: (none)
        userdata: (your custom info)

ans =

    Neural Network Layer

        name: 'Output'
    dimensions: 0

```

```
distanceFcn: (none)
distanceParam: (none)
distances: []
initFcn: 'initnw'
netInputFcn: 'netsum'
netInputParam: (none)
positions: []
range: []
size: 0
topologyFcn: (none)
transferFcn: 'softmax'
transferParam: (none)
userdata: (your custom info)
```

Train the Neural Network

The pattern recognition network uses the default Scaled Conjugate Gradient algorithm for training, but other algorithms are available (see the Deep Learning Toolbox documentation for a list of available functions). At each training cycle, the training sequences are presented to the network through the sliding window defined above, one residue at a time. Each hidden unit transforms the signals received from the input layer by using a transfer function `logsig` to produce an output signal that is between and close to either 0 or 1, simulating the firing of a neuron [2]. Weights are adjusted so that the error between the observed output from each unit and the desired output specified by the target matrix is minimized.

```
% === use the log sigmoid as transfer function
net.layers{1}.transferFcn = 'logsig';

% === train the network
[net,tr] = train(net,P,T);
```

Network Diagram

Training Results

Training finished: Met validation criterion 

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	53	1000
Elapsed Time	-	00:00:26	-
Performance	0.456	0.271	0
Gradient	0.283	0.0437	1e-06

Training Algorithms

Data Division: Random dividerand
 Training: Scaled Conjugate Gradient trainscg
 Performance: Cross Entropy crossentropy
 Calculations: MEX

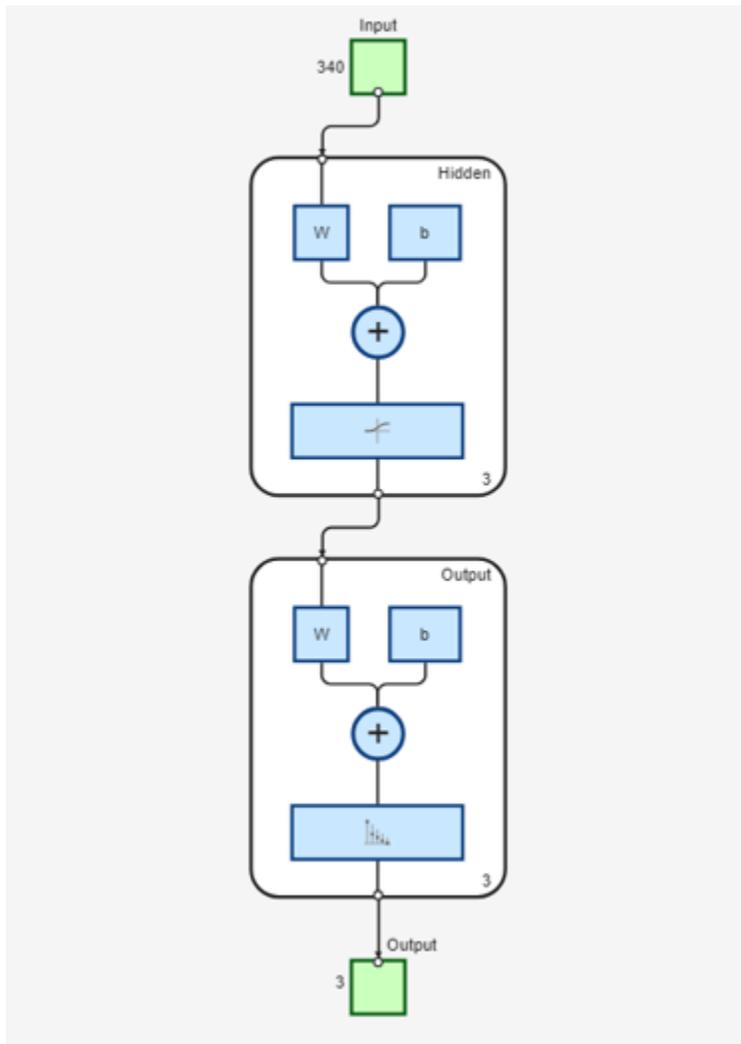
Training Plots

Performance Training State
 Error Histogram Confusion
 Receiver Operating Characteristic

During training, the training tool window opens and displays the progress. Training details such as the algorithm, the performance criteria, the type of error considered, etc. are shown.

Use the function view to generate a graphical view of the neural network.

```
view(net)
```

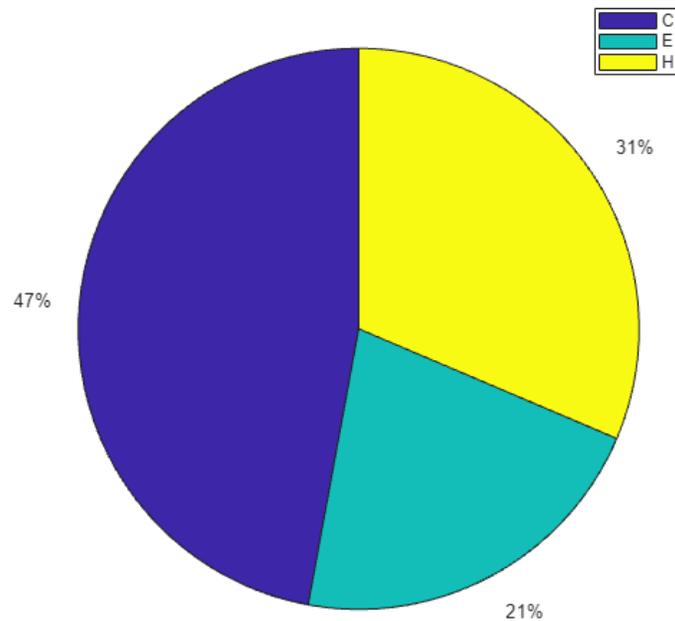


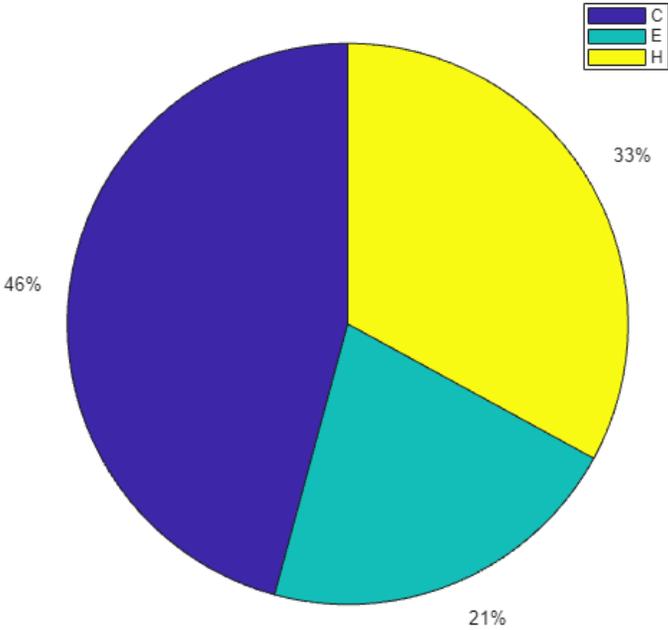
One common problem that occurs during neural network training is data overfitting, where the network tends to memorize the training examples without learning how to generalize to new situations. The default method for improving generalization is called early stopping and consists in dividing the available training data set into three subsets: (i) the training set, which is used for computing the gradient and updating the network weights and biases; (ii) the validation set, whose error is monitored during the training process because it tends to increase when data is overfitted; and (iii) the test set, whose error can be used to assess the quality of the division of the data set.

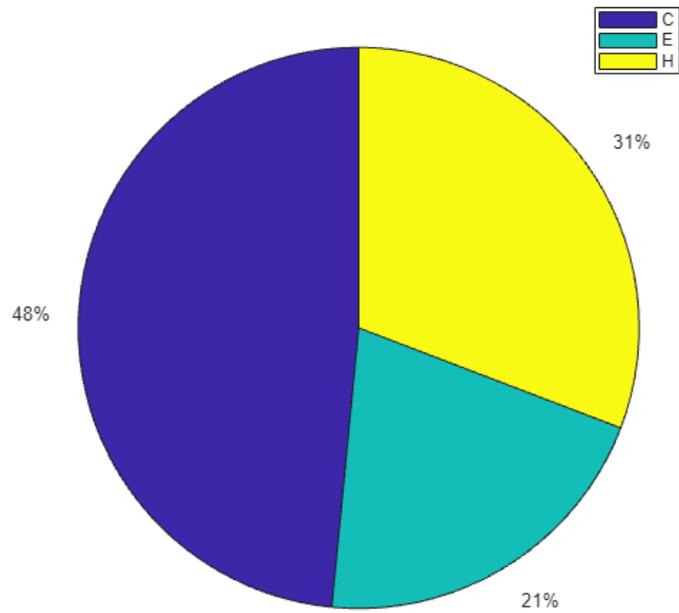
When using the function `train`, by default, the data is randomly divided so that 60% of the samples are assigned to the training set, 20% to the validation set, and 20% to the test set, but other types of partitioning can be applied by specifying the property `net.divideFcn` (default `dividerand`). The structural composition of the residues in the three subsets is comparable, as seen from the following survey:

```
[i,j] = find(T(:,tr.trainInd));
Ctrain = sum(i == 1)/length(i);
Etrain = sum(i == 2)/length(i);
Htrain = sum(i == 3)/length(i);
```

```
[i,j] = find(T(:,tr.valInd));  
Cval = sum(i == 1)/length(i);  
Eval = sum(i == 2)/length(i);  
Hval = sum(i == 3)/length(i);  
  
[i,j] = find(T(:,tr.testInd));  
Ctest = sum(i == 1)/length(i);  
Etest = sum(i == 2)/length(i);  
Htest = sum(i == 3)/length(i);  
  
figure()  
pie([Ctrain; Etrain; Htrain]);  
title('Structural assignments in training data set');  
legend('C', 'E', 'H')  
  
figure()  
pie([Cval; Eval; Hval]);  
title('Structural assignments in validation data set');  
legend('C', 'E', 'H')  
  
figure()  
pie([Ctest; Etest; Htest]);  
title('Structural assignments in testing data set ');  
legend('C', 'E', 'H')
```

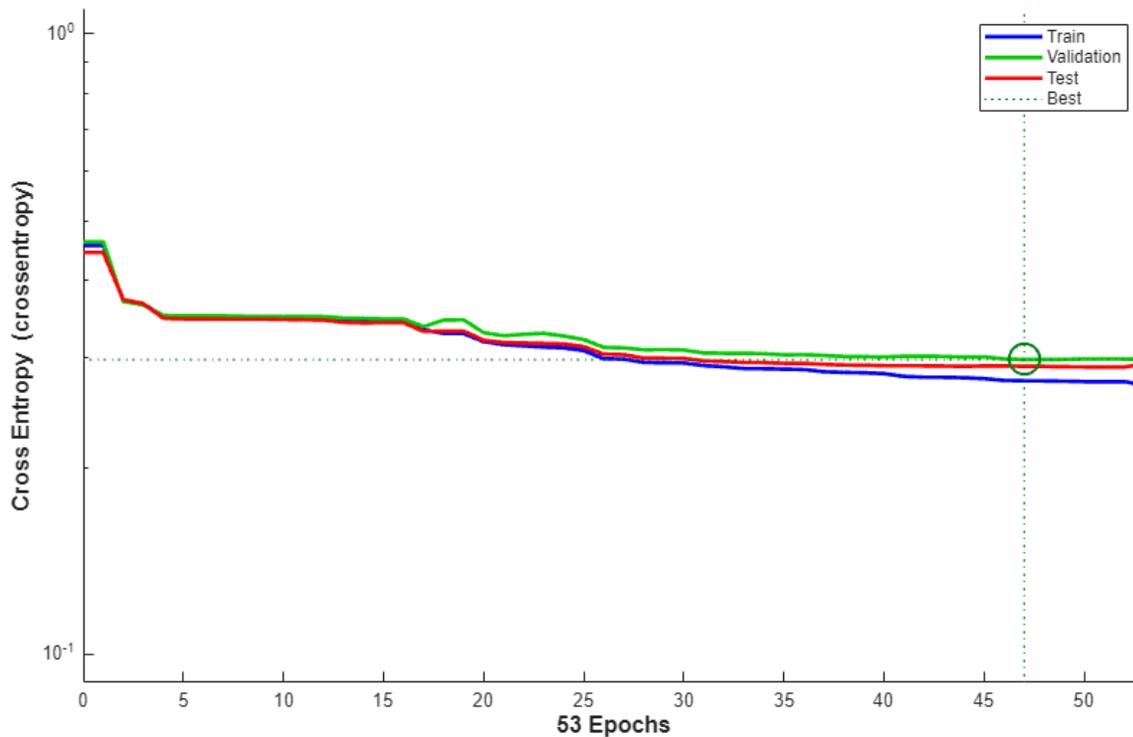






The function `plotperform` displays the trends of the training, validation, and test errors as training iterations pass.

```
figure()  
plotperform(tr)
```



The training process stops when one of several conditions (see `net.trainParam`) is met. For example, in the training considered, the training process stops when the validation error increases for a specified number of iterations (6) or the maximum number of allowed iterations is reached (1000).

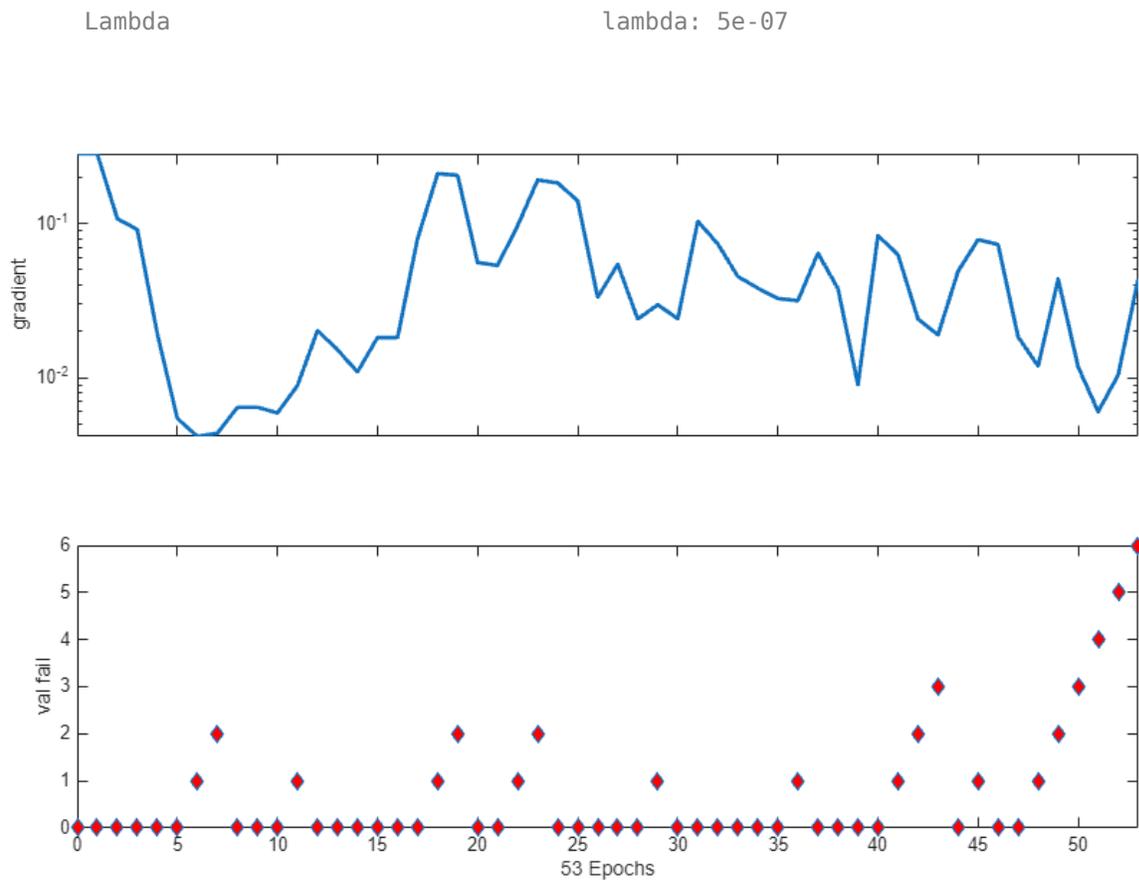
```
% === display training parameters
net.trainParam

% === plot validation checks and gradient
figure()
plottrainstate(tr)
```

```
ans =
```

```
Function Parameters for 'trainscg'
```

```
Show Training Window Feedback showWindow: true
Show Command Line Feedback showCommandLine: false
Command Line Frequency show: 25
Maximum Epochs epochs: 1000
Maximum Training Time time: Inf
Performance Goal goal: 0
Minimum Gradient min_grad: 1e-06
Maximum Validation Checks max_fail: 6
Sigma sigma: 5e-05
```



Analyze the Network Response

To analyze the network response, we examine the confusion matrix by considering the outputs of the trained network and comparing them to the expected results (targets).

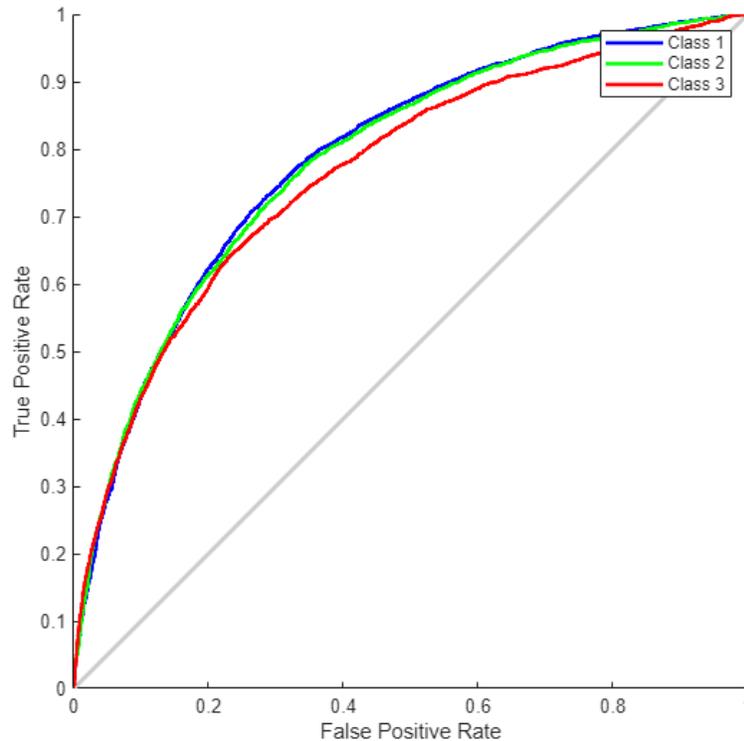
```
O = sim(net,P);  
figure()  
plotconfusion(T,O);
```

1	5635 36.4%	1148 7.4%	1641 10.6%	66.9% 33.1%
2	673 4.4%	1508 9.7%	640 4.1%	53.5% 46.5%
3	982 6.3%	645 4.2%	2599 16.8%	61.5% 38.5%
	77.3% 22.7%	45.7% 54.3%	53.3% 46.7%	63.0% 37.0%
	1	2	3	
	Target Class			

The diagonal cells show the number of residue positions that were correctly classified for each structural class. The off-diagonal cells show the number of residue positions that were misclassified (e.g. helical positions predicted as coiled positions). The diagonal cells correspond to observations that are correctly classified. Both the number of observations and the percentage of the total number of observations are shown in each cell. The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the precision (or positive predictive value) and false discovery rate, respectively. The row at the bottom of the plot shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the recall (or true positive rate) and false negative rate, respectively. The cell in the bottom right of the plot shows the overall accuracy.

We can also consider the Receiver Operating Characteristic (ROC) curve, a plot of the true positive rate (sensitivity) versus the false positive rate (1 - specificity).

```
figure()
plotroc(T,0);
```



Refine the Neural Network for More Accurate Results

The neural network that we have defined is relative simple. To achieve some improvements in the prediction accuracy we could try one of the following:

- Increase the number of training vectors. Increasing the number of sequences dedicated to training requires a larger curated database of protein structures, with an appropriate distribution of coiled, helical and sheet elements.
- Increase the number of input values. Increasing the window size or adding more relevant information, such as biochemical properties of the amino acids, are valid options.
- Use a different training algorithm. Various algorithms differ in memory and speed requirements. For example, the Scaled Conjugate Gradient algorithm is relatively slow but memory efficient, while the Levenberg-Marquardt is faster but more demanding in terms of memory.
- Increase the number of hidden neurons. By adding more hidden units we generally obtain a more sophisticated network with the potential for better performances but we must be careful not to overfit the data.

We can specify more hidden layers or increased hidden layer size when the pattern recognition network is created, as shown below:

```
hsize = [3 4 2];
net3 = patternnet(hsize);

hsize = 20;
net20 = patternnet(hsize);
```

We can also assign the network initial weights to random values in the range -0.1 to 0.1 as suggested by the study reported in [2] by setting the `net20.IW` and `net20.LW` properties as follows:

```
% === assign random values in the range -.1 and .1 to the weights
net20.IW{1} = -.1 + (.1 + .1) .* rand(size(net20.IW{1}));
net20.LW{2} = -.1 + (.1 + .1) .* rand(size(net20.LW{2}));
```

In general, larger networks (with 20 or more hidden units) achieve better accuracy on the protein training set, but worse accuracy in the prediction accuracy. Because a 20-hidden-unit network involves almost 7,000 weights and biases, the network is generally able to fit the training set closely but loses the ability of generalization. The compromise between intensive training and prediction accuracy is one of the fundamental limitations of neural networks.

```
net20 = train(net20,P,T);

O20 = sim(net20,P);
numWeightsAndBiases = length(getx(net20))

numWeightsAndBiases =

    6883
```

Network Diagram

Training Results

Training finished: Met validation criterion 

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	68	1000
Elapsed Time	-	00:00:12	-
Performance	1.47	0.271	0
Gradient	5.63	0.0715	1e-06

Training Algorithms

Data Division: Random dividerand
 Training: Scaled Conjugate Gradient trainscg
 Performance: Cross Entropy crossentropy
 Calculations: MEX

Training Plots

Performance Training State

Error Histogram Confusion

Receiver Operating Characteristic

You can display the confusion matrices for training, validation and test subsets by clicking on the corresponding button in the training tool window.

Assess Network Performance

You can evaluate structure predictions in detail by calculating prediction quality indices [3], which indicate how well a particular state is predicted and whether overprediction or underprediction has occurred. We define the index $pcObs(S)$ for state S ($S = \{C, E, H\}$) as the number of residues

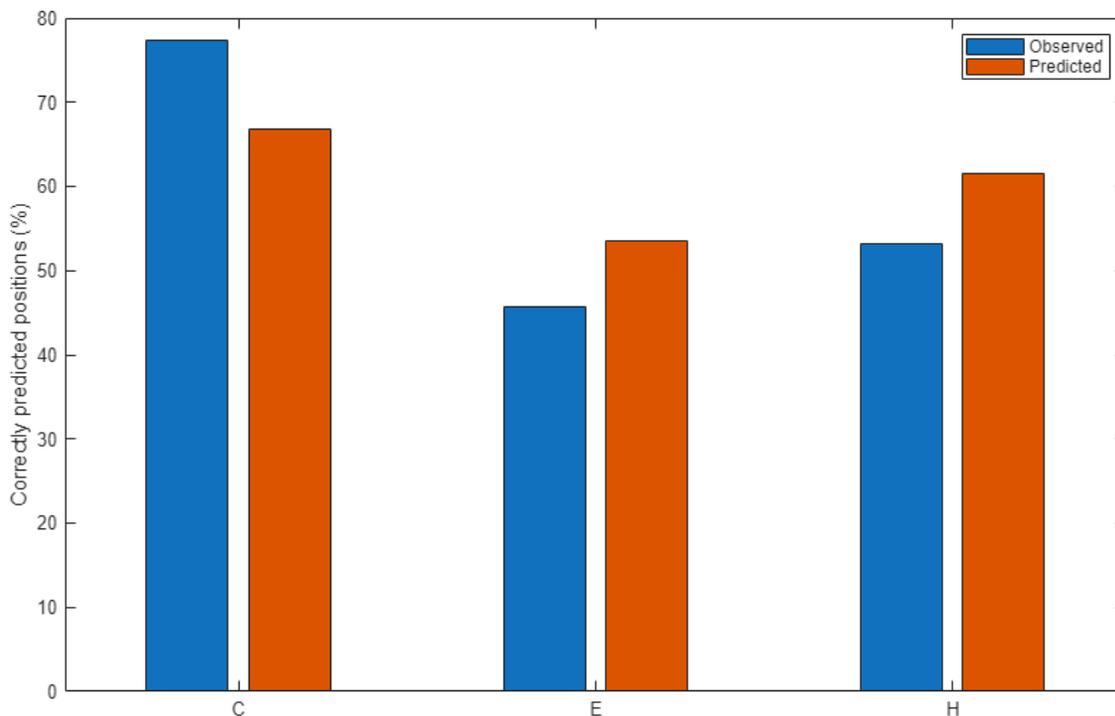
correctly predicted in state S, divided by the number of residues observed in state S. Similarly, we define the index $pcPred(S)$ for state S as the number of residues correctly predicted in state S, divided by the number of residues predicted in state S.

```
[i,j] = find(compet(0));
[u,v] = find(T);

% === compute fraction of correct predictions when a given state is observed
pcObs(1) = sum(i == 1 & u == 1)/sum (u == 1); % state C
pcObs(2) = sum(i == 2 & u == 2)/sum (u == 2); % state E
pcObs(3) = sum(i == 3 & u == 3)/sum (u == 3); % state H

% === compute fraction of correct predictions when a given state is predicted
pcPred(1) = sum(i == 1 & u == 1)/sum (i == 1); % state C
pcPred(2) = sum(i == 2 & u == 2)/sum (i == 2); % state E
pcPred(3) = sum(i == 3 & u == 3)/sum (i == 3); % state H

% === compare quality indices of prediction
figure()
bar([pcObs' pcPred'] * 100);
ylabel('Correctly predicted positions (%)');
ax = gca;
ax.XTickLabel = {'C';'E';'H'};
legend({'Observed', 'Predicted'});
```



These quality indices are useful for the interpretation of the prediction accuracy. In fact, in cases where the prediction technique tends to overpredict/underpredict a given state, a high/low prediction accuracy might just be an artifact and does not provide a measure of quality for the technique itself.

Conclusion

The method presented here predicts the structural state of a given protein residue based on the structural state of its neighbors. However, there are further constraints when predicting the content of structural elements in a protein, such as the minimum length of each structural element. Specifically, a helix is assigned to any group of four or more contiguous residues, and a sheet is assigned to any group of two or more contiguous residues. To incorporate this type of information, an additional network can be created so that the first network predicts the structural state from the amino acid sequence, and the second network predicts the structural element from the structural state.

References

- [1] Rost, B., and Sander, C., "Prediction of protein secondary structure at better than 70% accuracy", *Journal of Molecular Biology*, 232(2):584-99, 1993.
- [2] Holley, L.H. and Karplus, M., "Protein secondary structure prediction with a neural network", *PNAS*, 86(1):152-6, 1989.
- [3] Kabsch, W., and Sander, C., "How good are predictions of protein secondary structure?", *FEBS Letters*, 155(2):179-82, 1983.

Calculating and Visualizing Sequence Statistics

This example shows how to use basic sequence manipulation techniques and computes some useful sequence statistics. It also illustrates how to look for coding regions (such as proteins) and pursue further analysis of them.

The Human Mitochondrial Genome

In this example you will explore the DNA sequence of the human mitochondria. Mitochondria are structures, called organelles, that are found in the cytoplasm of the cell in hundreds to thousands for each cell. Mitochondria are generally the major energy production center in eukaryotes, they help to degrade fats and sugars.

The consensus sequence of the human mitochondria genome has accession number NC_012920. You can use the `getgenbank` function to get the latest annotated sequence from GenBank® into the MATLAB® workspace.

```
mitochondria_gbk = getgenbank('NC_012920');
```

For your convenience, previously downloaded sequence is included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load mitochondria
```

Copy just the DNA sequence to a new variable `mitochondria`. You can access parts of the DNA sequence by using regular MATLAB indexing commands.

```
mitochondria = mitochondria_gbk.Sequence;
mitochondria_length = length(mitochondria)
first_300_bases = seqdisp(mitochondria(1:300))
```

```
mitochondria_length =
```

```
16569
```

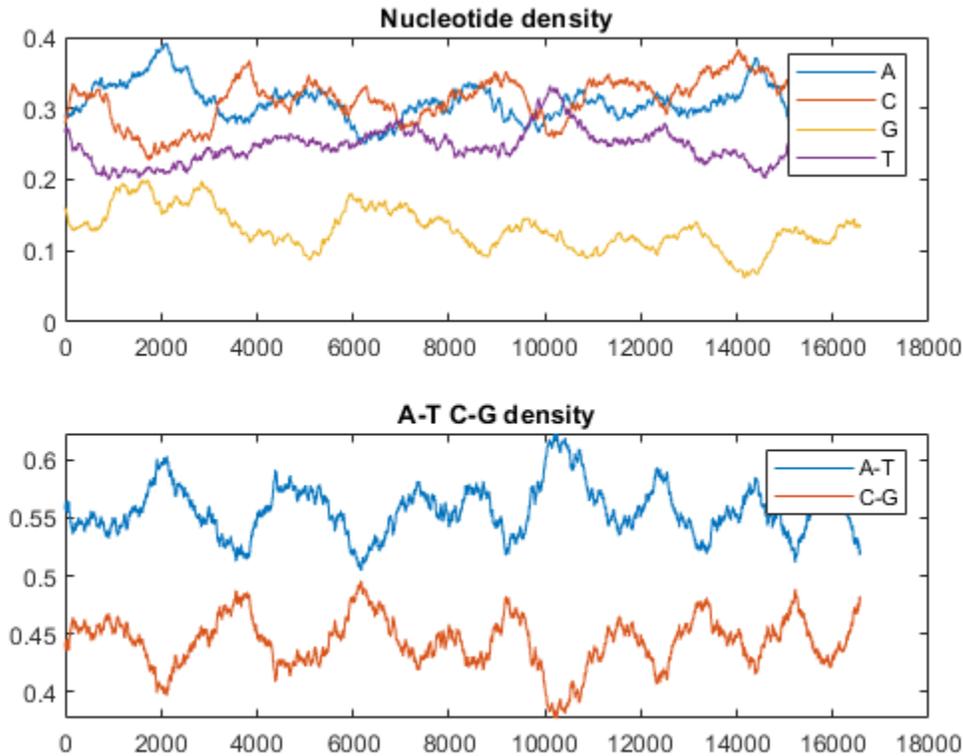
```
first_300_bases =
```

```
5×70 char array
```

```
' 1  GATCACAGGT  CTATCACCT  ATTAACCACT  CACGGGAGCT  CTCCATGCAT  TTGGTATTTT'
' 61  CGTCTGGGGG  GTATGCACGC  GATAGCATTG  CGAGACGCTG  GAGCCGGAGC  ACCCTATGTC'
'121  GCAGTATCTG  TCTTTGATTC  CTGCCTCATC  CTATTATTTA  TCGCACCTAC  GTTCAATATT'
'181  ACAGGCGAAC  AACTTACTA  AAGTGTGTTA  ATTAATTAAT  GCTTGTAGGA  CATAATAATA'
'241  ACAATTGAAT  GTCTGCACAG  CCACTTTCCA  CACAGACATC  ATAACAAAAA  ATTTCCACCA'
```

You can look at the composition of the nucleotides with the `ntdensity` function.

```
figure
ntdensity(mitochondria)
```



This shows that the mitochondria genome is A-T rich. The GC-content is sometimes used to classify organisms in taxonomy, it may vary between different species from ~30% up to ~70%. Measuring GC content is also useful for identifying genes and for estimating the annealing temperature of DNA sequence.

Calculating Sequence Statistics

Now, you will use some of the sequence statistics functions in the Bioinformatics Toolbox™ to look at various properties of the human mitochondrial genome. You can count the number of bases of the whole sequence using the `basecount` function.

```
bases = basecount(mitochondria)
```

```
bases =
```

```
struct with fields:
```

```
A: 5124
C: 5181
G: 2169
T: 4094
```

These are on the 5'-3' strand. You can look at the reverse complement case using the `seqrcomplement` function.

```
compBases = basecount(seqrcomplement(mitochondria))
```

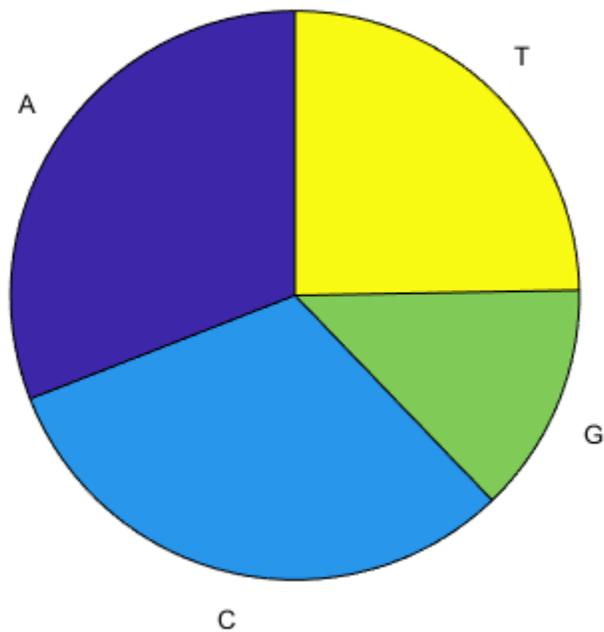
```
compBases =  
    struct with fields:  
        A: 4094  
        C: 2169  
        G: 5181  
        T: 5124
```

As expected, the base counts on the reverse complement strand are complementary to the counts on the 5'-3' strand.

You can use the chart option to basecount to display a pie chart of the distribution of the bases.

```
figure  
basecount(mitochondria,'chart','pie');  
title('Distribution of Nucleotide Bases for Human Mitochondrial Genome');
```

Distribution of Nucleotide Bases for Human Mitochondrial Genome



Now look at the dimers in the sequence and display the information in a bar chart using dimercount.

```
figure  
dimers = dimercount(mitochondria,'chart','bar')  
title('Mitochondrial Genome Dimer Histogram');
```

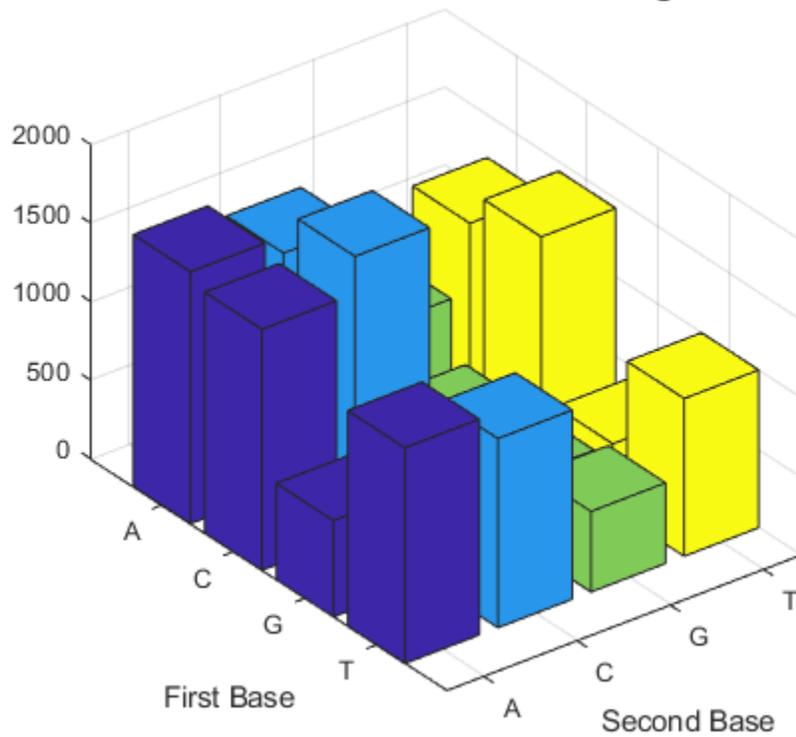
dimers =

struct with fields:

```

AA: 1604
AC: 1495
AG: 795
AT: 1230
CA: 1534
CC: 1771
CG: 435
CT: 1440
GA: 613
GC: 711
GG: 425
GT: 419
TA: 1373
TC: 1204
TG: 513
TT: 1004
    
```

Mitochondrial Genome Dimer Histogram



Exploring the Open Reading Frames (ORFs)

In a nucleotide sequence an obvious thing to look for is if there are any open reading frames. An ORF is any sequence of DNA or RNA that can be potentially translated into a protein. The function `seqshoworfs` can be used to visualize ORFs in a sequence.

Note: In the HTML tutorial only the first page of the output is shown, however when running the example you will be able to inspect the complete mitochondrial genome using the scrollbar on the figure.

```
seqshoworfs(mitochondria);
```

```

Frame 1

000001  gatcacagggtctatcacccctattaaccactcacgggagctctccatgcatttggtattttgcgc
000065  tgggggggtatgcacgcgatagcattgcgagacgctggagccggagcaccctatgtcgcagtatc
000129  tgtctttgatctcctgcctcctcctattatattatgcacctacgttcaatattacaggcgaacat
000193  acttactaaagtgtgttaattaattaatgcttgtaggacataataataacaattgaatgtctgc
000257  acagccactttccacacagacatcataacaaaaatttccaccaaaccctccccgcctc
000321  tggccacagcacttaaacacatctctgcccacccccaaaaacaaagaaccctaacaccagccta
000385  accagatttcaaattttatcttttggcggtatgcacttttaacagtcaccccccaactaacaca
000449  ttattttccccctcccactcccataactactaatctcatcaatacaacccccgcccactcctacca
000513  gcacacacacaccgctgctaaccatccccgaaccaaccaacccccaaagacacccccaca
000577  gtttatgtagcttacctcctcaagcaatacactgaaaatgttagacgggctcacatcacccc
000641  ataaacaaatagggttggctcctagcctttctattagctcttagtaagattacacatgcaagcat
000705  ccccggtccagtgagttcacctcctaaatcaccacgatcaaaaggaacaagcatcaagcacgca
000769  gcaatgcagctcaaaagccttagcctagccacacccccacgggaaacagcagtgattaacctt
000833  agcaataaacgaaagtttaactaagctataactaaccacaggggtgggtcaatttctgtgccagcca
000897  ccgcggtcacacgattaaccacagtcfaatagaagccggcgtaaagagtggttttagatcacccc
000961  tccccataaagctaaaactcacctgagttgtaaaaaactccagttgacacaaaatagactacg
001025  aaagtggctttaacatatactgaacacacaatagctaagacccaaactgggattagatacccac
001089  tatgcttagccctaaacctcaacagttaaatcaacaaaactgctcgcagaaactacgagcca
001153  cagcttaaaaactcaaggacctggcggtgcttcaatacctctagaggagcctgttctgtaatc
001217  gataaaccccgatcaacctcaccacctcttgcctcagcctatataccgccatcttcagcaaaccc
001281  tgatgaaggtacaaaagtaagcgaagtagccacgtaaagacgttaggtcaaggtgtagcccat
001345  gaggtggcaagaaatgggctacattttctaccccagaaaactacgatagcccttatgaaactta

```

If you compare this output to the genes shown on the NCBI page there seem to be slightly fewer ORFs, and hence fewer genes, than expected.

Vertebrate mitochondria do not use the Standard genetic code so some codons have different meaning in mitochondrial genomes. For more information about using different genetic codes in MATLAB see the help for the function `geneticcode`. The `GeneticCode` option to the `seqshoworfs` function allows you to look at the ORFs again but this time with the vertebrate mitochondrial genetic code.

Particularly, you can explore the annotated coding sequences (CDS) and compare them with the ORFs previously found. Use the `Sequence` option to the `featureparse` function to extract, when possible, the DNA sequences respective to each feature. The `featureparse` function will complement the pieces of the source sequence when appropriate.

```
features = featureparse(mitochondria_gbk, 'Sequence', true)
coding_sequences = features.CDS;
coding_sequences_id = sprintf('%s ', coding_sequences.gene)
```

```
features =
```

```
  struct with fields:
```

```
    source: [1×1 struct]
    D_loop: [1×1 struct]
    gene: [1×37 struct]
    tRNA: [1×22 struct]
    rRNA: [1×2 struct]
    STS: [1×28 struct]
    misc_feature: [1×1 struct]
    CDS: [1×13 struct]
```

```
coding_sequences_id =
```

```
  'ND1 ND2 COX1 COX2 ATP8 ATP6 COX3 ND3 ND4L ND4 ND5 ND6 CYTB '
```

```
ND2CDS = coding_sequences(2) % ND2 is in the 2nd position
COX1CDS = coding_sequences(3) % COX1 is in the 3rd position
```

```
ND2CDS =
```

```
  struct with fields:
```

```
    Location: '4470..5511'
    Indices: [4470 5511]
    gene: 'ND2'
    gene_synonym: 'MTND2'
    note: 'TAA stop codon is completed by the addition of 3' A residues to the mRNA'
    codon_start: '1'
    transl_except: '(pos:5511,aa:TERM)'
    transl_table: '2'
    product: 'NADH dehydrogenase subunit 2'
    protein_id: 'YP_003024027.1'
    db_xref: {'GI:251831108' 'GeneID:4536' 'HGNC:7456' 'MIM:516001'}
    translation: 'MNPLAQPVIIYSTIFAGTLITALSSHWFFTWVGLMMLAFIPVLTKKMNPRSTEAAIKYFLTQATASMILLMAILF
    Sequence: 'attaatcccctggccaaccgctcatctactctaccatctttgcaggcacactcatcacagcgctaagctgcact'
```

```
COX1CDS =
```

```
  struct with fields:
```

```
    Location: '5904..7445'
    Indices: [5904 7445]
```



```
ND2 = nt2aa(ND2CDS, 'GeneticCode', 'Vertebrate Mitochondrial');
disp(seqdisp(ND2))

  1  MNPLAQPVIY  STIFAGTLIT  ALSSHWFFTW  VGLEMNMLAF  IPVLTKKMNP  RSTEAAIKYF
 61  LTQATASMIL  LMAILFNNML  SGQWTMTNTT  NQYSSLMIMM  AMAMKLGMAP  FHFVWPEVTQ
121  GTPLTSGLLL  LTWQKLAPIS  IMYQISPSLN  VSLLLTLNIL  SIMAGSWGGL  NQTQLRKILA
181  YSSITHMGWM  MAVLPYNPNM  TILNLTIIYI  LTTTAFLLLN  LNSSTTTLLL  SRTWNKLTWL
241  TPLIPSTLLS  LGGLPPLTGF  LPKWAIIEEF  TKNNSLIIPT  IMATITLLNL  YFYLRLLIYST
301  SITLLPMSNN  VKMKWQFEHT  KPTPFLPTLI  ALTTLLLPIS  PFMLMIL
```

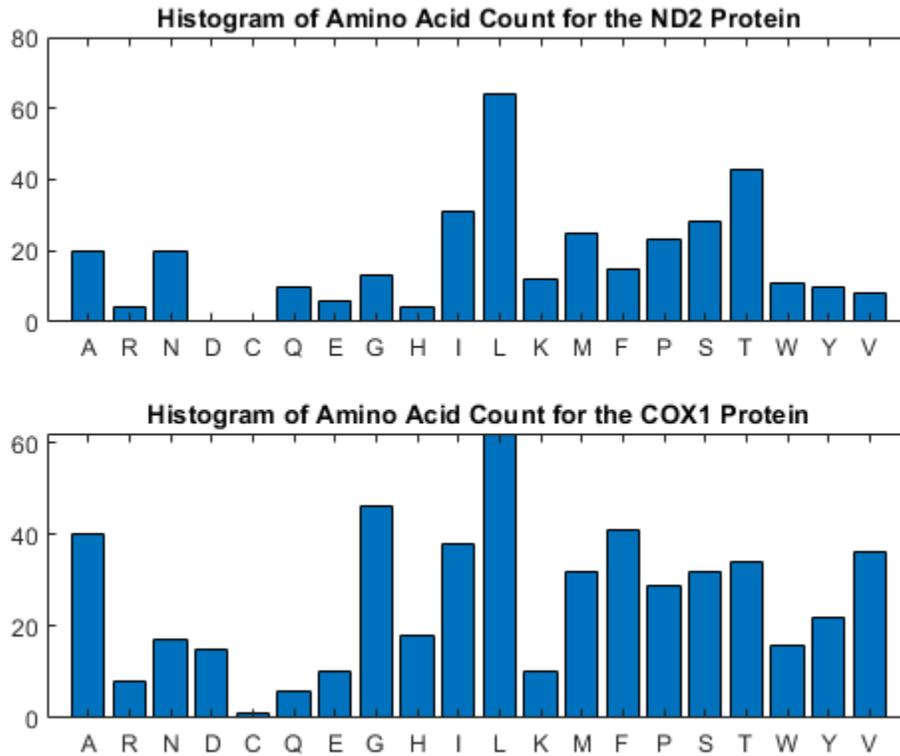
```
COX1 = nt2aa(COX1CDS, 'GeneticCode', 'Vertebrate Mitochondrial');
disp(seqdisp(COX1))
```

```
  1  MFADRWFST  NHKDIGTLYL  LFGAWAGVLG  TALSLLIRAE  LGQPGNLLGN  DHIYNVIVTA
 61  HAFVMIFFMV  MPIMIGGFNG  WLVPMLIGAP  DMAFPRMNNM  SFWLLPPSLL  LLLASAMVEA
121  GAGTGWTVYP  PLAGNYSHPG  ASVDLTIFSL  HLAGVSSILG  AINFITTIIN  MKPPAMTQYQ
181  TPLFVWSVLI  TAVLLLLSLP  VLAAGITMLL  TDRNLNTTFF  DPAGGGDPIL  YQHLFWFFGH
241  PEVYILILPG  FGMISHIVTY  YSGKKEPFGY  MGMVWAMMSI  GFLGFIVWAH  HMFTVGMDDV
301  TRAYFTSATM  IIAIPTGVKV  FSWLATLHGS  NMKWSAAVLW  ALGFIFLFTV  GGLTGIVLAN
361  SSLDIVLHDT  YYVVAHFHYV  LSMGAVFAIM  GGFHWFPLF  SGYTLDQTYA  KIHFTIMFIG
421  VNLTFFPQHF  LGLSGMPRRY  SDYPDAYTTW  NILSSVGSFI  SLTAVMLMIF  MIWEAFASKR
481  KVLMEEPSM  NLEWLYGCPP  PYHTFEPPVY  MKS*
```

You can get a more complete picture of the amino acid content with `aaccount`.

```
figure
subplot(2,1,1)
ND2aaCount = aaccount(ND2, 'chart', 'bar');
title('Histogram of Amino Acid Count for the ND2 Protein');

subplot(2,1,2)
COX1aaCount = aaccount(COX1, 'chart', 'bar');
title('Histogram of Amino Acid Count for the COX1 Protein');
```



Notice the high leucine, threonine and isoleucine content and also the lack of cysteine or aspartic acid.

You can use the `atomiccomp` and `molweight` functions to calculate more properties about the ND2 protein.

```
ND2AtomicComp = atomiccomp(ND2)
ND2MolWeight = molweight(ND2)
```

```
ND2AtomicComp =
```

```
struct with fields:
```

```
C: 1818
H: 2882
N: 420
O: 471
S: 25
```

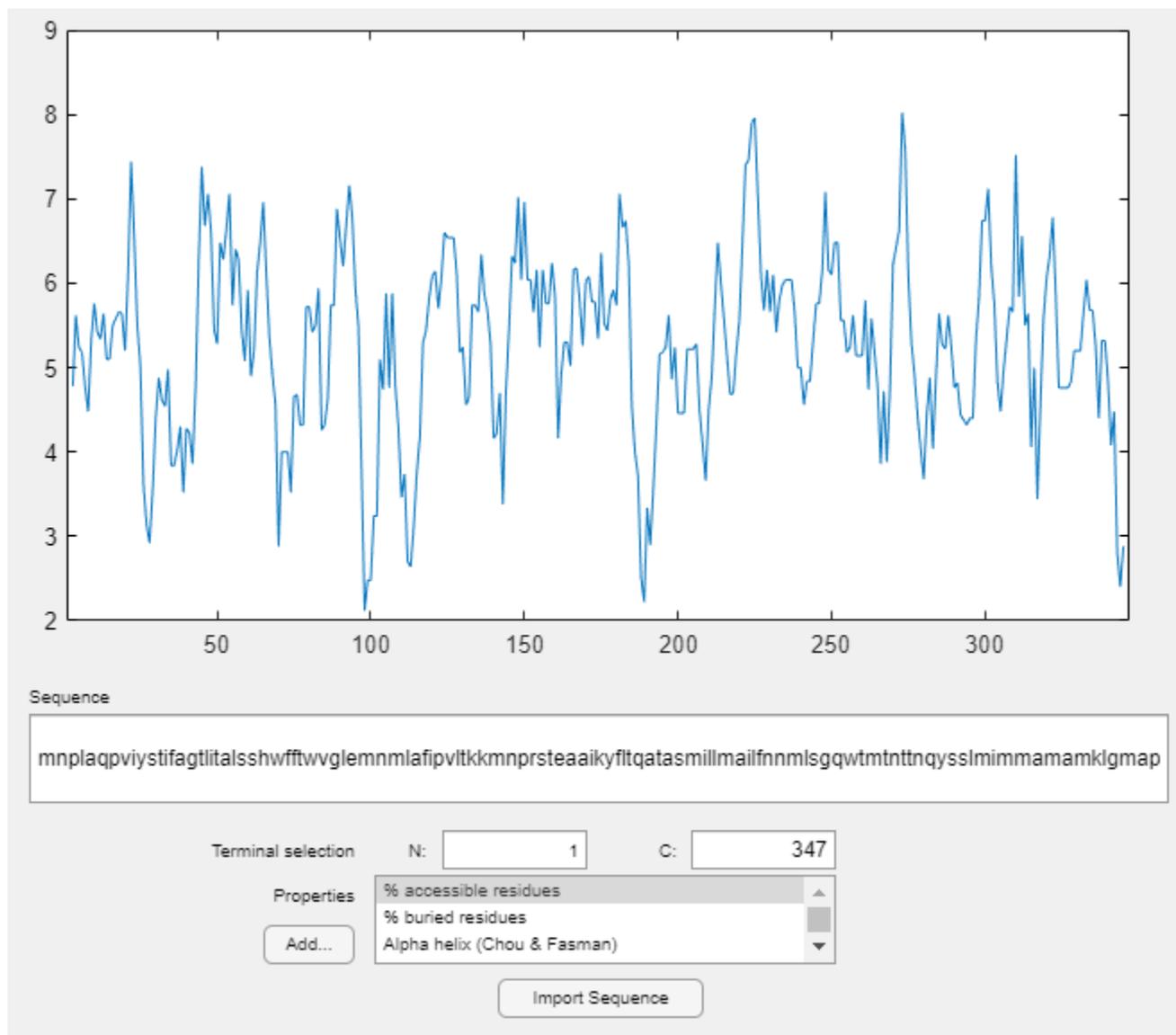
```
ND2MolWeight =
```

```
3.8960e+04
```

For further investigation of the properties of the ND2 protein, use `proteinplot`. This is a graphical user interface (GUI) that allows you to easily create plots of various properties, such as

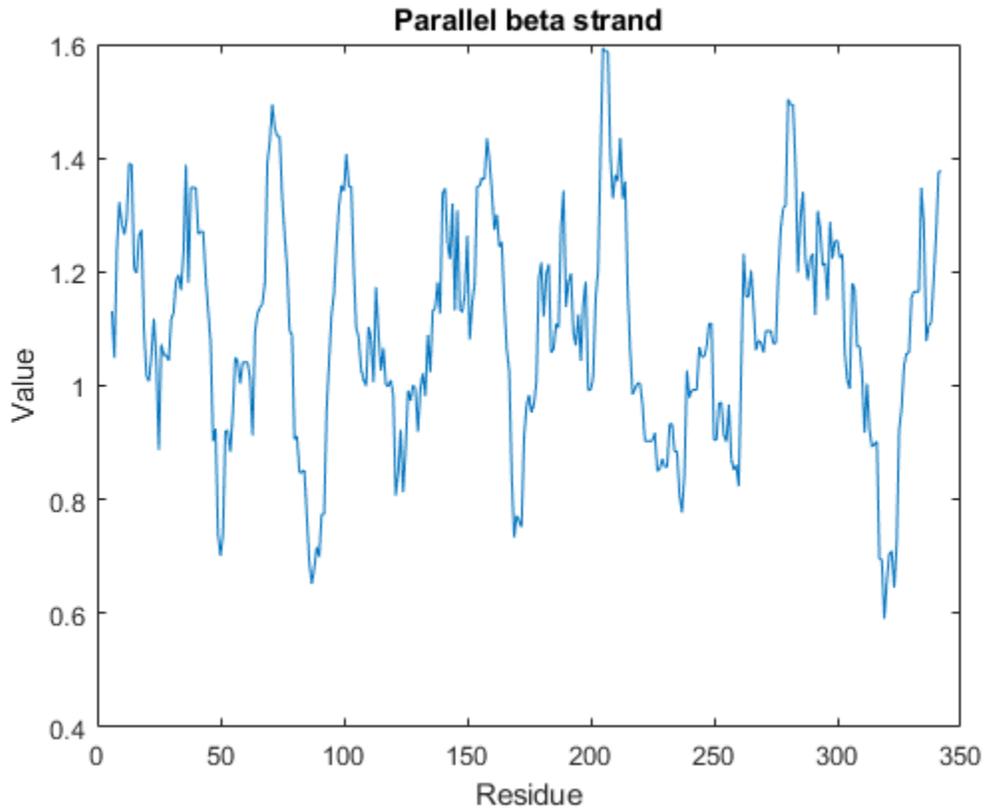
hydrophobicity, of a protein sequence. Click on the "Edit" menu to create new properties, to modify existing property values, or, to adjust the smoothing parameters. Click on the "Help" menu in the GUI for more information on how to use the tool.

```
proteinplot(ND2)
```



You can also programmatically create plots of various properties of the sequence using `proteinpropplot`.

```
figure
proteinpropplot(ND2, 'PropertyTitle', 'Parallel beta strand')
```



Calculating the Codon Frequency using all the Genes in the Human Mitochondrial Genome

The `codoncount` function counts the number of occurrences of each codon in the sequence and displays a formatted table of the result.

```
codoncount(ND2CDS)
```

AAA - 10	AAC - 14	AAG - 2	AAT - 6
ACA - 11	ACC - 24	ACG - 3	ACT - 5
AGA - 0	AGC - 4	AGG - 0	AGT - 1
ATA - 23	ATC - 24	ATG - 1	ATT - 8
CAA - 8	CAC - 3	CAG - 2	CAT - 1
CCA - 4	CCC - 12	CCG - 2	CCT - 5
CGA - 0	CGC - 3	CGG - 0	CGT - 1
CTA - 26	CTC - 18	CTG - 4	CTT - 7
GAA - 5	GAC - 0	GAG - 1	GAT - 0
GCA - 8	GCC - 7	GCG - 1	GCT - 4
GGA - 5	GGC - 7	GGG - 0	GGT - 1
GTA - 3	GTC - 2	GTG - 0	GTT - 3
TAA - 0	TAC - 8	TAG - 0	TAT - 2
TCA - 7	TCC - 11	TCG - 1	TCT - 4
TGA - 10	TGC - 0	TGG - 1	TGT - 0
TTA - 8	TTC - 7	TTG - 1	TTT - 8

Notice that in the ND2 gene there are more CTA, ATC and ACC codons than others. You can check what amino acids these codons get translated into using the `nt2aa` and `aminolookup` functions.

```
CTA_aa = aminolookup('code',nt2aa('CTA'))
ATC_aa = aminolookup('code',nt2aa('ATC'))
ACC_aa = aminolookup('code',nt2aa('ACC'))
```

```
CTA_aa =
    'Leu   Leucine
    '
```

```
ATC_aa =
    'Ile   Isoleucine
    '
```

```
ACC_aa =
    'Thr   Threonine
    '
```

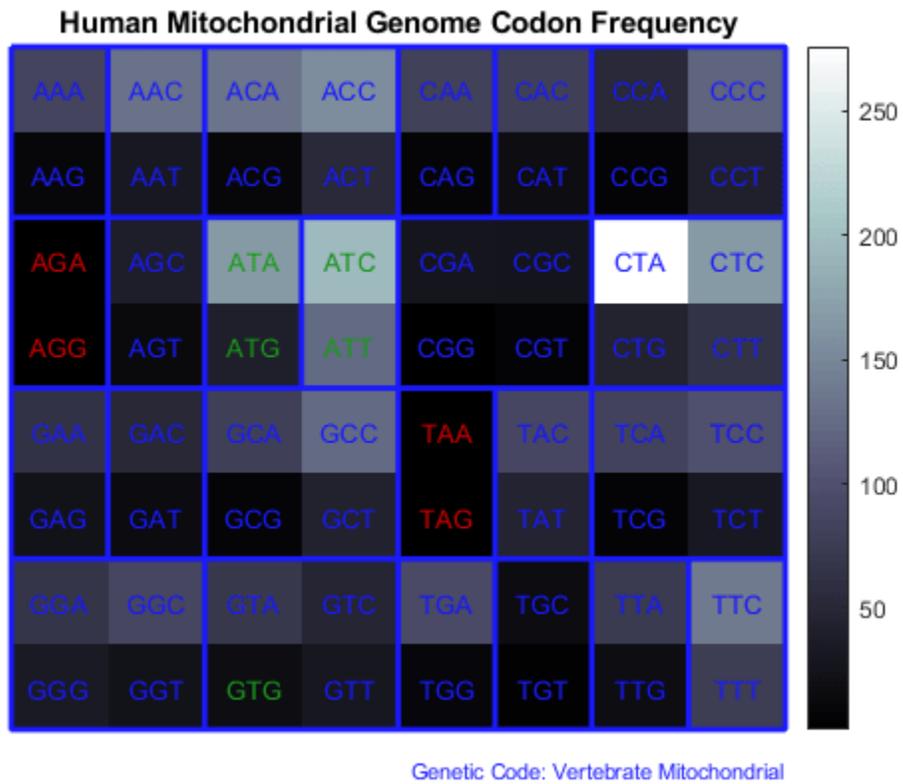
To calculate the codon frequency for all the genes you can concatenate them into a single sequence before using the function `codoncount`. You need to ensure that the codons are complete (three nucleotides each) so the read frame of the sequence is not lost at the concatenation.

```
numCDS = numel(coding_sequences);
CDS = cell(numCDS,1);
for i = 1:numCDS
    seq = coding_sequences(i).Sequence;
    CDS{i} = seq(1:3*floor(length(seq)/3));
end
allCDS = [CDS{:}];
codoncount(allCDS)
```

AAA - 85	AAC - 132	AAG - 10	AAT - 32
ACA - 134	ACC - 155	ACG - 10	ACT - 52
AGA - 1	AGC - 39	AGG - 1	AGT - 14
ATA - 167	ATC - 196	ATG - 40	ATT - 124
CAA - 82	CAC - 79	CAG - 8	CAT - 18
CCA - 52	CCC - 119	CCG - 7	CCT - 41
CGA - 28	CGC - 26	CGG - 2	CGT - 7
CTA - 276	CTC - 167	CTG - 45	CTT - 65
GAA - 64	GAC - 51	GAG - 24	GAT - 15
GCA - 80	GCC - 124	GCG - 8	GCT - 43
GGA - 67	GGC - 87	GGG - 34	GGT - 24
GTA - 70	GTC - 48	GTG - 18	GTT - 31
TAA - 3	TAC - 89	TAG - 2	TAT - 46
TCA - 83	TCC - 99	TCG - 7	TCT - 32
TGA - 93	TGC - 17	TGG - 11	TGT - 5
TTA - 73	TTC - 139	TTG - 18	TTT - 77

Use the `figure` option to the `codoncount` function to show a heat map with the codon frequency. Use the `geneticcode` option to overlay a grid on the figure that groups the synonymous codons according with the Vertebrate Mitochondrial genetic code. Observe the particular bias of Leucine (codons 'CTN').

```
figure
count = codoncount(allCDS, 'figure', true, 'geneticcode', 'Vertebrate Mitochondrial');
title('Human Mitochondrial Genome Codon Frequency')
```



```
close all
```

References

[1] Barrell, B.G., Bankier, A.T. and Drouin, J., "A different genetic code in human mitochondria", *Nature*, 282(5735):189-94, 1979.

Aligning Pairs of Sequences

This example shows how to extract some sequences from GenBank® and align the sequences using global and local alignment algorithms.

Accessing NCBI Data from the MATLAB Workspace

One of the many fascinating sections of the NCBI web site is the Genes and diseases section. This section provides a comprehensive introduction to medical genetics.

In this example you will be looking at genes associated with Tay-Sachs Disease. Tay-Sachs is an autosomal recessive disease caused by mutations in both alleles of a gene (HEXA, which codes for the alpha subunit of hexosaminidase A) on chromosome 15.

The NCBI reference sequence for HEXA has accession number NM_000520. You can use the `getgenbank` function to retrieve the sequence information from the NCBI data repository and load it into MATLAB®.

```
humanHEXA = getgenbank('NM_000520');
```

By doing a BLAST search or by searching in the mouse genome you can find an orthogonal gene, AK080777 is the accession number for a mouse hexosaminidase A gene.

```
mouseHEXA = getgenbank('AK080777');
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load('hexosaminidase.mat', 'humanHEXA', 'mouseHEXA')
```

Aligning the Sequences

The first step is to use global sequence alignment to look for similarities between these sequences. You could look at the alignment between the nucleotide sequences, but it is generally more instructive to look at the alignment between the protein sequences, in this example we know that the sequences are coding sequences. Use the `nt2aa` function to convert the nucleotide sequences into the corresponding amino acid sequences.

```
humanProtein = nt2aa(humanHEXA.Sequence);  
mouseProtein = nt2aa(mouseHEXA.Sequence);
```

One of the easiest ways to look for similarity between sequences is with a dot plot.

```
warnState = warning;  
warning('Off', 'bioinfo:seqdotplot:imageTooBigForScreen');  
seqdotplot(mouseProtein, humanProtein)  
xlabel('Human hexosaminidase A'); ylabel('Mouse hexosaminidase A');
```


Refining with Semi-global Alignment

The alignment is very good except for the terminal segments. For instance, notice the sparse matched pairs in the first positions. This occurs because a global alignment attempts to force the matching all the way to the ends and there is point where the penalty for opening new gaps is comparable to the score of matching residues. In some cases it is desirable to remove the gap penalty added at the ends of a global alignment; this allows you to better match this pair of sequences. This technique is commonly known as 'semi-global' alignment or 'glocal' alignment.

```
[score, globalAlignment] = nwalignment(humanProtein,mouseProtein,'glocal',true)
```

```
score =
1.0413e+03
```

```
globalAlignment = 3×825 char array
      'SCRRPAQSAARSRLRSRPEVKGQGVGPPGVAGAEPPLVT*FADKSRGRRSPDQGLTWPAPSERGDQR-AMTSSRLWFSLLLAAAFAGRAT
      '
      '
      '-----AAGRGAGRWAMAGCRLWVSLLLAAALACLAT
```

Refining the Alignment by Extracting the Protein Sequence

Another way to refine your alignment is by using only the protein sequences. Notice that the aligned region is delimited by start (M-methionine) and stop (*) amino acids in the sequences. If the sequence is shortened so that only the translated regions are considered, then it seems likely that you will get a better alignment. Use the `find` command to look for the index of the start amino acid in each sequence:

```
humanStart = find(humanProtein == 'M',1)
```

```
humanStart =
70
```

```
mouseStart = find(mouseProtein == 'M',1)
```

```
mouseStart =
11
```

Similarly, use the `find` command to look for the index of the first stop occurring after the start of the translation. Special care needs to be taken because there is also a stop at the very beginning of the `humanProtein` sequence.

```
humanStop = find(humanProtein(humanStart:end)=='*',1) + humanStart - 1
```

```
humanStop =
599
```

```
mouseStop = find(mouseProtein(mouseStart:end)=='*',1) + mouseStart - 1
```

```
mouseStop =
539
```

Use these indices to truncate the sequences.

```
humanSeq = humanProtein(humanStart:humanStop);
humanSeqFormatted = seqdisp(humanSeq)
```

```
humanSeqFormatted = 9×70 char array
      ' 1 MTSSRLWFSL LAAAFAGRA TALWPWPQNF QTSDQRYVLY PNNFQFYDV SSAAQPGCSV'
```



```
warning(warnState);  
close all;
```

Assessing the Significance of an Alignment

This example shows a method that can be used to investigate the significance of sequence alignments. The number of identities or positives in an alignment is not a clear indicator of a significant alignment. A permutation of a sequence from an alignment will have similar percentages of positives and identities when aligned against the original sequence. The score from an alignment is a better indicator of the significance of an alignment. This example uses the same Tay-Sachs disease related genes and proteins analyzed in “Aligning Pairs of Sequences” on page 3-138.

Accessing NCBI Data from the MATLAB® Workspace

In this example, you will work directly with protein data so use `getgenpept` instead of `getgenbank` to download the data from the NCBI site. First read the human protein information into MATLAB®.

```
humanProtein = getgenpept('NP_000511');
```

Results from a BLASTX search performed with this sequence showed that a *Drosophila* protein, GenPept accession number *AAM29423*, has some similarity to the human *HEXA* sequence. Use `getgenpept` to download this sequence.

```
flyProtein = getgenpept('AAM29423');
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load('flyandhumanproteins.mat','humanProtein','flyProtein')
seqdisp(humanProtein)
seqdisp(flyProtein)
```

```
ans =
```

```
10×70 char array
```

```
'>gi|189181666|gb|NP_000511.2| beta-hexosaminidase subunit alpha pre...'
' 1 MTSSRLWFSL LLAÁAFAGRA TALWPWPQNF QTSDQRYVLY PNNFQFQYDV SSAAQPGCSV'
' 61 LDEAFQRYRD LLFGSGSWPR PYLTGKRHTL EKNVLVSVV TPGCNQLPTL ESVENYTLTI'
'121 NDDQCLLLSE TVWGALRGL E TFSQLVWKA EGTFFFINKTE IEDFPRFPHR GLLLDTSRHY'
'181 LPLSSILDTL DVMAYNKLNV FHWHLVDDPS FPYESFTFPE LMRKGSYNPV THIYTAQDVK'
'241 EVIEYARLRG IRVLAEFDTP GHTLSWGPGI PGLLTPCYSG SEPSGTFGPV NPSLNNTYEF'
'301 MSTFFLEVSS VFPDFYLHLG GDEVDFTCWK SNPEIQDFMR KKGFGEDFKQ LESFYIQTLL'
'361 DIVSSYGKGY VVWQEVFDNK VKIQPDTIIQ VWREDIPVNY MKELELVTKA GFRALLSAPW'
'421 YLNRIISYGPD WKDFYIVEPL AFEGTPEQKA LVIGGEACMW GEYVDNTNLV PRLWPRAGAV'
'481 AERLWSNKLT SDLTFAYERL SHFRCELLRR GVQAQPLNVG FCEQEFEQT'
```

```
ans =
```

```
12×70 char array
```

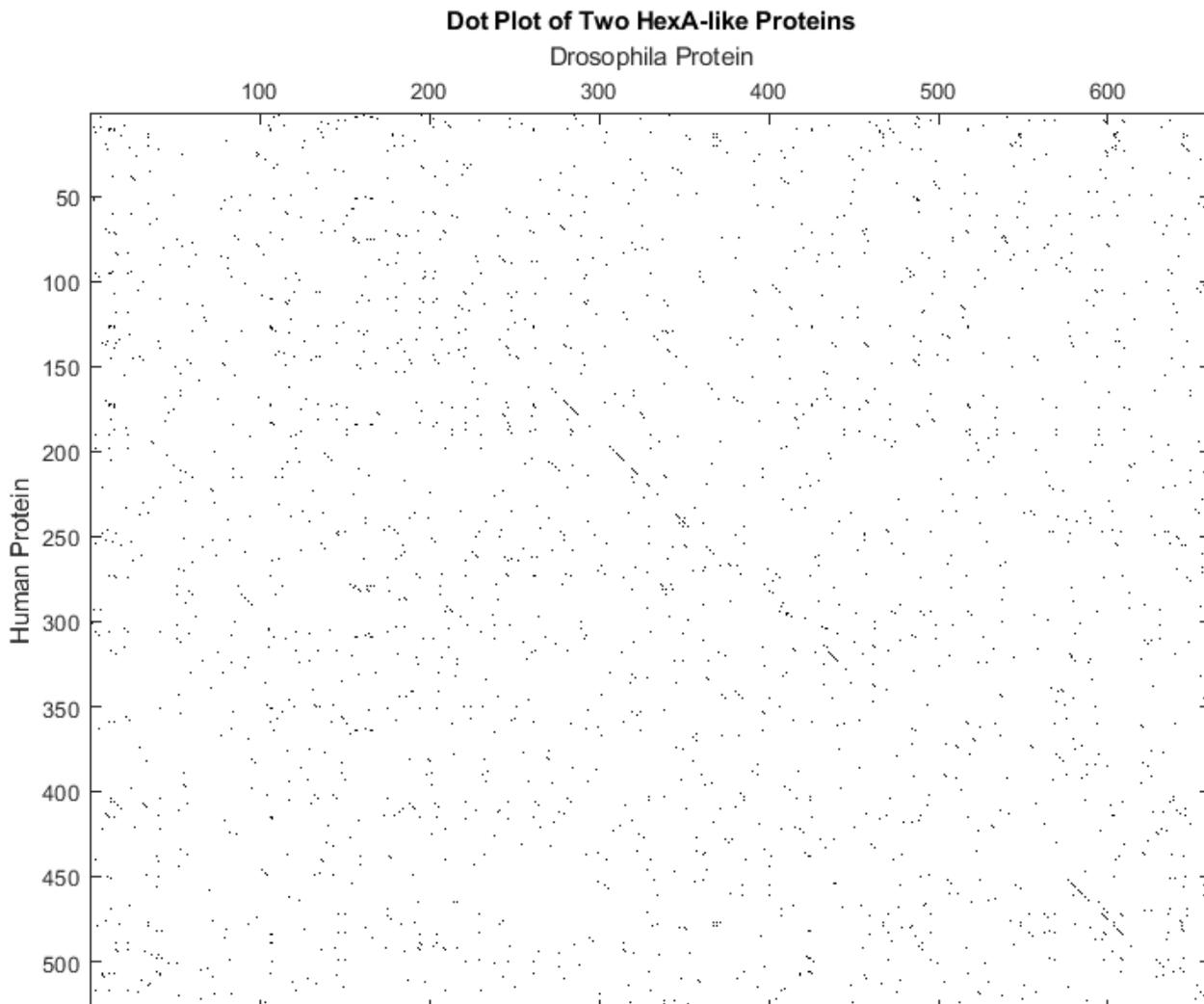
```
'>gi|21064387|gb|AAM29423.1| RE17456p [Drosophila melanogaster].
' 1 MSLAVSLRRA LLVLLTGAI F ILTVLYWNQG VTKAQAYNEA LERPHSHDA SGFPIPVEKS'
' 61 WTYKCENDRC MRVGHGKSA KRVSFISCSM TCGDVNIWPH PTQKFLSSQ THSFSVEDVQ'
'121 LHVDTAHREV RKQLQLAFDW FLKDLRLIQR LDYGGSSSEP TVSESSSKSR HHADLEPAAT'
'181 LFGATFGVKK AGDLTSVQVK ISVLKSGDLN FSLDNDETYQ LSTQTEGHR L QVEIIANSYF'
```

```
'241 GARHGLSTLQ QLIWFDDH LLHTYANSKV KDAPKFRYRG LMLDTSRHFF SVESIKRTIV'
'301 GMGLAKMNRH HWHLTDAQSF PYISRYYPEL AVHGAYSESE TYSEQDVREV AEFKIYGVQ'
'361 VIPEIDAPAH AGNGWDWGPK RGMGELAMCI NQPWFSFYCG EPPCGQLNPK NNYTYLILQR'
'421 IYEELLQHTG PTDFHLLGGD EVNLDCWAQY FNDDTLRGLW CDFMLQAMAR LKLANNGVAP'
'481 KHVAVWSSAL TNTKRLPNSQ FTVQVWGGST WQENYDLLDN GYNVIFSHVD AWYLDGFGS'
'541 WRATGDAACA QYRTWQNVYK HRPWERMRLD KKRKKQVLGG EVCMWTEQVD ENQLDNRLWP'
'601 RTAALAERLW TDPSDDHMDM IVPDVFRII SLFRNRLVEL GIRAEALFPK YCAQNPGECI'
```

A First Comparison and Global Alignment

The first thing to do is to use `seqdotplot` to see if there are any areas that are clearly aligned. This doesn't show any obvious alignments, but there are some areas of interest.

```
seqdotplot(humanProtein, flyProtein, 3, 2)
title('Dot Plot of Two HexA-like Proteins');
ylabel('Human Protein'); xlabel('Drosophila Protein');
```

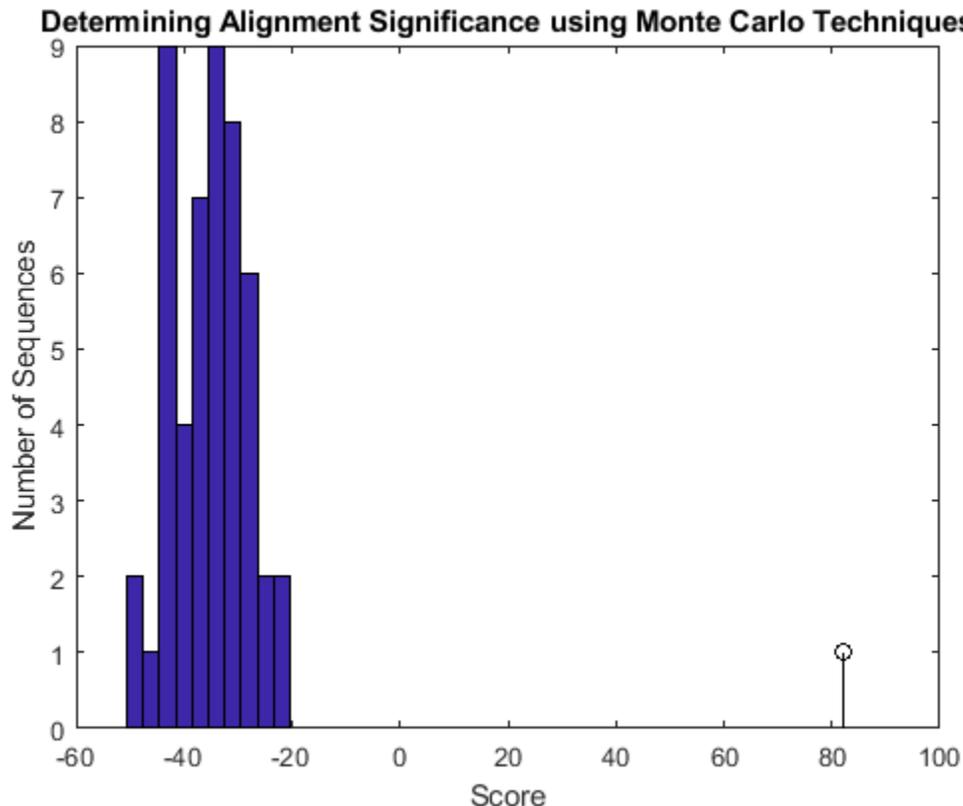


Initialize the state of the default random number generators to ensure that the figures and results generated match the ones in the HTML version of this example.

```
rng(0, 'twister')
n = 50;
globalscores = zeros(n,1);
flyLen = length(flyProtein.Sequence);
for i = 1:n
    perm = randperm(flyLen);
    permutedSequence = flyProtein.Sequence(perm);
    globalscores(i) = nwalign(humanProtein,permutedSequence, 'scoringmatrix', 'blosum30');
end
```

Now plot the scores as a bar chart. Note that because you are using randomly generated sequences.

```
figure
buckets = ceil(n/5);
hist(globalscores,buckets)
hold on;
stem(sc30,1,'k')
title('Determining Alignment Significance using Monte Carlo Techniques');
xlabel('Score'); ylabel('Number of Sequences');
```



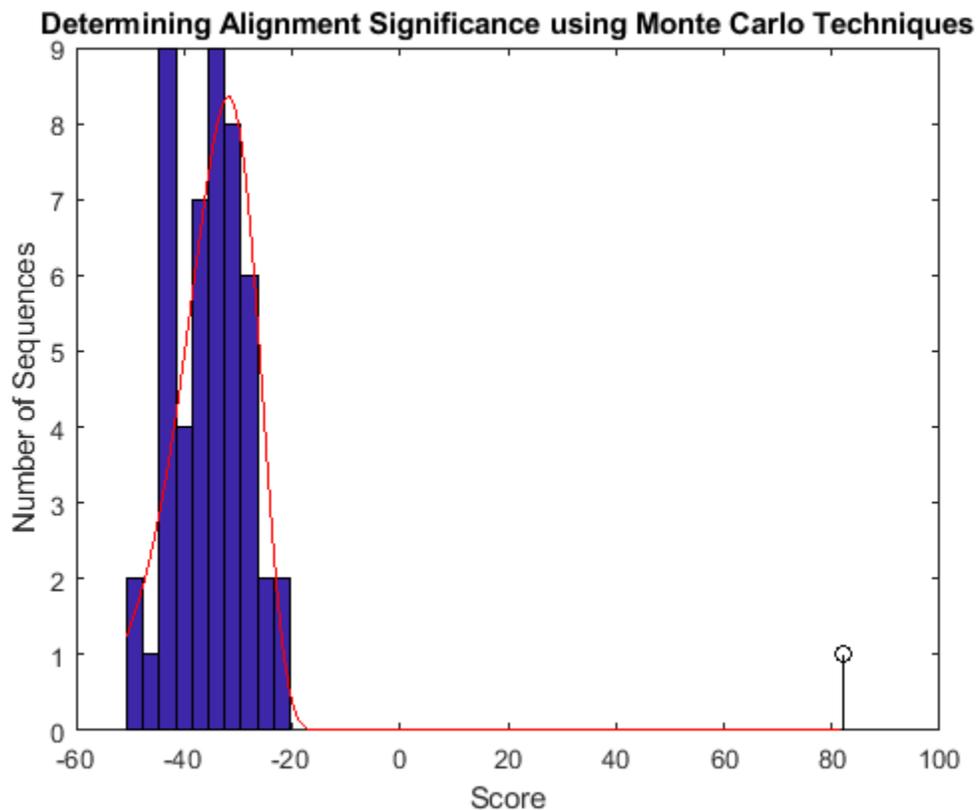
The scores of the alignments to the random sequences can be approximated by the type 1 extreme value distribution. Use the `evfit` function from the Statistics and Machine Learning Toolbox™ to estimate the parameters of this distribution.

```
parmhat = evfit(globalscores)
```

```
parmhat =
    -31.7597    6.6440
```

Overlay a plot of the probability density function of the estimated distribution.

```
x = min(globalscores):max([globalscores;sc30]);
y = evpdf(x,parmhat(1),parmhat(2));
[v, c] = hist(globalscores,buckets);
binWidth = c(2) - c(1);
scaleFactor = n*binWidth;
plot(x,scaleFactor*y,'r');
hold off;
```



From this plot you can see that the global alignment (globAlig30) is clearly statistically significant.

An Example Where the Score is Not Statistically Significant

In FLYBASE web site you can search for all *Drosophila* beta-N-acetylhexosaminidase genes. The gene that you have been looking at so far is referenced as *CG8824*. Now you want to take a look at another similar gene, for instance *Hexo1*.

```
flyHexo1 = getgenpept('AAL28566');
```

The fly *Hexo1* aminoacid sequence is also provided in the MAT-file `flyandhumanproteins.mat`.

```
load('flyandhumanproteins.mat','flyHexo1')
seqdisp(humanProtein)
```

```
ans =
```

```
10x70 char array
```

```
'>gi|189181666|gb|NP_000511.2| beta-hexosaminidase subunit alpha pre...
' 1 MTSSRLWFSL LLAAAFAGRA TALWPWPQNF QTSRQRYVLY PNNFQFQYDV SSAAQPGCSV'
' 61 LDEAFQRYRD LLFGSGSWPR PYLTGKRHTL EKNVLVVSIV TPGCNQLPTL ESVENYTLTI'
'121 NDDQCLLLSE TVWGALRGL E TFSQLVWKSA EGTFFINKTE IEDFPRFPHR GLLLDTSRHY'
'181 LPLSSILDTL DVMAYNKLNV FHWHLVDDPS FPYESFTFPE LMRKGSYNPV THIYTAQDVK'
'241 EVIEYARLRG IRVLAEFDTP GHTLSWGPPI PGLLTPCYSG SEPSGTFGPV NPSLNNTYEF'
'301 MSTFFLEVSS VFPDFYLHLG GDEVDFTCWK SNPEIQDFMR KKGFGEDFKQ LESFYIQTLL'
'361 DIVSSYGKGY VVWQEVFDNK VKIQPDTIIQ VWREDIPVNY MKELELVTKA GFRALLSAPW'
'421 YLNRISYGPD WKDFYIVEPL AFEGTPEQKA LVIGGEACMW GEYVDNTNLV PRLWPRAGAV'
'481 AERLWSNKLT SDLTFAYERL SHFRCELLRR GVQAQPLNVG FCEQEFEQT'
```

Repeat the process of generating a global alignment and then using random permutations of the amino acids to estimate the significance of the global alignment.

```
[Hexo1score,Hexo1Alignment] = nwalgn(humanProtein,flyHexo1,'scoringmatrix','blosum30')
fprintf('Score = %g \n',Hexo1score)
Hexo1globalscores = zeros(n,1);
flyLen = length(flyHexo1.Sequence);
for i = 1:n
    perm = randperm(flyLen);
    permutedSequence = flyHexo1.Sequence(perm);
    Hexo1globalscores(i) = nwalgn(humanProtein,permutedSequence,'scoringmatrix','blosum30');
end
```

```
Hexo1score =
```

```
-72.2000
```

```
Hexo1Alignment =
```

```
3x534 char array
```

```
'MTSSRL-WFSLLLAAFA-GRATALWPWPQNFQTSRQRYVLYPNNFQFQYDVSSAAQPGCSVLDEAFQRYRDLLFGSGSWPRPYLTGKRHTL
'|: :| | :::: : :::: :| :::: : :|| | | : : : ||:::~: | | : : |:: | : : | : : | :|
'MALVKLNTFHHITDSHSFPLEVKKRPELHKLGAYSQRQV-Y--T-R-R-DVAEVVEYG-RV--RGI-RVMP-EF-D-A-PAHVGEQWQH-
```

```
Score = -72.2
```

Plot the scores, calculate the parameters of the distribution and overlay the PDF on the bar chart.

```
figure
buckets = ceil(n/5);
hist(Hexo1globalscores,buckets)
title('Determining Alignment Significance using Monte Carlo Techniques');
xlabel('Score');
ylabel('Number of Sequences');
hold on;
```

```

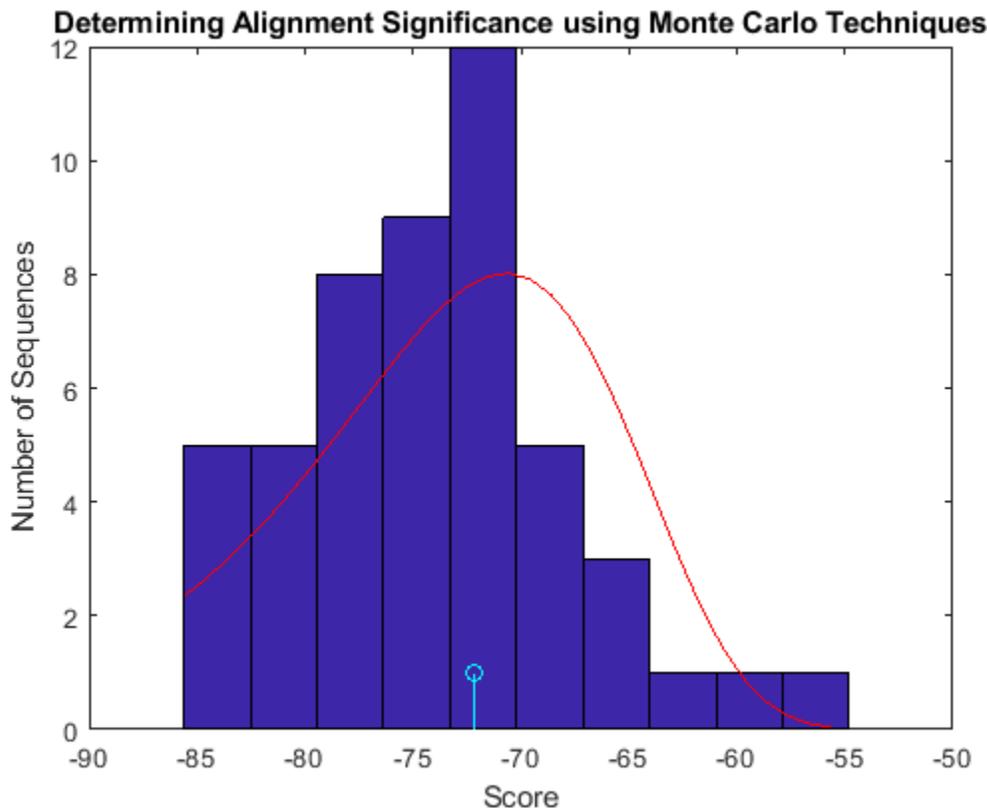
stem(Hexolscore,1,'c')
parmhat = evfit(Hexolglobalscores)
x = min(Hexolglobalscores):max([Hexolglobalscores;Hexolscore]);
y = evpdf(x,parmhat(1),parmhat(2));
[v, c] = hist(Hexolglobalscores,buckets);
binWidth = c(2) - c(1);
scaleFactor = n*binWidth;
plot(x,scaleFactor*y,'r');
hold off;

```

```

parmhat =
    -70.6926    7.0619

```



In this case it appears that the alignment is not statistically significant. Higher scoring alignments can easily be generated from a random permutation of the amino acids in the sequence. You can calculate an approximate p-value from the estimated extreme value CDF: However, far more than 50 random permutations are needed to get a reliable estimate of the extreme value pdf parameters from which to calculate a reasonably accurate p-value.

```
p = 1 - evcdf(Hexolscore,parmhat(1),parmhat(2))
```

```
p =
```



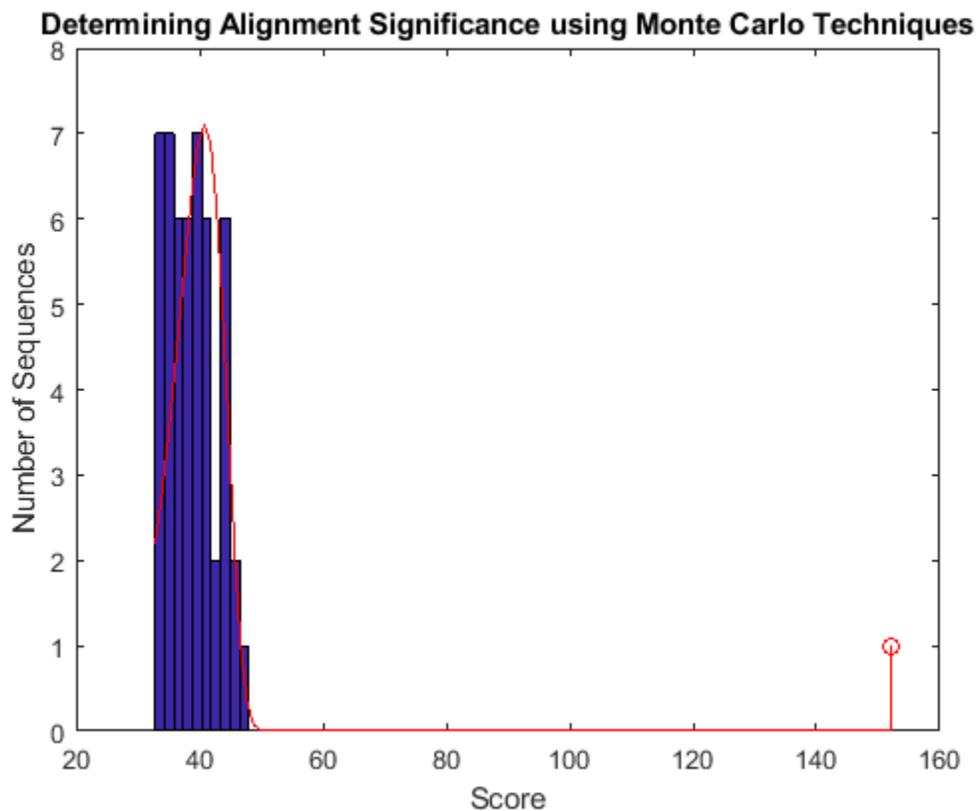
```

y = evpdf(x,parmhat(1),parmhat(2));
[v, c] = hist(localscores,buckets);
binWidth = c(2) - c(1);
scaleFactor = n*binWidth;
plot(x,scaleFactor*y,'r');
hold off;

```

```
parmhat =
```

```
40.8331 3.9312
```



You might like to experiment to see if there are significant differences in the distribution of scores generated with `randperm` and `randseq`.

With the local alignment it appears that the alignment is statistically significant. In fact, looking at the local alignment shows a very good alignment for the full length of the *Hexo1* sequence.

```
close all;
```

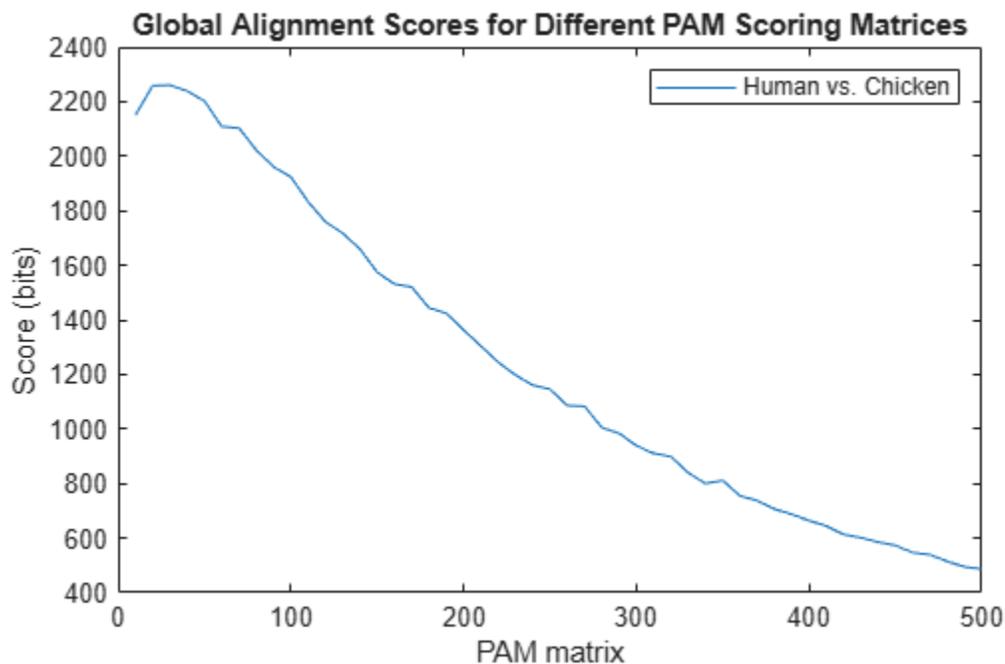

Trying different PAM matrices

```
for step = 1:50
    fprintf('.')
    PamNumber = step * 10;
    [matrix,info] = pam(PamNumber);
    score(step) = nwalgn(human,chicken,'scoringmatrix',matrix,'scale',info.Scale);
end
```

Plotting the Scores

You can use the plot function to create a graph of the results.

```
x = 10:10:500;
plot(x,score)
legend('Human vs. Chicken');
title('Global Alignment Scores for Different PAM Scoring Matrices');
xlabel('PAM matrix');ylabel('Score (bits)');
```



Finding the Best Score

You can use `max` with two outputs to find the highest score and the index in the results vector where the highest value occurred. In this case the highest score occurred with the third matrix, that is PAM30.

```
[bestScore, idx] = max(score)
```

```
bestScore =
2.2605e+03
```

```
idx =
3
```

Aligning to Other Organisms

Repeat this with different organisms: xenopus and rainbow trout.

```
xenopusScore = zeros(1,50);
troutScore = zeros(1,50);
fprintf('Trying different PAM matrices ')

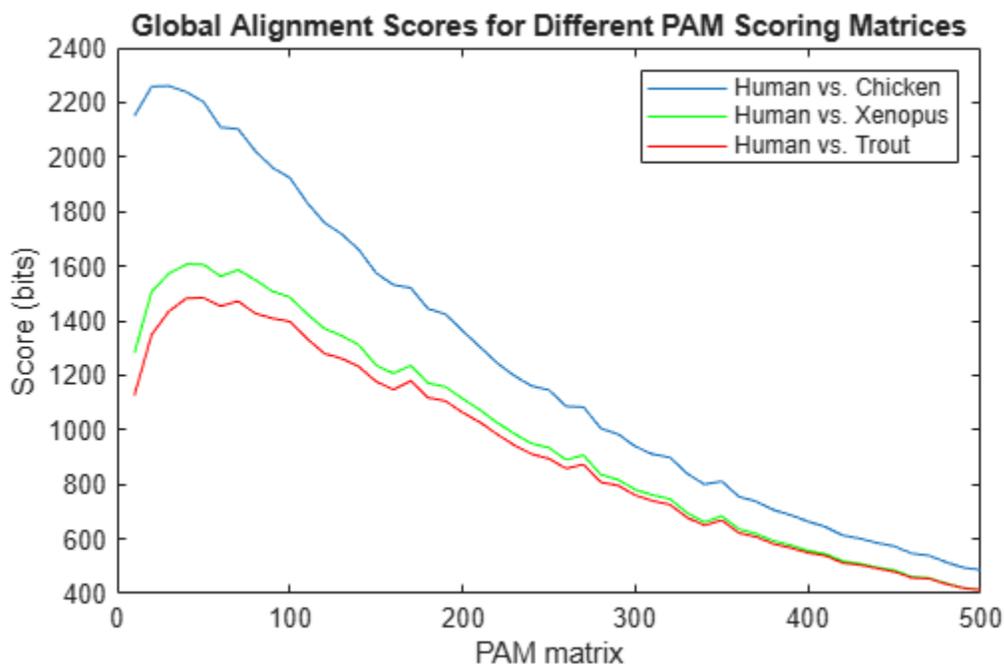
Trying different PAM matrices

for step = 1:50
    fprintf('.')
    PamNumber = step * 10;
    [matrix,info] = pam(PamNumber);
    xenopusScore(step) = nwalign(human,xenopus,'scoringmatrix',matrix,'scale',info.Scale);
    troutScore(step) = nwalign(human,trout,'scoringmatrix',matrix,'scale',info.Scale);
end
```

Adding More Lines to the Same Plot

You can use the command `hold on` to tell MATLAB® to add new plots to the existing figure. Once you have finished doing this you must remember to disable this feature by using `hold off`.

```
hold on
plot(x,xenopusScore,'g')
plot(x,troutScore,'r')
legend({'Human vs. Chicken','Human vs. Xenopus','Human vs. Trout'});box on
title('Global Alignment Scores for Different PAM Scoring Matrices');
xlabel('PAM matrix');ylabel('Score (bits)');
hold off
```



Finding the Best Scores

You will see that different matrices give the highest scores for the different organisms. For human and xenopus, the best score is with PAM40 and for human and trout the best score is PAM50.

```
[bestXScore, Xidx] = max(xenopusScore)
```

```
bestXScore =  
1607
```

```
Xidx =  
4
```

```
[bestTScore, Tidx] = max(troutScore)
```

```
bestTScore =  
1484
```

```
Tidx =  
5
```

The PAM scoring matrix giving the best alignment for two sequences is an indicator of the relative evolutionary interval since the organisms diverged: The smaller the PAM number, the more closely related the organisms. Since organisms, and protein families across organisms, evolve at widely varying rates, there is no simple correlation between PAM distance and evolutionary time. However, for an analysis of a specific protein family across multiple species, the corresponding PAM matrices will provide a relative evolutionary distance between the species and allow accurate phylogenetic mapping. In this example, the results indicate that the human sequence is more closely related to the chicken sequence than to the frog sequence, which in turn is more closely related than the trout sequence.

Calling Bioperl Functions from MATLAB

This example shows the interoperability between MATLAB® and Bioperl - passing arguments from MATLAB to Perl scripts and pulling BLAST search data back to MATLAB.

NOTE: Perl and the Bioperl modules must be installed to run the Perl scripts in this example. Since version 1.4, Bioperl modules have a warnings.pm dependency requiring at least version 5.6 of Perl. If you have difficulty running the Perl scripts, make sure your PERL5LIB environment variable includes the path to your Bioperl installation or try running from the Bioperl installation directory. See the links at <https://www.perl.com> and <https://bioperl.org/> for current release files and complete installation instructions.

Introduction

Gleevec™ (STI571 or imatinib mesylate) was the first approved drug to specifically turn off the signal of a known cancer-causing protein. Initially approved to treat chronic myelogenous leukemia (CML), it is also effective for treatment of gastrointestinal stromal tumors (GIST).

Research has identified several gene targets for Gleevec including: Proto-oncogene tyrosine-protein kinase ABL1 (NP_009297), Proto-oncogene tyrosine-protein kinase Kit (NP_000213), and Platelet-derived growth factor receptor alpha precursor (NP_006197).

```
target_ABL1 = 'NP_009297';
target_Kit = 'NP_000213';
target_PDGFRA = 'NP_006197';
```

Accessing Sequence Information

You can load the sequence information for these proteins from local GenPept text files using **genpeptread**.

```
ABL1_seq = getfield(genpeptread('ABL1_gp.txt'), 'Sequence');
Kit_seq = getfield(genpeptread('Kit_gp.txt'), 'Sequence');
PDGFRA_seq = getfield(genpeptread('PDGFRA_gp.txt'), 'Sequence');
```

Alternatively, you can obtain protein information directly from the online GenPept database maintained by the National Center for Biotechnology Information (NCBI).

Run these commands to download data from NCBI:

```
% ABL1_seq = getgenpept(target_ABL1, 'SequenceOnly', true);
% Kit_seq = getgenpept(target_Kit, 'SequenceOnly', true);
% PDGFRA_seq = getgenpept(target_PDGFRA, 'SequenceOnly', true);
```

The MATLAB **whos** command gives information about the size of these sequences.

```
whos ABL1_seq
whos Kit_seq
whos PDGFRA_seq
```

Name	Size	Bytes	Class	Attributes
ABL1_seq	1x1149	2298	char	

Name	Size	Bytes	Class	Attributes
------	------	-------	-------	------------

Kit_seq	1x976	1952	char	
Name	Size	Bytes	Class	Attributes
PDGFRA_seq	1x1089	2178	char	

Calling Perl Programs from MATLAB

From MATLAB, you can harness existing Bioperl modules to run a BLAST search on these sequences. MW_BLAST.pl is a Perl program based on the RemoteBlast Bioperl module. It reads sequences from FASTA files, so start by creating a FASTA file for each sequence.

```
fastawrite('ABL1.fa', 'ABL1 Proto-oncogene tyrosine-protein kinase (NP_009297)', ABL1_seq);
fastawrite('Kit.fa', 'Kit Proto-oncogene tyrosine-protein kinase (NP_000213)', Kit_seq);
fastawrite('PDGFRA.fa', 'PDGFRA alpha precursor (NP_006197)', PDGFRA_seq);
```

```
Warning: ABL1.fa already exists. The data will be appended to the file.
Warning: Kit.fa already exists. The data will be appended to the file.
Warning: PDGFRA.fa already exists. The data will be appended to the file.
```

BLAST searches can take a long time to return results, and the Perl program MW_BLAST includes a repeating sleep state to await the report. Sample results have been included with this example, but if you want to try running the BLAST search with the three sequences, uncomment the following commands. MW_BLAST.pl will save the BLAST results in three files on your disk, ABL1.out, Kit.out and PDGFRA.out. The process can take 15 minutes or more.

```
% try
% perl('MW_BLAST.pl', 'blastp', 'pdb', '1e-10', 'ABL1.fa', 'Kit.fa', 'PDGFRA.fa');
% catch
% error(message('bioinfo:bioperldemo:PerlError'))
% end
```

Here is the Perl code for MW_BLAST:

```
type MW_BLAST.pl
```

```
#!/usr/bin/perl -w
use Bio::Tools::Run::RemoteBlast;
use strict;
use 5.006;

# A sample Blast program based on the RemoteBlast.pm Bioperl module. Takes
# parameters for the BLAST search program, the database, and the expectation
# or E-value (defaults: blastp, pdb, 1e-10), followed by a list of FASTA files
# containing sequences to search.

# Copyright 2003-2004 The MathWorks, Inc.

# Retrieve arguments and set parameters
my $prog = shift @ARGV;
my $db = shift @ARGV;
my $e_val= shift @ARGV;

my @params = ('-prog' => $prog,
```

```

        '-data' => $db,
        '-expect' => $e_val,
        '-readmethod' => 'SearchIO' );

# Create a remote BLAST factory
my $factory = Bio::Tools::Run::RemoteBlast->new(@params);

# Change a parameter in RemoteBlast
$Bio::Tools::Run::RemoteBlast::HEADER{'ENTREZ_QUERY'} = 'Homo sapiens [ORGN]';

# Remove a parameter from RemoteBlast
delete $Bio::Tools::Run::RemoteBlast::HEADER{'FILTER'};

# Submit each file
while ( defined($ARGV[0])) {
    my $fa_file = shift @ARGV;
    my $str = Bio::SeqIO->new(-file=>$fa_file, '-format' => 'fasta' );
    my $r = $factory->submit_blast($fa_file);

    # Wait for the reply and save the output file
    while ( my @rids = $factory->each_rid ) {
        foreach my $rid ( @rids ) {
            my $src = $factory->retrieve_blast($rid);
            if( !ref($src) ) {
                if( $src < 0 ) {
                    $factory->remove_rid($rid);
                }
                sleep 5;
            } else {
                my $result = $src->next_result();
                my $filename = $result->query_name()."\.out";
                $factory->save_output($filename);
                $factory->remove_rid($rid);
            }
        }
    }
}

```

The next step is to parse the output reports and find scores ≥ 100 . You can then identify hits found by more than one protein for further research, possibly identifying new targets for drug therapy.

```

try
    protein_list = perl('MW_parse.pl', which('ABL1.out'), which('Kit.out'), which('PDGFRA.out'))
catch
    error(message('bioinfo:bioperldemo:PerlError'))
end

```

```

protein_list =
    ,
    /home/Data/ABL1.out
    1OPL, 2584, 0.0, Chain A, Structural Basis For The Auto-Inhibition Of C-Abl...
    1FMK, 923, 1e-100, Crystal Structure Of Human Tyrosine-Protein Kinase C-Src p...
    1QCF, 919, 1e-100, Chain A, Crystal Structure Of Hck In Complex With A Src Fa...
    1KSW, 916, 1e-100, Chain A, Structure Of Human C-Src Tyrosine Kinase (Thr338g...
    1AD5, 883, 6e-96, Chain A, Src Family Kinase Hck-Amp-Pnp Complex pdb|1AD5|B ...
    2ABL, 866, 5e-94, Sh3-Sh2 Domain Fragment Of Human Bcr-Abl Tyrosine Kinase

```

3LCK, 666, 9e-71, The Kinase Domain Of Human Lymphocyte Kinase (Lck), Activa...
 1QPE, 666, 9e-71, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
 1QPD, 656, 1e-69, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
 1K2P, 620, 2e-65, Chain A, Crystal Structure Of Bruton's Tyrosine Kinase Dom...
 1BYG, 592, 3e-62, Chain A, Kinase Domain Of Human C-Terminal Src Kinase (Csk...
 1M7N, 561, 1e-58, Chain A, Crystal Structure Of Unactivated Apo Insulin-Like...
 1JQH, 560, 2e-58, Chain A, Igf-1 Receptor Kinase Domain pdb|1JQH|B Chain B, ...
 1P40, 560, 2e-58, Chain A, Structure Of Apo Unactivated Igf-1r Kinase Domain...
 1K3A, 553, 1e-57, Chain A, Structure Of The Insulin-Like Growth Factor 1 Rec...
 1GJ0, 550, 2e-57, Chain A, The Fgfr2 Tyrosine Kinase Domain
 1FVR, 540, 3e-56, Chain A, Tie2 Kinase Domain pdb|1FVR|B Chain B, Tie2 Kinas...
 1AB2, 528, 9e-55, Proto-Oncogene Tyrosine Kinase (E.C.2.7.1.112) (Src Homolo...
 1IRK, 525, 2e-54, Insulin Receptor (Tyrosine Kinase Domain) Mutant With Cys ...
 1I44, 523, 3e-54, Chain A, Crystallographic Studies Of An Activation Loop Mu...
 1IR3, 522, 4e-54, Chain A, Phosphorylated Insulin Receptor Tyrosine Kinase I...
 1FGK, 522, 4e-54, Chain A, Crystal Structure Of The Tyrosine Kinase Domain O...
 1P14, 521, 6e-54, Chain A, Crystal Structure Of A Catalytic-Loop Mutant Of T...
 1M14, 496, 4e-51, Chain A, Tyrosine Kinase Domain From Epidermal Growth Fact...
 1PKG, 496, 4e-51, Chain A, Structure Of A C-Kit Kinase Product Complex pdb|1...
 1VR2, 463, 3e-47, Chain A, Human Vascular Endothelial Growth Factor Receptor...
 1JU5, 330, 8e-32, Chain C, Ternary Complex Of An Crk Sh2 Domain, Crk-Derived...
 1BBZ, 317, 3e-30, Chain A, Crystal Structure Of The Abl-Sh3 Domain Complexed...
 1AW0, 303, 1e-28, The Solution Nmr Structure Of Abl Sh3 And Its Relationship...
 1BBZ, 303, 1e-28, Chain E, Crystal Structure Of The Abl-Sh3 Domain Complexed...
 1G83, 287, 8e-27, Chain A, Crystal Structure Of Fyn Sh3-Sh2 pdb|1G83|B Chain...
 1LCK, 270, 7e-25, Chain A, Sh3-Sh2 Domain Fragment Of Human P56-Lck Tyrosine...
 1MU0, 233, 1e-20, Chain A, Crystal Structure Of Aurora-2, An Oncogenic Serin...
 1GRI, 232, 2e-20, Chain A, Grb2 pdb|1GRI|B Chain B, Grb2
 1A9U, 220, 4e-19, The Complex Structure Of The Map Kinase P38SB203580 pdb|1B...
 1BMK, 213, 3e-18, Chain A, The Complex Structure Of The Map Kinase P38SB2186...
 1IAN, 209, 8e-18, Human P38 Map Kinase Inhibitor Complex
 1GZ8, 208, 1e-17, Chain A, Human Cyclin Dependent Kinase 2 Complexed With Th...
 1OVE, 208, 1e-17, Chain A, The Structure Of P38 Alpha In Complex With A Dihy...
 1OIT, 207, 1e-17, Chain A, Imidazopyridines: A Potent And Selective Class Of...
 1B38, 206, 2e-17, Chain A, Human Cyclin-Dependent Kinase 2 pdb|1B39|A Chain ...
 1OGU, 206, 2e-17, Chain A, Structure Of Human Thr160-Phospho Cdk2CYCLIN A CO...
 1E9H, 206, 2e-17, Chain A, Thr 160 Phosphorylated Cdk2 - Human Cyclin A3 Com...
 1JST, 206, 2e-17, Chain A, Phosphorylated Cyclin-Dependent Kinase-2 Bound To...
 1WFC, 206, 2e-17, Structure Of Apo, Unphosphorylated, P38 Mitogen Activated ...
 1QMZ, 206, 2e-17, Chain A, Phosphorylated Cdk2-Cyclin A-Substrate Peptide C...
 1DI8, 206, 2e-17, Chain A, The Structure Of Cyclin-Dependent Kinase 2 (Cdk2)...
 1H1P, 206, 2e-17, Chain A, Structure Of Human Thr160-Phospho Cdk2CYCLIN A CO...
 1DI9, 205, 2e-17, Chain A, The Structure Of P38 Mitogen-Activated Protein Ki...
 1H4L, 202, 5e-17, Chain A, Structure And Regulation Of The Cdk5-P25(Nck5a) C...

----- WARNING -----
 MSG: No HSPs for this minimal Hit (pdb|1H01|A)
 If using NCBI BLAST, check bits() instead

----- WARNING -----
 MSG: No HSPs for this minimal Hit (pdb|1OIR|A)
 If using NCBI BLAST, check bits() instead

----- WARNING -----
 MSG: No HSPs for this minimal Hit (pdb|1GII|A)
 If using NCBI BLAST, check bits() instead

```

-----
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1CSY|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1F3M|C)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1A81|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1H1W|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1B6C|B)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1IG1|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1JJK|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1JOW|B)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1BI8|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|106K|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1GZK|A)
If using NCBI BLAST, check bits() instead
-----

----- WARNING -----

```

MSG: No HSPs for this minimal Hit (pdb|1GZN|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|106L|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1BHF|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1LCJ|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1PME|)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1CM8|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1A1A|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|3HCK|)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1AOT|F)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1PMQ|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1LKK|A)
If using NCBI BLAST, check bits() instead

----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1JNK|)
If using NCBI BLAST, check bits() instead

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1SHD|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1LKL|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1BM2|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1BMB|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1CWD|L)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1BHH|B)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1IA8|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1FBZ|A)
If using NCBI BLAST, check bits() instead
-----
```

```
----- WARNING -----
MSG: No HSPs for this minimal Hit (pdb|1IJR|A)
If using NCBI BLAST, check bits() instead
-----
```

```
/home/Data/Kit.out
```

```
1PKG, 974, 1e-106, Chain A, Structure Of A C-Kit Kinase Product Complex pdb|1...
1VR2, 805, 6e-87, Chain A, Human Vascular Endothelial Growth Factor Receptor...
1GJ0, 730, 3e-78, Chain A, The Fgfr2 Tyrosine Kinase Domain
1FGK, 700, 8e-75, Chain A, Crystal Structure Of The Tyrosine Kinase Domain O...
1OPL, 410, 4e-41, Chain A, Structural Basis For The Auto-Inhibition Of C-Abl...
1FVR, 405, 1e-40, Chain A, Tie2 Kinase Domain pdb|1FVR|B Chain B, Tie2 Kinas...
1M7N, 383, 5e-38, Chain A, Crystal Structure Of Unactivated Apo Insulin-Like...
1P40, 383, 5e-38, Chain A, Structure Of Apo Unactivated Igf-1r Kinase Domain...
1JQH, 381, 8e-38, Chain A, Igf-1 Receptor Kinase Domain pdb|1JQH|B Chain B, ...
1QCF, 377, 2e-37, Chain A, Crystal Structure Of Hck In Complex With A Src Fa...
1K3A, 371, 1e-36, Chain A, Structure Of The Insulin-Like Growth Factor 1 Rec...
```

```

1I44, 368, 3e-36, Chain A, Crystallographic Studies Of An Activation Loop Mu...
1IRK, 367, 3e-36, Insulin Receptor (Tyrosine Kinase Domain) Mutant With Cys ...
1P14, 361, 2e-35, Chain A, Crystal Structure Of A Catalytic-Loop Mutant Of T...
1IR3, 361, 2e-35, Chain A, Phosphorylated Insulin Receptor Tyrosine Kinase I...
3LCK, 354, 1e-34, The Kinase Domain Of Human Lymphocyte Kinase (Lck), Activa...
1QPE, 354, 1e-34, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
1QPD, 354, 1e-34, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
1AD5, 348, 6e-34, Chain A, Src Family Kinase Hck-Amp-Pnp Complex pdb|1AD5|B ...
1KSW, 344, 2e-33, Chain A, Structure Of Human C-Src Tyrosine Kinase (Thr338g...
1FMK, 344, 2e-33, Crystal Structure Of Human Tyrosine-Protein Kinase C-Src p...
1BYG, 342, 3e-33, Chain A, Kinase Domain Of Human C-Terminal Src Kinase (Csk...
1M14, 335, 2e-32, Chain A, Tyrosine Kinase Domain From Epidermal Growth Fact...
1K2P, 294, 1e-27, Chain A, Crystal Structure Of Bruton's Tyrosine Kinase Dom...
1H4L, 167, 5e-13, Chain A, Structure And Regulation Of The Cdk5-P25(Nck5a) C...
1PME, 158, 6e-12, Structure Of Penta Mutant Human Erk2 Map Kinase Complexed ...
1F3M, 156, 1e-11, Chain C, Crystal Structure Of Human SerineTHREONINE KINASE...

```

```
/home/Data/PDGFRA.out
```

```

1PKG, 625, 5e-66, Chain A, Structure Of A C-Kit Kinase Product Complex pdb|1...
1VR2, 550, 2e-57, Chain A, Human Vascular Endothelial Growth Factor Receptor...
1FGI, 500, 1e-51, Chain A, Crystal Structure Of The Tyrosine Kinase Domain O...
1GJ0, 492, 1e-50, Chain A, The Fgfr2 Tyrosine Kinase Domain
1FVR, 419, 4e-42, Chain A, Tie2 Kinase Domain pdb|1FVR|B Chain B, Tie2 Kinas...
1QCF, 380, 1e-37, Chain A, Crystal Structure Of Hck In Complex With A Src Fa...
1QPE, 364, 9e-36, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
1QPD, 364, 9e-36, Chain A, Structural Analysis Of The Lymphocyte-Specific Ki...
3LCK, 360, 2e-35, The Kinase Domain Of Human Lymphocyte Kinase (Lck), Activa...
1OPL, 358, 4e-35, Chain A, Structural Basis For The Auto-Inhibition Of C-Abl...
1FMK, 354, 1e-34, Crystal Structure Of Human Tyrosine-Protein Kinase C-Src p...
1KSW, 353, 2e-34, Chain A, Structure Of Human C-Src Tyrosine Kinase (Thr338g...
1AD5, 353, 2e-34, Chain A, Src Family Kinase Hck-Amp-Pnp Complex pdb|1AD5|B ...
1BYG, 352, 2e-34, Chain A, Kinase Domain Of Human C-Terminal Src Kinase (Csk...
1I44, 351, 3e-34, Chain A, Crystallographic Studies Of An Activation Loop Mu...
1IRK, 350, 4e-34, Insulin Receptor (Tyrosine Kinase Domain) Mutant With Cys ...
1M7N, 349, 5e-34, Chain A, Crystal Structure Of Unactivated Apo Insulin-Like...
1JQH, 349, 5e-34, Chain A, Igf-1 Receptor Kinase Domain pdb|1JQH|B Chain B, ...
1P40, 349, 5e-34, Chain A, Structure Of Apo Unactivated Igf-1r Kinase Domain...
1P14, 344, 2e-33, Chain A, Crystal Structure Of A Catalytic-Loop Mutant Of T...
1IR3, 343, 2e-33, Chain A, Phosphorylated Insulin Receptor Tyrosine Kinase I...
1K3A, 338, 9e-33, Chain A, Structure Of The Insulin-Like Growth Factor 1 Rec...
1M14, 332, 4e-32, Chain A, Tyrosine Kinase Domain From Epidermal Growth Fact...
1K2P, 315, 4e-30, Chain A, Crystal Structure Of Bruton's Tyrosine Kinase Dom...
1PME, 167, 6e-13, Structure Of Penta Mutant Human Erk2 Map Kinase Complexed ...
1JOW, 155, 1e-11, Chain B, Crystal Structure Of A Complex Of Human Cdk6 And ...
1BI8, 155, 1e-11, Chain A, Mechanism Of G1 Cyclin Dependent Kinase Inhibitio...
1F3M, 150, 6e-11, Chain C, Crystal Structure Of Human SerineTHREONINE KINASE...

```

This is the code for MW_parse:

```
type MW_parse.pl
```

```
#!/usr/bin/perl
use Bio::SearchIO;
use strict;
use 5.006;
```

```

# A sample BLAST parsing program based on the SearchIO.pm Bioperl module. Takes
# a list of BLAST report files and prints a list of the top hits from each
# report based on an arbitrary minimum score.

# Copyright 2003-2012 The MathWorks, Inc.

# Set a cutoff value for the raw score.
my $min_score = 100;

# Take each report name and print information about the top hits.
my $seq_count = 0;
while ( defined($ARGV[0]) ) {
    my $breport = shift @ARGV;
    print "\n$breport\n";
    my $in = new Bio::SearchIO(-format => 'blast',
                              -file   => $breport);

    my $num_hit = 0;
    my $short_desc;
    while ( my $result = $in->next_result ) {
        while ( my $curr_hit = $result->next_hit ) {
            if ( $curr_hit->raw_score >= $min_score ) {
                if (length($curr_hit->description) >= 60) {
                    $short_desc = substr($curr_hit->description, 0, 58)."...";
                } else {
                    $short_desc = $curr_hit->description;
                }
                print $curr_hit->accession, " ", " ",
                    $curr_hit->raw_score, " ", " ",
                    $curr_hit->significance, " ", " ",
                    $short_desc, "\n";
            }
            $num_hit++;
        }
    }
    $seq_count++;
}

```

Calling MATLAB Functions within Perl Programs

If you are running on Windows®, it is also possible to call MATLAB functions from Perl. You can launch MATLAB in an Automation Server mode by using the /Automation switch in the MATLAB startup command (e.g. D:\applications\matlab7x\bin\matlab.exe /Automation).

Here's a script to illustrate the process of launching an automation server, calling MATLAB functions and passing variables between Perl and MATLAB.

type [MATLAB_from_Perl.pl](#)

```

#!/usr/bin/perl -w
use Win32::OLE;
use Win32::OLE::Variant;

# Simple perl script to execute commands in Matlab.
# Note the name Win32::OLE is misleading and this actually uses COM!
#

```

```
# Use existing instance if Matlab is already running.
eval {$matlabApp = Win32::OLE->GetActiveObject('Matlab.Application')};
die "Matlab not installed" if $@;
unless (defined $matlabApp) {
    $matlabApp = Win32::OLE->new('Matlab.Application')
    or die "Oops, cannot start Matlab";
}

# Examples of executing MATLAB commands - these functions execute in
# MATLAB and return the status.

@exe_commands = ("IRK = pdbread('pdblirk.ent');",
                 "LCK = pdbread('pdb3lck.ent');",
                 "seqdisp(IRK)",
                 "seqdisp(LCK)",
                 "[Score, Alignment] = swalign(IRK, LCK, 'showscore', 1);");

# send the commands to Matlab
foreach $exe_command (@exe_commands)
{ $status = &send_to_matlab('Execute', $exe_command);
  print "Matlab status = ", $status, "\n";
}

sub send_to_matlab
{ my ($call, @command) = @_;
  my $status = 0;
  print "\n>> $call( @command )\n";
  $result = $matlabApp->Invoke($call, @command);
  if (defined($result))
  { unless ($result =~ s/^\.?{3}/Error:/)
    { print "$result\n" unless ($result eq "");
    }
    else
    { print "$result\n";
      $status = -1;
    }
  }
  return $status;
}

# Examples of passing variables between MATLAB and Perl.
#
# MATLAB supports passing character arrays directly with the following syntax:
#
# PutCharArray([in] BSTR name, [in] BSTR workspace, [in] BSTR string);
# GetCharArray([in] BSTR name, [in] BSTR workspace, [out] BSTR string);

&send_to_matlab('PutCharArray', 'centralDogma', 'base', 'DNA->RNA->Protein. ');
&send_to_matlab('GetCharArray', 'centralDogma', 'base');

# Numeric arrays can be passed by reference in a SAFEARRAY using the
# PutFullMatrix and GetFullMatrix functions.
#
# PutFullMatrix([in] BSTR name, [in] BSTR workspace, [in] BSTR data);
# GetFullMatrix([in] BSTR varname, [in] BSTR workspace, [out] BSTR retdata);
```

```

$mReal = Variant(VT_ARRAY|VT_R8, 4, 4);
$mImag = Variant(VT_ARRAY|VT_R8, 4, 4);

$mReal->Put([[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]);
print "\n>> PutFullMatrix( 'magicArray', 'base', ", '$mReal, $mImag', " )\n";
$matlabApp->PutFullMatrix('magicArray', 'base', $mReal, $mImag);
$matlabApp->Execute('magicArray = magic(4)');

$m2Real = Variant(VT_ARRAY|VT_R8|VT_BYREF,4,4);
$m2Imag = Variant(VT_ARRAY|VT_R8|VT_BYREF,4,4);
print "\n>> GetFullMatrix( 'magicArray', 'base', ", '$m2Real, $m2Imag', " )\n";
$matlabApp->GetFullMatrix('magicArray', 'base', $m2Real, $m2Imag);

for ($i = 0; $i < 4; $i++) {
    printf "%3d %3d %3d %3d\n", $m2Real->Get($i,0), $m2Real->Get($i,1),
        $m2Real->Get($i,2), $m2Real->Get($i,3);
}

# Additionally, you can use Variants to send scalar variables by reference
# to MATLAB for all data types except sparse arrays and function handles through
# PutWorkspaceData:
# PutWorkspaceData([in] BSTR name, [in] BSTR workspace, [in] BSTR data);
#
# Results are passed back to Perl directly with GetVariable:
# HRESULT = GetVariable([in] BSTR Name, [in] BSTR Workspace);

# Create and initialize a date Variant.
$dnaDate = Variant->new(VT_DATE|VT_BYREF, 'Feb 28, 1953');

&send_to_matlab('PutWorkspaceData', 'dnaDate', 'base', $dnaDate);
&send_to_matlab('Execute', 'dnaDate');

# Create and initialize a new string Variant.
$aminoString = Variant->new(VT_BSTR|VT_BYREF, 'matlap');
&send_to_matlab('PutWorkspaceData', 'aminoAcids', 'base', $aminoString);

# Change the value in MATLAB
&send_to_matlab('Execute', "aminoAcids = 'ARNDCQEGHILKMFSTWYV'");

# Bring the new value back
$aa = $matlabApp->GetVariable('aminoAcids', 'base');
printf "Amino acid codes: %s\n", $aa;

undef $matlabApp; # close Matlab if we opened it

```

Protein Analysis Tools in Bioinformatics Toolbox™

MATLAB offers additional tools for protein analysis and further research with these proteins. For example, to access the sequences and run a full Smith-Waterman alignment on the tyrosine kinase domain of the human insulin receptor (pdb 1IRK) and the kinase domain of the human lymphocyte kinase (pdb 3LCK), load the sequence data:

```

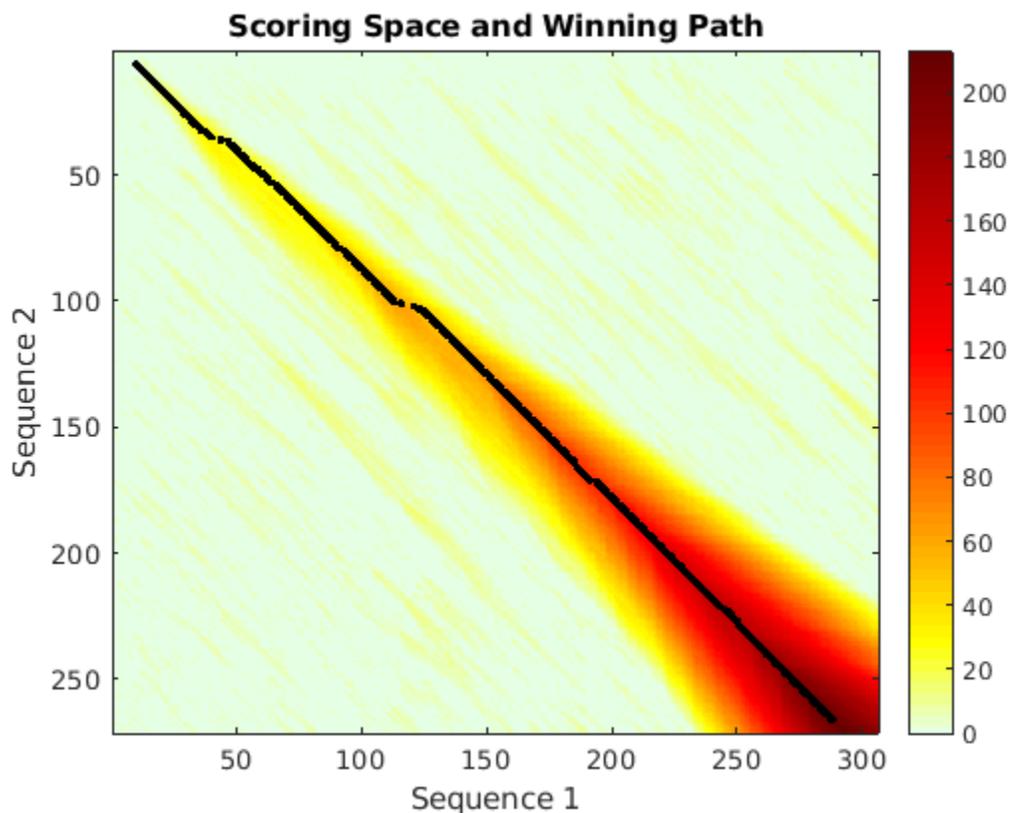
IRK = pdbread('pdb1irk.ent');
LCK = pdbread('pdb3lck.ent');

% Run these commands to bring the data from the Internet:
% IRK = getpdb('1IRK');
% LCK = getpdb('3LCK');

```

Now perform a local alignment with the Smith-Waterman algorithm. MATLAB uses BLOSUM 50 as the default scoring matrix for AA strings with a gap penalty of 8. Of course, you can change any of these parameters.

```
[Score, Alignment] = swalign(IRK, LCK, 'showscore', true);
```



MATLAB and the Bioinformatics Toolbox™ offer additional tools for investigating nucleotide and amino acid sequences. For example, **pdbdistplot** displays the distances between atoms and amino acids in a PDB structure, while **ramachandran** generates a plot of the torsion angle PHI and the torsion angle PSI of the protein sequence. The toolbox function **proteinplot** provides a graphical user interface (GUI) to easily import sequences and plot various properties such as hydrophobicity.

Accessing NCBI Entrez Databases with E-Utilities

This example shows how to programmatically search and retrieve data from NCBI's Entrez databases using NCBI's Entrez Utilities (E-Utilities).

Using NCBI E-Utilities to Retrieve Biological Data

E-Utilities (eUtils) are server-side programs (e.g. ESearch, ESummary, EFetch, etc.) developed and maintained by NCBI for searching and retrieving data from most Entrez Databases. You access tools via URLs with a strict syntax of a specific base URL, a call to the eUtil's script and its associated parameters. For more details on eUtils, see E-Utilities Help.

Searching Nucleotide Database with ESearch

In this example, we consider the genes sequenced from the H5N1 virus, isolated in 1997 from a chicken in Hong Kong as a starting point for our analysis. This particular virus jumped from chickens to humans, killing six people before the spread of the disease was brought under control by destroying all poultry in Hong Kong [1]. You can use ESearch to find the sequence data needed for the analysis. ESearch requires input of a database (`db`) and search term (`term`). Optionally, you can request for ESearch to store your search results on the NCBI history server through the `usehistory` parameter.

```
baseURL = 'https://eutils.ncbi.nlm.nih.gov/entrez/eutils/';
util = 'esearch.fcgi?';
dbParam = 'db=nucleotide';
termParam = '&term=A/chicken/Hong+Kong/915/97+OR+A/chicken/Hong+Kong/915/1997';
usehistoryParam = '&usehistory=y';
esearchURL = [baseURL, util, dbParam, termParam, usehistoryParam]
```

```
esearchURL =
```

```
'https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=nucleotide&term=A/chicken/Hong+Kong/915/97+OR+A/chicken/Hong+Kong/915/1997&usehistory=y'
```

The `term` parameter can be any valid Entrez query. Note that there cannot be any spaces in the URL, so parameters are separated by `&` and any spaces in a query term need to be replaced with `+` (e.g. `'Hong+Kong'`).

You can use `webread` to send the URL and return the results from ESearch as a character array.

```
wbo = weboptions('Timeout', 15); % allow 15 seconds before timeout
searchReport = webread(esearchURL,wbo)
```

```
searchReport =
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE eSearchResult PUBLIC "-//NLM/DTD esearch 20060628//EN" "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.dtd">
<eSearchResult><Count>8</Count><RetMax>8</RetMax><RetStart>0</RetStart><QueryKey>1</QueryKey>
<Id>6048875</Id>
<Id>6048849</Id>
<Id>6048770</Id>
<Id>6048802</Id>
<Id>6048927</Id>
```

```

<Id>6048903</Id>
<Id>6048829</Id>
<Id>3421265</Id>
</IdList><TranslationSet/><TranslationStack> <TermSet> <Term>A/chicken/Hong[All Fields

```

ESearch returns the search results in XML. The report contains information about the query performed, which database was searched and UIDs (unique IDs) to the records that match the query. If you use the history server, the report contains two additional IDs, `WebEnv` and `query_key`, for accessing the results. `WebEnv` is the location of the results on the server, and `query_key` is a number indexing the queries performed. Since `WebEnv` and `query_key` are query dependent they will change every time the search is executed. Either the UIDs or `WebEnv` and `query_key` can be parsed out of the XML report then passed to other eUtils. You can use `regexp` to do the parsing and store the tokens in the structure with fieldnames `WebEnv` and `QueryKey`.

```

ncbi = regexp(searchReport,...
  '<QueryKey>(?!<QueryKey>\w+)</QueryKey>\s*<WebEnv>(?!<WebEnv>\S+)</WebEnv>',...
  'names')

```

```
ncbi =
```

```
struct with fields:
```

```

  QueryKey: '1'
  WebEnv: 'MCID_677df81f3ff10ce8bc0edc67'

```

Finding Links Between Databases with ELink

It might be useful to have PubMed articles related to these genes records. ELink provides this functionality. It finds associations between records within or between databases. You can give ELink the `query_key` and `WebEnv` IDs from above and tell it to find records in the PubMed Database (`db` parameter) associated with your records from the Nucleotide (nuccore) Database (`dbfrom` parameter). ELink returns an XML report with the UIDs for the records in PubMed. These UIDs can be parsed out of the report and passed to other eUtils (e.g. ESummary). Use the stylesheet created for viewing ESummary reports to view the results of ELink.

```

elinkReport = webread([baseUrl...
  'elink.fcgi?dbfrom=nuccore&db=pubmed&WebEnv=', ncbi.WebEnv,...
  '&query_key=', ncbi.QueryKey], wbo);

```

Extract the PubMed UIDs from the ELink report.

```

pubmedIDs = regexp(elinkReport, '<Link>\s+<Id>(\w*)</Id>\s+</Link>', 'tokens');
NumberOfArticles = numel(pubmedIDs)

```

```
% Put PubMed UIDs into a string that can be read by EPost URL.
```

```

pubmed_str = [];
for ii = 1:NumberOfArticles
  pubmed_str = sprintf([pubmed_str '%s,'], char(pubmedIDs{ii}));
end

```

```
NumberOfArticles =
```

2

Posting UIDs to NCBI History Server with EPost

You can use EPost to posts UIDs to the history server. It returns an XML report with a `query_key` and `WebEnv` IDs pointing to the location of the history server. Again, these can be parsed out of the report and used with other eUtils calls.

```
epostReport = webread([baseUrl 'epost.fcgi?db=pubmed&id=',pubmed_str(1:end-1)]);
epostKeys = regexp(epostReport,...
    '<QueryKey>(?!<QueryKey>\w+)</QueryKey>\s*<WebEnv>(?!<WebEnv>\S+)</WebEnv>', 'names')

epostKeys =

    struct with fields:

        QueryKey: '1'
        WebEnv: 'MCID_677df8307a237112a60bbf0e'
```

Using ELink to Find Associated Files Within the Same Database

ELink can do "within" database searches. For example, you can query for a nucleotide sequence within Nucleotide (nuccore) database to find similar sequences, essentially performing a BLAST search. For "within" database searches, ELink returns an XML report containing the related records, along with a score ranking its relationship to the query record. From the above PubMed search, you might be interested in finding all articles related to those articles in PubMed. This is easy to do with ELink. To do a "within" database search, set `db` and `dbfrom` to PubMed. You can use the `query_key` and `WebEnv` from the EPost call.

```
pm2pmReport = webread([baseUrl...
    'elink.fcgi?dbfrom=pubmed&db=pubmed&query_key=', epostKeys.QueryKey,...
    '&WebEnv=', epostKeys.WebEnv]);
pubmedIDs = regexp(pm2pmReport, '(?!<Id>)\w*(?!</Id>)', 'match');
NumberOfArticles = numel(unique(pubmedIDs))

pubmed_str = [];
for ii = 1:NumberOfArticles
    pubmed_str = sprintf([pubmed_str '%s,'], char(pubmedIDs{ii}));
end
```

```
NumberOfArticles =

    403
```

References

[1] Cristianini, N. and Hahn, M.W. "Introduction to Computational Genomics: A Case Studies Approach", Cambridge University Press, 2007.

Microarray Analysis

- “Managing Gene Expression Data in Objects” on page 4-2
- “Representing Expression Data Values in DataMatrix Objects” on page 4-5
- “Representing Expression Data Values in ExptData Objects” on page 4-9
- “Representing Sample and Feature Metadata in MetaData Objects” on page 4-12
- “Representing Experiment Information in a MIAME Object” on page 4-16
- “Representing All Data in an ExpressionSet Object” on page 4-19
- “Analyzing Illumina Bead Summary Gene Expression Data” on page 4-23
- “Detecting DNA Copy Number Alteration in Array-Based CGH Data” on page 4-44
- “Analyzing Array-Based CGH Data Using Bayesian Hidden Markov Modeling” on page 4-60
- “Visualizing Microarray Data” on page 4-74
- “Gene Expression Profile Analysis” on page 4-95
- “Working with GEO Series Data” on page 4-112
- “Identifying Biomolecular Subgroups Using Attractor Metagenes” on page 4-123
- “Working with the Clustergram Function” on page 4-135
- “Working with Objects for Microarray Experiment Data” on page 4-152

Managing Gene Expression Data in Objects

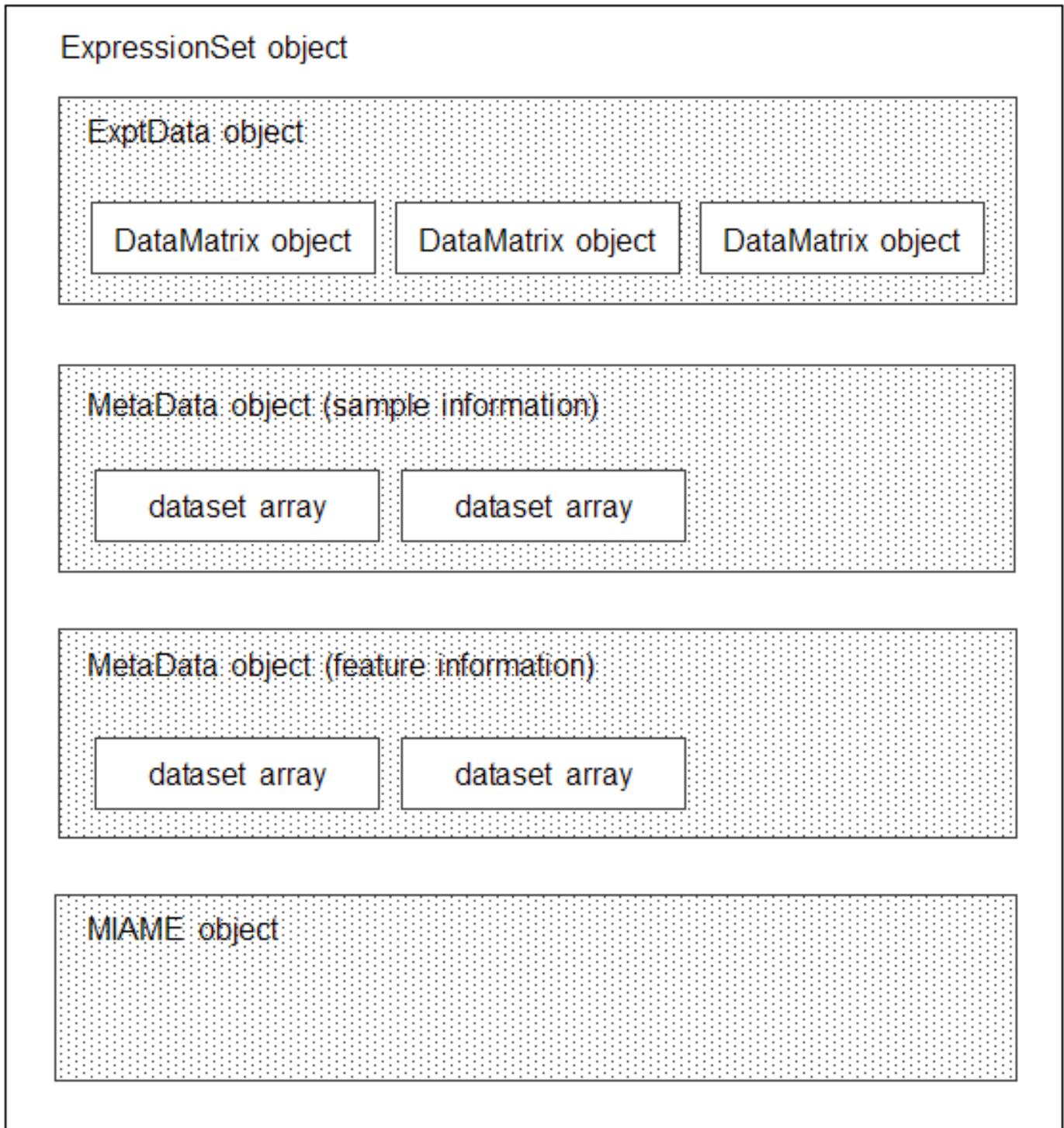
Microarray gene expression experiments are complex, containing data and information from various sources. The data and information from such an experiment is typically subdivided into four categories:

- Measured expression data values
- Sample metadata
- Microarray feature metadata
- Descriptions of experiment methods and conditions

In MATLAB, you can represent all the previous data and information in an ExpressionSet object, which typically contains the following objects:

- One ExptData object containing expression values from a microarray experiment in one or more DataMatrix objects
- One MetaData object containing *sample* metadata in two dataset arrays
- One MetaData object containing *feature* metadata in two dataset arrays
- One MIAME object containing experiment descriptions

The following graphic illustrates a typical ExpressionSet object and its component objects.



Each element (DataMatrix object) in the ExpressionSet object has an element name. Also, there is always one DataMatrix object whose element name is Expressions.

An ExpressionSet object lets you store, manage, and subset the data from a microarray gene expression experiment. An ExpressionSet object includes properties and methods that let you access,

retrieve, and change data, metadata, and other information about the microarray experiment. These properties and methods are useful to view and analyze the data. For a list of the properties and methods, see ExpressionSet class.

To learn more about constructing and using objects for microarray gene expression data and information, see:

- “Representing Expression Data Values in DataMatrix Objects” on page 4-5
- “Representing Expression Data Values in ExptData Objects” on page 4-9
- “Representing Sample and Feature Metadata in MetaData Objects” on page 4-12
- “Representing Experiment Information in a MIAME Object” on page 4-16
- “Representing All Data in an ExpressionSet Object” on page 4-19

Representing Expression Data Values in DataMatrix Objects

In this section...

“Overview of DataMatrix Objects” on page 4-5
 “Constructing DataMatrix Objects” on page 4-5
 “Getting and Setting Properties of a DataMatrix Object” on page 4-6
 “Accessing Data in DataMatrix Objects” on page 4-6

Overview of DataMatrix Objects

The toolbox includes functions, objects, and methods for creating, storing, and accessing microarray data.

The object constructor function, `DataMatrix`, lets you create a `DataMatrix` object to encapsulate data and metadata (row and column names) from a microarray experiment. A `DataMatrix` object stores experimental data in a matrix, with rows typically corresponding to gene names or probe identifiers, and columns typically corresponding to sample identifiers. A `DataMatrix` object also stores metadata, including the gene names or probe identifiers (as the row names) and sample identifiers (as the column names).

You can reference microarray expression values in a `DataMatrix` object the same way you reference data in a MATLAB array, that is, by using linear or logical indexing. Alternately, you can reference this experimental data by gene (probe) identifiers and sample identifiers. Indexing by these identifiers lets you quickly and conveniently access subsets of the data without having to maintain additional index arrays.

Many MATLAB operators and arithmetic functions are available to `DataMatrix` objects by means of methods. These methods let you modify, combine, compare, analyze, plot, and access information from `DataMatrix` objects. Additionally, you can easily extend the functionality by using general element-wise functions, `dmarrayfun` and `dmbsxfun`, and by manually accessing the properties of a `DataMatrix` object.

Note For tables describing the properties and methods of a `DataMatrix` object, see the `DataMatrix` object reference page.

Constructing DataMatrix Objects

- 1 Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a 614-by-7 matrix of gene expression data, `genes`, a cell array of 614 GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a 1-by-7 vector of time values for labeling the columns in `yeastvalues`.

```
load filteredyeastdata
```

- 2 Create variables to contain a subset of the data, specifically the first five rows and first four columns of the `yeastvalues` matrix, the `genes` cell array, and the `times` vector.

```
yeastvalues = yeastvalues(1:5,1:4);
genes = genes(1:5,:);
times = times(1:4);
```

- 3 Import the microarray object package so that the `DataMatrix` constructor function will be available.

```
import bioma.data.*
```

- 4 Use the `DataMatrix` constructor function to create a small `DataMatrix` object from the gene expression data.

```
dmo = DataMatrix(yeastvalues,genes,times)
```

```
dmo =
```

	0	9.5	11.5	13.5
SS DNA	-0.131	1.699	-0.026	0.365
YAL003W	0.305	0.146	-0.129	-0.444
YAL012W	0.157	0.175	0.467	-0.379
YAL026C	0.246	0.796	0.384	0.981
YAL034C	-0.235	0.487	-0.184	-0.669

Getting and Setting Properties of a DataMatrix Object

You use the `get` and `set` methods to retrieve and set properties of a `DataMatrix` object.

- 1 Use the `get` method to display the properties of the `DataMatrix` object, `dmo`.

```
get(dmo)
  Name: ''
  RowNames: {5x1 cell}
  ColNames: {' 0' ' 9.5' '11.5' '13.5'}
  NRows: 5
  NCols: 4
  NDims: 2
  ElementClass: 'double'
```

- 2 Use the `set` method to specify a name for the `DataMatrix` object, `dmo`.

```
dmo = set(dmo,'Name','MyDMObject');
```

- 3 Use the `get` method again to display the properties of the `DataMatrix` object, `dmo`.

```
get(dmo)
  Name: 'MyDMObject'
  RowNames: {5x1 cell}
  ColNames: {' 0' ' 9.5' '11.5' '13.5'}
  NRows: 5
  NCols: 4
  NDims: 2
  ElementClass: 'double'
```

Note For a description of all properties of a `DataMatrix` object, see the `DataMatrix` object reference page.

Accessing Data in DataMatrix Objects

`DataMatrix` objects support the following types of indexing to extract, assign, and delete data:

- Parenthesis () indexing
- Dot . indexing

Parentheses () Indexing

Use parenthesis indexing to extract a subset of the data in `dmo` and assign it to a new DataMatrix object `dmo2`:

```
dmo2 = dmo(1:5,2:3)
dmo2 =
```

	9.5	11.5
SS DNA	1.699	-0.026
YAL003W	0.146	-0.129
YAL012W	0.175	0.467
YAL026C	0.796	0.384
YAL034C	0.487	-0.184

Use parenthesis indexing to extract a subset of the data using row names and column names, and assign it to a new DataMatrix object `dmo3`:

```
dmo3 = dmo({'SS DNA', 'YAL012W', 'YAL034C'}, '11.5')
dmo3 =
```

	11.5
SS DNA	-0.026
YAL012W	0.467
YAL034C	-0.184

Note If you use a cell array of row names or column names to index into a DataMatrix object, the names must be unique, even though the row names or column names within the DataMatrix object are not unique.

Use parenthesis indexing to assign new data to a subset of the elements in `dmo2`:

```
dmo2({'SS DNA', 'YAL003W'}, 1:2) = [1.700 -0.030; 0.150 -0.130]
dmo2 =
```

	9.5	11.5
SS DNA	1.7	-0.03
YAL003W	0.15	-0.13
YAL012W	0.175	0.467
YAL026C	0.796	0.384
YAL034C	0.487	-0.184

Use parenthesis indexing to delete a subset of the data in `dmo2`:

```
dmo2({'SS DNA', 'YAL003W'}, :) = []
dmo2 =
```

	9.5	11.5
YAL012W	0.175	0.467
YAL026C	0.796	0.384
YAL034C	0.487	-0.184

Dot . Indexing

Note In the following examples, notice that when using dot indexing with DataMatrix objects, you specify all rows or all columns using a colon within single quotation marks, (':').

Use dot indexing to extract the data from the 11.5 column only of dmo:

```
timeValues = dmo.(':')( '11.5' )
timeValues =
```

```
-0.0260
-0.1290
 0.4670
 0.3840
-0.1840
```

Use dot indexing to assign new data to a subset of the elements in dmo:

```
dmo.(1:2)( ':' ) = 7
dmo =
```

	0	9.5	11.5	13.5
SS DNA	7	7	7	7
YAL003W	7	7	7	7
YAL012W	0.157	0.175	0.467	-0.379
YAL026C	0.246	0.796	0.384	0.981
YAL034C	-0.235	0.487	-0.184	-0.669

Use dot indexing to delete an entire variable from dmo:

```
dmo.YAL034C = []
dmo =
```

	0	9.5	11.5	13.5
SS DNA	7	7	7	7
YAL003W	7	7	7	7
YAL012W	0.157	0.175	0.467	-0.379
YAL026C	0.246	0.796	0.384	0.981

Use dot indexing to delete two columns from dmo:

```
dmo.( ':' )(2:3) = []
```

```
dmo =
```

	0	13.5
SS DNA	7	7
YAL003W	7	7
YAL012W	0.157	-0.379
YAL026C	0.246	0.981

Representing Expression Data Values in ExptData Objects

In this section...

“Overview of ExptData Objects” on page 4-9
 “Constructing ExptData Objects” on page 4-9
 “Using Properties of an ExptData Object” on page 4-10
 “Using Methods of an ExptData Object” on page 4-10
 “References” on page 4-11

Overview of ExptData Objects

You can use an ExptData object to store expression values from a microarray experiment. An ExprData object stores the data values in one or more DataMatrix objects, each having the same row names (feature names) and column names (sample names). Each element (DataMatrix object) in the ExptData object has an element name.

The following illustrates a small DataMatrix object containing expression values from three samples (columns) and seven features (rows):

	A	B	C
100001_at	2.26	20.14	31.66
100002_at	158.86	236.25	206.27
100003_at	68.11	105.45	82.92
100004_at	74.32	96.68	84.87
100005_at	75.05	53.17	57.94
100006_at	80.36	42.89	77.21
100007_at	216.64	191.32	219.48

An ExptData object lets you store, manage, and subset the data values from a microarray experiment. An ExptData object includes properties and methods that let you access, retrieve, and change data values from a microarray experiment. These properties and methods are useful to view and analyze the data. For a list of the properties and methods, see ExptData class.

Constructing ExptData Objects

The mouseExprsData.txt file used in this example contains data from Hovatta et al., 2005.

- 1 Import the bioma.data package so that the DataMatrix and ExptData constructor functions are available.

```
import bioma.data.*
```

- 2 Use the DataMatrix constructor function to create a DataMatrix object from the gene expression data in the mouseExprsData.txt file. This file contains a table of expression values and metadata (sample and feature names) from a microarray experiment done using the Affymetrix® MGU74Av2 GeneChip® array. There are 26 sample names (A through Z), and 500 feature names (probe set names).

```
dmObj = DataMatrix('File', 'mouseExprsData.txt');
```

- 3 Use the ExptData constructor function to create an ExptData object from the DataMatrix object.

```
EDObj = ExptData(dmObj);
```

- 4** Display information about the ExptData object, EDObj.

```
EDObj
```

```
Experiment Data:  
  500 features, 26 samples  
  1 elements  
Element names: Elmt1
```

Note For complete information on constructing ExptData objects, see ExptData class.

Using Properties of an ExptData Object

To access properties of an ExptData object, use the following syntax:

```
objectname.propertyname
```

For example, to determine the number of elements (DataMatrix objects) in an ExptData object:

```
EDObj.NElements
```

```
ans =
```

```
1
```

To set properties of an ExptData object, use the following syntax:

```
objectname.propertyname = propertyvalue
```

For example, to set the Name property of an ExptData object:

```
EDObj.Name = 'MyExptDataObject'
```

Note Property names are case sensitive. For a list and description of all properties of an ExptData object, see ExptData class.

Using Methods of an ExptData Object

To use methods of an ExptData object, use either of the following syntaxes:

```
objectname.methodname
```

or

```
methodname(objectname)
```

For example, to retrieve the sample names from an ExptData object:

```
EDObj.sampleNames
```

```
Columns 1 through 9
```

```
'A'    'B'    'C'    'D'    'E'    'F'    'G'    'H'    'I'    ...
```

To return the size of an ExptData object:

```
size(EDObj)
ans =
    500    26
```

Note For a complete list of methods of an ExptData object, see ExptData class.

References

- [1] Hovatta, I., Tennant, R S., Helton, R., et al. (2005). Glyoxalase 1 and glutathione reductase 1 regulate anxiety in mice. *Nature* 438, 662-666.

Representing Sample and Feature Metadata in MetaData Objects

In this section...

“Overview of MetaData Objects” on page 4-12
 “Constructing MetaData Objects” on page 4-13
 “Using Properties of a MetaData Object” on page 4-15
 “Using Methods of a MetaData Object” on page 4-15

Overview of MetaData Objects

You can store either sample or feature metadata from a microarray gene expression experiment in a MetaData object. The metadata consists of variable names, for example, related to either samples or microarray features, along with descriptions and values for the variables.

A MetaData object stores the metadata in two dataset arrays:

- **Values dataset array** — A dataset array containing the measured value of each variable per sample or feature. In this dataset array, the columns correspond to variables and rows correspond to either samples or features. The number and names of the columns in this dataset array must match the number and names of the rows in the Descriptions dataset array. If this dataset array contains *sample* metadata, then the number and names of the rows (samples) must match the number and names of the columns in the DataMatrix objects in the same ExpressionSet object. If this dataset array contains *feature* metadata, then the number and names of the rows (features) must match the number and names of the rows in the DataMatrix objects in the same ExpressionSet object.
- **Descriptions dataset array** — A dataset array containing a list of the variable names and their descriptions. In this dataset array, each row corresponds to a variable. The row names are the variable names, and a column, named `VariableDescription`, contains a description of the variable. The number and names of the rows in the Descriptions dataset array must match the number and names of the columns in the Values dataset array.

The following illustrates a dataset array containing the measured value of each variable per sample or feature:

	Gender	Age	Type	Strain	Source
A	'Male'	8	'Wild type'	'129S6/SvEvTac'	'amygdala'
B	'Male'	8	'Wild type'	'129S6/SvEvTac'	'amygdala'
C	'Male'	8	'Wild type'	'129S6/SvEvTac'	'amygdala'
D	'Male'	8	'Wild type'	'A/J '	'amygdala'
E	'Male'	8	'Wild type'	'A/J '	'amygdala'
F	'Male'	8	'Wild type'	'C57BL/6J '	'amygdala'

The following illustrates a dataset array containing a list of the variable names and their descriptions:

	VariableDescription
id	'Sample identifier'
Gender	'Gender of the mouse in study'
Age	'The number of weeks since mouse birth'
Type	'Genetic characters'
Strain	'The mouse strain'
Source	'The tissue source for RNA collection'

A MetaData object lets you store, manage, and subset the metadata from a microarray experiment. A MetaData object includes properties and methods that let you access, retrieve, and change metadata from a microarray experiment. These properties and methods are useful to view and analyze the metadata. For a list of the properties and methods, see MetaData class

Constructing MetaData Objects

Constructing a MetaData Object from Two dataset Arrays

- 1 Import the `bioma.data` package so that the MetaData constructor function is available.

```
import bioma.data.*
```

- 2 Load some sample data, which includes Fisher's iris data of 5 measurements on a sample of 150 irises.

```
load fisheriris
```

- 3 Create a dataset array from some of Fisher's iris data. The dataset array will contain 750 measured values, one for each of 150 samples (iris replicates) at five variables (species, SL, SW, PL, PW). In this dataset array, the rows correspond to samples, and the columns correspond to variables.

```
irisValues = dataset({nominal(species), 'species'}, ...
                    {meas, 'SL', 'SW', 'PL', 'PW'});
```

- 4 Create another dataset array containing a list of the variable names and their descriptions. This dataset array will contain five rows, each corresponding to the five variables: species, SL, SW, PL, and PW. The first column will contain the variable name. The second column will have a column header of `VariableDescription` and contain a description of the variable.

```
% Create 5-by-1 cell array of description text for the variables
varDesc = {'Iris species', 'Sepal Length', 'Sepal Width', ...
           'Petal Length', 'Petal Width'};
```

```
% Create the dataset array from the variable descriptions
irisVarDesc = dataset(varDesc, ...
                     'ObsNames', {'species', 'SL', 'SW', 'PL', 'PW'}, ...
                     'VarNames', {'VariableDescription'})
```

```
irisVarDesc =
```

	VariableDescription
species	'Iris species'
SL	'Sepal Length'
SW	'Sepal Width'
PL	'Petal Length'
PW	'Petal Width'

- 5 Create a MetaData object from the two dataset arrays.

```
MDObj1 = MetaData(irisValues, irisVarDesc);
```

Constructing a MetaData Object from a Text File

- 1 Import the `bioma.datapackage` so that the MetaData constructor function is available.

```
import bioma.data.*
```

- 2 View the `mouseSampleData.txt` file included with the Bioinformatics Toolbox software.

Note that this text file contains two tables. One table contains 130 measured values, one for each of 26 samples (A through Z) at five variables (Gender, Age, Type, Strain, and Source). In this table, the rows correspond to samples, and the columns correspond to variables. The second table has lines prefaced by the # symbol. It contains five rows, each corresponding to the five variables: Gender, Age, Type, Strain, and Source. The first column contains the variable name. The second column has a column header of VariableDescription and contains a description of the variable.

```
# id: Sample identifier
# Gender: Gender of the mouse in study
# Age: The number of weeks since mouse birth
# Type: Genetic characters
# Strain: The mouse strain
# Source: The tissue source for RNA collection
ID   Gender   Age   Type      Strain      Source
A    Male     8    Wild type  129S6/SvEvTac  amygdala
B    Male     8    Wild type  129S6/SvEvTac  amygdala
C    Male     8    Wild type  129S6/SvEvTac  amygdala
D    Male     8    Wild type  A/J          amygdala
E    Male     8    Wild type  A/J          amygdala
F    Male     8    Wild type  C57BL/6J     amygdala
G    Male     8    Wild type  C57BL/6J     amygdala
H    Male     8    Wild type  129S6/SvEvTac  cingulate cortex
I    Male     8    Wild type  129S6/SvEvTac  cingulate cortex
J    Male     8    Wild type  A/J          cingulate cortex
K    Male     8    Wild type  A/J          cingulate cortex
L    Male     8    Wild type  A/J          cingulate cortex
M    Male     8    Wild type  C57BL/6J     cingulate cortex
N    Male     8    Wild type  C57BL/6J     cingulate cortex
O    Male     8    Wild type  129S6/SvEvTac  hippocampus
P    Male     8    Wild type  129S6/SvEvTac  hippocampus
Q    Male     8    Wild type  A/J          hippocampus
R    Male     8    Wild type  A/J          hippocampus
S    Male     8    Wild type  C57BL/6J     hippocampus
T    Male     8    Wild type  C57BL/6J4    hippocampus
U    Male     8    Wild type  129S6/SvEvTac  hypothalamus
V    Male     8    Wild type  129S6/SvEvTac  hypothalamus
W    Male     8    Wild type  A/J          hypothalamus
X    Male     8    Wild type  A/J          hypothalamus
Y    Male     8    Wild type  C57BL/6J     hypothalamus
Z    Male     8    Wild type  C57BL/6J     hypothalamus
```

3 Create a MetaData object from the metadata in the mouseSampleData.txt file.

```
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#')
```

Sample Names:

```
A, B, ...,Z (26 total)
```

Variable Names and Meta Information:

	VariableDescription
Gender	' Gender of the mouse in study'
Age	' The number of weeks since mouse birth'
Type	' Genetic characters'
Strain	' The mouse strain'
Source	' The tissue source for RNA collection'

For complete information on constructing MetaData objects, see MetaData class.

Using Properties of a MetaData Object

To access properties of a MetaData object, use the following syntax:

```
objectname.propertyname
```

For example, to determine the number of variables in a MetaData object:

```
MDObj2.NVariables
```

```
ans =
```

```
    5
```

To set properties of a MetaData object, use the following syntax:

```
objectname.propertyname = propertyvalue
```

For example, to set the Description property of a MetaData object:

```
MDObj1.Description = 'This is my MetaData object for my sample metadata'
```

Note Property names are case sensitive. For a list and description of all properties of a MetaData object, see MetaData class.

Using Methods of a MetaData Object

To use methods of a MetaData object, use either of the following syntaxes:

```
objectname.methodname
```

or

```
methodname(objectname)
```

For example, to access the dataset array in a MetaData object that contains the variable values:

```
MDObj2.variableValues;
```

To access the dataset array of a MetaData object that contains the variable descriptions:

```
variableDesc(MDObj2)
```

```
ans =
```

```

      VariableDescription
Gender   ' Gender of the mouse in study'
Age      ' The number of weeks since mouse birth'
Type     ' Genetic characters'
Strain   ' The mouse strain'
Source   ' The tissue source for RNA collection'
```

Note For a complete list of methods of a MetaData object, see MetaData class.

Representing Experiment Information in a MIAME Object

In this section...

“Overview of MIAME Objects” on page 4-16
 “Constructing MIAME Objects” on page 4-16
 “Using Properties of a MIAME Object” on page 4-17
 “Using Methods of a MIAME Object” on page 4-18

Overview of MIAME Objects

You can store information about experimental methods and conditions from a microarray gene expression experiment in a MIAME object. It loosely follows the Minimum Information About a Microarray Experiment (MIAME) specification. It can include information about:

- Experiment design
- Microarrays used
- Samples used
- Sample preparation and labeling
- Hybridization procedures and parameters
- Normalization controls
- Preprocessing information
- Data processing specifications

A MIAME object includes properties and methods that let you access, retrieve, and change experiment information related to a microarray experiment. These properties and methods are useful to view and analyze the information. For a list of the properties and methods, see MIAME class.

Constructing MIAME Objects

For complete information on constructing MIAME objects, see MIAME class.

Constructing a MIAME Object from a GEO Structure

- 1 Import the `bioma.data` package so that the MIAME constructor function is available.

```
import bioma.data.*
```

- 2 Use the `getgeodata` function to return a MATLAB structure containing Gene Expression Omnibus (GEO) Series data related to accession number GSE4616.

```
geoStruct = getgeodata('GSE4616')
```

```
geoStruct =
```

```
Header: [1x1 struct]
Data: [12488x12 bioma.data.DataMatrix]
```

- 3 Use the MIAME constructor function to create a MIAME object from the structure.

```
MIAMEObj1 = MIAME(geoStruct);
```

4 Display information about the MIAME object, MIAMEObj.

```
MIAMEObj1

MIAMEObj1 =

Experiment Description:
  Author name: Mika,,Silvennoinen
  Riikka,,KivelÃ
  Maarit,,Lehti
  Anna-Maria,,Touvras
  Jyrki,,Komulainen
  Veikko,,Vihko
  Heikki,,Kainulainen
  Laboratory: LIKES - Research Center
  Contact information: Mika,,Silvennoinen
  URL:
  PubMedIDs: 17003243
  Abstract: A 90 word abstract is available. Use the Abstract property.
  Experiment Design: A 234 word summary is available. Use the ExptDesign property.
  Other notes:
    [1x80 char]
```

Constructing a MIAME Object from Properties**1** Import the `bioma.data` package so that the MIAME constructor function is available.

```
import bioma.data.*
```

2 Use the MIAME constructor function to create a MIAME object using individual properties.

```
MIAMEObj2 = MIAME('investigator', 'Jane Researcher',...
                  'lab', 'One Bioinformatics Laboratory',...
                  'contact', 'jresearcher@lab.not.exist',...
                  'url', 'www.lab.not.exist',...
                  'title', 'Normal vs. Diseased Experiment',...
                  'abstract', 'Example of using expression data',...
                  'other', {'Notes:Created from a text file.'});
```

3 Display information about the MIAME object, MIAMEObj2.

```
MIAMEObj2

MIAMEObj2 =

Experiment Description:
  Author name: Jane Researcher
  Laboratory: One Bioinformatics Laboratory
  Contact information: jresearcher@lab.not.exist
  URL: www.lab.not.exist
  PubMedIDs:
  Abstract: A 4 word abstract is available. Use the Abstract property.
  No experiment design summary available.
  Other notes:
    'Notes:Created from a text file.'
```

Using Properties of a MIAME Object

To access properties of a MIAME object, use the following syntax:

```
objectname.propertyname
```

For example, to retrieve the PubMed identifier of publications related to a MIAME object:

```
MIAMEObj1.PubMedID
```

```
ans =
```

17003243

To set properties of a MIAME object, use the following syntax:

objectname.propertyname = propertyvalue

For example, to set the Laboratory property of a MIAME object:

```
MIAMEObj1.Laboratory = 'XYZ Lab'
```

Note Property names are case sensitive. For a list and description of all properties of a MIAME object, see MIAME class.

Using Methods of a MIAME Object

To use methods of a MIAME object, use either of the following syntaxes:

objectname.methodname

or

methodname(objectname)

For example, to determine if a MIAME object is empty:

```
MIAMEObj1.isempty
```

```
ans =
```

```
    0
```

Note For a complete list of methods of a MIAME object, see MIAME class.

Representing All Data in an ExpressionSet Object

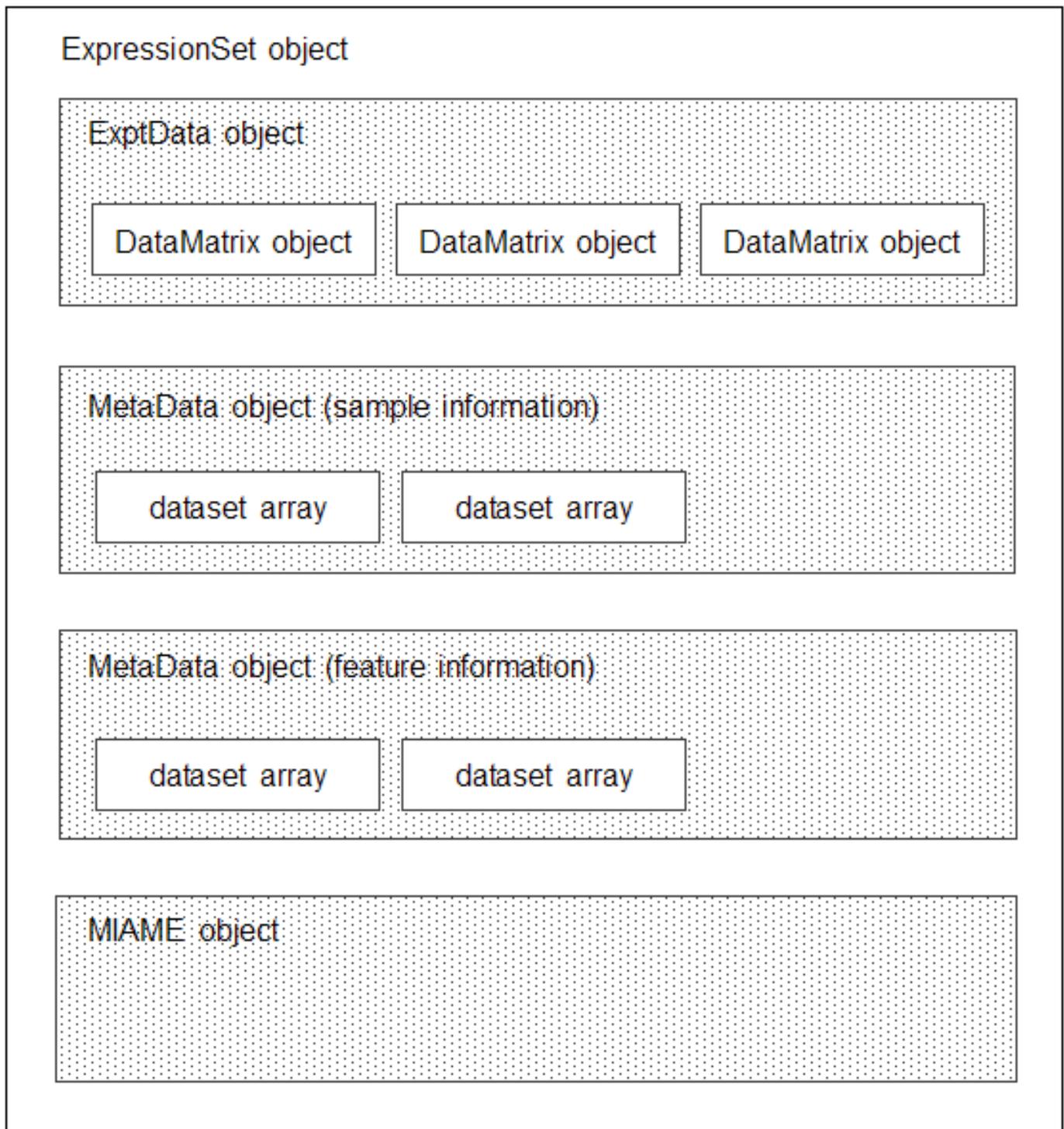
In this section...
“Overview of ExpressionSet Objects” on page 4-19
“Constructing ExpressionSet Objects” on page 4-21
“Using Properties of an ExpressionSet Object” on page 4-21
“Using Methods of an ExpressionSet Object” on page 4-22

Overview of ExpressionSet Objects

You can store all microarray experiment data and information in one object by assembling the following into an ExpressionSet object:

- One ExptData object containing expression values from a microarray experiment in one or more DataMatrix objects
- One MetaData object containing *sample* metadata in two dataset arrays
- One MetaData object containing *feature* metadata in two dataset arrays
- One MIAME object containing experiment descriptions

The following graphic illustrates a typical ExpressionSet object and its component objects.



Each element (DataMatrix object) in the ExpressionSet object has an element name. Also, there is always one DataMatrix object whose element name is Expressions.

An ExpressionSet object lets you store, manage, and subset the data from a microarray gene expression experiment. An ExpressionSet object includes properties and methods that let you access,

retrieve, and change data, metadata, and other information about the microarray experiment. These properties and methods are useful to view and analyze the data. For a list of the properties and methods, see ExpressionSet class.

Constructing ExpressionSet Objects

Note The following procedure assumes you have executed the example code in the previous sections:

- “Representing Expression Data Values in ExptData Objects” on page 4-9
- “Representing Sample and Feature Metadata in MetaData Objects” on page 4-12
- “Representing Experiment Information in a MIAME Object” on page 4-16

- 1 Import the bioma package so that the ExpressionSet constructor function is available.

```
import bioma.*
```

- 2 Construct an ExpressionSet object from EDObj, an ExptData object, MDObj2, a MetaData object containing sample variable information, and MIAMEObj, a MIAME object.

```
ESObj = ExpressionSet(EDObj, 'SData', MDObj2, 'EInfo', MIAMEObj1);
```

- 3 Display information about the ExpressionSet object, ESObj.

```
ESObj
```

```
ExpressionSet
Experiment Data: 500 features, 26 samples
  Element names: Expressions
Sample Data:
  Sample names:      A, B, ...,Z (26 total)
  Sample variable names and meta information:
    Gender: Gender of the mouse in study
    Age: The number of weeks since mouse birth
    Type: Genetic characters
    Strain: The mouse strain
    Source: The tissue source for RNA collection
Feature Data: none
Experiment Information: use 'exptInfo(obj)'
```

For complete information on constructing ExpressionSet objects, see ExpressionSet class.

Using Properties of an ExpressionSet Object

To access properties of an ExpressionSet object, use the following syntax:

```
objectname.propertyname
```

For example, to determine the number of samples in an ExpressionSet object:

```
ESObj.NSamples
```

```
ans =
```

```
26
```

Note Property names are case sensitive. For a list and description of all properties of an ExpressionSet object, see ExpressionSet class.

Using Methods of an ExpressionSet Object

To use methods of an ExpressionSet object, use either of the following syntaxes:

objectname.methodname

or

methodname(objectname)

For example, to retrieve the sample variable names from an ExpressionSet object:

```
ESObj.sampleVarNames
```

```
ans =
```

```
  'Gender'  'Age'  'Type'  'Strain'  'Source'
```

To retrieve the experiment information contained in an ExpressionSet object:

```
exptInfo(ESObj)
```

```
ans =
```

```
Experiment description
```

```
  Author name: Mika,,Silvennoinen
```

```
Riikka,,KivelÃ
```

```
Maarit,,Lehti
```

```
Anna-Maria,,Touvras
```

```
Jyrki,,Komulainen
```

```
Veikko,,Vihko
```

```
Heikki,,Kainulainen
```

```
  Laboratory: XYZ Lab
```

```
  Contact information: Mika,,Silvennoinen
```

```
  URL:
```

```
  PubMedIDs: 17003243
```

```
  Abstract: A 90 word abstract is available Use the Abstract property.
```

```
  Experiment Design: A 234 word summary is available Use the ExptDesign property.
```

```
  Other notes:
```

```
  [1x80 char]
```

Note For a complete list of methods of an ExpressionSet object, see ExpressionSet class.

Analyzing Illumina Bead Summary Gene Expression Data

This example shows how to analyze Illumina® BeadChip gene expression summary data using MATLAB® and Bioinformatics Toolbox™ functions.

Introduction

This example shows how to import and analyze gene expression data from the Illumina BeadChip microarray platform. The data set in the example is from the study of gene expression profiles of human spermatogenesis by Platts et al. [1]. The expression levels were measured on Illumina Sentrix Human 6 (WG6) BeadChips.

The data from most microarray gene expression experiments generally consists of four components: experiment data values, sample information, feature annotations, and information about the experiment. This example uses microarray data, constructs each of the four components, assembles them into an `ExpressionSet` object, and finds the differentially expressed genes. For more examples about the `ExpressionSet` class, see “Working with Objects for Microarray Experiment Data” on page 4-152.

Importing Experimental Data from the GEO Database

Samples were hybridized on three Illumina Sentrix Human 6 (WG6) BeadChips. Retrieve the GEO Series data *GSE6967* using `getgeodata` function.

```
TNGEOData = getgeodata('GSE6967')
```

```
TNGEOData =
```

```
struct with fields:
```

```
Header: [1x1 struct]
Data: [47293x13 bioma.data.DataMatrix]
```

The `TNGEOData` structure contains `Header` and `Data` fields. The `Header` field has two fields, `Series` and `Samples`, containing a description of the experiment and samples respectively. The `Data` field contains a `DataMatrix` of normalized summary expression levels from the experiment.

Determine the number of samples in the experiment.

```
nSamples = numel(TNGEOData.Header.Samples.geo_accession)
```

```
nSamples =
```

```
13
```

Inspect the sample titles from the `Header.Samples` field.

```
TNGEOData.Header.Samples.title'
```

```
ans =
```

13×1 cell array

```
{'Teratozoospermic individual: Sample T2'}
{'Teratozoospermic individual: Sample T1'}
{'Teratozoospermic individual: Sample T6'}
{'Teratozoospermic individual: Sample T4'}
{'Teratozoospermic individual: Sample T8'}
{'Normospermic individual: Sample N11' }
{'Teratozoospermic individual: Sample T3'}
{'Teratozoospermic individual: Sample T7'}
{'Teratozoospermic individual: Sample T5'}
{'Normospermic individual: Sample N6' }
{'Normospermic individual: Sample N12' }
{'Normospermic individual: Sample N5' }
{'Normospermic individual: Sample N1' }
```

For simplicity, extract the sample labels from the sample titles.

```
sampleLabels = cellfun(@(x) char(regexp(x, '\w\d+', 'match')),...
    TNGE0Data.Header.Samples.title, 'UniformOutput', false)
```

sampleLabels =

1×13 cell array

Columns 1 through 7

```
{'T2'} {'T1'} {'T6'} {'T4'} {'T8'} {'N11'} {'T3'}
```

Columns 8 through 13

```
{'T7'} {'T5'} {'N6'} {'N12'} {'N5'} {'N1'}
```

Importing Expression Data from Illumina BeadStudio Summary Files

Download the supplementary file GSE6967_RAW.tar and unzip the file to access the 13 text files produced by the BeadStudio software, which contain the raw, non-normalized bead summary values.

The raw data files are named with their GSM accession numbers. For this example, construct the file names of the text data files using the path where the text files are located.

```
rawDataFiles = cell(1,nSamples);
for i = 1:nSamples
    rawDataFiles {i} = [TNGE0Data.Header.Samples.geo_accession{i} '.txt'];
end
```

Set the variable rawDataPath to the path and directory to which you extracted the data files.

```
rawDataPath = 'C:\Examples\illuminagedemo\GSE6967';
```

Use the ilmnbsread function to read the first of the summary files and store the content in a structure.

```
rawData = ilmnbsread(fullfile(rawDataPath, rawDataFiles{1}))
```

rawData =

```

struct with fields:
    Header: [1x1 struct]
    TargetID: {47293x1 cell}
    ColumnNames: {1x8 cell}
    Data: [47293x8 double]
    TextColumnNames: {}
    TextData: {}

```

Inspect the column names in the `rawData` structure.

```
rawData.ColumnNames'
```

```
ans =
```

```

8x1 cell array

{'MIN_Signal-1412091085_A' }
{'AVG_Signal-1412091085_A' }
{'MAX_Signal-1412091085_A' }
{'NARRAYS-1412091085_A' }
{'ARRAY_STDEV-1412091085_A' }
{'BEAD_STDEV-1412091085_A' }
{'Avg_NBEADS-1412091085_A' }
{'Detection-1412091085_A' }

```

Determine the number of target probes.

```
nTargets = size(rawData.Data, 1)
```

```
nTargets =
```

```
47293
```

Read the non-normalized expression values (`Avg_Signal`), the detection confidence limits and the Sentrix chip IDs from the summary data files. The gene expression values are identified with Illumina probe target IDs. You can specify the columns to read from the data file.

```

rawMatrix = bioma.data.DataMatrix(zeros(nTargets, nSamples),...
    rawData.TargetID, sampleLabels);
detectionConf = bioma.data.DataMatrix(zeros(nTargets, nSamples),...
    rawData.TargetID, sampleLabels);
chipIDs = categorical([]);

for i = 1:nSamples
    rawData = ilmnbsread(fullfile(rawDataPath, rawDataFiles{i}),...
        'COLUMNS', {'AVG_Signal', 'Detection'});
    chipIDs(i) = regexp(rawData.ColumnNames(1), '\d*', 'match', 'once');
    rawMatrix(:, i) = rawData.Data(:, 1);
    detectionConf(:, i) = rawData.Data(:, 2);
end

```

There are three Sentrix BeadChips used in the experiment. Inspect the Illumina Sentrix BeadChip IDs in `chipIDs` to determine the number of samples hybridized on each chip.

```
summary(chipIDs)
```

```
samplesChip1 = sampleLabels(chipIDs=='1412091085')
samplesChip2 = sampleLabels(chipIDs=='1412091086')
samplesChip3 = sampleLabels(chipIDs=='1477791158')
```

```
<strong>chipIDs</strong>: 1×13 categorical
```

```
      <strong>1412091085</strong>      <strong>1412091086</strong>      <strong>1477791158</strong>
      6              4              3              0
```

```
samplesChip1 =
```

```
1×6 cell array
```

```
{'T2'} {'T1'} {'T6'} {'T4'} {'T8'} {'N11'}
```

```
samplesChip2 =
```

```
1×4 cell array
```

```
{'T3'} {'T7'} {'T5'} {'N6'}
```

```
samplesChip3 =
```

```
1×3 cell array
```

```
{'N12'} {'N5'} {'N1'}
```

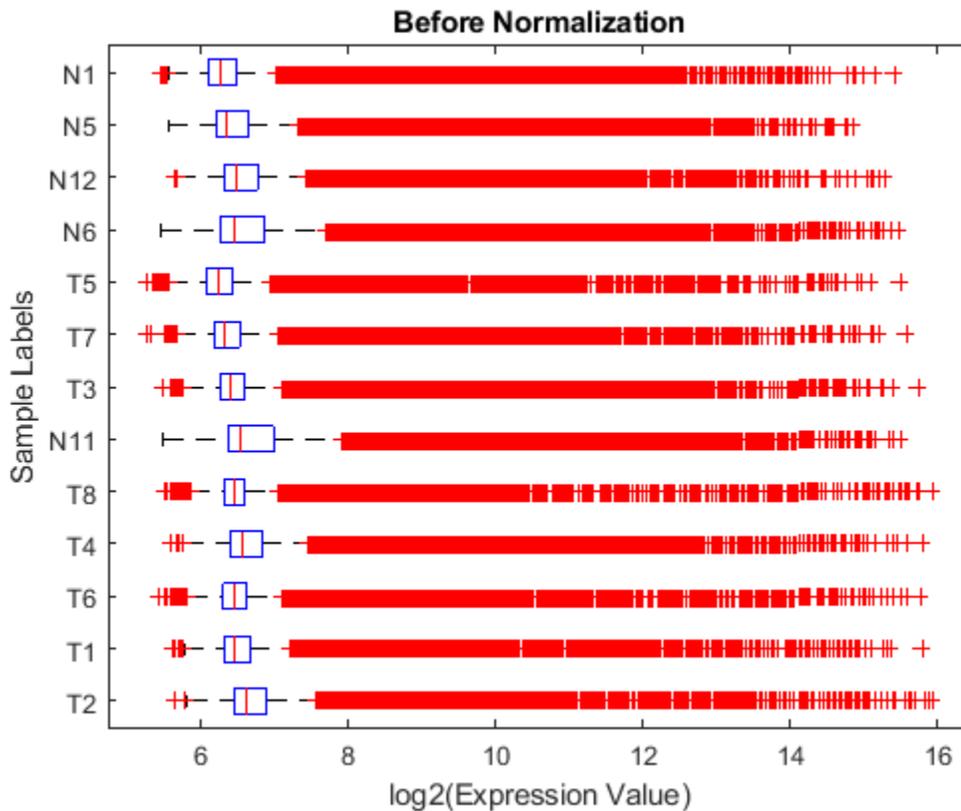
Six samples (T2, T1, T6, T4, T8, and N11) were hybridized to six arrays on the first chip, four samples (T3, T7, T5, and N6) on the second chip, and three samples (N12, N5, and N1) on the third chip.

Normalizing the Expression Data

Use a boxplot to view the raw expression levels of each sample in the experiment.

```
logRawExprs = log2(rawMatrix);
```

```
maboxplot(logRawExprs, 'Orientation', 'horizontal')
ylabel('Sample Labels')
xlabel('log2(Expression Value)')
title('Before Normalization')
```

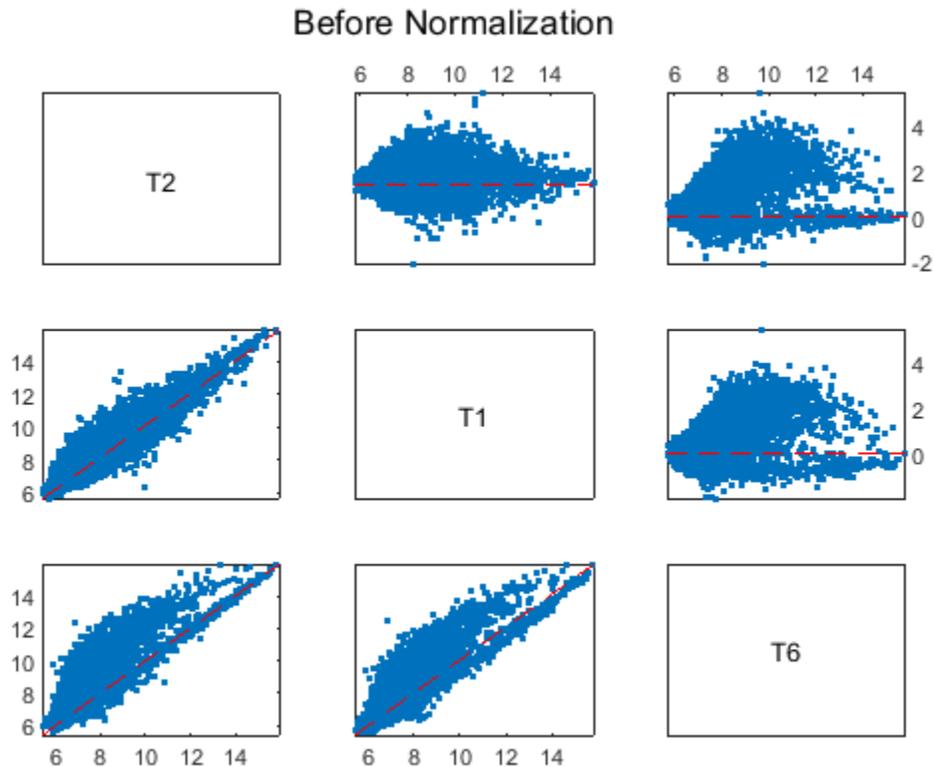


The difference in intensities between samples on the same chip and samples on different chips does not seem too large. The first BeadChip, containing samples T2, T1, T6, T4, T8 and N11, seems to be slightly more variable than others.

Using MA and XY plots to do a pairwise comparison of the arrays on a BeadChip can be informative. On an MA plot, the average (A) of the expression levels of two arrays are plotted on the x axis, and the difference (M) in the measurement on the y axis. An XY plot is a scatter plot of one array against another. This example uses the helper function `maxyplot` to plot MAXY plots for a pairwise comparison of the three arrays on the first chip hybridized with teratozoospermic samples (T2, T1 and T6).

Note: You can also use the `mairplot` function to create the MA or IR (Intensity/Ratio) plots for comparison of specific arrays.

```
inspectIdx = 1:3;
maxyplot(rawMatrix, inspectIdx)
sgtitle('Before Normalization')
```



In an MAXY plot, the MA plots for all pairwise comparisons are in the upper-right corner. In the lower-left corner are the XY plots of the comparisons. The MAXY plot shows the two arrays, T1 and T2, to be quite similar, while different from the other array, T6.

The expression box plots and MAXY plots reveal that there are differences in expression levels within chips and between chips; hence, the data requires normalization. Use the `quantilenorm` function to apply quantile normalization to the raw data.

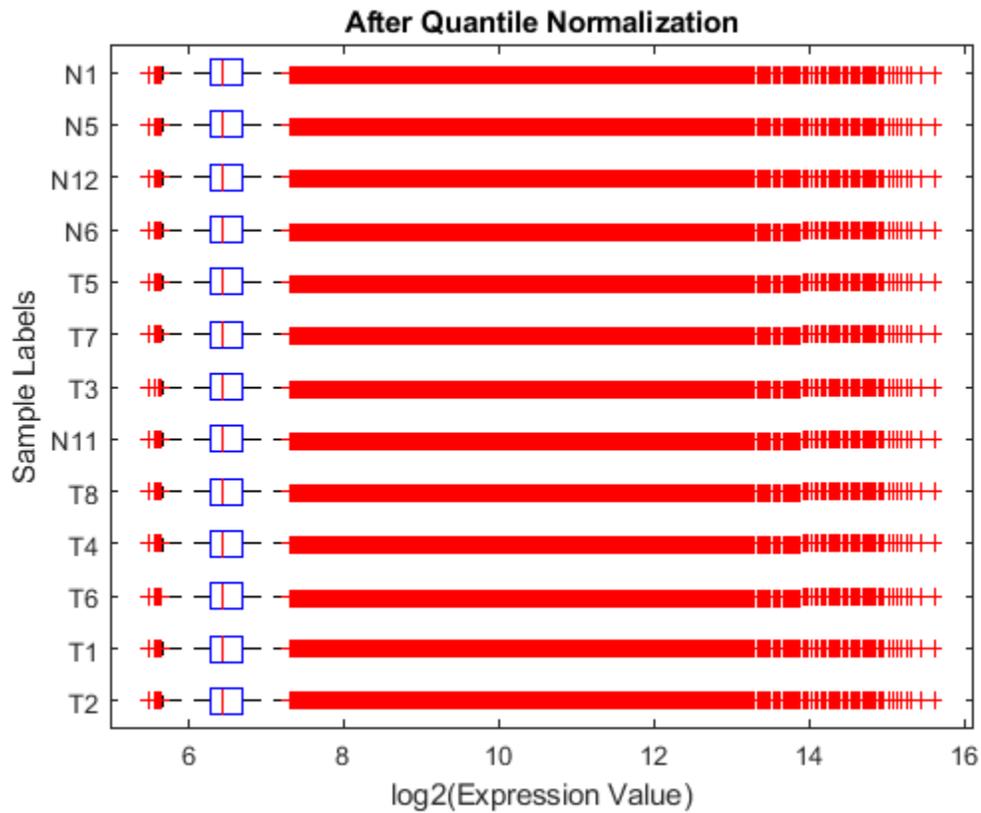
Note: You can also try invariant set normalization using the `mainvarsetnorm` function.

```
normExprs = rawMatrix;
normExprs(:, :) = quantilenorm(rawMatrix.(':'))(':');
```

```
log2NormExprs = log2(normExprs);
```

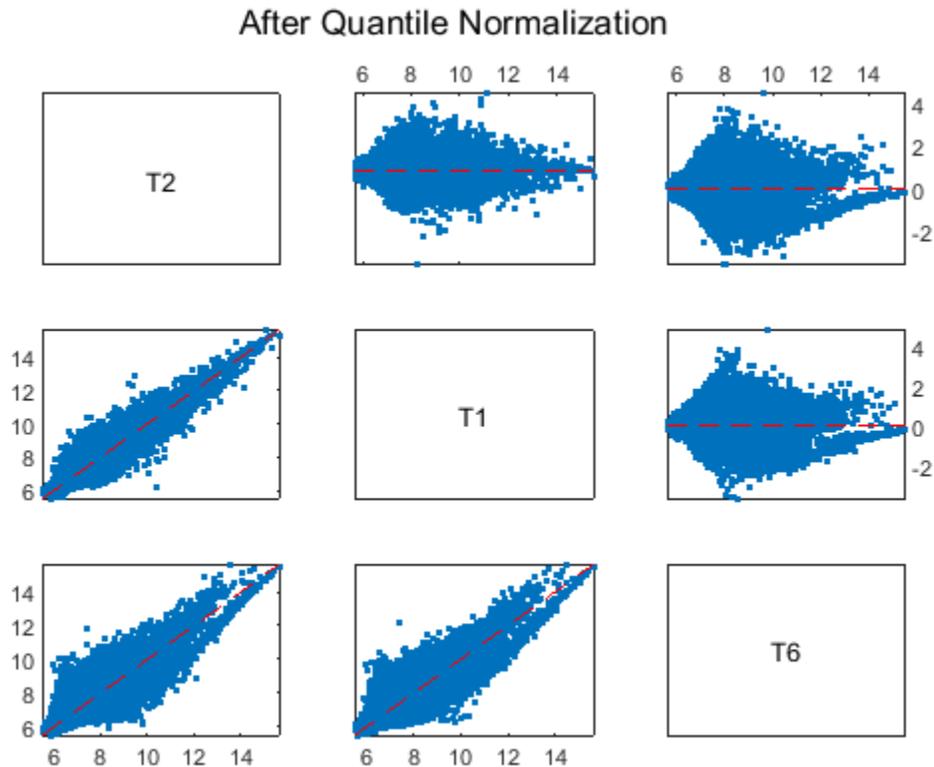
Display and inspect the normalized expression levels in a box plot.

```
figure;
maboxplot(log2NormExprs, 'ORIENTATION', 'horizontal')
ylabel('Sample Labels')
xlabel('log2(Expression Value)')
title('After Quantile Normalization')
```



Display and inspect the MAXY plot of the three arrays (T2, T1 and T6) on the first chip after the normalization.

```
maxyplot(normExprs, inspectIdx)  
sgtitle('After Quantile Normalization')
```



Many of the genes in this study are not expressed, or have only small variability across the samples.

First, you can remove genes with very low absolute expression values by using `genelowvalfilter`.

```
[mask, log2NormExprs] = genelowvalfilter(log2NormExprs);
detectionConf = detectionConf(mask, :);
```

Second, filter out genes with a small variance across samples using `genevarfilter`.

```
[mask, log2NormExprs] = genevarfilter(log2NormExprs);
detectionConf = detectionConf(mask, :);
```

Importing Feature Metadata from a BeadChip Annotation File

Microarray manufacturers usually provide annotations of a collection of features for each type of chip. The chip annotation files contain metadata such as the gene name, symbol, NCBI accession number, chromosome location and pathway information. Before assembling an `ExpressionSet` object for the experiment, obtain the annotations about the features or probes on the BeadChip. You can download the `Human_WG-6.csv` annotation file for Sentrix Human 6 (WG6) BeadChips from the Support page at the Illumina web site and save the file locally. Read the annotation file into MATLAB as a `dataset` array. Set the variable `annotPath` to the path and directory to which you downloaded the annotation file.

```
annotPath = fullfile('C:\Examples\illuminagedemo\Annotation');
WG6Annot = dataset('xlsfile', fullfile(annotPath, 'Human_WG-6.csv'));
```

Inspect the properties of this `dataset` array.

```
get(WG6Annot)
```

```
  Description: ''
  VarDescription: {}
    Units: {}
  DimNames: {'Observations' 'Variables'}
  UserData: []
  ObsNames: {}
  VarNames: {1x13 cell}
```

Get the names of variables describing the features on the chip.

```
fDataVariables = get(WG6Annot, 'VarNames')
```

```
fDataVariables =
```

```
1x13 cell array
```

```
Columns 1 through 5
```

```
 {'Search_key'} {'Target'} {'ProbeId'} {'Gid'} {'Transcript'}
```

```
Columns 6 through 10
```

```
 {'Accession'} {'Symbol'} {'Type'} {'Start'} {'Probe_Sequence'}
```

```
Columns 11 through 13
```

```
 {'Definition'} {'Ontology'} {'Synonym'}
```

Check the number of probe target IDs in the annotation file.

```
numel(WG6Annot.Target)
```

```
ans =
```

```
47296
```

Because the expression data in this example is only a small set of the full expression values, you will work with only the features in the `DataMatrix` object `log2NormExprs`. Find the matching features in `log2NormExprs` and `WG6Annot.Target`.

```
[commTargets, fI, WGI] = intersect(rownames(log2NormExprs), WG6Annot.Target);
```

Building an ExpressionSet Object for Experimental Data

You can store the preprocessed expression values and detection limits of the annotated probe targets as an `ExptData` object.

```
fNames = rownames(log2NormExprs);
TNExptData = bioma.data.ExptData({log2NormExprs(fI, :), 'ExprsValues'},...
                                {detectionConf(fI, :), 'DetectionConfidences'})
```

```
TNExptData =
```

```
Experiment Data:
 42313 features, 13 samples
 2 elements
Element names: ExprsValues, DetectionConfidences
```

Building an ExpressionSet Object for Sample Information

The sample data in the `Header.Samples` field of the `TNGEOData` structure can be overwhelming and difficult to navigate through. From the data in `Header.Samples` field, you can gather the essential information about the samples, such as the sample titles, GEO sample accession numbers, etc., and store the sample data as a `MetaData` object.

Retrieve the descriptions about sample characteristics.

```
sampleChars = cellfun(@(x) char(regex(x, '\w*tile', 'match')),...
    TNGEOData.Header.Samples.characteristics_ch1, 'UniformOutput', false)
```

```
sampleChars =
 1x13 cell array

Columns 1 through 4
    {'Infertile'}    {'Infertile'}    {'Infertile'}    {'Infertile'}

Columns 5 through 8
    {'Infertile'}    {'Fertile'}     {'Infertile'}    {'Infertile'}

Columns 9 through 13
    {'Infertile'}    {'Fertile'}     {'Fertile'}     {'Fertile'}     {'Fertile'}
```

Create a `dataset` array to store the sample data you just extracted.

```
sampleDS = dataset({TNGEOData.Header.Samples.geo_accession', 'GSM'},...
    {strtok(TNGEOData.Header.Samples.title)', 'Type'},...
    {sampleChars', 'Characteristics'}, 'obsnames', sampleLabels')
```

```
sampleDS =
      GSM                                Type                                Characteristics
T2      {'GSM160620'}                   {'Teratozoospermic'}                {'Infertile'}
T1      {'GSM160621'}                   {'Teratozoospermic'}                {'Infertile'}
T6      {'GSM160622'}                   {'Teratozoospermic'}                {'Infertile'}
T4      {'GSM160623'}                   {'Teratozoospermic'}                {'Infertile'}
T8      {'GSM160624'}                   {'Teratozoospermic'}                {'Infertile'}
N11     {'GSM160625'}                   {'Normospermic'}                    {'Fertile'}
T3      {'GSM160626'}                   {'Teratozoospermic'}                {'Infertile'}
T7      {'GSM160627'}                   {'Teratozoospermic'}                {'Infertile'}
T5      {'GSM160628'}                   {'Teratozoospermic'}                {'Infertile'}
N6      {'GSM160629'}                   {'Normospermic'}                    {'Fertile'}
N12     {'GSM160630'}                   {'Normospermic'}                    {'Fertile'}
N5      {'GSM160631'}                   {'Normospermic'}                    {'Fertile'}
```

```
N1      {'GSM160632'}      {'Normospermic'      }      {'Fertile'      }
```

Store the sample metadata as an object of the `MetaData` class, including a short description for each variable.

```
TNSData = bioma.data.MetaData(sampleDS,...
  {'Sample GEO accession number',...
  'Spermic type',...
  'Fertility characteristics'})
```

```
TNSData =
```

```
Sample Names:
```

```
T2, T1, ...,N1 (13 total)
```

```
Variable Names and Meta Information:
```

```
VariableDescription
GSM      {'Sample GEO accession number'}
Type     {'Spermic type'      }
Characteristics {'Fertility characteristics' }
```

Building an ExpressionSet Object for Feature Annotations

The collection of feature metadata for Sentrix Human 6 BeadChips is large and diverse. Select information about features that are unique to the experiment and save the information as a `MetaData` object. Extract annotations of interest, for example, `Accession` and `Symbol`.

```
fIdx = ismember(fDataVariables, {'Accession', 'Symbol'});
```

```
featureAnnot = WG6Annot(WGI, fDataVariables(fIdx));
featureAnnot = set(featureAnnot, 'ObsNames', WG6Annot.Target(WGI));
```

Create a `MetaData` object for the feature annotation information with brief descriptions about the two variables of the metadata.

```
WG6FData = bioma.data.MetaData(featureAnnot, ...
  {'Accession number of probe target', 'Gene Symbol of probe target'})
```

```
WG6FData =
```

```
Sample Names:
```

```
GI_10047089-S, GI_10047091-S, ...,hmm9988-S (42313 total)
```

```
Variable Names and Meta Information:
```

```
VariableDescription
Accession {'Accession number of probe target'}
Symbol   {'Gene Symbol of probe target'      }
```

Building an ExpressionSet Object for Experiment Information

Most of the experiment descriptions in the `Header.Series` field of the `TNGEOData` structure can be reorganized and stored as a `MIAME` object, which you will use to assemble the `ExpressionSet` object for the experiment.

```
TNExptInfo = bioma.data.MIAME(TNGEOData)
```

```
TNExptInfo =  
  
Experiment Description:  
  Author name: Adrian,E,Platts  
David,J,Dix  
Hector,E,Chemes  
Kary,E,Thompson  
Robert,,Goodrich  
John,C,Rockett  
Vanesa,Y,Rawe  
Silvina,,Quintana  
Michael,P,Diamond  
Lillian,F,Strader  
Stephen,A,Krawetz  
  Laboratory: Wayne State University  
  Contact information: Stephen,A,Krawetz  
  URL: http://compbio.med.wayne.edu  
  PubMedIDs: 17327269  
  Abstract: A 82 word abstract is available. Use the Abstract property.  
  Experiment Design: A 61 word summary is available. Use the ExptDesign property.  
  Other notes:  
    {'ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE6nnn/GSE6967/suppl/GSE6967_RAW.tar'}
```

Assembling an ExpressionSet Object

Now that you've created all the components, you can create an object of the ExpressionSet class to store the expression values, sample information, chip feature annotations and description information about this experiment.

```
TNExprSet = bioma.ExpressionSet(TNExptData, 'sData', TNSData,...  
                                'fData', WG6FData,...  
                                'eInfo', TNExptInfo)
```

```
TNExprSet =  
  
ExpressionSet  
Experiment Data: 42313 features, 13 samples  
  Element names: Expressions, DetectionConfidences  
Sample Data:  
  Sample names:      T2, T1, ...,N1 (13 total)  
  Sample variable names and meta information:  
    GSM: Sample GEO accession number  
    Type: Spermic type  
    Characteristics: Fertility characteristics  
Feature Data:  
  Feature names:      GI_10047089-S, GI_10047091-S, ...,hmm9988-S (42313 total)  
  Feature variable names and meta information:  
    Accession: Accession number of probe target  
    Symbol: Gene Symbol of probe target  
Experiment Information: use 'exptInfo(obj)'
```

Note: The ExprsValues element in the ExptData object, TNExptData, is renamed to Expressions in TNGeneExprSet.

You can save an object of ExpressionSet class as a MAT file for further data analysis.

```
save TNGeneExprSet TNEExprSet
```

Profiling Gene Expression by Using Permutation T-tests

Load the experiment data saved from the previous section. You will use this data to find differentially expressed genes between the teratozoospermia and normal samples.

```
load TNGeneExprSet
```

To identify the differential changes in the levels of transcripts in normospermic Ns and teratozoospermic Tz samples, compare the gene expression values between the two groups of data: Tz and Ns.

```
TNSamples = sampleNames(TNEExprSet);
Tz = strncmp(TNSamples, 'T', 1);
Ns = strncmp(TNSamples, 'N', 1);
nTz = sum(Tz)
nNs = sum(Ns)

TNEExprs = expressions(TNEExprSet);
TzData = TNEExprs(:,Tz);
NsData = TNEExprs(:,Ns);
meanTzData = mean(TzData,2);
meanNsData = mean(NsData,2);
groupLabels = [TNSamples(Tz), TNSamples(Ns)];
```

```
nTz =
      8
```

```
nNs =
      5
```

Perform a permutation t-test using the `mattest` function to permute the columns of the gene expression data matrix of `TzData` and `NsData`. Note: Depending on the sample size, it may not be feasible to consider all possible permutations. Usually, a random subset of permutations are considered in the case of a large sample size.

Use the `nchoosek` function in Statistics and Machine Learning Toolbox™ to determine the number of all possible permutations of the samples in this example.

```
perms = nchoosek(1:nTz+nNs, nTz);
nPerms = size(perms,1)
```

```
nPerms =
      1287
```

Use the `PERMUTE` option of the `mattest` function to compute the p-values of all the permutations.

```
pValues = mattest(TzData, NsData, 'Permute', nPerms);
```

You can also compute the differential score from the p-values using the following anonymous function [1].

```
diffscore = @(p, avgTest, avgRef) -10*sign(avgTest - avgRef).*log10(p);
```

A differential score of 13 corresponds to a p-value of 0.05, a differential score of 20 corresponds to a p-value of 0.01, and a differential score of 30 corresponds to a p-value of 0.001. A positive differential score represents up-regulation, while a negative score represents down-regulation of genes.

```
diffScores = diffscore(pValues, meanTzData, meanNsData);
```

Determine the number of genes considered to have a differential score greater than 20. Note: You may get a different number of genes due to the permutation test outcome.

```
up = sum(diffScores > 20)
```

```
down = sum(diffScores < -20)
```

```
up =
```

```
    3741
```

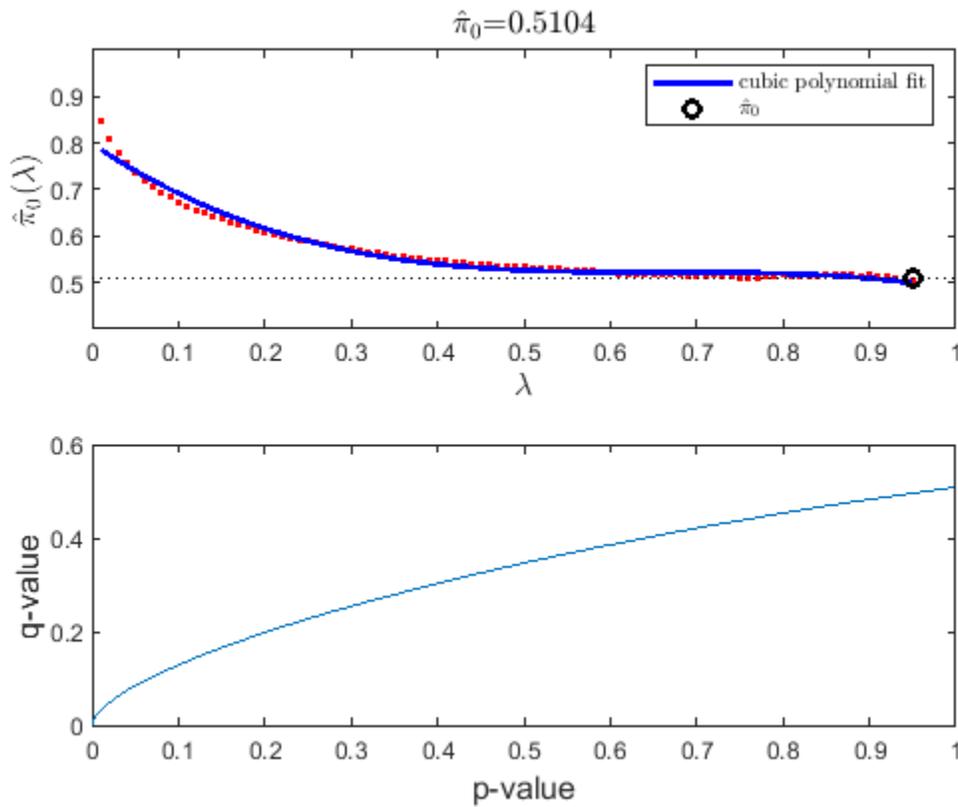
```
down =
```

```
    3033
```

Estimating False Discovery Rate (FDR)

In multiple hypothesis testing, where we simultaneously tests the null hypothesis of thousands of genes, each test has a specific false positive rate, or a false discovery rate (FDR) [2]. Estimate the FDR and q-values for each test using the `mafdr` function.

```
figure;  
[pFDR, qValues] = mafdr(pValues, 'showplot', true);  
diffScoresFDRQ = diffscore(qValues, meanTzData, meanNsData);
```



Determine the number of genes with an absolute differential score greater than 20. Note: You may get a different number of genes due to the permutation test and the bootstrap outcomes.

```
sum(abs(diffScoresFDRQ)>=20)
```

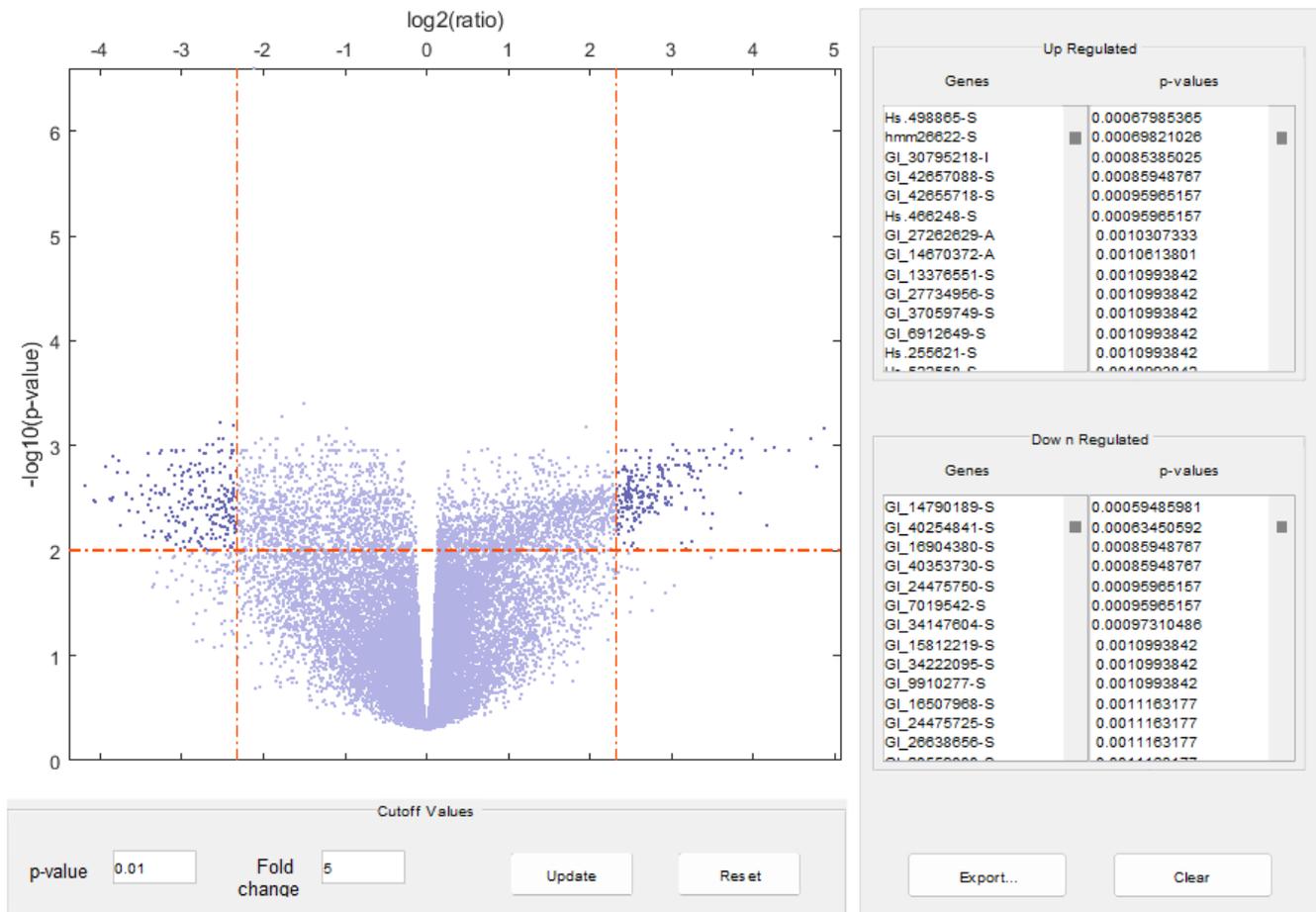
```
ans =
```

```
3122
```

Identifying Genes that Are Differentially Expressed

Plot the $-\log_{10}$ of p-values against fold changes in a volcano plot.

```
diffStruct = mavolcanoplot(TzData, NsData, qValues, ...
                          'pcutoff', 0.01, 'foldchange', 5);
```



Note: From the volcano plot UI, you can interactively change the p-value cutoff and fold-change limit, and export differentially expressed genes.

Determine the number of differentially expressed genes.

```
nDiffGenes = numel(diffStruct.GeneLabels)
```

```
nDiffGenes =
```

```
451
```

Get the list of up-regulated genes for the Tz samples compared to the Ns samples.

```
up_genes = diffStruct.GeneLabels(diffStruct.FoldChanges > 0);
nUpGenes = length(up_genes)
```

```
nUpGenes =
```

```
223
```

Get the list of down-regulated genes for the Tz samples compared to the Ns samples.

```
down_genes = diffStruct.GeneLabels(diffStruct.FoldChanges < 0);
nDownGenes = length(down_genes)
```

```
nDownGenes =
    228
```

Extract a list of differentially expressed genes.

```
diff_geneidx = zeros(nDiffGenes, 1);
for i = 1:nDiffGenes
    diff_geneidx(i) = find(strncmpi(TNExprSet.featureNames, ...
        diffStruct.GeneLabels{i}, length(diffStruct.GeneLabels{i})), 1);
end
```

You can get the subset of experiment data containing only the differentially expressed genes.

```
TNDiffExprSet = TNExprSet(diff_geneidx, groupLabels);
```

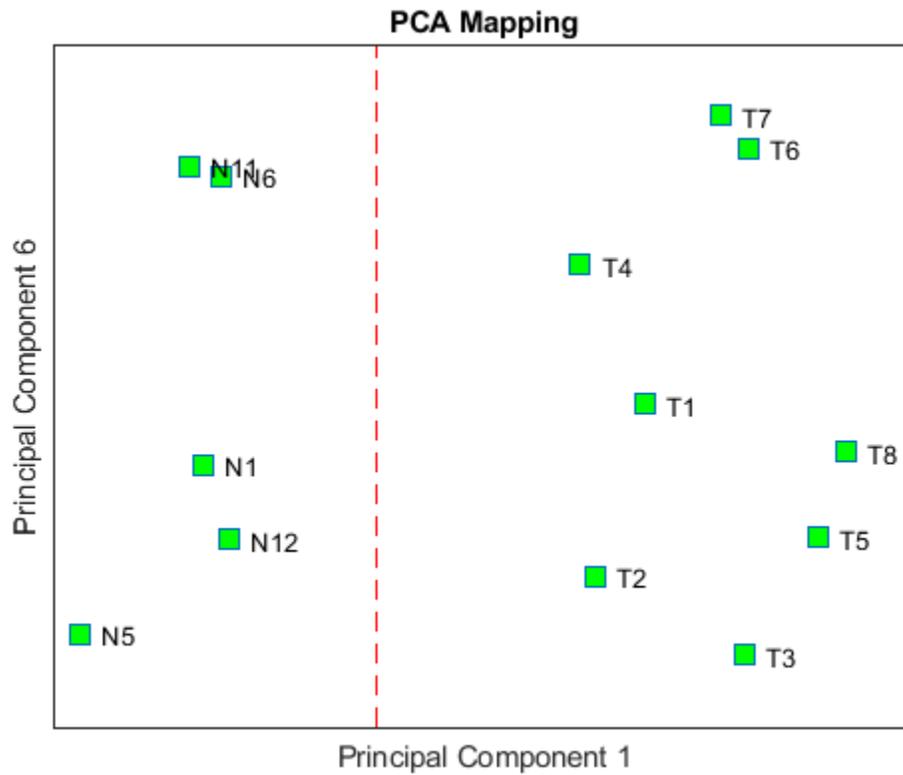
Performing PCA and Clustering Analysis of Significant Gene Profiles

Principal component analysis (PCA) on differentially expressed genes shows linear separability of the Tz samples from the Ns samples.

```
PCAScore = pca(TNDiffExprSet.expressions);
```

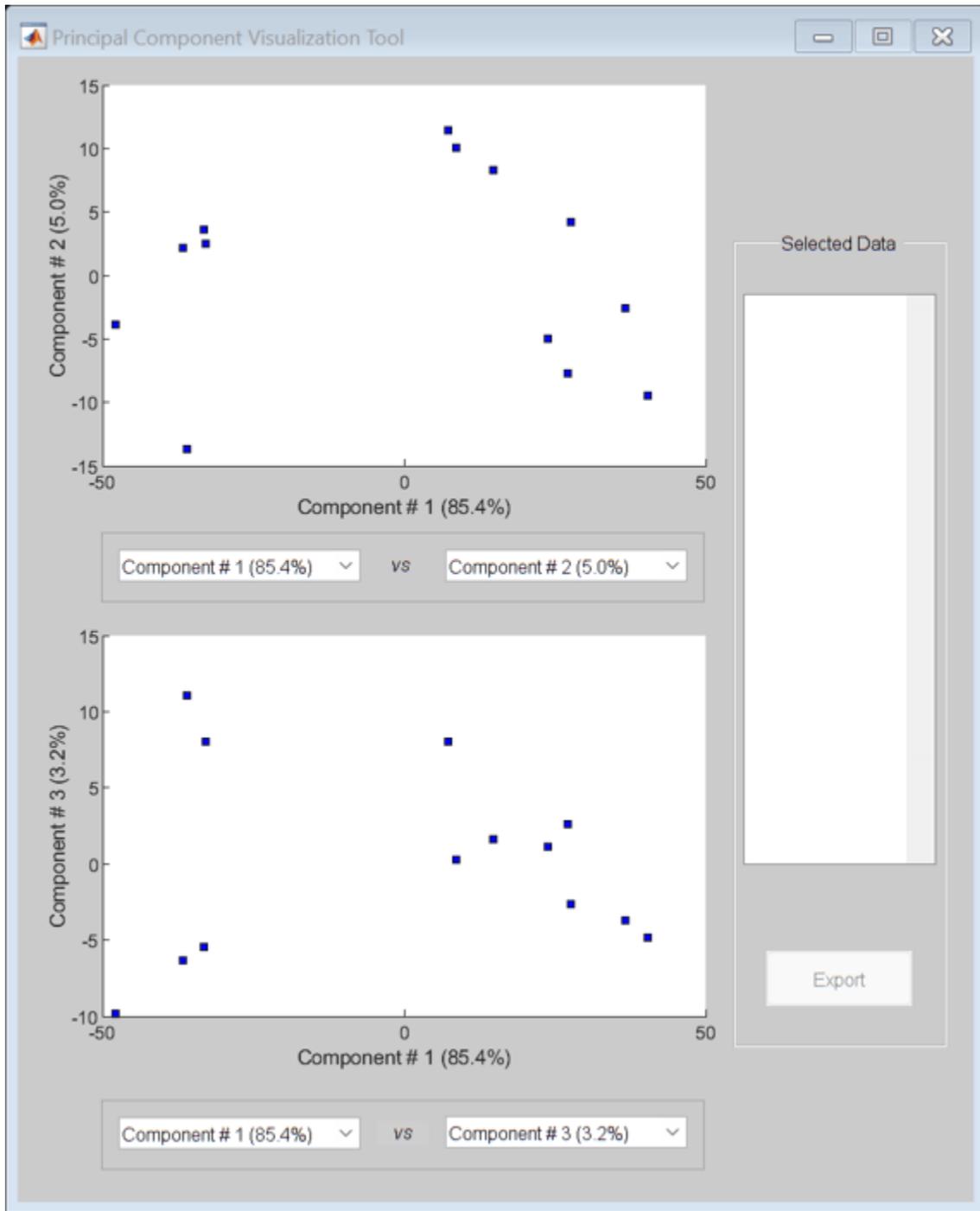
Display the coefficients of the first and sixth principal components.

```
figure;
plot(PCAScore(:,1), PCAScore(:,6), 's', 'MarkerSize',10, 'MarkerFaceColor','g');
hold on
text(PCAScore(:,1)+0.02, PCAScore(:,6), TNDiffExprSet.sampleNames)
plot([0,0], [-0.5 0.5], '--r')
ax = gca;
ax.XTick = [];
ax.YTick = [];
ax.YTickLabel = [];
title('PCA Mapping')
xlabel('Principal Component 1')
ylabel('Principal Component 6')
```



You can also use the interactive tool created by the `mapcaplot` function to perform principal component analysis.

```
mapcaplot((TNDiffExprSet.expressions)')
```

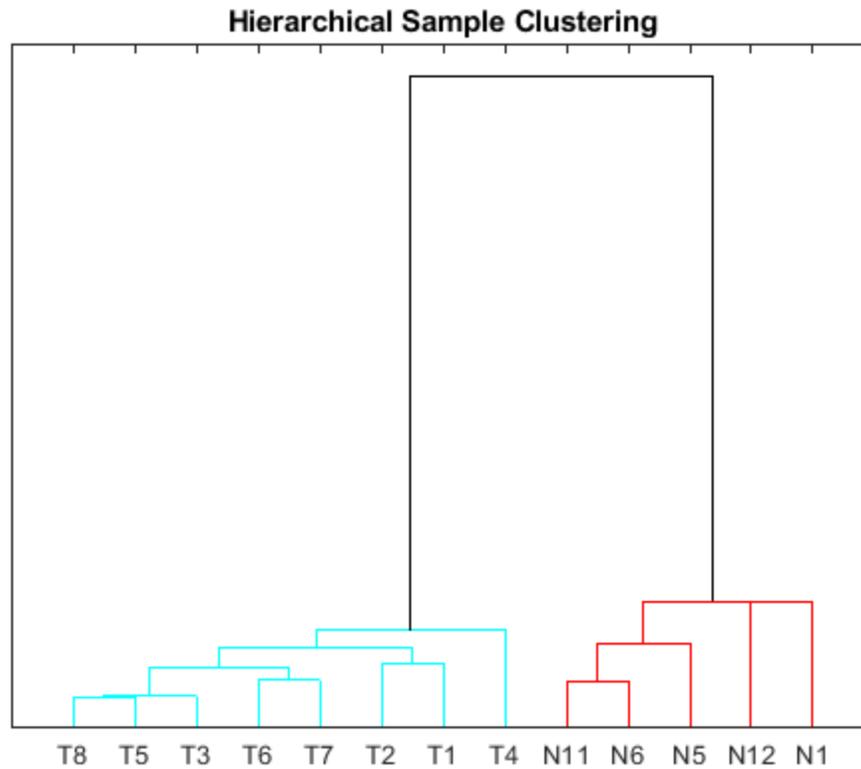


Perform unsupervised hierarchical clustering of the significant gene profiles from the Tz and Ns groups using correlation as the distance metric to cluster the samples.

```
sampleDist = pdist(TNDiffExprSet.expressions', 'correlation');
sampleLink = linkage(sampleDist);
```

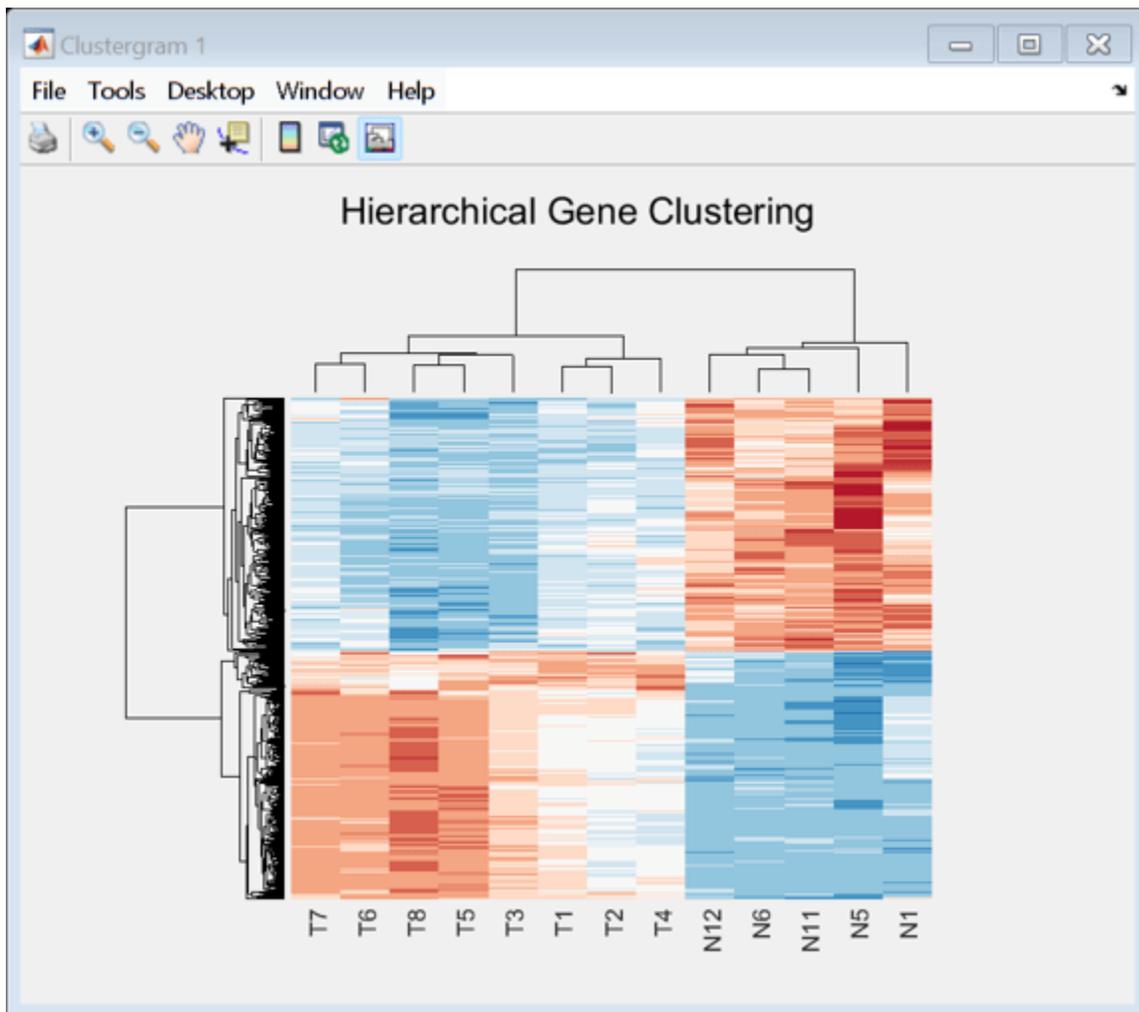
```
figure;
dendrogram(sampleLink, 'labels', TNDiffExprSet.sampleNames, 'ColorThreshold', 0.5)
```

```
ax = gca;  
ax.YTick = [];  
ax.Box = 'on';  
title('Hierarchical Sample Clustering')
```



Use the `clustergram` function to create the hierarchical clustering of differentially expressed genes, and apply the colormap `redbluecmap` to the clustergram.

```
cmap = redbluecmap(9);  
cg = clustergram(TNDiffExprSet.expressions, 'Colormap', cmap, 'Standardize', 2);  
addTitle(cg, 'Hierarchical Gene Clustering')
```



Clustering of the most differentially abundant transcripts clearly partitions teratozoospermic (Tz) and normospermic (Ns) spermatozoal RNAs.

References

- [1] Platts, A.E., et al., "Success and failure in human spermatogenesis as revealed by teratozoospermic RNAs", *Human Molecular Genetics*, 16(7):763-73, 2007.
- [2] Storey, J.D. and Tibshirani, R., "Statistical significance for genomewide studies", *PNAS*, 100(16):9440-5, 2003.

Detecting DNA Copy Number Alteration in Array-Based CGH Data

This example shows how to detect DNA copy number alterations in genome-wide array-based comparative genomic hybridization (CGH) data.

Introduction

Copy number changes or alterations is a form of genetic variation in the human genome [1]. DNA copy number alterations (CNAs) have been linked to the development and progression of cancer and many diseases.

DNA microarray based comparative genomic hybridization (CGH) is a technique allows simultaneous monitoring of copy number of thousands of genes throughout the genome [2,3]. In this technique, DNA fragments or "clones" from a test sample and a reference sample differentially labeled with dyes (typically, Cy3 and Cy5) are hybridized to mapped DNA microarrays and imaged. Copy number alterations are related to the Cy3 and Cy5 fluorescence intensity ratio of the targets hybridized to each probe on a microarray. Clones with normalized test intensities significantly greater than reference intensities indicate copy number gains in the test sample at those positions. Similarly, significantly lower intensities in the test sample are signs of copy number loss. BAC (bacterial artificial chromosome) clone based CGH arrays have a resolution in the order of one million base pairs (1Mb) [3]. Oligonucleotide and cDNA arrays provide a higher resolution of 50-100kb [2].

Array CGH log₂-based intensity ratios provide useful information about genome-wide CNAs. In humans, the normal DNA copy number is two for all the autosomes. In an ideal situation, the normal clones would correspond to a log₂ ratio of zero. The log₂ intensity ratios of a single copy loss would be -1, and a single copy gain would be 0.58. The goal is to effectively identify locations of gains or losses of DNA copy number.

The data in this example is the Coriell cell line BAC array CGH data analyzed by Snijders et al.(2001). The Coriell cell line data is widely regarded as a "gold standard" data set. You can download this data of normalized log₂-based intensity ratios and the supplemental table of known karyotypes from <https://www.nature.com/articles/ng754#supplementary-information>. You will compare these cytogenetically mapped alterations with the locations of gains or losses identified with various functions of MATLAB and its toolboxes.

For this example, the Coriell cell line data are provided in a MAT file. The data file `coriell_baccgh.mat` contains `coriell_data`, a structure containing of the normalized average of the log₂-based test to reference intensity ratios of 15 fibroblast cell lines and their genomic positions. The BAC targets are ordered by genome position beginning at *1p* and ending at *Xq*.

```
load coriell_baccgh
coriell_data

coriell_data =

  struct with fields:

      Sample: {1×15 cell}
      Chromosome: [2285×1 int8]
      GenomicPosition: [2285×1 int32]
      Log2Ratio: [2285×15 double]
```

```
FISHMap: {2285x1 cell}
```

Visualizing the Genome Profile of the Array CGH Data Set

You can plot the genome wide log₂-based test/reference intensity ratios of DNA clones. In this example, you will display the log₂ intensity ratios for cell line GM03576 for chromosomes 1 through 23.

Find the sample index for the GM03576 cell line.

```
sample = find(strcmpi(coriell_data.Sample, 'GM03576'))
```

```
sample =
```

```
8
```

To label chromosomes and draw the chromosome borders, you need to find the number of data points of in each chromosome.

```
chr_nums = zeros(1, 23);
chr_data_len = zeros(1,23);
for c = 1:23
    tmp = coriell_data.Chromosome == c;
    chr_nums(c) = find(tmp, 1, 'last');
    chr_data_len(c) = length(find(tmp));
end

% Draw a vertical bar at the end of a chromosome to indicate the border
x_vbar = repmat(chr_nums, 3, 1);
y_vbar = repmat([2;-2;NaN], 1, 23);

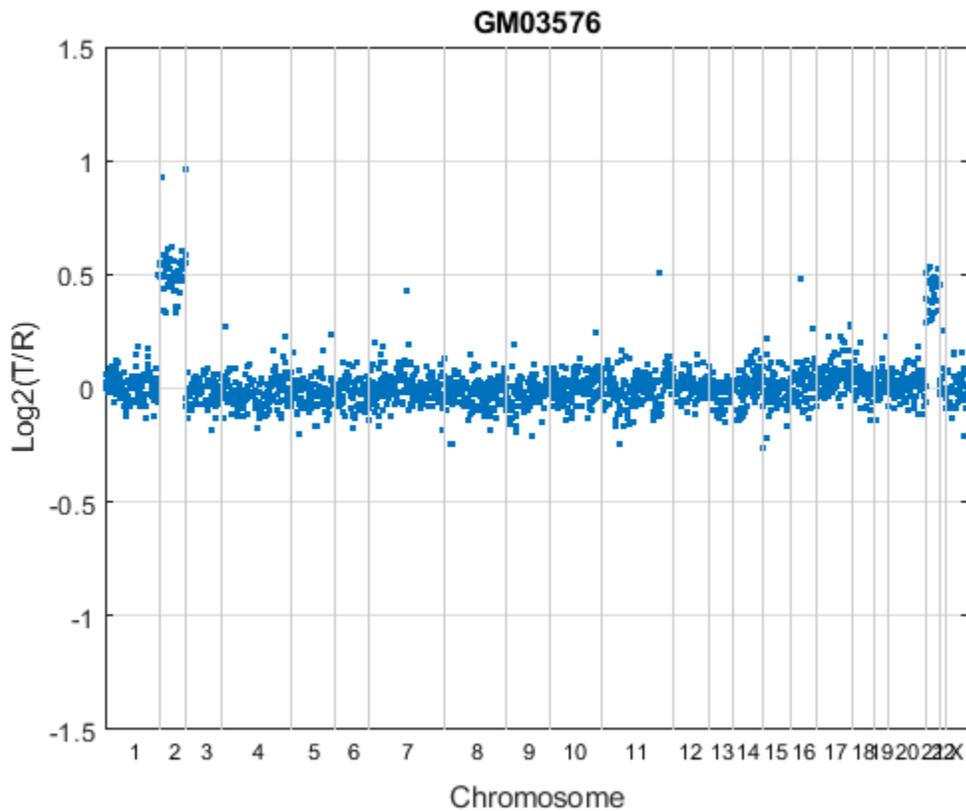
% Label the autosomes with their chromosome numbers, and the sex chromosome
% with X.
x_label = chr_nums - ceil(chr_data_len/2);
y_label = zeros(1, length(x_label)) - 1.6;
chr_labels = num2str((1:1:23)');
chr_labels = cellstr(chr_labels);
chr_labels{end} = 'X';

figure
hold on
h_ratio = plot(coriell_data.Log2Ratio(:,sample), '.');
h_vbar = line(x_vbar, y_vbar, 'color', [0.8 0.8 0.8]);
h_text = text(x_label, y_label, chr_labels,...
             'fontsize', 8, 'HorizontalAlignment', 'center');

h_axis = h_ratio.Parent;
h_axis.XTick = [];
h_axis.YGrid = 'on';
h_axis.Box = 'on';
xlim([0 chr_nums(23)])
ylim([-1.5 1.5])

title(coriell_data.Sample{sample})
xlabel({'', 'Chromosome'})
```

```
ylabel('Log2(T/R)')
hold off
```



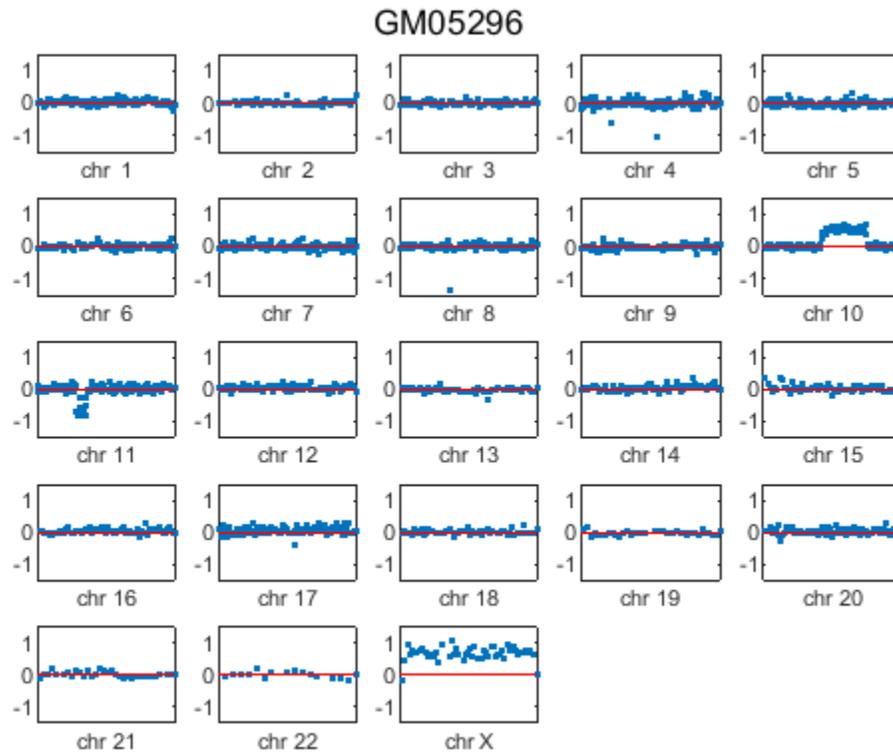
In the plot, borders between chromosomes are indicated by grey vertical bars. The plot indicates that the GM03576 cell line is trisomic for chromosomes 2 and 21 [3].

You can also plot the profile of each chromosome in a genome. In this example, you will display the log₂ intensity ratios for each chromosome in cell line GM05296 individually.

```
sample = find(strcmpi(coriell_data.Sample, 'GM05296'));
figure;
for c = 1:23
    idx = coriell_data.Chromosome == c;
    chr_y = coriell_data.Log2Ratio(idx, sample);
    subplot(5,5,c);

    hp = plot(chr_y, '.');
    line([0, chr_data_len(c)], [0,0], 'color', 'r');

    h_axis = hp.Parent;
    h_axis.XTick = [];
    h_axis.Box = 'on';
    xlim([0 chr_data_len(c)])
    ylim([-1.5 1.5])
    xlabel(['chr ' chr_labels{c}], 'FontSize', 8)
end
sgtitle('GM05296');
```



The plot indicates the GM05296 cell line has a partial trisomy at chromosome 10 and a partial monosomy at chromosome 11.

Observe that the gains and losses of copy number are discrete. These alterations occur in contiguous regions of a chromosome that cover several clones to entitle chromosome.

The array-based CGH data can be quite noisy. Therefore, accurate identification of chromosome regions of equal copy number that accounts for the noise in the data requires robust computational methods. In the rest of this example, you will work with the data of chromosomes 9, 10 and 11 of the GM05296 cell line.

Initialize a structure array for the data of these three chromosomes.

```
GM05296_Data = struct('Chromosome', {9 10 11},...
                    'GenomicPosition', {[], [], []},...
                    'Log2Ratio', {[], [], []},...
                    'SmoothedRatio', {[], [], []},...
                    'DiffRatio', {[], [], []},...
                    'SegIndex', {[], [], []});
```

Filtering and Smoothing Data

A simple approach to perform high-level smoothing is to use a nonparametric filter. The function `mslowess` implements a linear fit to samples within a shifting window, in this example you use a SPAN of 15 samples.

```
for iloop = 1:length(GM05296_Data)
```

```

idx = coriell_data.Chromosome == GM05296_Data(iloop).Chromosome;
chr_x = coriell_data.GenomicPosition(idx);
chr_y = coriell_data.Log2Ratio(idx, sample);

% Remove NaN data points
idx = ~isnan(chr_y);
GM05296_Data(iloop).GenomicPosition = double(chr_x(idx));
GM05296_Data(iloop).Log2Ratio = chr_y(idx);

% Smoother
GM05296_Data(iloop).SmoothedRatio = ...
    mslowess(GM05296_Data(iloop).GenomicPosition,...
             GM05296_Data(iloop).Log2Ratio,...
             'SPAN',15);

% Find the derivative of the smoothed ratio
GM05296_Data(iloop).DiffRatio = ...
    diff([0; GM05296_Data(iloop).SmoothedRatio]);
end

```

To better visualize and later validate the locations of copy number changes, we need cytoband information. Read the human cytoband information from the `hs_cytoBand.txt` data file using the `cytobandread` function. It returns a structure of human cytoband information [4].

```

hs_cytobands = cytobandread('hs_cytoBand.txt')

% Find the centromere positions for the chromosomes.
acen_idx = strcmpi(hs_cytobands.GieStains, 'acen');
acen_ends = hs_cytobands.BandEndBPs(acen_idx);

% Convert the cytoband data from bp to kilo bp because the genomic
% positions in Coriell Cell Line data set are in kilo base pairs.
acen_pos = acen_ends(1:2:end)/1000;

```

```
hs_cytobands =
```

```
struct with fields:
```

```

ChromLabels: {862x1 cell}
BandStartBPs: [862x1 int32]
BandEndBPs: [862x1 int32]
BandLabels: {862x1 cell}
GieStains: {862x1 cell}

```

You can inspect the data by plotting the log₂-based ratios, the smoothed ratios and the derivative of the smoothed ratios together. You can also display the centromere position of a chromosome in the data plots. The magenta vertical bar marks the centromere of the chromosome.

```

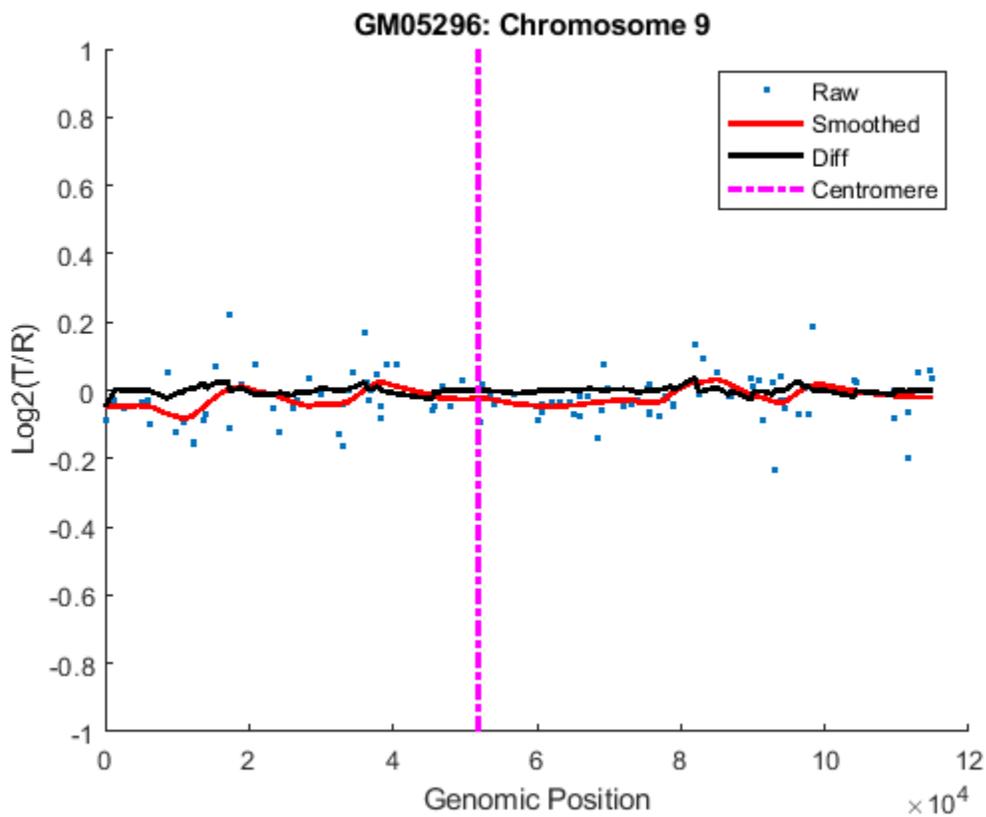
for iloop = 1:length(GM05296_Data)
chr = GM05296_Data(iloop).Chromosome;
chr_x = GM05296_Data(iloop).GenomicPosition;
figure
hold on
plot(chr_x, GM05296_Data(iloop).Log2Ratio, '.');
line(chr_x, GM05296_Data(iloop).SmoothedRatio,...

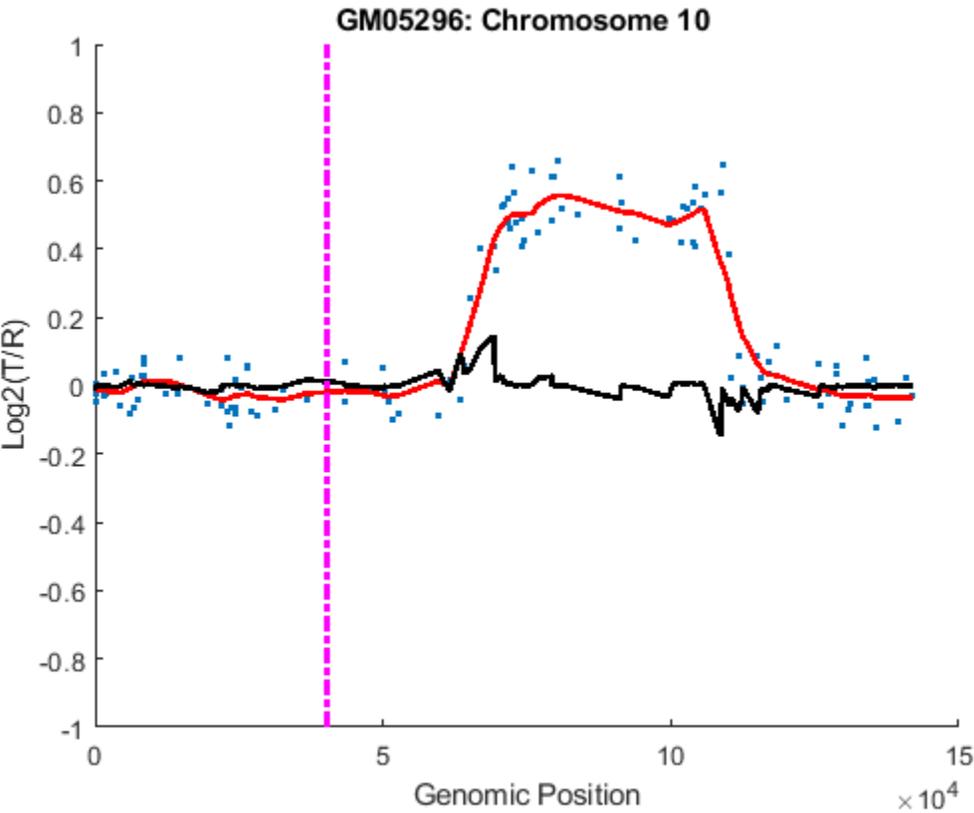
```

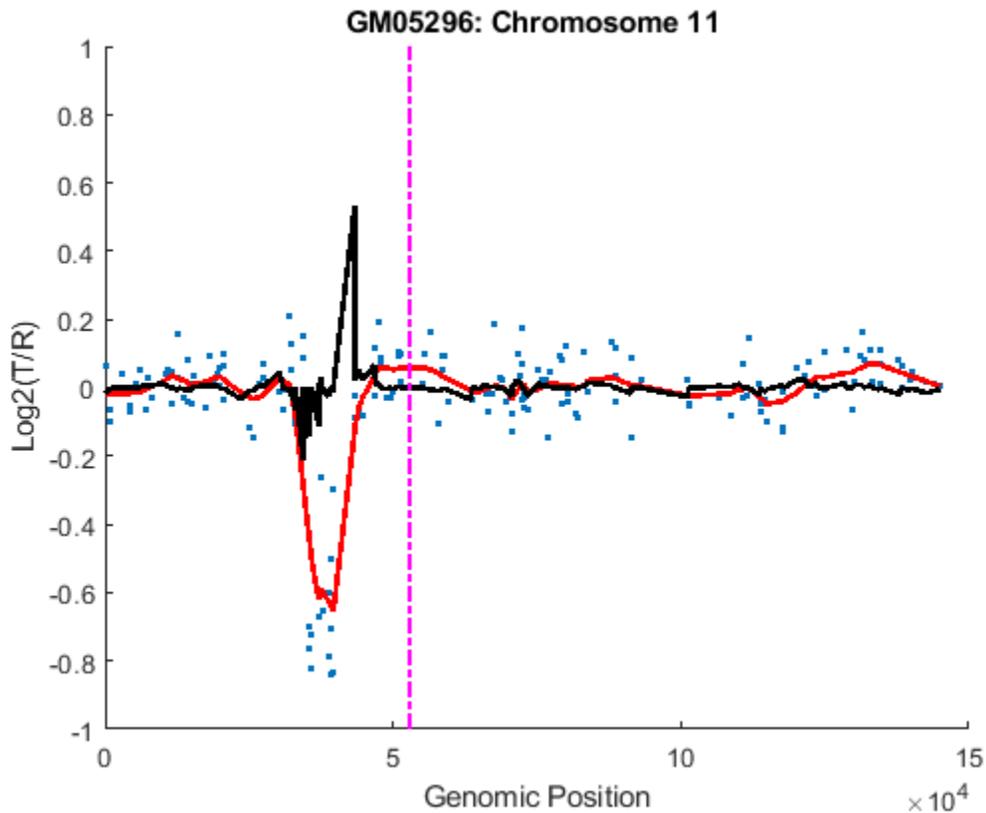
```

        'Color', 'r', 'LineWidth', 2);
line(chr_x, GM05296_Data(iloop).DiffRatio,...
      'Color', 'k', 'LineWidth', 2);
line([acen_pos(chr), acen_pos(chr)], [-1, 1],...
      'Color', 'm', 'LineWidth', 2, 'LineStyle', '-.');
if iloop == 1
    legend('Raw', 'Smoothed', 'Diff', 'Centromere');
end
ylim([-1, 1])
xlabel('Genomic Position')
ylabel('Log2(T/R)')
title(sprintf('GM05296: Chromosome %d ', chr))
hold off
end

```







Detecting Change-Points

The derivatives of the smoothed ratio over a certain threshold usually indicate substantial changes with large peaks, and provide the estimate of the change-point indices. For this example you will select a threshold of 0.1.

```
thrd = 0.1;

for iloop = 1:length(GM05296_Data)
    idx = find(abs(GM05296_Data(iloop).DiffRatio) > thrd );
    N = numel(GM05296_Data(iloop).SmoothedRatio);
    GM05296_Data(iloop).SegIndex = [1;idx;N];

    % Number of possible segments found
    fprintf('%d segments initially found on Chromosome %d.\n',...
           numel(GM05296_Data(iloop).SegIndex) - 1,...
           GM05296_Data(iloop).Chromosome)
end
```

```
1 segments initially found on Chromosome 9.
4 segments initially found on Chromosome 10.
5 segments initially found on Chromosome 11.
```

Optimizing Change-Points by GM Clustering

Gaussian Mixture (GM) or Expectation-Maximization (EM) clustering can provide fine adjustments to the change-point indices [5]. The convergence to statistically optimal change-point indices can be

facilitated by surrounding each index with equal-length set of adjacent indices. Thus each edge is associated with left and right distributions. The GM clustering learns the maximum-likelihood parameters of the two distributions. It then optimally adjusts the indices given the learned parameters.

You can set the length for the set of adjacent positions distributed around the change-point indices. For this example, you will select a length of 5. You can also inspect each change-point by plotting its GM clusters. In this example, you will plot the GM clusters for the Chromosome 10 data.

```
len = 5;
for iloop = 1:length(GM05296_Data)
    seg_num = numel(GM05296_Data(iloop).SegIndex) - 1;
    if seg_num > 1
        % Plot the data points in chromosome 10 data
        if GM05296_Data(iloop).Chromosome == 10
            figure
            hold on;
            plot(GM05296_Data(iloop).GenomicPosition,...
                GM05296_Data(iloop).Log2Ratio, '.')
            ylim([-0.5, 1])
            xlabel('Genomic Position')
            ylabel('Log2(T/R)')
            title(sprintf('Chromosome %d - GM05296', ...
                GM05296_Data(iloop).Chromosome))
        end

        segidx = GM05296_Data(iloop).SegIndex;
        segidx_emadj = GM05296_Data(iloop).SegIndex;

        for jloop = 2:seg_num
            ileft = min(segidx(jloop) - len, segidx(jloop));
            irect = max(segidx(jloop) + len, segidx(jloop));
            gmx = GM05296_Data(iloop).GenomicPosition(ileft:irect);
            gmy = GM05296_Data(iloop).SmoothedRatio(ileft:irect);

            % Select initial guess for the cluster index for each point.
            gmpart = (gmy > (min(gmy) + range(gmy)/2)) + 1;

            % Create a Gaussian mixture model object
            gm = gmdistribution.fit(gmy, 2, 'start', gmpart);
            gmid = cluster(gm,gmy);

            segidx_emadj(jloop) = find(abs(diff(gmid))==1) + ileft;

            % Plot GM clusters for the change-points in chromosome 10 data
            if GM05296_Data(iloop).Chromosome == 10
                plot(gmx(gmid==1),gmy(gmid==1), 'g.',...
                    gmx(gmid==2), gmy(gmid==2), 'r.')
            end
        end
    end

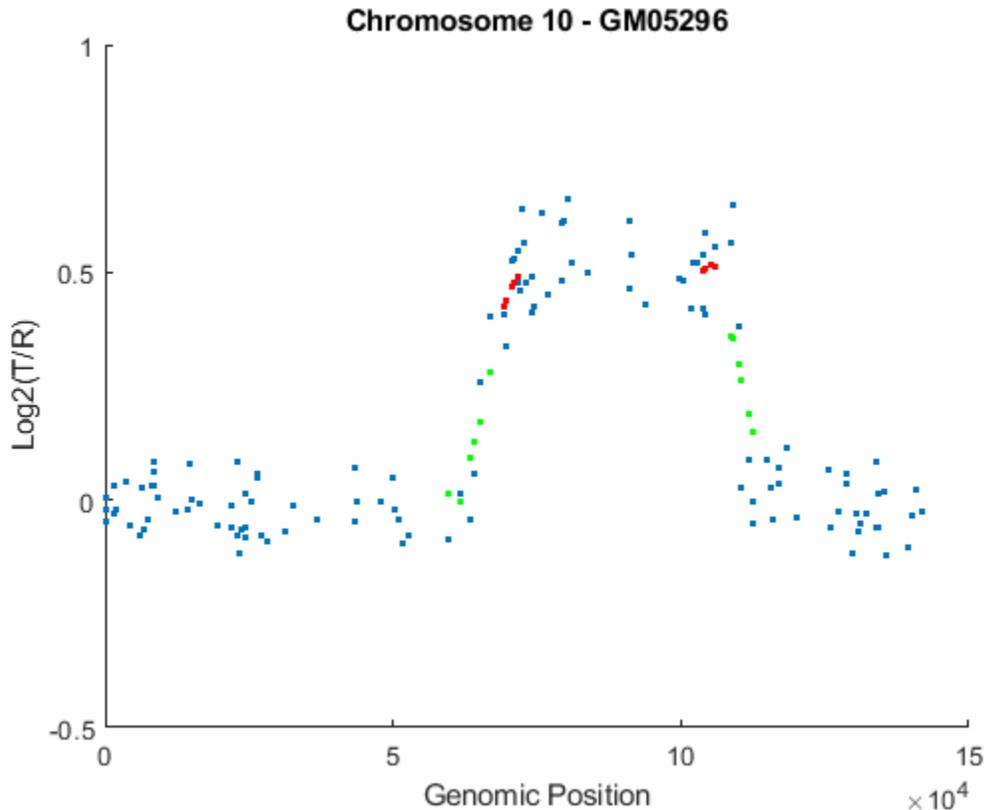
    % Remove repeat indices
    zeroidx = [diff(segidx_emadj) == 0; 0];
    GM05296_Data(iloop).SegIndex = segidx_emadj(~zeroidx);
end

% Number of possible segments found
```

```

fprintf('%d segments found on Chromosome %d after GM clustering adjustment.\n',...
        numel(GM05296_Data(iloop).SegIndex) - 1,...
        GM05296_Data(iloop).Chromosome)
end
hold off;

1 segments found on Chromosome 9 after GM clustering adjustment.
3 segments found on Chromosome 10 after GM clustering adjustment.
5 segments found on Chromosome 11 after GM clustering adjustment.
    
```



Testing Change-Point Significance

Once you determine the optimal change-point indices, you also need to determine if each segment represents a statistically significant changes in DNA copy number. You will perform permutation t-tests to assess the significance of the segments identified. A segment includes all the data points from one change-point to the next change-point or the chromosome end. In this example, you will perform 10,000 permutations of the data points on two consecutive segments along the chromosome at the significance level of 0.01.

```

alpha = 0.01;
for iloop = 1:length(GM05296_Data)
    seg_num = numel(GM05296_Data(iloop).SegIndex) - 1;
    seg_index = GM05296_Data(iloop).SegIndex;
    if seg_num > 1
        ppvals = zeros(seg_num+1, 1);

        for sloop = 1:seg_num-1
    
```

```

    seg1idx = seg_index(sloop):seg_index(sloop+1)-1;

    if sloop== seg_num-1
        seg2idx = seg_index(sloop+1):(seg_index(sloop+2));
    else
        seg2idx = seg_index(sloop+1):(seg_index(sloop+2)-1);
    end

    seg1 = GM05296_Data(iloop).SmoothedRatio(seg1idx);
    seg2 = GM05296_Data(iloop).SmoothedRatio(seg2idx);

    n1 = numel(seg1);
    n2 = numel(seg2);
    N = n1+n2;
    segs = [seg1;seg2];

    % Compute observed t statistics
    t_obs = mean(seg1) - mean(seg2);

    % Permutation test
    iter = 10000;
    t_perm = zeros(iter,1);
    for i = 1:iter
        randseg = segs(randperm(N));
        t_perm(i) = abs(mean(randseg(1:n1))-mean(randseg(n1+1:N)));
    end
    ppvals(sloop+1) = sum(t_perm >= abs(t_obs))/iter;
end

sigidx = ppvals < alpha;
GM05296_Data(iloop).SegIndex = seg_index(sigidx);
end

% Number segments after significance tests
fprintf('%d segments found on Chromosome %d after significance tests.\n',...
        numel(GM05296_Data(iloop).SegIndex) - 1, GM05296_Data(iloop).Chromosome)
end

1 segments found on Chromosome 9 after significance tests.
3 segments found on Chromosome 10 after significance tests.
4 segments found on Chromosome 11 after significance tests.

```

Assessing Copy Number Alterations

Cytogenetic study indicates cell line GM05296 has a trisomy at *10q21-10q24* and a monosomy at *11p12-11p13* [3]. Plot the segment means of the three chromosomes over the original data with bold red lines, and add the chromosome ideograms to the plots using the `chromosomeplot` function. Note that the genomic positions in the Coriell cell line data set are in kilo base pairs. Therefore, you will need to convert cytoband data from bp to kilo bp when adding the ideograms to the plot.

```

for iloop = 1:length(GM05296_Data)
    figure;
    seg_num = numel(GM05296_Data(iloop).SegIndex) - 1;
    seg_mean = ones(seg_num,1);
    chr_num = GM05296_Data(iloop).Chromosome;
    for jloop = 2:seg_num+1
        idx = GM05296_Data(iloop).SegIndex(jloop-1):GM05296_Data(iloop).SegIndex(jloop);
    end
end

```

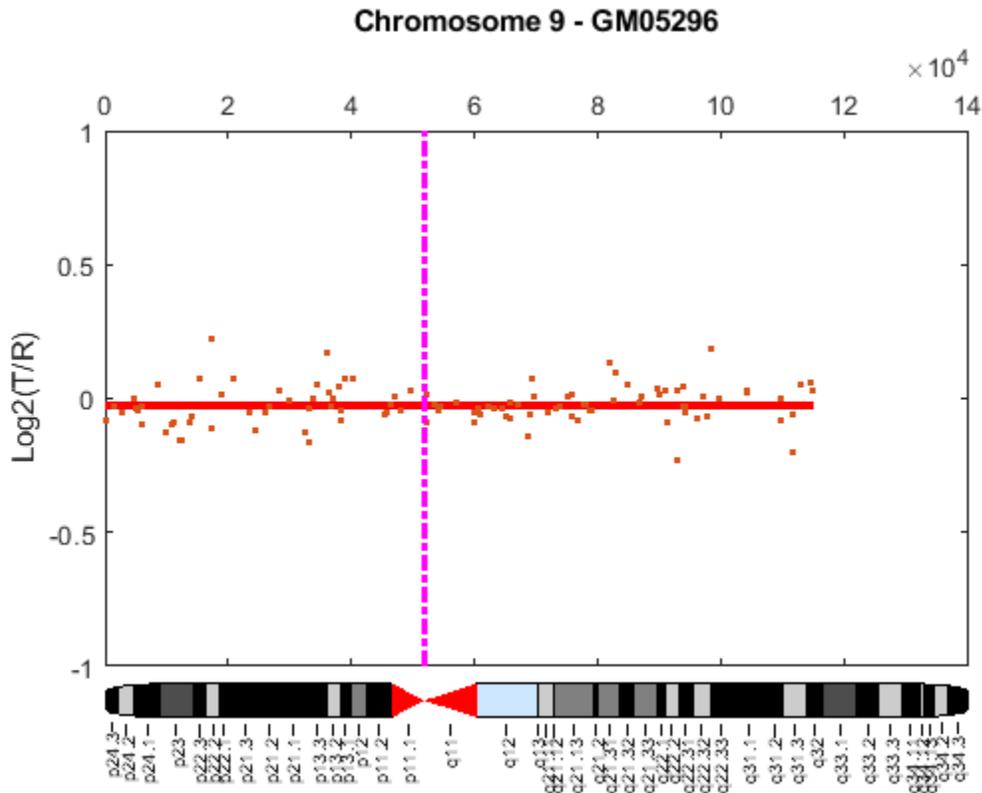
```

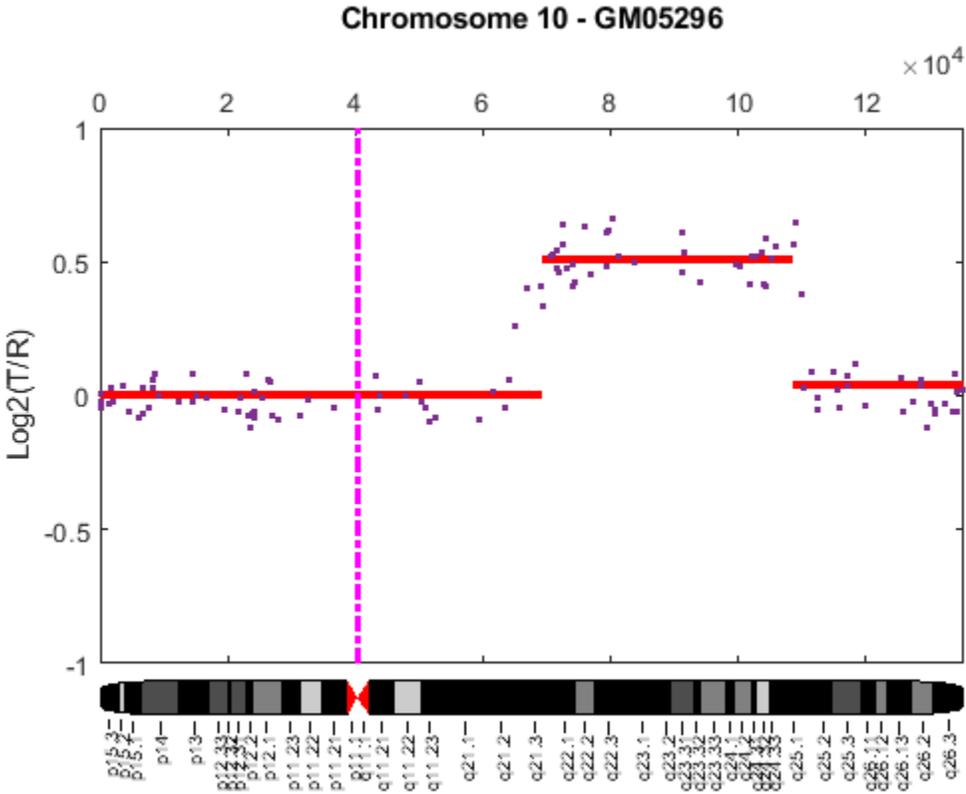
    seg_mean(idx) = mean(GM05296_Data(iloop).Log2Ratio(idx));
    line(GM05296_Data(iloop).GenomicPosition(idx), seg_mean(idx),...
        'color', 'r', 'linewidth', 3);
end
line(GM05296_Data(iloop).GenomicPosition, GM05296_Data(iloop).Log2Ratio,...
    'linestyle', 'none', 'Marker', '.');
line([acen_pos(chr_num), acen_pos(chr_num)], [-1, 1],...
    'linewidth', 2,...
    'color', 'm',...
    'linestyle', '-.');

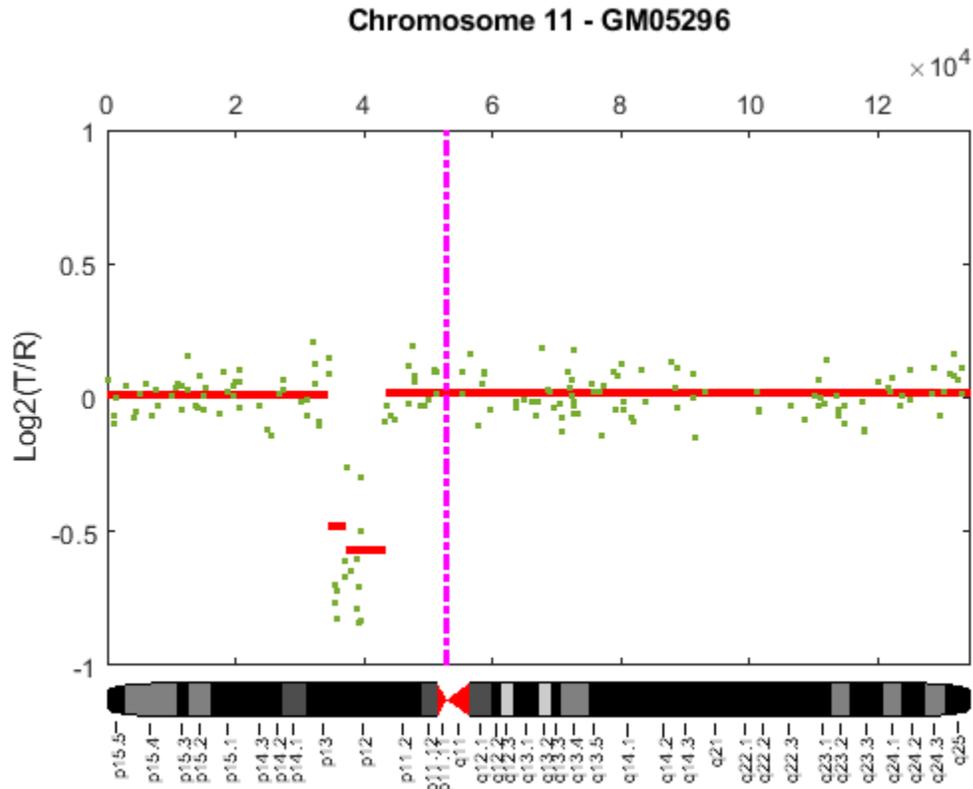
ylabel('Log2(T/R)')
ax = gca;
ax.Box = 'on';
ylim([-1, 1])
title(sprintf('Chromosome %d - GM05296', chr_num));
chromosomeplot(hs_cytobands, chr_num, 'addtoplot', gca, 'unit', 2)

end

```







As shown in the plots, no copy number alterations were found on chromosome 9, there is copy number gain span from *10q21* to *10q24*, and a copy number loss region from *11p12* to *11p13*. The CNAs found match the known results in cell line GM05296 determined by cytogenetic analysis.

You can also display the CNAs of the GM05296 cell line align to the chromosome ideogram summary view using the `chromosomeplot` function. Determine the genomic positions for the CNAs on chromosomes 10 and 11.

```
chr10_idx = GM05296_Data(2).SegIndex(2):GM05296_Data(2).SegIndex(3)-1;
chr10_cna_start = GM05296_Data(2).GenomicPosition(chr10_idx(1))*1000;
chr10_cna_end = GM05296_Data(2).GenomicPosition(chr10_idx(end))*1000;
```

```
chr11_idx = GM05296_Data(3).SegIndex(2):GM05296_Data(3).SegIndex(3)-1;
chr11_cna_start = GM05296_Data(3).GenomicPosition(chr11_idx(1))*1000;
chr11_cna_end = GM05296_Data(3).GenomicPosition(chr11_idx(end))*1000;
```

Create a structure containing the copy number alteration data from the GM05296 cell line data according to the input requirements of the `chromosomeplot` function.

```
cna_struct = struct('Chromosome', [10 11],...
                  'CNVType', [2 1],...
                  'Start', [chr10_cna_start, chr11_cna_start],...
                  'End', [chr10_cna_end, chr11_cna_end])
```

```
cna_struct =
```

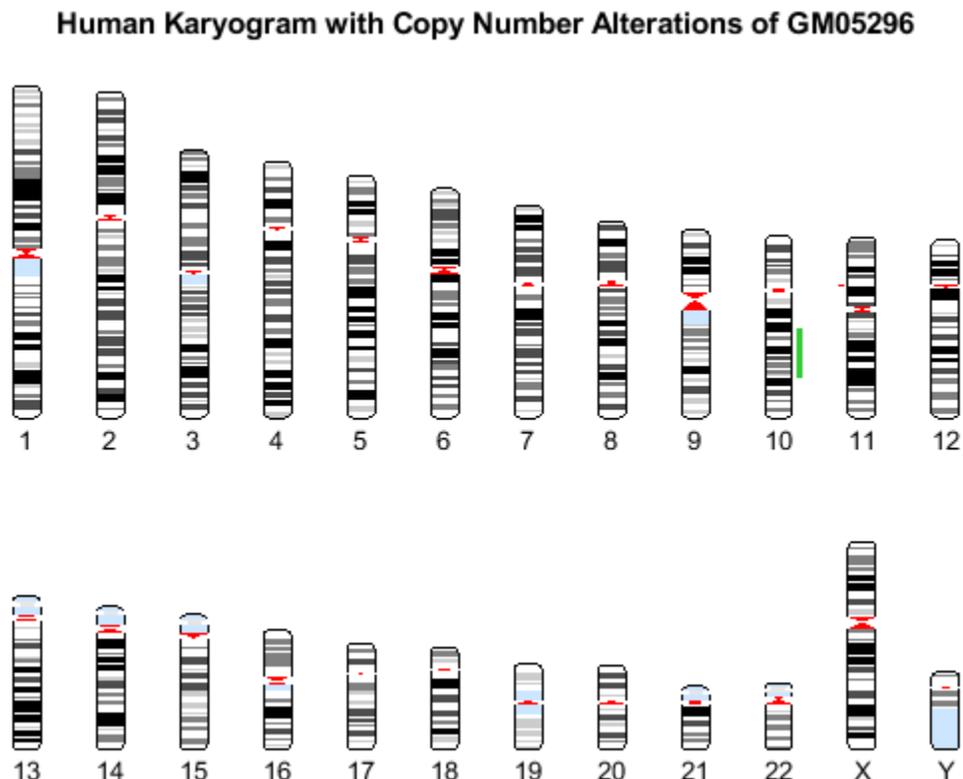
```

struct with fields:

  Chromosome: [10 11]
  CNVType: [2 1]
  Start: [69209000 34420000]
  End: [105905000 35914000]

chromosomeplot(hs_cytobands, 'cnv', cna_struct, 'unit', 2)
title('Human Karyogram with Copy Number Alterations of GM05296')

```



This example shows how MATLAB and its toolboxes provide tools for the analysis and visualization of copy-number alterations in array-based CGH data.

References

- [1] Redon, R., et al., "Global variation in copy number in the human genome", *Nature*, 444(7118):444-54, 2006.
- [2] Pinkel, D., et al., "High resolution analysis of DNA copy number variations using comparative genomic hybridization to microarrays", *Nature Genetics*, 20(2):207-11, 1998.
- [3] Snijders, A.M., et al., "Assembly of microarrays for genome-wide measurement of DNA copy number", *Nature Genetics*, 29(3):263-4, 2001.
- [4] Human Genome NCBI Build 36.

[5] Myers, C.L., et al., "Accurate detection of aneuploidies in array CGH and gene expression microarray data", *Bioinformatics*, 20(18):3533-43, 2004.

Analyzing Array-Based CGH Data Using Bayesian Hidden Markov Modeling

This example shows how to use a Bayesian hidden Markov model (HMM) technique to identify copy number alteration in array-based comparative genomic hybridization (CGH) data.

Introduction

Array-based CGH is a high-throughput technique to measure DNA copy number change across the genome. The DNA fragments or "clones" of test and reference samples are hybridized to mapped array fragments. Log₂ intensity ratios of test to reference provide useful information about genome-wide profiles in copy number. In an ideal situation, the log₂ ratio of normal (copy-neutral) clones is $\log_2(2/2) = 0$, single copy losses is $\log_2(1/2) = -1$, and single copy gains is $\log_2(3/2) = 0.58$. Multiple copy gains or amplifications would have values of $\log_2(4/2)$, $\log_2(5/2)$,.... Loss of both copies, or a deletion would correspond to the value of $-\infty$. In real applications, even after accounting for measurement error, the log₂ ratios differ considerably from the theoretical values. The ratios typically shrink towards zero. One main factor is contamination of the tumor samples with normal cells. There is also a dependence between the intensity ratios of neighboring clones. These issues necessitate the use of efficient statistical algorithms characterizing the genomic profiles.

Bayesian HMM

Guha et al., (2006) proposed a Bayesian statistical approach depending on a hidden Markov model (HMM) for analyzing array CGH data. The hidden Markov model accounts for the dependence between neighboring clones. The intensity ratios are generated by hidden copy number states. Bayesian learning is used to identify genome-wide changes in copy number from the data. Posterior inferences are made about the copy number gains and losses.

In this Bayesian HMM algorithm, there are four states, defined as copy number loss state (1), copy number neutral state (2), single copy gain state (3), and amplification (multiple gain) state (4). A copy number state is associated with each clone. The normalized log₂ ratios are assumed to be distributed as

$$Y_k \sim N(\mu_{sk}, \sigma_{sk}^2)$$

The μ is an unknown parameter for each state with this constraint:

$$\mu_1 < \mu_2 < \mu_3 < \mu_4$$

The priors for mean copy number changes are:

$$\mu_1 \sim N(-1, \tau_1^2) \cdot I(\mu_1 < -\epsilon)$$

$$\mu_2 \sim N(0, \tau_2^2) \cdot I(-\epsilon < \mu_2 < \epsilon)$$

$$\mu_3 \sim N(0.58, \tau_3^2) \cdot I(\epsilon < \mu_3 < 0.58)$$

$$[\mu_4 | \mu_3, \sigma_3] \sim N(1, \tau_4^2) \cdot I(\mu_4 > \mu_3 + 3\sigma_3)$$

Guha et al., (2006) also described an Metropolis-within-Gibbs algorithm to generate posterior samples. The MCMC algorithm is independently run for each chromosome to generate an MCMC

sample for the chromosome parameters. The starting values of the parameters are generated from the priors. The generated copy number states represent draws from the marginal posterior of interest. For each MCMC draw, the generated states are inspected and classified as focal aberrations, transition points, amplifications, outliers and whole chromosomal changes.

In this example, you will apply the Bayesian HMM algorithm to analyze the array CGH profiles of some pancreatic cancer samples [2].

Loading the Data

The data in this example is the array CGH profiles of 24 pancreatic adenocarcinoma cell lines and 13 primary tumor specimens from Alguirre et al.,(2004). Labeled DNA fragments were hybridized to Agilent® human cDNA microarrays containing 14,160 cDNA clones. About 9,420 clones have unique map positions with a median interval between mapped elements of 100.1 kb. More details of the data and experiment can be found in [2]. For convenience, the data has already been normalized and the log₂ based intensity ratios are provided by the MAT file `pancrea_oligocgh.mat`.

You will apply the Bayesian HMM algorithm to analyze chromosome 12 of sample 6 of the pancreatic adenocarcinoma data, and compare the results with the segments found by the circular binary segmentation (CBS) algorithm of Olshen et al.,(2004).

Load the MAT file containing the log₂ intensity ratios and mapped genomic positions of the 37 samples.

```
load pancrea_oligocgh
pancrea_data
```

```
pancrea_data =
  struct with fields:
      Sample: {37×1 cell}
      Chromosome: [13446×1 int8]
      GenomicPosition: [13446×1 int32]
      Log2Ratio: [13446×37 double]
      Log2RatioMed: [13446×37 double]
      Log2RatioSeg: [13446×37 double]
      CloneIDs: [13446×1 int32]
```

Specify the chromosome number and sample to analyze.

```
sampleIndex = 6;
chromID = 12;
sample = pancrea_data.Sample{sampleIndex}
```

```
sample =
  'PA.C.Dan.G'
```

Load and plot the log₂ ratio data of chromosome 12 from sample *PA.C.Dan.G*.

```
idx = pancrea_data.Chromosome == chromID;
X = double(pancrea_data.GenomicPosition(idx));
```

```

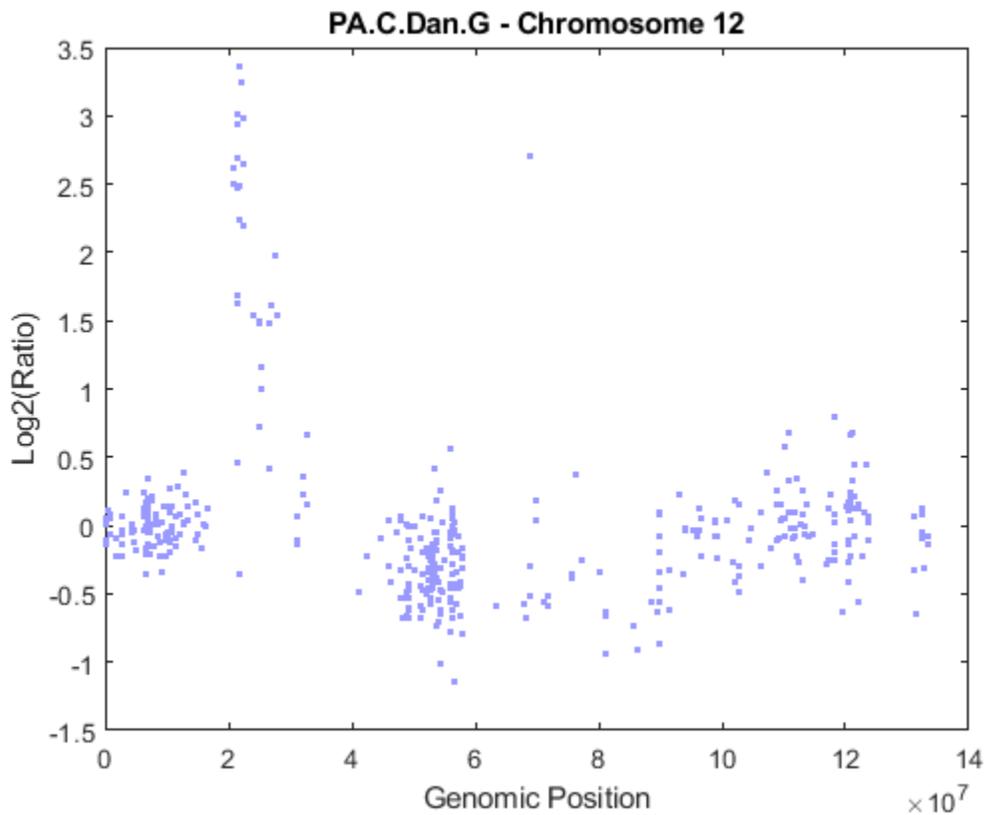
Y = pancrea_data.Log2Ratio(idx, sampleIndex);

% Remove NaN data points
idx = ~isnan(Y);
X = X(idx);
Y = Y(idx);

% Plot the data
figure;
plot(X, Y, '.', 'color', [0.6 0.6 1])

ylims = [-1.5, 3.5];
ylim(gca, ylims)
title(sprintf('%s - Chromosome %d', sample, chromID))
xlabel('Genomic Position');
ylabel('Log2(Ratio)')

```



Number of clones on chromosome 12 to be analyzed

```
N = numel(Y)
```

```
N =
```

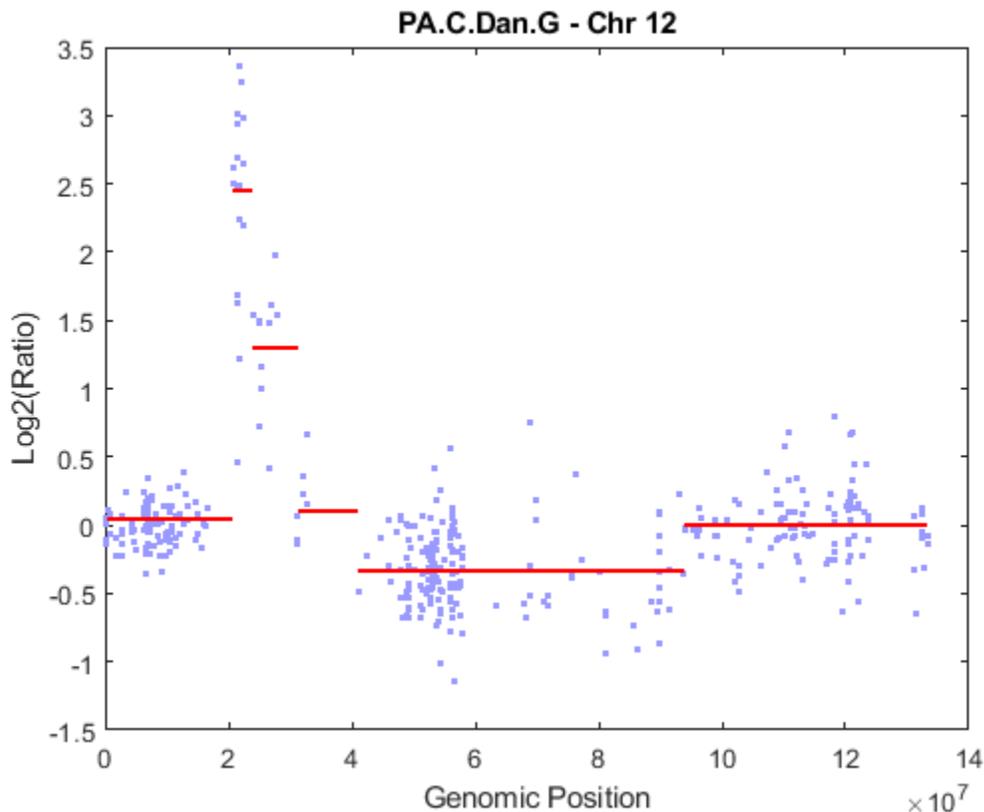
```
437
```

Performing Circular Binary Segmentation

You can start the analysis by performing chromosomal segmentation using the CBS algorithm [3], which is implemented in the `cgHcbs` function. The process will take several seconds. You can view the plot of the segment means over the original data by specifying the `SHOWPLOT` parameter. Note: You can type `doc cgHcbs` for more details on this function.

```
PS = cgHcbs(pancrea_data, 'SampleInd', sampleIndex, ...
            'Chromosome', chromID, 'ShowPlot', chromID);
ylim(gca, ylims)
```

Analyzing: PA.C.Dan.G. Current chromosome 12



As shown in the figure, the CBS procedure declared the set of high intensity ratios as two separate segments. The CBS procedure also found a region with copy number losses.

Initializing Parameters

The Bayesian HMM approach uses a Metropolis-within-Gibbs algorithm to generate posterior samples of the parameters [1]. The model parameters are grouped into four blocks. The algorithm iteratively generates each of the four blocks conditional on the remaining blocks and the data.

To analyze the data with the Bayesian HMM algorithm, you need to initialize the parameters. More details on prior parameters can be found in references [1] and [4].

Initialize the state of the random number generator to ensure that the figures generated by these command match the figures in the HTML version of this example.

```
rng('default');
```

Define the number of states

```
NS = 4;
```

Define the number of MCMC iterations

```
NMC = 100;
```

Determine the hyperparameters of the prior distributions for the four states.

```
mus_hyper = [-1, 0, 0.58, 1];  
taus_hyper = [1, 1, 1, 2];
```

Set the parameter epsilon which determines the constrains of the means.

```
eps = 0.1;
```

Set the bounds of the prior means of each state.

```
mu_low_bounds = [-Inf, -eps, eps, 0.58];  
mu_up_bounds = [-eps, eps, 0.58, Inf];
```

Guha et al., (2006) assumes the inverse of the prior error variances (σ^2) as gamma distributions with lower bounds of 0.41 for states 1, 2 and 3. Set the scale parameters for the gamma distributions for each state.

```
sg_alpha = [1 1 1 1];  
sg_beta = [1, 1, 1, 1];  
sg_bounds = [0.41 0.41 0.41 1];
```

Define a variable `states` to store the copy number state sequences of the clones for each MCMC iteration.

```
states = zeros(N, NMC);
```

Define a variable `st_counts` to hold the state transition counts for each copy number state.

```
st_counts = zeros(NS, NS);
```

Determining the Prior Distributions

The MCMC iteration starts at

```
iloop = 1;
```

Determine sigmas for the four states by sampling from gamma distribution with prior scale parameter alpha and beta.

```
sigmas = zeros(NS, NMC);  
for i = 1:NS  
    sigmas(i, iloop) = acghhmsample('gamma', sg_alpha(i), sg_beta(i), sg_bounds(i));  
end
```

Determine means for the four states by sampling from truncated normal distribution between the lower and upper bounds of the means. Note: The fourth state lower bound will be determined by the third state.

```

mus = zeros(NS, NMC);
for i = 1:NS
    if i == 4
        mu_low_bounds(4) = mus(3,iloop) + 3*sigmas(3,iloop);
    end
    mus(i, iloop) = acghhmsample('normal', mus_hyper(i), taus_hyper(i),...
        mu_low_bounds(i), mu_up_bounds(i));
end

```

Assume independent Dirichlet priors for the rows of the stochastic 4x4 transition probability matrix [1]. Generate the stochastic prior transition matrix A from the Dirichlet distributions.

```

a = ones(NS, NS);
A = acghhmsample('dirichlet', a, NS);

```

The transition matrix has a unique stationary distribution. The stationary distribution PI is an eigenvector of the transition matrix associated with the eigenvalue 1.

```

PI = @(x, n) (ones(1,n)/(eye(n) -x + ones(n)))';

```

Generate the prior stationary distribution PI.

```

Pi = PI(A, NS);

```

Generate the initial emission matrix B

```

B = zeros(NS, N);
for i = 1:NS
    B(i,:) = normpdf(Y, mus(i,iloop), sigmas(i,iloop));
end

```

Decode initial hidden states of the clones using a stochastic forward-backward algorithm [4].

```

states(:, iloop) = acghhmmfb(Pi, A, B);

```

Generating Posterior Samples

For each MCMC iteration, the four blocks of parameters are generated as follows [1]: Update block B1 using a Metropolis-Hastings step to generate the transition matrix, update block B2 the copy number states using a stochastic forward propagate backward sampling algorithm, update block B3 by computing the *mus*, and update block B4 to generate *sigmas*.

```

for iloop = 2:NMC
    % Compute the number of transitions from state i to state j
    for i = 1:NS
        for j = 1:NS
            st_counts(i, j) = sum((states(1:N-1, iloop-1) == i) .* (states(2:N, iloop-1) == j));
        end
    end

    % Updating block B1
    % Generate the transition matrix from the Dirichlet distributions
    C = acghhmsample('dirichlet', st_counts + 1, NS);

    % Compute the state probabilities under stationary distribution of a
    % given transition matrix C.
    PiC = PI(C, NS);

```

```

% Compute the accepting probability using a Metropolis-Hastings step
beta = min([1, exp(log(PiC(states(1, iloop-1))) - log(Pi(states(1, iloop-1))))]);
if rand < beta
    A = C;
    Pi = PiC;
end

% Updating block B2
% Generate copy number states using Forward propagate, backward sampling [4].
states(:, iloop) = acghmmfb(Pi, A, B);

% Updating blocks B3 and B4
for i = 1:NS
    idx_s = states(:, iloop) == i;
    num_states = sum(idx_s);

    % If state i is not observed, then draw from its prior parameters
    if num_states == 0
        mus(i, iloop) = acghmmsample('normal', mus_hyper(i),...
            taus_hyper(i), mu_low_bounds(i), mu_up_bounds(i));
        sigmas(i, iloop) = acghmmsample('gamma', sg_alpha(i),...
            sg_beta(i), sg_bounds(i));
    else
        Y_avg = mean(Y(idx_s));
        theta_prec = 1/taus_hyper(i)^2 + num_states/sigmas(i, iloop-1)^2;
        weight_means = (mus_hyper(i)/(taus_hyper(i)^2) +...
            Y_avg * num_states/(sigmas(i, iloop-1)^2))/theta_prec;
        % Compute mus - B3
        mus(i, iloop) = acghmmsample('normal', weight_means, ...
            1/sqrt(theta_prec), mu_low_bounds(i), mu_up_bounds(i));
        % Compute sigmas - B4
        Y_v = sum((Y(idx_s) - mus(i, iloop)).^2);
        sigmas(i, iloop) = acghmmsample('gamma', sg_alpha(i)+num_states/2,...
            sg_beta(i)+Y_v/2, sg_bounds(i));
    end
    % Update the emission matrix with new mus and sigmas.
    B(i,:) = normpdf(Y, mus(i, iloop), sigmas(i, iloop));
end
end

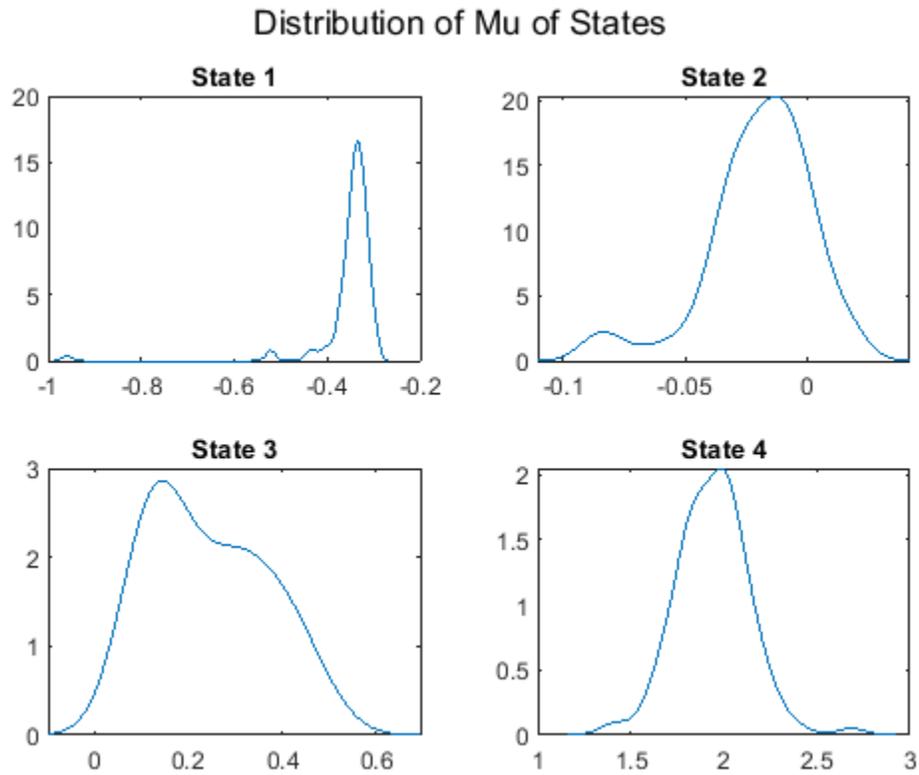
```

Plot the posterior mean *mu* distributions of the four states.

```

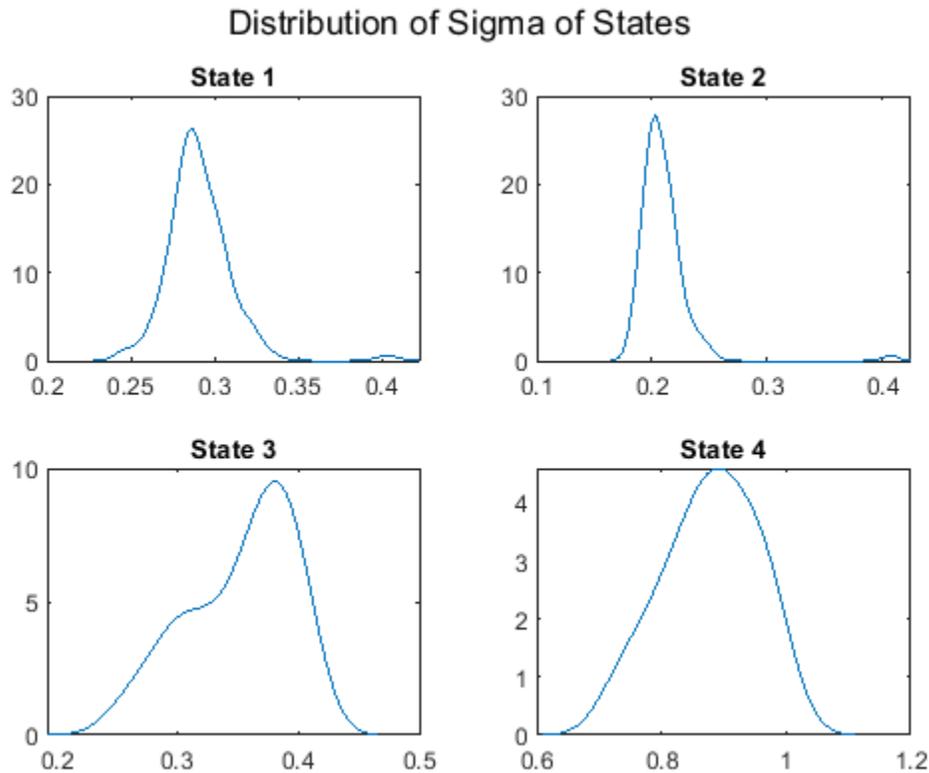
figure;
for j = 1:NS
    subplot(2,2,j)
    ksdensity(mus(j,:));
    title(sprintf('State %d', j))
end
sgtitle('Distribution of Mu of States');
hold off;

```



Plot the posterior *sigma* distributions of the four states.

```
figure;
for j = 1:NS
    subplot(2,2,j)
    ksdensity(sigmas(j,:));
    title(sprintf('State %d', j))
end
sgtitle('Distribution of Sigma of States');
hold off;
```



Posterior Inference

Draw a state label for each clone from the MCMC sampling and compute the posterior probabilities of each state.

```
clone_states = zeros(1, N);
state_prob = zeros(NS, N);
state_count = zeros(NS, N);

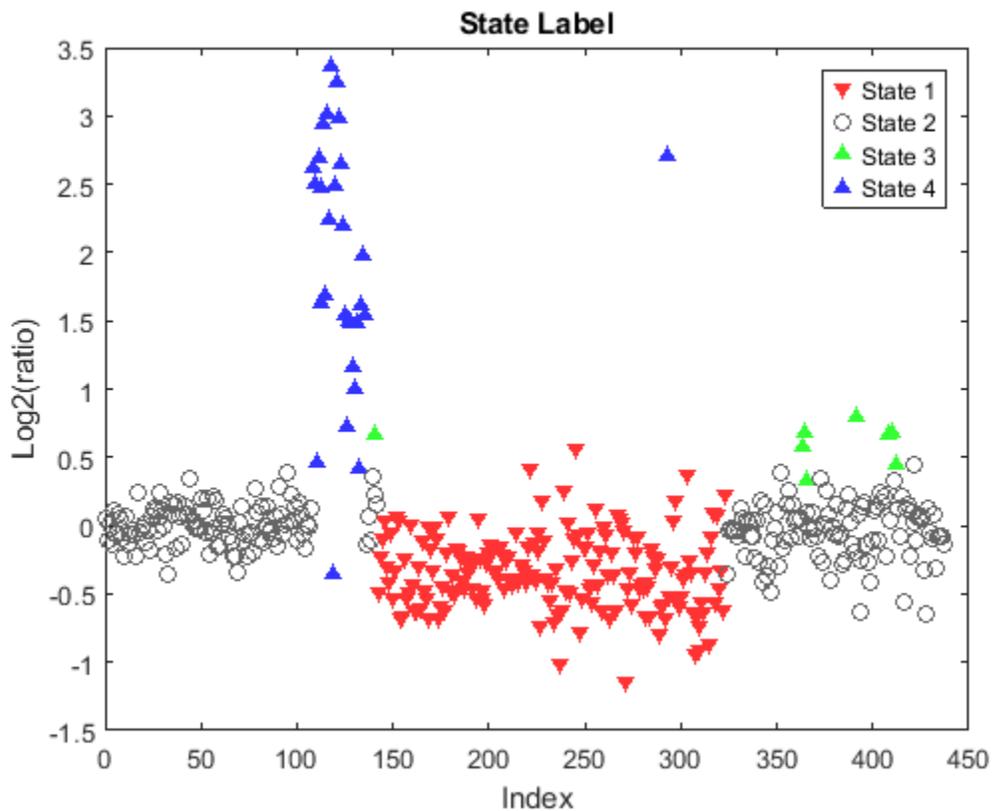
for i = 1:N % for each clone
    state = states(i, :);
    for j=1:NS
        state_count(j, i) = sum(state == j);
    end

    selState = find(state_count(:,i) == max(state_count(:,i)));
    if length(selState) > 1
        if i ~= 1
            clone_states(i) = clone_states(i-1);
        else
            clone_states(i) = min(selState);
        end
    else
        clone_states(i) = selState;
    end
    state_prob(:, i) = state_count(:,i)/NMC;
end
```

```
end
clone_states = clone_states';
```

Plot the state label for each clone on chromosome 12 of sample *PA.C.Dan.G*.

```
figure;
leg = zeros(1,4);
for i = 1:N
    if clone_states(i) == 1
        leg(1) = plot(i,Y(i),'v', 'MarkerFaceColor', [1 0.2 0.2],...
                    'MarkerEdgeColor', 'none');
    elseif clone_states(i) == 2
        leg(2) = plot(i,Y(i),'o', 'Color', [0.4 0.4 0.4]);
    elseif clone_states(i) == 3
        leg(3) = plot(i,Y(i),'^', 'MarkerFaceColor', [0.2 1 0.2],...
                    'MarkerEdgeColor', 'none');
    elseif clone_states(i) == 4
        leg(4) = plot(i, Y(i), '^', 'MarkerFaceColor', [0.2 0.2 1],...
                    'MarkerEdgeColor', 'none');
    end
    hold on;
end
ylim(gca, ylims)
legend(leg, 'State 1', 'State 2', 'State 3', 'State 4')
xlabel('Index')
ylabel('Log2(ratio)')
title('State Label')
hold off
```



Classifying Array CGH Profiles

For each MCMC draw, the generated states can be classified as focal aberrations, transition points, amplifications, outliers and whole chromosomal changes [1]. In this example, you will find the high-level amplifications, transition points and outliers on chromosome 12 of sample *PA.C.Dan.G*.

A clone with state = 4 is considered a high-level amplification [1]. Find high-level amplifications.

```
high_lvl_amp_idx = find(clone_states == 4);
```

A transition point is associated with large-scale regions of gains and losses and is declared when the width of the altered region exceeds 5 mega base pair [1]. Find transition points.

```
region_lim = 5e6;
focalabr_idx=[1;find(diff(clone_states)~=0);N];
istranspoint = false(length(focalabr_idx), 1);
for i = 1:length(focalabr_idx)-1
    region_x = X(focalabr_idx(i+1)) - X(focalabr_idx(i));
    istranspoint(i+1) = region_x > region_lim;
end
trans_idx = focalabr_idx(istranspoint);
% Remove adjacent trans_idx that have the same states.
hasadjacentstate = [diff(clone_states(trans_idx))==0; true];
trans_idx = trans_idx(~hasadjacentstate)
focalabr_idx = focalabr_idx(~istranspoint);
focalabr_idx = focalabr_idx(2:end-1);
```

```
trans_idx =
```

```
107
135
323
```

An outlier for gains is a focal aberration satisfying its z-score greater than 2, while an outlier for losses has a z-score less than -2 [1].

Find outliers for losses

```
outlier_loss_idx = focalabr_idx(clone_states(focalabr_idx) == 1)
if ~isempty(outlier_loss_idx)
    [F,Xi] = ksdensity(mus(1,:));
    [dummy, idx] = max(F);
    mu_1 = Xi(idx);

    [F,Xi] = ksdensity(sigmas(1,:));
    [dummy, idx] = max(F);
    sigma_1 = Xi(idx);
    outlier_loss_idx = outlier_loss_idx((Y(outlier_loss_idx) - mu_1)/sigma_1 < -2)
end
```

```
outlier_loss_idx =
```

```
0×1 empty double column vector
```

Find outliers for gains

```

outlier_gain_idx = focalabr_idx(clone_states(focalabr_idx) == 3);
if ~isempty(outlier_gain_idx)
    [F,Xi] = ksdensity(mus(3,:));
    [dummy, idx] = max(F);
    mu_1 = Xi(idx);

    [F,Xi] = ksdensity(sigmas(3,:));
    [dummy, idx] = max(F);
    sigma_1 = Xi(idx);
    outlier_gain_idx = outlier_gain_idx((Y(outlier_gain_idx) - mu_1)/sigma_1 > 2)
end

outlier_gain_idx =

    0×1 empty double column vector

```

Add the classified labels to the intensity ratio plot of chromosome 12 of sample *PA.C.Dan.G*. Plot the segment means from the CBS procedure for comparison.

```

figure;
hl1 = plot(X, Y, '.', 'color', [0.4 0.4 0.4]);
hold on;
if ~isempty(high_lvl_amp_idx)
    hl2 = line(X(high_lvl_amp_idx), Y(high_lvl_amp_idx),...
        'LineStyle', 'none',...
        'Marker', '^',...
        'MarkerFaceColor', [0.2 0.2 1],...
        'MarkerEdgeColor', 'none');
end

if ~isempty(trans_idx)
    for i = 1:numel(trans_idx)
        hl3 = line(ones(1,2)*X(trans_idx(i)), [-3.5, 3.5],...
            'LineStyle', '--',...
            'Color', [1 0.6 0.2]);
    end
end

if ~isempty(outlier_gain_idx)
    line(X(outlier_gain_idx), Y(outlier_gain_idx),...
        'LineStyle', 'none',...
        'Marker', 'v',...
        'MarkerFaceColor', [1 0 0],...
        'MarkerEdgeColor', 'none');
end

if ~isempty(outlier_loss_idx)
    hl4 = line(X(outlier_loss_idx), Y(outlier_loss_idx),...
        'LineStyle', 'none',...
        'Marker', 'v',...
        'MarkerFaceColor', [1 0 0],...
        'MarkerEdgeColor', 'none');
end

% Plot segment means from the CBS procedure.
for i = 1:numel(PS.SegmentData.Start)

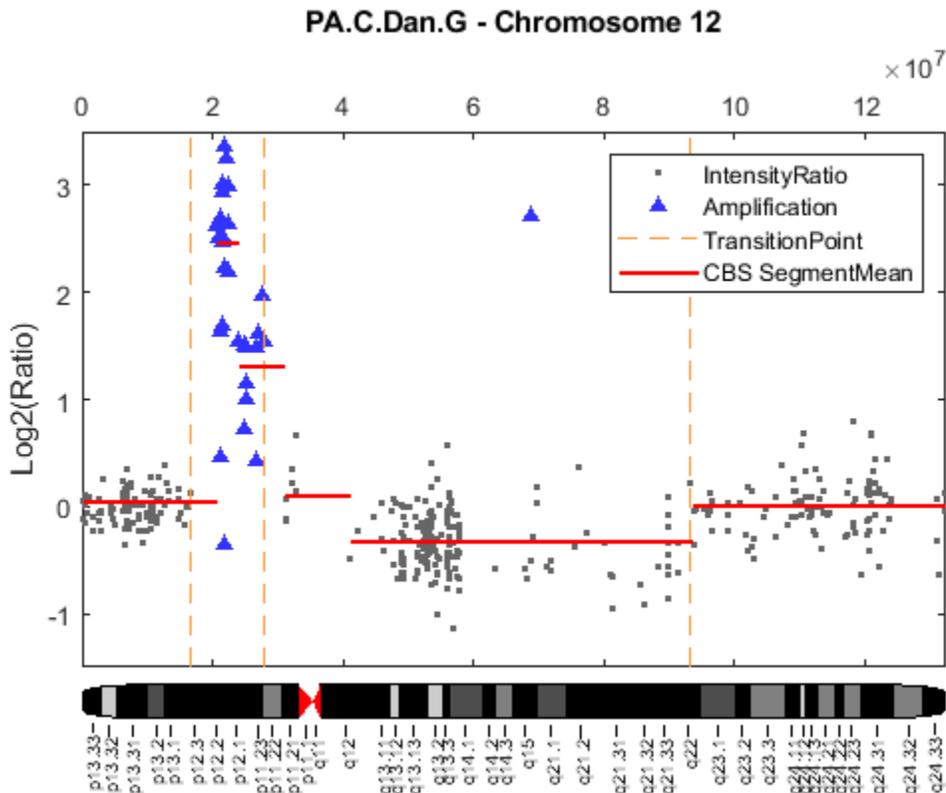
```

```

h15 = line([PS.SegmentData.Start(i) PS.SegmentData.End(i)],...
          [PS.SegmentData.Mean(i) PS.SegmentData.Mean(i)],...
          'Color', [1 0 0],...
          'LineWidth', 1.5);
end
ylim(gca, ylims)
ylabel('Log2(Ratio)')
title(sprintf('%s - Chromosome %d', sample, chromID))

% Adding chromosome 12 ideogram and legends to the plot.
chromosomeplot('hs_cytoBand.txt', chromID, 'addtoplot', gca)
legend([h11, h12, h13, h15], 'IntensityRatio', 'Amplification',...
      'TransitionPoint', 'CBS SegmentMean')

```



The Bayesian HMM algorithm found 3 transition points indicated by the broken vertical lines in the plot. The Bayesian HMM algorithm identified two high-level amplified regions marked by blue up-triangles in the plot. The two high-level amplified regions correspond to the two minimal common regions (MCRs)[2] on chromosome 12, associated with copy number gains as explained by Aguirre et al.,(2004). The Bayesian HMM declared the first set of high intensity ratios as a single region of high-level amplification. In comparison, the CBS procedure failed to detect the second MCR and segmented the first MCR into two regions. No outlier was detected in this example.

References

[1] Guha, S., Li, Y. and Neuberg, D., "Bayesian hidden Markov modeling of array CGH data", Journal of the American Statistical Association, 103(482):485-497, 2008.

- [2] Aguirre, A.J., et al., "High-resolution characterization of the pancreatic adenocarcinoma genome", PNAS, 101(24):9067-72, 2004.
- [3] Olshen, A.B., et al., "Circular binary segmentation for the analysis of array-based DNA copy number data", Biostatistics, 5(4):557-7, 2004.
- [4] Shah, S.P., et al., "Integrating copy number polymorphisms into array CGH analysis using a robust HMM", Bioinformatics, 22(14):e431-e439, 2006

Visualizing Microarray Data

This example shows various ways to explore and visualize raw microarray data. The example uses microarray data from a study of gene expression in mouse brains [1].

Exploring the Microarray Data Set

Brown, V.M et.al. [1] used microarrays to explore the gene expression patterns in the brain of a mouse in which a pharmacological model of Parkinson's disease (PD) was induced using methamphetamine. The raw data for this experiment is available from the Gene Expression Omnibus website using the accession number GSE30 [1].

The file `mouse_h3pd.gpr` contains the data for one of the microarrays used in the study, specifically from a sample collected from voxel H3 of the brain in a Parkinson's Disease (PD) model mouse. The file uses the GenePix® GPR file format. The voxel sample was labeled with Cy3 (green) and the control (RNA from a total, not voxelated, normal mouse brain) was labeled with Cy5.

GPR formatted files provide a large amount of information about the array including the mean, median and standard deviation of the foreground and background intensities of each spot at the 635nm wavelength (the red, Cy5 channel) and the 532nm wavelength (the green, Cy3 channel).

The command `gprread` reads the data from the file into a structure.

```
pd = gprread('mouse_h3pd.gpr')
```

```
pd =
```

```
struct with fields:
    Header: [1x1 struct]
    Data: [9504x38 double]
    Blocks: [9504x1 double]
    Columns: [9504x1 double]
    Rows: [9504x1 double]
    Names: {9504x1 cell}
    IDs: {9504x1 cell}
    ColumnNames: {38x1 cell}
    Indices: [132x72 double]
    Shape: [1x1 struct]
```

You can access the fields of a structure using dot notation. For example, access the first ten column names.

```
pd.ColumnNames(1:10)
```

```
ans =
```

```
10x1 cell array
    {'X'          }
    {'Y'          }
    {'Dia.'       }
    {'F635 Median' }
```

```
{'F635 Mean'   }
{'F635 SD'    }
{'B635 Median' }
{'B635 Mean'  }
{'B635 SD'    }
{'% > B635+1SD'}
```

You can also access the first ten gene names.

```
pd.Names(1:10)
```

```
ans =
```

```
10×1 cell array
```

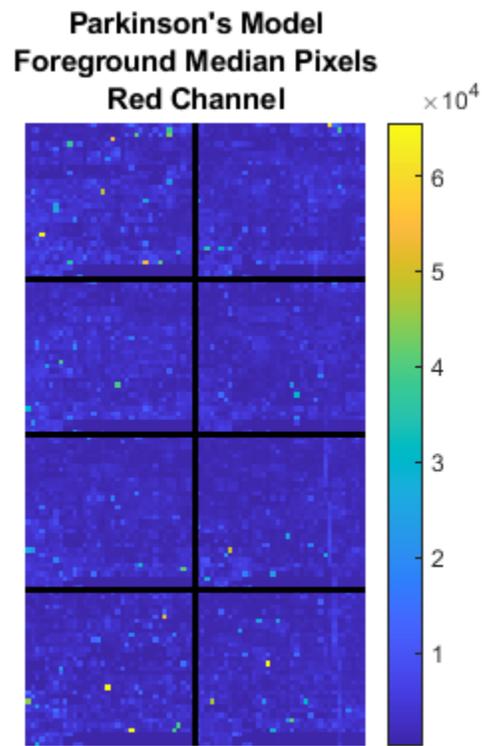
```
{'AA467053' }
{'AA388323' }
{'AA387625' }
{'AA474342' }
{'Myo1b'    }
{'AA473123' }
{'AA387579' }
{'AA387314' }
{'AA467571' }
{0×0 char  }
```

Spatial Images of Microarray Data

The `mimage` command can take the microarray data structure and create a pseudocolor image of the data arranged in the same order as the spots on the array, i.e., a spatial plot of the microarray. The "F635 Median" field shows the median pixel values for the foreground of the red (Cy5) channel.

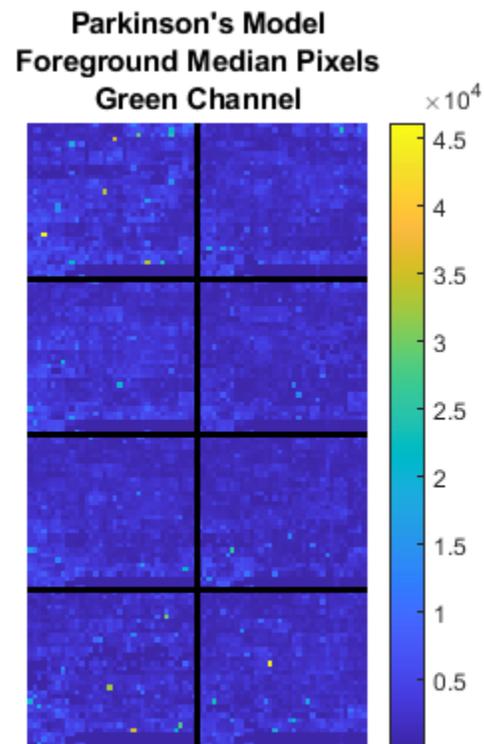
```
figure
```

```
mimage(pd,'F635 Median','title',{'Parkinson's Model','Foreground Median Pixels','Red Channel'})
```



The "F532 Median" field corresponds to the foreground of the green (Cy3) channel.

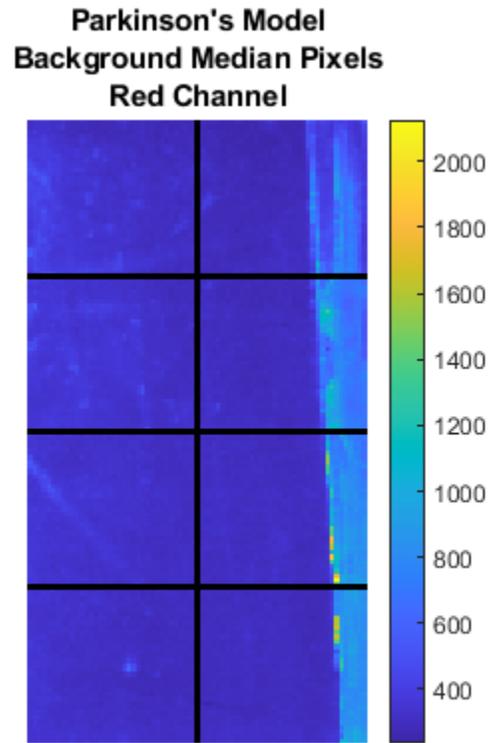
```
figure  
mimage(pd, 'F532 Median', 'title', {'Parkinson's Model', 'Foreground Median Pixels', 'Green Channel
```



The "B635 Median" field shows the median values for the background of the red channel. Notice the very high background levels down the right side of the array.

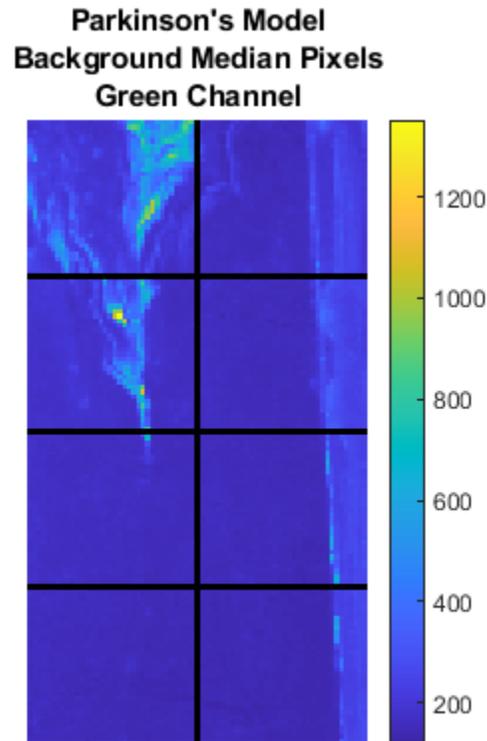
figure

```
mimage(pd, 'B635 Median', 'title', {'Parkinson's Model', 'Background Median Pixels', 'Red Channel'})
```



The "B532 Median" shows the median values for the background of the green channel.

```
figure  
mimage(pd, 'B532 Median', 'title', {'Parkinson's Model', 'Background Median Pixels', 'Green Channel'})
```



You can now consider the data obtained for the same brain voxel in an untreated control mouse. In this case, the voxel sample was labeled with Cy3, and the control (RNA from a total, not voxelated brain) was labeled with Cy5.

```
wt = gprread('mouse_h3wt.gpr')
```

```
wt =
```

```
struct with fields:
```

```
Header: [1x1 struct]
Data: [9504x38 double]
Blocks: [9504x1 double]
Columns: [9504x1 double]
Rows: [9504x1 double]
Names: {9504x1 cell}
IDs: {9504x1 cell}
ColumnNames: {38x1 cell}
Indices: [132x72 double]
Shape: [1x1 struct]
```

Use `mimage` to show pseudocolor images of the foreground and background corresponding to the untreated mouse. The `subplot` command can be used to combine the plots.

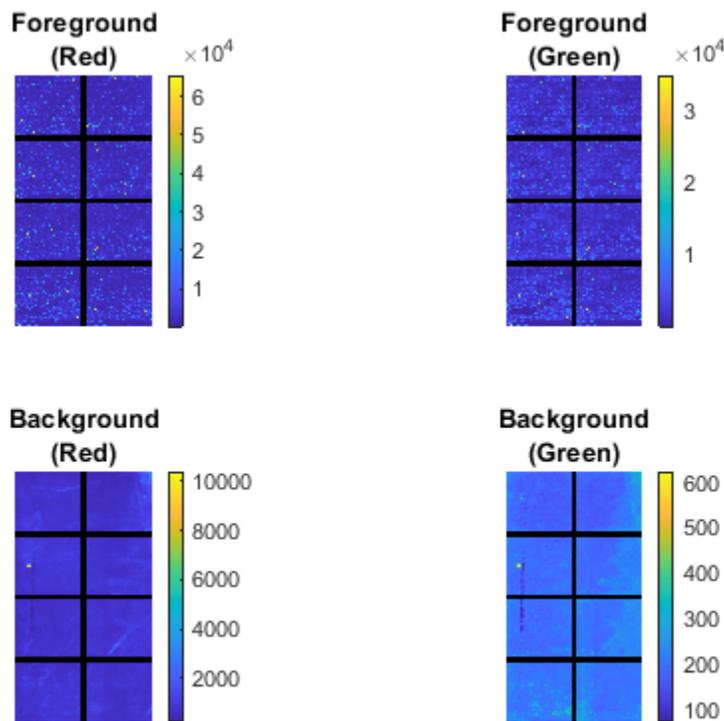
```
figure
subplot(2,2,1);
```

```

mimage(wt, 'F635 Median', 'title', {'Foreground', '(Red)'})
subplot(2,2,2);
mimage(wt, 'F532 Median', 'title', {'Foreground', '(Green)'})
subplot(2,2,3);
mimage(wt, 'B635 Median', 'title', {'Background', '(Red)'})
subplot(2,2,4);
mimage(wt, 'B532 Median', 'title', {'Background', '(Green)'})

annotation('textbox', 'String', 'Wild Type Median Pixel Values', ...
           'Position', [0.3 0.05 0.9 0.01], 'EdgeColor', 'none', 'FontSize', 12);

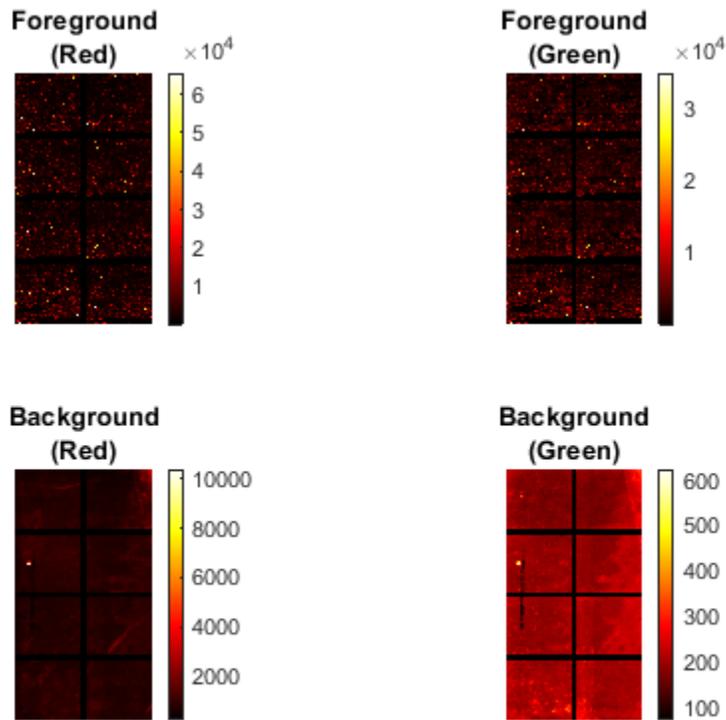
```



Wild Type Median Pixel Values

If you look at the scale for the background images, you will notice that the background levels are much higher than those for the PD mouse and there appears to be something non random affecting the background of the Cy3 channel of this slide. Changing the colormap can sometimes provide more insight into what is going on in pseudocolor plots. For more control over the color, try the `colormapeditor` function. You can also right-click on the colorbar to bring up various options for modifying the colormap of the plot including interactive colormap shifting.

```
colormap hot
```



Wild Type Median Pixel Values

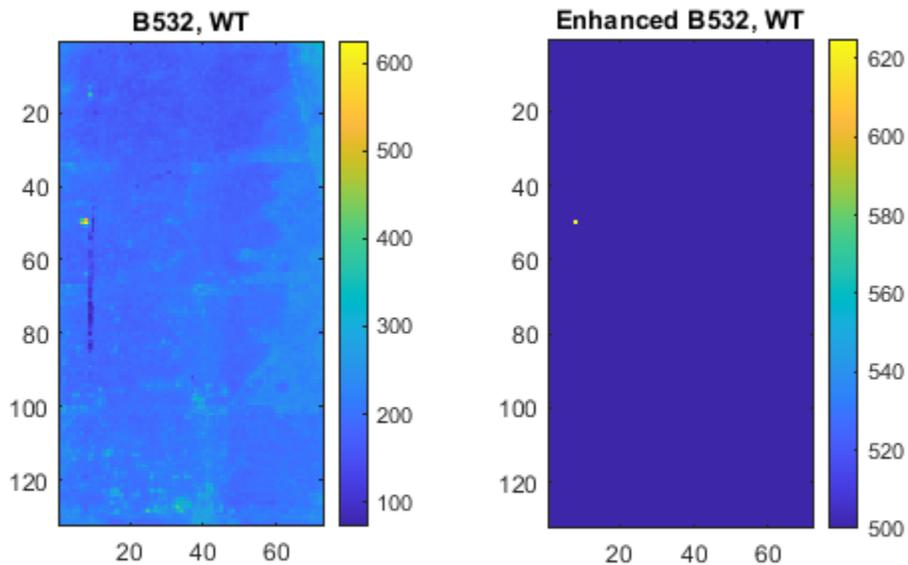
The `mimage` command is a simple way to quickly create pseudocolor images of microarray data. However, sometimes it is convenient to create customizable plots using the `imagesc` command, as shown below.

Use `magetfield` to extract data for the B532 median field and the Indices field to index into the Data. You can bound the intensities of the background plot to give more contrast in the image.

```
b532Data = magetfield(wt, 'B532 Median');
maskedData = b532Data;
maskedData(b532Data < 500) = 500;
maskedData(b532Data > 2000) = 2000;
```

```
figure
subplot(1,2,1);
imagesc(b532Data(wt.Indices))
axis image
colorbar
title('B532, WT')

subplot(1,2,2);
imagesc(maskedData(wt.Indices))
axis image
colorbar
title('Enhanced B532, WT')
```

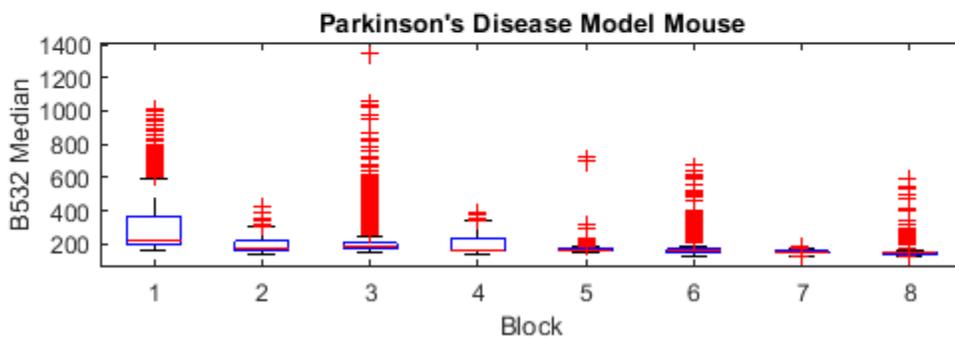
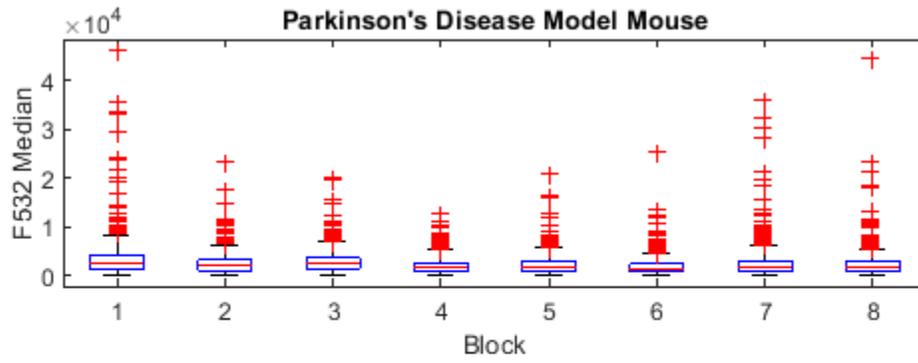


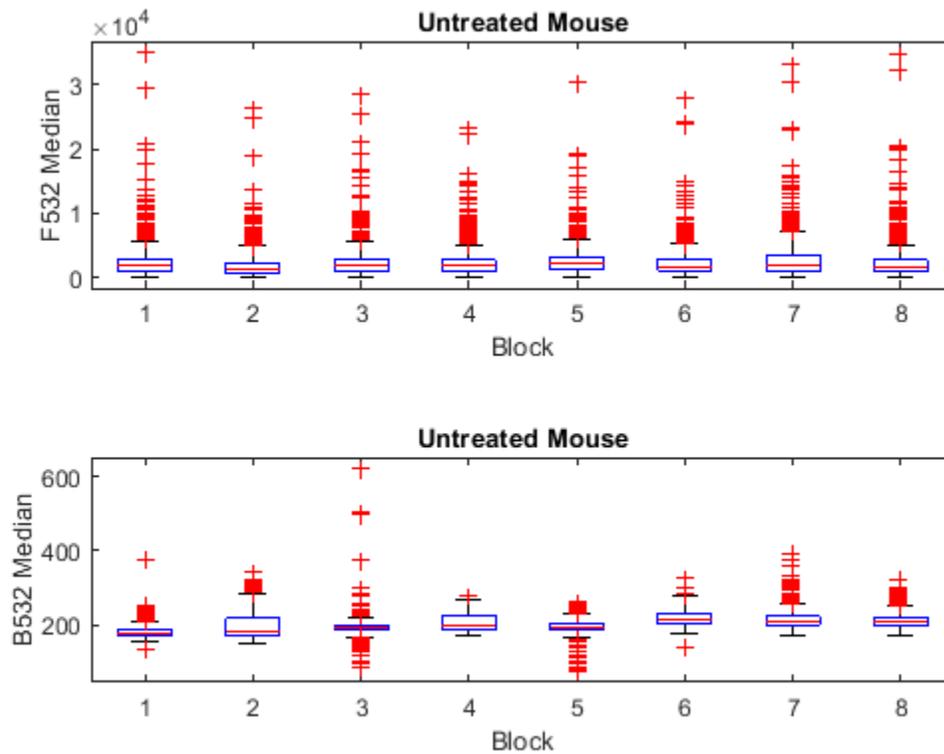
Statistics of the Microarrays

The `maboxplot` function can be used to look at the distribution of data in each of the blocks.

```
figure
subplot(2,1,1)
maboxplot(pd, 'F532 Median', 'title', 'Parkinson''s Disease Model Mouse')
subplot(2,1,2)
maboxplot(pd, 'B532 Median', 'title', 'Parkinson''s Disease Model Mouse')
```

```
figure
subplot(2,1,1)
maboxplot(wt, 'F532 Median', 'title', 'Untreated Mouse')
subplot(2,1,2)
maboxplot(wt, 'B532 Median', 'title', 'Untreated Mouse')
```



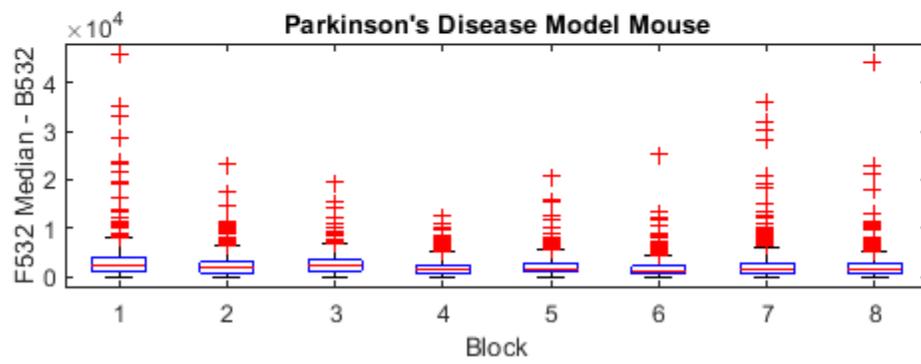
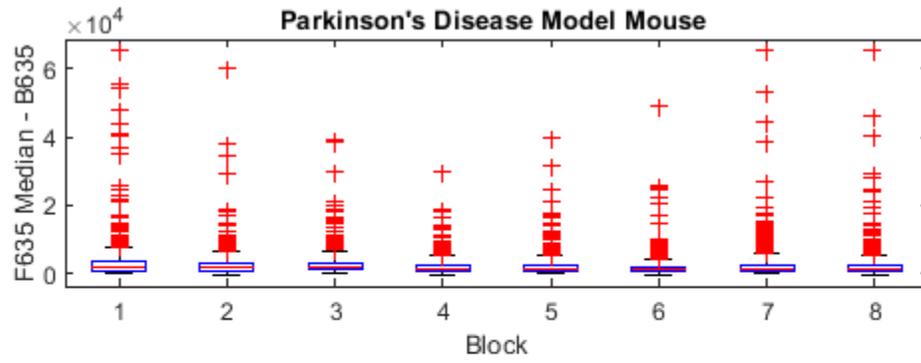


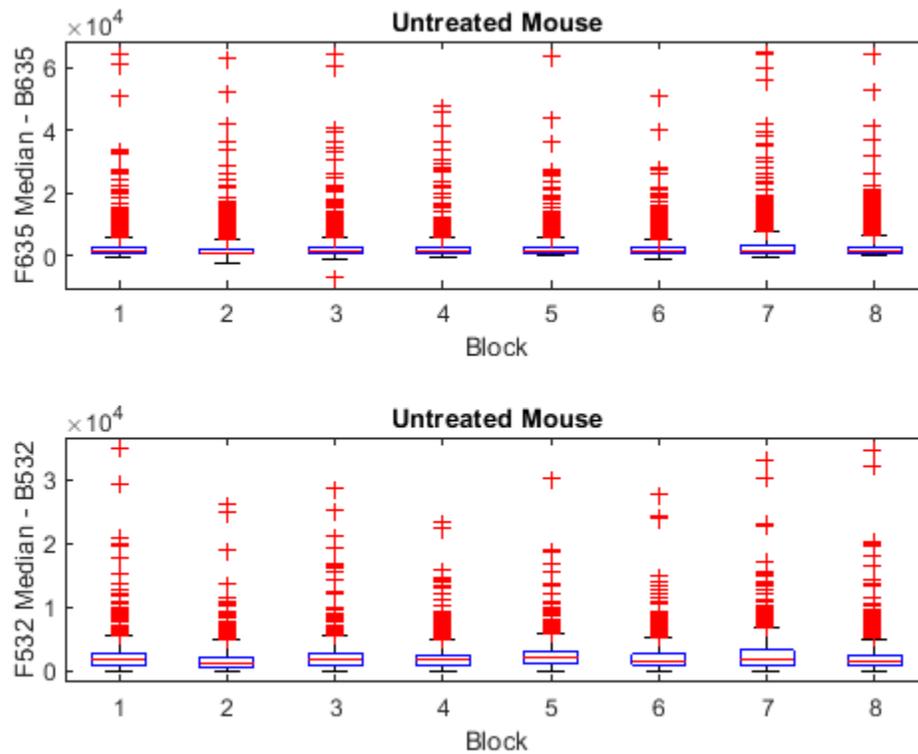
From the box plots you can clearly see the spatial effects in the background intensities. Blocks number 1,3,5 and 7 are on the left side of the arrays, and blocks number 2,4,6 and 8 are on the right side.

There are two columns in the microarray data structure labeled "F635 Median - B635" and "F532 Median - B532". These columns are the differences between the median foreground and the median background for the 635 nm channel and 532 nm channel respectively. These give a measure of the actual expression levels. The spatial effect is less noticeable in these plots.

```
figure
subplot(2,1,1)
maboxplot(pd, 'F635 Median - B635', 'title', 'Parkinson''s Disease Model Mouse ')
subplot(2,1,2)
maboxplot(pd, 'F532 Median - B532', 'title', 'Parkinson''s Disease Model Mouse')
```

```
figure
subplot(2,1,1)
maboxplot(wt, 'F635 Median - B635', 'title', 'Untreated Mouse')
subplot(2,1,2)
maboxplot(wt, 'F532 Median - B532', 'title', 'Untreated Mouse')
```





Scatter Plots of Microarray Data

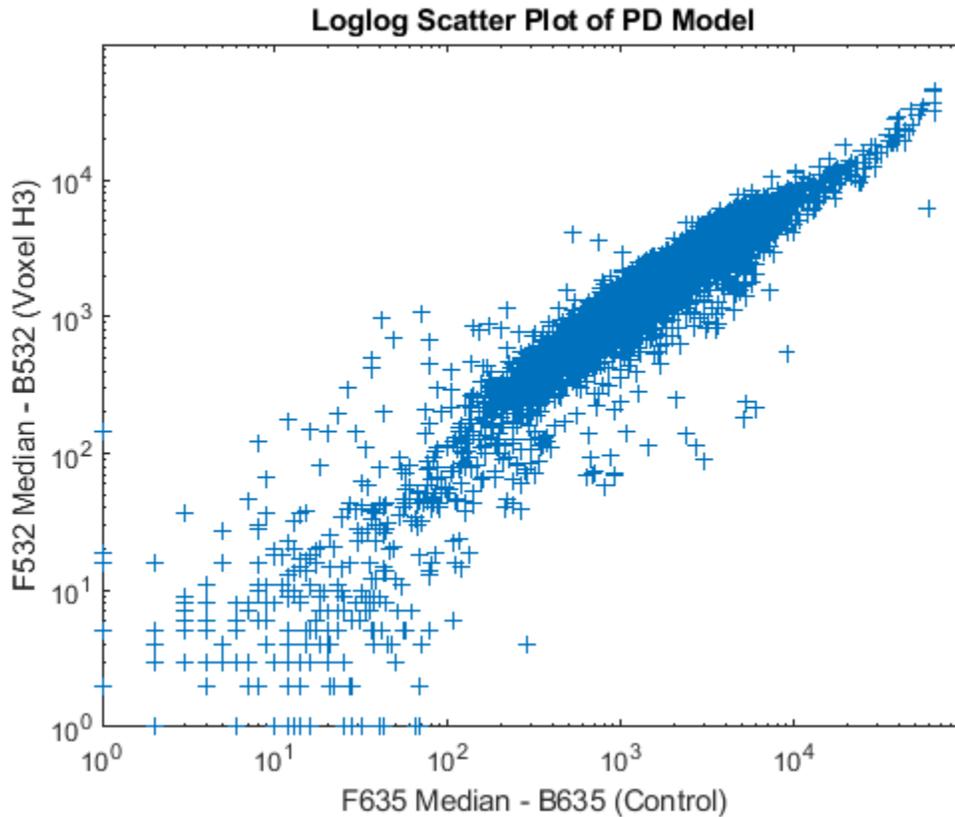
Rather than work with the data in the larger structure, it is often easier to extract the data into separate variables.

```
cy5Data = magetfield(pd, 'F635 Median - B635');
cy3Data = magetfield(pd, 'F532 Median - B532');
```

A simple way to compare the two channels is with a `loglog` plot. The function `maloglog` is used to do this. Points that are above the diagonal in this plot correspond to genes that have higher expression levels in the H3 voxel than in the brain as a whole.

```
figure
maloglog(cy5Data, cy3Data)
title('Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```

```
Warning: Zero values are ignored.
Warning: Negative values are ignored.
```

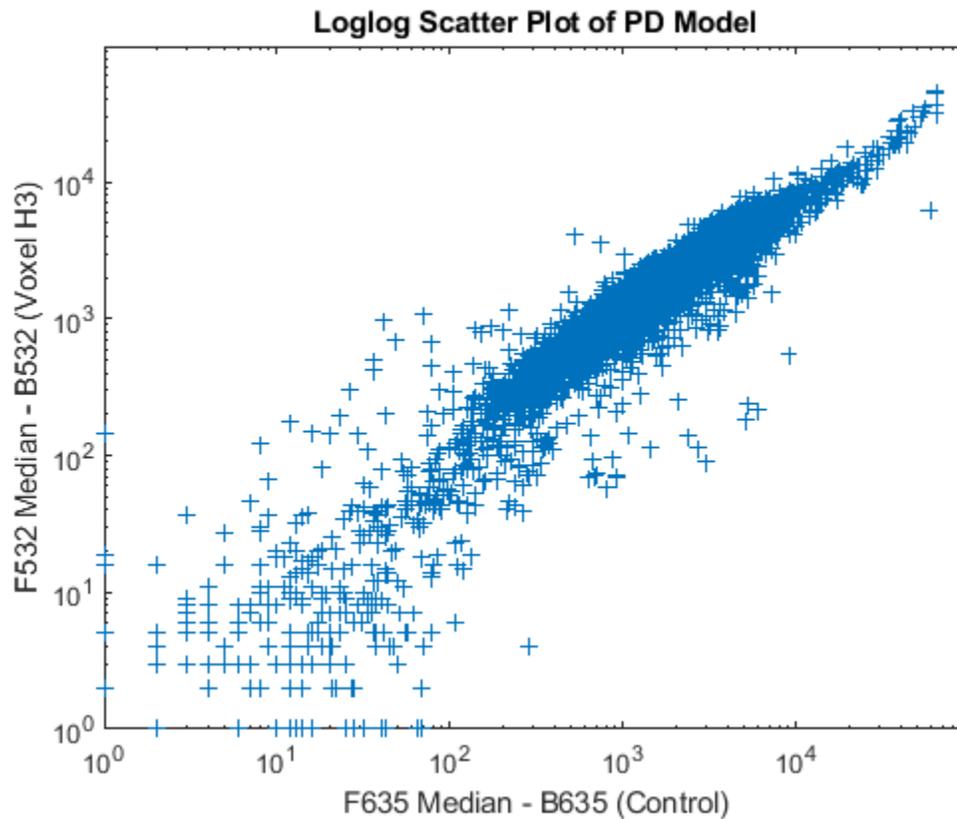


Notice how the `loglog` function gives some warnings about negative and zero elements. This is because some of the values in the 'F635 Median - B635' and 'F532 Median - B532' columns are zero or less than zero. Spots where this happened might be bad spots or spots that failed to hybridize. Similarly, spots with positive, but very small, differences between foreground and background are also considered bad spots. These warnings can be disabled using the warning command.

```
warnState = warning; % Save the current warning state
warning('off', 'bioinfo:maloglog:ZeroValues');
warning('off', 'bioinfo:maloglog:NegativeValues');
```

```
figure
maloglog(cy5Data, cy3Data)
title('Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```

```
warning(warnState); % Reset the warning state
```

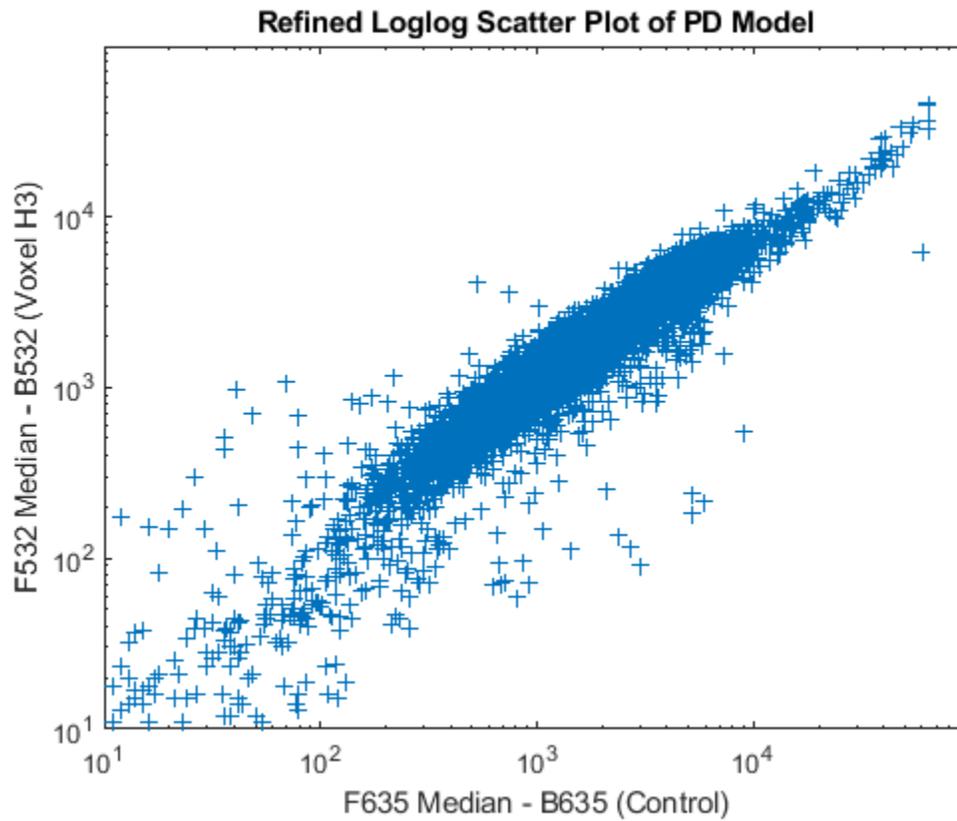


An alternative to simply ignoring or disabling the warnings is to remove the bad spots from the data set. This can be done by finding points where either the red or green channel have values less than or equal to a threshold value, for example 10.

```
threshold = 10;
badPoints = (cy5Data <= threshold) | (cy3Data <= threshold);
```

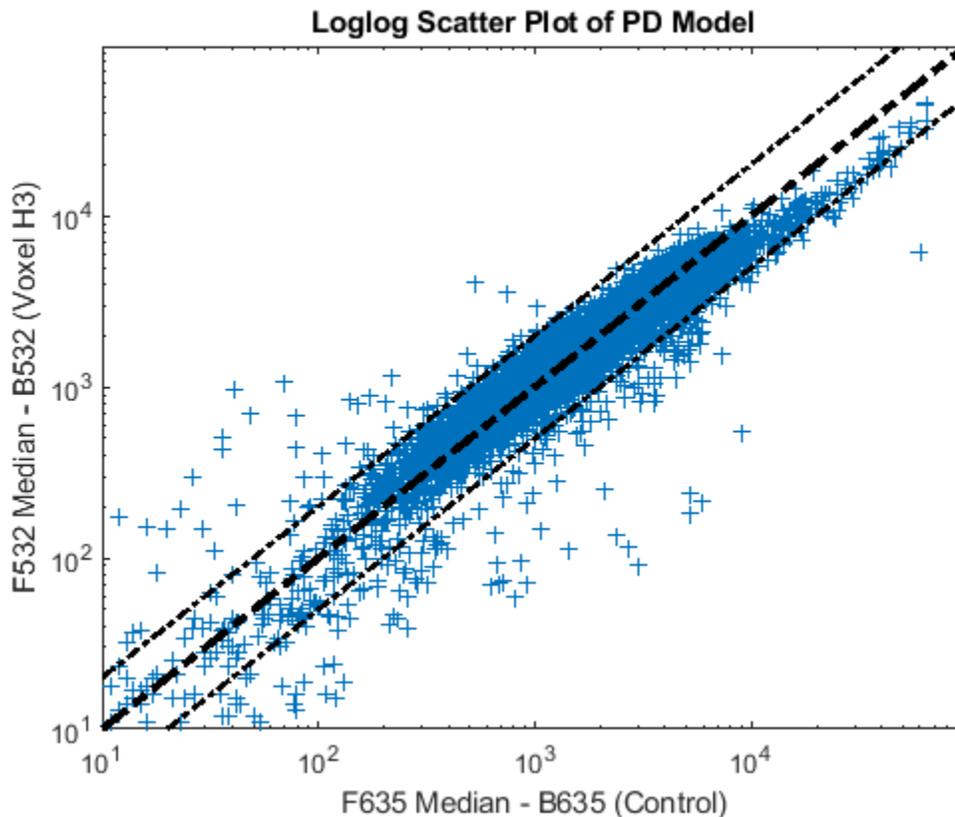
You can then remove these points and redraw the loglog plot.

```
cy5Data(badPoints) = []; cy3Data(badPoints) = [];
figure
maloglog(cy5Data,cy3Data)
title('Refined Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```



The distribution plot can be annotated by labeling the various points with the corresponding genes.

```
figure
maloglog(cy5Data,cy3Data,'labels',pd.Names(~badPoints),'factorlines',2)
title('Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```

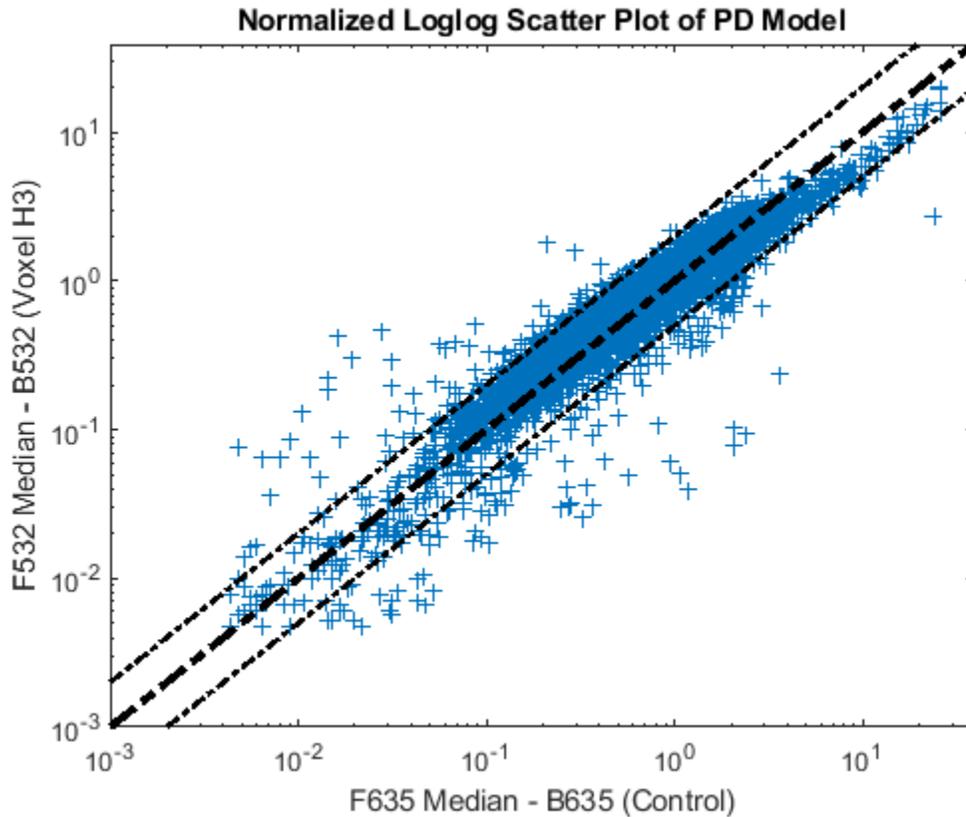


Try using the mouse to click on some of the outlier points. You will see the gene name associated with the point. Most of the outliers are below the $y = x$ line. In fact most of the points are below this line. Ideally the points should be evenly distributed on either side of this line. In order for this to happen, the points need to be normalized. You can use the `manorm` function to perform global mean normalization.

```
normcy5 = manorm(cy5Data);
normcy3 = manorm(cy3Data);
```

If you plot the normalized data you will see that the points are more evenly distributed about the $y = x$ line.

```
figure
maloglog(normcy5,normcy3,'labels',pd.Names(~badPoints),'factorlines',2)
title('Normalized Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```



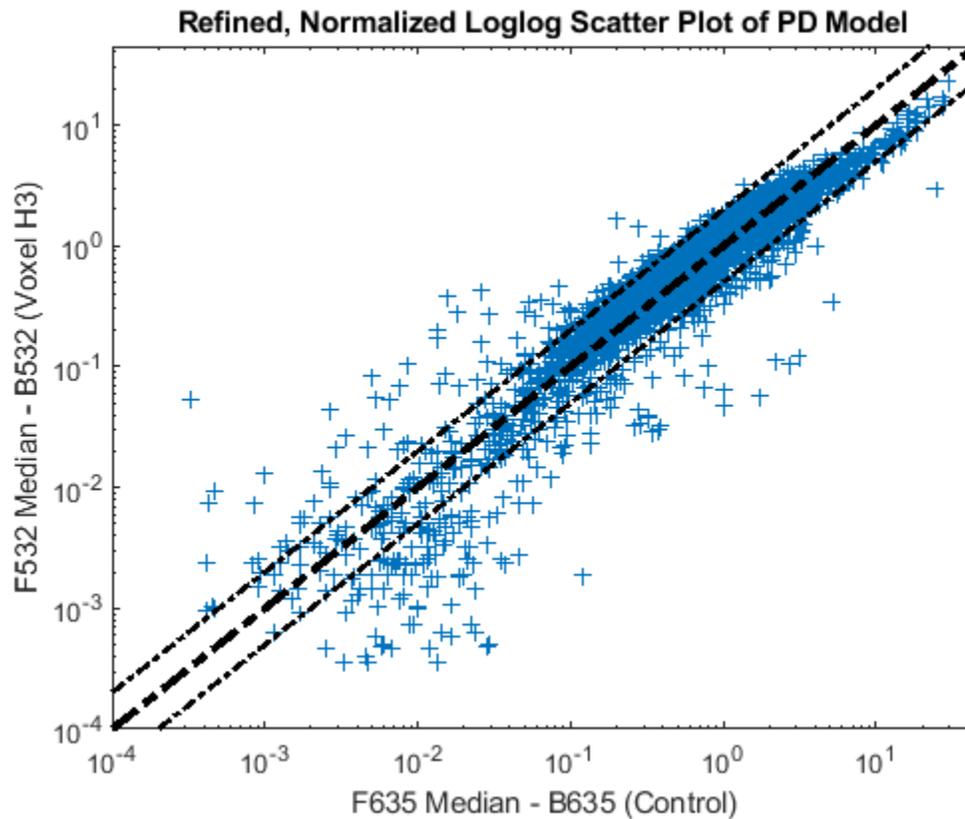
You will recall that the background of the chips was not uniform. You can use print-tip (block) normalization to normalize each block separately. The function `manorm` will perform block normalization automatically if block information is available in the microarray data structure.

```
bn_cy5Data = manorm(pd, 'F635 Median - B635');
bn_cy3Data = manorm(pd, 'F532 Median - B532');
```

Instead of removing negative or points below the threshold, you can set them to NaN. This does not change the size or shape of the data, but NaN points will not be displayed on plots.

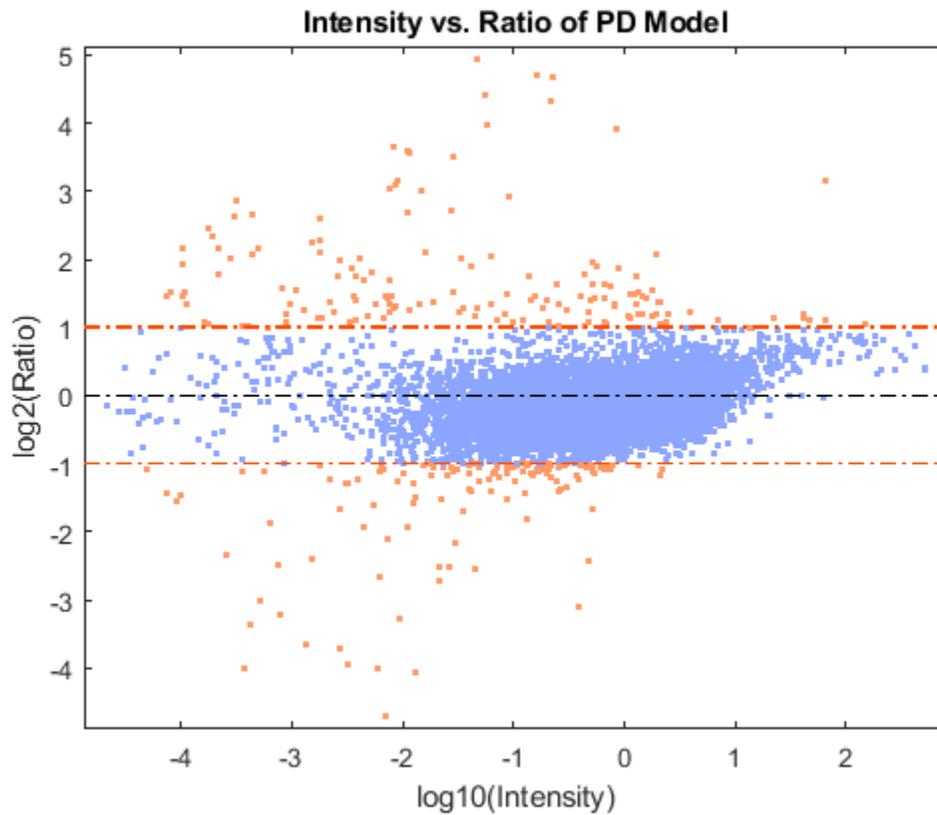
```
bn_cy5Data(bn_cy5Data <= 0) = NaN;
bn_cy3Data(bn_cy3Data <= 0) = NaN;
```

```
figure
maloglog(bn_cy5Data, bn_cy3Data, 'labels', pd.Names, 'factorlines', 2)
title('Refined, Normalized Loglog Scatter Plot of PD Model');
xlabel('F635 Median - B635 (Control)');
ylabel('F532 Median - B532 (Voxel H3)');
```



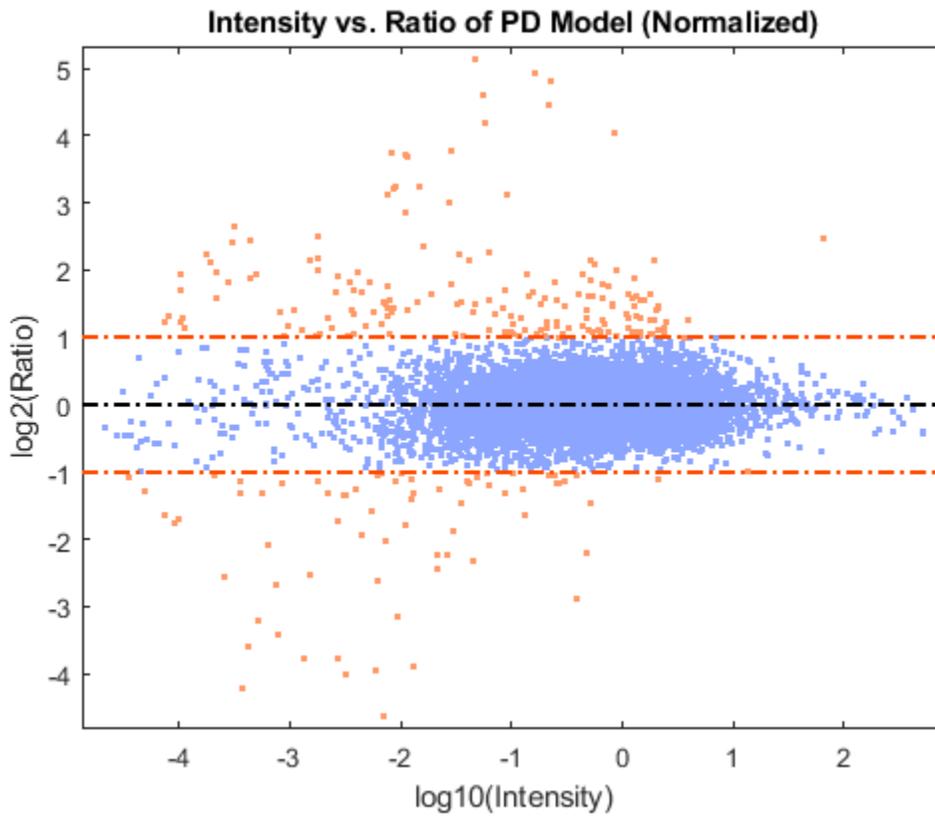
The function `mairplot` is used to create an Intensity vs. Ratio plot for the normalized data. If the name-value pair `'PlotOnly'` is set to false, you can explore the data interactively, such as select points to see the names of the associated genes, normalize the data, highlight gene names in the up-regulated or down-regulated lists, or change the values of the factor lines.

```
mairplot(normc5,normc3,'labels',pd.Names(~badPoints),'PlotOnly',true,...  
         'title','Intensity vs. Ratio of PD Model');
```



You can use the `Normalize` option to `mairplot` to perform Lowess normalization on the data.

```
mairplot(normcy5,normcy3,'labels',pd.Names(~badPoints),'PlotOnly',true,...  
         'Normalize',true,'title','Intensity vs. Ratio of PD Model (Normalized)');
```



GenePix is a registered trademark of Axon Instruments, Inc.

References

- [1] Brown, V.M., et al., "Multiplex three dimensional brain gene expression mapping in a mouse model of Parkinson's disease", Genome Research, 12(6):868-84, 2002.

Gene Expression Profile Analysis

This example shows a number of ways to look for patterns in gene expression profiles.

Exploring the Data Set

This example uses data from the microarray study of gene expression in yeast published by DeRisi, et al. 1997 [1]. The authors used DNA microarrays to study temporal gene expression of almost all genes in *Saccharomyces cerevisiae* during the metabolic shift from fermentation to respiration. Expression levels were measured at seven time points during the diauxic shift. The full data set can be downloaded from the Gene Expression Omnibus website, <https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE28>.

The MAT-file `yeastdata.mat` contains the expression values (\log_2 of ratio of CH2DN_MEAN and CH1DN_MEAN) from the seven time steps in the experiment, the names of the genes, and an array of the times at which the expression levels were measured.

```
load yeastdata.mat
```

To get an idea of the size of the data you can use `numel(genes)` to show how many genes are included in the data set.

```
numel(genes)
```

```
ans =
```

```
6400
```

You can access the genes names associated with the experiment by indexing the variable `genes`, a cell array representing the gene names. For example, the 15th element in `genes` is YAL054C. This indicates that the 15th row of the variable `yeastvalues` contains expression levels for YAL054C.

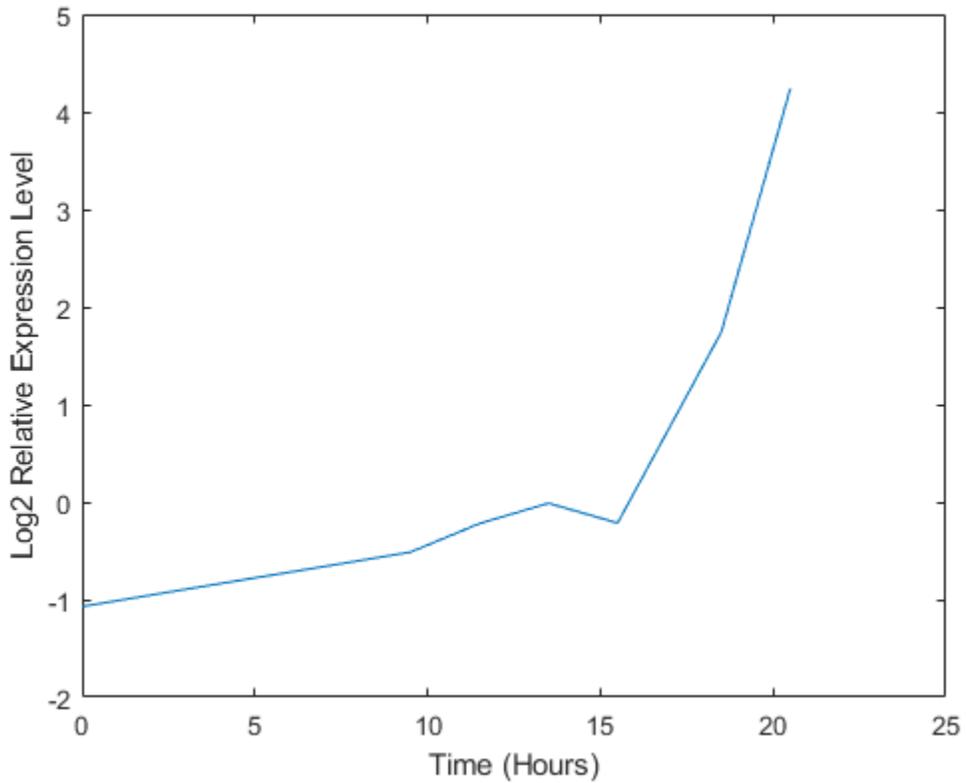
```
genes{15}
```

```
ans =
```

```
'YAL054C'
```

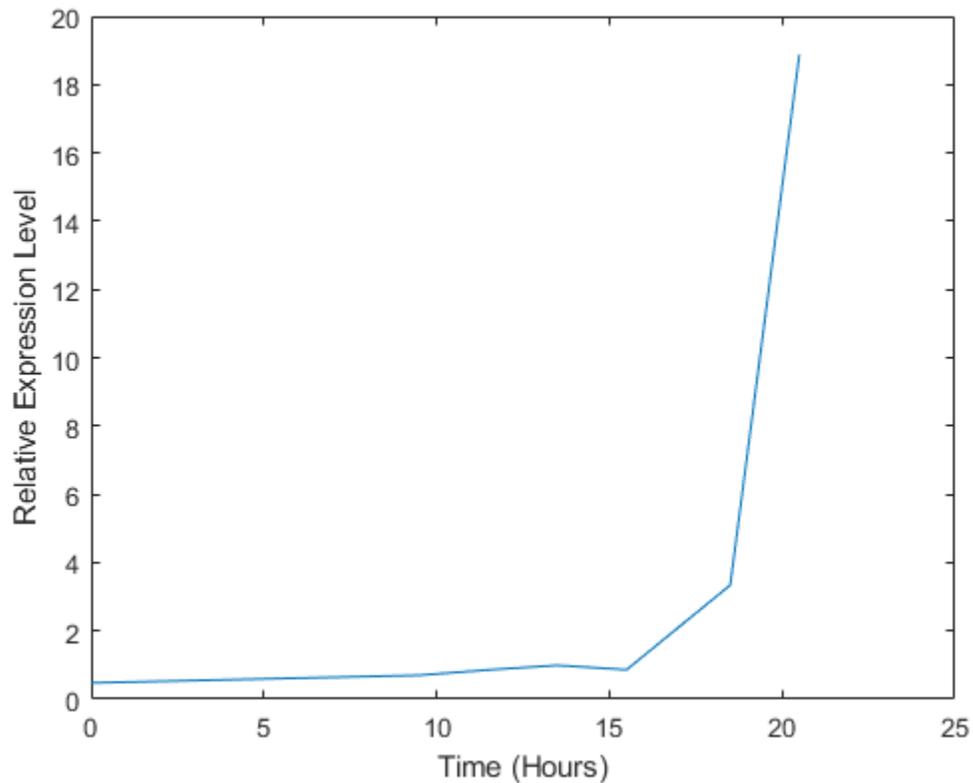
A simple plot can be used to show the expression profile for this ORF.

```
plot(times, yeastvalues(15,:))  
xlabel('Time (Hours)');  
ylabel('Log2 Relative Expression Level');
```



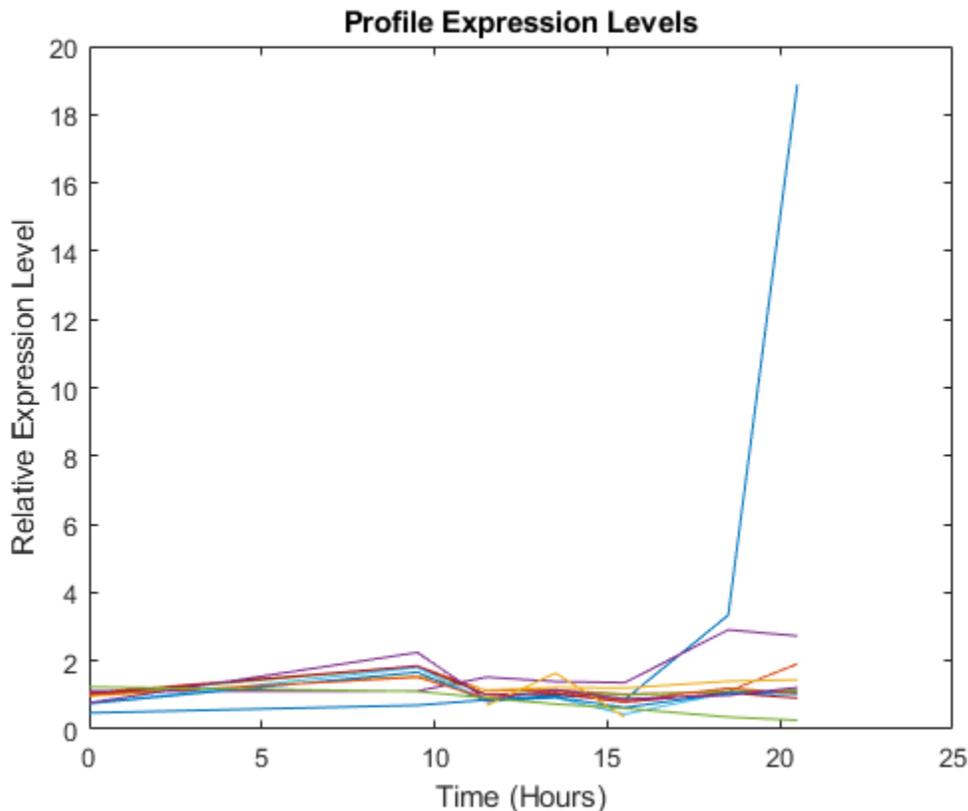
You can also plot the actual expression ratios, rather than the log2-transformed values.

```
plot(times, 2.^yeastvalues(15,:))  
xlabel('Time (Hours)');  
ylabel('Relative Expression Level');
```



The gene associated with this ORF, ACS1, appears to be strongly up-regulated during the diauxic shift. You can compare the expression of this gene to the expression of other genes by plotting multiple lines on the same figure.

```
hold on
plot(times, 2.^yeastvalues(16:26,:))
xlabel('Time (Hours)');
ylabel('Relative Expression Level');
title('Profile Expression Levels');
```



Filtering the Genes

Typically, a gene expression dataset includes information corresponding to genes that do not show any interesting changes during the experiment. To make it easier to find the interesting genes, you can reduce the size of the data set to some subset that contains only the most significant genes.

If you look through the gene list, you will see several spots marked as 'EMPTY'. These are empty spots on the array, and while they might have data associated with them, for the purposes of this example, you can consider these points to be noise. These points can be found using the `strcmp` function and removed from the data set with indexing commands.

```
emptySpots = strcmp('EMPTY',genes);
yeastvalues(emptySpots,:) = [];
genes(emptySpots) = [];
numel(genes)
```

```
ans =
```

```
6314
```

There are also see several places in the dataset where the expression level is marked as *NaN*. This indicates that no data was collected for this spot at the particular time step. One approach to dealing with these missing values would be to impute them using the mean or median of data for the particular gene over time. This example uses a less rigorous approach of simply throwing away the data for any genes where one or more expression level was not measured. The function `isnan` is used

to identify the genes with missing data and indexing commands are used to remove the genes with missing data.

```
nanIndices = any(isnan(yeastvalues),2);
yeastvalues(nanIndices,:) = [];
genes(nanIndices) = [];
numel(genes)
```

```
ans =
```

```
6276
```

If you were to plot the expression profiles of all the remaining profiles, you would see that most profiles are flat and not significantly different from the others. This flat data is obviously of use as it indicates that the genes associated with these profiles are not significantly affected by the diauxic shift; however, in this example, you are interested in the genes with large changes in expression accompanying the diauxic shift. You can use filtering functions in the Bioinformatics Toolbox™ to remove genes with various types of profiles that do not provide useful information about genes affected by the metabolic change.

You can use the `genevarfilter` function to filter out genes with small variance over time. The function returns a logical array (i.e., a mask) of the same size as the variable `genes` with ones corresponding to rows of `yeastvalues` with variance greater than the 10th percentile and zeros corresponding to those below the threshold. You can use the mask to index into the values and remove the filtered genes.

```
mask = genevarfilter(yeastvalues);
yeastvalues = yeastvalues(mask,:);
genes = genes(mask);
numel(genes)
```

```
ans =
```

```
5648
```

The function `genelowvalfilter` removes genes that have very low absolute expression values. Note that these filter functions can also automatically calculate the filtered data and names, so it is not necessary to index the original data using the mask.

```
[mask,yeastvalues,genes] = genelowvalfilter(yeastvalues,genes,'absval',log2(3));
numel(genes)
```

```
ans =
```

```
822
```

Finally, you can use the function `geneentropyfilter` to remove genes whose profiles have low entropy, for example entropy levels in the 15th percentile of the data.

```
[mask,yeastvalues,genes] = geneentropyfilter(yeastvalues,genes,'prctile',15);
numel(genes)
```

```
ans =  
    614
```

Cluster Analysis

Now that you have a manageable list of genes, you can look for relationships between the profiles using some different clustering techniques from the Statistics and Machine Learning Toolbox™. For hierarchical clustering, the function `pdist` calculates the pairwise distances between profiles and `linkage` creates the hierarchical cluster tree.

```
corrDist = pdist(yeastvalues, 'corr');  
clusterTree = linkage(corrDist, 'average');
```

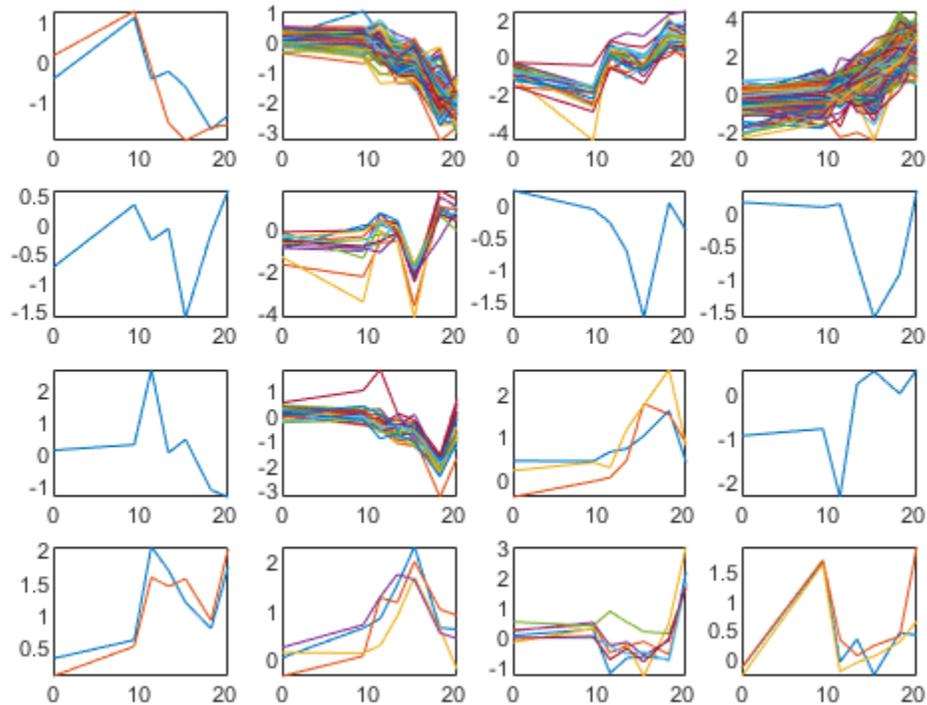
The `cluster` function calculates the clusters based on either a cutoff distance or a maximum number of clusters. In this case, the `maxclust` option is used to identify 16 distinct clusters.

```
clusters = cluster(clusterTree, 'maxclust', 16);
```

The profiles of the genes in these clusters can be plotted together using a simple loop and the `subplot` command.

```
figure  
for c = 1:16  
    subplot(4,4,c);  
    plot(times, yeastvalues((clusters == c),:));  
    axis tight  
end  
sgtitle('Hierarchical Clustering of Profiles');
```

Hierarchical Clustering of Profiles



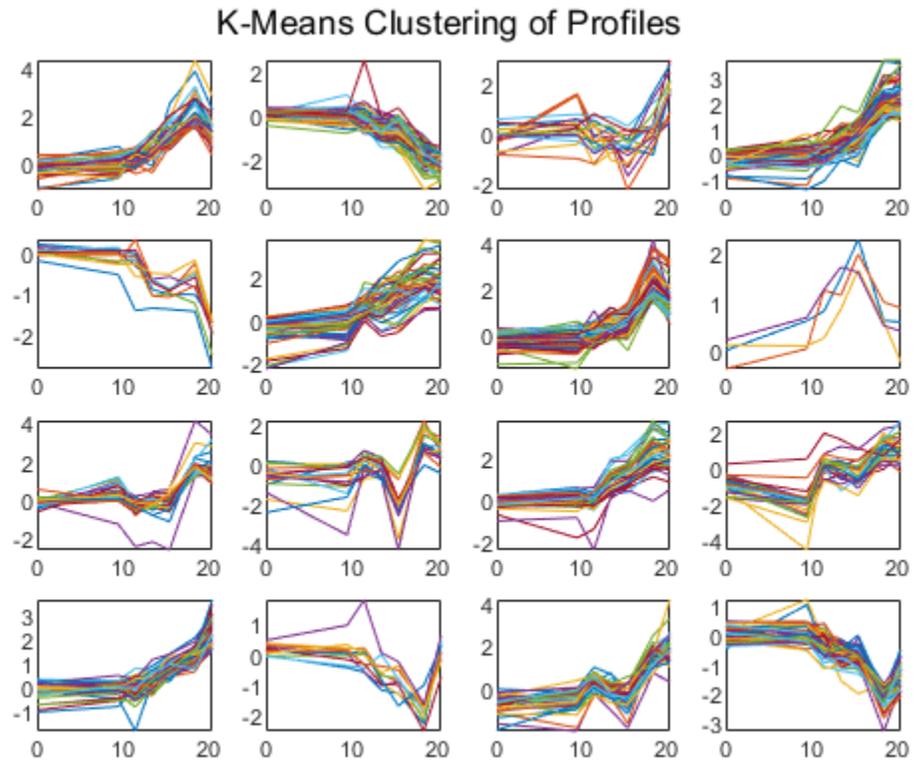
The Statistics and Machine Learning Toolbox also has a K-means clustering function. Again, sixteen clusters are found, but because the algorithm is different these will not necessarily be the same clusters as those found by hierarchical clustering.

Initialize the state of the random number generator to ensure that the figures generated by these command match the figures in the HTML version of this example.

```
rng('default');

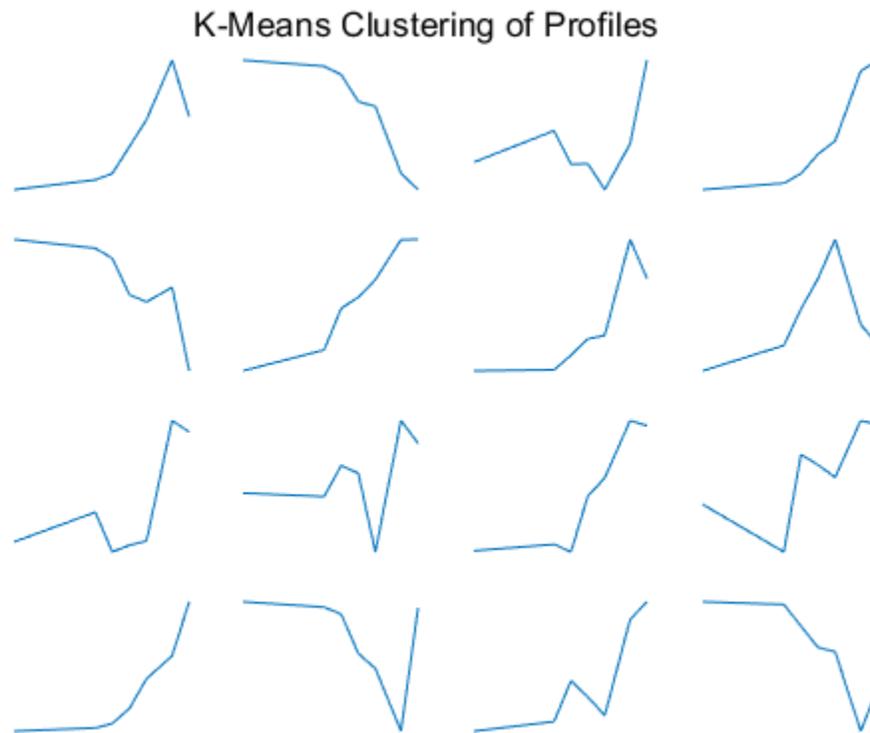
[cidx, ctrs] = kmeans(yeastvalues,16,'dist','corr','rep',5,'disp','final');
figure
for c = 1:16
    subplot(4,4,c);
    plot(times,yeastvalues((cidx == c),:));
    axis tight
end
sgtitle('K-Means Clustering of Profiles');
```

```
Replicate 1, 21 iterations, total sum of distances = 23.4699.
Replicate 2, 22 iterations, total sum of distances = 23.5615.
Replicate 3, 10 iterations, total sum of distances = 24.823.
Replicate 4, 28 iterations, total sum of distances = 23.4501.
Replicate 5, 19 iterations, total sum of distances = 23.5109.
Best total sum of distances = 23.4501
```



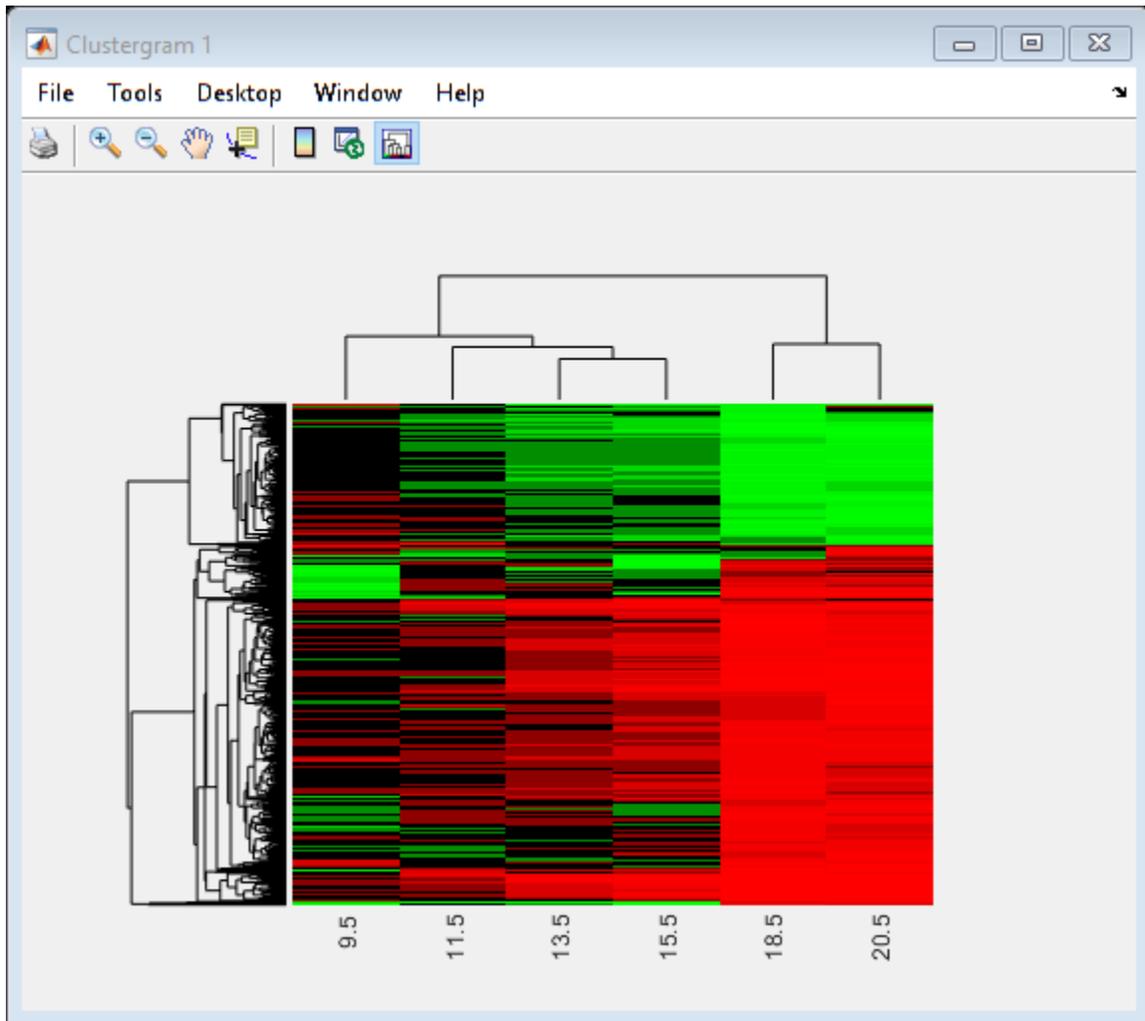
Instead of plotting all the profiles, you can plot just the centroids.

```
figure
for c = 1:16
    subplot(4,4,c);
    plot(times, ctrs(c,:));
    axis tight
    axis off
end
sgtitle('K-Means Clustering of Profiles');
```



You can use the `clustergram` function to create a heat map of the expression levels and a dendrogram from the output of the hierarchical clustering.

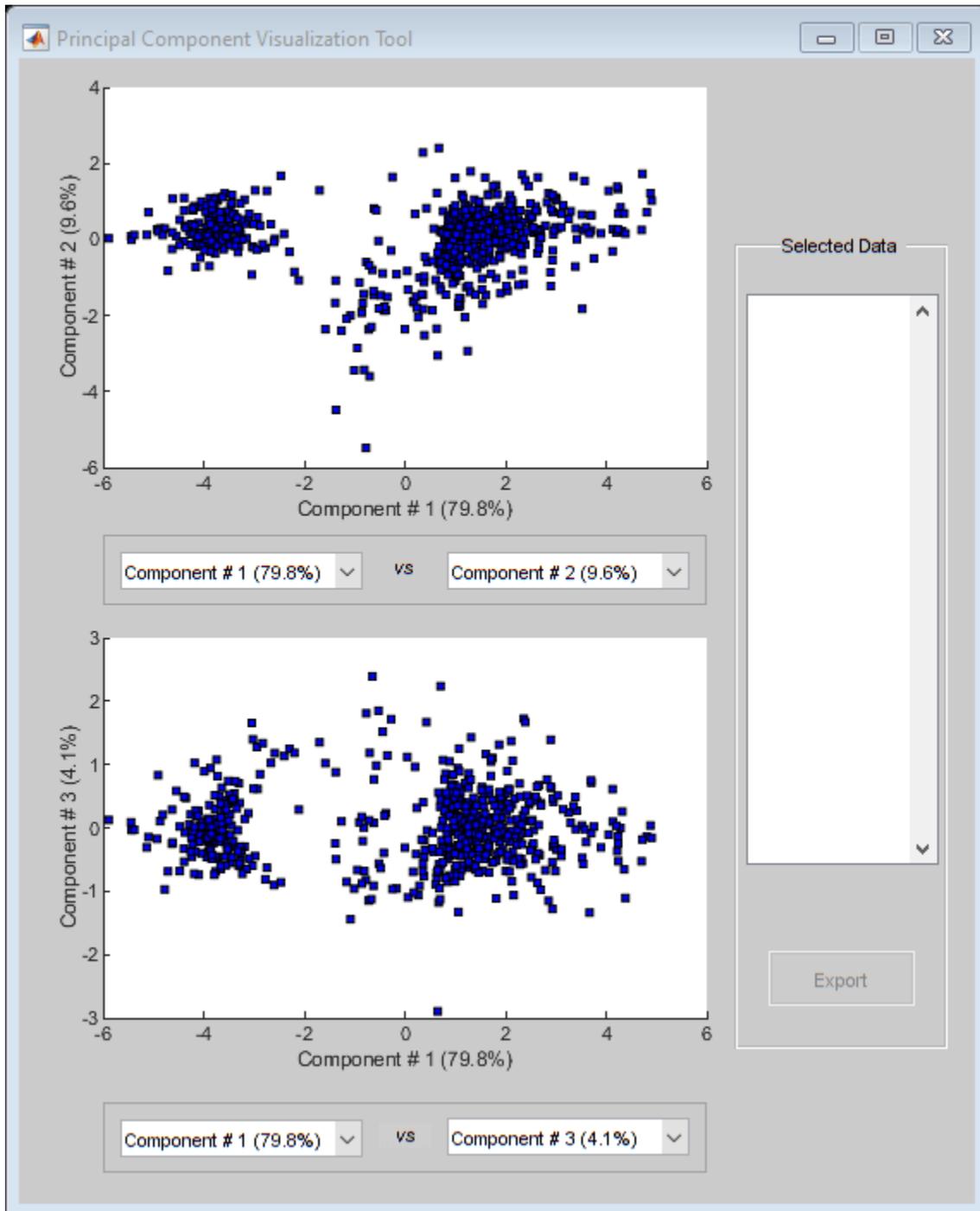
```
cgObj = clustergram(yeastvalues(:,2:end), 'RowLabels', genes, 'ColumnLabels', times(2:end));
```



Principal Component Analysis

Principal-component analysis (PCA) is a useful technique that can be used to reduce the dimensionality of large data sets, such as those from microarrays. PCA can also be used to find signals in noisy data. The function `mapcaplot` calculates the principal components of a data set and create scatter plots of the results. You can interactively select data points from one of the plots, and these points are automatically highlighted in the other plot. This lets you visualize multiple dimensions simultaneously.

```
h = mapcaplot(yeastvalues, genes);
```



Notice that the scatter plot of the scores of the first two principal components shows that there are two distinct regions. This is not unexpected as the filtering process removed many of the genes with low variance or low information. These genes would have appeared in the middle of the scatter plot.

If you want to look at the values of the principal components, the `pca` function in the Statistics and Machine Learning Toolbox is used to calculate the principal components of a data set.

```
[pc, zscores, pcvars] = pca(yeastvalues);
```

The first output, `pc`, is a matrix of the principal components of the `yeastvalues` data. The first column of the matrix is the first principal component, the second column is the second principal component, and so on. The second output, `zscores`, consists of the principal component scores, i.e., a representation of `yeastvalues` in the principal component space. The third output, `pcvars`, contains the principal component variances, which give a measure of how much of the variance of the data is accounted for by each of the principal components.

It is clear that the first principal component accounts for a majority of the variance in the model. You can compute the exact percentage of the variance accounted for by each component as shown below.

```
pcvars./sum(pcvars) * 100
```

```
ans =
```

```
79.8316  
9.5858  
4.0781  
2.6486  
2.1723  
0.9747  
0.7089
```

This means that almost 90% of the variance is accounted for by the first two principal components. You can use the `cumsum` command to see the cumulative sum of the variances.

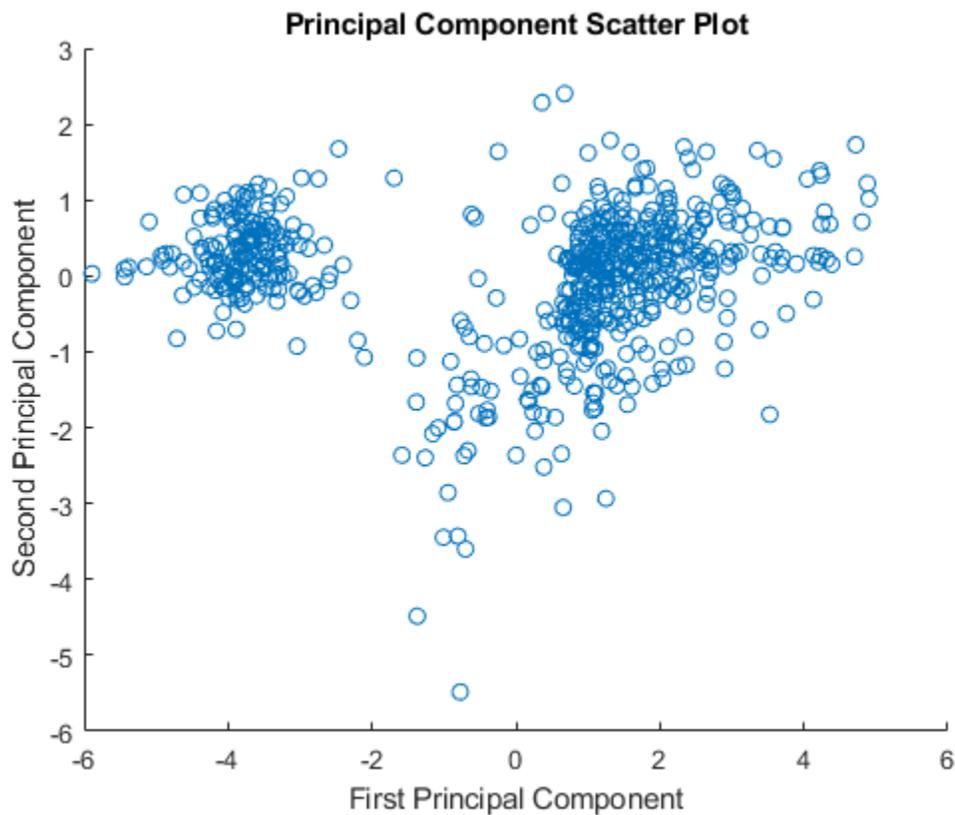
```
cumsum(pcvars./sum(pcvars) * 100)
```

```
ans =
```

```
79.8316  
89.4174  
93.4955  
96.1441  
98.3164  
99.2911  
100.0000
```

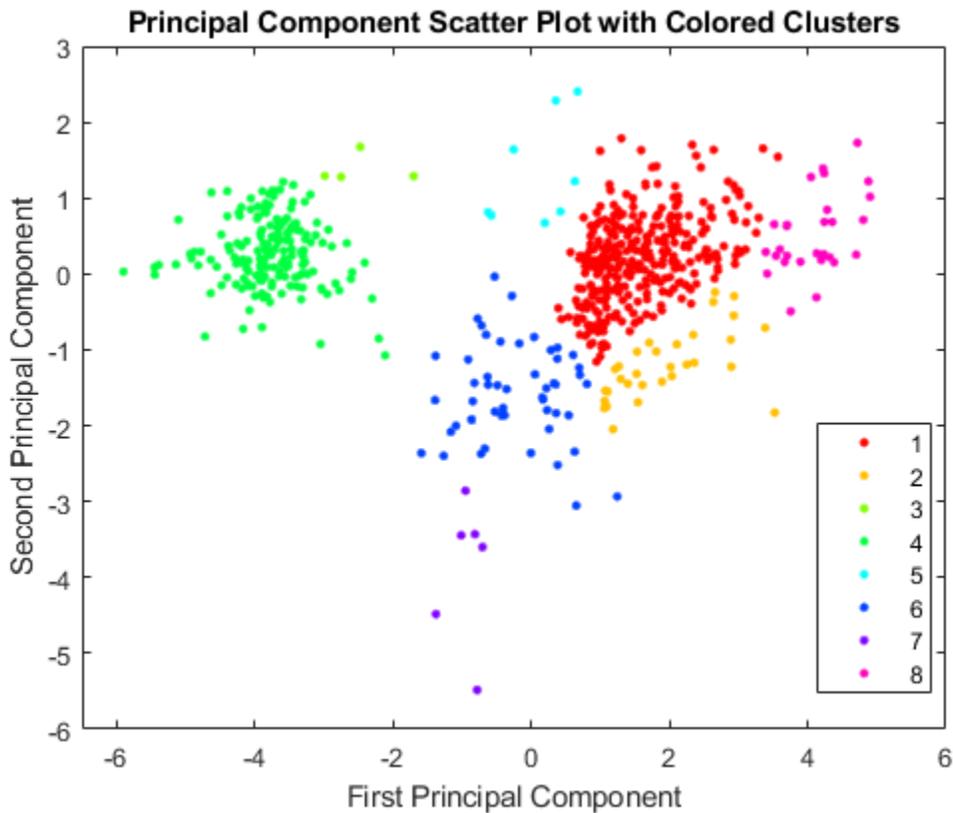
If you want to have more control over the plotting of the principal components, you can use the `scatter` function.

```
figure  
scatter(zscores(:,1),zscores(:,2));  
xlabel('First Principal Component');  
ylabel('Second Principal Component');  
title('Principal Component Scatter Plot');
```



An alternative way to create a scatter plot is with the function `gscatter` from the Statistics and Machine Learning Toolbox. `gscatter` creates a grouped scatter plot where points from each group have a different color or marker. You can use `clusterdata`, or any other clustering function, to group the points.

```
figure
pcclusters = clusterdata(zscores(:,1:2),'maxclust',8,'linkage','av');
gscatter(zscores(:,1),zscores(:,2),pcclusters,hsv(8))
xlabel('First Principal Component');
ylabel('Second Principal Component');
title('Principal Component Scatter Plot with Colored Clusters');
```



Self-Organizing Maps

If you have the Deep Learning Toolbox™, you can use a self-organizing map (SOM) to cluster the data.

```
% Check to see if the Deep Learning Toolbox is installed
if ~exist(which('selforgmap'),'file')
    disp('The Self-Organizing Maps section of this example requires the Deep Learning Toolbox.')
    return
end
```

The `selforgmap` function creates a new SOM network object. This example will generate a SOM using the first two principal components.

```
P = zscores(:,1:2)';
net = selforgmap([4 4]);
```

Train the network using the default parameters.

```
net = train(net,P);
```

Network Diagram

Training Results

Training finished: Reached maximum number of epochs ✔

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200
Elapsed Time	-	00:02:40	-

Training Algorithms

Data Division: Batch Weight/Bias Rule trainbu

Performance: Mean Squared Error mse

Calculations: MATLAB

Training Plots

_SOM

SOM Neighbor Connections

SOM Neighbor Distances

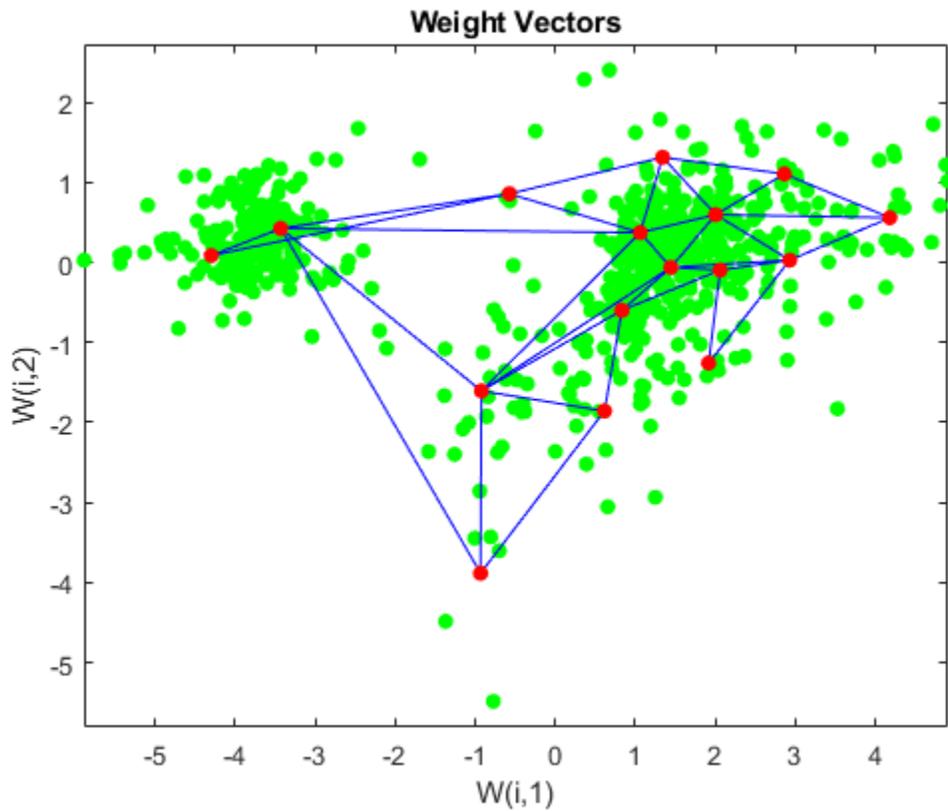
SOM Input Planes

SOM Sample Hits

SOM Weight Positions

Use `plotsom` to display the network over a scatter plot of the data. Note that the SOM algorithm uses random starting points so the results will vary from run to run.

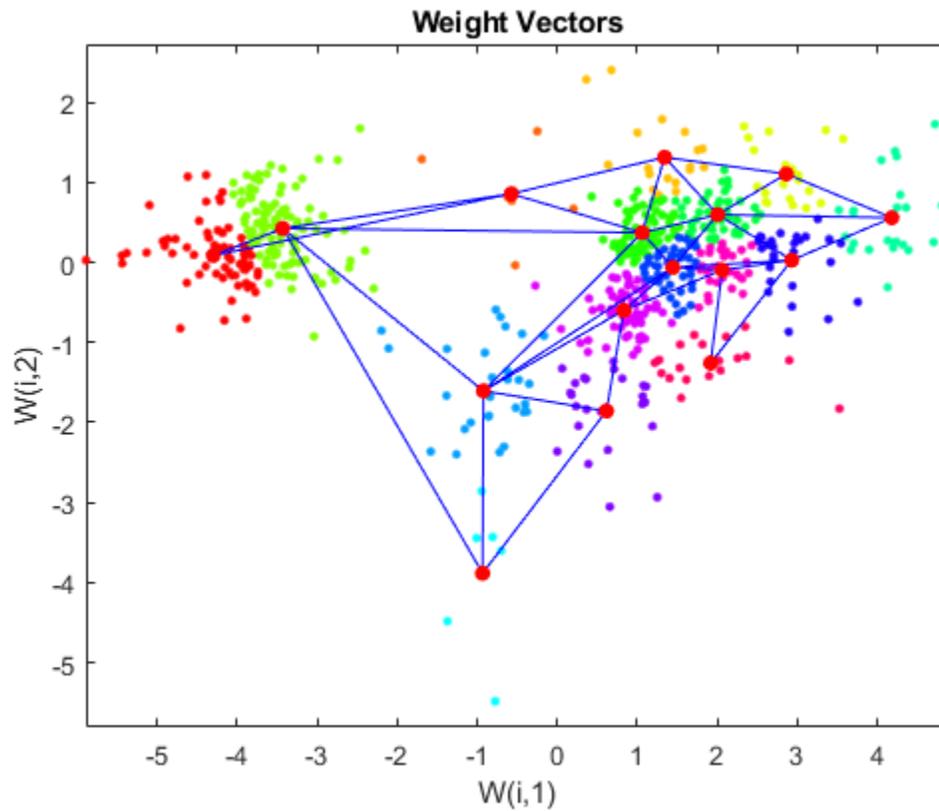
```
figure
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```



You can assign clusters using the SOM by finding the nearest node to each point in the data set.

```
distances = dist(P',net.IW{1}');
[d,cndx] = min(distances,[],2); % cndx contains the cluster index
```

```
figure
gscatter(P(1,:),P(2,:),cndx,hsv(numel(unique(cndx)))); legend off;
hold on
plotsom(net.iw{1,1},net.layers{1}.distances);
hold off
```



Close all figures.

```
close('all');  
delete(gcf);  
delete(h);
```

References

[1] DeRisi, J.L., Iyer, V.R. and Brown, P.O., "Exploring the metabolic and genetic control of gene expression on a genomic scale", *Science*, 278(5338):680-6, 1997.

Working with GEO Series Data

This example shows how to retrieve gene expression data series (GSE) from the NCBI Gene Expression Omnibus (GEO) and perform basic analysis on the expression profiles.

Introduction

The NCBI Gene Expression Omnibus (GEO) is the largest public repository of high-throughput microarray experimental data. GEO data have four entity types including GEO Platform (GPL), GEO Sample (GSM), GEO Series (GSE) and curated GEO DataSet (GDS).

A Platform record describes the list of elements on the array in the experiment (e.g., cDNAs, oligonucleotide probesets). Each Platform record is assigned a unique and stable GEO accession number (GPLxxx).

A Sample record describes the conditions under which an individual Sample was handled, the manipulations it underwent, and the abundance measurement of each element derived from it. Each Sample record is assigned a unique and stable GEO accession number (GSMxxx).

A Series record defines a group of related Samples and provides a focal point and description of the whole study. Series records may also contain tables describing extracted data, summary conclusions, or analyses. Each Series record is assigned a unique GEO accession number (GSExxx).

A DataSet record (GDSxxx) represents a curated collection of biologically and statistically comparable GEO Samples. GEO DataSets (GDSxxx) are curated sets of GEO Sample data.

More information about GEO can be found in GEO Overview. Bioinformatics Toolbox™ provides functions that can retrieve and parse GEO format data files. GSE, GSM, GSD and GPL data can be retrieved by using the `getgeodata` function. The `getgeodata` function can also save the retrieved data in a text file. GEO Series records are available in SOFT format files and in tab-delimited text format files. The function `geoseriesread` reads the GEO Series text format file. The `geosoftread` function reads the usually quite large SOFT format files.

In this example, you will retrieve the GSE5847 data set from GEO database, and perform statistical testing on the data. GEO Series GSE5847 contains experimental data from a gene expression study of tumor stroma and epithelium cells from 15 inflammatory breast cancer (IBC) cases and 35 non-inflammatory breast cancer cases (Boersma et al. 2008).

Retrieving GEO Series Data

The function `getgeodata` returns a structure containing data retrieved from the GEO database. You can also save the returned data in its original format to your local file system for use in subsequent MATLAB® sessions. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
gseData = getgeodata('GSE5847', 'ToFile', 'GSE5847.txt')
```

Use the `geoseriesread` function to parse the previously downloaded GSE text format file.

```
gseData = geoseriesread('GSE5847.txt')
```

```
gseData =
```

```

struct with fields:
  Header: [1x1 struct]
  Data: [22283x95 bioma.data.DataMatrix]

```

The structure returned contains a **Header** field that stores the metadata of the Series data, and a **Data** field that stores the Series matrix data.

Exploring GSE Data

The GSE5847 matrix data in the **Data** field are stored as a **DataMatrix** object. A **DataMatrix** object is a data structure like MATLAB 2-D array, but with additional metadata of row names and column names. The properties of a **DataMatrix** can be accessed like other MATLAB objects.

```

get(gseData.Data)
  Name: ''
  RowNames: {22283x1 cell}
  ColNames: {1x95 cell}
  NRows: 22283
  NCols: 95
  NDims: 2
  ElementClass: 'double'

```

The row names are the identifiers of the probe sets on the array; the column names are the GEO Sample accession numbers.

```
gseData.Data(1:5, 1:5)
```

```

ans =
      GSM136326   GSM136327   GSM136328   GSM136329   GSM136330
1007_s_at      10.45      9.3995      9.4248      9.4729      9.2788
1053_at        5.7195      4.8493      4.7321      4.7289      5.3264
117_at         5.9387      6.0833      6.448       6.1769      6.5446
121_at         8.0231      7.8947      8.345       8.1632      8.2338
1255_g_at      3.9548      3.9632      3.9641      4.0878      3.9989

```

The Series metadata are stored in the **Header** field. The structure contains Series information in the **Header.Series** field, and sample information in the **Header.Sample** field.

```
gseData.Header
```

```

ans =
  struct with fields:
    Series: [1x1 struct]
    Samples: [1x1 struct]

```

The **Series** field contains the title of the experiment and the microarray GEO Platform ID.

```
gseData.Header.Series
```

```
ans =
```

```
struct with fields:
```

```

        title: 'Tumor and stroma from breast by LCM'
    geo_accession: 'GSE5847'
        status: 'Public on Sep 30 2007'
    submission_date: 'Sep 15 2006'
    last_update_date: 'Nov 14 2012'
        pubmed_id: '17999412'
        summary: 'Tumor epithelium and surrounding stromal cells were isolated us
    overall_design: 'We applied LCM to obtain samples enriched in tumor epithelium an
        type: 'Expression profiling by array'
    contributor: 'Stefan,,Ambs-Brenda,,Boersma-Mark,,Reimers'
        sample_id: 'GSM136326 GSM136327 GSM136328 GSM136329 GSM136330 GSM136331 GSM
    contact_name: 'Stefan,,Ambs'
    contact_laboratory: 'LHC'
    contact_institute: 'NCI'
        contact_address: '37 Convent Dr Bldg 37 Room 3050'
        contact_city: 'Bethesda'
        contact_state: 'MD'
    contact_zip postal_code: '20892'
        contact_country: 'USA'
    supplementary_file: 'ftp://ftp.ncbi.nlm.nih.gov/pub/geo/DATA/supplementary/series/GS
        platform_id: 'GPL96'
    platform_taxid: '9606'
        sample_taxid: '9606'
        relation: 'BioProject: http://www.ncbi.nlm.nih.gov/bioproject/97251'

```

`gseData.Header.Samples`

```
ans =
```

```
struct with fields:
```

```

        title: {1×95 cell}
    geo_accession: {1×95 cell}
        status: {1×95 cell}
    submission_date: {1×95 cell}
    last_update_date: {1×95 cell}
        type: {1×95 cell}
    channel_count: {1×95 cell}
    source_name_ch1: {1×95 cell}
        organism_ch1: {1×95 cell}
    characteristics_ch1: {2×95 cell}
        molecule_ch1: {1×95 cell}
    extract_protocol_ch1: {1×95 cell}
        label_ch1: {1×95 cell}
    label_protocol_ch1: {1×95 cell}
        taxid_ch1: {1×95 cell}
    hyb_protocol: {1×95 cell}
    scan_protocol: {1×95 cell}
        description: {1×95 cell}
    data_processing: {1×95 cell}
        platform_id: {1×95 cell}

```

```

        contact_name: {1×95 cell}
    contact_laboratory: {1×95 cell}
    contact_institute: {1×95 cell}
    contact_address: {1×95 cell}
        contact_city: {1×95 cell}
        contact_state: {1×95 cell}
    contact_zip0x2Fpostal_code: {1×95 cell}
        contact_country: {1×95 cell}
    supplementary_file: {1×95 cell}
    data_row_count: {1×95 cell}

```

The `data_processing` field contains the information of the preprocessing methods, in this case the Robust Multi-array Average (RMA) procedure.

```
gseData.Header.Samples.data_processing(1)
```

```

ans =

    1×1 cell array

    {'RMA'}

```

The `source_name_ch1` field contains the sample source:

```

sampleSources = unique(gseData.Header.Samples.source_name_ch1);
sampleSources{:}

```

```

ans =

    'human breast cancer stroma'

```

```

ans =

    'human breast cancer tumor epithelium'

```

The field `Header.Samples.characteristics_ch1` contains the characteristics of the samples.

```
gseData.Header.Samples.characteristics_ch1(:,1)
```

```

ans =

    2×1 cell array

    {'IBC'      }
    {'Deceased'}

```

Determine the IBC and non-IBC labels for the samples from the `Header.Samples.characteristics_ch1` field as group labels.

```
sampleGrp = gseData.Header.Samples.characteristics_ch1(1,:);
```

Retrieving GEO Platform (GPL) Data

The Series metadata told us the array platform id: GPL96, which is an Affymetrix® GeneChip® Human Genome U133 array set HG-U133A. Retrieve the GPL96 SOFT format file from GEO using the `getgeodata` function. For example, assume you used the `getgeodata` function to retrieve the GPL96 Platform file and saved it to a file, such as `GPL96.txt`. Use the `geosoftread` function to parse this SOFT format file.

```
gplData = geosoftread('GPL96.txt')

gplData =
  struct with fields:
      Scope: 'PLATFORM'
      Accession: 'GPL96'
      Header: [1x1 struct]
      ColumnDescriptions: {16x1 cell}
      ColumnNames: {16x1 cell}
      Data: {22283x16 cell}
```

The `ColumnNames` field of the `gplData` structure contains names of the columns for the data:

```
gplData.ColumnNames

ans =
  16x1 cell array

  {'ID'
  {'GB_ACC'
  {'SPOT_ID'
  {'Species Scientific Name'
  {'Annotation Date'
  {'Sequence Type'
  {'Sequence Source'
  {'Target Description'
  {'Representative Public ID'
  {'Gene Title'
  {'Gene Symbol'
  {'ENTREZ_GENE_ID'
  {'RefSeq Transcript ID'
  {'Gene Ontology Biological Process'}
  {'Gene Ontology Cellular Component'}
  {'Gene Ontology Molecular Function'}
```

You can get the probe set ids and gene symbols for the probe sets of platform GPL69.

```
gplProbesetIDs = gplData.Data(:, strcmp(gplData.ColumnNames, 'ID'));
geneSymbols = gplData.Data(:, strcmp(gplData.ColumnNames, 'Gene Symbol'));
```

Use gene symbols to label the genes in the `DataMatrix` object `gseData.Data`. Be aware that the probe set IDs from the platform file may not be in the same order as in `gseData.Data`. In this example they are in the same order.

Change the row names of the expression data to gene symbols.

```
gseData.Data = rownames(gseData.Data, ':', geneSymbols);
```

Display the first five rows and five columns of the expression data with row names as gene symbols.

```
gseData.Data(1:5, 1:5)
```

```
ans =
```

	GSM136326	GSM136327	GSM136328	GSM136329	GSM136330
DDR1	10.45	9.3995	9.4248	9.4729	9.2788
RFC2	5.7195	4.8493	4.7321	4.7289	5.3264
HSPA6	5.9387	6.0833	6.448	6.1769	6.5446
PAX8	8.0231	7.8947	8.345	8.1632	8.2338
GUCA1A	3.9548	3.9632	3.9641	4.0878	3.9989

Analyzing the Data

Bioinformatics Toolbox and Statistics and Machine Learning Toolbox™ offer a wide spectrum of analysis and visualization tools for microarray data analysis. However, because it is not our main goal to explain the analysis methods in this example, you will apply only a few of the functions to the gene expression profile from stromal cells. For more elaborate examples about feature selection tools available, see “Select Features for Classifying High-Dimensional Data”.

The experiment was performed on IBC and non-IBC samples derived from stromal cells and epithelial cells. In this example, you will work with the gene expression profile from stromal cells. Determine the sample indices for the stromal cell type from the `gseData.Header.Samples.source_name_ch1` field.

```
stromaIdx = strcmpi(sampleSources{1}, gseData.Header.Samples.source_name_ch1);
```

Determine number of samples from stromal cells.

```
nStroma = sum(stromaIdx)
```

```
nStroma =
```

```
47
```

```
stromaData = gseData.Data(:, stromaIdx);
stromaGrp = sampleGrp(stromaIdx);
```

Determine the number of IBC and non-IBC stromal cell samples.

```
nStromaIBC = sum(strcmp('IBC', stromaGrp))
```

```
nStromaIBC =
```

```
13
```

```
nStromaNonIBC = sum(strcmp('non-IBC', stromaGrp))
```

```
nStromaNonIBC =
```

```
34
```

You can also label the columns in `stromaData` with the group labels:

```
stromaData = colnames(stromaData, ':', stromaGrp);
```

Display the histogram of the normalized gene expression measurements of a specified gene. The x-axes represent the normalized expression level. For example, inspect the distribution of the gene expression values of these genes.

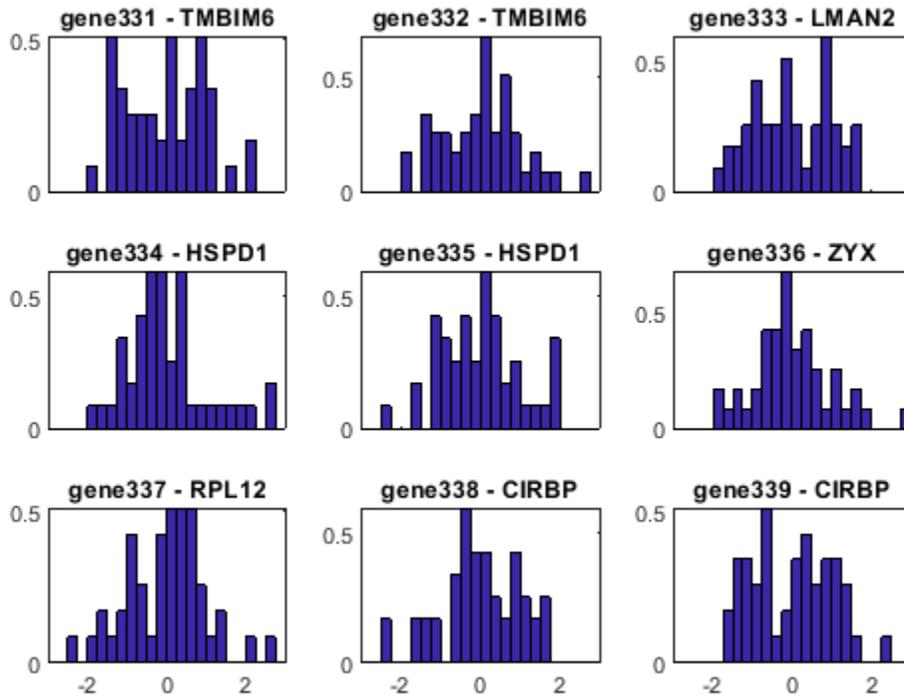
```
fID = 331:339;

zValues = zscore(stromaData.(':')(':'), 0, 2);
bw = 0.25;
edges = -10:bw:10;
bins = edges(1:end-1) + diff(edges)/2;

histStroma = histc(zValues(fID, :)', edges) ./ (stromaData.NCols*bw);

figure;
for i = 1:length(fID)
    subplot(3,3,i);
    bar(edges, histStroma(:,i), 'histc')
    xlim([-3 3])
    if i <= length(fID)-3
        ax = gca;
        ax.XTickLabel = [];
    end
    title(sprintf('gene%d - %s', fID(i), stromaData.RowNames{fID(i)}))
end
sgtitle('Gene Expression Value Distributions')
```

Gene Expression Value Distributions



The gene expression profile was accessed using the Affymetrix GeneChip more than 22,000 features on a small number of samples (~100). Among the 47 tumor stromal samples, there are 13 IBC and 34 non-IBC. But not all the genes are differentially expressed between IBC and non-IBC tumors. Statistical tests are needed to identify a gene expression signature that distinguish IBC from non-IBC stromal samples.

Use `genevarfilter` to filter out genes with a small variance across samples.

```
[mask, stromaData] = genevarfilter(stromaData);
```

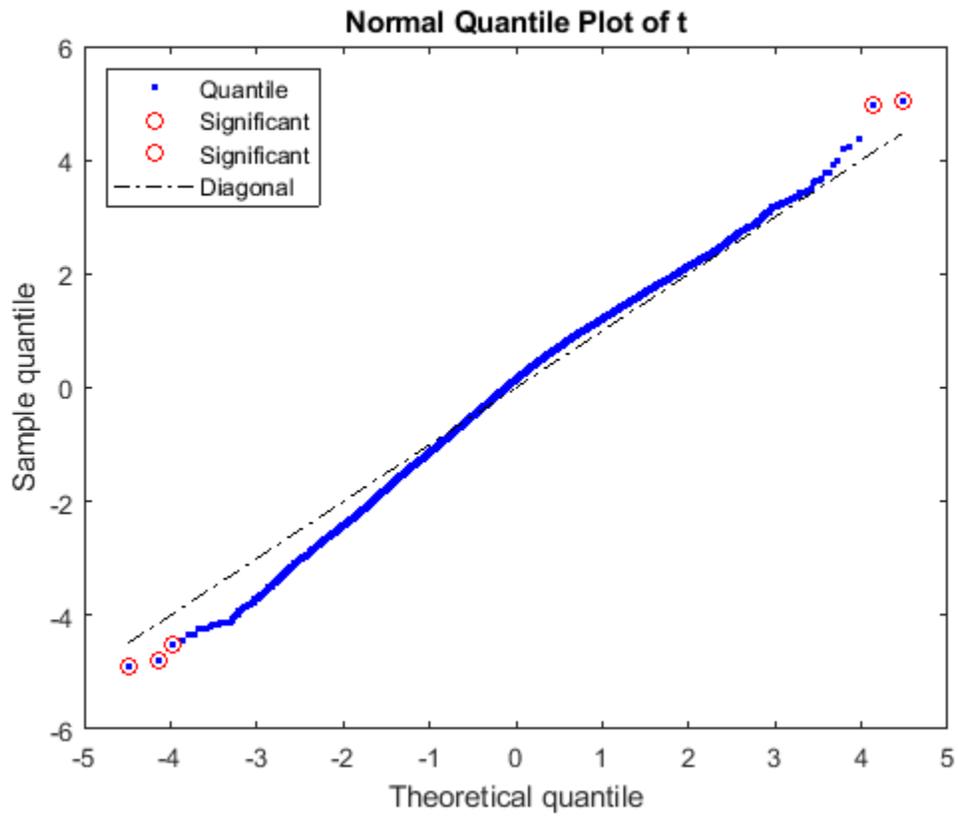
```
stromaData.NRows
```

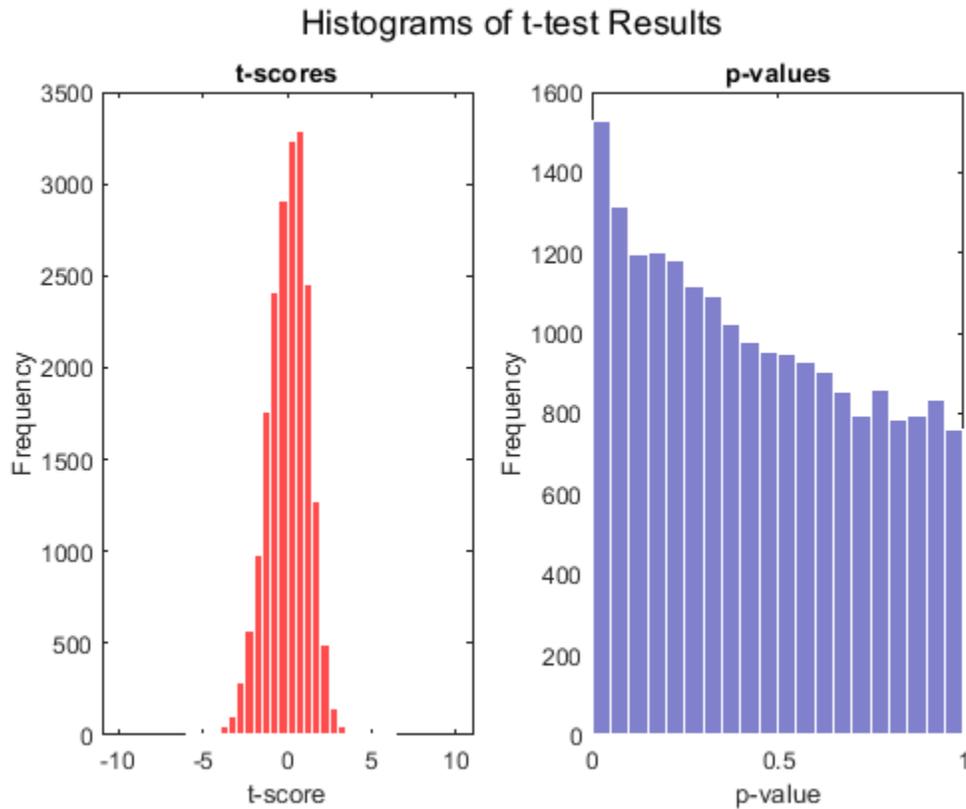
```
ans =
```

```
20055
```

Apply a t-statistic on each gene and compare *p-values* for each gene to find significantly differentially expressed genes between IBC and non-IBC groups by permuting the samples (1,000 times for this example).

```
rng default
[pvalues, tscores] = mattest(stromaData(:, 'IBC'), stromaData(:, 'non-IBC'),...
    'Showhist', true, 'showplot', true, 'permute', 1000);
```





Select the genes at a specified p-value.

```
sum(pvalues < 0.001)
```

```
ans =
```

```
52
```

There are about 50 genes selected directly at $p\text{-values} < 0.001$.

Sort and list the top 20 genes:

```
testResults = [pvalues, tscores];
testResults = sortrows(testResults);
testResults(1:20, :)
```

```
ans =
```

	p-values	t-scores
INPP5E	2.3318e-05	5.0389
ARFRP1 /// IGLJ3	2.7575e-05	4.9753
USP46	3.4336e-05	-4.9054
GOLGB1	4.7706e-05	-4.7928
TTC3	0.00010695	-4.5053
THUMPD1	0.00013164	-4.4317

	0.00016042	4.3656
MAGED2	0.00017042	-4.3444
DNAJB9	0.0001782	-4.3266
KIF1C	0.00022122	4.2504
	0.00022237	-4.2482
DZIP3	0.00022414	-4.2454
COPB1	0.00023199	-4.2332
PSD3	0.00024649	-4.2138
PLEKHA4	0.00026505	4.186
DNAJB9	0.0002767	-4.1708
CNPY2	0.0002801	-4.1672
USP9X	0.00028442	-4.1619
SEC22B	0.00030146	-4.1392
GFER	0.00030506	-4.1352

References

[1] Boersma, B.J., Reimers, M., Yi, M., Ludwig, J.A., et al. "A stromal gene signature associated with inflammatory breast cancer", *International Journal of Cancer*, 122(6):1324-32, 2008.

Identifying Biomolecular Subgroups Using Attractor Metagenes

This example shows workflows for the analysis of gene expression data with the attractor metagene algorithm. Gene expression data is available for many model organisms and disease conditions. This example shows how to use the `metafeatures` function to explore biomolecular phenotypes in breast cancer.

The Cancer Genome Atlas Data

The Cancer Genome Atlas (TCGA) includes several kinds of data across multiple cancer indications. TCGA includes measurements of gene expression, protein expression, clinical outcomes, and more. In this example, you explore breast cancer gene expression.

Researchers collected tumor samples, and used Agilent G4502A microarrays to measure their gene expression. In this example you use the Level-3 expression data, which has been post-processed from the original measurements into the expression calls. Data was retrieved May 20, 2014.

Load the data into MATLAB®. The MAT-file `TCGA_Breast_Gene_Expression.mat` contains gene expression data of 17814 genes for 590 different patients. The expression data is stored in the variable `geneExpression`. The gene names are stored in the variable `geneNames`.

```
load TCGA_Breast_Gene_Expression
```

To see for the organization of the data, check number of genes and samples in this data set.

```
size(geneExpression)
ans = 1×2
      17814      590
```

`geneNames` is a cell array of the gene names. You can access the entries using MATLAB cell array indexing:

```
geneNames{655}
ans =
'EGFR'
```

This cell array indicates that the 655th row of the variable `geneExpression` contains expression measurements for the gene expression of Epidermal Growth Factor Receptor (EGFR).

Attractor Metagene Algorithm

The attractor metagene algorithm was developed as part of the DREAM 8 challenge to develop prognostic biomarkers for breast cancer survival. The attractor metagene approach discovers and quantifies underlying biomolecular events. These events reduce the dimensionality of the gene expression data, and also allow for subtype classification and investigation of regulatory machinery [1].

A metagene is defined as any weighted sum of gene expression. Suppose you have a collection of co-expressed genes. You can create a metagene by averaging the expression levels of the genes in the collection.

There is the potential to refine our understanding of the gene expression captured in this metagene. Suppose you create a set of weights that quantify the similarity between the genes in our collection and the metagene. Genes that are more similar to the metagene receive larger weights, while genes that are less similar receive smaller weights. Using these new weights, you can form a new metagene that is a weighted average of gene expression. The new metagene better captures a biomolecular event that governs some element of gene regulation in the expression data.

This procedure forms the core of the attractor metagene algorithm. Form a metagene using some current estimate of the weights, then update the weights based on a measure of similarity. Attractor metagenes are defined as the attracting fixed points of this iterative process.

The algorithm exists within the broad family of unsupervised machine learning algorithms. Related algorithms include principal component analysis, various clustering algorithms (especially fuzzy c-means), non-negative matrix factorization, and others. The main advantage of the metagene approach is that the results of the algorithm tend to be more clearly linked with a phenotype defined by gene expression.

Concretely, in the i th iteration of the algorithm. You have a vector of weights, W_i , of size 1-by-number of genes. The estimate of the metagene during the i th iteration is:

$$M_i = W_i * G$$

G is the number of genes by number of samples gene expression matrix. To update the weights:

$$W_{j,i+1} = J(M_i, G_j)$$

$W_{j,i+1}$ is the j th element of W_{i+1} , G_j is the j th row of G , and J is a similarity metric. In the metagene attractor algorithm, J is defined as:

$$J(M_i, G_j) = MI(M_i, G_j)^\alpha$$

if the correlation between M_i and G_j is greater than 0. MI is the mutual information between M_i and G_j . The function `metafeatures` uses the B-spline estimator of mutual information described in [3].

If, instead, the correlation between M_i and G_j is less than or equal to 0, then:

$$J(M_i, G_j) = 0$$

The weights are all greater than or equal to zero. Because mutual information is scale invariant, you can normalize the weights in whatever way you choose. Here, they are normalized so their sum is 1.

The algorithm is initialized by either random or user-selected weights. It proceeds until the change in M_i between iterations is small, or a prespecified number of iterations is exhausted.

Cleaning the Data

The data has several NaN values. To check how many, sum over an indicator returned by `isnan`.

```
sum(sum(isnan(geneExpression)))
```

```
ans =  
1695
```

Out of the approximately 10 million entries of `geneExpression`, there are 1695 missing entries. Before proceeding you will need to deal with these missing entries.

There are several ways to impute these missing values. You can use a simple method called K nearest neighbor imputation supplied by the Bioinformatics Toolbox (TM). K-nearest neighbor imputation works by replacing missing data with the corresponding value from a weighted average of the k nearest columns to the column with the missing data.

Use $k = 3$, and replace the current value of `geneExpression` with one that has no NaN values.

```
geneExpression = knnimpute(geneExpression,3);
```

The variable `geneExpression` has no NaN values.

```
sum(sum(isnan(geneExpression)))
```

```
ans =  
0
```

For more information about `knnimpute`, see the Bioinformatics Toolbox documentation.

doc `knnimpute`

Identifying Biomolecular Events Using the Attractor Metagene Algorithm

The function `metafeatures` uses the attractor metagene algorithm to identify motifs of gene regulation.

Setup an options structure. In this case, set the `display` to provide the information about the algorithm at each iteration.

```
opts = struct('Display','iter');
```

`metafeatures` also allows for specifying start values. You can seed the starting weights to emphasize genes that you are interested in. There are three common drivers of breast cancer, ERBB2 (also called HER2), estrogen, and progesterone.

Set the weight for each of these genes to 1 in three different rows of `startValues`. Each row corresponds to initial values for a different replicate. `strcmp` compares the genes of interest and the list of genes in the data set. `find` returns the index in the list of the gene.

```
erbb          = find(strcmp('ERBB2',geneNames));  
estrogen      = find(strcmp('ESR1',geneNames));  
progesterone  = find(strcmp('PGR',geneNames));
```

```
startValues = zeros(size(geneExpression,1),3);  
startValues(erbb,1)      = 1;  
startValues(estrogen,2)  = 1;  
startValues(progesterone,3) = 1;
```

Call `metafeatures` with the imputed data set. The second argument, `geneNames` is the list of all the genes in the data set. Supplying the gene names is not required. However, the gene names can allow exploration of the highly ranked genes that are returned by the algorithm to get insights into the biomolecular event described by the metagene.

```
[meta, weights, genes_sorted] = metafeatures(geneExpression,geneNames,'start',startValues,'option
```

```
Caching self information ...  
... done. Took 26.2175 seconds.  
Caching entropy and binning information...  
... done. Took 14.1267 seconds.
```

Found	iter	non-zero diff	weights
1	1	1.26e+01	8924
1	2	7.29e+00	8885
1	3	4.22e+00	8796
1	4	2.54e+00	8761
1	5	1.63e+00	8745
1	6	1.14e+00	8720
1	7	8.59e-01	8706
1	8	7.18e-01	8682
1	9	7.04e-01	8687
1	10	6.44e-01	8680
1	11	5.53e-01	8676
1	12	4.56e-01	8664
1	13	3.67e-01	8654
1	14	2.91e-01	8649
1	15	2.30e-01	8642
1	16	1.83e-01	8636
1	17	1.46e-01	8634
1	18	1.17e-01	8631
1	19	9.45e-02	8632
1	20	7.65e-02	8634
1	21	6.22e-02	8633
1	22	5.06e-02	8631
1	23	4.13e-02	8635
1	24	3.38e-02	8639
1	25	2.76e-02	8636
1	26	2.26e-02	8633
1	27	1.85e-02	8633
1	28	1.51e-02	8635
1	29	1.24e-02	8635
1	30	1.02e-02	8634
1	31	8.35e-03	8633
1	32	6.85e-03	8633
1	33	5.57e-03	8633
1	34	4.59e-03	8631
1	35	3.78e-03	8631
1	36	3.07e-03	8632
1	37	2.53e-03	8632
1	38	2.06e-03	8632
1	39	1.70e-03	8632
1	40	1.40e-03	8632
1	41	1.15e-03	8632
1	42	9.24e-04	8632
1	43	7.70e-04	8632
1	44	6.21e-04	8632
1	45	5.20e-04	8632
1	46	4.43e-04	8632
1	47	3.49e-04	8632
1	48	2.97e-04	8632
1	49	2.36e-04	8632
1	50	1.93e-04	8632
1	51	1.56e-04	8632
1	52	1.42e-04	8632
1	53	8.98e-05	8632
1	54	9.72e-05	8632
1	55	5.37e-05	8632
1	56	7.47e-05	8632

1	57	5.17e-05	8632
1	58	4.81e-05	8632
1	59	2.85e-05	8632
1	60	1.97e-05	8632
1	61	3.05e-05	8632
1	62	1.41e-05	8632
1	63	1.02e-05	8632
1	64	7.89e-06	8632
1	65	9.34e-06	8632
1	66	2.07e-05	8632
1	67	1.52e-05	8632
1	68	2.26e-05	8632
1	69	1.55e-05	8632
1	70	2.24e-05	8632
1	71	1.75e-05	8632
1	72	2.01e-05	8632
1	73	6.47e-06	8632
1	74	1.62e-05	8632
1	75	2.23e-05	8632
1	76	1.93e-05	8632
1	77	1.71e-05	8632
1	78	6.94e-06	8632
1	79	3.21e-06	8632
1	80	1.58e-05	8632
1	81	2.02e-05	8632
1	82	1.99e-05	8632
1	83	2.12e-05	8632
1	84	1.79e-05	8632
1	85	1.60e-05	8632
1	86	1.78e-05	8632
1	87	1.87e-05	8632
1	88	1.66e-05	8632
1	89	5.98e-06	8632
1	90	1.26e-05	8632
1	91	2.14e-05	8632
1	92	1.82e-05	8632
1	93	6.97e-06	8632
1	94	1.04e-05	8632
1	95	2.13e-05	8632
1	96	6.39e-06	8632
1	97	1.75e-05	8632
1	98	2.37e-05	8632
1	99	2.01e-05	8632
1	100	1.98e-05	8632

Warning: 'Maximum iterations exceeded, terminating early.'

2	1	1.93e+01	9893
2	2	6.04e+00	9885
2	3	3.80e+00	9883
2	4	2.53e+00	9886
2	5	1.73e+00	9881
2	6	1.13e+00	9873
2	7	7.19e-01	9869
2	8	4.63e-01	9866
2	9	3.08e-01	9870
2	10	2.13e-01	9874
2	11	1.54e-01	9872

2	12	1.15e-01	9874
2	13	8.72e-02	9874
2	14	6.68e-02	9874
2	15	5.14e-02	9874
2	16	3.97e-02	9875
2	17	3.07e-02	9875
2	18	2.37e-02	9873
2	19	1.84e-02	9871
2	20	1.42e-02	9871
2	21	1.10e-02	9871
2	22	8.54e-03	9872
2	23	6.62e-03	9872
2	24	5.05e-03	9872
2	25	4.01e-03	9872
2	26	3.09e-03	9872
2	27	2.38e-03	9872
2	28	1.85e-03	9872
2	29	1.43e-03	9872
2	30	1.09e-03	9872
2	31	8.46e-04	9872
2	32	6.73e-04	9872
2	33	5.10e-04	9872
2	34	3.81e-04	9872
2	35	2.98e-04	9872
2	36	2.46e-04	9872
2	37	1.51e-04	9872
2	38	1.63e-04	9872
2	39	1.15e-04	9872
2	40	7.11e-05	9872
2	41	1.18e-04	9872
2	42	7.28e-05	9872
2	43	1.89e-05	9872
2	44	4.24e-05	9872
2	45	1.60e-05	9872
2	46	6.75e-06	9872
2	47	4.81e-05	9872
2	48	2.47e-05	9872
2	49	1.04e-05	9872
2	50	7.46e-06	9872
2	51	9.31e-06	9872
2	52	5.25e-06	9872
2	53	3.89e-05	9872
2	54	9.38e-06	9872
2	55	3.33e-05	9872
2	56	1.48e-05	9872
2	57	2.45e-05	9872
2	58	2.58e-05	9872
2	59	1.00e-05	9872
2	60	1.86e-05	9872
2	61	5.87e-05	9872
2	62	2.97e-05	9872
2	63	1.07e-05	9872
2	64	8.84e-06	9872
2	65	8.29e-06	9872
2	66	1.58e-05	9872
2	67	1.48e-05	9872
2	68	5.00e-06	9872
2	69	2.74e-05	9872

2	70	1.20e-05	9872
2	71	2.91e-05	9872
2	72	9.45e-06	9872
2	73	1.75e-05	9872
2	74	1.56e-05	9872
2	75	6.56e-06	9872
2	76	1.79e-05	9872
2	77	2.67e-05	9872
2	78	5.55e-05	9872
2	79	2.55e-05	9872
2	80	1.03e-05	9872
2	81	2.74e-05	9872
2	82	2.04e-05	9872
2	83	1.00e-05	9872
2	84	1.11e-05	9872
2	85	9.83e-06	9872
2	86	2.71e-05	9872
2	87	1.42e-05	9872
2	88	1.28e-05	9872
2	89	2.24e-05	9872
2	90	4.58e-05	9872
2	91	3.36e-05	9872
2	92	9.74e-06	9872
2	93	1.06e-05	9872
2	94	1.50e-05	9872
2	95	5.05e-05	9872
2	96	1.12e-05	9872
2	97	2.52e-05	9872
2	98	9.77e-06	9872
2	99	6.10e-06	9872
2	100	2.97e-05	9872

Warning: 'Maximum iterations exceeded, terminating early.'

3	1	3.75e+00	9963
3	2	1.08e+00	9966
3	3	4.29e-01	9959
3	4	1.87e-01	9961
3	5	8.45e-02	9958
3	6	3.88e-02	9957
3	7	1.80e-02	9956
3	8	8.36e-03	9956
3	9	3.89e-03	9956
3	10	1.78e-03	9956
3	11	8.68e-04	9956
3	12	3.96e-04	9956
3	13	1.89e-04	9956
3	14	8.92e-05	9956
3	15	4.25e-05	9956
3	16	1.16e-05	9956
3	17	1.57e-05	9956
3	18	1.67e-05	9956
3	19	1.59e-05	9956
3	20	1.07e-05	9956
3	21	9.21e-06	9956
3	22	1.59e-05	9956
3	23	6.23e-06	9956
3	24	8.68e-06	9956

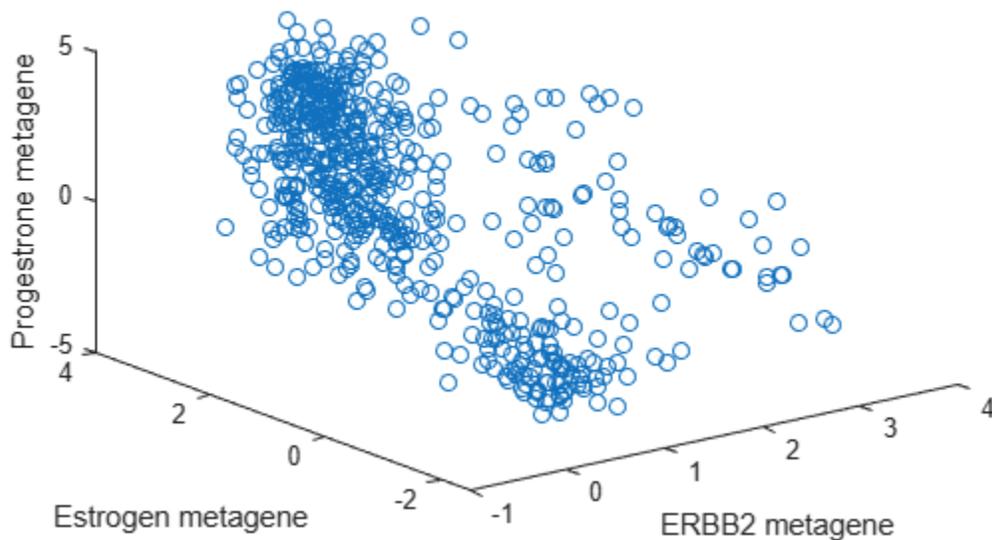
3	25	1.56e-05	9956
3	26	1.55e-05	9956
3	27	9.65e-06	9956
3	28	9.74e-06	9956
3	29	9.75e-06	9956
3	30	9.75e-06	9956
3	31	9.84e-06	9956
3	32	1.49e-05	9956
3	33	1.05e-05	9956
3	34	1.43e-05	9956
3	35	2.14e-05	9956
3	36	6.64e-06	9956
3	37	1.54e-06	9956
3	38	2.23e-06	9956
3	39	2.98e-06	9956
3	40	8.89e-06	9956
3	41	1.60e-05	9956
3	42	1.06e-05	9956
3	43	9.08e-06	9956
3	44	1.60e-05	9956
3	45	6.74e-06	9956
3	46	8.71e-06	9956
3	47	9.45e-06	9956
3	48	1.48e-05	9956
3	49	1.05e-05	9956
3	50	1.43e-05	9956
3	51	2.15e-05	9956
3	52	6.64e-06	9956
3	53	1.41e-06	9956
3	54	1.98e-06	9956
3	55	2.58e-06	9956
3	56	9.07e-06	9956
3	57	6.54e-06	9956
3	58	5.44e-06	9956
3	59	4.36e-06	9956
3	60	8.43e-06	9956
3	61	1.08e-05	9956
3	62	9.73e-06	9956
3	63	9.72e-06	9956
3	64	9.72e-06	9956
3	65	9.75e-06	9956
3	66	9.78e-06	9956
3	67	9.82e-06	9956
3	68	1.50e-05	9956
3	69	1.02e-05	9956
3	70	1.34e-05	9956
3	71	2.08e-05	9956
3	72	1.30e-05	9956
3	73	2.11e-05	9956
3	74	1.56e-05	9956
3	75	9.45e-06	9956
3	76	1.48e-05	9956
3	77	1.11e-05	9956
3	78	8.97e-06	9956
3	79	1.31e-05	9956
3	80	2.19e-05	9956
3	81	8.99e-06	9956
3	82	1.60e-05	9956

3	83	7.51e-06	9956
3	84	6.78e-06	9956
3	85	7.51e-06	9956
3	86	1.10e-05	9956
3	87	1.39e-05	9956
3	88	6.38e-06	9956
3	89	6.05e-06	9956
3	90	4.66e-06	9956
3	91	7.28e-06	9956
3	92	7.98e-06	9956
3	93	1.15e-05	9956
3	94	8.72e-06	9956
3	95	1.56e-05	9956
3	96	1.82e-05	9956
3	97	1.23e-05	9956
3	98	6.69e-06	9956
3	99	1.63e-06	9956
3	100	1.15e-06	9956

Warning: 'Maximum iterations exceeded, terminating early.'

The variable `meta` has the value of the three metagenes discovered for each sample. You can plot the three metagenes to gain insight into the nature of gene regulation across different phenotypes of breast cancer.

```
plot3(meta(1,:),meta(2,:),meta(3:),'o')
xlabel('ERBB2 metagene')
ylabel('Estrogen metagene')
zlabel('Progesterone metagene')
```



In the plot you can observe a few things.

In the plot, there is a group of points bunched together with low values for all three metagenes. Based on mRNA levels, the expectation is that points are associated with tumor samples that are triple-negative or basal type.

There is also a group of points that have high estrogen receptor metagene expression. This group spans both high and low progesterone metagene expression. There are no points with high progesterone metagene expression and low estrogen metagene expression. This finding is consistent with the observation that ER-/PR+ breast cancers are extremely rare [2].

The remaining points are the ERBB2 positive cancers. They have less representation in this data set than the hormone-driven and triple-negative cancers. There are also no firmly established relationships between hormone receptor expression and ERBB2 status.

To develop a better understanding of the gene regulation captured by the metagenes, take a closer look at the metagene discovered by initializing the estrogen receptor to have weight 1. You can list the top ten genes contributing to the metagene for the 11th metagene discovered.

```
genes_sorted(1:10,2)
```

```
ans = 10x1 cell
      {'AGR3' }
      {'ESR1' }
      {'CA12' }
      {'AGR2' }
      {'MLPH' }
      {'FOX A1' }
      {'THSD4' }
      {'FSIP1' }
      {'ANXA9' }
      {'XBP1' }
```

This metagene captures the biomolecular event associated with the transition to estrogen-driven breast cancer. The four, top-ranked, genes listed are:

- Anterior Gradient Homolog 3 (AGR3)
- Estrogen Receptor 1 (ESR1)
- Carbonic anhydrase 12 (CA12)
- Anterior Gradient Homolog 2 (AGR2)

Transcriptional changes in each of these genes are implicated in estrogen-driven breast cancer. The three genes other than ESR1 are known to be coexpressed with ESR1. Identification of these genes illustrates the power of the attractor metagene algorithm to link gene expression with phenotypes.

Similar versions of the estrogen metagene and the ERBB2 metagene are described in [1]. The ordering of the gene contributions differs slightly between this analysis and [1] because a different breast cancer data set was used. Variations in the weights are to be expected, but the ordering of the genes by weights are roughly the same. Specifically, genes with the top 10 weights are mostly the same between this version, and the version described in [1]. Similarly, there is significant overlap between the genes with the top 100 weights.

Genes can contribute to multiple metagenes. In this sense, the attractor metagene algorithm is a "soft" clustering technique. In this example, finding metagenes in breast cancer data, there is overlap in the sets of genes that have larger contribution weights to the estrogen and progesterone metagenes.

If a weight is "elevated" when it is larger than .001, then:

```
elevated_weights = weights>.001;
```

The column sum of the `elevated_weights` is the total number of elevated weights in each of the three metagenes.

```
sum(elevated_weights)
```

```
ans = 1×3
```

```
    19    96    27
```

Of the 96 elevated weights for the estrogen metagene, and the 27 for the progesterone metagene, there are 22 elevated weights that are in both sets.

```
sum(elevated_weights(:,2) & elevated_weights(:,3))
```

```
ans =
```

```
22
```

However, there is no overlap between the ERBB2 metagene and the estrogen metagene:

```
sum(elevated_weights(:,1) & elevated_weights(:,2))
```

```
ans =
```

```
0
```

as well as no overlap between the ERBB2 metagene and the progesterone metagene:

```
sum(elevated_weights(:,1) & elevated_weights(:,3))
```

```
ans =
```

```
0
```

The Role of Alpha

In the similarity metric of the algorithm, the parameter alpha controls the degree of nonlinearity. As alpha is increased, the number of metagenes tends to increase. The default alpha is 5, because this value was good for the work in [1], but for different data sets or use cases, you must adjust alpha.

To illustrate the effects of alpha, if alpha is 1 in the breast cancer analysis, then the progesterone and estrogen metagenes are not distinct.

```
[meta_alpha_1, weights_alpha_1, genes_sorted_alpha_1] = ...
    metafeatures(geneExpression, geneNames, 'start', startValues, 'alpha', 1);
```

```
Warning: 'Maximum iterations exceeded, terminating early.'
```

```
Warning: 'Maximum iterations exceeded, terminating early.'
```

```
Warning: 'Maximum iterations exceeded, terminating early.'
```

In this case, only two metagenes are returned, despite the fact that we ran the algorithm three times.

```
size(meta_alpha_1)
```

```
ans = 1×2
```

This result is because, by default, `metafeatures` returns only the unique metagenes. The initialization with the weight for ESR1 set to 1, and the initialization with the weight for PGR set to 1, both converge to metagenes that are effectively the same.

References

- [1] Cheng, Wei-Yi, Tai-Hsien Ou Yang, and Dimitris Anastassiou. "Biomolecular events in cancer revealed by attractor metagenes." *PLoS computational biology* 9.2 (2013): e1002920.
- [2] Hefti, Marco M., et al. "Estrogen receptor negative/progesterone receptor positive breast cancer is not a reproducible subtype." *Breast Cancer Research* 15.4 (2013): R68.
- [3] Daub, Carsten O., et al. "Estimating mutual information using B-spline functions?an improved similarity measure for analysing gene expression data." *BMC bioinformatics* 5.1 (2004): 118.

Working with the Clustergram Function

This example shows how to work with the `clustergram` function.

The `clustergram` function creates a heat map with dendrograms to show hierarchical clustering of data. These types of heat maps have become a standard visualization method for microarray data since first applied by Eisen et al. [1]. This example illustrates some of the options of the `clustergram` function. The example uses data from the van't Veer et al. breast cancer microarray study [2].

Importing Data

A study by van't Veer et al. investigated whether tumor ability for metastasis is obtained later in development or inherent in the initial gene expression signature [2]. The study analyzed tumor samples from 117 young breast cancer patients, of whom 78 were sporadic lymph-node-negative. The gene expression profiles of these 78 patients were searched for prognostic signatures. Of the 78 patients, 44 exhibited non-recurrences within five years of surgical treatment while 34 had recurrences. Samples were hybridized to Agilent® two-color oligonucleotide microarrays representing approximately 25,000 human genes. The authors selected 4,918 significant genes that had at least a two-fold differential expression relative to the reference and a p-value for being expressed < 0.01 in at least 3 samples. By using supervised classification, the authors identified a poor prognosis gene expression signature of 231 genes [2].

A subset of the preprocessed gene expression data from [2] is provided in the `bc_train_filtered.mat` MAT-file. Samples for 78 lymph-node-negative patients are included, each one containing the gene expression values for the 4,918 significant genes. Gene expression values have already been preprocessed, by normalization and background subtraction, as described in [2].

```
load bc_train_filtered
bcTrainData

bcTrainData =

    struct with fields:

        Samples: {78×1 cell}
        Log10Ratio: [4918×78 single]
        Accession: {4918×1 cell}
```

The list of 231 genes in the prognosis profile proposed by van't Veer et al. is also provided in the `bc_proggenes231.mat` MAT-file. Genes are ordered according to their correlation coefficient with the prognostic groups.

```
load bc_proggenes231
```

Extract the gene expression values for the prognosis profile.

```
[tf, idx] = ismember(bcProgGeneList.Accession, bcTrainData.Accession);
progValues = bcTrainData.Log10Ratio(idx, :);
progAccession = bcTrainData.Accession(idx);
progSamples = bcTrainData.Samples;
```

For this example, you will work with the 35 most positive correlated genes and the 35 most negative correlated genes.

```
progValues = progValues([1:35 197:231],:);
progAccession = progAccession([1:35 197:231]);
```

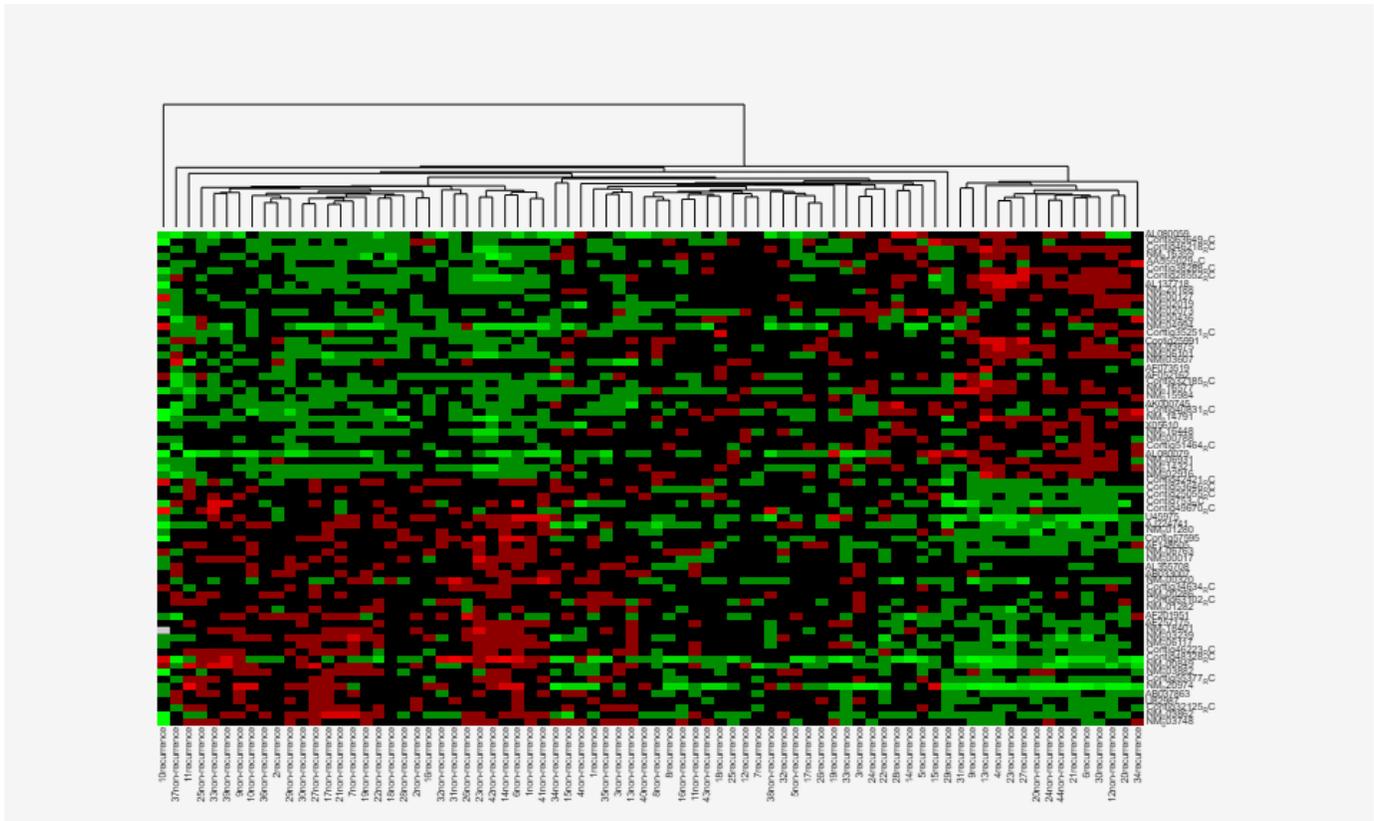
Clustering

You will use the `clustergram` function to perform hierarchical clustering and generate a heat map and dendrogram of the data. The simplest form of `clustergram` clusters the rows or columns of a data set using Euclidean distance metric and average linkage. In this example, you will cluster the samples (columns) only.

The matrix of gene expression data, `progValues`, contains some missing data. These are marked as `NaN`. You need to provide an imputation function name or function handle to impute values for missing data. In this example, you will use the k-nearest neighbors imputation procedure implemented in the function `knnimpute`.

```
cg_s = clustergram(progValues, 'RowLabels', progAccession,...
    'ColumnLabels', progSamples,...
    'Cluster', 'Row',...
    'ImputeFun', @knnimpute)
```

Clustergram object with 78 columns of nodes.



The dendrogram at the top of the heat map shows the clustering of samples. The missing data are shown in the heat map in gray. The data has been standardized across all samples for each gene, so that the mean is 0 and the standard deviation is 1.

Inspecting and Changing Clustering Options

You can determine and change properties of a clustergram object. For example, you can find out which distance metric was used in the clustering.

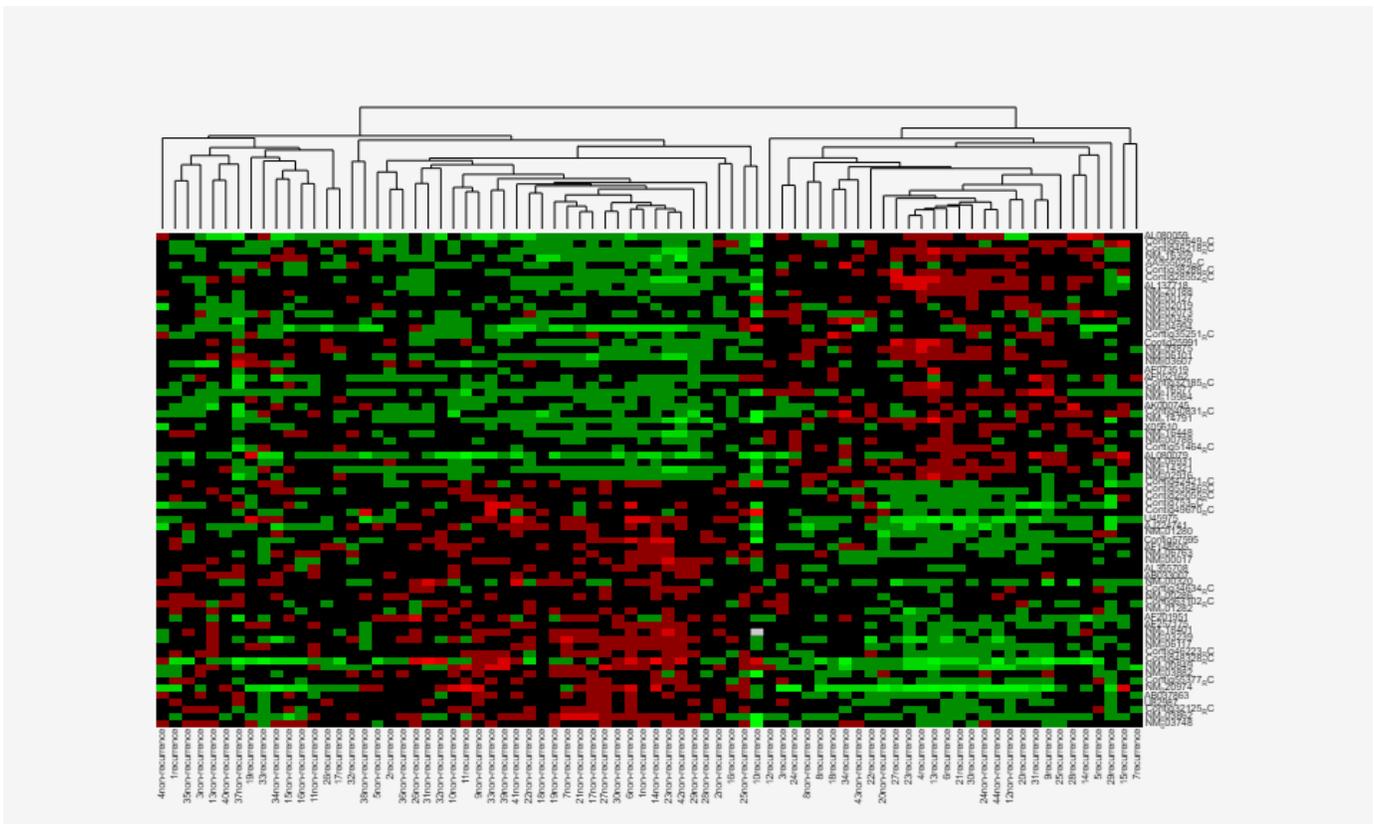
```
cg_s.ColumnPDist
```

```
ans =
```

```
1x1 cell array
    {'Euclidean'}
```

Then you can change the distance metric for the columns to correlation.

```
cg_s.ColumnPDist = 'correlation';
```



By changing the distance metric from Euclidean to correlation, the tumor samples are clearly clustered into a good prognosis group and a poor prognosis group.

To see all the properties of the clustergram, simply use the `get` method.

```
get(cg_s)
```

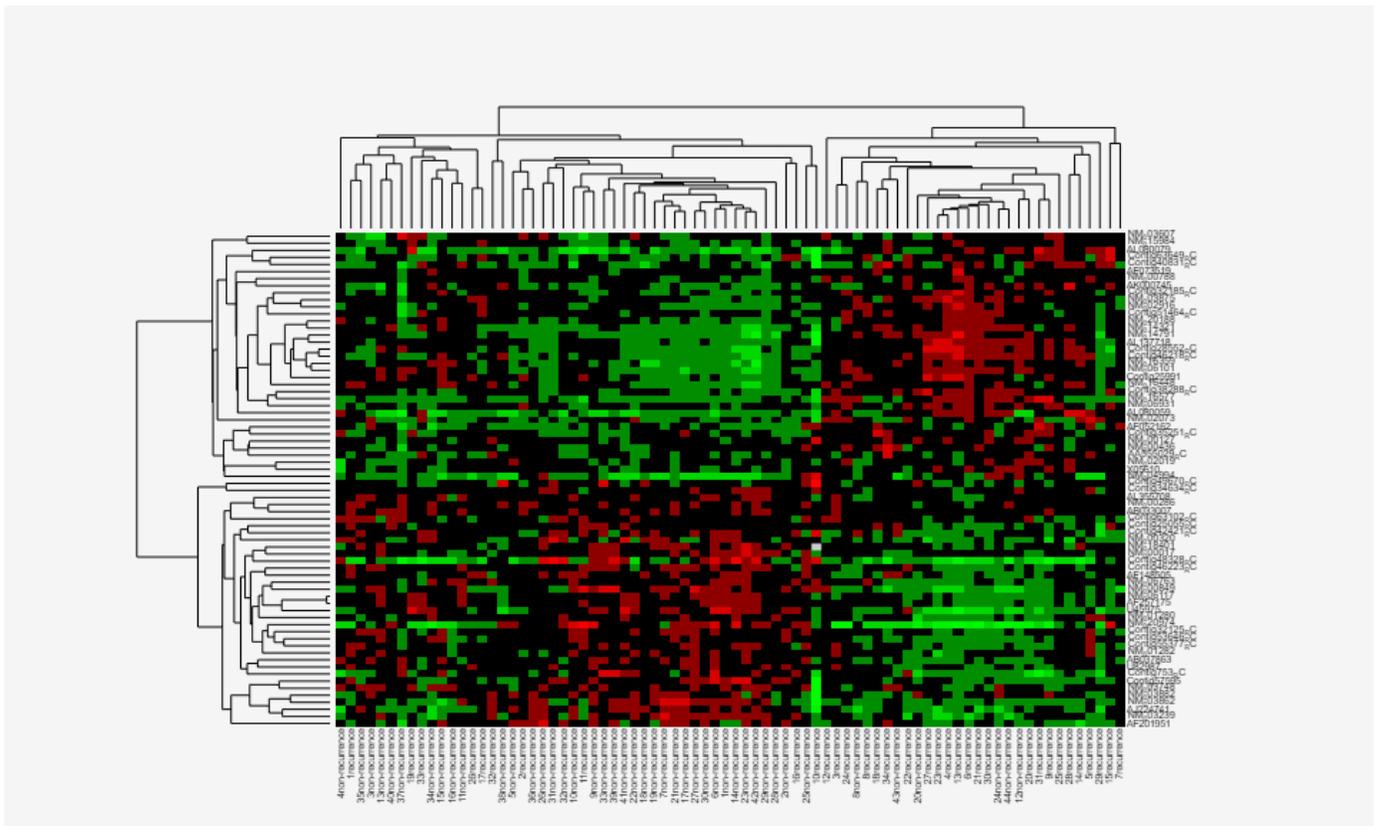
```
Cluster: 'ROW'  
RowPDist: {'Euclidean'}  
ColumnPDist: {'correlation'}  
Linkage: {'Average'}  
Dendrogram: {}  
OptimalLeafOrder: 1  
LogTrans: 0  
DisplayRatio: [0.2000 0.2000]  
RowGroupMarker: []  
ColumnGroupMarker: []  
ShowDendrogram: 'on'  
Standardize: 'NONE'  
Symmetric: 1  
DisplayRange: 3  
Colormap: [11x3 double]  
ImputeFun: {[@knnimpute]}  
ColumnLabels: {1x78 cell}  
RowLabels: {70x1 cell}  
ColumnLabelsRotate: 90  
RowLabelsRotate: 0  
Annotate: 'off'  
AnnotPrecision: 2  
AnnotColor: 'w'  
ColumnLabelsColor: []  
RowLabelsColor: []  
LabelsWithMarkers: 0
```

Clustering the Rows and the Columns of a Data Set

Next, you will cluster both the rows and the columns of the data to produce a heat map with two dendrograms. In this example, the left dendrogram shows the clustering of the genes (rows), and the top dendrogram shows the clustering of the samples (columns).

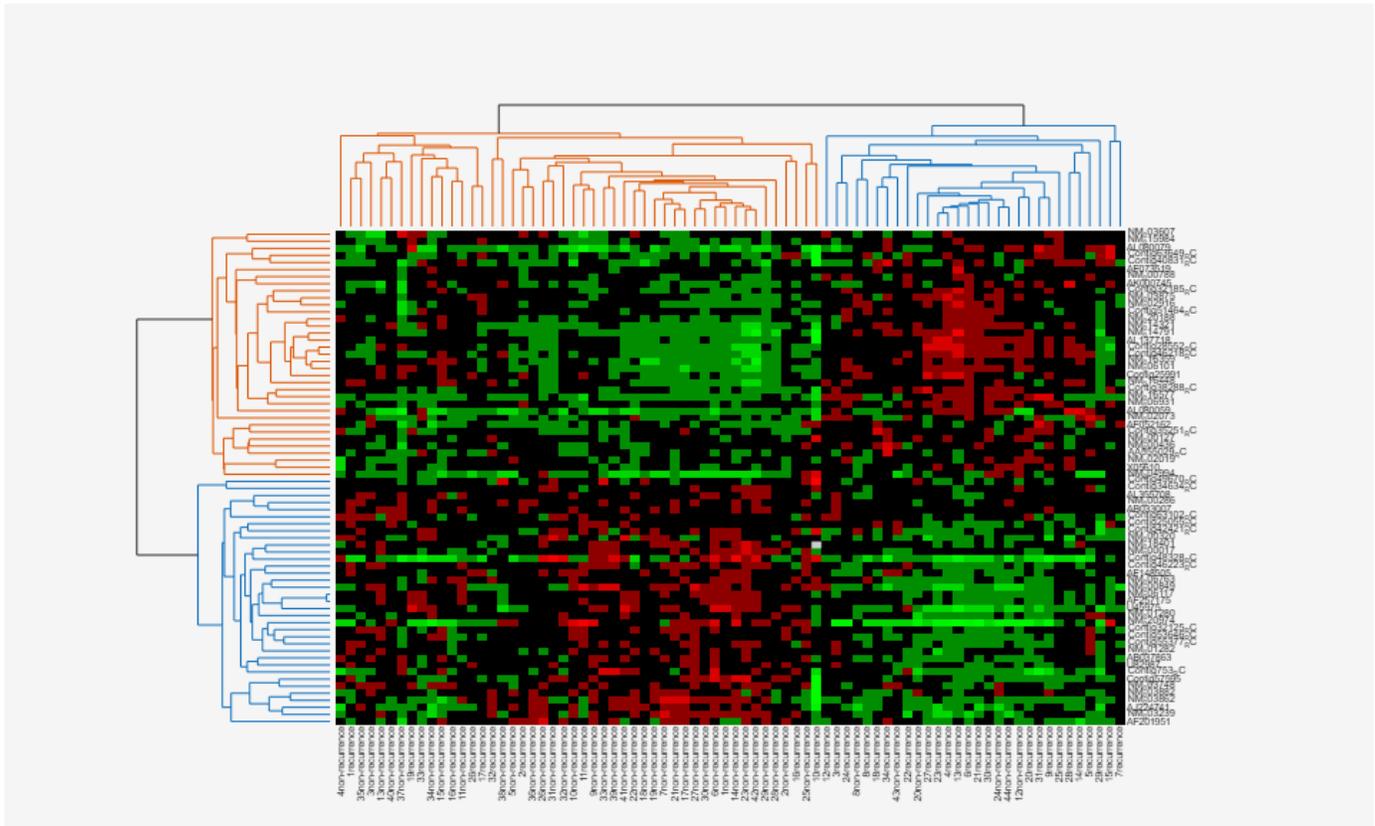
```
cg = clustergram(progValues, 'RowLabels', progAccession,...  
                        'ColumnLabels', progSamples,...  
                        'RowPdist', 'correlation',...  
                        'ColumnPdist', 'correlation',...  
                        'ImputeFun', @knnimpute)
```

Clustergram object with 70 rows of nodes and 78 columns of nodes.



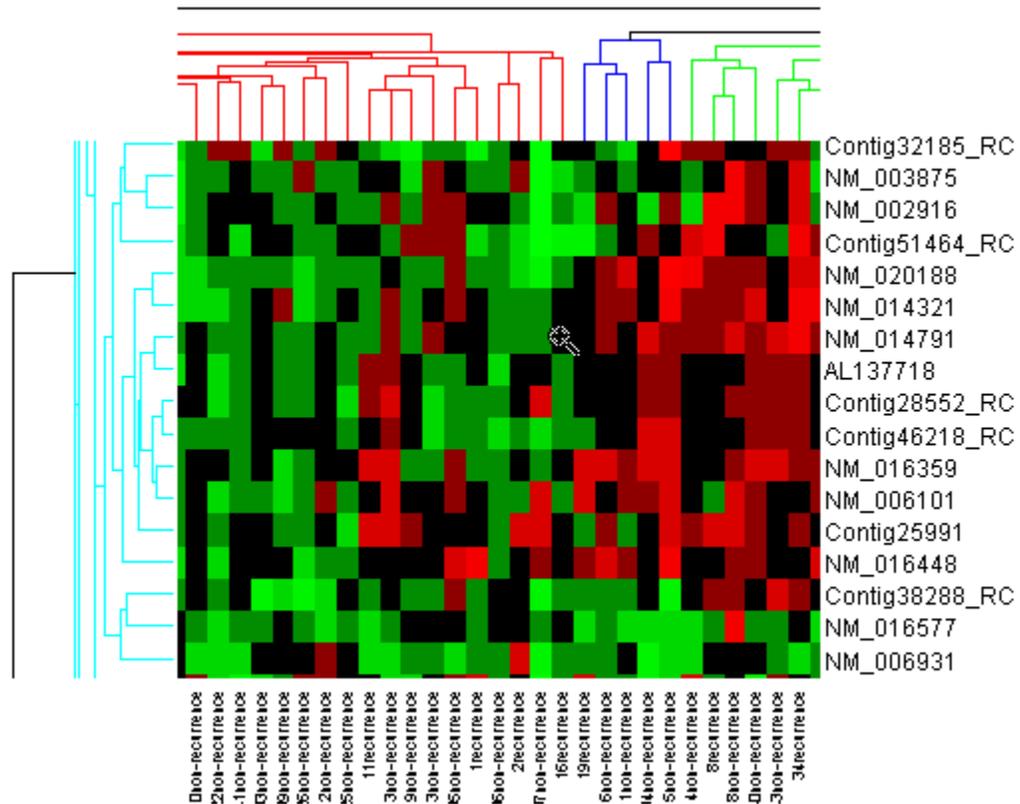
You can also change the dendrogram option to differentiate clusters of genes and clusters of samples with distances 1 unit apart.

```
cg.Dendrogram = 1;
```

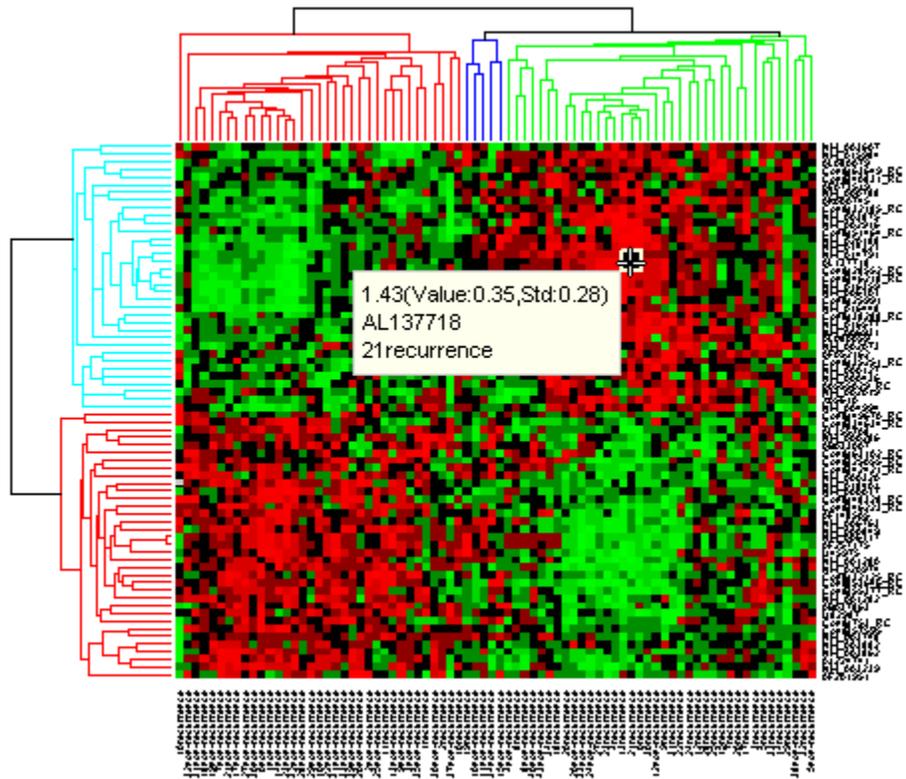


Interacting with the Heat Map

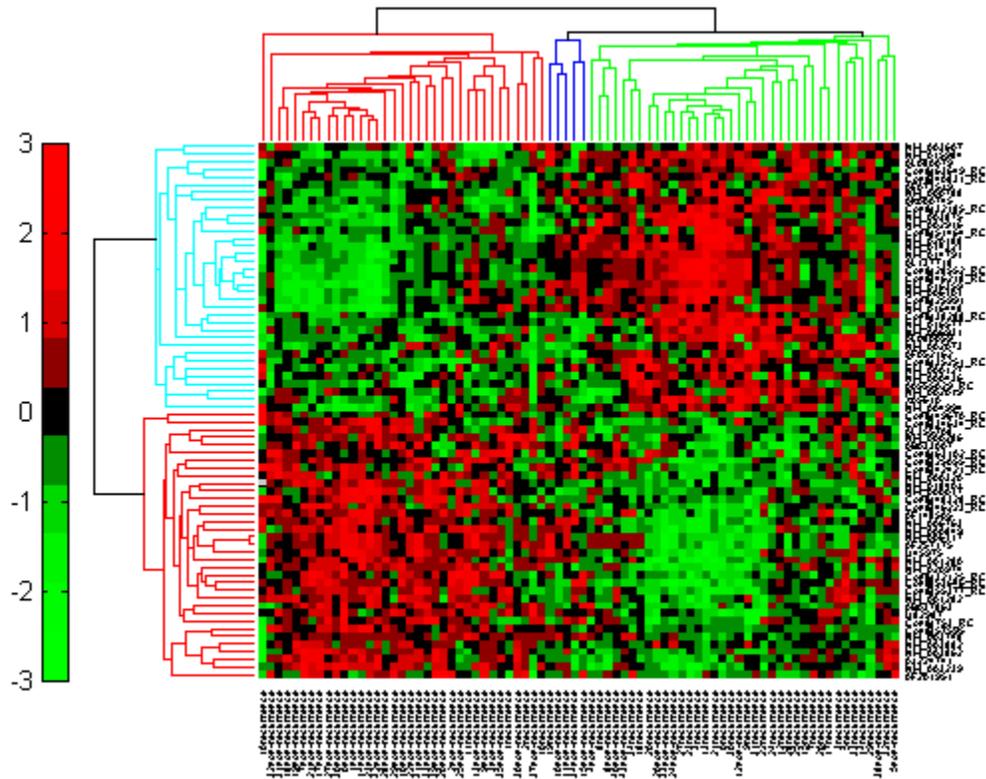
You can zoom in, zoom out and pan the heat map by selecting the corresponding toolbar buttons or menu items.



Click the **Data Cursor** button or select **Tools > Data Cursor** to activate Data Cursor Mode. In this mode, click the heatmap to display a data tip showing the expression value, the gene label and the sample label of current data point. You can click-drag the data tip to other data points in the heatmap. To delete the data tip, right-click, then select **Delete Current Datatip** from the context menu.

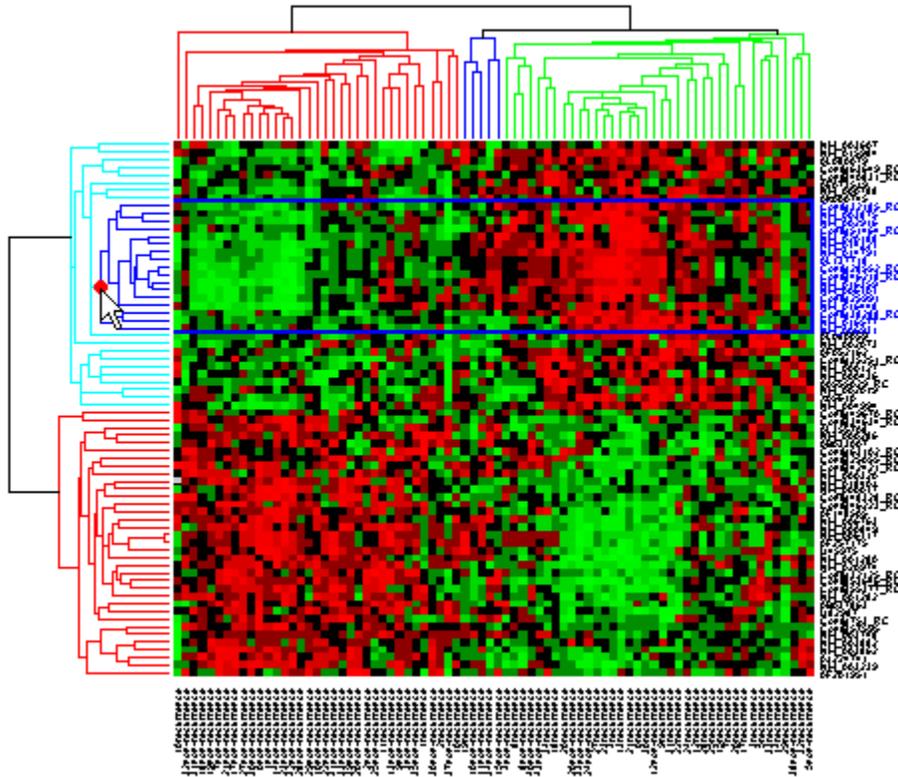


Click the **Insert Colorbar** button to show the color scale of the heat map.

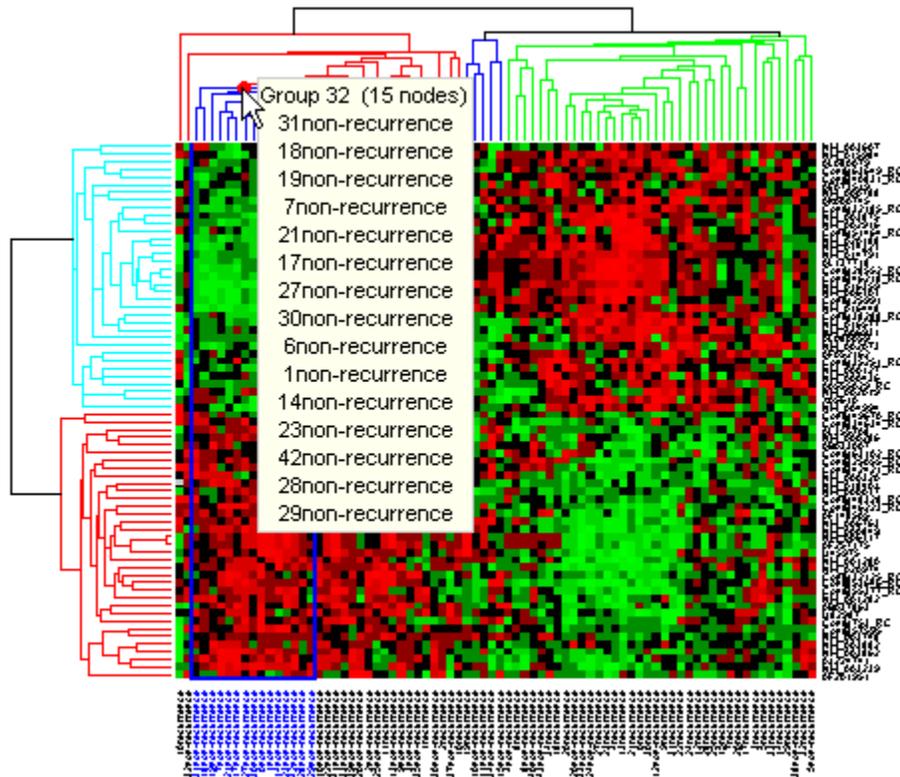


Interacting with the Dendrogram

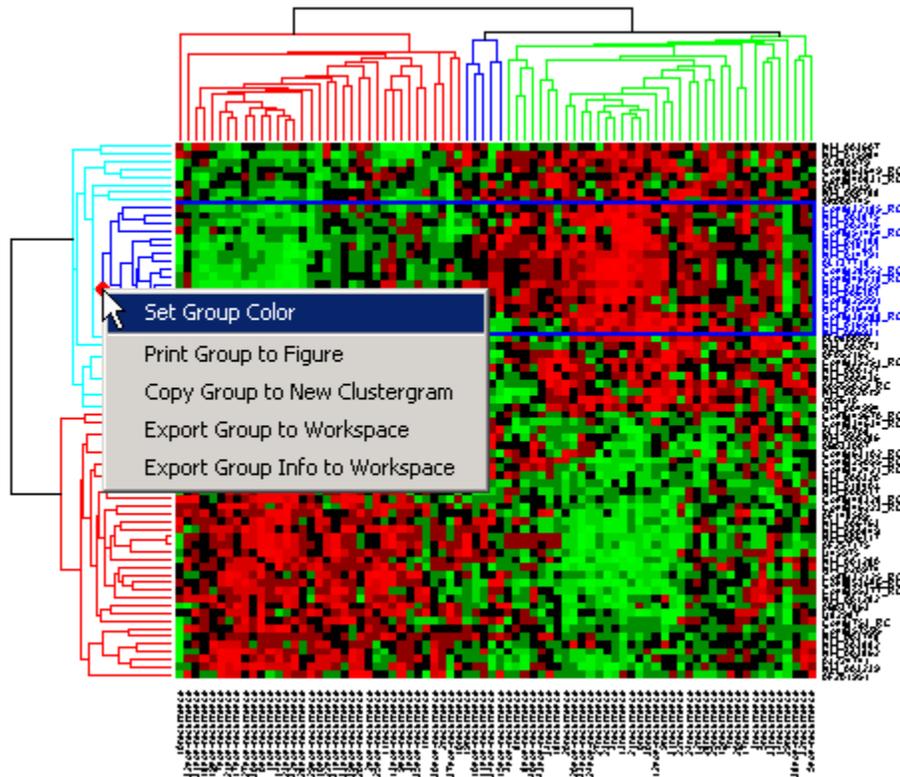
To interact with dendrogram, be sure that the **Data Cursor Mode** is deactivated (click the **Data Cursor** button again). Move the mouse over the dendrogram. When the mouse is over a branch node a red marker appears and the branch is highlighted.



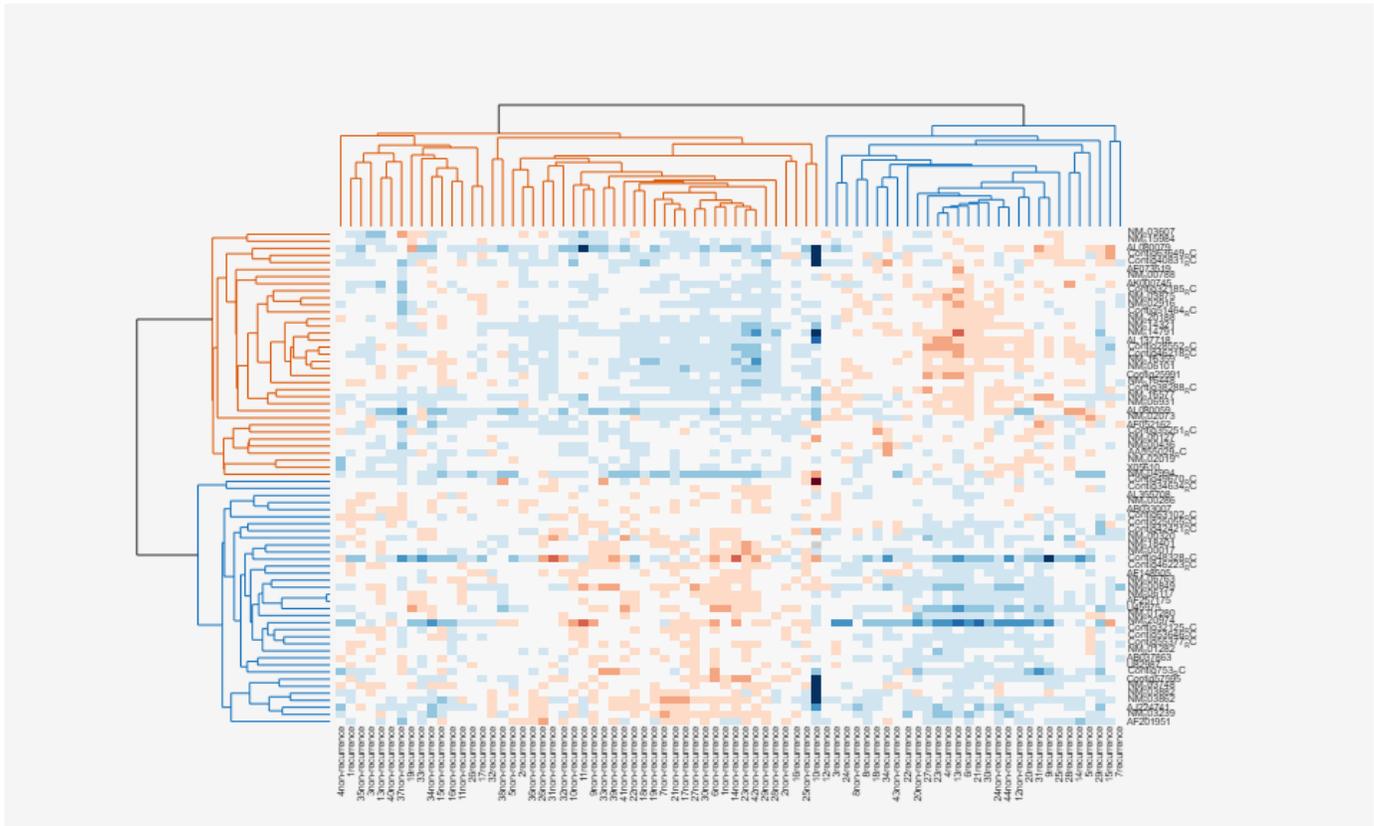
Click and hold the red marker to display a data tip with the group number and the number of nodes in the group. If the space is available, it also displays the labels for the nodes. For example, mouse over and click on a dendrogram clustering group of the samples.



Right-click the red marker to display a context menu. From the context menu you can change the dendrogram color for the select group, print the group to a separate Figure window, copy the group to a new Clustergram window, export it as a clustergram object to the MATLAB® Workspace, or export the clustering group information as a structure to the MATLAB® Workspace.



For example, select group 55 from the gene clustering dendrogram, and export it to the MATLAB® Workspace by right-clicking then selecting **Export Group to Workspace**. You can view the dendrograms and heat map for this clustergram object in a new Clustergram window by using the view method.



Adding Color Markers

The `clustergram` function also lets you add color markers and text labels to annotate specific regions of rows or columns. For example, to denote specific dendrogram groups of genes and groups of samples, create structure arrays to specify the annotations for each dimension.

Create a structure array to annotate groups 34 and 50 in the gene dendrogram.

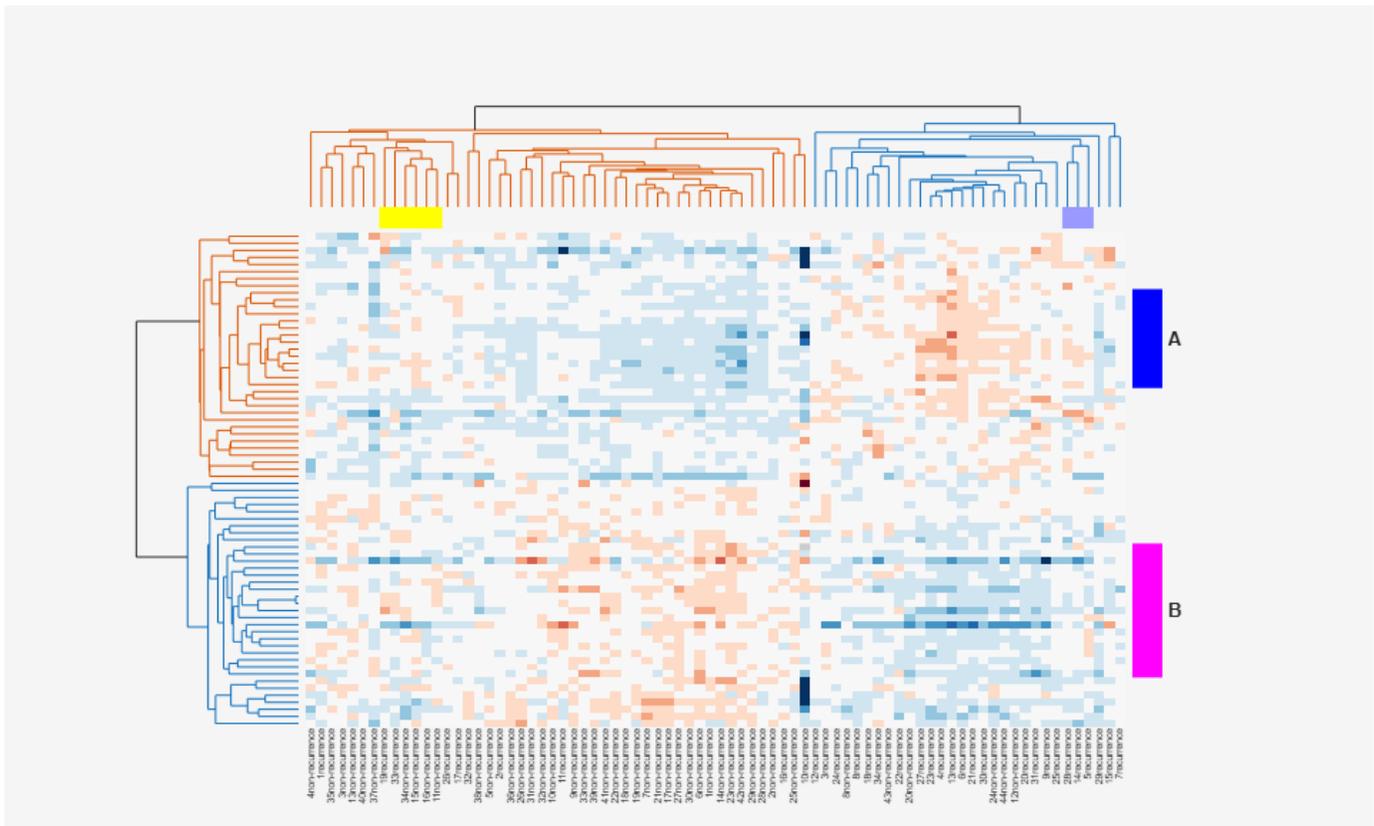
```
gene_markers = struct('GroupNumber', {34, 50},...
                    'Annotation', {'A', 'B'},...
                    'Color', {'b', 'm'});
```

Create a structure array to annotate groups 63 and 65 of the sample dendrogram.

```
sample_markers = struct('GroupNumber', {63, 65},...
                      'Annotation', {'Recurrences', 'Non-recurrences'},...
                      'Color', {[1 1 0], [0.6 0.6 1]});
```

Add the markers to the clustergram.

```
cg.RowGroupMarker = gene_markers;
cg.ColumnGroupMarker = sample_markers;
```

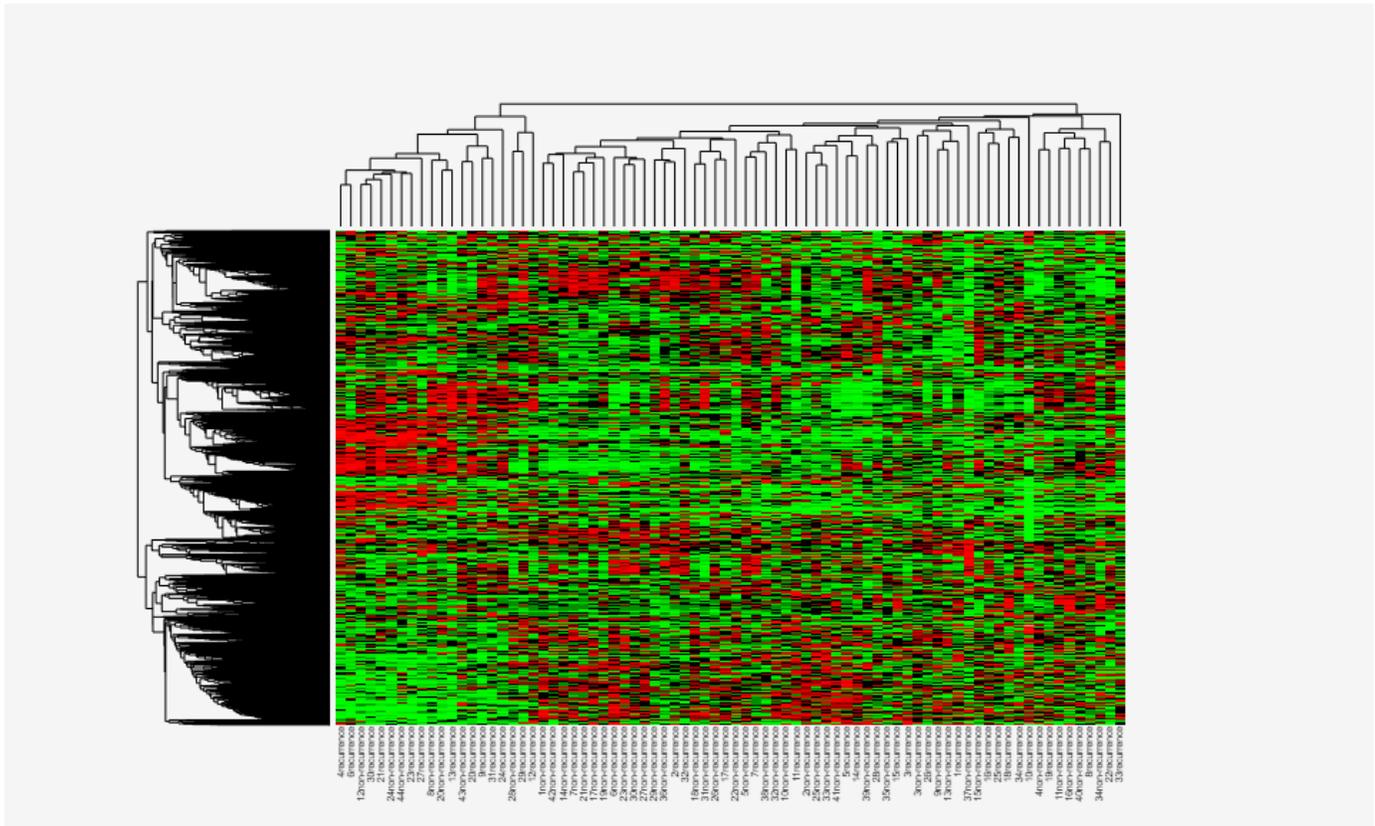


Clustering 5000 Significant Genes

In this example, you will perform hierarchical clustering for almost 5,000 genes of the filtered data [2].

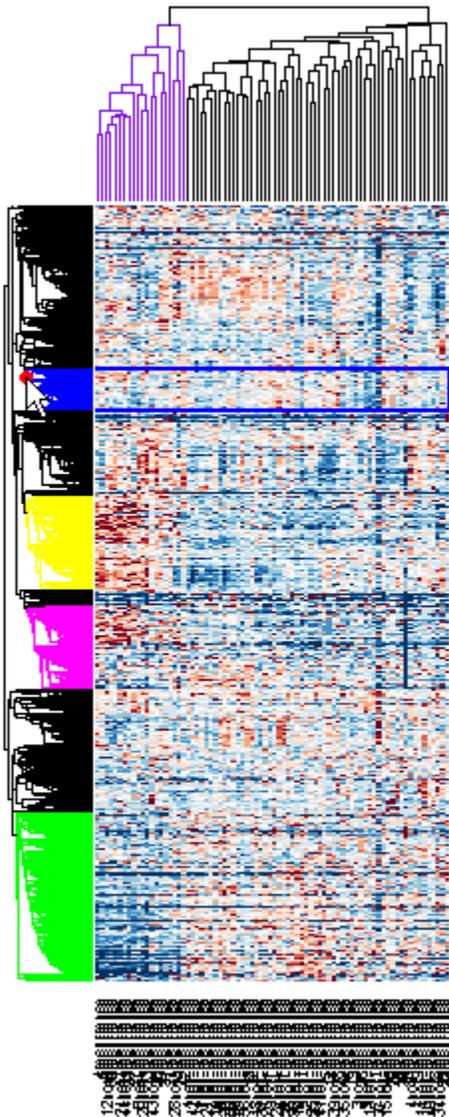
```
cg_all = clustergram(bcTrainData.Log10Ratio,...
                    'RowLabels', bcTrainData.Accession,...
                    'ColumnLabels', bcTrainData.Samples,...
                    'RowPdist', 'correlation',...
                    'ColumnPdist', 'correlation',...
                    'Displayrange', 0.6,...
                    'Standardize', 3,...
                    'ImputeFun', @knnimpute)
```

Clustergram object with 4918 rows of nodes and 78 columns of nodes.



Tip: When working with large data sets, MATLAB® can run out of memory during the clustering computation. You can convert double precision data to single precision using the `single` function. Note that the gene expression data in `bcTrainData` are already single precision.

You can resize a clustergram window like any other MATLAB® Figure window by click-dragging the edge of the window.



If you want even more control over the clustering, you can use the clustering functions in the Statistics and Machine Learning Toolbox™ directly. See the “Gene Expression Profile Analysis” on page 4-95 example for some examples of how to do this.

References

- [1] Eisen, M. B., et al., "Cluster analysis and display of genome-wide expression patterns", PNAS, 95(25):14863-8, 1998.
- [2] van't Veer, L., et al., "Gene expression profiling predicts clinical outcome of breast cancer", Nature, 415(6871):530-6, 2002.

Working with Objects for Microarray Experiment Data

This example shows how to create and manipulate MATLAB® containers designed for storing data from a microarray experiment.

Containers for Gene Expression Experiment Data

Microarray experimental data are very complex, usually consisting of data and information from a number of different sources. Storing and managing the large and complex data sets in a coherent manner is a challenge. Bioinformatics Toolbox™ provides a set of objects to represent the different pieces of data from a microarray experiment.

The `ExpressionSet` class is a single, convenient data structure for storing and managing different types of data from a microarray gene expression experiment.

An `ExpressionSet` object consists of these four components that are common to all microarray gene expression experiments:

Experiment data: Expression values from microarray experiments. These data are stored as an instance of the `ExptData` class.

Sample information: The metadata describing the samples in the experiment. The sample metadata are stored as an instance of the `MetaData` class.

Array feature annotations: The annotations about the features or probes on the array used in the experiment. The annotations can be stored as an instance of the `MetaData` class.

Experiment descriptions: Information to describe the experiment methods and conditions. The information can be stored as an instance of the `MIAME` class.

The `ExpressionSet` class coordinates and validates these data components. The class provides methods for retrieving and setting the data stored in an `ExpressionSet` object. An `ExpressionSet` object also behaves like many other MATLAB data structures that can be subsetted and copied.

Experiment Data

In a microarray gene expression experiment, the measured expression values for each feature per sample can be represented as a two-dimensional matrix. The matrix has F rows and S columns, where F is the number of features on the array, and S is the number of samples on which the expression values were measured. A `DataMatrix` object is a two-dimensional matrix that you can index by row and column numbers, logical vectors, or row and column names.

Create a `DataMatrix` with row and column names.

```
dm = bioma.data.DataMatrix(rand(5,4), 'RowNames','Feature', 'ColNames', 'Sample')
```

```
dm =
```

	Sample1	Sample2	Sample3	Sample4
Feature1	0.81472	0.09754	0.15761	0.14189
Feature2	0.90579	0.2785	0.97059	0.42176
Feature3	0.12699	0.54688	0.95717	0.91574
Feature4	0.91338	0.95751	0.48538	0.79221

```
Feature5    0.63236    0.96489    0.80028    0.95949
```

The function `size` returns the number of rows and columns in a `DataMatrix` object.

```
size(dm)
```

```
ans =
```

```
    5    4
```

You can index into a `DataMatrix` object like other MATLAB numeric arrays by using row and column numbers. For example, you can access the elements at rows 1 and 2, column 3.

```
dm(1:2, 3)
```

```
ans =
```

```
      Sample3
Feature1  0.15761
Feature2  0.97059
```

You can also index into a `DataMatrix` object by using its row and column names. Reassign the elements in row 2 and 3, column 1 and 4 to different values.

```
dm({'Feature2', 'Feature3'}, {'Sample1', 'Sample4'}) = [2, 3; 4, 5]
```

```
dm =
```

```
      Sample1  Sample2  Sample3  Sample4
Feature1  0.81472  0.09754  0.15761  0.14189
Feature2     2     0.2785  0.97059     3
Feature3     4  0.54688  0.95717     5
Feature4  0.91338  0.95751  0.48538  0.79221
Feature5  0.63236  0.96489  0.80028  0.95949
```

The gene expression data used in this example is a small set of data from a microarray experiment profiling adult mouse gene expression patterns in common strains on the Affymetrix® MG-U74Av2 array [1].

Read the expression values from the tab-formatted file `mouseExprsData.txt` into MATLAB Workspace as a `DataMatrix` object.

```
exprsData = bioma.data.DataMatrix('file', 'mouseExprsData.txt');
class(exprsData)
```

```
ans =
```

```
'bioma.data.DataMatrix'
```

Get the properties of the `DataMatrix` object, `exprsData`.

```
get(exprsData)
```

```
      Name: 'mouseExprsData'  
    RowNames: {500×1 cell}  
    ColNames: {1×26 cell}  
      NRows: 500  
      NCols: 26  
      NDims: 2  
    ElementClass: 'double'
```

Check the sample names.

```
colnames(exprsData)
```

```
ans =
```

```
1×26 cell array
```

```
Columns 1 through 8
```

```
 {'A'} {'B'} {'C'} {'D'} {'E'} {'F'} {'G'} {'H'}
```

```
Columns 9 through 16
```

```
 {'I'} {'J'} {'K'} {'L'} {'M'} {'N'} {'O'} {'P'}
```

```
Columns 17 through 24
```

```
 {'Q'} {'R'} {'S'} {'T'} {'U'} {'V'} {'W'} {'X'}
```

```
Columns 25 through 26
```

```
 {'Y'} {'Z'}
```

View the first 10 rows and 5 columns.

```
exprsData(1:10, 1:5)
```

```
ans =
```

	A	B	C	D	E
100001_at	2.26	20.14	31.66	14.58	16.04
100002_at	158.86	236.25	206.27	388.71	388.09
100003_at	68.11	105.45	82.92	82.9	60.38
100004_at	74.32	96.68	84.87	72.26	98.38
100005_at	75.05	53.17	57.94	60.06	63.91
100006_at	80.36	42.89	77.21	77.24	40.31
100007_at	216.64	191.32	219.48	237.28	298.18
100009_r_at	3806.7	1425	2468.5	2172.7	2237.2
100010_at	NaN	NaN	NaN	7.18	22.37
100011_at	81.72	72.27	127.61	91.01	98.13

Perform a log₂ transformation of the expression values.

```
exprsData_log2 = log2(exprsData);
exprsData_log2(1:10, 1:5)
```

```
ans =
```

	A	B	C	D	E
100001_at	1.1763	4.332	4.9846	3.8659	4.0036
100002_at	7.3116	7.8842	7.6884	8.6026	8.6002
100003_at	6.0898	6.7204	6.3736	6.3733	5.916
100004_at	6.2157	6.5951	6.4072	6.1751	6.6203
100005_at	6.2298	5.7325	5.8565	5.9083	5.998
100006_at	6.3284	5.4226	6.2707	6.2713	5.3331
100007_at	7.7592	7.5798	7.7779	7.8904	8.22
100009_r_at	11.894	10.477	11.269	11.085	11.127
100010_at	NaN	NaN	NaN	2.844	4.4835
100011_at	6.3526	6.1753	6.9956	6.508	6.6166

Change the Name property to be more descriptive|.

```
exprsData_log2 = set(exprsData_log2, 'Name', 'Log2 Based mouseExprsData');
get(exprsData_log2)
```

```
    Name: 'Log2 Based mouseExprsData'
  RowNames: {500x1 cell}
  ColNames: {1x26 cell}
    NRows: 500
     NCols: 26
    NDims: 2
ElementClass: 'double'
```

In a microarray experiment, the data set often contains one or more matrices that have the same number of rows and columns and identical row names and column names. `ExptData` class is designed to contain and coordinate one or more data matrices having identical row and column names with the same dimension size. The data values are stored as `DataMatrix` objects. Each `DataMatrix` object is an element of an `ExptData` object. The `ExptData` class is responsible for data validation and coordination between these `DataMatrix` objects.

Store the gene expression data of natural scale and log2 base expression values separately in an `ExptData` object.

```
mouseExptData = bioma.data.ExptData(exprsData, exprsData_log2,...
    'ElementNames', {'naturalExprs', 'log2Exprs'})
```

```
mouseExptData =
```

```
Experiment Data:
  500 features, 26 samples
  2 elements
  Element names: naturalExprs, log2Exprs
```

Access a `DataMatrix` element in `mouseExptData` using the element name.

```
exprsData2 = mouseExptData('log2Exprs');
get(exprsData2)
```

```
Name: 'Log2 Based mouseExprsData'  
RowNames: {500x1 cell}  
ColNames: {1x26 cell}  
NRows: 500  
NCols: 26  
NDims: 2  
ElementClass: 'double'
```

Sample Metadata

The metadata about the samples in a microarray experiment can be represented as a table with S rows and V columns, where S is the number of samples, and V is the number of variables. The contents of the table are the values of each variable for each sample. For example, the file `mouseSampleData.txt` contains such a table. The description of each sample variable is marked by a `#` symbol.

The `MetaData` class is designed for storing and manipulating variable values and their metadata in a coordinated fashion. You can read the `mouseSampleData.txt` file into MATLAB as a `MetaData` object.

```
sData = bioma.data.MetaData('file', 'mouseSampleData.txt', 'vardescchar', '#')
```

```
sData =
```

```
Sample Names:
```

```
A, B, ...,Z (26 total)
```

```
Variable Names and Meta Information:
```

```
VariableDescription  
Gender {' Gender of the mouse in study' }  
Age {' The number of weeks since mouse birth'}  
Type {' Genetic characters' }  
Strain {' The mouse strain' }  
Source {' The tissue source for RNA collection' }
```

The properties of `MetaData` class provide information about the samples and variables.

```
numSamples = sData.NSamples  
numVariables = sData.NVariables
```

```
numSamples =
```

```
26
```

```
numVariables =
```

```
5
```

The variable values and the variable descriptions for the samples are stored as two `dataset` arrays in a `MetaData` class. The `MetaData` class provides access methods to the variable values and the meta information describing the variables.

Access the sample metadata using the `variableValues` method.

```
sData.variableValues
```

```
ans =
```

	Gender	Age	Type	Strain
A	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
B	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
C	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
D	{'Male'}	8	{'Wild type'}	{'A/J '}
E	{'Male'}	8	{'Wild type'}	{'A/J '}
F	{'Male'}	8	{'Wild type'}	{'C57BL/6J '}
G	{'Male'}	8	{'Wild type'}	{'C57BL/6J '}
H	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
I	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
J	{'Male'}	8	{'Wild type'}	{'A/J'}
K	{'Male'}	8	{'Wild type'}	{'A/J'}
L	{'Male'}	8	{'Wild type'}	{'A/J'}
M	{'Male'}	8	{'Wild type'}	{'C57BL/6J'}
N	{'Male'}	8	{'Wild type'}	{'C57BL/6J'}
O	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
P	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
Q	{'Male'}	8	{'Wild type'}	{'A/J'}
R	{'Male'}	8	{'Wild type'}	{'A/J'}
S	{'Male'}	8	{'Wild type'}	{'C57BL/6J'}
T	{'Male'}	8	{'Wild type'}	{'C57BL/6J4'}
U	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
V	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
W	{'Male'}	8	{'Wild type'}	{'A/J'}
X	{'Male'}	8	{'Wild type'}	{'A/J'}
Y	{'Male'}	8	{'Wild type'}	{'C57BL/6J'}
Z	{'Male'}	8	{'Wild type'}	{'C57BL/6J'}

	Source
A	{'amygdala'}
B	{'amygdala'}
C	{'amygdala'}
D	{'amygdala'}
E	{'amygdala'}
F	{'amygdala'}
G	{'amygdala'}
H	{'cingulate cortex'}
I	{'cingulate cortex'}
J	{'cingulate cortex'}
K	{'cingulate cortex'}
L	{'cingulate cortex'}
M	{'cingulate cortex'}
N	{'cingulate cortex'}
O	{'hippocampus'}
P	{'hippocampus'}
Q	{'hippocampus'}
R	{'hippocampus'}
S	{'hippocampus'}
T	{'hippocampus'}
U	{'hypothalamus'}
V	{'hypothalamus'}
W	{'hypothalamus'}

```
X {'hypothalamus' }
Y {'hypothalamus' }
Z {'hypothalamus' }
```

View a summary of the sample metadata.

```
summary(sData.variableValues)
```

```
Gender: [26x1 cell array of character vectors]
```

```
Age: [26x1 double]
```

```
    min    1st quartile    median    3rd quartile    max
     8         8           8         8           8
```

```
Type: [26x1 cell array of character vectors]
```

```
Strain: [26x1 cell array of character vectors]
```

```
Source: [26x1 cell array of character vectors]
```

The `sampleNames` and `variableNames` methods are convenient ways to access the names of samples and variables. Retrieve the variable names of the `sData` object.

```
variableNames(sData)
```

```
ans =
```

```
1x5 cell array
```

```
 {'Gender'} {'Age'} {'Type'} {'Strain'} {'Source'}
```

You can retrieve the meta information about the variables describing the samples using the `variableDesc` method. In this example, it contains only the descriptions about the variables.

```
variableDesc(sData)
```

```
ans =
```

```
      VariableDescription
Gender {' Gender of the mouse in study' }
Age    {' The number of weeks since mouse birth' }
Type   {' Genetic characters' }
Strain {' The mouse strain' }
Source {' The tissue source for RNA collection' }
```

You can subset the sample data `sData` object using numerical indexing.

```
sData(3:6, :)
```

```
ans =
```

```

Sample Names:
  C, D, ...,F (4 total)
Variable Names and Meta Information:
  VariableDescription
  Gender      {' Gender of the mouse in study'      }
  Age         {' The number of weeks since mouse birth'}
  Type        {' Genetic characters'                }
  Strain      {' The mouse strain'                  }
  Source      {' The tissue source for RNA collection' }

```

You can display the mouse strain of specific samples by using numerical indexing.

```
sData.Strain([2 14])
```

```

ans =

  2×1 cell array

    {'129S6/SvEvTac'}
    {'C57BL/6J'      }

```

Note that the row names in `sData` and the column names in `exprsData` are the same. It is an important relationship between the expression data and the sample data in the same experiment.

```
all(ismember(sampleNames(sData), colnames(exprsData)))
```

```

ans =

  logical

   1

```

Feature Annotation Metadata

The metadata about the features or probe set on an array can be very large and diverse. The chip manufacturers usually provide a specific annotation file for the features of each type of array. The metadata can be stored as a `MetaData` object for a specific experiment. In this example, the annotation file for the MG-U74Av2 array can be downloaded from the Affymetrix web site. You will need to convert the file from CSV to XLSX format using a spreadsheet software application.

Read the entire file into MATLAB as a `dataset` array. Alternatively, you can use the `Range` option in the `dataset` constructor. Any blank spaces in the variable names are removed to make them valid MATLAB variable names. A warning is displayed each time this happens.

```
mgU74Av2 = table2dataset(readtable('MG_U74Av2_annot.xlsx'));
```

```

Warning: Column headers from the file were modified to make them valid MATLAB
identifiers before creating variable names for the table. The original column
headers are saved in the VariableDescriptions property.
Set 'VariableNamingRule' to 'preserve' to use the original column headers as
table variable names.

```

Inspect the properties of this `dataset` array.

```
get(mgU74Av2)
  Description: ''
  VarDescription: {1×43 cell}
  Units: {}
  DimNames: {'Row' 'Variables'}
  UserData: []
  ObsNames: {}
  VarNames: {1×43 cell}
```

Determine the number of probe set IDs in the annotation file.

```
numel(mgU74Av2.ProbeSetID)
```

```
ans =
    12488
```

Retrieve the names of variables describing the features on the array and view the first 20 variable names.

```
fDataVariables = get(mgU74Av2, 'VarNames');
fDataVariables(1:20)'
```

```
ans =
    20×1 cell array

    {'ProbeSetID'          }
    {'GeneChipArray'      }
    {'SpeciesScientificName' }
    {'AnnotationDate'     }
    {'SequenceType'       }
    {'SequenceSource'     }
    {'TranscriptID_ArrayDesign_' }
    {'TargetDescription'   }
    {'RepresentativePublicID' }
    {'ArchivalUniGeneCluster' }
    {'UniGeneID'          }
    {'GenomeVersion'      }
    {'Alignments'         }
    {'GeneTitle'          }
    {'GeneSymbol'         }
    {'ChromosomalLocation' }
    {'UnigeneClusterType' }
    {'Ensembl'            }
    {'EntrezGene'         }
    {'SwissProt'          }
```

Set the `ObsNames` property to the probe set IDs, so that you can access individual gene annotations by indexing with probe set IDs.

```
mgU74Av2 = set(mgU74Av2, 'ObsNames', mgU74Av2.ProbeSetID);
mgU74Av2('100709_at', {'GeneSymbol', 'ChromosomalLocation'})
```

```
ans =
      100709_at      GeneSymbol      ChromosomalLocation
              {'Tpbpa'}      {'chr13 B2|13 36.0 cM'}
```

In some cases, it is useful to extract specific annotations that are relevant to the analysis. Extract annotations for `GeneTitle`, `GeneSymbol`, `ChromosomalLocation`, and `Pathway` relative to the features in `exprsData`.

```
mgU74Av2 = mgU74Av2(:, {'GeneTitle', ...
                       'GeneSymbol', ...
                       'ChromosomalLocation', ...
                       'Pathway'});
```

```
mgU74Av2 = mgU74Av2(rownames(exprsData), :);
get(mgU74Av2)
```

```
  Description: ''
VarDescription: {1x4 cell}
      Units: {}
  DimNames: {'Row' 'Variables'}
  UserData: []
  ObsNames: {500x1 cell}
  VarNames: {1x4 cell}
```

You can store the feature annotation dataset array as an instance of the `MetaData` class.

```
fData = bioma.data.MetaData(mgU74Av2)
```

```
fData =
```

```
Sample Names:
  100001_at, 100002_at, ..., 100717_at (500 total)
Variable Names and Meta Information:
      VariableDescription
GeneTitle      {'NA'}
GeneSymbol     {'NA'}
ChromosomalLocation {'NA'}
Pathway        {'NA'}
```

Notice that there are no descriptions for the feature variables in the `fData` `MetaData` object. You can add descriptions about the variables in `fData` using the `variableDesc` method.

```
fData = variableDesc(fData, {'Gene title of a probe set', ...
                             'Probe set gene symbol', ...
                             'Probe set chromosomal locations', ...
                             'The pathway the genes involved in'})
```

```
fData =
```

```
Sample Names:
  100001_at, 100002_at, ..., 100717_at (500 total)
Variable Names and Meta Information:
```

```
VariableDescription
GeneTitle      {'Gene title of a probe set'      }
GeneSymbol     {'Probe set gene symbol'          }
ChromosomalLocation {'Probe set chromosomal locations' }
Pathway        {'The pathway the genes involved in'}
```

Experiment Information

The MIAME class is a flexible data container designed for a collection of basic descriptions about a microarray experiment, such as investigators, laboratories, and array designs. The MIAME class loosely follows the Minimum Information About a Microarray Experiment (MIAME) specification [2].

Create a MIAME object by providing some basic information.

```
expDesc = bioma.data.MIAME('investigator', 'Jane OneName',...
                           'lab',          'Bioinformatics Laboratory',...
                           'title',       'Example Gene Expression Experiment',...
                           'abstract',    'An example of using microarray objects.',...
                           'other',      {'Notes: Created from a text files.'})
```

```
expDesc =
```

```
Experiment Description:
  Author name: Jane OneName
  Laboratory: Bioinformatics Laboratory
  Contact information:
  URL:
  PubMedIDs:
  Abstract: A 5 word abstract is available. Use the Abstract property.
  No experiment design summary available.
  Other notes:
  {'Notes: Created from a text files.'}
```

Another way to create a MIAME object is from GEO series data. The MIAME class will populate the corresponding properties from the GEO series structure. The information associated with the gene profile experiment in this example is available from the GEO database under the accession number GSE3327 [1]. Retrieve the GEO Series data using the `getgeodata` function.

```
getgeodata('GSE3327', 'ToFile', 'GSE3327.txt');
```

Read the data into a structure.

```
geoSeries = geoseriesread('GSE3327.txt')
```

```
geoSeries =
```

```
struct with fields:
  Header: [1x1 struct]
  Data: [12488x87 bioma.data.DataMatrix]
```

Create a MIAME object.

```
exptGSE3327 = bioma.data.MIAME(geoSeries)
```

```

exptGSE3327 =
Experiment Description:
  Author name: Iris,,Hovatta
David,J,Lockhart
Carrolee,,Barlow
  Laboratory: The Salk Institute for Biological Studies
  Contact information: Carrolee,,Barlow
  URL:
  PubMedIDs: 16244648
  Abstract: A 14 word abstract is available. Use the Abstract property.
  Experiment Design: A 8 word summary is available. Use the ExptDesign property.
  Other notes:
    {'ftp://ftp.ncbi.nlm.nih.gov/pub/geo/DATA/supplementary/series/GSE3327/GSE3327_RAW.tar'}

```

View the abstract of the experiment and its PubMed IDs.

```

abstract = exptGSE3327.Abstract
pubmedID = exptGSE3327.PubMedID

```

```

abstract =
  'Adult mouse gene expression patterns in common strains
  Keywords: mouse strain and brain region comparison'

```

```

pubmedID =
  '16244648'

```

Creating an ExpressionSet Object

The `ExpressionSet` class is designed specifically for microarray gene expression experiment data. Assemble an `ExpressionSet` object for the example mouse gene expression experiment from the different data objects you just created.

```

exptSet = bioma.ExpressionSet(exprsData, 'SData', sData,...
                               'FData', fData,...
                               'Einfo', exptGSE3327)

```

```

exptSet =
ExpressionSet
Experiment Data: 500 features, 26 samples
  Element names: Expressions
Sample Data:
  Sample names:      A, B, ...,Z (26 total)
  Sample variable names and meta information:
    Gender: Gender of the mouse in study
    Age: The number of weeks since mouse birth
    Type: Genetic characters
    Strain: The mouse strain
    Source: The tissue source for RNA collection

```

```
Feature Data:
  Feature names:      100001_at, 100002_at, ...,100717_at (500 total)
  Feature variable names and meta information:
    GeneTitle: Gene title of a probe set
    GeneSymbol: Probe set gene symbol
    ChromosomalLocation: Probe set chromosomal locations
    Pathway: The pathway the genes involved in
  Experiment Information: use 'exptInfo(obj)'
```

You can also create an `ExpressionSet` object with only the expression values in a `DataMatrix` or a numeric matrix.

```
miniExprSet = bioma.ExpressionSet(exprsData)
```

```
miniExprSet =

ExpressionSet
Experiment Data: 500 features, 26 samples
  Element names: Expressions
Sample Data: none
Feature Data: none
Experiment Information: none
```

Saving and Loading an ExpressionSet Object

The data objects for a microarray experiment can be saved as *MAT* files. Save the `ExpressionSet` object `exptSet` to a *MAT* file named `mouseExpressionSet.mat`.

```
save mouseExpressionSet exptSet
```

Clear variables from the MATLAB Workspace.

```
clear dm exprs* mouseExptData ME sData
```

Load the *MAT* file `mouseExpressionSet` into the MATLAB Workspace.

```
load mouseExpressionSet
```

Inspect the loaded `ExpressionSet` object.

```
exptSet.elementNames
```

```
ans =
```

```
  1×1 cell array
    {'Expressions'}
```

```
exptSet.NSamples
```

```
ans =
```

```
  26
```

```
exptSet.NFeatures
```

```
ans =
    500
```

Accessing Data Components of an ExpressionSet Object

A number of methods are available to access and update data stored in an ExpressionSet object.

You can access the columns of the sample data using dot notation.

```
exptSet.Strain(1:5)
```

```
ans =
    5x1 cell array
    {'129S6/SvEvTac'}
    {'129S6/SvEvTac'}
    {'129S6/SvEvTac'}
    {'A/J '          }
    {'A/J '          }
```

Retrieve the feature names using the featureNames method. In this example, the feature names are the probe set identifiers on the array.

```
featureNames(exptSet, 1:5)
```

```
ans =
    5x1 cell array
    {'100001_at'}
    {'100002_at'}
    {'100003_at'}
    {'100004_at'}
    {'100005_at'}
```

The unique identifier of the samples can be accessed via the sampleNames method.

```
exptSet.sampleNames(1:5)
```

```
ans =
    1x5 cell array
    {'A'}  {'B'}  {'C'}  {'D'}  {'E'}
```

The sampleVarNames method lists the variable names in the sample data.

```
exptSet.sampleVarNames
```

ans =

1x5 cell array

```
{'Gender'}    {'Age'}    {'Type'}    {'Strain'}    {'Source'}
```

Extract the **dataset** array containing sample information.

```
sDataset = sampleVarValues(exptSet)
```

sDataset =

	Gender	Age	Type	Strain
A	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
B	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
C	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
D	{'Male'}	8	{'Wild type'}	{'A/J' }
E	{'Male'}	8	{'Wild type'}	{'A/J' }
F	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
G	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
H	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
I	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
J	{'Male'}	8	{'Wild type'}	{'A/J' }
K	{'Male'}	8	{'Wild type'}	{'A/J' }
L	{'Male'}	8	{'Wild type'}	{'A/J' }
M	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
N	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
O	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
P	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
Q	{'Male'}	8	{'Wild type'}	{'A/J' }
R	{'Male'}	8	{'Wild type'}	{'A/J' }
S	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
T	{'Male'}	8	{'Wild type'}	{'C57BL/6J4' }
U	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
V	{'Male'}	8	{'Wild type'}	{'129S6/SvEvTac'}
W	{'Male'}	8	{'Wild type'}	{'A/J' }
X	{'Male'}	8	{'Wild type'}	{'A/J' }
Y	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }
Z	{'Male'}	8	{'Wild type'}	{'C57BL/6J' }

	Source
A	{'amygdala' }
B	{'amygdala' }
C	{'amygdala' }
D	{'amygdala' }
E	{'amygdala' }
F	{'amygdala' }
G	{'amygdala' }
H	{'cingulate cortex'}
I	{'cingulate cortex'}
J	{'cingulate cortex'}
K	{'cingulate cortex'}
L	{'cingulate cortex'}
M	{'cingulate cortex'}

```

N   {'cingulate cortex'}
O   {'hippocampus'     }
P   {'hippocampus'     }
Q   {'hippocampus'     }
R   {'hippocampus'     }
S   {'hippocampus'     }
T   {'hippocampus'     }
U   {'hypothalamus'    }
V   {'hypothalamus'    }
W   {'hypothalamus'    }
X   {'hypothalamus'    }
Y   {'hypothalamus'    }
Z   {'hypothalamus'    }

```

Retrieve the `ExptData` object containing expression values. There may be more than one `DataMatrix` object with identical dimensions in an `ExptData` object. In an `ExpressionSet` object, there is always an element `DataMatrix` object named `Expressions` containing the expression matrix.

```
exptDS = exptData(exptSet)
```

```
exptDS =
```

```

Experiment Data:
 500 features, 26 samples
 1 elements
Element names: Expressions

```

Extract only the expression `DataMatrix` instance.

```
dMatrix = expressions(exptSet);
```

The returned expression `DataMatrix` should be identical to the `exprsData` `DataMatrix` object that you created earlier.

```
get(dMatrix)
```

```

      Name: 'mouseExprsData'
RowNames: {500x1 cell}
ColNames: {1x26 cell}
   NRows: 500
    NCols: 26
    NDims: 2
ElementClass: 'double'

```

Get PubMed IDs for the experiment stored in `exptSet`.

```
exptSet.pubMedID
```

```
ans =
```

```
'16244648'
```

Subsetting an ExpressionSet Object

You can subset an ExpressionSet object so that you can focus on the samples and features of interest. The first indexing argument subsets the features and the second argument subsets the samples.

Create a new ExpressionSet object consisting of the first five features and the samples named A, B, and C.

```
mySet = exptSet(1:5, {'A', 'B', 'C'})
```

```
mySet =
```

```
ExpressionSet
Experiment Data: 5 features, 3 samples
  Element names: Expressions
Sample Data:
  Sample names:      A, B, C
  Sample variable names and meta information:
    Gender: Gender of the mouse in study
    Age: The number of weeks since mouse birth
    Type: Genetic characters
    Strain: The mouse strain
    Source: The tissue source for RNA collection
Feature Data:
  Feature names:      100001_at, 100002_at, ...,100005_at (5 total)
  Feature variable names and meta information:
    GeneTitle: Gene title of a probe set
    GeneSymbol: Probe set gene symbol
    ChromosomalLocation: Probe set chromosomal locations
    Pathway: The pathway the genes involved in
Experiment Information: use 'exptInfo(obj)'
```

```
size(mySet)
```

```
ans =
      5      3
```

```
featureNames(mySet)
```

```
ans =
5x1 cell array
    {'100001_at'}
    {'100002_at'}
    {'100003_at'}
    {'100004_at'}
    {'100005_at'}
```

```
sampleNames(mySet)
```

```
ans =
```

```
1×3 cell array
    {'A'}    {'B'}    {'C'}
```

You can also create a subset consisting of only the samples from hippocampus tissues.

```
hippocampusSet = exptSet(:, nominal(exptSet.Source)=='hippocampus')
```

```
hippocampusSet =
```

```
ExpressionSet
Experiment Data: 500 features, 6 samples
Element names: Expressions
Sample Data:
Sample names:      0, P, ...,T (6 total)
Sample variable names and meta information:
Gender: Gender of the mouse in study
Age: The number of weeks since mouse birth
Type: Genetic characters
Strain: The mouse strain
Source: The tissue source for RNA collection
Feature Data:
Feature names:      100001_at, 100002_at, ...,100717_at (500 total)
Feature variable names and meta information:
GeneTitle: Gene title of a probe set
GeneSymbol: Probe set gene symbol
ChromosomalLocation: Probe set chromosomal locations
Pathway: The pathway the genes involved in
Experiment Information: use 'exptInfo(obj)'
```

```
hippocampusSet.Source
```

```
ans =
```

```
6×1 cell array
    {'hippocampus'}
    {'hippocampus'}
    {'hippocampus'}
    {'hippocampus'}
    {'hippocampus'}
    {'hippocampus'}
```

```
hippocampusExprs = expressions(hippocampusSet);
```

```
get(hippocampusExprs)
```

```
Name: 'mouseExprsData'
RowNames: {500×1 cell}
ColNames: {'0' 'P' 'Q' 'R' 'S' 'T'}
NRows: 500
NCols: 6
NDims: 2
ElementClass: 'double'
```

References

- [1] Hovatta, I., et al., "Glyoxalase 1 and glutathione reductase 1 regulate anxiety in mice", *Nature*, 438(7068):662-6, 2005.
- [2] Brazma, A., et al., "Minimum information about a microarray experiment (MIAME) - toward standards for microarray data", *Nat. Genet.* 29(4):365-371, 2001.

Phylogenetic Analysis

- “Using the Phylogenetic Tree App” on page 5-2
- “Building a Phylogenetic Tree for the Hominidae Species” on page 5-19
- “Analyzing the Origin of the Human Immunodeficiency Virus” on page 5-25
- “Bootstrapping Phylogenetic Trees” on page 5-32

Using the Phylogenetic Tree App

In this section...

“Overview of the Phylogenetic Tree App” on page 5-2

“Opening the Phylogenetic Tree App” on page 5-2

“File Menu” on page 5-3

“Tools Menu” on page 5-11

“Window Menu” on page 5-17

“Help Menu” on page 5-18

Overview of the Phylogenetic Tree App

The Phylogenetic Tree app allows you to view, edit, format, and explore phylogenetic tree data. With this app you can prune, reorder, rename branches, and explore distances. You can also open or save Newick or ClustalW tree formatted files. The following sections give a description of menu commands and features for creating publishable tree figures.

Opening the Phylogenetic Tree App

This section illustrates how to draw a phylogenetic tree from data in a `phytree` object or a previously saved file.

The Phylogenetic Tree app can read data from Newick and ClustalW tree formatted files.

This procedure uses the phylogenetic tree data stored in the file `pf00002.tree` as an example. The data was retrieved from the protein family (PFAM) Web database and saved to a file using the accession number PF00002 and the function `gethmmtree`.

- 1 Create a `phytree` object. For example, to create a `phytree` object from tree data in the file `pf00002.tree`, type

```
tr = phytread('pf00002.tree')
```

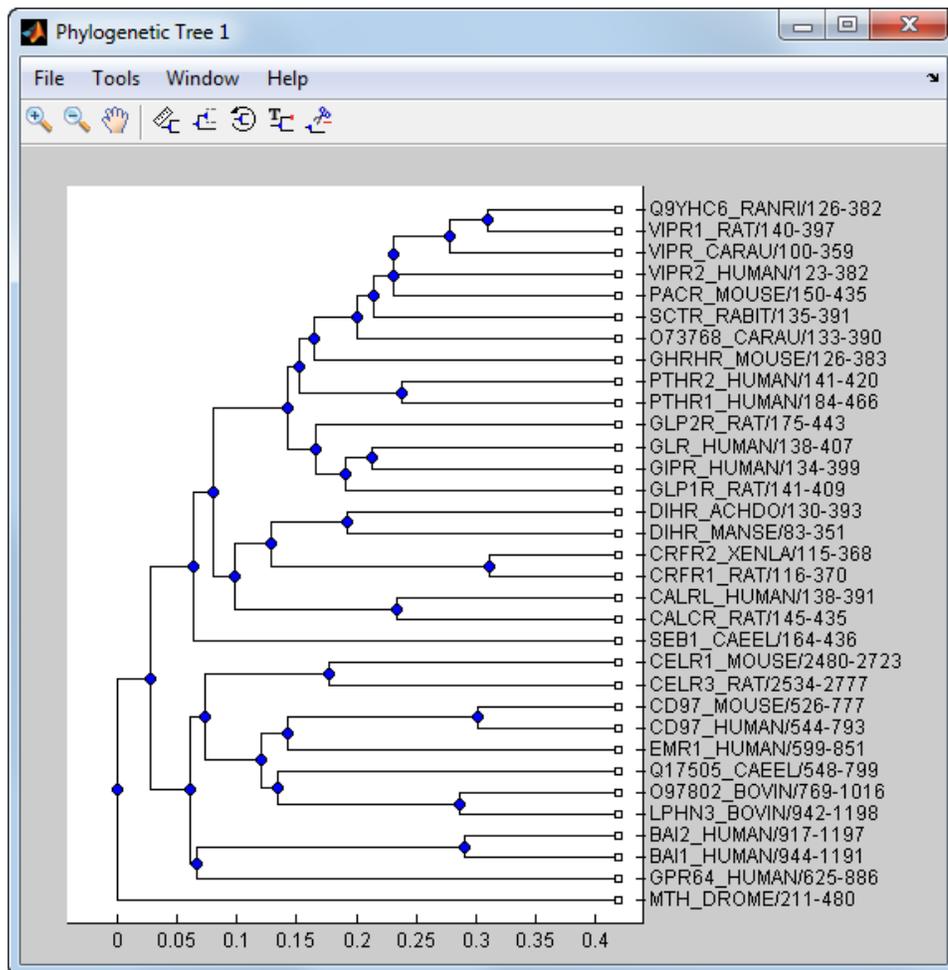
The MATLAB software creates a `phytree` object.

```
Phylogenetic tree object with 33 leaves (32 branches)
```

- 2 View the phylogenetic tree using the app.

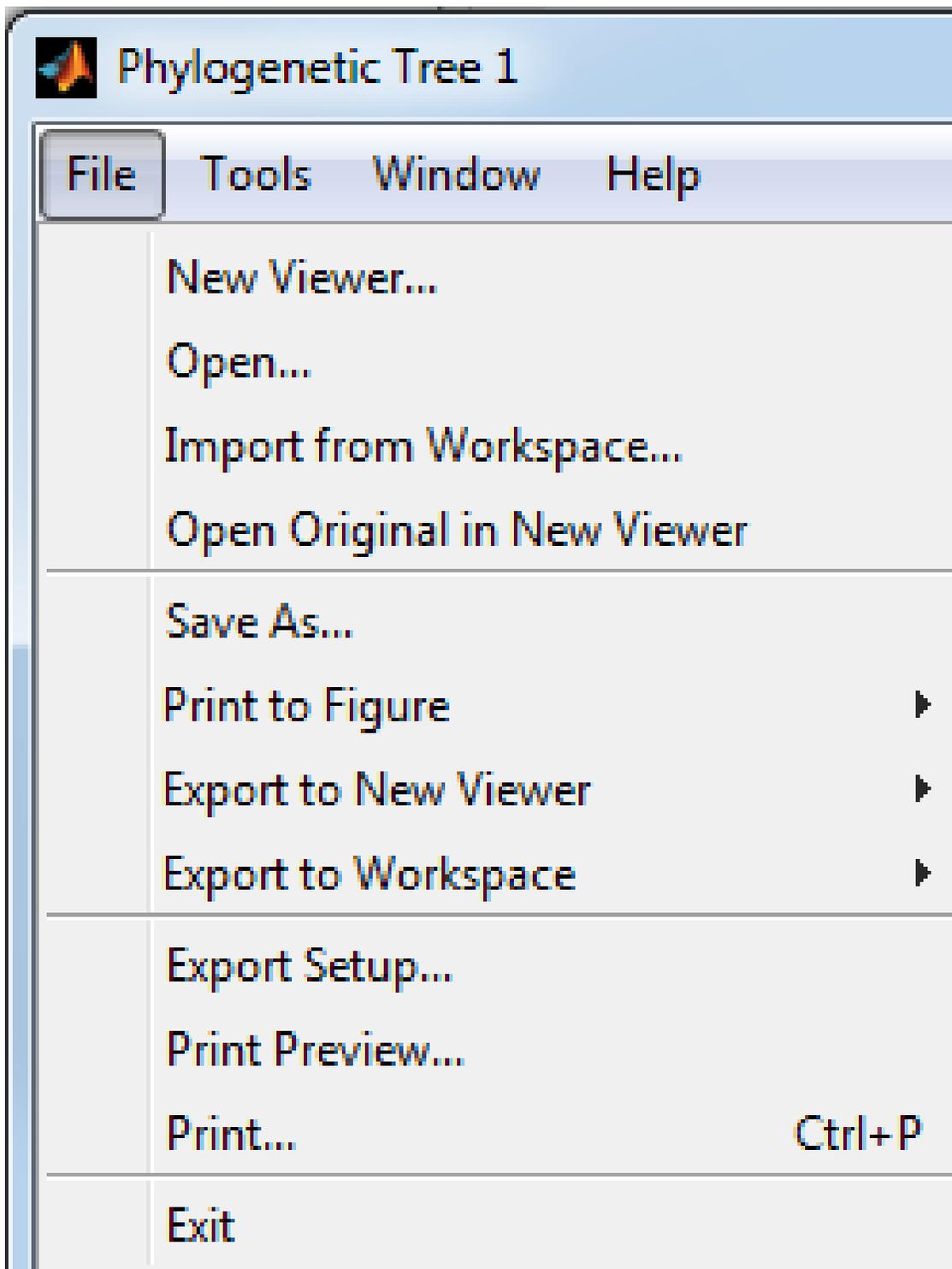
```
phytreeviewer(tr)
```

Alternatively, click **Phylogenetic Tree** on the **Apps** tab.



File Menu

The **File** menu includes the standard commands for opening and closing a file, and it includes commands to use phytree object data from the MATLAB Workspace. The **File** menu commands are shown below.

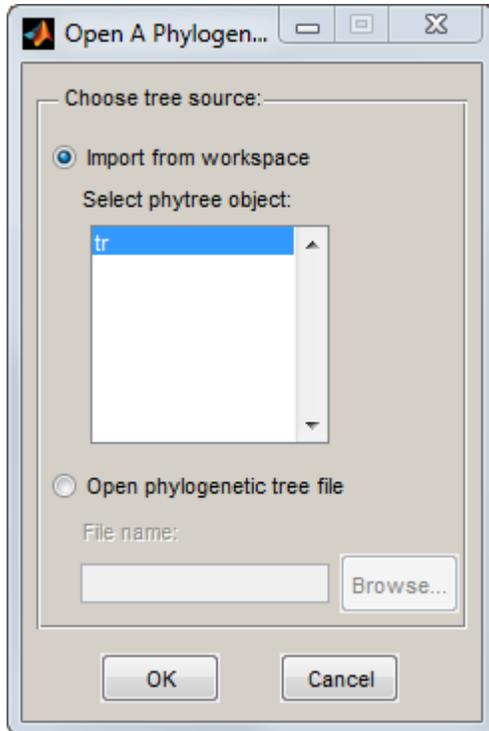


New Viewer Command

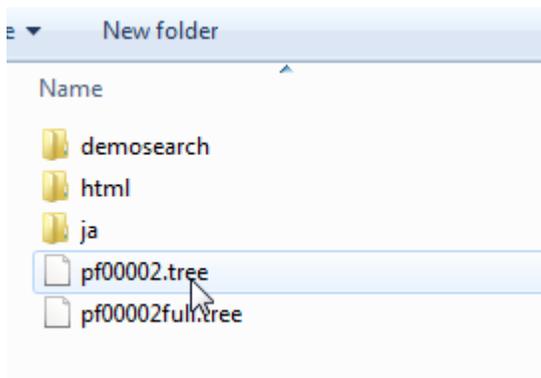
Use the **New Viewer** command to open tree data from a file into a second Phylogenetic Tree viewer.

- 1 From the **File** menu, select **New Viewer**.

The **Open A Phylogenetic Tree** dialog box opens.



- 2 Choose the source for a tree.
 - MATLAB Workspace — Select the **Import from Workspace** options, and then select a phytree object from the list.
 - File — Select the **Open phylogenetic tree file** option, click the **Browse** button, select a directory, select a file with the extension `.tree`, and then click **Open**. The toolbox uses the file extension `.tree` for Newick-formatted files, but you can use any Newick-formatted file with any extension.



A second Phylogenetic Tree viewer opens with tree data from the selected file.

Open Command

Use the **Open** command to read tree data from a Newick-formatted file and display that data in the app.

- 1 From the **File** menu, click **Open**.

The **Select Phylogenetic Tree File** dialog box opens.

- 2 Select a directory, select a Newick-formatted file, and then click **Open**. The app uses the file extension `.tree` for Newick-formatted files, but you can use any Newick-formatted file with any extension.

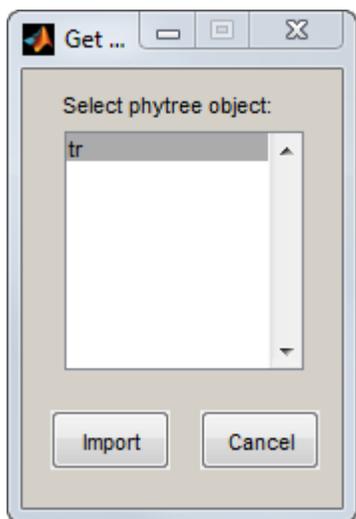
The app replaces the current tree data with data from the selected file.

Import from Workspace Command

Use the **Import from Workspace** command to read tree data from a `phytree` object in the MATLAB Workspace and display the data using the app.

- 1 From the **File** menu, select **Import from Workspace**.

The **Get Phytree Object** dialog box opens.



- 2 From the list, select a `phytree` object in the MATLAB Workspace.
- 3 Click the **Import** button.

The app replaces the current tree data with data from the selected object.

Open Original in New Viewer

There may be times when you make changes that you would like to undo. The **Phylogenetic Tree** app does not have an undo command, but you can get back to the original tree you started viewing with the **Open Original in New Viewer** command.

From the **File** menu, select **Open Original in New Viewer**.

A new Phylogenetic Tree viewer opens with the original tree.

Save As Command

After you create a `phytree` object or prune a tree from existing data, you can save the resulting tree in a Newick-formatted file. The sequence data used to create the `phytree` object is not saved with the tree.

- 1 From the **File** menu, select **Save As**.

The **Save Phylogenetic tree as** dialog box opens.

- 2 In the **Filename** box, enter the name of a file. The toolbox uses the file extension `.tree` for Newick-formatted files, but you can use any file extension.
- 3 Click **Save**.

The app saves tree data without the deleted branches, and it saves changes to branch and leaf names. Formatting changes such as branch rotations, collapsed branches, and zoom settings are not saved in the file.

Export to New Viewer Command

Because some of the Phylogenetic Tree viewer commands cannot be undone (for example, the Prune command), you might want to make a copy of your tree before trying a command. At other times, you might want to compare two views of the same tree, and copying a tree to a new tool window allows you to make changes to both tree views independently.

- 1 Select **File > Export to New Viewer**, and then select either **With Hidden Nodes** or **Only Displayed**.

A new Phylogenetic Tree viewer opens with a copy of the tree.

- 2 Use the new figure to continue your analysis.

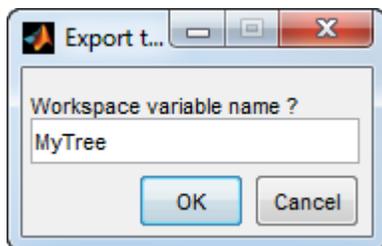
Export to Workspace Command

The **Phylogenetic Tree** app can open Newick-formatted files with tree data. However, it does not create a `phytree` object in the MATLAB Workspace. If you want to programmatically explore phylogenetic trees, you need to use the **Export to Workspace** command.

- 1 Select **File > Export to Workspace**, and then select either **With Hidden Nodes** or **Only Displayed**.

The **Export to Workspace** dialog box opens.

- 2 In the **Workspace variable name** box, enter the name for your phylogenetic tree data. For example, enter `MyTree`.



- 3 Click **OK**.

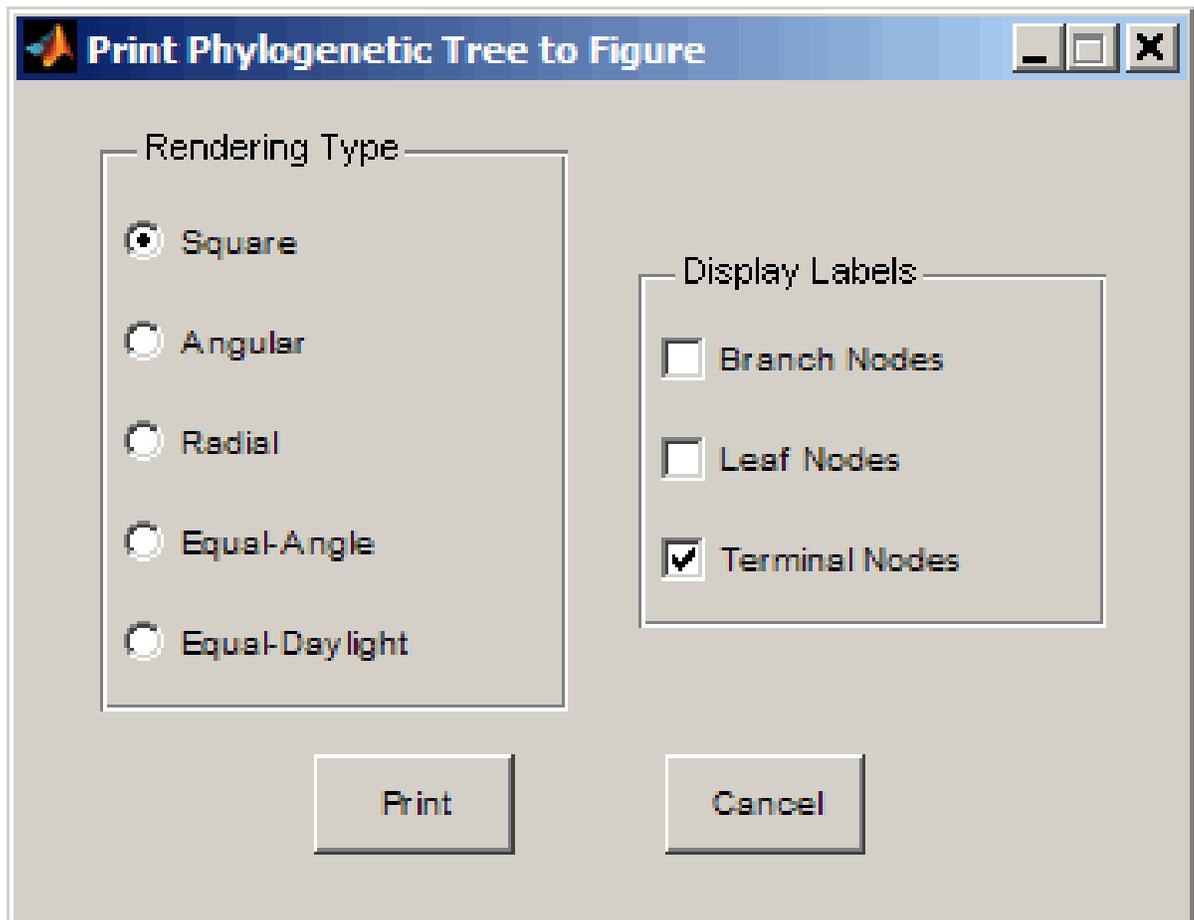
The app creates a `phytree` object in the MATLAB Workspace.

Print to Figure Command

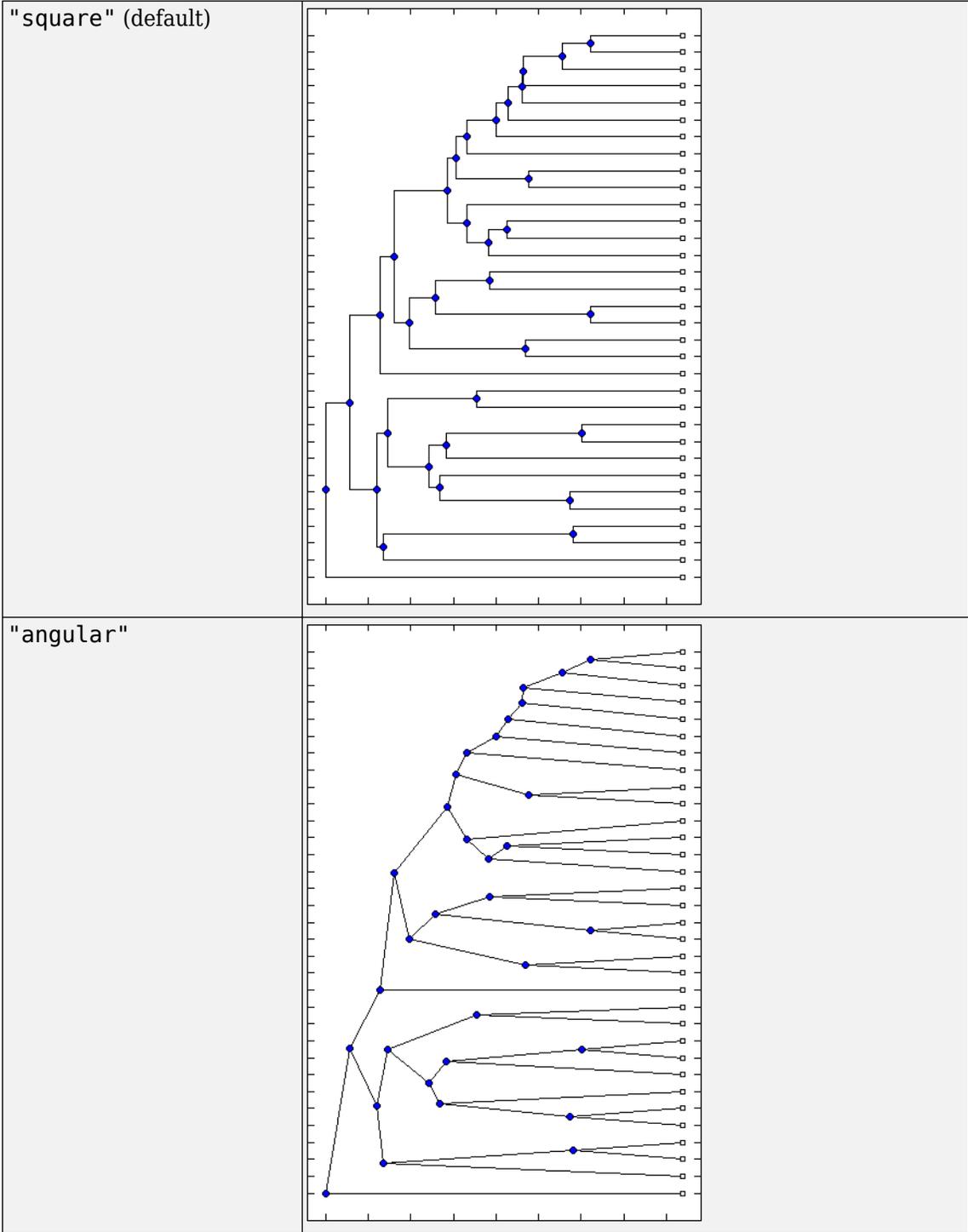
After you have explored the relationships between branches and leaves in your tree, you can copy the tree to a MATLAB Figure window. Using a Figure window lets you use all the features for annotating, changing font characteristics, and getting your figure ready for publication. Also, from the Figure window, you can save an image of the tree as it was displayed in the **Phylogenetic Tree** app.

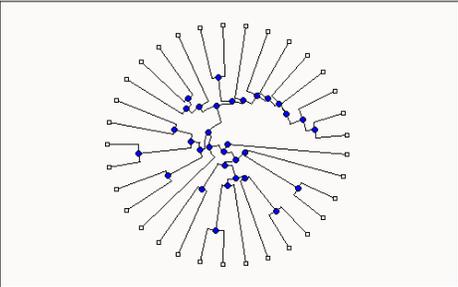
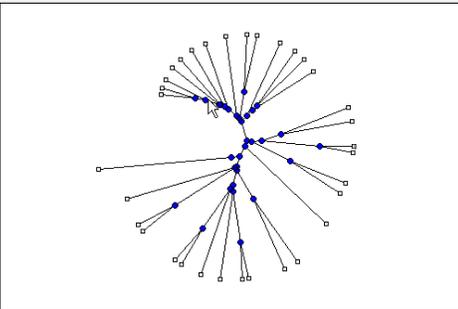
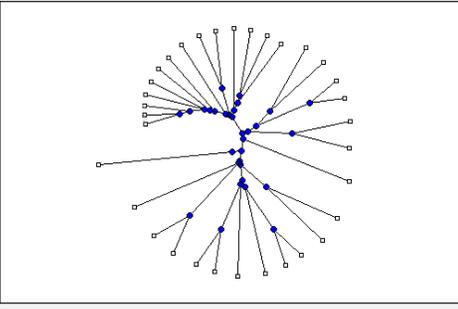
- 1 From the **File** menu, select **Print to Figure**, and then select either **With Hidden Nodes** or **Only Displayed**.

The **Print Phylogenetic Tree to Figure** dialog box opens.



- 2 Select one of the **Rendering Types**.



<p>"radial"</p>	
<p>"equalangle"</p>	<p>This method hides the significance of the root node and emphasizes clusters, thereby making it useful for visually assessing clusters and detecting outliers.</p> 
<p>"equaldaylight"</p>	<p>This method hides the significance of the root node and emphasizes clusters, thereby making it useful for visually assessing clusters and detecting outliers.</p> 

3 Select the **Display Labels** you want on your figure. You can select from all to none of the options.

- **Branch Nodes** — Display branch node names on the figure.
- **Leaf Nodes** — Display leaf node names on the figure.
- **Terminal Nodes** — Display terminal node names on the right border.

4 Click the **Print** button.

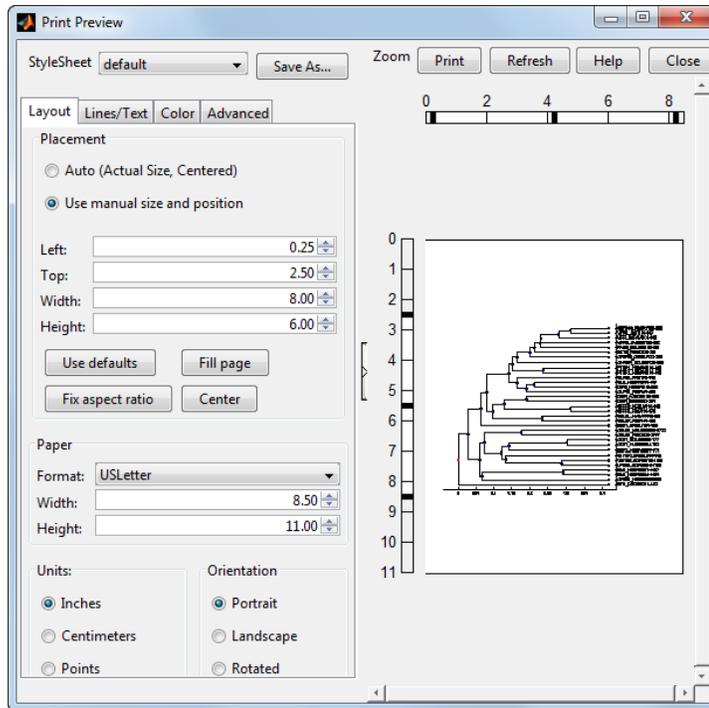
A new Figure window opens with the characteristics you selected.

Print Preview Command

When you print from the **Phylogenetic Tree** app or a MATLAB Figure window (with a tree published from the viewer), you can specify setup options for printing a tree.

- 1 From the **File** menu, select **Print Preview**.

The **Print Preview** window opens, which you can use to select page formatting options.



- 2 Select the page formatting options and values you want, and then click **Print**.

Print Command

Use the **Print** command to make a copy of your phylogenetic tree after you use the **Print Preview** command to select formatting options.

- 1 From the **File** menu, select **Print**.

The **Print** dialog box opens.

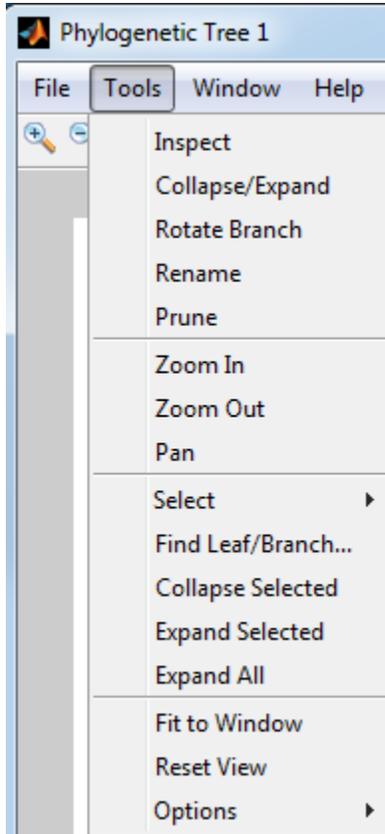
- 2 From the **Name** list, select a printer, and then click **OK**.

Tools Menu

Use the **Tools** menu to:

- Explore branch paths
- Rotate branches
- Find, rename, hide, and prune branches and leaves.

The **Tools** menu and toolbar contain most of the commands specific to trees and phylogenetic analysis. Use these commands and modes to edit and format your tree interactively. The **Tools** menu commands are:



Inspect Mode

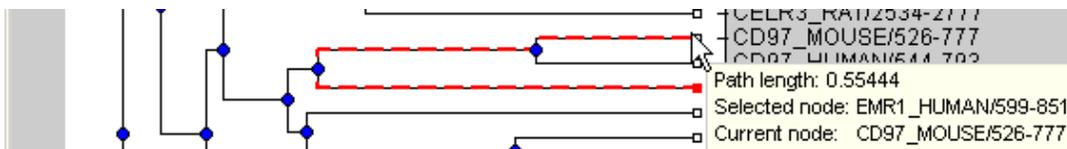
Viewing a phylogenetic tree in the **Phylogenetic Tree** app provides a rough idea of how closely related two sequences are. However, to see exactly how closely related two sequences are, measure the distance of the path between them. Use the **Inspect** command to display and measure the path between two sequences.

- 1 Select **Tools > Inspect**, or from the toolbar, click the **Inspect Tool Mode** icon .

The app is set to inspect mode.

- 2 Click a branch or leaf node (selected node), and then hover your cursor over another branch or leaf node (current node).

The app highlights the path between the two nodes and displays the path length in the pop-up window. The path length is the patristic distance calculated by the `seqpdist` function.



Collapse and Expand Branch Mode

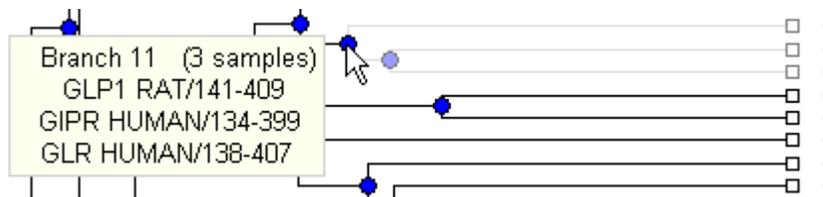
Some trees have thousands of leaf and branch nodes. Displaying all the nodes can create an unreadable tree diagram. By collapsing some branches, you can better see the relationships between the remaining nodes.

- 1 Select **Tools > Collapse/Expand**, or from the toolbar, click the **Collapse/Expand Branch Mode** icon .

The app is set to collapse/expand mode.

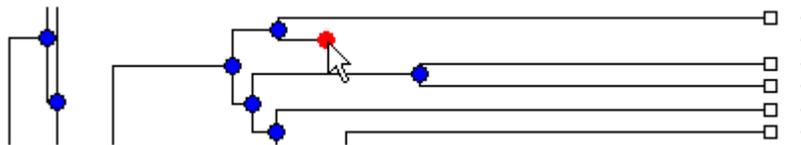
- 2 Point to a branch.

The paths, branch nodes, and leaf nodes below the selected branch appear in gray, indicating you selected them to collapse (hide from view).



- 3 Click the branch node.

The app hides the display of paths, branch nodes, and leaf nodes below the selected branch. However, it does not remove the data.



- 4 To expand a collapsed branch, click it or select **Tools > Reset View**.

Tip After collapsing nodes, you can redraw the tree by selecting **Tools > Fit to Window**.

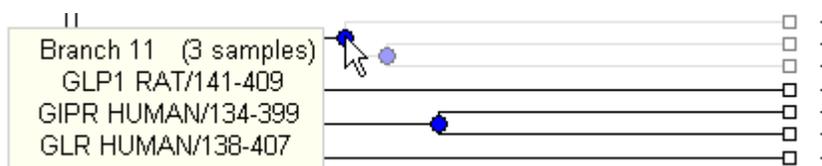
Rotate Branch Mode

A phylogenetic tree is initially created by pairing the two most similar sequences and then adding the remaining sequences in a decreasing order of similarity. You can rotate branches to emphasize the direction of evolution.

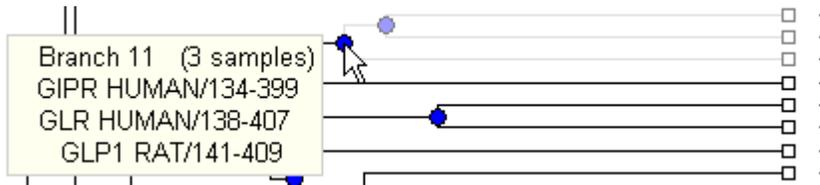
- 1 Select **Tools > Rotate Branch**, or from the toolbar, click the **Rotate Branch Mode** icon .

The app is set to rotate branch mode.

- 2 Point to a branch node.



- Click the branch node.



The branch and leaf nodes below the selected branch node rotate 180 degrees around the branch node.

- To undo the rotation, simply click the branch node again.

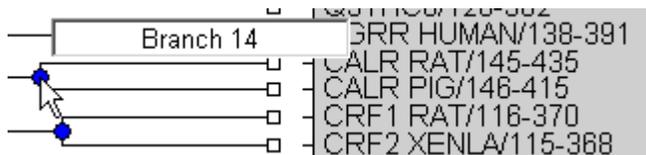
Rename Leaf or Branch Mode

The **Phylogenetic Tree** app takes the node names from a phyt tree object and creates numbered branch names starting with Branch 1. You can edit any of the leaf or branch names.

- Select **Tools > Rename**, or from the toolbar, click the **Rename Leaf/Branch Mode** icon .

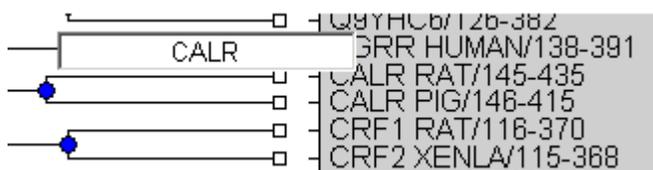
The app is set to rename mode.

- Click a branch or leaf node.



A text box opens with the current name of the node.

- In the text box, edit or enter a new name.



- To accept your changes and close the text box, click outside of the text box. To save your changes, select **File > Save As**.

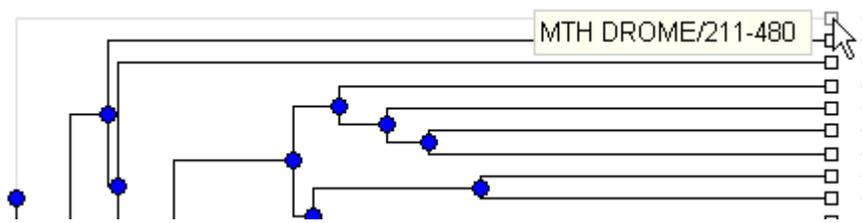
Prune (Delete) Leaf or Branch Mode

Your tree can contain leaves that are far outside the phylogeny, or it can have duplicate leaves that you want to remove.

- Select **Tools > Prune**, or from the toolbar, click the **Prune (delete) Leaf/Branch Mode** icon .

The app is set to prune mode.

- Point to a branch or leaf node.



For a leaf node, the branch line connected to the leaf appears in gray. For a branch node, the branch lines below the node appear in gray.

Note If you delete nodes (branches or leaves), you cannot undo the changes. The Phylogenetic Tree app does not have an Undo command.

- 3 Click the branch or leaf node.

The tool removes the branch from the figure and rearranges the other nodes to balance the tree structure. It does not recalculate the phylogeny.

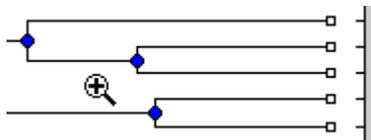
Tip After pruning nodes, you can redraw the tree by selecting **Tools > Fit to Window**.

Zoom In, Zoom Out, and Pan Commands

The Zoom and Pan commands are the standard controls for resizing and moving the screen in any MATLAB Figure window.

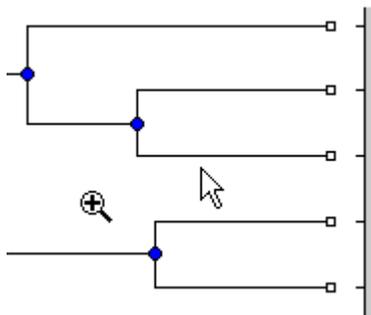
- 1 Select **Tools > Zoom In**, or from the toolbar, click the **Zoom In** icon .

The app activates zoom in mode and changes the cursor to a magnifying glass.



- 2 Place the cursor over the section of the tree diagram you want to enlarge and then click.

The tree diagram doubles its size.



- 3 From the toolbar click the **Pan** icon .

- 4 Move the cursor over the tree diagram, left-click, and drag the diagram to the location you want to view.

Tip After zooming and panning, you can reset the tree to its original view, by selecting **Tools > Reset View**.

Select Submenu

Select a single branch or leaf node by clicking it. Select multiple branch or leaf nodes by **Shift**-clicking the nodes, or click-dragging to draw a box around nodes.

Use the **Select** submenu to select specific branch and leaf nodes based on different criteria.

- **Select By Distance** — Displays a slider bar at the top of the window, which you slide to specify a distance threshold. Nodes whose distance from the selected node are below this threshold appear in red. Nodes whose distance from the selected node are above this threshold appear in blue.
- **Select Common Ancestor** — For all selected nodes, highlights the closest common ancestor branch node in red.
- **Select Leaves** — If one or more nodes are selected, highlights the nodes that are leaf nodes in red. If no nodes are selected, highlights all leaf nodes in red.
- **Propagate Selection** — For all selected nodes, highlights the descendant nodes in red.
- **Swap Selection** — Clears all selected nodes and selects all deselected nodes.

After selecting nodes using one of the previous commands, hide and show the nodes using the following commands:

- **Collapse Selected**
- **Expand Selected**
- **Expand All**

Clear all selected nodes by clicking anywhere else in the Phylogenetic Tree app.

Find Leaf or Branch Command

Phylogenetic trees can have thousands of leaves and branches, and finding a specific node can be difficult. Use the **Find Leaf/Branch** command to locate a node using its name or part of its name.

- 1 Select **Tools > Find Leaf/Branch**.

The Find Leaf/Branch dialog box opens.



- 2 In the **Regular Expression to match** box, enter a name or partial name of a branch or leaf node.
- 3 Click **OK**.

The branch or leaf nodes that match the expression appear in red.

After selecting nodes using the **Find Leaf/Branch** command, you can hide and show the nodes using the following commands:

- **Collapse Selected**
- **Expand Selected**
- **Expand All**

Collapse Selected, Expand Selected, and Expand All Commands

When you select nodes, either manually or using the previous commands, you can then collapse them by selecting **Tools > Collapse Selected**.

The data for branches and leaves that you hide using the **Collapse/Expand** or **Collapse Selected** command are not removed from the tree. You can display selected or all hidden data using the **Expand Selected** or **Expand All** command.

Fit to Window Command

After you hide nodes with the collapse commands, or delete nodes with the **Prune** command, there can be extra space in the tree diagram. Use the **Fit to Window** command to redraw the tree diagram to fill the entire Figure window.

Select **Tools > Fit to Window**.

Reset View Command

Use the **Reset View** command to remove formatting changes such as collapsed branches and zooms.

Select **Tools > Reset View**.

Options Submenu

Use the **Options** command to select the behavior for the zoom and pan modes.

- **Unconstrained Zoom** — Allow zooming in both horizontal and vertical directions.
- **Horizontal Zoom** — Restrict zooming to the horizontal direction.
- **Vertical Zoom** (default) — Restrict zooming to the vertical direction.
- **Unconstrained Pan** — Allow panning in both horizontal and vertical directions.
- **Horizontal Pan** — Restrict panning to the horizontal direction.
- **Vertical Pan** (default) — Restrict panning to the vertical direction.

Window Menu

This section illustrates how to switch to any open window.

The **Window** menu is standard on MATLAB interfaces and Figure windows. Use this menu to select any opened window.

Help Menu

This section illustrates how to select quick links to the Bioinformatics Toolbox documentation for phylogenetic analysis functions, tutorials, and the **Phylogenetic Tree** app reference.

Use the **Help** menu to select quick links to the Bioinformatics Toolbox documentation for phylogenetic analysis functions, tutorials, and the `phytreviewer` reference.

Building a Phylogenetic Tree for the Hominidae Species

This example shows how to construct phylogenetic trees from mtDNA sequences for the Hominidae taxa (also known as pongidae). This family embraces the gorillas, chimpanzees, orangutans and humans.

Introduction

The mitochondrial D-loop is one of the fastest mutating sequence regions in animal DNA, and therefore, is often used to compare closely related organisms. The origin of modern man is a highly debated issue that has been addressed by using mtDNA sequences. The limited genetic variability of human mtDNA has been explained in terms of a recent common genetic ancestry, thus implying that all modern-population mtDNAs likely originated from a single woman who lived in Africa less than 200,000 years.

Retrieving Sequence Data from GenBank®

This example uses mitochondrial D-loop sequences isolated for different hominidae species with the following GenBank Accession numbers.

```
%      Species Description      GenBank Accession
data = {'German_Neanderthal'    'AF011222';
        'Russian_Neanderthal'   'AF254446';
        'European_Human'       'X90314'   ;
        'Mountain_Gorilla_Rwanda' 'AF089820';
        'Chimp_Troglodytes'     'AF176766';
        'Puti_Orangutan'        'AF451972';
        'Jari_Orangutan'        'AF451964';
        'Western_Lowland_Gorilla' 'AY079510';
        'Eastern_Lowland_Gorilla' 'AF050738';
        'Chimp_Schweinfurthii'  'AF176722';
        'Chimp_Vellerosus'      'AF315498';
        'Chimp_Verus'          'AF176731';
    };
```

You can use the `getgenbank` function inside a for-loop to retrieve the sequences from the NCBI data repository and load them into MATLAB®.

```
for ind = 1:length(data)
    primates(ind).Header = data{ind,1};
    primates(ind).Sequence = getgenbank(data{ind,2}, 'sequenceonly', 'true');
end
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore, the results of this example might be slightly different when you use up-to-date sequences.

```
load('primates.mat')
```

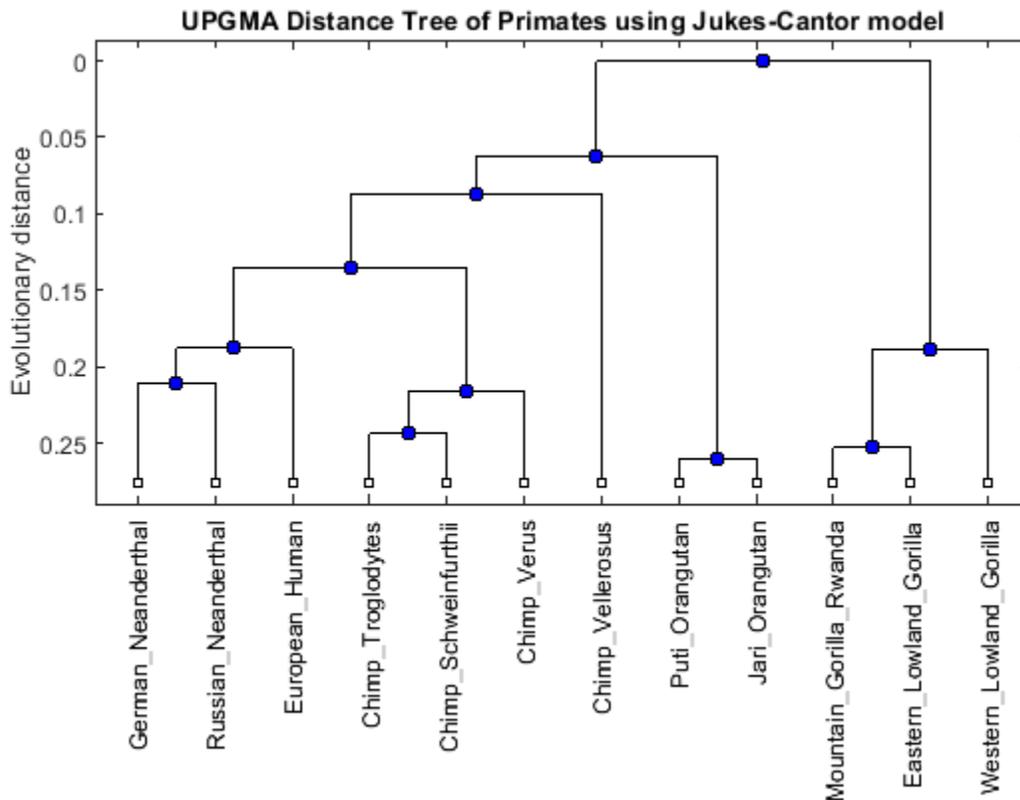
Building a UPGMA Phylogenetic Tree using Distance Methods

Compute pairwise distances using the 'Jukes-Cantor' formula and the phylogenetic tree with the 'UPGMA' distance method. Since the sequences are not pre-aligned, `seqpdist` performs a pairwise alignment before computing the distances.

```
distances = seqpdist(primates, 'Method', 'Jukes-Cantor', 'Alpha', 'DNA');
UPGMAtree = seqlinkage(distances, 'UPGMA', primates)
```

```
h = plot(UPGMAtree, 'orient', 'top');
title('UPGMA Distance Tree of Primates using Jukes-Cantor model');
ylabel('Evolutionary distance')
```

Phylogenetic tree object with 12 leaves (11 branches)



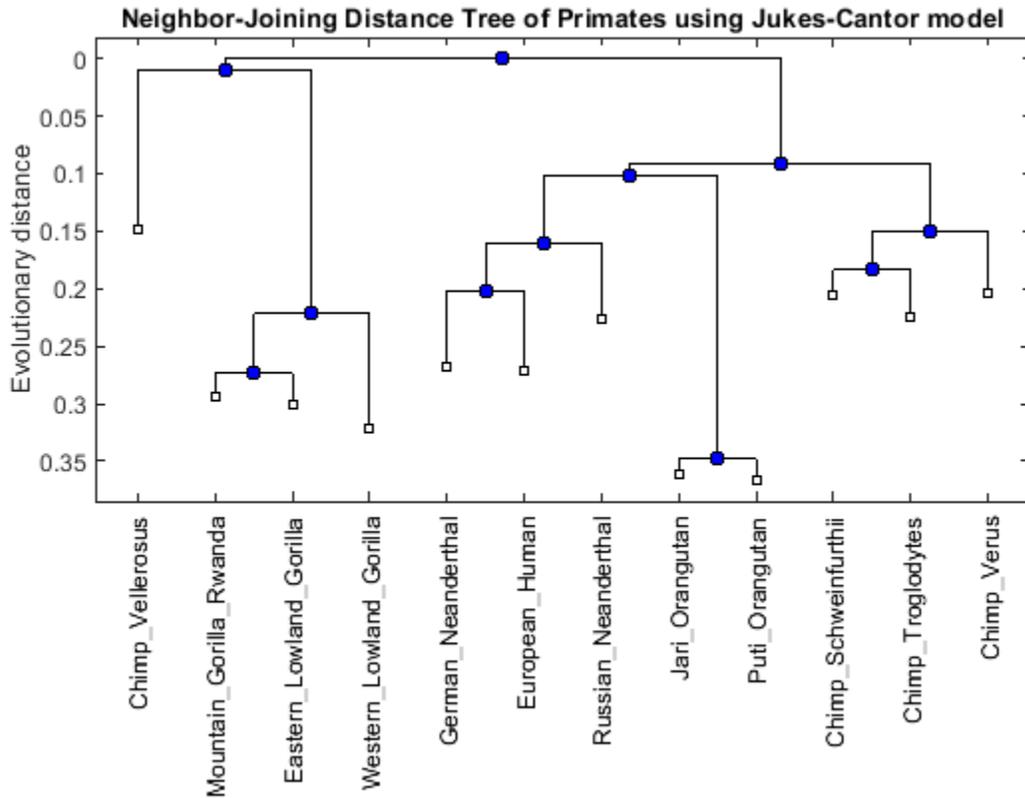
Building a Neighbor-Joining Phylogenetic Tree using Distance Methods

Alternate tree topologies are important to consider when analyzing homologous sequences between species. A neighbor-joining tree can be built using the `seqneighjoin` function. Neighbor-joining trees use the pairwise distance calculated above to construct the tree. This method performs clustering using the minimum evolution method.

```
NJtree = seqneighjoin(distances, 'equivar', primates)
```

```
h = plot(NJtree, 'orient', 'top');
title('Neighbor-Joining Distance Tree of Primates using Jukes-Cantor model');
ylabel('Evolutionary distance')
```

Phylogenetic tree object with 12 leaves (11 branches)



Comparing Tree Topologies

Notice that different phylogenetic reconstruction methods result in different tree topologies. The neighbor-joining tree groups Chimp Vellerosus in a clade with the gorillas, whereas the UPGMA tree groups it near chimps and orangutans. The `getcanonical` function can be used to compare these isomorphic trees.

```
sametree = isequal(getcanonical(UPGMAtree), getcanonical(NJtree))
```

```
sametree =
```

```
logical
```

```
0
```

Exploring the UPGMA Phylogenetic Tree

You can explore the phylogenetic tree by considering the nodes (leaves and branches) within a given patristic distance from the 'European Human' entry and reduce the tree to the sub-branches of interest by pruning away non-relevant nodes.

```
names = get(UPGMAtree, 'LeafNames')
```

```
[h_all, h_leaves] = select(UPGMAtree, 'reference', 3, 'criteria', 'distance', 'threshold', 0.3);
```

```
subtree_names = names(h_leaves)
```

```
leaves_to_prune = ~h_leaves;

pruned_tree = prune(UPGMAtree,leaves_to_prune)
h = plot(pruned_tree,'orient','top');
title('Pruned UPGMA Distance Tree of Primates using Jukes-Cantor model');
ylabel('Evolutionary distance')

names =

    12x1 cell array

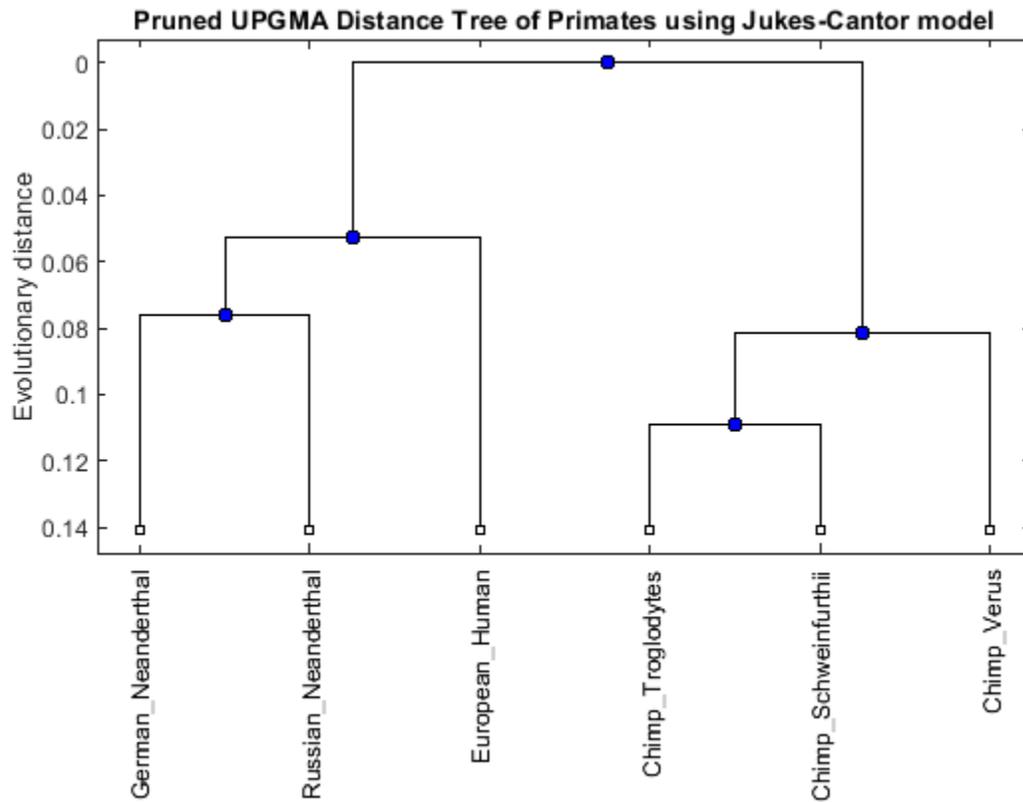
    {'German_Neanderthal'    }
    {'Russian_Neanderthal'  }
    {'European_Human'       }
    {'Chimp_Troglodytes'    }
    {'Chimp_Schweinfurthii' }
    {'Chimp_Verus'          }
    {'Chimp_Vellerosus'     }
    {'Puti_Orangutan'       }
    {'Jari_Orangutan'       }
    {'Mountain_Gorilla_Rwanda'}
    {'Eastern_Lowland_Gorilla'}
    {'Western_Lowland_Gorilla'}

subtree_names =

    6x1 cell array

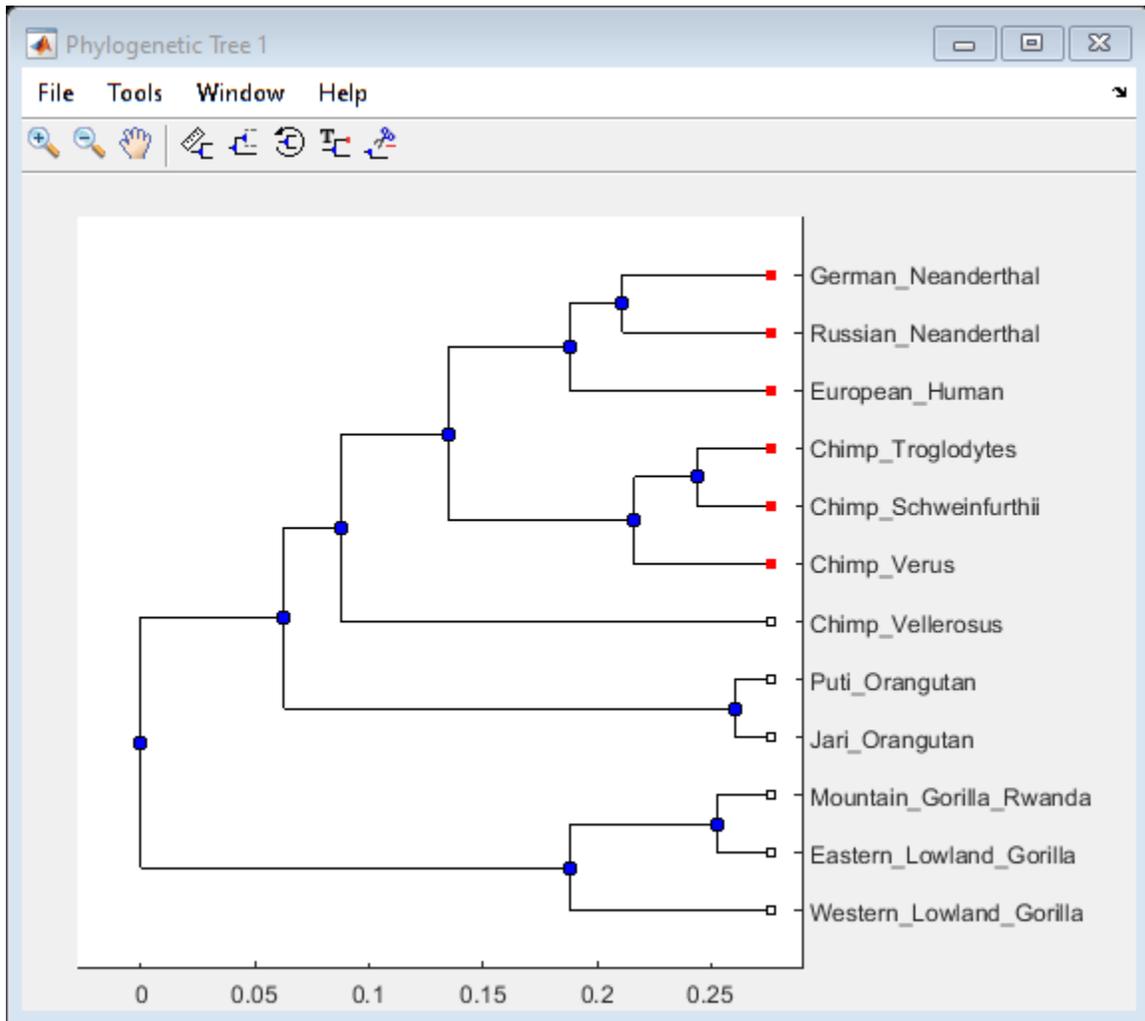
    {'German_Neanderthal' }
    {'Russian_Neanderthal' }
    {'European_Human'     }
    {'Chimp_Troglodytes'  }
    {'Chimp_Schweinfurthii'}
    {'Chimp_Verus'        }

    Phylogenetic tree object with 6 leaves (5 branches)
```



With view you can further explore/edit the phylogenetic tree using an interactive tool. See also [phytreviewer](#).

```
view(UPGMAtree,h_leaves)
```



References

- [1] Ovchinnikov, I.V., et al., "Molecular analysis of Neanderthal DNA from the northern Caucasus", *Nature*, 404(6777):490-3, 2000.
- [2] Sajantila, A., et al., "Genes and languages in Europe: an analysis of mitochondrial lineages", *Genome Research*, 5(1):42-52, 1995.
- [3] Krings, M., et al., "Neandertal DNA sequences and the origin of modern humans", *Cell*, 90(1):19-30, 1997.
- [4] Jensen-Seaman, M.I. and Kidd, K.K., "Mitochondrial DNA variation and biogeography of eastern gorillas", *Molecular Ecology*, 10(9):2241-7, 2001.

Analyzing the Origin of the Human Immunodeficiency Virus

This example shows how to construct phylogenetic trees from multiple strains of the HIV and SIV viruses.

Introduction

Mutations accumulate in the genomes of pathogens, in this case the human/simian immunodeficiency virus, during the spread of an infection. This information can be used to study the history of transmission events, and also as evidence for the origins of the different viral strains.

There are two characterized strains of human AIDS viruses: type 1 (HIV-1) and type 2 (HIV-2). Both strains represent cross-species infections. The primate reservoir of HIV-2 has been clearly identified as the sooty mangabey (*Cercocebus atys*). The origin of HIV-1 is believed to be the common chimpanzee (*Pan troglodytes*).

Retrieve Sequence Information from GenBank®

In this example, the variations in three longest coding regions from seventeen different isolated strains of the Human and Simian immunodeficiency virus are used to construct a phylogenetic tree. The sequences for these virus strains can be retrieved from GenBank® using their accession numbers. The three coding regions of interest, the gag protein, the pol polyprotein and the envelope polyprotein precursor, can then be extracted from the sequences using the CDS information in the GenBank records.

```
%      Description                Accession  CDS:gag/pol/env
data = {'HIV-1 (Zaire)'           'K03454'   [1 2 8] ;
        'HIV1-NDK (Zaire)'       'M27323'   [1 2 8] ;
        'HIV-2 (Senegal)'        'M15390'   [1 2 8] ;
        'HIV2-MCN13'             'AY509259' [1 2 8] ;
        'HIV-2UC1 (IvoryCoast)'   'L07625'   [1 2 8] ;
        'SIVMM251 Macaque'        'M19499'   [1 2 8] ;
        'SIVAGM677A Green monkey' 'M58410'   [1 2 7] ;
        'SIVlhoest L''Hoest monkeys' 'AF075269' [1 2 7] ;
        'SIVcpz Chimpanzees Cameroon' 'AF115393' [1 2 8] ;
        'SIVmnd5440 Mandrillus sphinx' 'AY159322' [1 2 8] ;
        'SIVAGM3 Green monkeys'   'M30931'   [1 2 7] ;
        'SIVMM239 Simian macaque'  'M33262'   [1 2 8] ;
        'SIVcpzUS Chimpanzee'     'AF103818' [1 2 8] ;
        'SIVmon Cercopithecus Monkeys' 'AY340701' [1 2 8] ;
        'SIVcpzTAN1 Chimpanzee'    'AF447763' [1 2 8] ;
        'SIVsmSL92b Sooty Mangabey' 'AF334679' [1 2 8] ;
        };
```

```
numViruses = size(data,1)
```

```
numViruses =
```

```
16
```

You can use the `getgenbank` function to copy the data from GenBank into a structure in MATLAB®. The `SearchURL` field of the structure contains the address of the actual GenBank record. You can browse this record using the `web` command.

```
acc_num = data{1,2};  
lentivirus = getgenbank(acc_num);  
web(lentivirus(1).SearchURL)
```

Retrieve the sequence information from the NCBI GenBank database for the rest of the accession numbers.

```
for ind = 2:numViruses  
    lentivirus(ind) = getgenbank(data{ind,2});  
end
```

For your convenience, previously downloaded sequences are included in a MAT-file. Note that data in public repositories is frequently curated and updated; therefore the results of this example might be slightly different when you use up-to-date datasets.

```
load('lentivirus.mat')
```

Extract CDS for the GAG, POL, and ENV coding regions. Then extract the nucleotide sequences using the CDS pointers.

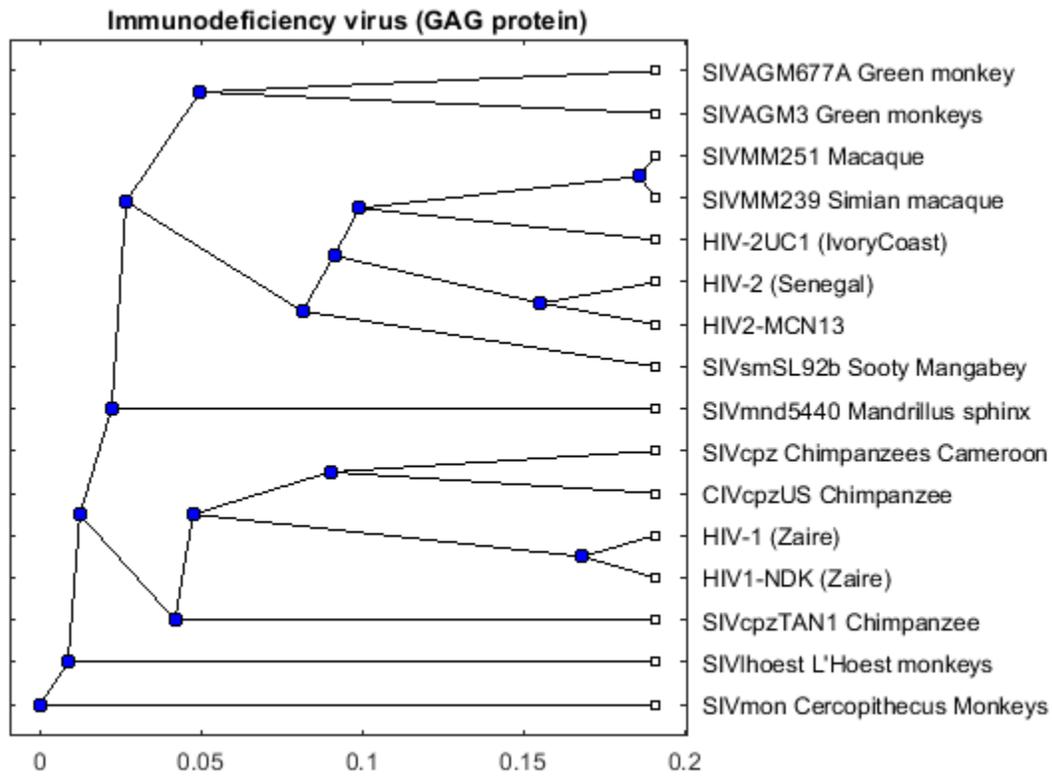
```
for ind = 1:numViruses  
    temp_seq = lentivirus(ind).Sequence;  
    temp_seq = regexprep(temp_seq, '[nry]', 'a');  
    CDSs = lentivirus(ind).CDS(data{ind,3});  
    gag(ind).Sequence = temp_seq(CDSs(1).indices(1):CDSs(1).indices(2));  
    pol(ind).Sequence = temp_seq(CDSs(2).indices(1):CDSs(2).indices(2));  
    env(ind).Sequence = temp_seq(CDSs(3).indices(1):CDSs(3).indices(2));  
end
```

Phylogenetic Tree Reconstruction

The `seqpdist` and `seqlinkage` commands are used to construct a phylogenetic tree for the GAG coding region using the 'Tajima-Nei' method to measure the distance between the sequences and the unweighted pair group method using arithmetic averages, or 'UPGMA' method, for the hierarchical clustering. The 'Tajima-Nei' method is only defined for nucleotides, therefore nucleotide sequences are used rather than the translated amino acid sequences. The distance calculation may take quite a few minutes as it is very computationally intensive.

```
gagd = seqpdist(gag, 'method', 'Tajima-Nei', 'Alphabet', 'NT', 'indel', 'pair');  
gagtree = seqlinkage(gagd, 'UPGMA', data(:,1))  
plot(gagtree, 'type', 'angular');  
title('Immunodeficiency virus (GAG protein)')
```

```
Phylogenetic tree object with 16 leaves (15 branches)
```



Next construct a phylogenetic tree for the POL polyproteins using the 'Jukes-Cantor' method to measure distance between sequences and the weighted pair group method using arithmetic averages, or 'WPGMA' method, for the hierarchical clustering. The 'Jukes-Cantor' method is defined for amino-acids sequences, which, being significantly shorter than the corresponding nucleotide sequences, means that the calculation of the pairwise distances will be significantly faster.

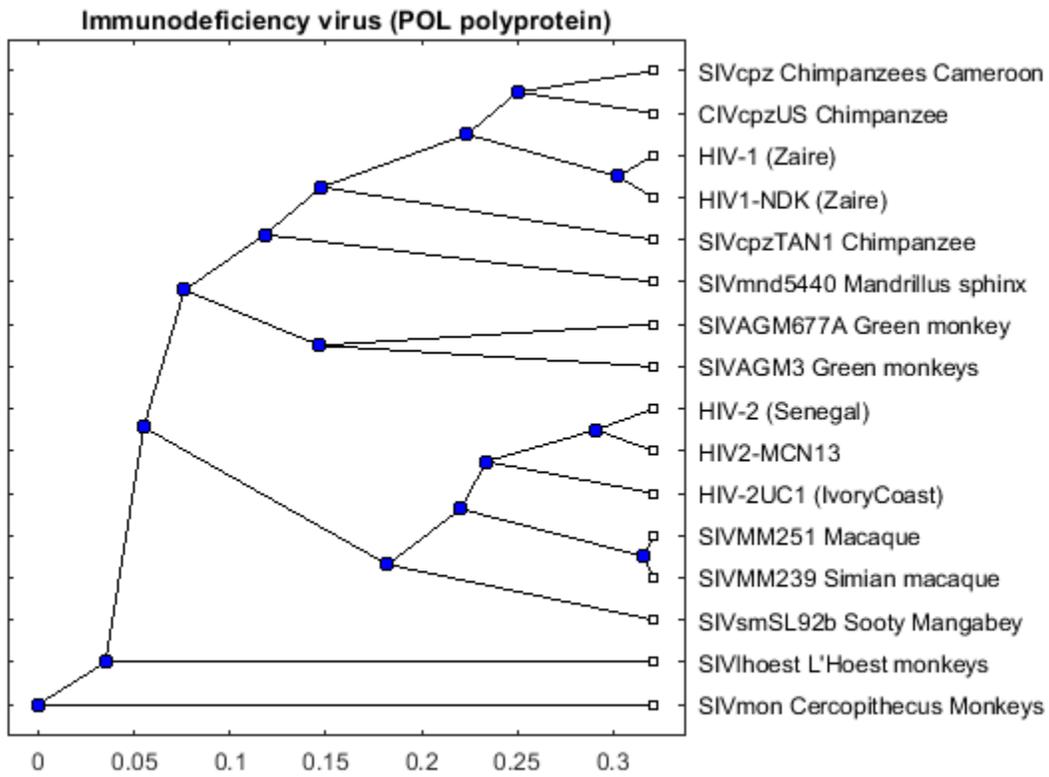
Convert nucleotide sequences to amino acid sequences using `nt2aa`.

```
for ind = 1:numViruses
    aagag(ind).Sequence = nt2aa(gag(ind).Sequence);
    aapol(ind).Sequence = nt2aa(pol(ind).Sequence);
    aaenv(ind).Sequence = nt2aa(env(ind).Sequence);
end
```

Calculate the distance and linkage, and then generate the tree.

```
pold = seqpdist(aapol,'method','Jukes-Cantor','indel','pair');
poltree = seqlinkage(pold,'WPGMA',data(:,1))
plot(poltree,'type','angular');
title('Immunodeficiency virus (POL polyprotein)')
```

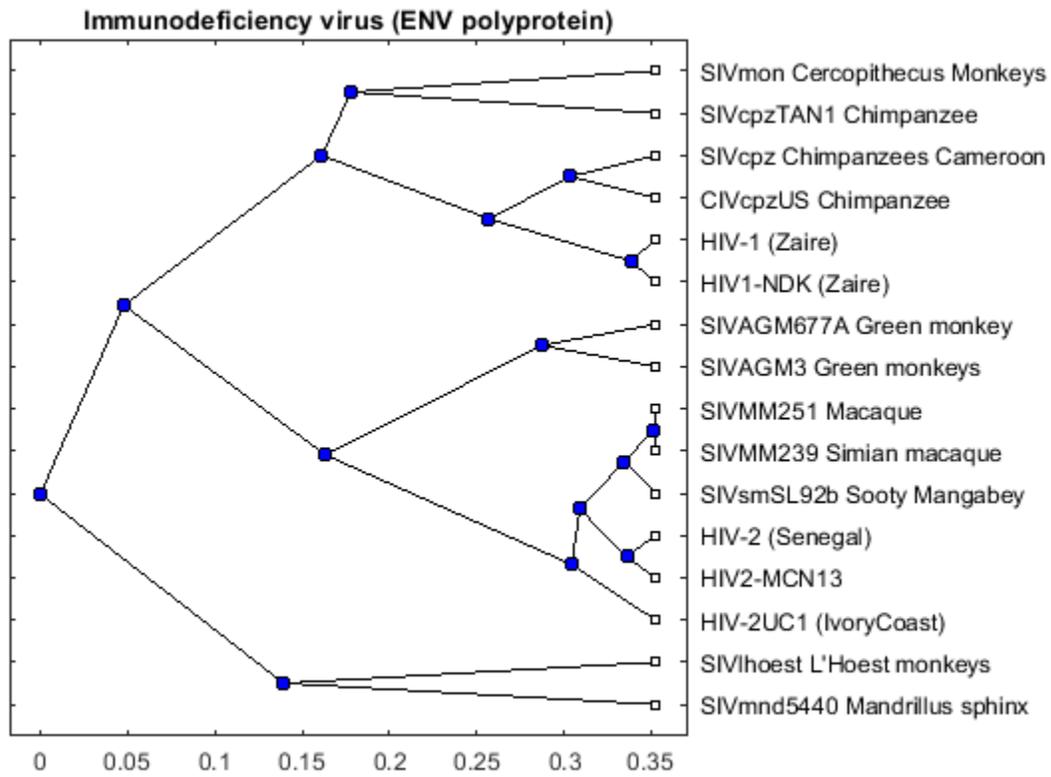
Phylogenetic tree object with 16 leaves (15 branches)



Construct a phylogenetic tree for the ENV polyproteins using the normalized pairwise alignment scores as distances between sequences and the 'UPGMA' method for hierarchical clustering.

```
envd = seqpdist(aaenv, 'method', 'Alignment', 'indel', 'score', ...
               'ScoringMatrix', 'Blosum62');
envtree = seqlinkage(envd, 'UPGMA', data(:,1))
plot(envtree, 'type', 'angular');
title('Immunodeficiency virus (ENV polyprotein)')
```

Phylogenetic tree object with 16 leaves (15 branches)



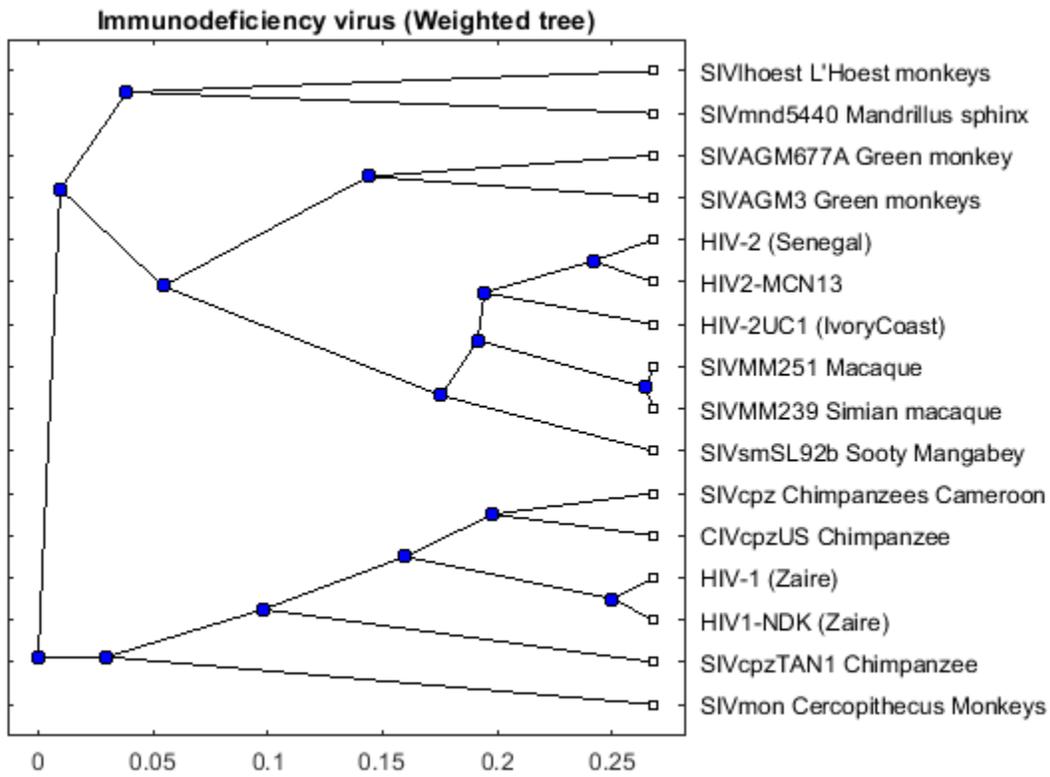
Build a Consensus Tree

The three trees are similar but there are some interesting differences. For example in the POL tree, the 'SIVmnd5440 Mandrillus sphinx' sequence is placed close to the HIV-1 strains, but in the ENV tree it is shown as being very distant to the HIV-1 sequences. Given that the three trees show slightly different results, a consensus tree using all three regions, may give better general information about the complete viruses. A consensus tree can be built using a weighted average of the three trees.

```
weights = [sum(gagd) sum(pold) sum(envd)];
weights = weights / sum(weights);
dist = gagd .* weights(1) + pold .* weights(2) + envd .* weights(3);
```

Note that different metrics were used in the calculation of the pairwise distances. This could bias the consensus tree. You may wish to recalculate the distances for the three regions using the same metric to get an unbiased tree.

```
tree_hiv = seqlinkage(dist, 'average', data(:,1));
plot(tree_hiv, 'type', 'angular');
title('Immunodeficiency virus (Weighted tree)')
```



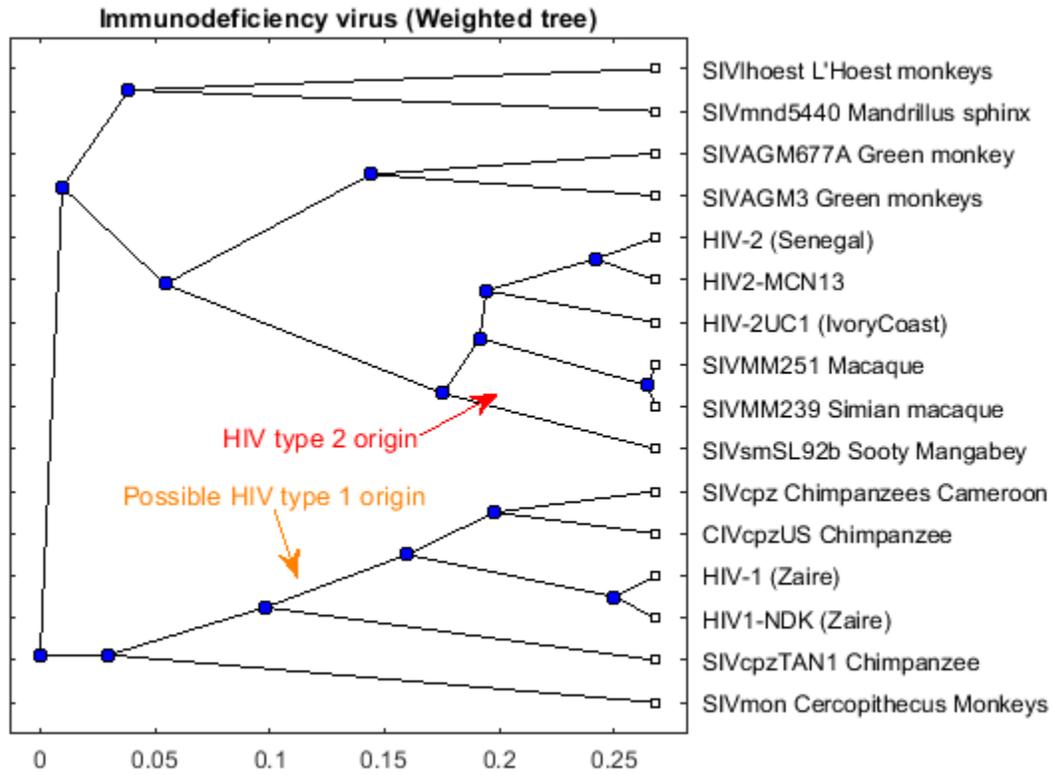
Origins of the HIV Virus

The phylogenetic tree resulting from our analysis illustrates the presence of two clusters and some other isolated strains. The most compact cluster includes all the HIV2 samples; at the top branch of this cluster we observe the sooty mangabey which has been identified as the origin of this lentivirus in humans. The cluster containing the HIV1 strain, however is not as compact as the HIV2 cluster. From the tree it appears that the Chimpanzee is the source of HIV1, however, the origin of the cross-species transmission to humans is still a matter of debate amongst HIV researchers.

```
% Add annotations
```

```
annotation(gcf,'textarrow',[0.29 0.31],[0.36 0.28],'Color',[1 0.5 0],...
    'String',{'Possible HIV type 1 origin'},'TextColor',[1 0.5 0]);
```

```
annotation(gcf,'textarrow',[0.42 0.49],[0.45 0.50],'Color',[1 0 0],...
    'String',{'HIV type 2 origin'},'TextColor',[1 0 0]);
```



References:

- [1] Gao, F., et al., "Origin of HIV-1 in the chimpanzee *Pan troglodytes troglodytes*", *Nature*, 397(6718):436-41, 1999.
- [2] Kestler, H.W., et al., "Comparison of simian immunodeficiency virus isolates", *Nature*, 331(6157):619-22, 1998.
- [3] Alizon, M., et al., "Genetic variability of the AIDS virus: nucleotide sequence analysis of two isolates from African patients", *Cell*, 46(1):63-74, 1986.

Bootstrapping Phylogenetic Trees

This example shows how to generate bootstrap replicates of DNA sequences. The data generated by bootstrapping is used to estimate the confidence of the branches in a phylogenetic tree.

Introduction

Bootstrap, jackknife, and permutation tests are common tests used in phylogenetics to estimate the significance of the branches of a tree. This process can be very time consuming because of the large number of samples that have to be taken in order to have an accurate confidence estimate. The more times the data are sampled the better the analysis. A cluster of computers can shorten the time needed for this analysis by distributing the work to several machines and recombining the data.

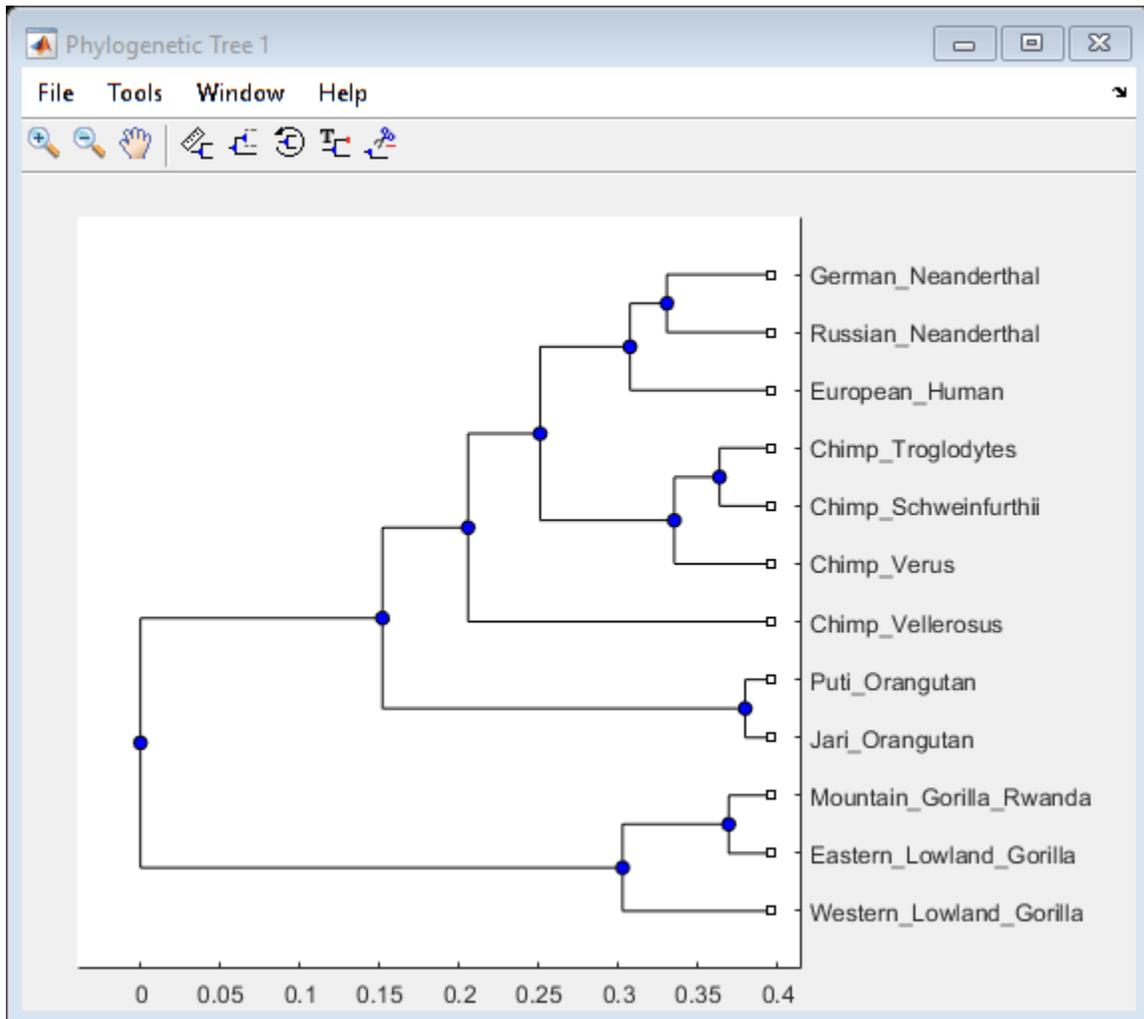
Loading Sequence Data and Building the Original Tree

This example uses 12 pre-aligned sequences isolated from different hominidae species and stored in a FASTA-formatted file. A phylogenetic tree is constructed by using the UPGMA method with pairwise distances. More specifically, the `seqpdist` function computes the pairwise distances among the considered sequences and then the function `seqlinkage` builds the tree and returns the data in a `phytree` object. You can use the `phytreeviewer` function to visualize and explore the tree.

```
primates = fastaread('primatesaligned.fa');
num_seqs = length(primates)

num_seqs =
12

orig_primates_dist = seqpdist(primates);
orig_primates_tree = seqlinkage(orig_primates_dist, 'average', primates);
phytreeviewer(orig_primates_tree);
```



Making Bootstrap Replicates from the Data

A bootstrap replicate is a shuffled representation of the DNA sequence data. To make a bootstrap replicate of the primates data, bases are sampled randomly from the sequences with replacement and concatenated to make new sequences. The same number of bases as the original multiple alignment is used in this analysis, and then gaps are removed to force a new pairwise alignment. The function `randsample` samples the data with replacement. This function can also sample the data randomly without replacement to perform jackknife analysis. For this analysis, 100 bootstrap replicates for each sequence are created.

```
num_boots = 100;
seq_len = length(primates(1).Sequence);

boots = cell(num_boots,1);
for n = 1:num_boots
    reorder_index = randsample(seq_len,seq_len,true);
    for i = num_seqs:-1:1 %reverse order to preallocate memory
        bootseq(i).Header = primates(i).Header;
        bootseq(i).Sequence = strep(primates(i).Sequence(reorder_index),'-', '');
    end
end
```

```
boots{n} = bootseq;
end
```

Computing the Distances Between Bootstraps and Phylogenetic Reconstruction

Determining the distances between DNA sequences for a large data set and building the phylogenetic trees can be time-consuming. Distributing these calculations over several machines/cores decreases the computation time. This example assumes that you have already started a MATLAB® pool with additional parallel resources. For information about setting up and selecting parallel configurations, see "Programming with User Configurations" in the Parallel Computing Toolbox™ documentation. If you do not have the Parallel Computing Toolbox™, the following PARFOR loop executes sequentially without any further modification.

```
fun = @(x) seqlinkage(x, 'average', {primates.Header});
boot_trees = cell(num_boots,1);
parpool('local');
```

Starting parallel pool (parpool) using the 'local' profile ...

```
parfor (n = 1:num_boots)
    dist_tmp = seqpdist(boots{n});
    boot_trees{n} = fun(dist_tmp);
end
delete(gcf('nocreate'));
```

Counting the Branches with Similar Topology

The topology of every bootstrap tree is compared with that of the original tree. Any interior branch that gives the same partition of species is counted. Since branches may be ordered differently among different trees but still represent the same partition of species, it is necessary to get the canonical form for each subtree before comparison. The first step is to get the canonical subtrees of the original tree using the `subtree` and `getcanonical` methods from the Bioinformatics Toolbox™.

```
for i = num_seqs-1:-1:1 % for every branch, reverse order to preallocate
    branch_pointer = i + num_seqs;
    sub_tree = subtree(orig_primates_tree,branch_pointer);
    orig_pointers{i} = getcanonical(sub_tree);
    orig_species{i} = sort(get(sub_tree,'LeafNames'));
end
```

Now you can get the canonical subtrees for all the branches of every bootstrap tree.

```
for j = num_boots:-1:1
    for i = num_seqs-1:-1:1 % for every branch
        branch_ptr = i + num_seqs;
        sub_tree = subtree(boot_trees{j},branch_ptr);
        clusters_pointers{i,j} = getcanonical(sub_tree);
        clusters_species{i,j} = sort(get(sub_tree,'LeafNames'));
    end
end
```

For each subtree in the original tree, you can count how many times it appears within the bootstrap subtrees. To be considered as similar, they must have the same topology and span the same species.

```
count = zeros(num_seqs-1,1);
for i = 1 : num_seqs -1 % for every branch
    for j = 1 : num_boots * (num_seqs-1)
        if isequal(orig_pointers{i},clusters_pointers{j})
```

```

        if isequal(orig_species{i},clusters_species{j})
            count(i) = count(i) + 1;
        end
    end
end
end
end

Pc = count ./ num_boots    % confidence probability (Pc)

Pc = 11x1

    1.0000
    1.0000
    0.9900
    0.9900
    0.5400
    0.5400
    1.0000
    0.4300
    0.3900
    0.3900
    ⋮

```

Visualizing the Confidence Values in the Original Tree

The confidence information associated with each branch node can be stored within the tree by annotating the node names. Thus, you can create a new tree, equivalent to the original primates tree, and annotate the branch names to include the confidence levels computed above. `phytreviewer` displays this data in datatips when the mouse is hovered over the internal nodes of the tree.

```

[ptrs,dist,names] = get(orig_primates_tree,'POINTERS','DISTANCES','NODENAMES');

for i = 1:num_seqs -1 % for every branch
    branch_ptr = i + num_seqs;
    names{branch_ptr} = [names{branch_ptr} ', confidence: ' num2str(100*Pc(i)) ' %'];
end

tr = phytree(ptrs,dist,names)

    Phylogenetic tree object with 12 leaves (11 branches)

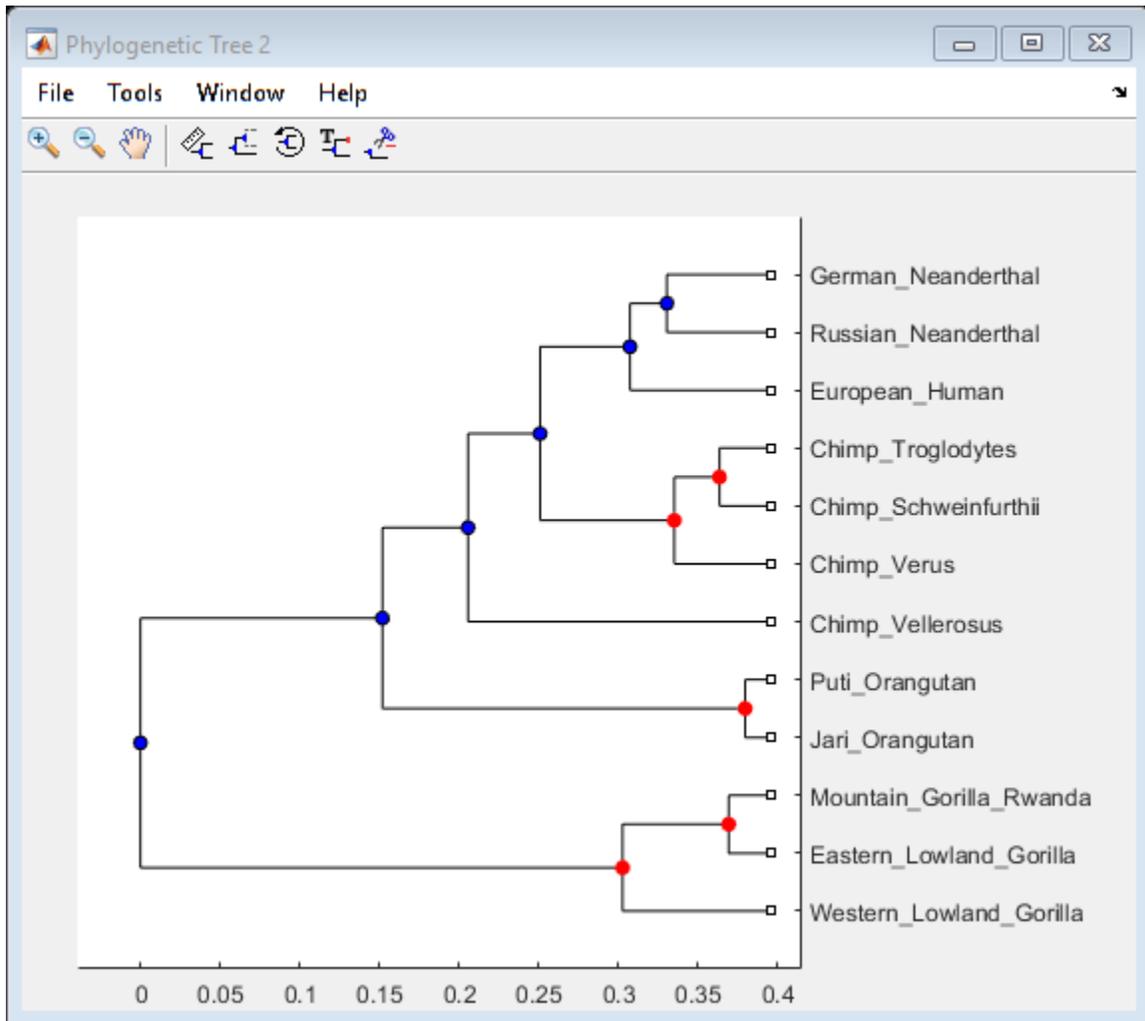
```

You can select the branch nodes with a confidence level greater than a given threshold, for example 0.9, and view these corresponding nodes in the Phylogenetic Tree app. You can also select these branch nodes interactively within the app.

```

high_conf_branch_ptr = find(Pc > 0.9) + num_seqs;
view(tr, high_conf_branch_ptr)

```



References

- [1] Felsenstein, J., "Inferring Phylogenies", Sinaur Associates, Inc., 2004.
- [2] Nei, M. and Kumar, S., "Molecular Evolution and Phylogenetics", Oxford University Press. Chapter 4, 2000.

Mass Spectrometry and Bioanalytics

- “Preprocessing Raw Mass Spectrometry Data” on page 6-2
- “Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling” on page 6-19
- “Identifying Significant Features and Classifying Protein Profiles” on page 6-38
- “Differential Analysis of Complex Protein and Metabolite Mixtures Using Liquid Chromatography/Mass Spectrometry (LC/MS)” on page 6-52
- “Genetic Algorithm Search for Features in Mass Spectrometry Data” on page 6-71
- “Batch Processing of Spectra Using Sequential and Parallel Computing” on page 6-77

Preprocessing Raw Mass Spectrometry Data

This example shows how to improve the quality of raw mass spectrometry data. In particular, this example illustrates the typical steps for preprocessing protein surface-enhanced laser desorption/ionization-time of flight mass spectra (SELDI-TOF).

Loading the Data

Mass spectrometry data can be stored in different formats. If the data is stored in text files with two columns (the mass/charge (M/Z) ratios and the corresponding intensity values), you can use one of the following MATLAB® I/O functions: `importdata`, `dlmread`, or `textscan`. Alternatively, if the data is stored in JCAMP-DX formatted files, you can use the function `jcampread`. If the data is contained in a spreadsheet of an Excel® workbook, you can use the function `xlsread`. If the data is stored in mzXML formatted files, you can use the function `mzxmlread`, and finally, if the data is stored in SPC formatted files, you can use `tgspcread`.

This example uses spectrograms taken from one of the low-resolution ovarian cancer NCI/FDA data sets from the FDA-NCI Clinical Proteomics Program Databank. These spectra were generated using the WCX2 protein-binding chip, two with manual sample handling and two with a robotic sample dispenser/processor.

```
sample = importdata('mspec01.csv')

sample =
    struct with fields:
        data: [15154x2 double]
        textdata: {'M/Z' 'Intensity'}
        colheaders: {'M/Z' 'Intensity'}
```

The M/Z ratios are in the first column of the `data` field and the ion intensities are in the second.

```
MZ = sample.data(:,1);
Y = sample.data(:,2);
```

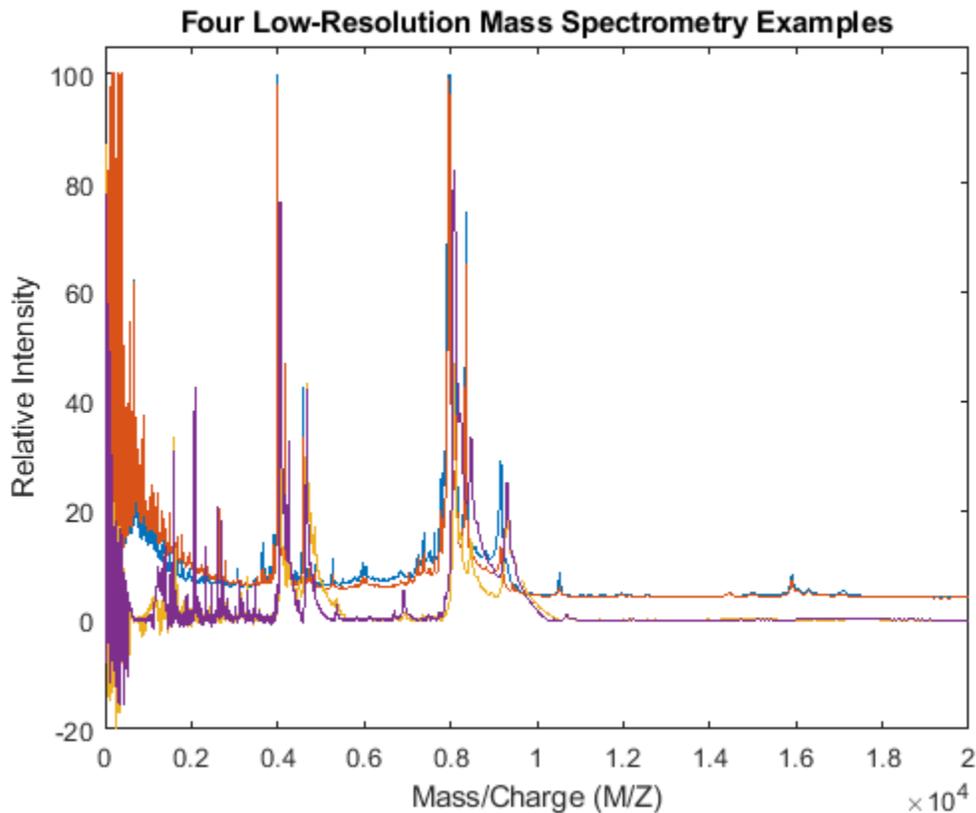
For better manipulation of the data, you can load multiple spectrograms and concatenate them into a single matrix. Use the `dlmread` function to read comma separated value files. Note: This example assumes that the M/Z ratios are the same for the four files. For data sets with different M/Z ratios, use `msresample` to create a uniform M/Z vector.

```
files = {'mspec01.csv', 'mspec02.csv', 'mspec03.csv', 'mspec04.csv'};

for i = 1:4
    Y(:,i) = dlmread(files{i}, ',', 1, 1); % skips the first row (header)
end
```

Use the `plot` command to inspect the loaded spectrograms.

```
plot(MZ, Y)
axis([0 20000 -20 105])
xlabel('Mass/Charge (M/Z)')
ylabel('Relative Intensity')
title('Four Low-Resolution Mass Spectrometry Examples')
```



Resampling the Spectra

Resampling mass spectrometry data has several advantages. It homogenizes the mass/charge (M/Z) vector, allowing you to compare different spectra under the same reference and at the same resolution. In high-resolution data sets, the large size of the files leads to computationally intensive algorithms. However, high-resolution spectra can be redundant. By resampling, you can decimate the signal into a more manageable M/Z vector, preserving the information content of the spectra. The `msresample` function allows you to select a new M/Z vector and also applies an antialias filter that prevents high-frequency noise from folding into lower frequencies.

Load a high-resolution spectrum taken from the high-resolution ovarian cancer NCI/FDA data set. For convenience, the spectrum is included in a MAT-formatted file.

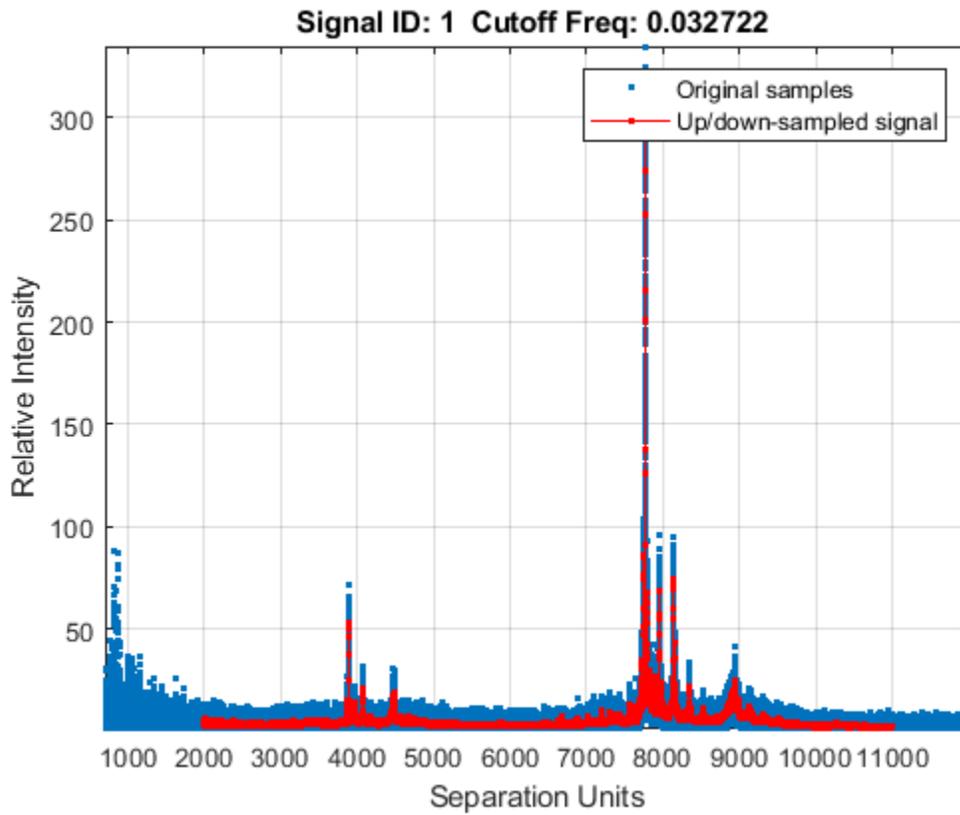
```
load sample_hi_res
numel(MZ_hi_res)
```

```
ans =
```

```
355760
```

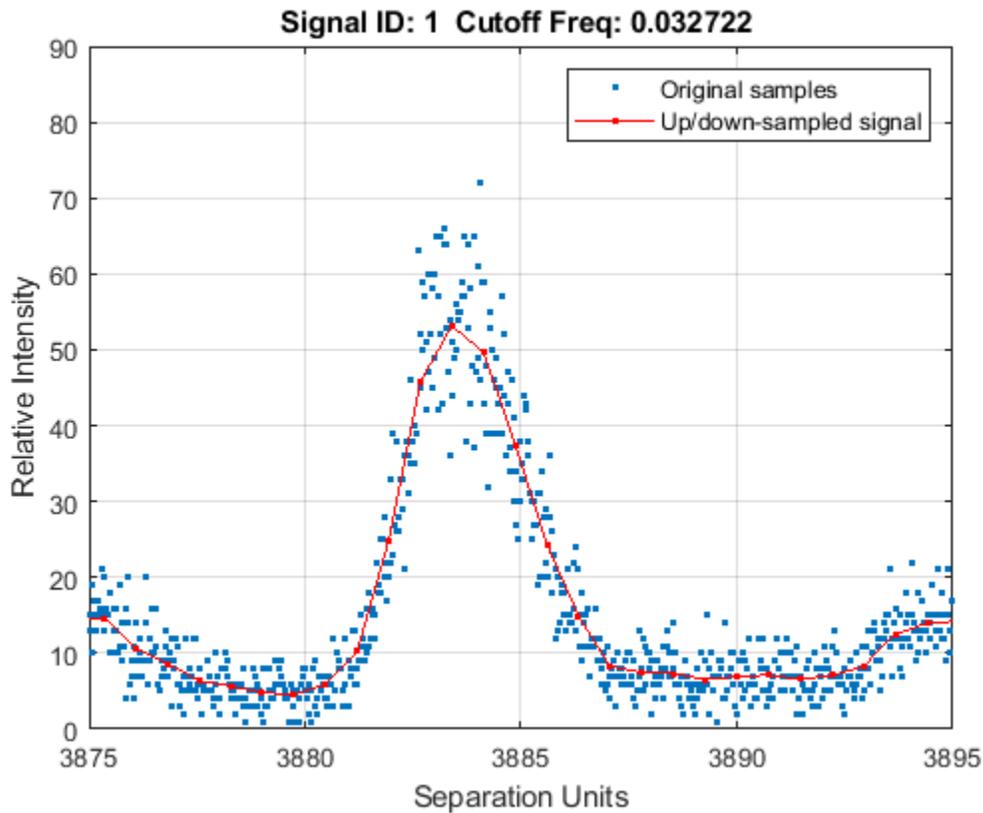
Down-sample the spectra to 10,000 M/Z points between 2,000 and 11,000. Use the `SHOWPLOT` property to create a customized plot that lets you follow and assess the quality of the preprocessing action.

```
[MZ,YH] = msresample(MZ_hi_res,Y_hi_res,10000,'RANGE',[2000 11000],'SHOWPLOT',true);
```



Zooming into a reduced region reveals the detail of the down-sampling procedure.

```
axis([3875 3895 0 90])
```

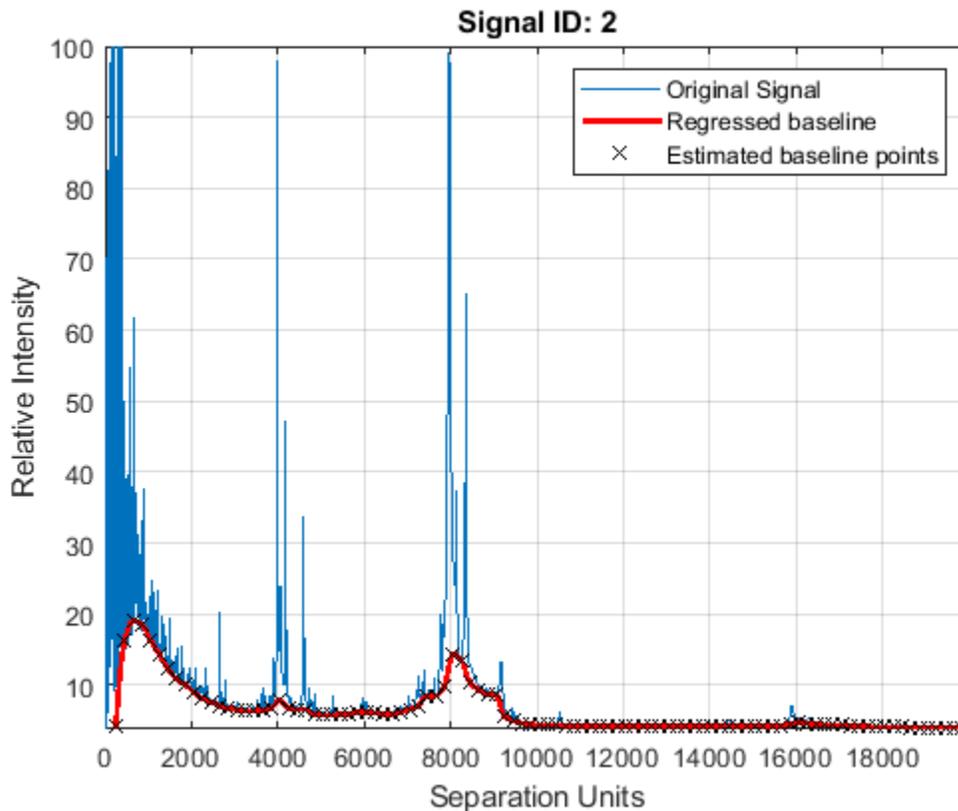


Baseline Correction

Mass spectrometry data usually show a varying baseline caused by the chemical noise in the matrix or by ion overloading. The `msbackadj` function estimates a low-frequency baseline, which is hidden among high-frequency noise and signal peaks. It then subtracts the baseline from the spectrogram.

Adjust the baseline of the set of spectrograms and show only the second one and its estimated background.

```
YB = msbackadj(MZ,Y, 'WINDOWSIZE',500, 'QUANTILE',0.20, 'SHOWPLOT',2);
```



Spectral Alignment of Profiles

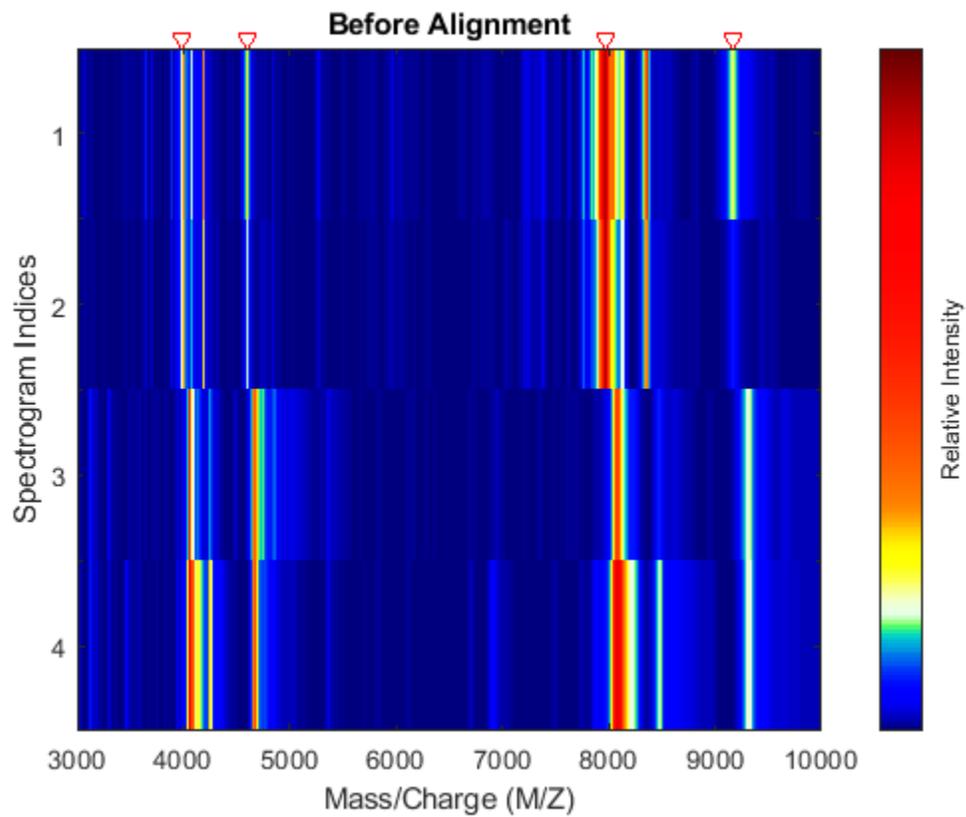
Miscalibration of the mass spectrometer leads to variations of the relationship between the observed M/Z vector and the true time-of-flight of the ions. Therefore, systematic shifts can appear in repeated experiments. When a known profile of peaks is expected in the spectrogram, you can use the function `msalign` to standardize the M/Z values.

To align the spectrograms, provide a set of M/Z values where reference peaks are expected to appear. You can also define a vector with relative weights that is used by the aligning algorithm to emphasize peaks with small area.

```
P = [3991.4 4598 7964 9160]; % M/Z location of reference peaks
W = [60 100 60 100];       % Weight for reference peaks
```

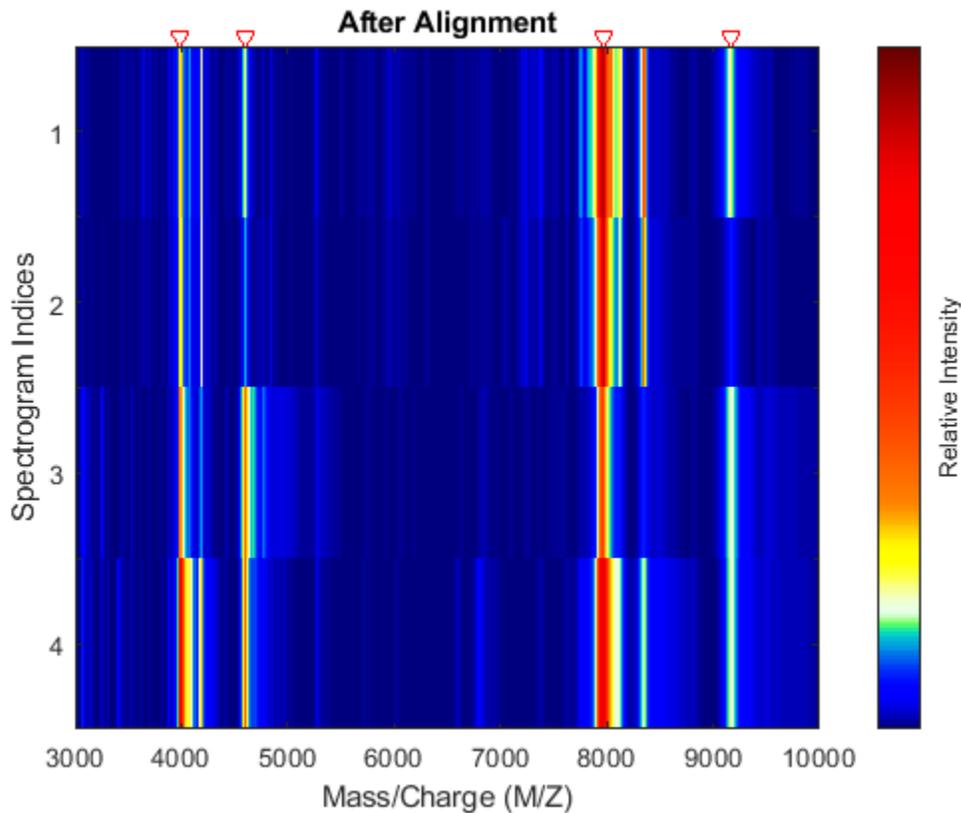
Display a heat map to observe the alignment of the spectra before and after applying the alignment algorithm.

```
msheatmap(MZ,YB,'MARKERS',P,'RANGE',[3000 10000])
title('Before Alignment')
```



Align the set of spectrograms to the reference peaks given.

```
YA = msalign(MZ,YB,P,'WEIGHTS',W);  
msheatmap(MZ,YA,'MARKERS',P,'RANGE',[3000 10000])  
title('After Alignment')
```

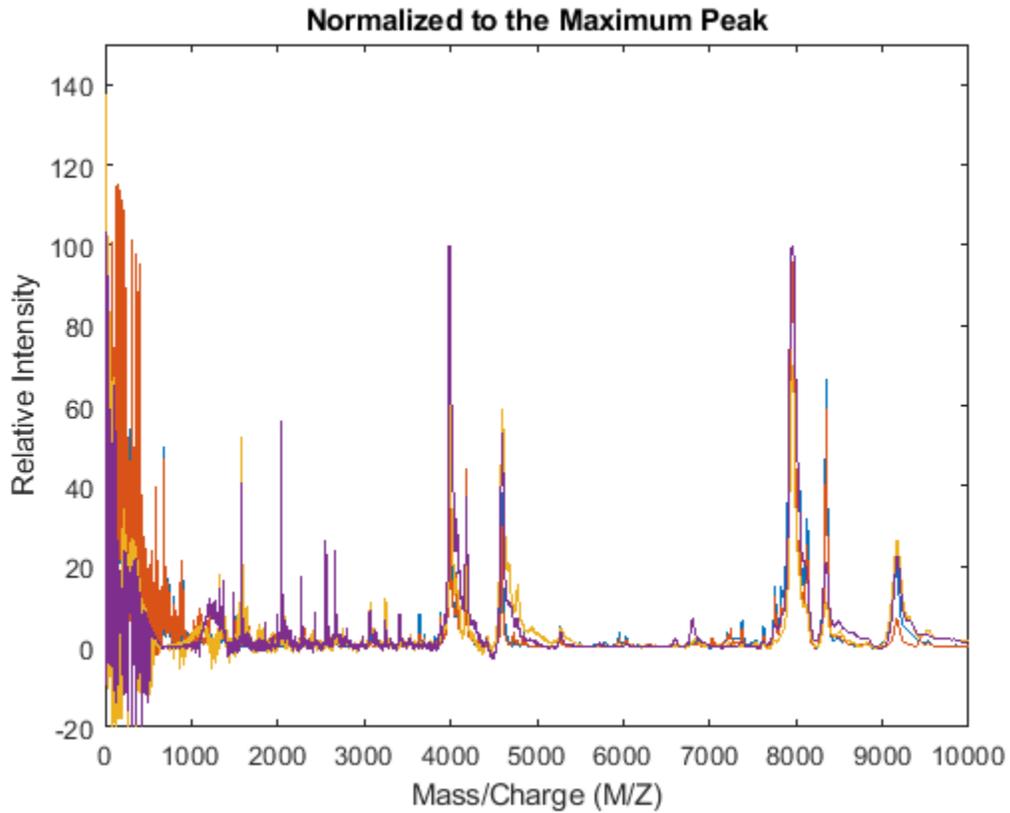


Normalization

In repeated experiments, it is common to find systematic differences in the total amount of desorbed and ionized proteins. The `msnorm` function implements several variations of typical normalization (or standardization) methods.

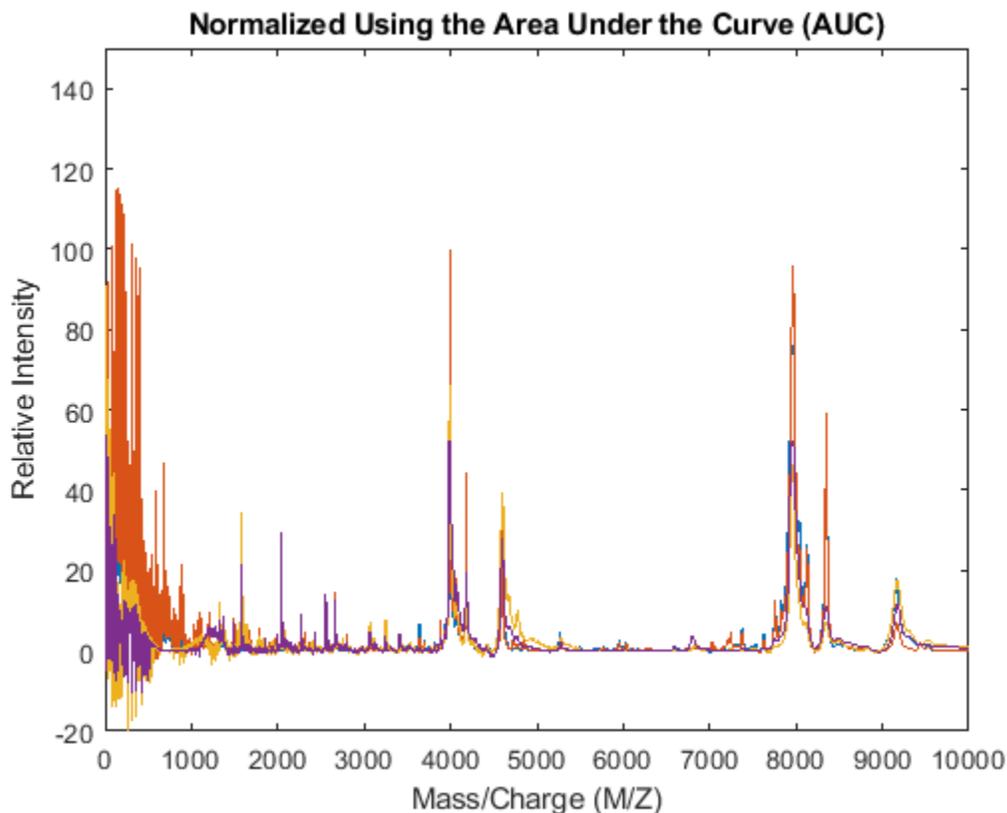
For example, one of many methods to standardize the values of the spectrograms is to rescale the maximum intensity of every signal to a specific value, for instance 100. It is also possible to ignore problematic regions; for example, in serum samples you might want to ignore the low-mass region ($M/Z < 1000$ Da.).

```
YN1 = msnorm(MZ,YA,'QUANTILE',1,'LIMITS',[1000 inf],'MAX',100);
figure
plot(MZ,YN1)
axis([0 10000 -20 150])
xlabel('Mass/Charge (M/Z)')
ylabel('Relative Intensity')
title('Normalized to the Maximum Peak')
```



The `msnorm` function can also standardize by using the area under the curve (AUC) and then rescale the spectrograms to have relative intensities below 100.

```
YN2 = msnorm(MZ,YA,'LIMITS',[1000 inf],'MAX',100);  
figure  
plot(MZ,YN2)  
axis([0 10000 -20 150])  
xlabel('Mass/Charge (M/Z)')  
ylabel('Relative Intensity')  
title('Normalized Using the Area Under the Curve (AUC)')
```

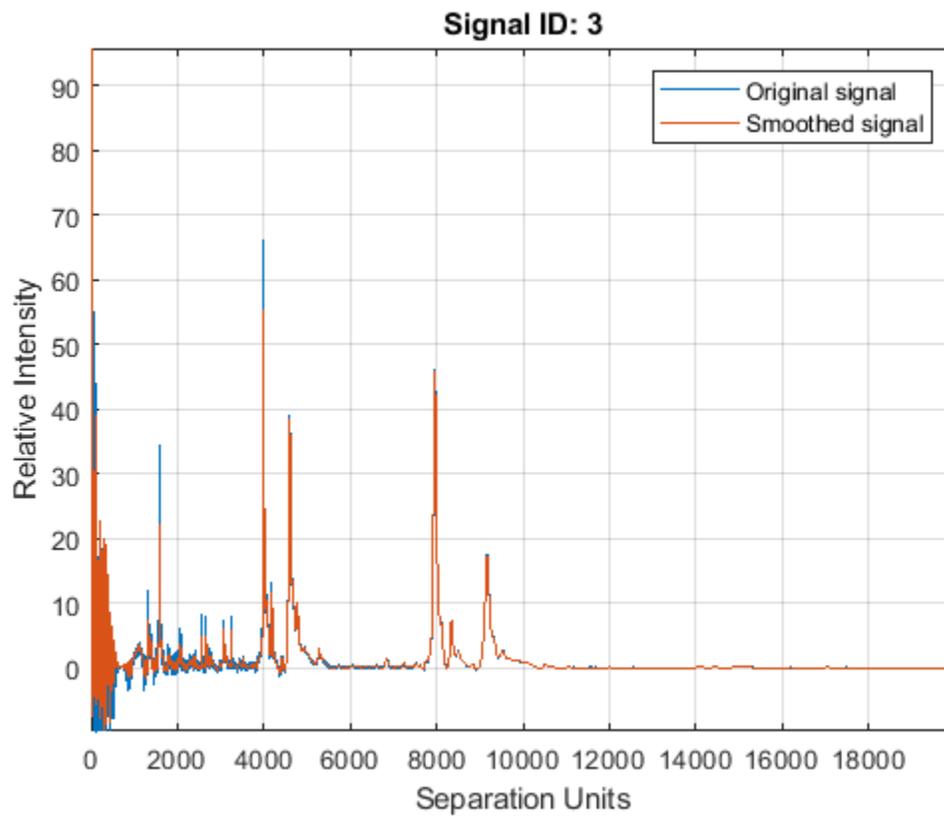


Peak Preserving Noise Reduction

Standardized spectra usually contain a mixture of noise and signal. Some applications require denoising of the spectrograms to improve the validity and precision of the observed mass/charge values of the peaks in the spectra. For the same reason, denoising also improves further peak detection algorithms. However, it is important to preserve the sharpness (or high-frequency components) of the peak as much as possible. For this, you can use Lowess smoothing (`mslowess`) and polynomial filters (`mssgolay`).

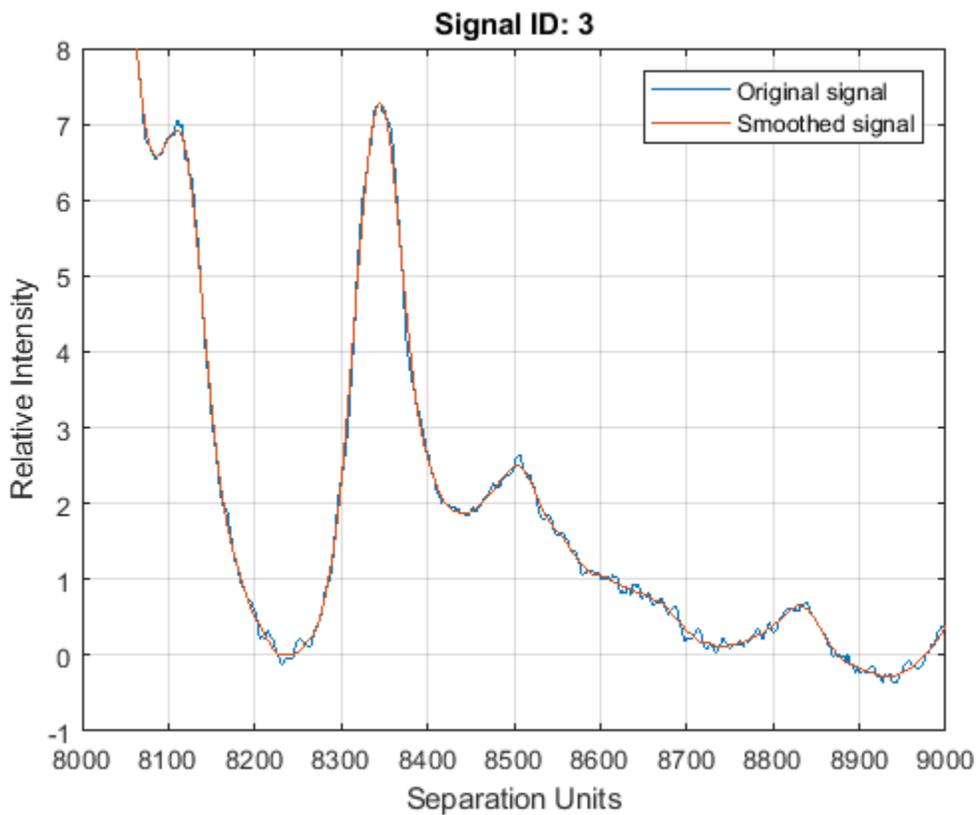
Smooth the spectrograms with a polynomial filter of second order.

```
YS = mssgolay(MZ,YN2, 'SPAN',35, 'SHOWPLOT',3);
```



Zooming into a reduced region reveals the detail of the smoothing algorithm.

```
axis([8000 9000 -1 8])
```



Peak Finding with Wavelets Denoising

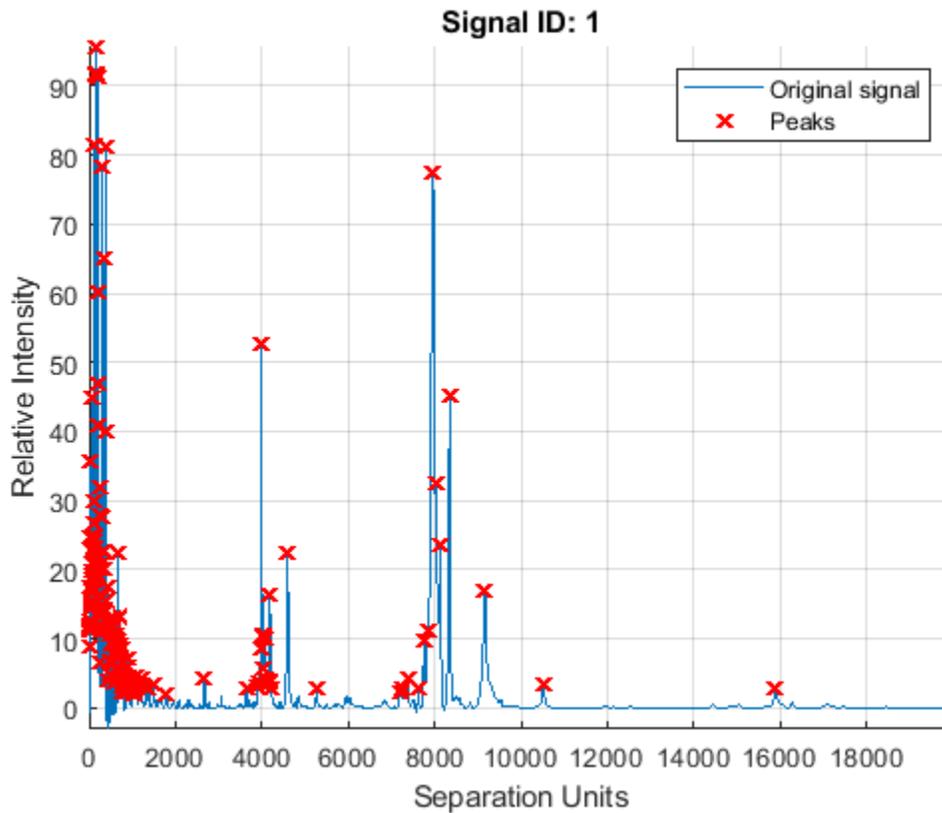
A simple approach to find putative peaks is to look at the first derivative of the smoothed signal, then filter some of these locations to avoid small ion-intensity peaks.

```
P1 = mspeaks(MZ,YS, 'DENOISING', false, 'HEIGHTFILTER', 2, 'SHOWPLOT', 1)
```

P1 =

4×1 cell array

```
{164×2 double}
{171×2 double}
{169×2 double}
{147×2 double}
```



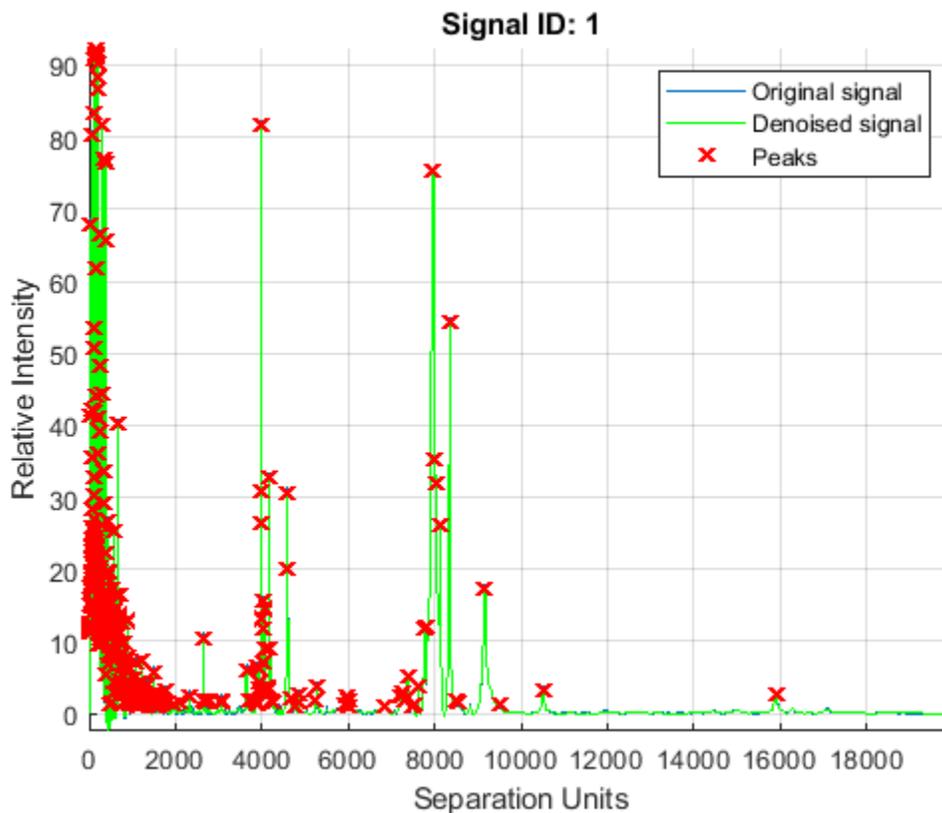
The `mspeaks` function can also estimate the noise using wavelets denoising. This method is generally more robust, because peak detection can be achieved directly over noisy spectra. The algorithm will adapt to varying noise conditions of the signal, and peaks can be resolved even if low resolution or oversegmentation exists.

```
P2 = mspeaks(MZ,YN2, 'BASE',12, 'MULTIPLIER',10, 'HEIGHTFILTER',1, 'SHOWPLOT',1)
```

```
P2 =
```

```
4×1 cell array
```

```
{322×2 double}  
{370×2 double}  
{324×2 double}  
{295×2 double}
```



Eliminate extra peaks in the low-mass region

```
P3 = cellfun( @(x) x(x(:,1)>1500,:),P2,'UNIFORM',false)
```

P3 =

4×1 cell array

```
{81×2 double}
{93×2 double}
{57×2 double}
{53×2 double}
```

Binning: Peak Coalescing by Hierarchical Clustering

Peaks corresponding to similar compounds may still be reported with slight mass/charge differences or drifts. Assuming that the four spectrograms correspond to comparable biological/chemical samples, it might be useful to compare peaks from different spectra, which requires peak binning (a.k.a. peak coalescing). The crucial task in data binning is to create a common mass/charge reference vector (or bins). Ideally, bins should collect one peak from each signal and should avoid collecting multiple relevant peaks from the same signal into the same bin.

This example uses hierarchical clustering to calculate a common mass/charge reference vector. The approach is sufficient when using low-resolution spectra; however, for high-resolution spectra or for

data sets with many spectrograms, the function `malign` provides other scalable methods to estimate a common mass/charge reference and perform data binning.

Put all the peaks into a single array and construct a vector with the spectrogram index for each peak.

```
allPeaks = cell2mat(P3);
numPeaks = cellfun(@length, P3);
Sidx = accumarray(cumsum(numPeaks), 1);
Sidx = cumsum(Sidx) - Sidx;
```

Create a custom distance function that penalizes clusters containing peaks from the same spectrogram, then perform hierarchical clustering.

```
distfun = @(x,y) (x(:,1)-y(:,1)).^2 + (x(:,2)~=y(:,2))*10^6
```

```
tree = linkage(pdist([allPeaks(:,1),Sidx],distfun));
clusters = cluster(tree,'CUTOFF',75,'CRITERION','Distance');
```

```
distfun =
```

```
function_handle with value:
```

```
@(x,y)(x(:,1)-y(:,1)).^2+(x(:,2)~=y(:,2))*10^6
```

The common mass/charge reference vector (CMZ) is found by calculating the centroids for each cluster.

```
CMZ = accumarray(clusters,prod(allPeaks,2))./accumarray(clusters,allPeaks(:,2));
```

Similarly, the maximum peak intensity of every cluster is also computed.

```
PR = accumarray(clusters,allPeaks(:,2),[],@max);
```

```
[CMZ,h] = sort(CMZ);
```

```
PR = PR(h);
```

```
figure
```

```
hold on
```

```
box on
```

```
plot([CMZ CMZ],[-10 100],'-k')
```

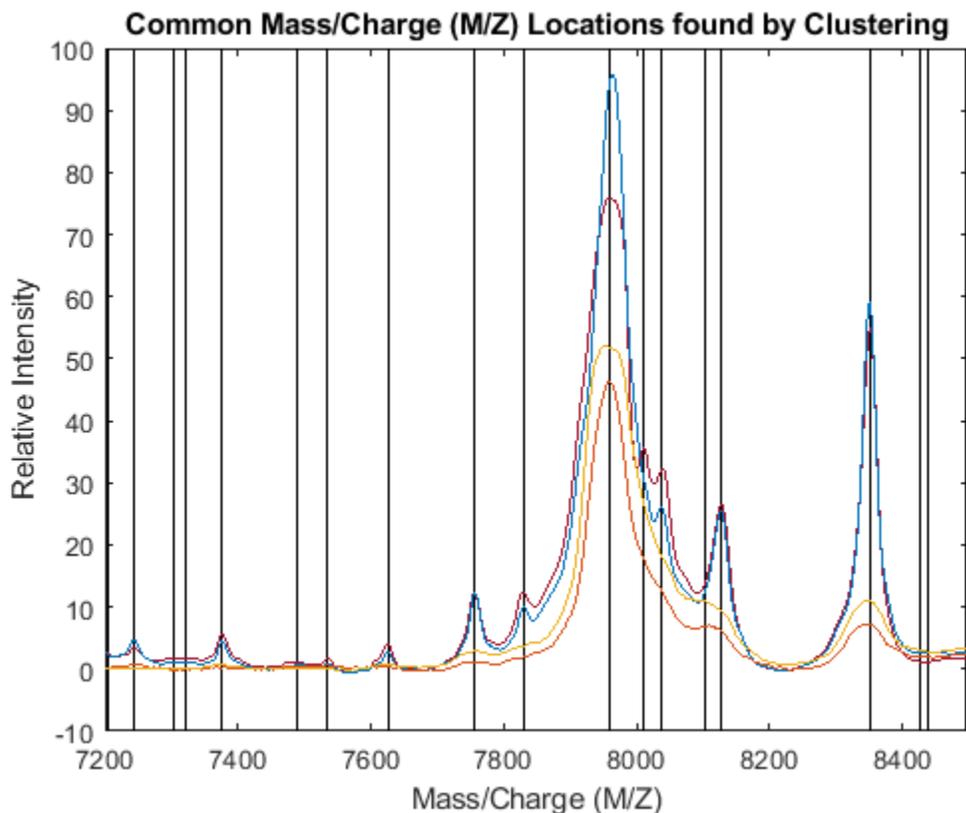
```
plot(MZ,YN2)
```

```
axis([7200 8500 -10 100])
```

```
xlabel('Mass/Charge (M/Z)')
```

```
ylabel('Relative Intensity')
```

```
title('Common Mass/Charge (M/Z) Locations found by Clustering')
```



Dynamic Programming Binning

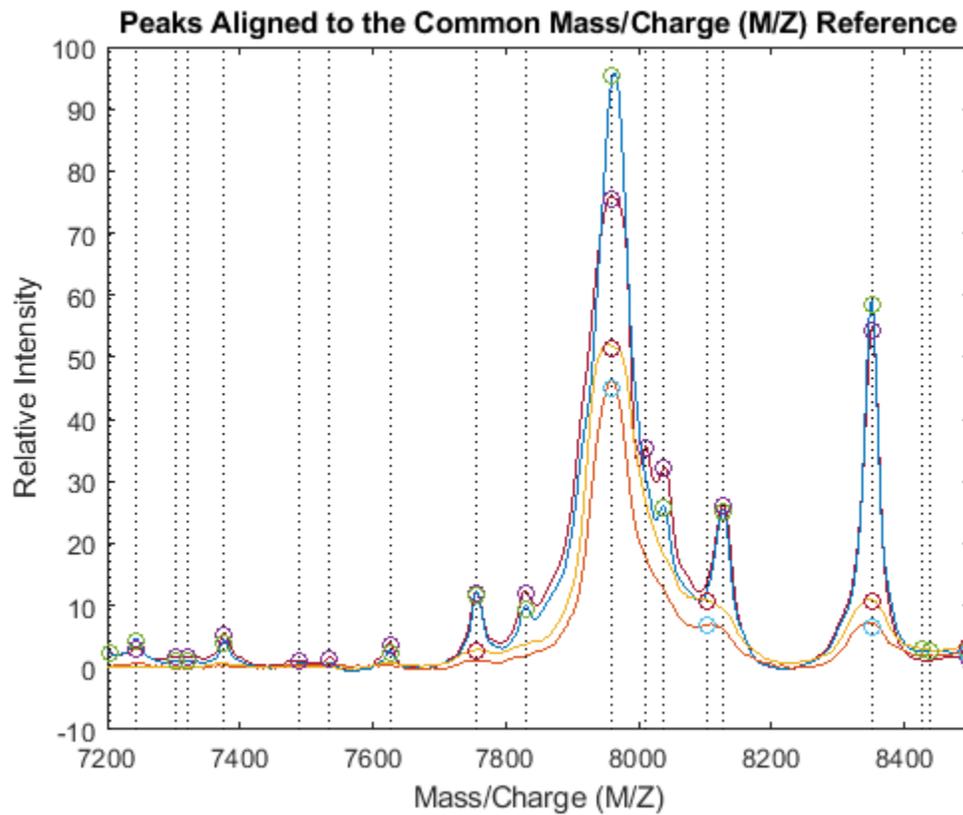
The `samplealign` function allows you to use a dynamic programming algorithm to assign the observed peaks in each spectrogram to the common mass/charge reference vector (CMZ).

When simpler binning approaches are used, such as rounding the mass/charge values or using nearest neighbor quantization to the CMZ vector, the same peak from different spectra may be assigned to different bins due to the small drifts that still exist. To circumvent this problem, the bin size can be increased with the sacrifice of mass spectrometry peak resolution. By using dynamic programming binning, you preserve the resolution while minimizing the problem of assigning similar compounds from different spectrograms to different peak locations.

```
PA = nan(numel(CMZ),4);
for i = 1:4
    [j,k] = samplealign([CMZ PR],P3{i},'BAND',15,'WEIGHTS',[1 .1]);
    PA(j,i) = P3{i}(k,2);
end
```

```
figure
hold on
box on
plot([CMZ CMZ],[-10 100],':k')
plot(MZ,YN2)
plot(CMZ,PA,'o')
axis([7200 8500 -10 100])
xlabel('Mass/Charge (M/Z)')
```

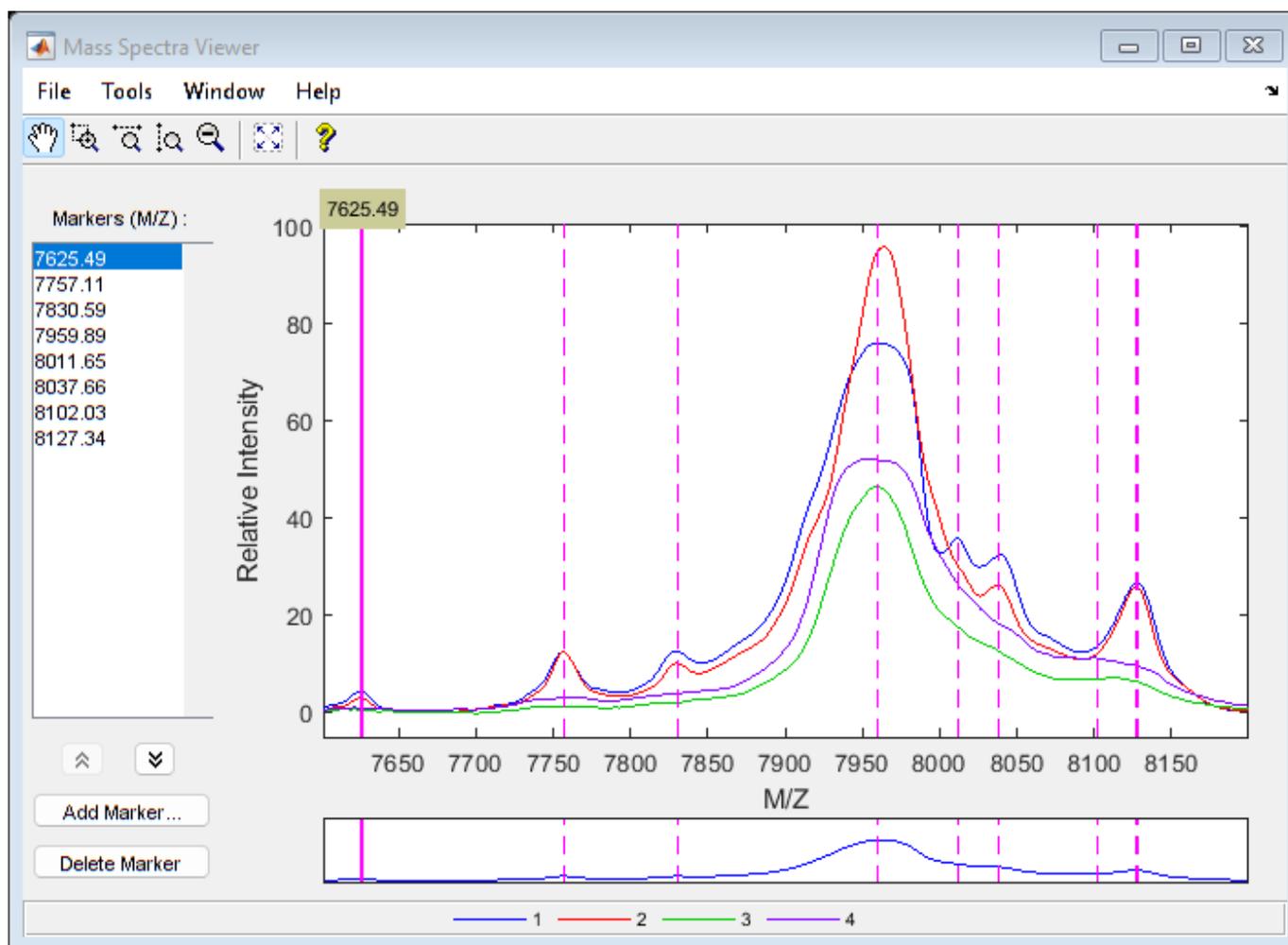
```
ylabel('Relative Intensity')
title('Peaks Aligned to the Common Mass/Charge (M/Z) Reference')
```



Use `msviewer` to inspect the preprocessed spectrograms on a given range (for example, between values 7600 and 8200).

```
r1 = 7600;
r2 = 8200;
range = MZ > r1 & MZ < r2;
rangeMarkers = CMZ(CMZ > r1 & CMZ < r2);

msviewer(MZ(range), YN2(range, :), 'MARKERS', rangeMarkers, 'GROUP', 1:4)
```



See Also

[mssgolay](#) | [msnorm](#) | [msalign](#) | [msheatmap](#) | [msbackadj](#) | [msresample](#) | [mspeaks](#) | [msviewer](#)

Related Examples

- “Batch Processing of Spectra Using Sequential and Parallel Computing” on page 6-77
- “Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling” on page 6-19
- “Identifying Significant Features and Classifying Protein Profiles” on page 6-38

Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling

This example shows how to manipulate, preprocess and visualize data from Liquid Chromatography coupled with Mass Spectrometry (LC/MS). These large and high dimensional data sets are extensively utilized in proteomics and metabolomics research. Visualizing complex peptide or metabolite mixtures provides an intuitive method to evaluate the sample quality. In addition, methodical correction and preprocessing can lead to automated high throughput analysis of samples allowing accurate identification of significant metabolites and specific peptide features in a biological sample.

Introduction

In a typical hyphenated mass spectrometry experiment, proteins and metabolites are harvested from cells, tissues, or body fluids, dissolved and denatured in solution, and enzymatically digested into mixtures. These mixtures are then separated either by High Performance Liquid Chromatography (HPLC), capillary electrophoresis (CE), or gas chromatography (GC) and coupled to a mass-spectrometry identification method, such as Electrospray Ionization Mass Spectrometry (ESI-MS), matrix assisted ionization (MALDI or SELDI TOF-MS), or tandem mass spectrometry (MS/MS).

Open Data Repositories and mzXML File Format

For this example, we use a test sample LC-ESI-MS data set with a seven protein mix. The data in this example (7MIX_STD_110802_1.mzXML) is not distributed with MATLAB®. To complete this example, you must download a similar data set into a local directory or your own repository. You can try other data sets available in other public databases for protein expression data such as the Peptide Atlas at the Institute of Systems Biology [1].

Most of the current mass spectrometers can translate or save the acquisition data using the mzXML schema. This standard is an XML (eXtensible Markup Language)-based common file format developed by the Sashimi project to address the challenges involved in representing data sets from different manufacturers and from different experimental setups into a common and expandable schema. mzXML files used in hyphenated mass spectrometry are usually very large. The MZXMLINFO function allows you to obtain basic information about the dataset without reading it into memory. For example, you can retrieve the number of scans, the range of the retention time, and the number of tandem MS instruments (levels).

```
info = mzxmlinfo('7MIX_STD_110802_1.mzXML','NUMOFLEVELS',true)
```

```
info =
```

```
struct with fields:
```

```
    Filename: '7MIX_STD_110802_1.mzXML'  
    FileModDate: '01-Feb-2013 11:54:30'  
    FileSize: 26789612  
    NumberOfScans: 7161  
    StartTime: 'PT0.00683333S'  
    EndTime: 'PT200.036S'  
    DataProcessingIntensityCutoff: 'N/A'  
    DataProcessingCentroided: 'true'  
    DataProcessingDeisotoped: 'N/A'  
    DataProcessingChargeDeconvoluted: 'N/A'
```

```
DataProcessingSpotIntegration: 'N/A'
      NumberOfMSLevels: 2
```

The MZXMLREAD function reads the XML document into a MATLAB structure. The fields `scan` and `index` are placed at the first level of the output structure for improved access to the spectral data. The remainder of the mzXML document tree is parsed according to the schema specifications. This LC/MS data set contains 7161 scans with two MS levels. For this example you will use only the first level scans. Second level spectra are usually used for peptide/protein identification, and come at a later stage in some types of workflow analyses. MZXMLREAD can filter the desired scans without loading all the dataset into memory:

```
mzXML_struct = mzxmlread('7MIX_STD_110802_1.mzXML','LEVEL',1)
```

```
mzXML_struct =
```

```
struct with fields:
    scan: [2387x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

If you receive any errors related to memory or java heap space during the loading, try increasing your java heap space as described [here](#).

More detailed information pertaining the mass-spectrometer and the experimental conditions are found in the field `msRun`.

```
mzXML_struct.mzXML.msRun
```

```
ans =
```

```
struct with fields:
    scanCount: 7161
    startTime: "PT0.00683333S"
    endTime: "PT200.036S"
    parentFile: [1x1 struct]
    msInstrument: [1x1 struct]
    dataProcessing: [1x1 struct]
```

To facilitate the handling of the data, the MZXML2PEAKS function extracts the list of peaks from each scan into a cell array (`peaks`) and their respective retention time into a column vector (`time`). You can extract the spectra of certain level by setting the `LEVEL` input parameter.

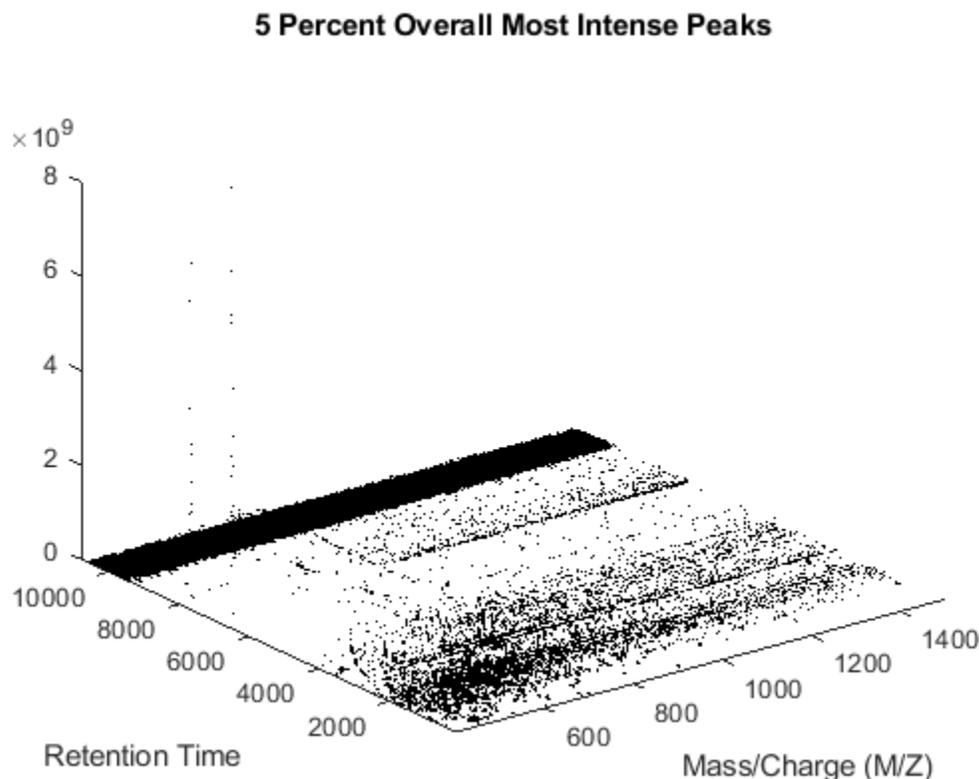
```
[peaks,time] = mzxml2peaks(mzXML_struct);
numScans = numel(peaks)
```

```
numScans =
```

```
2387
```

The MSDOTPLOT function creates an overview display of the most intense peaks in the entire data set. In this case, we visualize only the most intense 5% ion intensity peaks by setting the input parameter QUANTILE to 0.95.

```
h = msdplot(peaks,time,'quantile',.95);
title('5 Percent Overall Most Intense Peaks')
```



You can also filter the peaks individually for each scan using a percentile of the base peak intensity. The base peak is the most intense peak found in each scan [2]. This parameter is given automatically by most of the spectrometers. This operation requires querying into the mxXML structure to obtain the base peak information. Note that you could also find the base peak intensity by iterating the MAX function over the peak list.

```
basePeakInt = [mzXML_struct.scan.basePeakIntensity]';
peaks_fil = cell(numScans,1);
for i = 1:numScans
    h = peaks{i}(:,2) > (basePeakInt(i).*0.75);
    peaks_fil{i} = peaks{i}(h,:);
end
```

```
whos('basePeakInt','level_1','peaks','peaks_fil')
msdplot(peaks_fil,time)
title('Peaks Above (0.75 x Base Peak Intensity) for Each Scan')
```

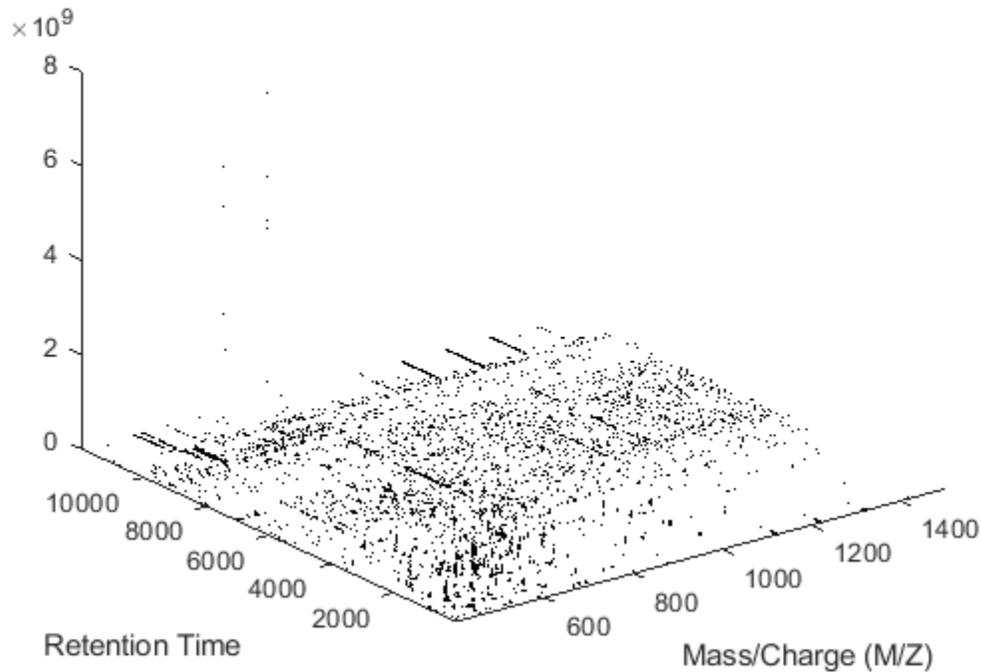
Name	Size	Bytes	Class	Attributes
basePeakInt	2387x1	19096	double	

```

peaks          2387x1          14069992  cell
peaks_fil      2387x1          327760    cell

```

Peaks Above (0.75 x Base Peak Intensity) for Each Scan



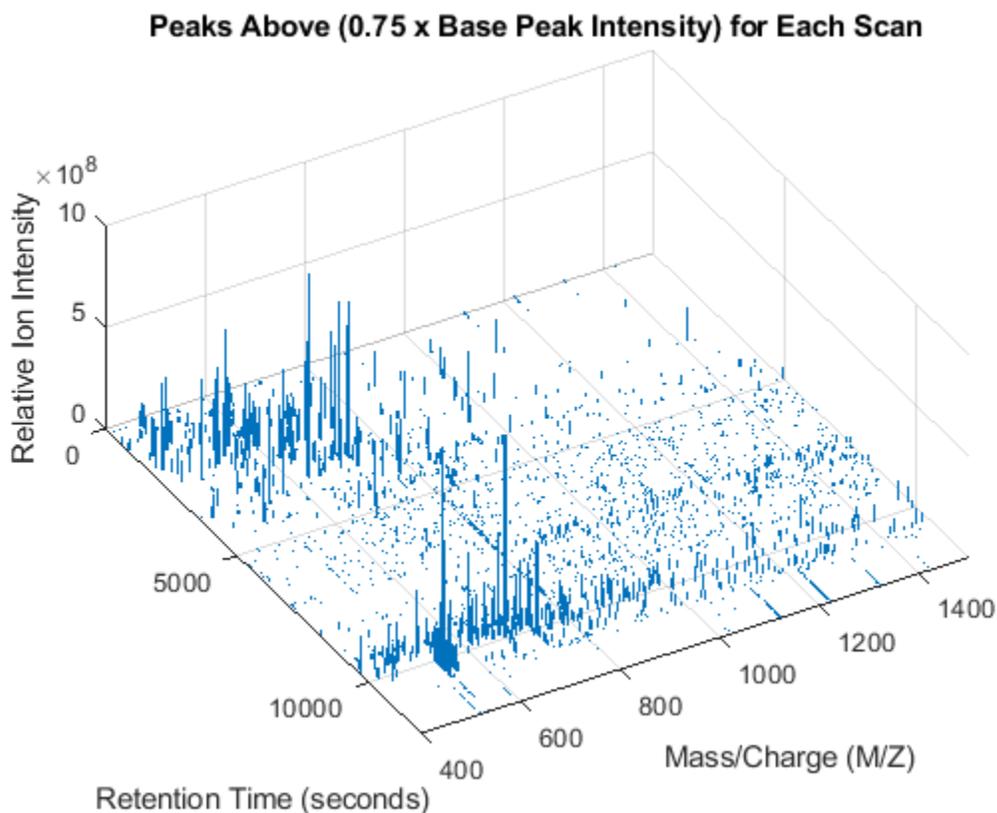
You can customize a 3-D overview of the filtered peaks using the `STEM3` function. The `STEM3` function requires to put the data into three vectors, whose elements form the triplets (the retention time, the mass/charge, and the intensity value) that represent every stem.

```

peaks_3D = cell(numScans,1);
for i = 1:numScans
    peaks_3D{i}(:,[2 3]) = peaks_fil{i};
    peaks_3D{i}(:,1) = time(i);
end
peaks_3D = cell2mat(peaks_3D);

figure
stem3(peaks_3D(:,1),peaks_3D(:,2),peaks_3D(:,3),'marker','none')
axis([0 12000 400 1500 0 1e9])
view(60,60)
xlabel('Retention Time (seconds)')
ylabel('Mass/Charge (M/Z)')
zlabel('Relative Ion Intensity')
title('Peaks Above (0.75 x Base Peak Intensity) for Each Scan')

```

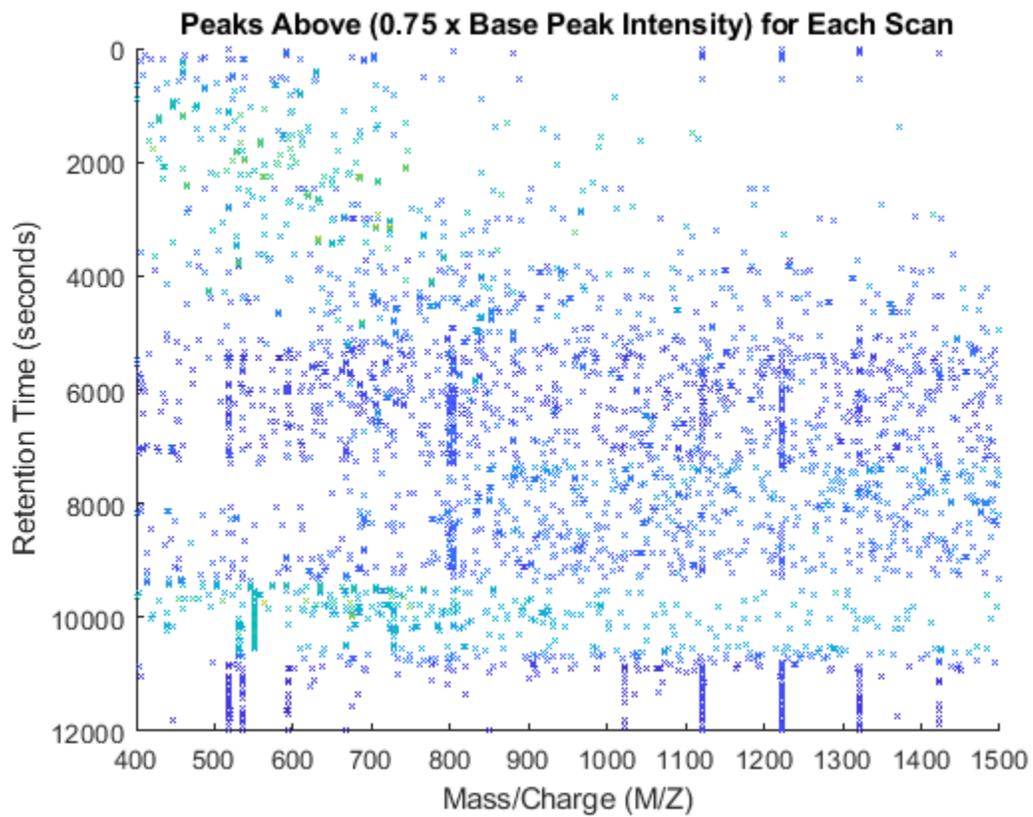


You can plot colored stems using the PATCH function. For every triplet in `peaks_3D`, interleave a new triplet with the intensity value set to zero. Then create a color vector dependent on the intensity of the stem. A logarithmic transformation enhances the dynamic range of the colormap. For the interleaved triplets assign a NaN, so that PATCH function does not draw lines connecting contiguous stems.

```
peaks_patch = sortrows(repmat(peaks_3D,2,1));
peaks_patch(2:2:end,3) = 0;

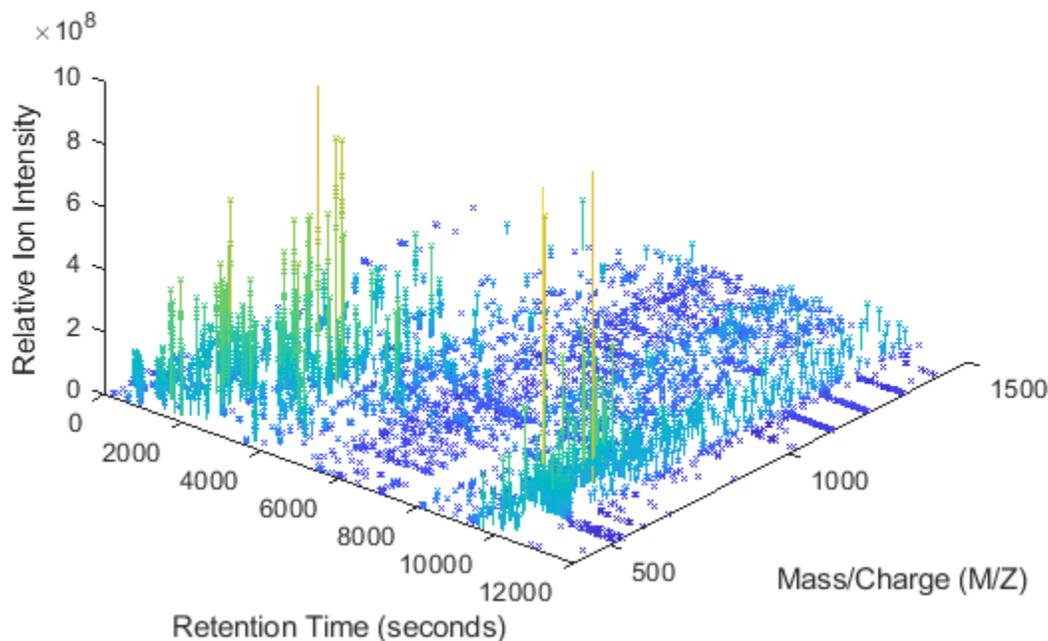
col_vec = log(peaks_patch(:,3));
col_vec(2:2:end) = NaN;

figure
patch(peaks_patch(:,1),peaks_patch(:,2),peaks_patch(:,3),col_vec,...
      'edgeColor','flat','markeredgecolor','flat','Marker','x','MarkerSize',3);
axis([0 12000 400 1500 0 1e9])
view(90,90)
xlabel('Retention Time (seconds)')
ylabel('Mass/Charge (M/Z)')
zlabel('Relative Ion Intensity')
title('Peaks Above (0.75 x Base Peak Intensity) for Each Scan')
```



view(40,40)

Peaks Above (0.75 x Base Peak Intensity) for Each Scan



Creating Heat Maps of LC/MS Data Sets

Common techniques in the industry work with peak information (a.k.a. centroided data) instead of raw signals. This may save memory, but some important details are not visible, especially when it is necessary to inspect samples with complex mixtures. To further analyze this data set, we can create a common grid in the mass/charge dimension. Since not all of the scans have enough information to reconstruct the original signal, we use a **peak preserving** resampling method. By choosing the appropriate parameters for the MSPPRESAMPLE function, you can ensure that the resolution of the spectra is not lost, and that the maximum values of the peaks correlate to the original peak information.

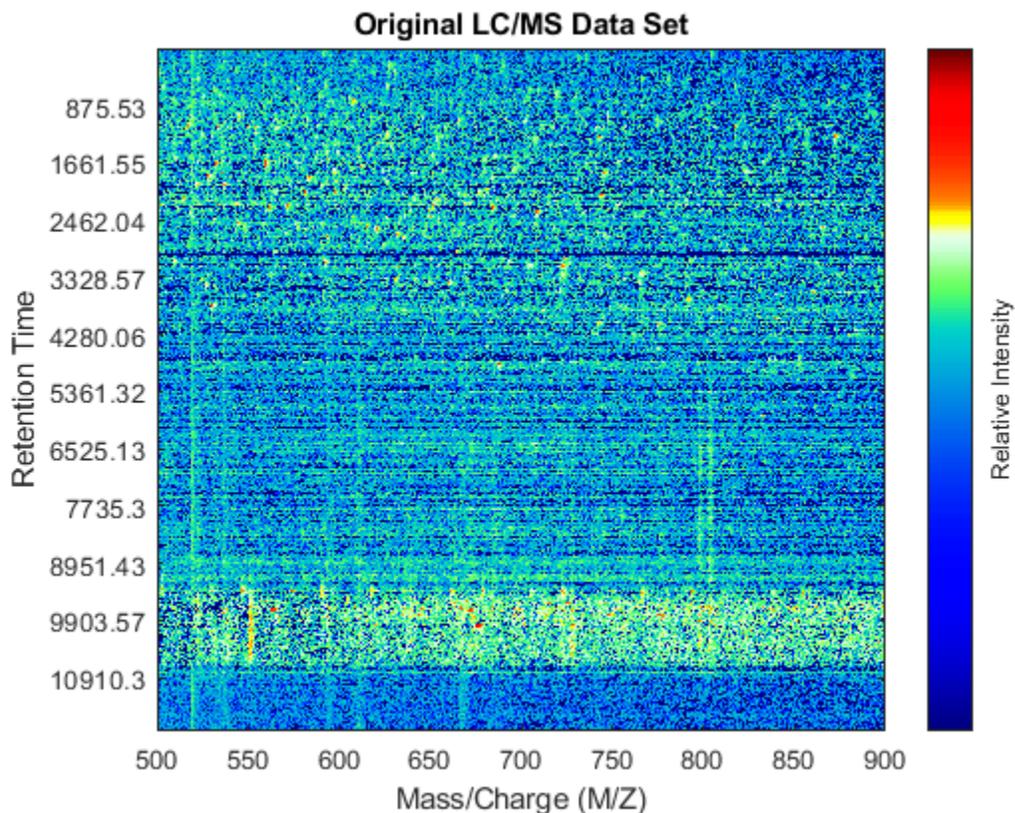
```
[MZ, Y] = mspresample(peaks, 5000);
whos('MZ', 'Y')
```

Name	Size	Bytes	Class	Attributes
MZ	5000x1	20000	single	
Y	5000x2387	47740000	single	

With this matrix of ion intensities, Y, you can create a colored heat map. The MSHEATMAP function automatically adjusts the colorbar utilized to show the statistically significant peaks with hot colors and the noisy peaks with cold colors. The algorithm is based on clustering significant peaks and noisy peaks by estimating a mixture of Gaussians with an Expectation-Maximization approach. Additionally, you can use the MIDPOINT input parameter to select an arbitrary threshold to separate noisy peaks from significant peaks, or you can interactively shift the colormap to hide or unhide peaks. When

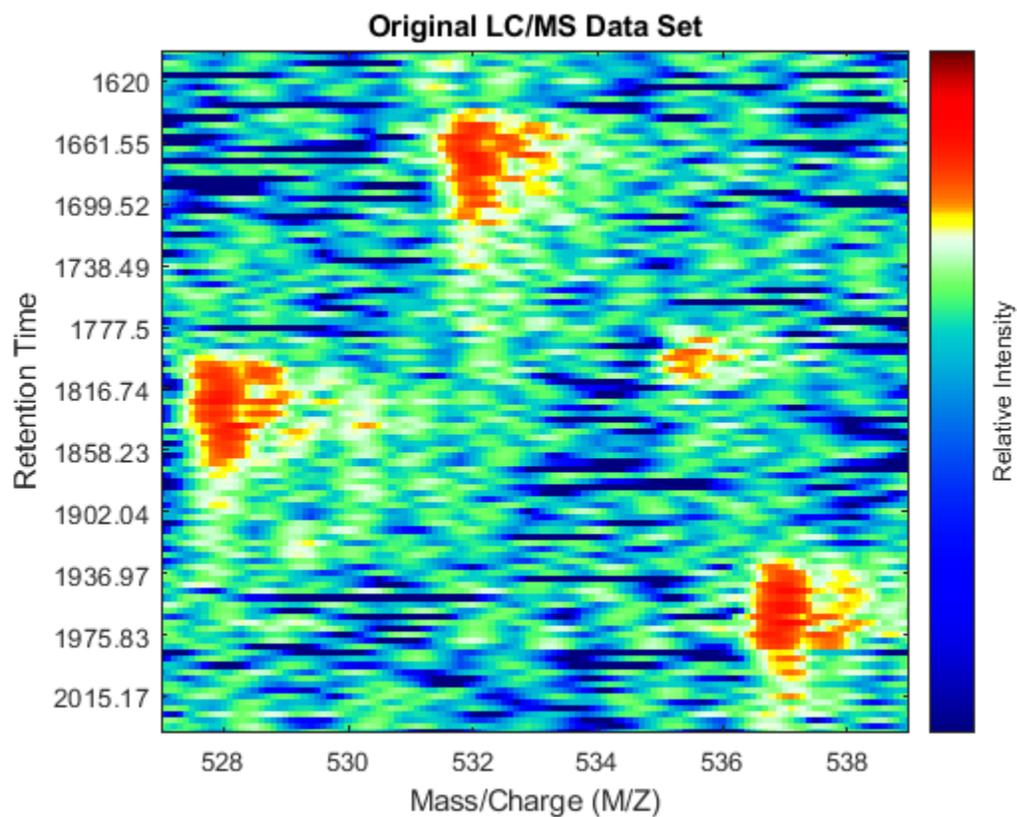
working with heat maps, it is common to display the logarithm of the ion intensities, which enhances the dynamic range of the colormap.

```
fh1 = msheatmap(MZ,time,log(Y),'resolution',.1,'range',[500 900]);  
title('Original LC/MS Data Set')
```



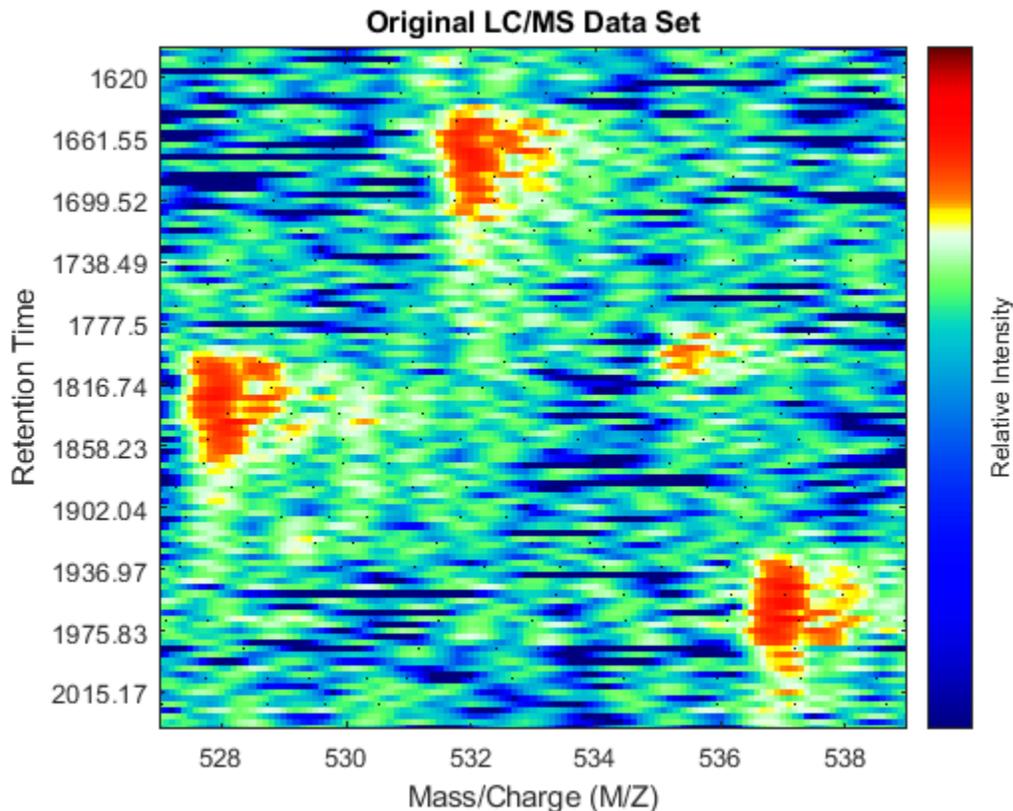
You can zoom to small regions of interest to observe the data, either interactively or programmatically using the `AXIS` function. We observe some regions with high relative ion intensity. These represent peptides in the biological sample.

```
axis([527 539 385 496])
```



You can overlay the original peak information of the LC/MS data set. This lets you evaluate the performance of the peak-preserving resampling operation. You can use the returned handle to hide/unhide the dots.

```
dp1 = msdotplot(peaks,time);
```



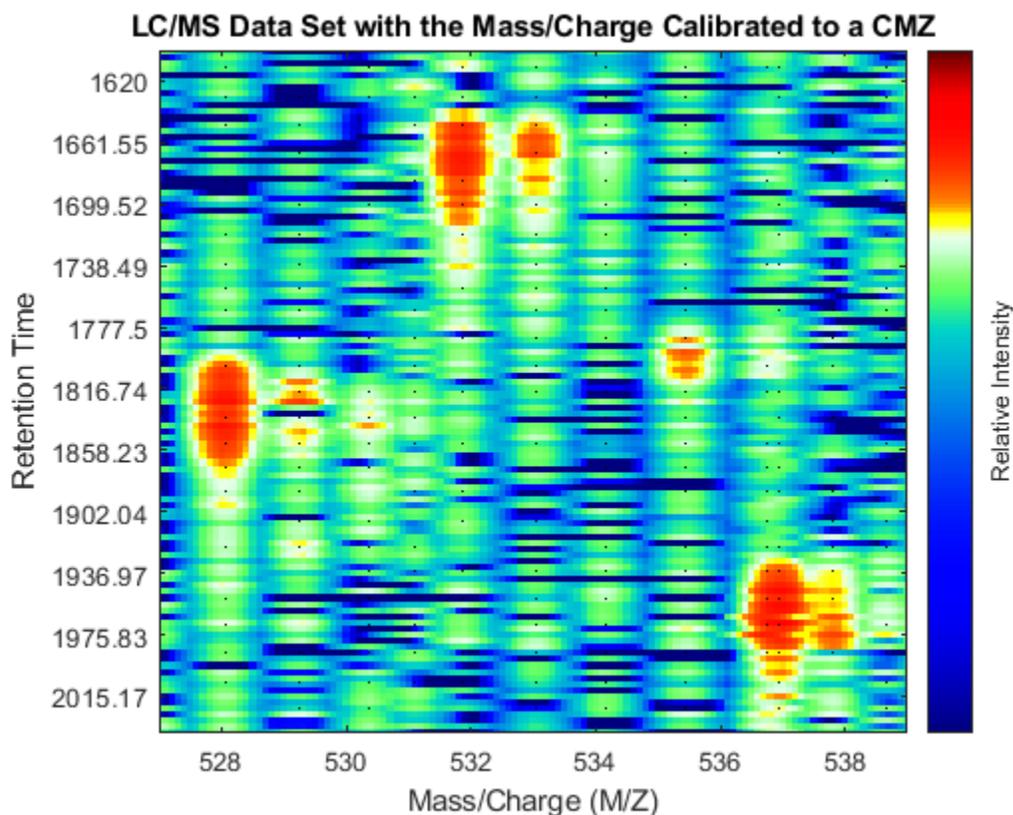
Calibrating the Mass/Charge Location of Peaks to a Common Grid

The two dimensional peaks appear to be noisy or they do not show a compact shape in contiguous spectra. This is a common problem for many mass spectrometers. Random fluctuations of the mass/charge value extracted from peaks of replicate profiles are reported to range from 0.1% to 0.3% [3]. Such variability can be caused by several factors, e.g. poor calibration of the detector, low signal-to-noise ratio, or problems in the peak extraction algorithms. The MSPALIGN function implements advanced data binning algorithms that synchronize all the spectra in a data set to a common mass/charge grid (CMZ). CMZ can be chosen arbitrarily or it can be estimated after analyzing the data [2,4,5]. The peak matching procedure can use either a nearest neighbor rule or a dynamic programming alignment.

```
[CMZ, peaks_CMZ] = mspalign(peaks);
```

Repeat the visualization process with the aligned peaks: perform peak preserving resampling, create a heat map, overlay the aligned peak information, and zoom into the same region of interest as before. When the spectrum is re-calibrated, it is possible to distinguish the isotope patterns of some of the peptides.

```
[MZ_A,Y_A] = msppresample(peaks_CMZ,5000);
fh2 = msheatmap(MZ_A,time,log(Y_A),'resolution',.10,'range',[500 900]);
title('LC/MS Data Set with the Mass/Charge Calibrated to a CMZ')
dp2 = msdotplot(peaks_CMZ,time);
axis([527 539 385 496])
```



Calibrating the Mass/Charge Location of Peaks Locally

MSPALIGN computes a single CMZ for the whole LC/MS data set. This may not be the ideal case for samples with more complex mixtures of peptides and/or metabolites than the data set utilized in this example. In the case of complex mixtures, you can align each spectrum to a local set of spectra that contain only informative peaks (high intensity) with similar retention times, otherwise the calibration process in regions with small peaks (low intensity) can be biased by other peaks that share similar mass/charge values but are at different retention times. To perform a finer calibration, you can employ the SAMPLEALIGN function. This function is a generalization of the Constrained Dynamic Time Warping (CDTW) algorithms commonly utilized in speech processing [6]. In the following for loop, we maintain a buffer with the intensities of the previous aligned spectra (LAI). The ion intensities of the spectra are scaled with the anonymous function SF (inside SAMPLEALIGN) to reduce the distance between large peaks. SAMPLEALIGN reduces the overall distance of all matched points and introduces gaps as necessary. In this case we use a finer MZ vector (FMZ), such that we preserve the correct value of the mass/charge of the peaks as much as possible. Note: this may take some time, as the CDTW algorithm is executed 2,387 times.

```
SF = @(x) 1-exp(-x./5e7); % scaling function
DF = @(R,S) sqrt((SF(R(:,2))-SF(S(:,2))).^2 + (R(:,1)-S(:,1)).^2);

FMZ = (500:0.15:900)'; % setup a finer MZ vector
LAI = zeros(size(FMZ)); % init buffer for the last alignment intensities

peaks_FMZ = cell(numScans,1);
for i = 1:numScans
    % show progress
```

```

if ~rem(i,250)
    fprintf(' %d...',i);
end
% align peaks in current scan to LAI
[k,j] = samplealign([FMZ,LAI],double(peaks{i}),'band',1.5,'gap',[0,2],'dist',DF);
% updating the LAI buffer
LAI = LAI*.25;
LAI(k) = LAI(k) + peaks{i}(j,2);
% save the alignment
peaks_FMZ{i} = [FMZ(k) peaks{i}(j,2)];
end

```

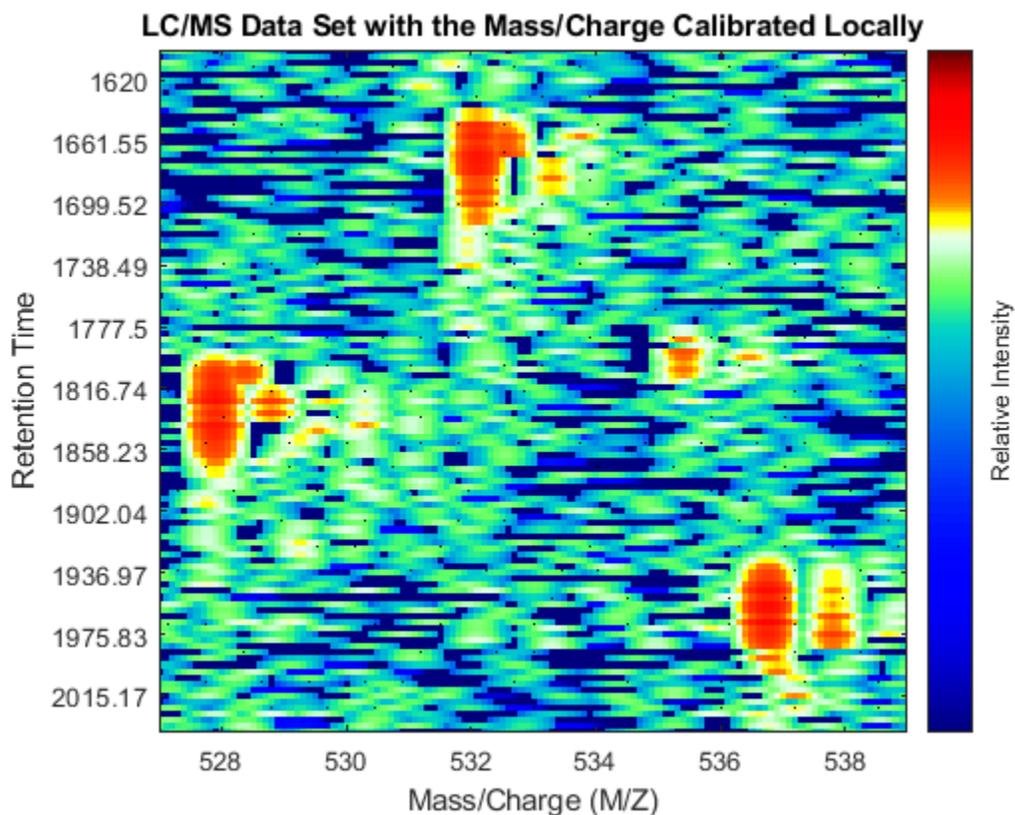
250... 500... 750... 1000... 1250... 1500... 1750... 2000... 2250...

Repeat the visualization process and zoom to the region of interest.

```

[MZ_B,Y_B] = msppresample(peaks_FMZ,4000);
fh3 = msheatmap(MZ_B,time,log(Y_B),'resolution',.10,'range',[500 900]);
title('LC/MS Data Set with the Mass/Charge Calibrated Locally')
dp3 = msdotplot(peaks_FMZ,time);
axis([527 539 385 496])

```



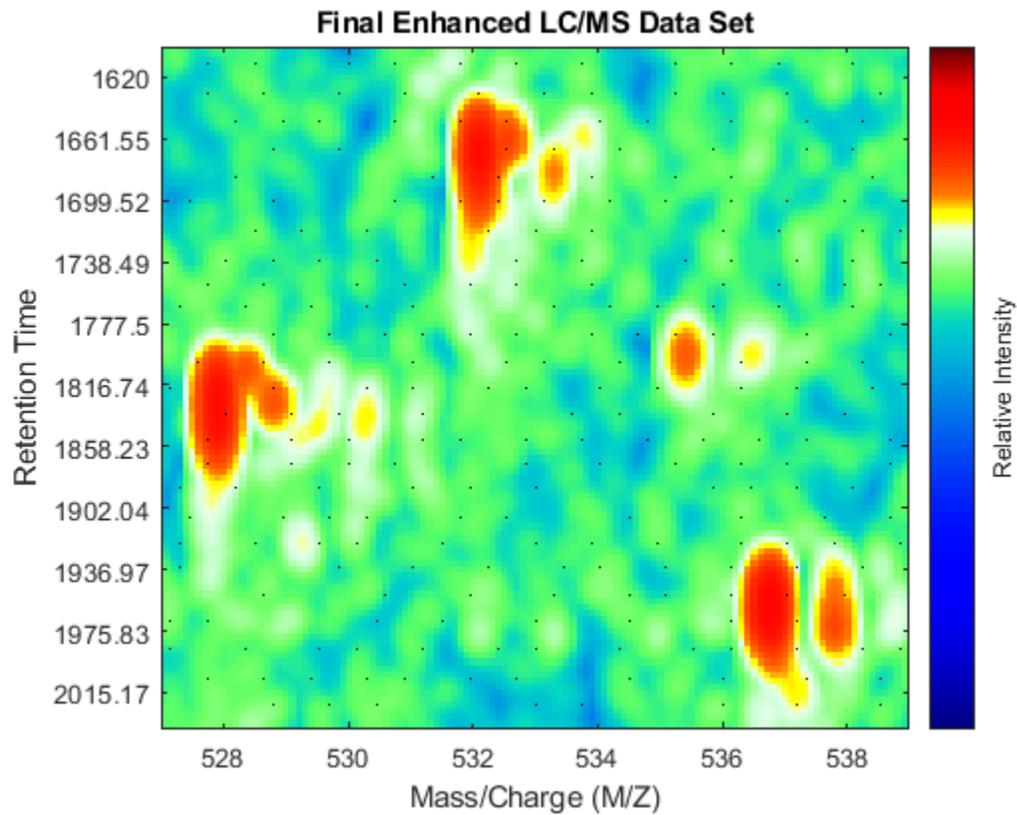
As a final step to improve the image, you can apply a Gaussian filter in the chromatographic direction to smooth the whole data set.

```

Gpulse = exp(-.1*(-10:10).^2)./sum(exp(-.1*(-10:10).^2));
YF = convn(Y_B,Gpulse,'same');
fh4 = msheatmap(MZ_B,time,log(YF),'resolution',.10,'limits',[500 900]);

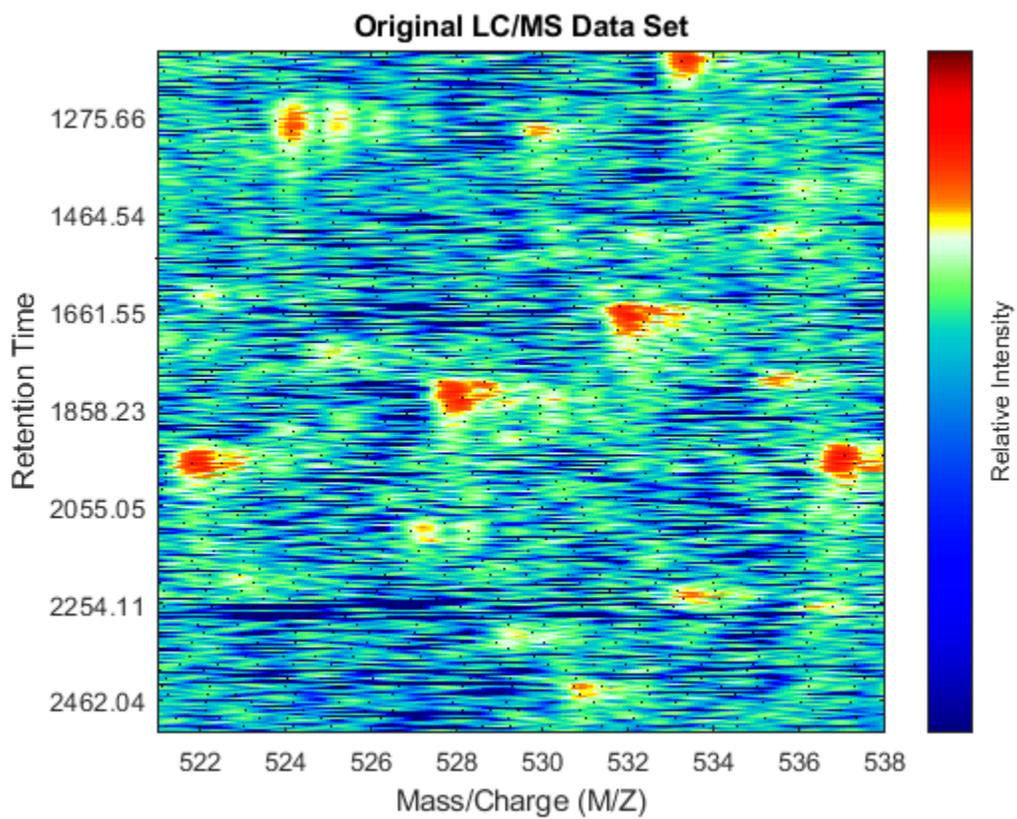
```

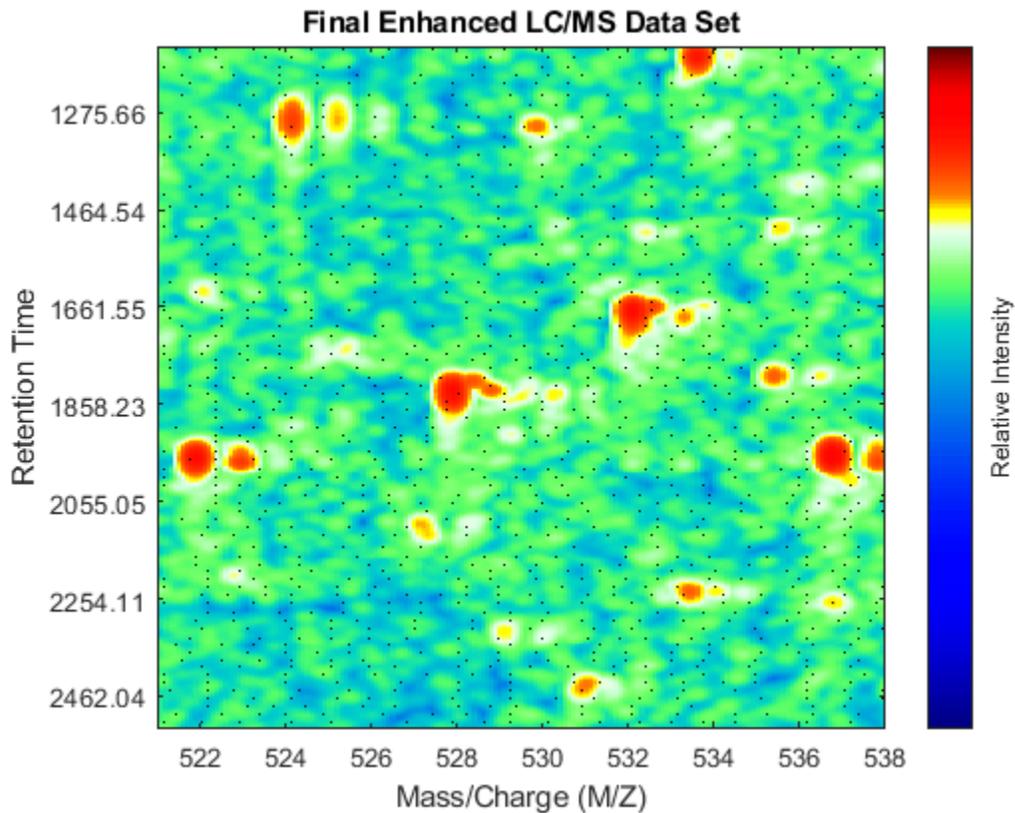
```
title('Final Enhanced LC/MS Data Set')  
dp4 = msdotplot(peaks_FMZ,time);  
axis([527 539 385 496])
```



You can link the axes of two heat maps, to interactively or programmatically compare regions between two data sets. In this case we compare the original and the final enhanced LC/MS matrices.

```
linkaxes(findobj([fh1 fh4], 'Tag', 'MSHeatMap'))  
axis([521 538 266 617])
```





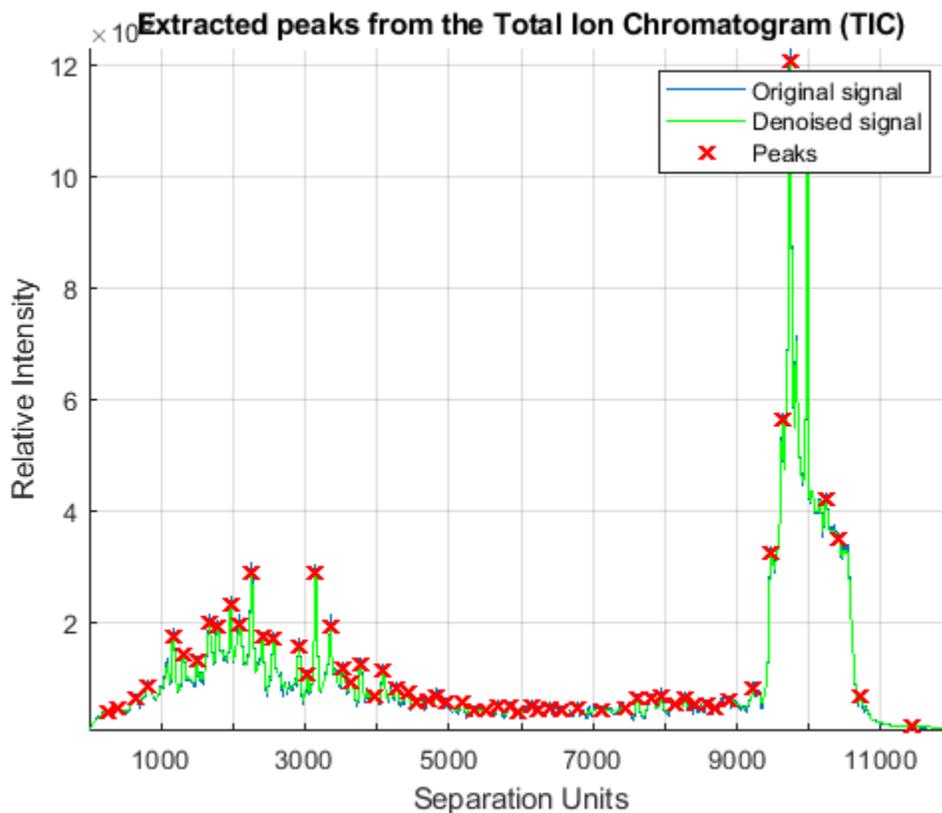
Extracting Spectra Using the Total Ion Chromatogram

Once the LC/MS data set is smoothed and resampled into a regular grid, it is possible to extract the most informative spectra by looking at the local maxima of the Total Ion Chromatogram (TIC). The TIC is straightforwardly computed by summing the rows of YF. Then, use the MSPEAKS function to find the retention time values for extracting selected subsets of spectra.

```
TIC = mean(YF);
pt = mspeaks(time,TIC,'multiplier',10,'overseg',100,'showplot',true);
title('Extracted peaks from the Total Ion Chromatogram (TIC)')
pt(pt(:,1)>4000,:) = []; % remove spectra above 4000 seconds
numPeaks = size(pt,1)
```

numPeaks =

22



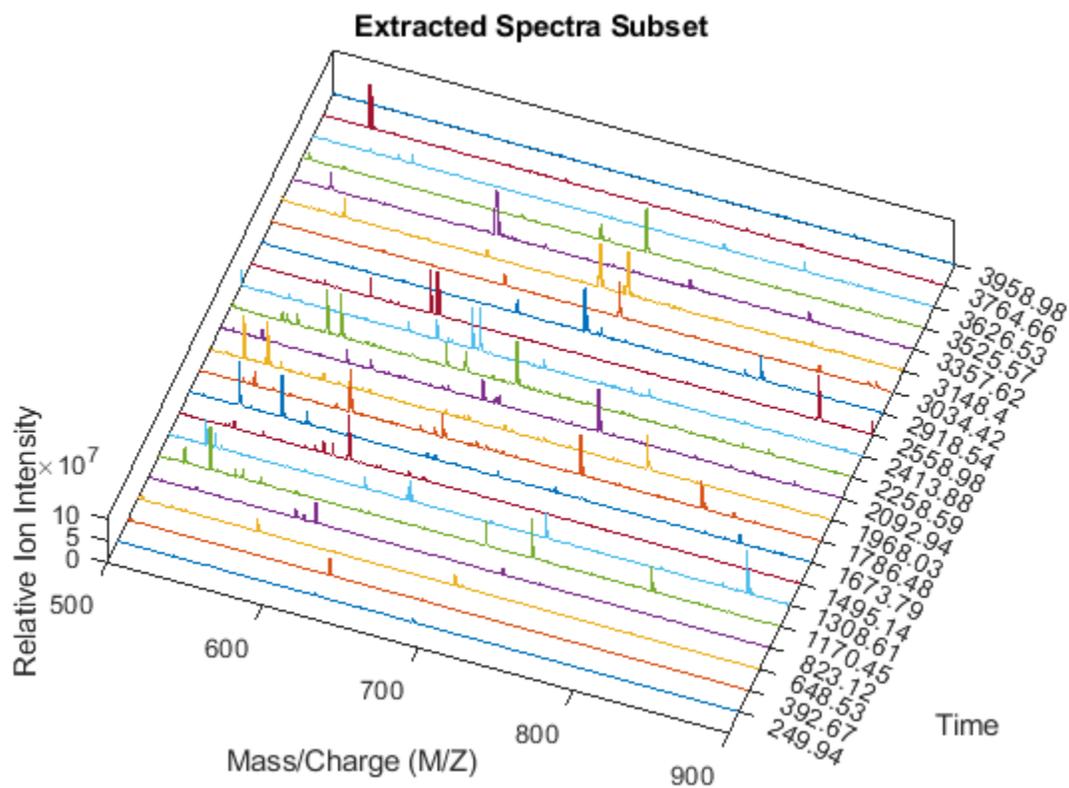
Create a 3-D plot of the selected spectra.

```
xRows = samplealign(time,pt(:,1),'width',1); % finds the time index for every peak
xSpec = YF(:,xRows); % gets the signals to plot
```

```
figure;
hold on
box on
plot3(repmat(MZ_B,1,numPeaks),repmat(1:numPeaks,numel(MZ_B),1),xSpec)
view(20,85)
```

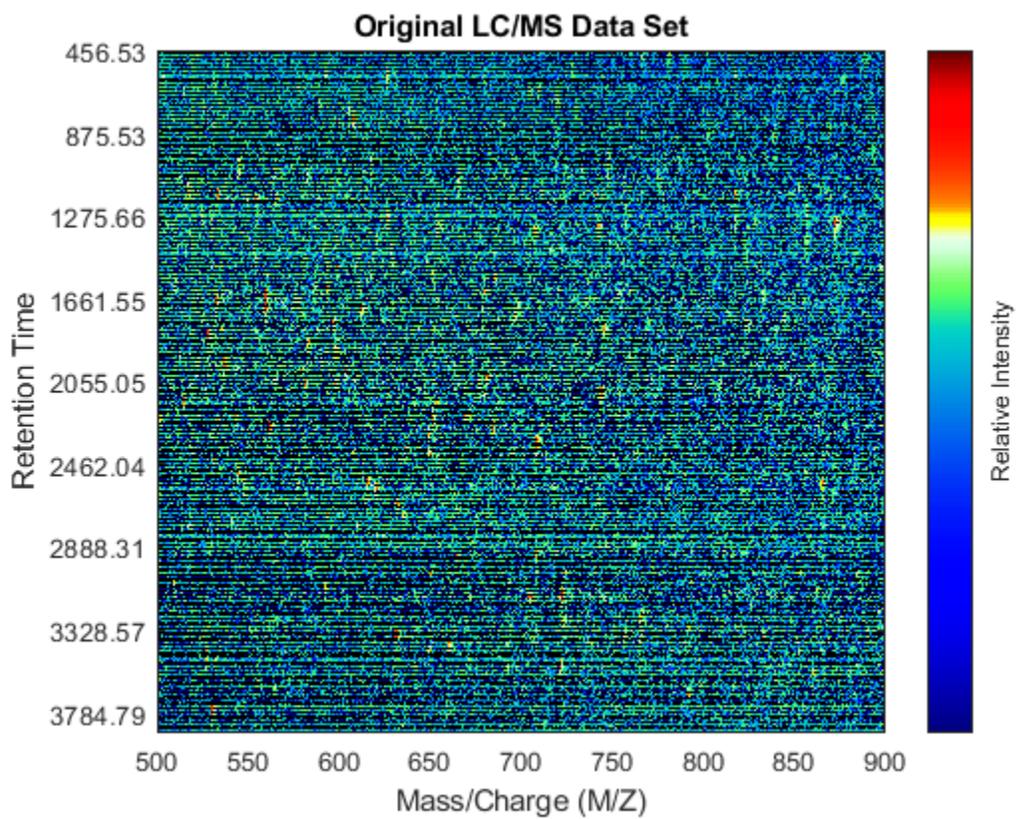
```
ax = gca;
ax.YTick = 1:numPeaks;
ax.YTickLabel = num2str(time(xRows));
axis([500 900 0 numPeaks 0 1e8])
```

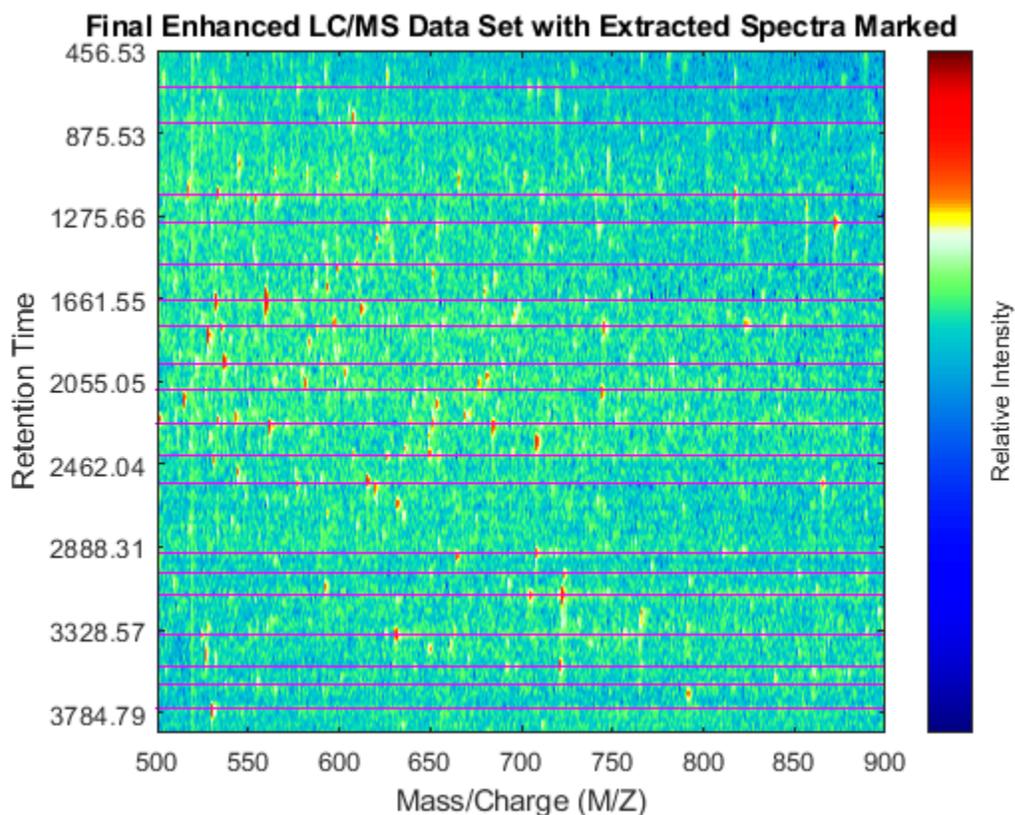
```
xlabel('Mass/Charge (M/Z)')
ylabel('Time')
zlabel('Relative Ion Intensity')
title('Extracted Spectra Subset')
```



Overlay markers for the extracted spectra over the enhanced heatmap.

```
linkaxes(findobj(fh4, 'Tag', 'MSHeatMap'), 'off')
figure(fh4)
hold on
for i = 1:numPeaks
    plot([400 1500], xRows([i i]), 'm')
end
axis([500 900 100 925])
dp4.Visible = 'off';
title('Final Enhanced LC/MS Data Set with Extracted Spectra Marked')
```





References

- [1] Desiere, F. et al., "The Peptide Atlas Project", *Nucleic Acids Research*, 34:D655-8, 2006.
- [2] Purvine, S., Kolker, N., and Kolker, E., "Spectral Quality Assessment for High-Throughput Tandem Mass Spectrometry Proteomics", *OMICS: A Journal of Integrative Biology*, 8(3):255-65, 2004.
- [3] Kazmi, A.S., et al., "Alignment of high resolution mass spectra: Development of a heuristic approach for metabolomics", *Metabolomics*, 2(2):75-83, 2006.
- [4] Jeffries, N., "Algorithms for alignment of mass spectrometry proteomic data", *Bioinformatics*, 21(14):3066-3073, 2005.
- [5] Yu, W., et al., "Multiple peak alignment in sequential data analysis: A scale-space based approach", *IEEE®/ACM Trans. Computational Biology and Bioinformatics*, 3(3):208-219, 2006.
- [6] Sakoe, H. and Chiba s., "Dynamic programming algorithm optimization for spoken word recognition", *IEEE Trans. Acoustics, Speech and Signal Processing*, ASSP-26(1):43-9, 1978.

Identifying Significant Features and Classifying Protein Profiles

This example shows how to classify mass spectrometry data and use some statistical tools to look for potential disease markers and proteomic pattern diagnostics.

Introduction

Serum proteomic pattern diagnostics can be used to differentiate samples from patients with and without disease. Profile patterns are generated using surface-enhanced laser desorption and ionization (SELDI) protein mass spectrometry. This technology has the potential to improve clinical diagnostics tests for cancer pathologies. The goal is to select a reduced set of measurements or "features" that can be used to distinguish between cancer and control patients. These features will be ion intensity levels at specific mass/charge values.

Preprocess Data

The ovarian cancer data set in this example is from the FDA-NCI Clinical Proteomics Program Databank. The data set was generated using the WCX2 protein array. The data set includes 95 controls and 121 ovarian cancers. For a detailed description of this data set, see [1] and [4].

This example assumes that you already have the preprocessed data `OvarianCancerQAQCdataset.mat`. However, if you do not have the data file, you can recreate by following the steps in the example "Batch Processing of Spectra Using Sequential and Parallel Computing" on page 6-77.

Alternatively, you can run the provided script `msseqprocessing.m`.

The preprocessing steps from the script and example listed above are intended to illustrate a representative set of possible pre-processing procedures. Using different steps or parameters may lead to different and possibly improved results of this example.

Load Data

Once you have the preprocessed data, you can load it into MATLAB.

```
load OvarianCancerQAQCdataset
whos
```

Name	Size	Bytes	Class	Attributes
MZ	15000x1	120000	double	
Y	15000x216	25920000	double	
grp	216x1	25056	cell	

There are three variables: **MZ**, **Y**, **grp**. **MZ** is the mass/charge vector, **Y** is the intensity values for all 216 patients (control and cancer), and **grp** holds the index information as to which of these samples represent cancer patients and which ones represent normal patients.

Initialize some variables that will be used through out the example.

```
N = numel(grp); % Number of samples
Cidx = strcmp('Cancer',grp); % Logical index vector for Cancer samples
```

```

Nidx = strcmp('Normal',grp);           % Logical index vector for Normal samples
Cvec = find(Cidx);                     % Index vector for Cancer samples
Nvec = find(Nidx);                     % Index vector for Normal samples
xAxisLabel = 'Mass/Charge (M/Z)';      % x label for plots
yAxisLabel = 'Ion Intensity';         % y label for plots

```

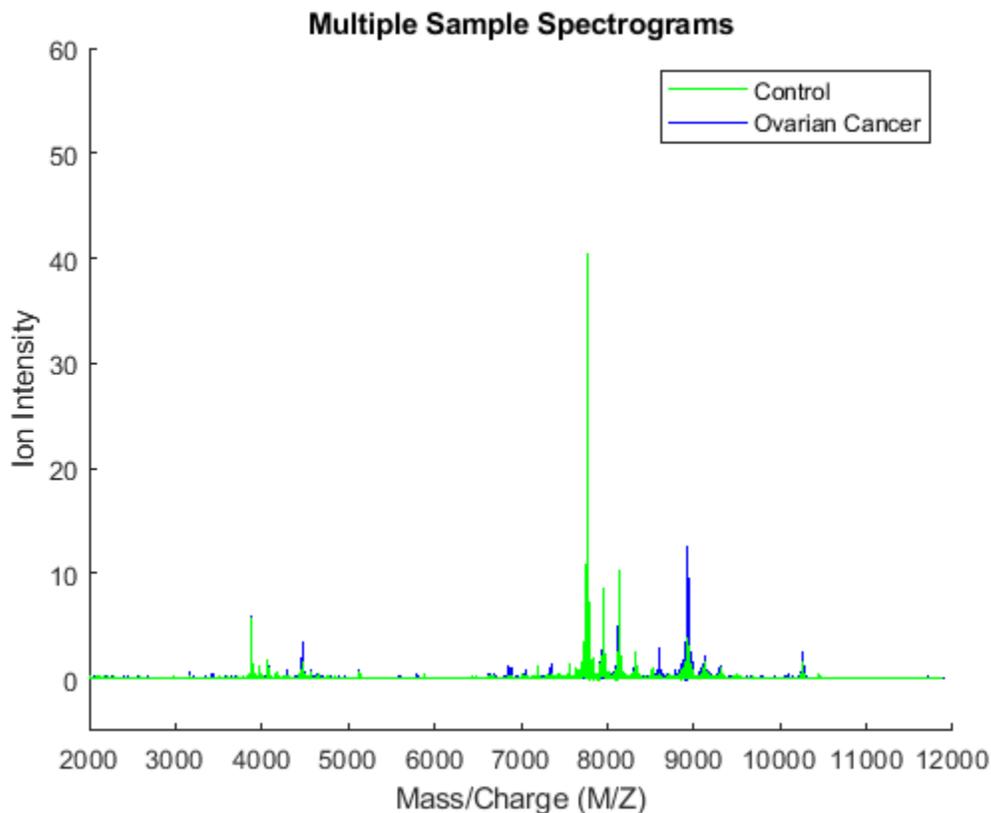
Visualizing Some of the Samples

You can plot some data sets into a figure window to visually compare profiles from the two groups; in this example five spectrograms from cancer patients (blue) and five from control patients (green) are displayed.

```

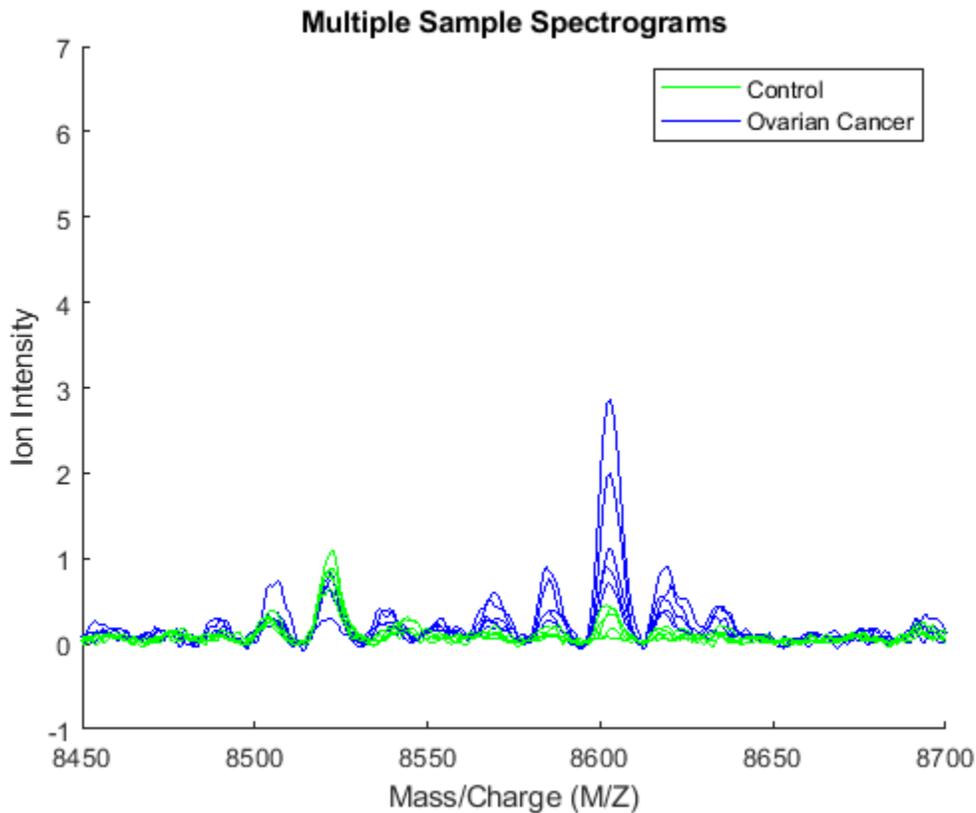
figure; hold on;
hC = plot(MZ,Y(:,Cvec(1:5)),'b');
hN = plot(MZ,Y(:,Nvec(1:5)),'g');
xlabel(xAxisLabel); ylabel(yAxisLabel);
axis([2000 12000 -5 60])
legend([hN(1),hC(1)],{'Control','Ovarian Cancer'})
title('Multiple Sample Spectrograms')

```



Zooming in on the region from 8500 to 8700 M/Z shows some peaks that might be useful for classifying the data.

```
axis([8450,8700,-1,7])
```



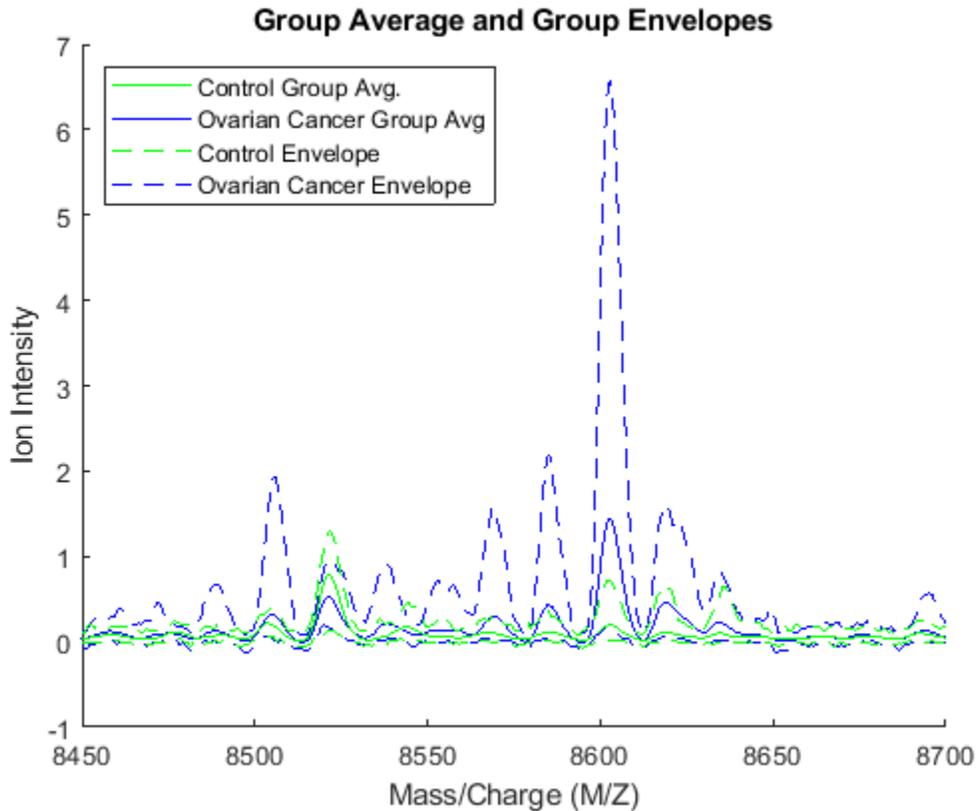
Another way to visualize the whole data set is to look at the group average signal for the control and cancer samples. You can plot the group average and the envelopes of each group.

```

mean_N = mean(Y(:,Nidx),2); % group average for control samples
max_N = max(Y(:,Nidx),[],2); % top envelopes of the control samples
min_N = min(Y(:,Nidx),[],2); % bottom envelopes of the control samples
mean_C = mean(Y(:,Cidx),2); % group average for cancer samples
max_C = max(Y(:,Cidx),[],2); % top envelopes of the control samples
min_C = min(Y(:,Cidx),[],2); % bottom envelopes of the control samples

figure; hold on;
hC = plot(MZ,mean_C,'b');
hN = plot(MZ,mean_N,'g');
gC = plot(MZ,[max_C min_C],'b--');
gN = plot(MZ,[max_N min_N],'g--');
xlabel(xAxisLabel); ylabel(yAxisLabel);
axis([8450,8700,-1,7])
legend([hN,hC,gN(1),gC(1)],{'Control Group Avg.','Ovarian Cancer Group Avg',...
    'Control Envelope','Ovarian Cancer Envelope'},...
    'Location','NorthWest')
title('Group Average and Group Envelopes')

```



Observe that apparently there is no single feature that can discriminate both groups perfectly.

Ranking Key Features

A simple approach for finding significant features is to assume that each M/Z value is independent and compute a two-way t-test. `rankfeatures` returns an index to the most significant M/Z values, for instance 100 indices ranked by the absolute value of the test statistic. This feature selection method is also known as a filtering method, where the learning algorithm is not involved on how the features are selected.

```
[feat,stat] = rankfeatures(Y,grp,'CRITERION','ttest','NUMBER',100);
```

The first output of `rankfeatures` can be used to extract the M/Z values of the significant features.

```
sig_Masses = MZ(feat);
sig_Masses(1:7)' %display the first seven
```

```
ans =
```

```
1.0e+03 *
 8.1009  8.1016  8.1024  8.1001  8.1032  7.7366  7.7359
```

The second output of `rankfeatures` is a vector with the absolute value of the test statistic. You can plot it over the spectra using `yyaxis`.

```

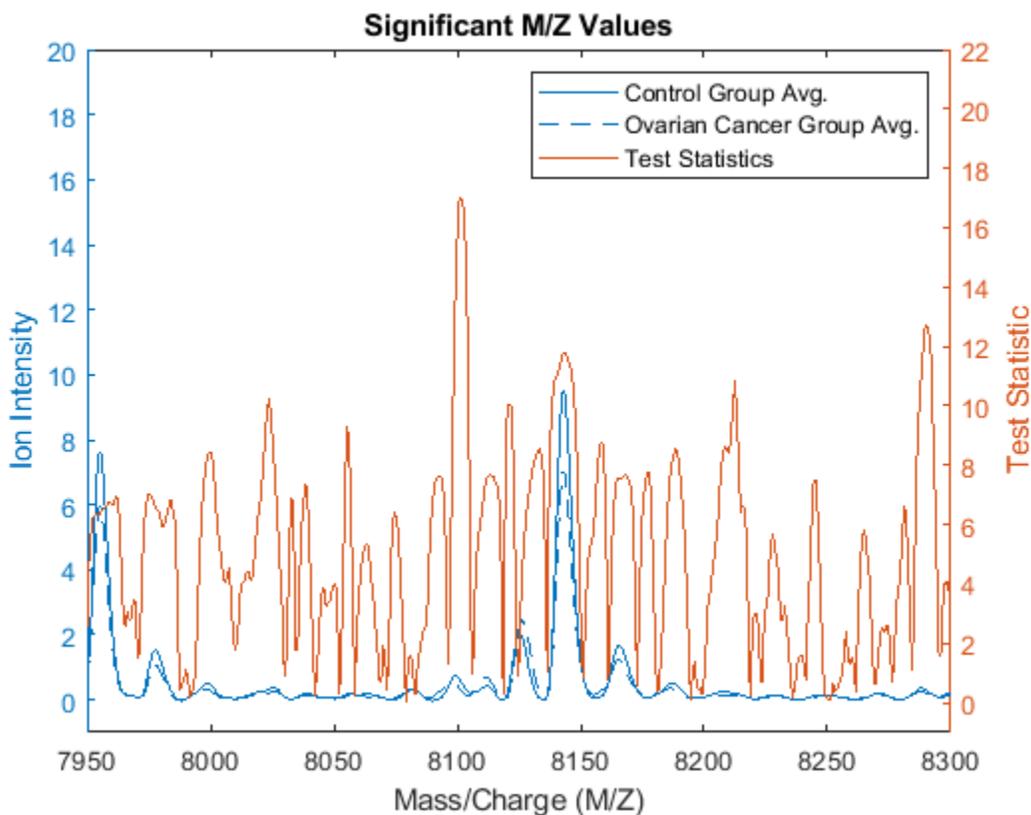
figure;

yyaxis left
plot(MZ, [mean_N mean_C]);
ylim([-1,20])
xlim([7950,8300])
title('Significant M/Z Values')
xlabel(xAxisLabel);
ylabel(yAxisLabel);

yyaxis right
plot(MZ,stat);
ylim([-1,22])
ylabel('Test Statistic');

legend({'Control Group Avg.', 'Ovarian Cancer Group Avg.', 'Test Statistics'})

```



Notice that there are significant regions at high M/Z values but low intensity (~8100 Da.). Other approaches to measure class separability are available in rankfeatures, such as entropy based, Bhattacharyya, or the area under the empirical receiver operating characteristic (ROC) curve.

Blind Classification Using Linear Discriminant Analysis (LDA)

Now that you have identified some significant features, you can use this information to classify the cancer and normal samples. Due to the small number of samples, you can run a cross-validation using the 20% holdout to have a better estimation of the classifier performance. `cvpartition` allows you

to set the training and test indices for different types of system evaluation methods, such as hold-out, K-fold and Leave-M-Out.

```
per_eval = 0.20;           % training size for cross-validation
rng('default');          % initialize random generator to the same state
                           % used to generate the published example
cv = cvpartition(grp, 'holdout', per_eval)
```

```
cv =
```

```
Hold-out cross validation partition
  NumObservations: 216
    NumTestSets: 1
      TrainSize: 173
        TestSize: 43
          IsCustom: 0
```

Observe that features are selected only from the training subset and the validation is performed with the test subset. `classperf` allows you to keep track of multiple validations.

```
cp_lda1 = classperf(grp); % initializes the CP object
for k=1:10 % run cross-validation 10 times
    cv = repartition(cv);
    feat = rankfeatures(Y(:,training(cv)),grp(training(cv)), 'NUMBER',100);
    c = classify(Y(feat,test(cv))',Y(feat,training(cv))',grp(training(cv)));
    classperf(cp_lda1,c,test(cv)); % updates the CP object with current validation
end
```

After the loop you can assess the performance of the overall blind classification using any of the properties in the CP object, such as the error rate, sensitivity, specificity, and others.

```
cp_lda1
```

```
cp_lda1 =
```

```
classperformance with properties:
    ClassLabels: {2x1 cell}
    GroundTruth: [216x1 double]
    NumberOfObservations: 216
    ValidationCounter: 10
    SampleDistribution: [216x1 double]
    ErrorDistribution: [216x1 double]
    SampleDistributionByClass: [2x1 double]
    ErrorDistributionByClass: [2x1 double]
    CountingMatrix: [3x2 double]
    CorrectRate: 0.8488
    ErrorRate: 0.1512
    LastCorrectRate: 0.8837
    LastErrorRate: 0.1163
    InconclusiveRate: 0
    ClassifiedRate: 1
    Sensitivity: 0.8208
    Specificity: 0.8842
    PositivePredictiveValue: 0.8995
    NegativePredictiveValue: 0.7962
```

```

PositiveLikelihood: 7.0890
NegativeLikelihood: 0.2026
  Prevalence: 0.5581
  DiagnosticTable: [2x2 double]
    Label: ''
  Description: ''
  ControlClasses: 2
  TargetClasses: 1

```

This naive approach for feature selection can be improved by eliminating some features based on the regional information. For example, 'NWEIGHT' in rankfeatures outweighs the test statistic of neighboring M/Z features such that other significant M/Z values can be incorporated into the subset of selected features

```

cp_lda2 = classperf(grp); % initializes the CP object
for k=1:10 % run cross-validation 10 times
  cv = repartition(cv);
  feat = rankfeatures(Y(:,training(cv)),grp(training(cv)), 'NUMBER',100, 'NWEIGHT',5);
  c = classify(Y(feat,test(cv))',Y(feat,training(cv))',grp(training(cv)));
  classperf(cp_lda2,c,test(cv)); % updates the CP object with current validation
end
cp_lda2.CorrectRate % average correct classification rate

```

```

ans =

    0.9023

```

PCA/LDA Reduction of the Data Dimensionality

Lilien et al. presented in [2] an algorithm to reduce the data dimensionality that uses principal component analysis (PCA), then LDA is used to classify the groups. In this example 2000 of the most significant features in the M/Z space are mapped to the 150 principal components

```

cp_pcalda = classperf(grp); % initializes the CP object
for k=1:10 % run cross-validation 10 times
  cv = repartition(cv);
  % select the 2000 most significant features.
  feat = rankfeatures(Y(:,training(cv)),grp(training(cv)), 'NUMBER',2000);
  % PCA to reduce dimensionality
  P = pca(Y(feat,training(cv))');
  % Project into PCA space
  x = Y(feat,:) * P(:,1:150);
  % Use LDA
  c = classify(x(test(cv),:),x(training(cv),:),grp(training(cv)));
  classperf(cp_pcalda,c,test(cv));
end
cp_pcalda.CorrectRate % average correct classification rate

```

```

ans =

    0.9814

```

Randomized Search for Subset Feature Selection

Feature selection can also be reinforced by classification, this approach is usually referred to as a wrapper selection method. Randomized search for feature selection generates random subsets of features and assesses their quality independently with the learning algorithm. Later, it selects a pool of the most frequent good features. Li et al. in [3] apply this concept to the analysis of protein expression patterns. The `randfeatures` function allows you to search a subset of features using LDA or a k-nearest neighbor classifier over randomized subsets of features.

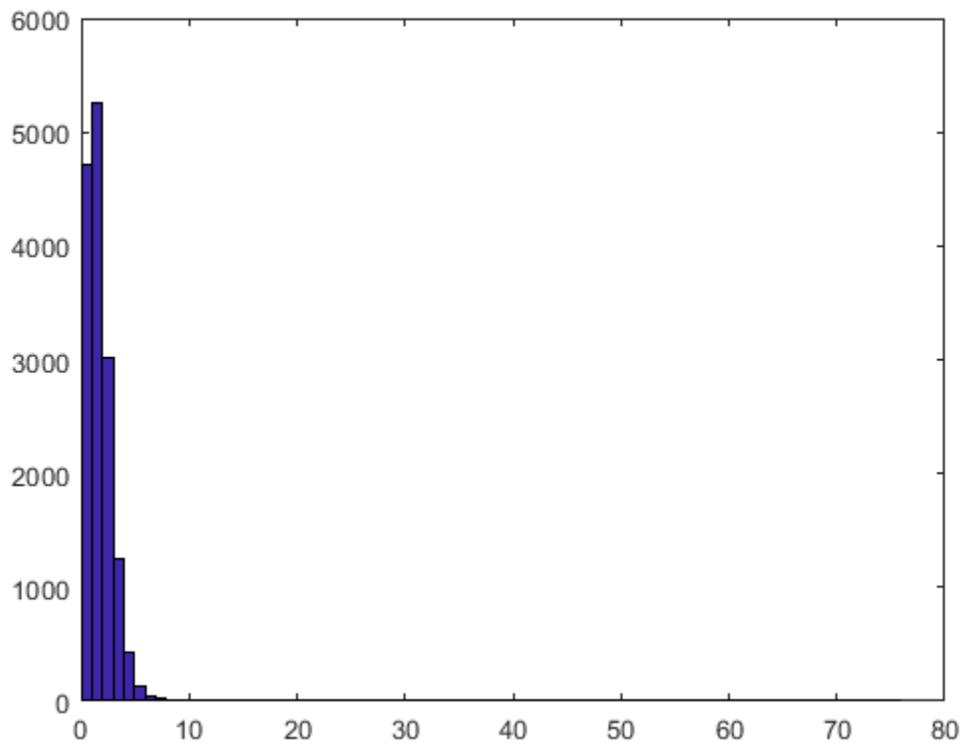
Note: the following example is computationally intensive, so it has been disabled from the example. Also, for better results you should increase the pool size and the stringency of the classifier from the default values in `randfeatures`. Type `help randfeatures` for more information.

```
if 0 % <== change to 1 to enable. This may use extensive time to complete.
    cv = repartition(cv);
    [feat,fCount] = randfeatures(Y(:,training(cv)),grp(training(cv)),...
                               'CLASSIFIER','da','PerformanceThreshold',0.90);
else
    load randFeatCancerDetect
end
```

Assess the Quality of the Selected Features with the Evaluation Set

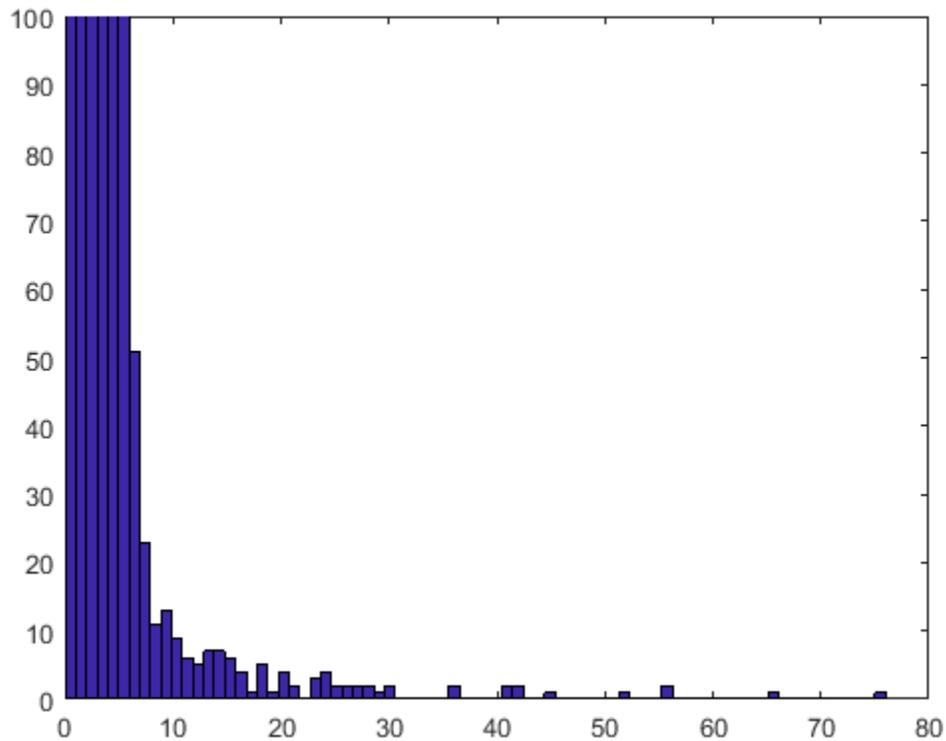
The first output from `randfeatures` is an ordered list of indices of MZ values. The first item occurs most frequently in the subsets where good classification was achieved. The second output is the actual counts of the number of times each value was selected. You can use `hist` to look at this distribution.

```
figure;
hist(fCount,max(fCount)+1);
```



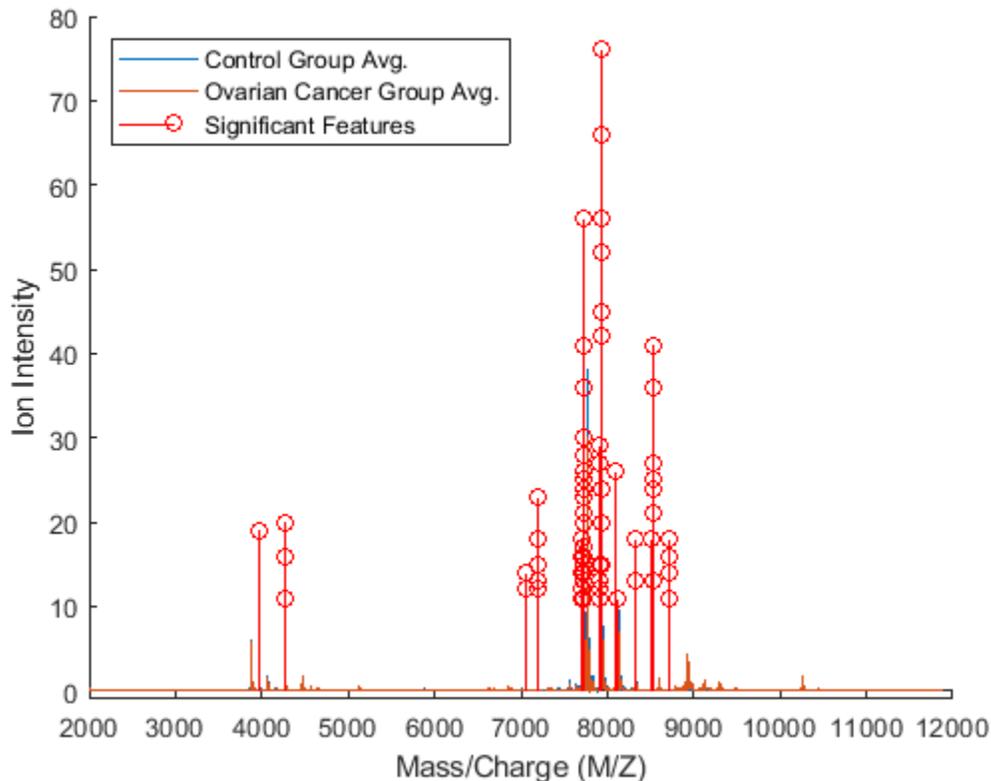
You will see that most values appear at most once in a selected subset. Zooming in gives a better idea of the details for the more frequently selected values.

```
axis([0 80 0 100])
```



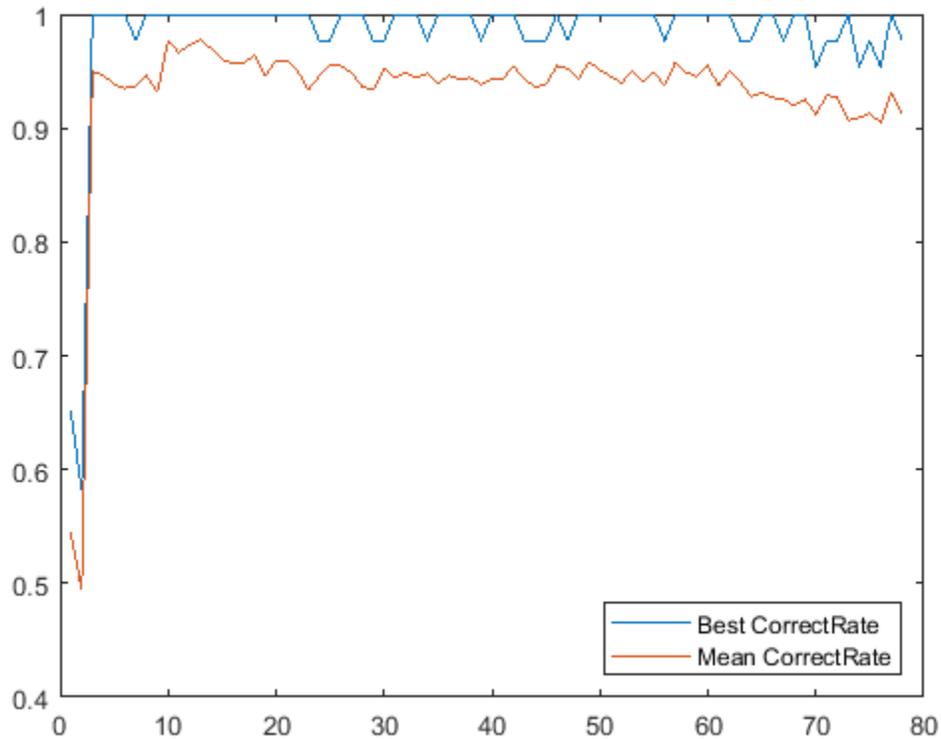
Only a few values were selected more than 10 times. You can visualize these by using a stem plot to show the most frequently selected features.

```
figure; hold on;
sigFeats = fCount;
sigFeats(sigFeats<=10) = 0;
plot(MZ,[mean_N mean_C]);
stem(MZ(sigFeats>0),sigFeats(sigFeats>0),'r');
axis([2000,12000,-1,80])
legend({'Control Group Avg.','Ovarian Cancer Group Avg.','Significant Features'}, ...
       'Location','NorthWest')
xlabel(xAxisLabel); ylabel(yAxisLabel);
```



These features appear to clump together in several groups. You can investigate further how many of the features are significant by running the following experiment. The most frequently selected feature is used to classify the data, then the two most frequently selected features are used and so on until all the features that were selected more than 10 times are used. You can then see if adding more features improves the classifier.

```
nSig = sum(fCount>10);
cp_rndfeat = zeros(20,nSig);
for i = 1:nSig
    for j = 1:20
        cv = repartition(cv);
        P = pca(Y(feats(1:i),training(cv))');
        x = Y(feats(1:i),:)' * P;
        c = classify(x(test(cv),:),x(training(cv),:),grp(training(cv)));
        cp = classperf(grp,c,test(cv));
        cp_rndfeat(j,i) = cp.CorrectRate; % average correct classification rate
    end
end
figure
plot(1:nSig, [max(cp_rndfeat);mean(cp_rndfeat)]);
legend({'Best CorrectRate','Mean CorrectRate'},'Location','SouthEast')
```

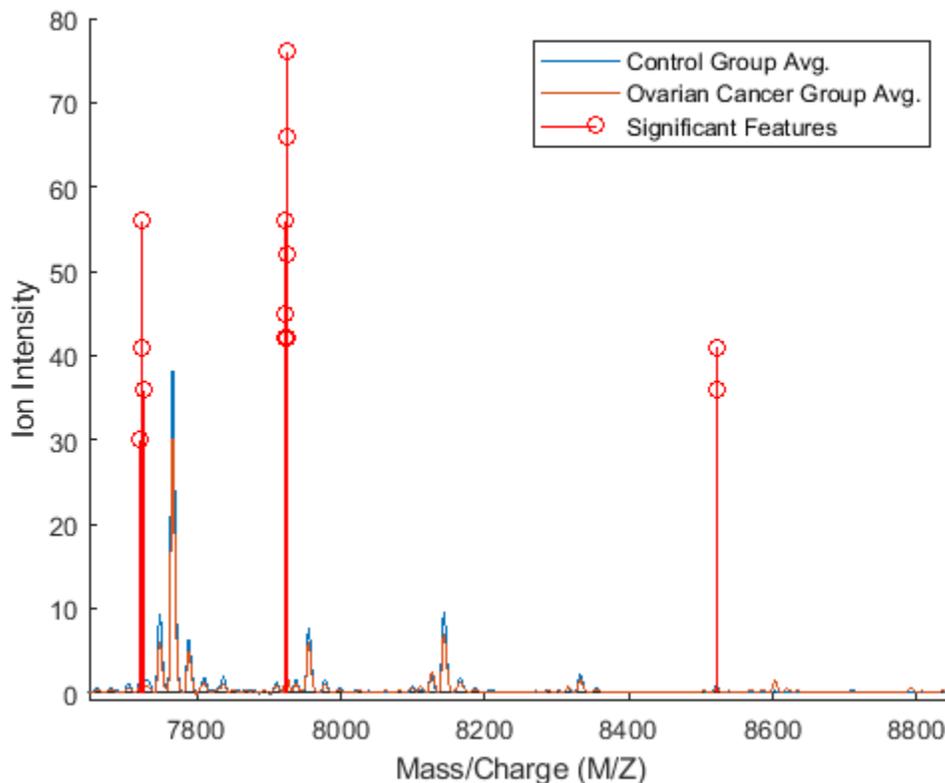


From this graph you can see that for as few as three features it is sometimes possible to get perfect classification. You will also notice that the maximum of the mean correct rate occurs for a small number of features and then gradually decreases.

```
[bestAverageCR, bestNumFeatures] = max(mean(cp_rndfeat));
```

You can now visualize the features that give the best average classification. You can see that these actually correspond to only three peaks in the data.

```
figure; hold on;
sigFeats = fCount;
sigFeats(sigFeats<=10) = 0;
ax_handle = plot(MZ,[mean_N mean_C]);
stem(MZ(feats(1:bestNumFeatures)),sigFeats(feats(1:bestNumFeatures)),'r');
axis([7650,8850,-1,80])
legend({'Control Group Avg.','Ovarian Cancer Group Avg.','Significant Features'})
xlabel(xAxisLabel); ylabel(yAxisLabel);
```



Alternative Statistical Learning Algorithms

There are many classification tools in MATLAB® that you can also use to analyze proteomic data. Among them are support vector machines (`fitcsvm`), k-nearest neighbors (`fitcknn`), neural networks (Deep Learning Toolbox™), and classification trees (`fitctree`). For feature selection, you can also use sequential subset feature selection (`sequentialfs`) or optimize the randomized search methods by using a genetic algorithm (Global Optimization Toolbox). For example, see “Genetic Algorithm Search for Features in Mass Spectrometry Data” on page 6-71.

References

- [1] Conrads, T P, V A Fusaro, S Ross, D Johann, V Rajapakse, B A Hitt, S M Steinberg, et al. “High-Resolution Serum Proteomic Features for Ovarian Cancer Detection.” *Endocrine-Related Cancer*, June 2004, 163-78.
- [2] Lilien, Ryan H., Hany Farid, and Bruce R. Donald. “Probabilistic Disease Classification of Expression-Dependent Proteomic Data from Mass Spectrometry of Human Serum.” *Journal of Computational Biology* 10, no. 6 (December 2003): 925-46.
- [3] Li, L., D. M. Umbach, P. Terry, and J. A. Taylor. “Application of the GA/KNN Method to SELDI Proteomics Data.” *Bioinformatics* 20, no. 10 (July 1, 2004): 1638-40.
- [4] Petricoin, Emanuel F, Ali M Ardekani, Ben A Hitt, Peter J Levine, Vincent A Fusaro, Seth M Steinberg, Gordon B Mills, et al. “Use of Proteomic Patterns in Serum to Identify Ovarian Cancer.” *The Lancet* 359, no. 9306 (February 2002): 572-77.

See Also

`msnorm` | `rankfeatures` | `classperf`

Related Examples

- “Batch Processing of Spectra Using Sequential and Parallel Computing” on page 6-77

Differential Analysis of Complex Protein and Metabolite Mixtures Using Liquid Chromatography/Mass Spectrometry (LC/MS)

This example shows how the `SAMPLEALIGN` function can correct nonlinear warping in the chromatographic dimension of hyphenated mass spectrometry data sets without the need for full identification of the sample compounds and/or the use of internal standards. By correcting such warping between a pair (or set) of biologically related samples, differential analysis is enhanced and can be automated.

Introduction

The use of complex peptide and metabolite mixtures in LC/MS requires label-free alignment procedures. The analysis of this type of data requires searching for statistically significant differences between biologically related data sets, without the need for a full identification of all the compounds in the sample (either peptides/proteins or metabolites). Comparing compounds requires alignment in two dimensions, the mass-charge dimension and the retention time dimension [1]. In the examples “Preprocessing Raw Mass Spectrometry Data” on page 6-2 and “Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling” on page 6-19, you can learn how to use the `MSALIGN`, `MSPALIGN`, and `SAMPLEALIGN` functions to warp or calibrate different type of anomalies in the mass/charge dimension. In this example, you will learn how to use the `SAMPLEALIGN` function to also correct the nonlinear and unpredicted variations in the retention time dimension.

While it is possible to implement alternative methods for aligning retention times, other approaches typically require identification of compounds, which is not always feasible, or manual manipulations that thwart attempts to automate for high throughput data analysis.

Data Set Description

This example uses two samples in PAe000153 and PAe000155 available from Peptide Atlas [2]. The samples are LC-ESI-MS scans of four salt protein fractions from the *saccharomyces cerevisiae* each containing more than 1000 peptides. Yeast samples were treated with different chemicals (glycine and serine) in order to get two biologically diverse samples. Time alignment of these two data sets is one of the most challenging cases reported in [3]. The data sets are not distributed with MATLAB®. You can try other data sets available in public databases for protein data, such as Peptide Atlas Repository. If you receive any errors related to memory or java heap space, try increasing your java heap space as described here. LC/MS data analysis requires extended amounts of memory from the operating system; if you receive “Out of memory” errors when running this example, try increasing the virtual memory (or swap space) of your operating system. For details, see “Resolve “Out of Memory” Errors”.

Read and extract the lists of peaks from the XML files containing the intensity data for the sample treated with Serine and the sample treated with Glycine.

```
ser = mzxmlread('005_1.mzXML')
[ps,ts] = mzxml2peaks(ser,'level',1);
gly = mzxmlread('005a.mzXML')
[pg,tg] = mzxml2peaks(gly,'level',1);
```

```
ser =
```

```
struct with fields:

    scan: [5610x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]

gly =

struct with fields:

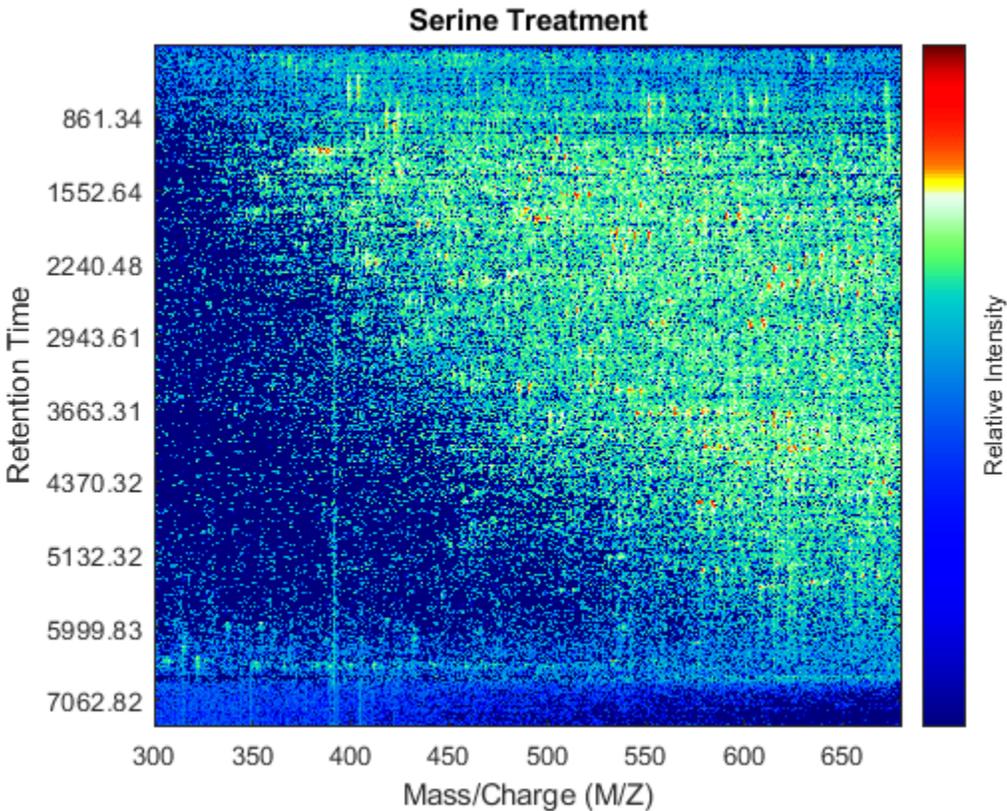
    scan: [5518x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

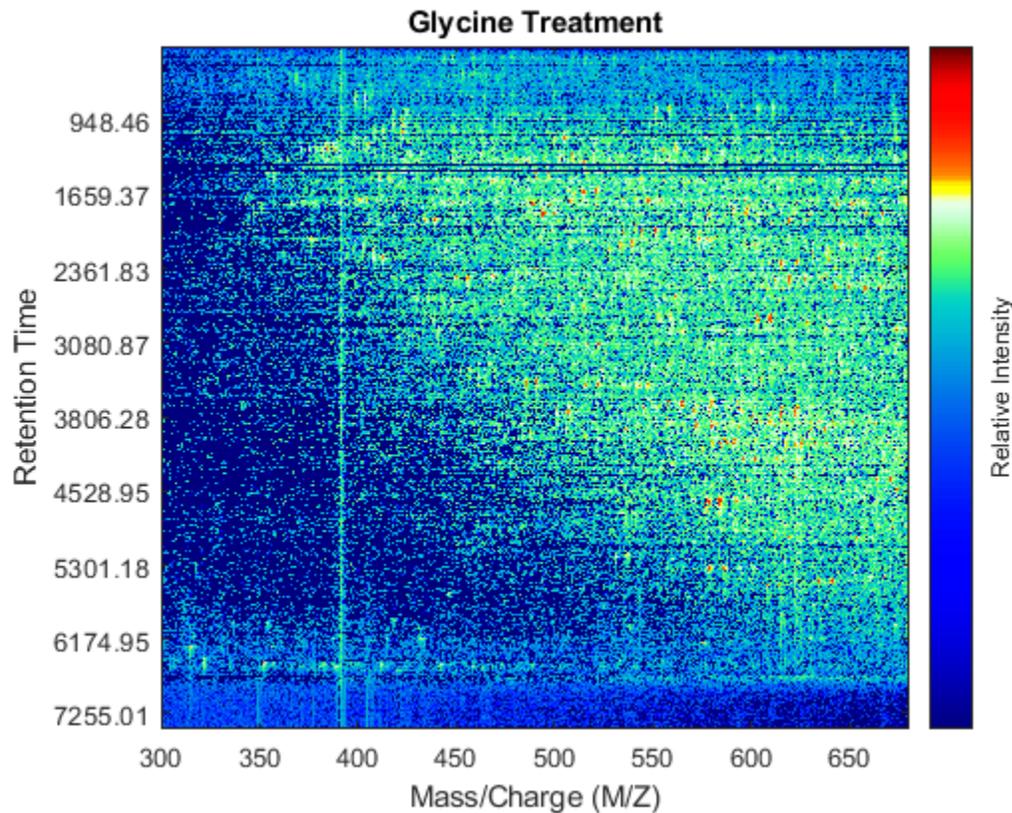
Use the `MSPPRESAMPLE` function to resample the data sets while preserving the peak heights and locations in the mass/charge dimension. Data sets are resampled to have both a common grid with 5,000 mass/charge values. A common grid is desirable for comparative visualization, and for differential analysis.

```
[MZs,Ys] = mspresample(ps,5000);
[MZg,Yg] = mspresample(pg,5000);
```

Use the `MSHEATMAP` function to visualize both samples. When working with heat maps it is a common technique to display the logarithm of the ion intensities, which enhances the dynamic range of the colormap.

```
fh1 = msheatmap(MZs,ts,log(Ys),'resolution',0.15);
title('Serine Treatment')
fh2 = msheatmap(MZg,tg,log(Yg),'resolution',0.15);
title('Glycine Treatment')
```



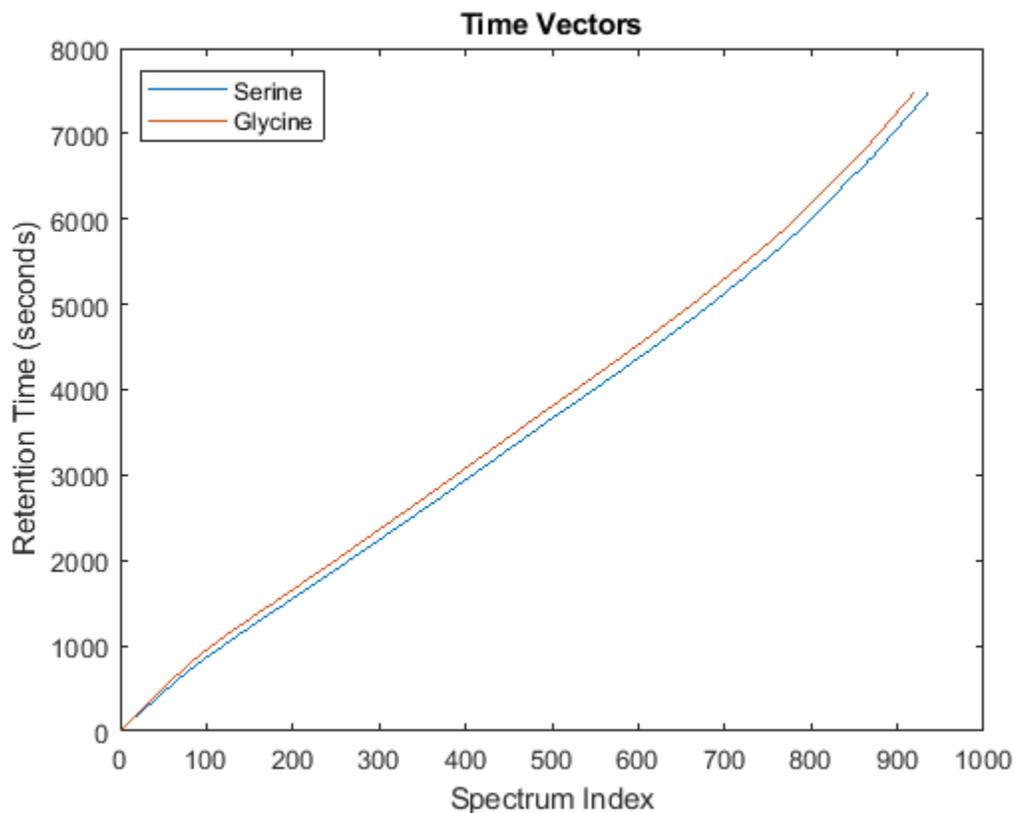


Detailed Inspection of the Misalignment Problems

Notice you can visualize the data sets separately; however, the time vectors have different size, and therefore the heat maps have different number of rows (or Ys and Yg have different number of columns). Moreover, the sampling rate is not constant and the shift between the time vectors is not linear.

```
whos('Ys','Yg','ts','tg')
figure
plot(1:numel(ts),ts,1:numel(tg),tg)
legend('Serine','Glycine','Location','NorthWest')
title('Time Vectors')
xlabel('Spectrum Index')
ylabel('Retention Time (seconds)')
```

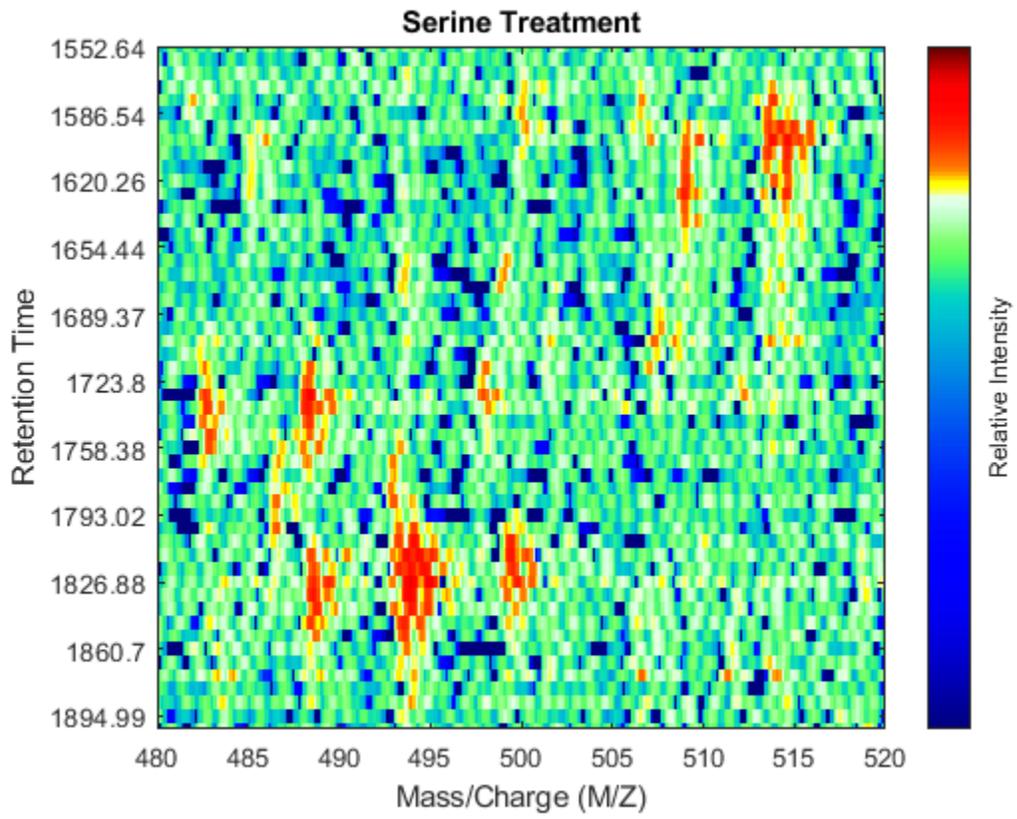
Name	Size	Bytes	Class	Attributes
Yg	5000x921	18420000	single	
Ys	5000x937	18740000	single	
tg	921x1	7368	double	
ts	937x1	7496	double	

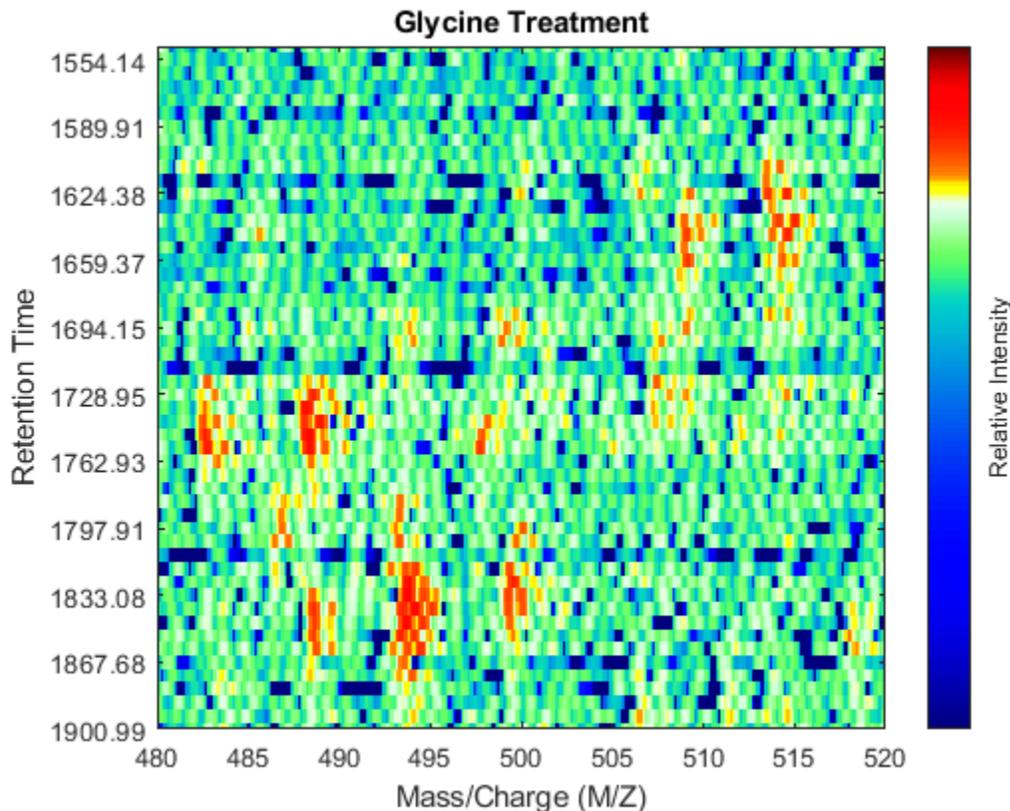


To observe the same region of interest in both data sets, you need to calculate the appropriate row indices in each matrix. For example, to inspect the peptide peaks in the 480-520 Da MZ range and 1550-1900 seconds retention time range, you need to find the closest matches for this range in the time vectors and then zoom in each figure:

```
ind_ser = samplealign(ts,[1550;1900]);  
figure(fh1);  
axis([480 520 ind_ser'])
```

```
ind_gly = samplealign(tg,[1550;1900]);  
figure(fh2);  
axis([480 520 ind_gly'])
```





Even though you zoomed in the same range, you can still observe that the top-right peptide in the axes is shifted in the retention time dimension. In the sample treated with serine, the center of this peak appears to occur at approximately 1595 seconds, while in the sample treated with glycine the putative same peptide occurs at approximately 1630 seconds. This will prevent you from a accurate comparative analysis, even if you resample the data sets to the same time vector. In addition to the shift in the retention time, the data set seems to be improperly calibrated in the mass/charge dimension, because the peaks do not have a compact shape in contiguous spectra.

Mass/Charge Calibration and Enhancement of the Matrices

Before correcting the retention time, you can enhance the samples using an approach similar to the one described in the example “Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling” on page 6-19. For brevity, we only display the MATLAB code without any further explanation:

```
SF = @(x) 1-exp(-x./5e7); % scaling function
DF = @(R,S) sqrt((SF(R(:,2))-SF(S(:,2))).^2 + (R(:,1)-S(:,1)).^2);
CMZ = (315:.10:680)'; % Common Mass/Charge Vector with a finer grid

% Align peaks of the serine sample in the MZ direction
LAI = zeros(size(CMZ));
for i = 1:numel(ps)
    if ~rem(i,250), fprintf(' %d...',i); end
    [k,j] = samplealign([CMZ,LAI],double(ps{i}),'band',1.5,'gap',[0 2],'dist',DF);
    LAI = LAI*.25;
    LAI(k) = LAI(k) + ps{i}(j,2);
    psa{i,1} = [CMZ(k) ps{i}(j,2)];
end
```

```

end

% Align peaks of the glycine sample in the MZ direction
LAI = zeros(size(CMZ));
for i = 1:numel(pg)
    if ~rem(i,250), fprintf(' %d...',i); end
    [k,j] = samplealign([CMZ,LAI],double(pg{i}),'band',1.5,'gap',[0 2],'dist',DF);
    LAI = LAI*.25;
    LAI(k) = LAI(k) + pg{i}(j,2);
    pga{i,1} = [CMZ(k) pg{i}(j,2)];
end

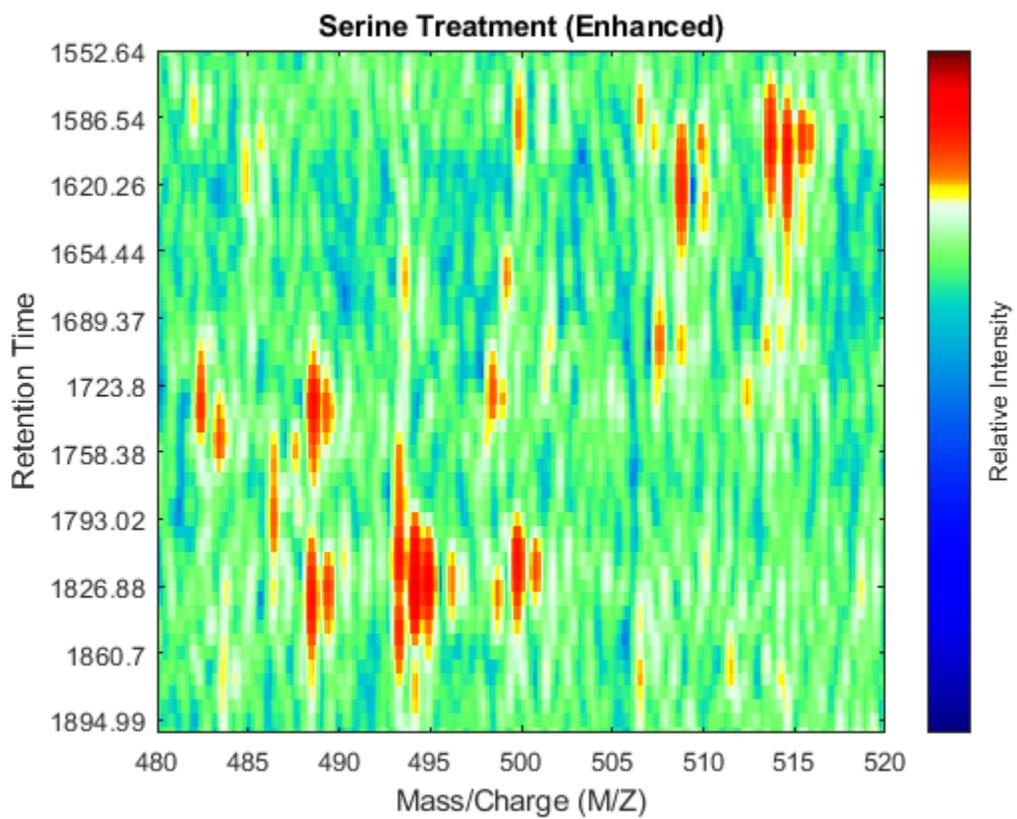
% Peak-preserving resample
[MZs,Ys] = mspresample(psa,5000);
[MZg,Yg] = mspresample(pga,5000);

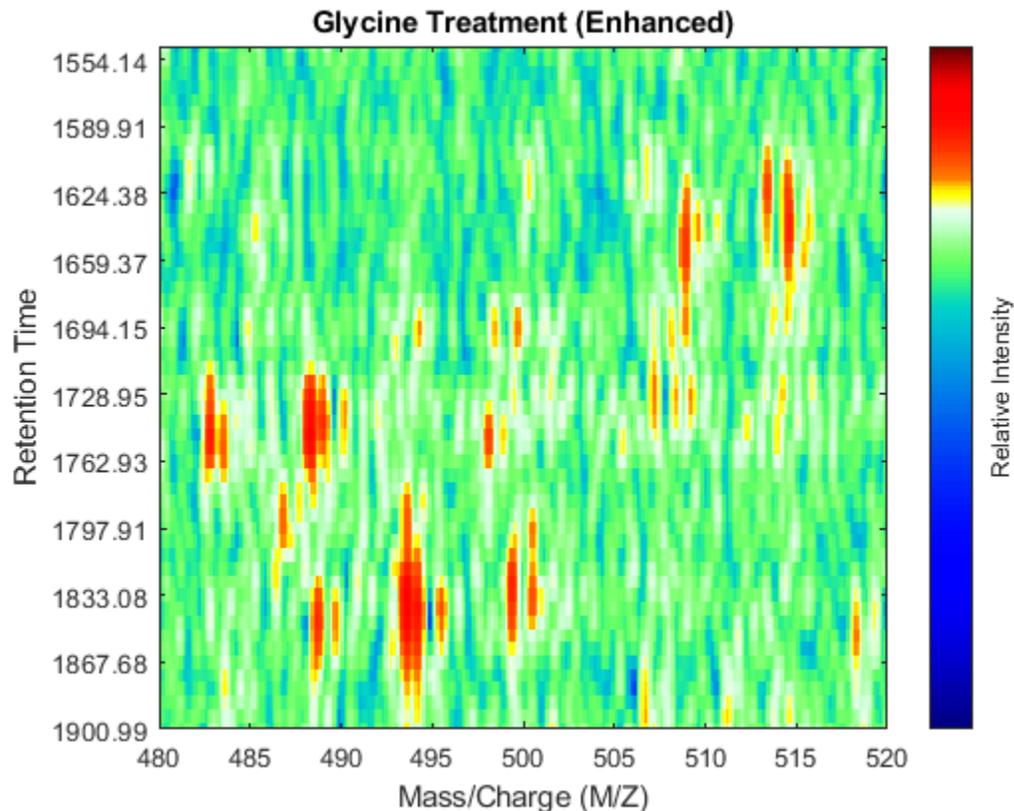
% Gaussian Filtering
Gpulse = exp(-.5*(-10:10).^2)./sum(exp(-.05*(-10:10).^2));
Ysf = convn(Ys,Gpulse,'same');
Ygf = convn(Yg,Gpulse,'same');

% Visualization
fh3 = msheatmap(MZs,ts,log(Ysf),'resolution',0.15);
title('Serine Treatment (Enhanced)')
axis([480 520 ind_ser'])
fh4 = msheatmap(MZg,tg,log(Ygf),'resolution',0.15);
title('Glycine Treatment (Enhanced)')
axis([480 520 ind_gly'])

250... 500... 750... 250... 500... 750...

```





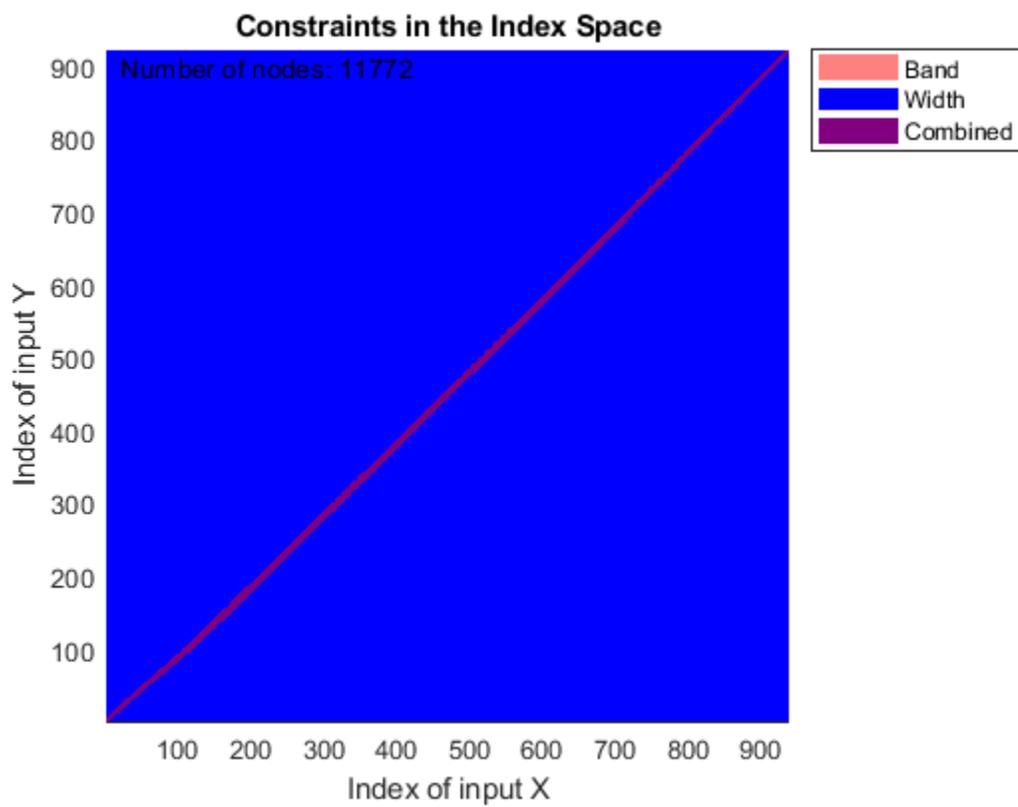
Chromatographic Alignment

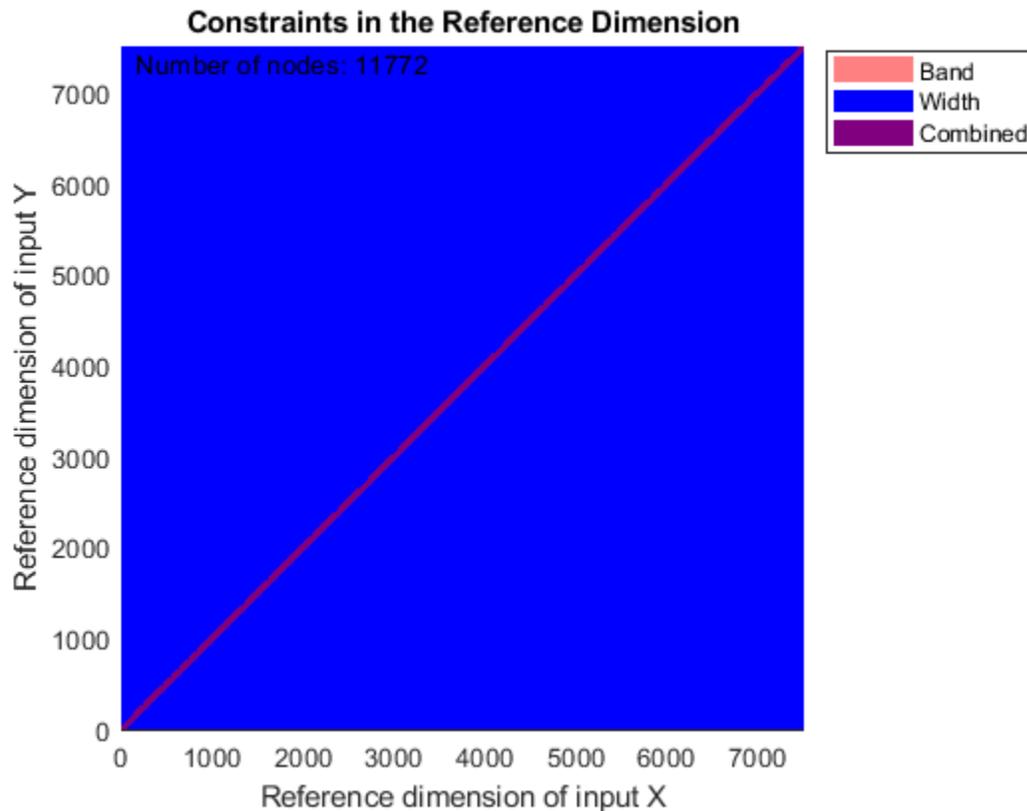
At this point, you have mass/charge calibrated and smoothed the two LC/MS data sets, but you are still unable to perform a differential analysis because the data sets have a small misalignment along the retention time axis.

You can use `SAMPLEALIGN` to correct the drift in the chromatographic domain. First, you should inspect the data and look for the worst case shift, this helps you to estimate the `BAND` constraint. By panning over both heat maps you can observe that common peptide peaks are not shifted more than 50 seconds. Use the input argument `SHOWCONSTRAINTS` to display the constraint space for the time warping operation and assess if the Dynamic Programming (DP) algorithm can handle this problem size. In this case you have less than 12,000 nodes. By omitting the output arguments, `SAMPLEALIGN` displays only the constraints without running the DP algorithm. Also note that the input signals are the filtered and enhanced data sets, but these have been upsampled to 5,000 MZ values, which are very computationally demanding if you use all. Therefore, use the function `MSPALIGN` to obtain a reduced list of mass/charge values (RMZ) indicating where the most intense peaks are, then use the `SAMPLEALIGN` function also to find the indices of MZs (or MZg) that best match the reduced mass/charge vector:

```
RMZ = mspalign([ps;pg])';
idx = samplealign(MZs,RMZ,'width',1); % with these input parameters this
                                     % operation is equivalent to find the
                                     % nearest neighbor for each RMZ in
                                     % MZs.

samplealign([ts Ysf(idx,:)],[tg Ygf(idx,:)'],'band',50,'showconstraints',true)
```





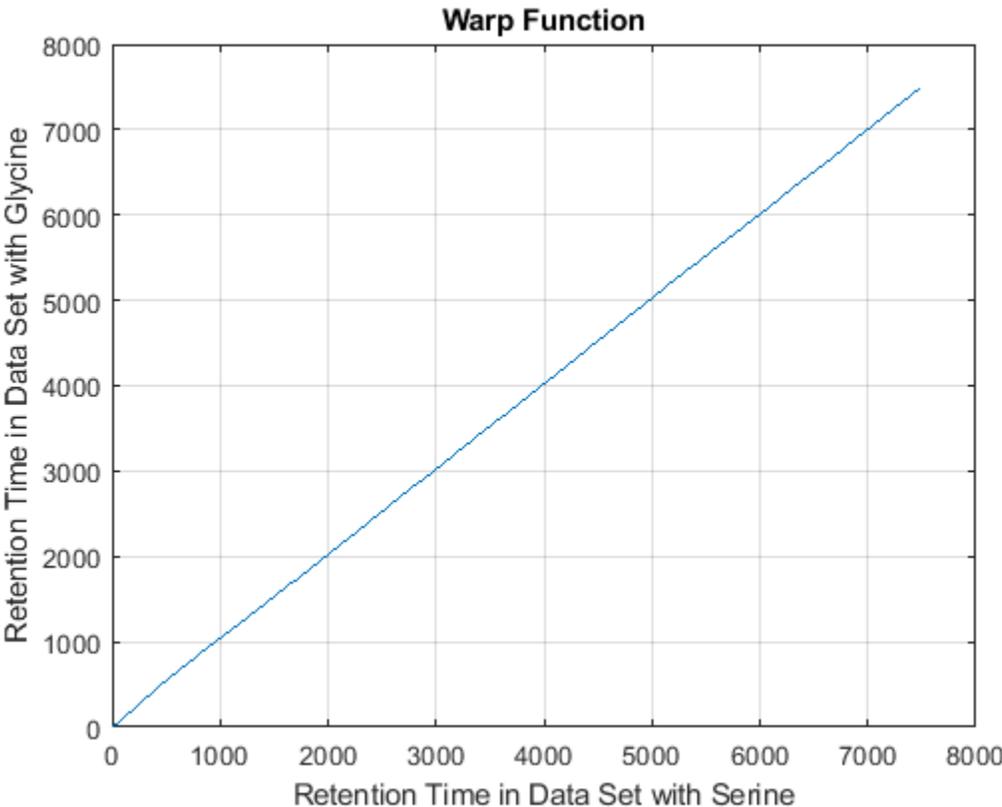
SAMPLEALIGN uses the Euclidean distance as default to score matched pairs of samples. In LC/MS data sets each sample corresponds to a spectrum at a given time, therefore, the cross-correlation between each pair of matched spectra provides a better distance metric. SAMPLEALIGN allows you to define your own metric to calculate the distance between spectra, it is also possible to envision a metric that rewards more spectra pairs that match high ion intensity peaks rather than low ion intensity noisy peaks. Use the input argument WEIGHT to remove the first column from the inputs, which represents the retention time, so the scoring metric between spectra is based only on the ion intensities.

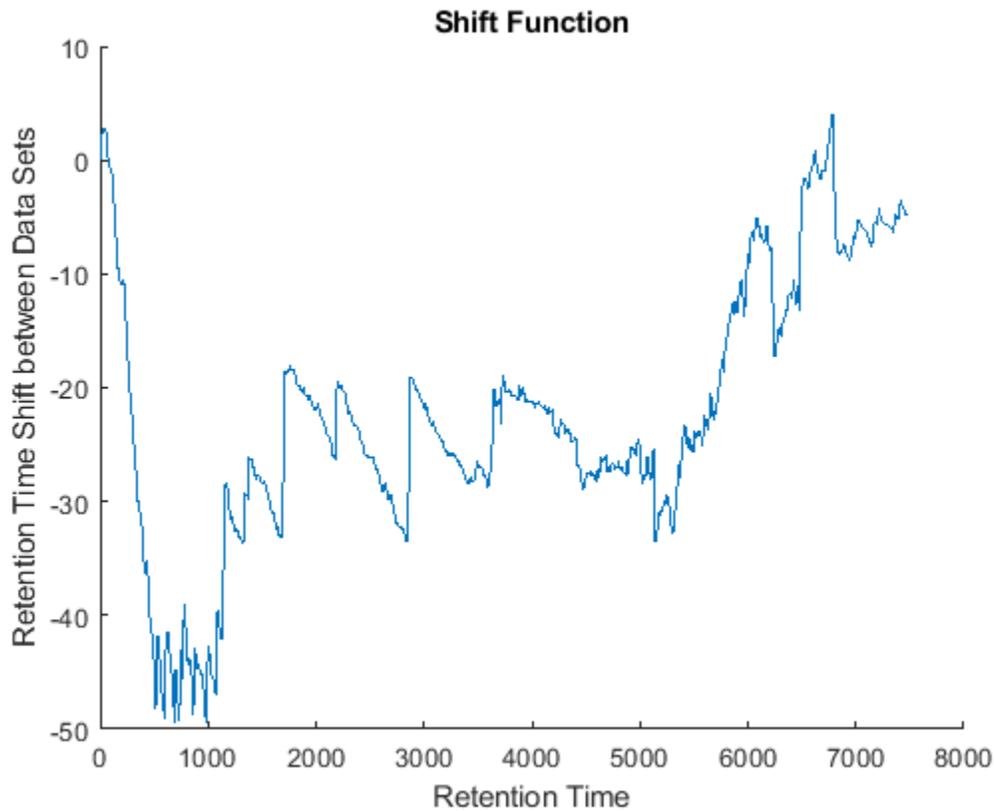
```
cc = @(Xu,Yu) (mean(Xu.*Yu,2).^2)./mean(Xu.*Xu,2)./mean(Yu.*Yu,2);
ub = @(X) bsxfun(@minus,X,mean(X,2));
df = @(x,y) 1-cc(ub(x),ub(y));

[i,j] = samplealign([ts Ysf(idx,:)'],[tg Ygf(idx,:)'], 'band',50,...
    'distance',df , 'weight',[false true(1,numel(idx))]);

fh5 = figure;
plot(ts(i),tg(j)); grid
title('Warp Function')
xlabel('Retention Time in Data Set with Serine')
ylabel('Retention Time in Data Set with Glycine')

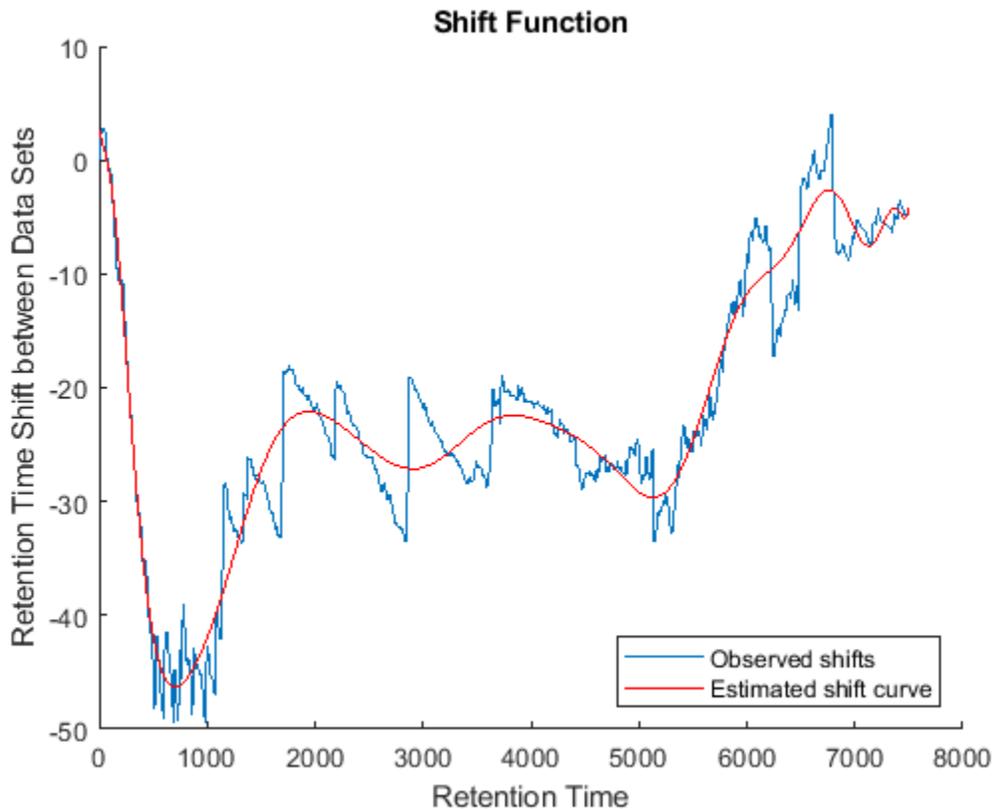
fh6 = figure; hold on
plot((ts(i)+tg(j))/2,ts(i)-tg(j))
title('Shift Function')
xlabel('Retention Time')
ylabel('Retention Time Shift between Data Sets')
```





Because it is expected that the real shift function between both data sets is continuous, you can smooth the observed shifts or regress a continuous model to create a warp model between the two time axes. In this example, we simply regress the drifting to a polynomial function:

```
[p,p_struct,mu] = polyfit((ts(i)+tg(j))/2,ts(i)-tg(j),20);  
sf = @(z) polyval(p,(z-mu(1))./mu(2));  
figure(fh6)  
plot(tg,sf(tg),'r')  
legend('Observed shifts','Estimated shift curve','Location','SouthEast')
```



Comparative Analysis

To carry out a comparative analysis, resample the LC/MS data sets to a common time vector. When resampling we use the estimated shift function to correct the retention time dimension. In this example, each spectrum in both data sets is shifted half the distance of the shift function. In the case of multiple alignment of data sets, it is possible to calculate the pairwise shift functions between all data sets, and then apply the corrections to a common reference in such a way that the overall shifts are minimized [4].

```
t = (max(min(tg),min(ts)):mean(diff(tg)):min(max(tg),max(ts)))';
```

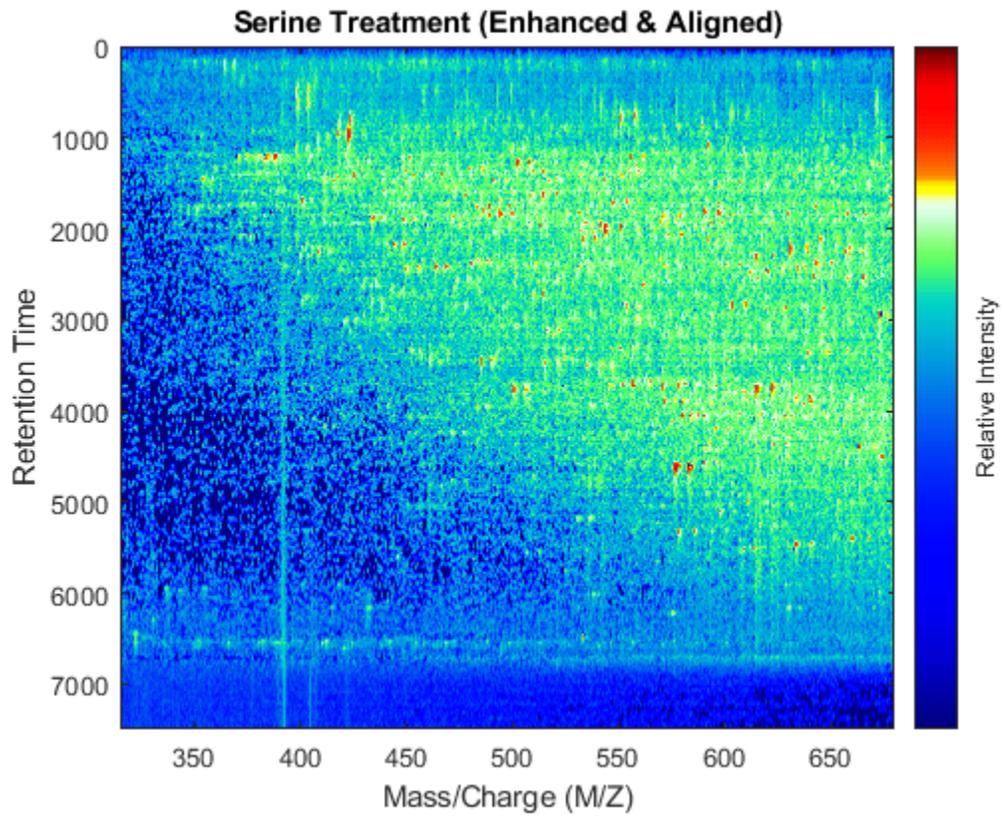
```
% Visualization
```

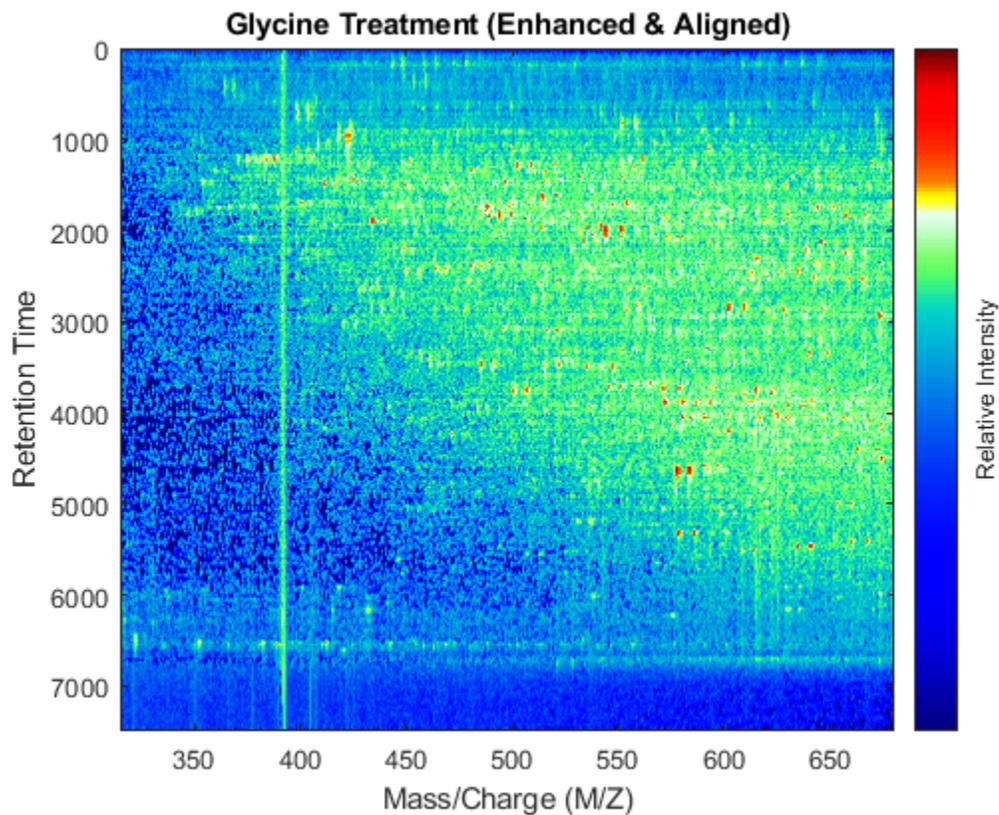
```
fh7 = msheatmap(MZs,t,log(interplq(ts,Ysf',t+sf(t)/2)),'resolution',0.15);
```

```
title('Serine Treatment (Enhanced & Aligned)')
```

```
fh8 = msheatmap(MZg,t,log(interplq(tg,Ygf',t-sf(t)/2)),'resolution',0.15);
```

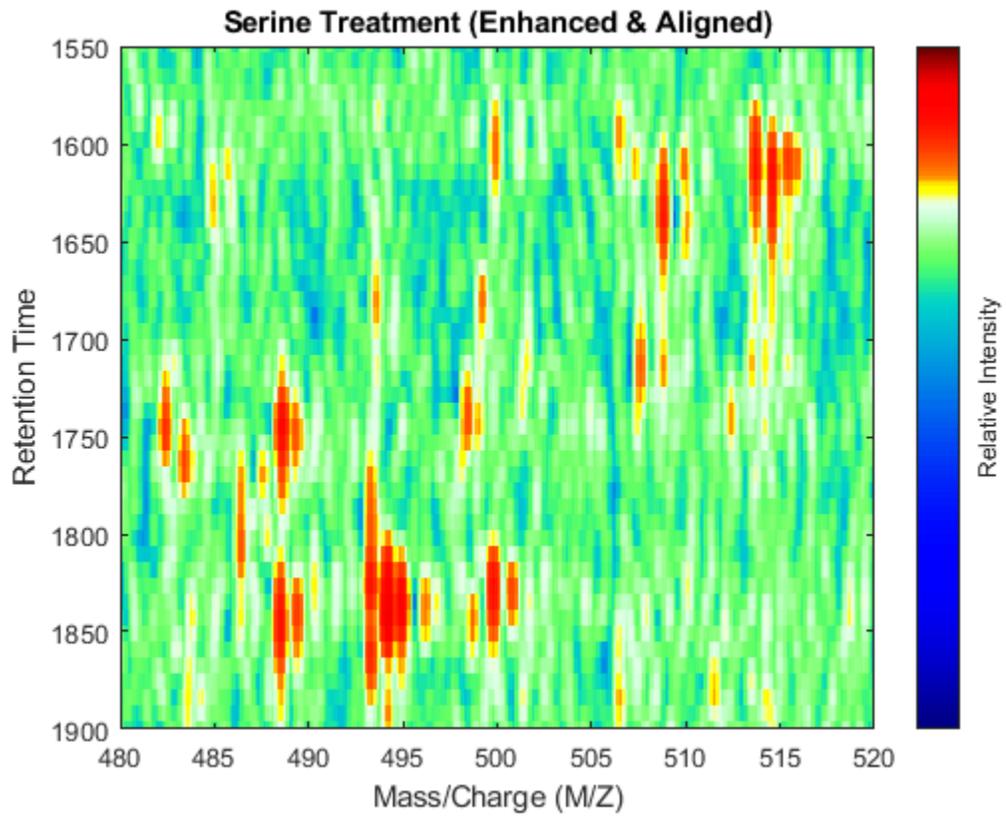
```
title('Glycine Treatment (Enhanced & Aligned)')
```

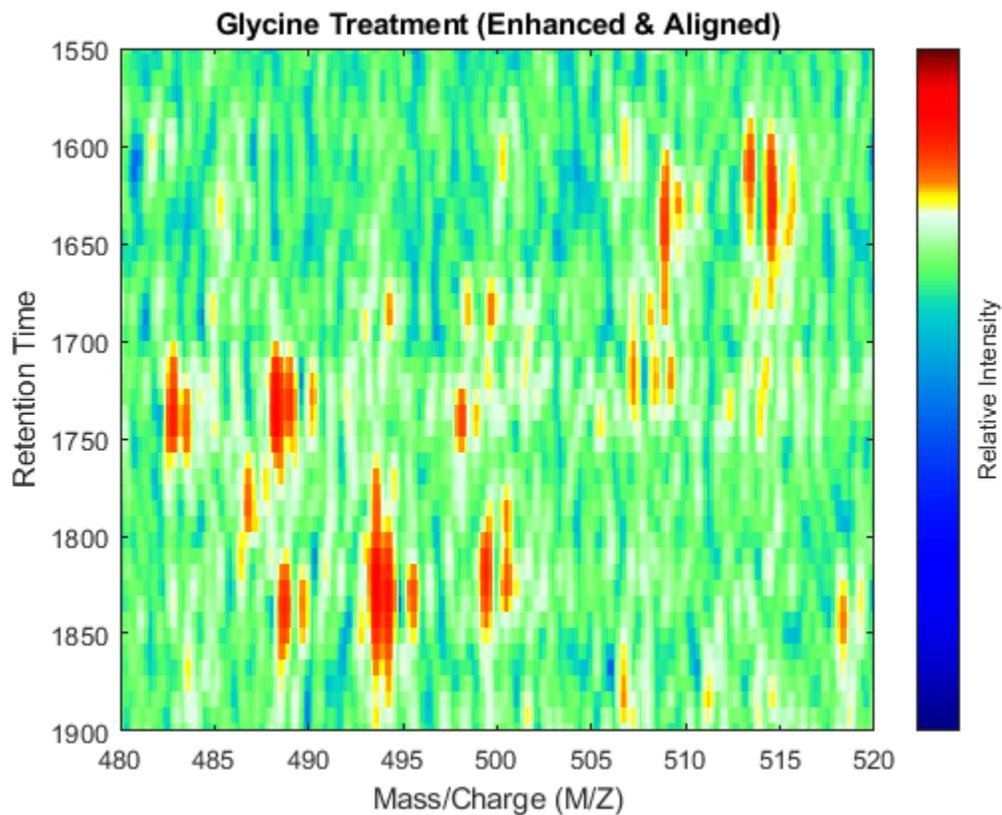




To interactively or programmatically compare regions between two enhanced and time aligned data sets, you can link the axes of two heat maps. Because the heat maps now use a regularly spaced time vector, you can zoom in by using the `AXIS` function without having to search the appropriate row indices of the matrices.

```
linkaxes(findobj([fh7 fh8], 'Tag', 'MSHeatMap'))  
axis([480 520 1550 1900])
```





References

- [1] Listgarten, J. and Emili, A., "Statistical and computational methods for comparative proteomic profiling using liquid chromatography tandem mass spectrometry", *Molecular and Cell Proteomics*, 4(4):419-34, 2005.
- [2] Desiere, F., et al., "The Peptide Atlas Project", *Nucleic Acids Research*, 34:D655-8, 2006.
- [3] Prince, J.T. and Marcotte, E.M., "Chromatographic alignment of ESI-LC-MS proteomics data sets by ordered bijective interpolated warping", *Analytical Chemistry*, 78(17):6140-52, 2006.
- [4] Andrade L. and Manolakos E.S., "Automatic Estimation of Mobility Shift Coefficients in DNA Chromatograms", *Neural Networks for Signal Processing Proceedings*, 91-100, 2003.

Genetic Algorithm Search for Features in Mass Spectrometry Data

This example shows how to use the **Global Optimization Toolbox** with the **Bioinformatics Toolbox™** to optimize the search for features to classify mass spectrometry (SELDI) data.

Introduction

Genetic algorithms optimize search results for problems with large data sets. You can use the MATLAB® genetic algorithm function to solve these problems in Bioinformatics. Genetic algorithms have been applied to phylogenetic tree building, gene expression and mass spectrometry data analysis, and many other areas of Bioinformatics that have large and computationally expensive problems. This example searches for optimal features (peaks) in mass spectrometry data. We will look for specific peaks in the data that distinguish cancer patients from control patients.

Global Optimization Toolbox

First familiarize yourself with the Global Optimization Toolbox. The documentation describes how a genetic algorithm works and how to use it in MATLAB. To access the documentation, use the **doc** command.

```
doc ga
```

Preprocess Mass Spectrometry Data

The original data in this example is from the FDA-NCI Clinical Proteomics Program Databank. It is a collection of samples from 121 ovarian cancer patients and 95 control patients. For a detailed description of this data set, see [1] and [2].

This example assumes that you already have the preprocessed data `OvarianCancerQAQCdataset.mat`. However, if you do not have the data file, you can recreate by following the steps in the example “Batch Processing of Spectra Using Sequential and Parallel Computing” on page 6-77.

Alternatively, you can run the provided script `msseqprocessing.m`.

Load Mass Spectrometry Data into MATLAB®

Once you have the preprocessed data, you can load it into MATLAB.

```
load OvarianCancerQAQCdataset
whos
```

Name	Size	Bytes	Class	Attributes
MZ	15000x1	120000	double	
Y	15000x216	25920000	double	
grp	216x1	25056	cell	

There are three variables: **MZ**, **Y**, **grp**. **MZ** is the mass/charge vector, **Y** is the intensity values for all 216 patients (control and cancer), and **grp** holds the index information as to which of these samples represent cancer patients and which ones represent normal patients. To visualize this data, see the example “Identifying Significant Features and Classifying Protein Profiles” on page 6-38.

Initialize the variables used in the example.

```
[numPoints, numSamples] = size(Y); % total number of samples and data points
id = grp2idx(grp); % ground truth: Cancer=1, Control=2
```

Create a Fitness Function for the Genetic Algorithm

A genetic algorithm requires an objective function, also known as the fitness function, which describes the phenomenon that we want to optimize. In this example, the genetic algorithm machinery tests small subsets of M/Z values using the fitness function and then determines which M/Z values get passed on to or removed from each subsequent generation. The fitness function **biogafit** is passed to the genetic algorithm solver using a function handle. For the implementation details, see the provided script `biogafit.m`. In this example, **biogafit** maximizes the separability of two classes by using a linear combination of 1) the a-posteriori probability and 2) the empirical error rate of a linear classifier (**classify**). You can create your own fitness function to try different classifiers or alternative methods for assessing the performance of the classifiers.

Create an Initial Population

Users can change how the optimization is performed by the genetic algorithm by creating custom functions for crossover, fitness scaling, mutation, selection, and population creation. In this example you will use the **biogacreate** function written for this example to create initial random data points from the mass spectrometry data. For the implementation details, see the provided script `biogacreate.m`. The function header requires specific input parameters as specified by the GA documentation. There is a default creation function in the toolbox for creating initial populations of data points.

Set Genetic Algorithm Options

The GA function uses an options structure to hold the algorithm parameters that it uses when performing a minimization with a genetic algorithm. The **optimoptions** function will create this options structure. For the purposes of this example, the genetic algorithm will run only for 50 generations. However, you may set 'Generations' to a larger value.

```
options = optimoptions('ga','CreationFcn',{@biogacreate,Y,id},...
    'PopulationSize',120,...
    'Generations',50,...
    'Display','iter')
```

```
options =
```

```
ga options:
```

```
Set properties:
```

```
CreationFcn: {1x3 cell}
Display: 'iter'
MaxGenerations: 50
PopulationSize: 120
```

```
Default properties:
```

```
ConstraintTolerance: 1.0000e-03
CrossoverFcn: []
CrossoverFraction: 0.8000
EliteCount: '0.05*PopulationSize'
FitnessLimit: -Inf
FitnessScalingFcn: @fitscalingrank
```

```

FunctionTolerance: 1.0000e-06
HybridFcn: []
InitialPopulationMatrix: []
InitialPopulationRange: []
InitialScoresMatrix: []
MaxStallGenerations: 50
MaxStallTime: Inf
MaxTime: Inf
MutationFcn: []
NonlinearConstraintAlgorithm: 'auglag'
OutputFcn: []
PlotFcn: []
PopulationType: 'doubleVector'
SelectionFcn: []
UseParallel: 0
UseVectorized: 0

```

Run GA to Find 20 Discriminative Features

Use **ga** to start the genetic algorithm function. 100 groups of 20 datapoints each will evolve over 50 generations. Selection, crossover, and mutation events generate a new population in every generation.

```

% initialize the random generators to the same state used to generate the
% published example
rng('default')
nVars = 12; % set the number of desired features
FitnessFcn = {@biogafit,Y,id}; % set the fitness function
feat = ga(FitnessFcn,nVars,options); % call the Genetic Algorithm

feat = round(feat);
Significant_Masses = MZ(feat)

cp = classperf(classify(Y(feat,:),Y(feat,:),id),id);
cp.CorrectRate

```

Single objective optimization:
12 Variables

```

Options:
CreationFcn: @biogacreate
CrossoverFcn: @crossoversscattered
SelectionFcn: @selectionstochunif
MutationFcn: @mutationgaussian

```

Generation	Func-count	Best f(x)	Mean f(x)	Stall Generations
1	240	2.827	8.928	0
2	354	2.827	8.718	1
3	468	0.9663	8.001	0
4	582	0.9516	7.249	0
5	696	0.9516	6.903	1
6	810	0.4926	6.804	0
7	924	0.4926	6.301	1
8	1038	0.02443	5.215	0

9	1152	0.02443	4.77	1
10	1266	0.02101	4.084	0
11	1380	0.02101	3.792	1
12	1494	0.01854	3.437	0
13	1608	0.01606	3.44	0
14	1722	0.01372	2.768	0
15	1836	0.01218	2.74	0
16	1950	0.01204	2.471	0
17	2064	0.01204	2.649	1
18	2178	0.01189	2.326	0
19	2292	0.01189	2.003	0
20	2406	0.0118	2.341	0
21	2520	0.01099	1.714	0
22	2634	0.01094	1.828	0
23	2748	0.01094	1.94	1
24	2862	0.01094	2.285	2
25	2976	0.009843	2.026	0
26	3090	0.009843	1.899	1
27	3204	0.009183	1.802	0
28	3318	0.007877	1.5	0
29	3432	0.007788	1.793	0
30	3546	0.007788	1.756	1

Generation	Func-count	Best f(x)	Mean f(x)	Stall Generations
31	3660	0.007091	1.719	0
32	3774	0.006982	1.598	0
33	3888	0.006982	1.269	1
34	4002	0.006732	1.279	0
35	4116	0.005008	1.229	0
36	4230	0.004325	1.179	0
37	4344	0.004325	1.534	1
38	4458	0.003982	1.15	0
39	4572	0.003982	0.9602	1
40	4686	0.003982	0.8547	2
41	4800	0.003891	0.9083	0
42	4914	0.003683	0.7409	0
43	5028	0.003683	0.516	1
44	5142	0.003364	0.5153	0
45	5256	0.003172	0.4218	0
46	5370	0.003172	0.3783	1
47	5484	0.002997	0.1883	0
48	5598	0.002675	0.1297	0
49	5712	0.002611	0.04382	0
50	5826	0.002519	0.007859	0

ga stopped because it exceeded options.MaxGenerations.

Significant_Masses =

1.0e+03 *

7.6861
7.9234
8.9834
8.6171
7.1808
7.3057
8.1132

```

8.5241
7.0527
7.7600
7.7442
7.7245

```

```
ans =
```

```
1
```

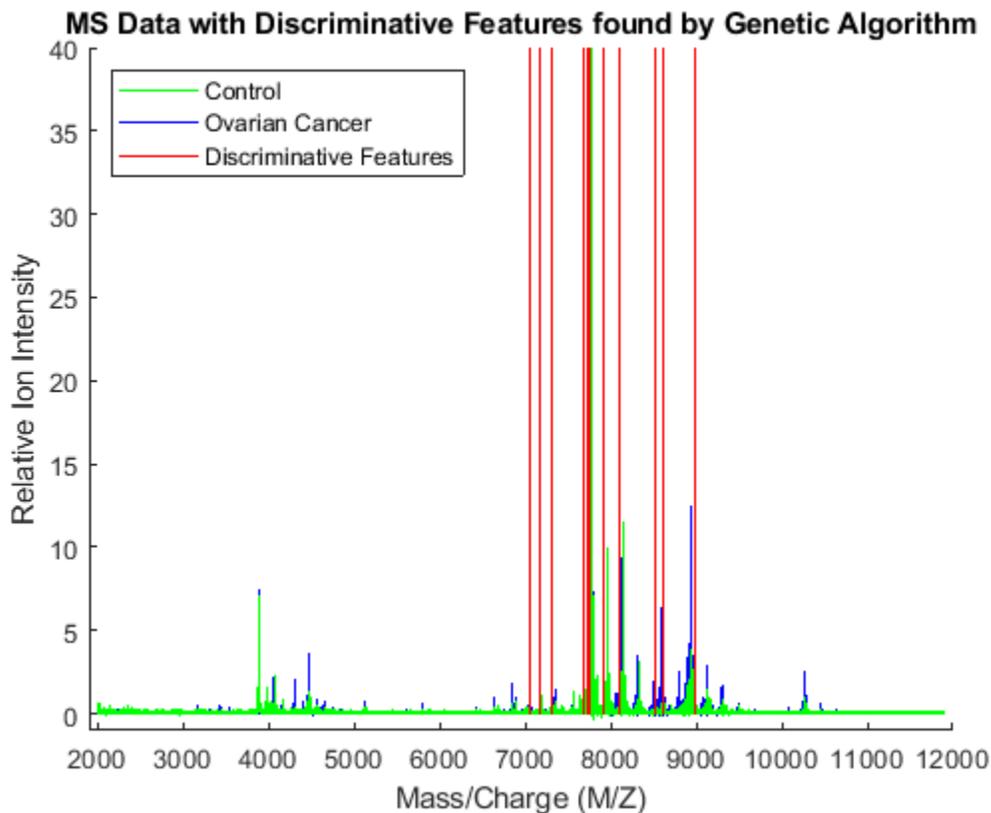
Display the Features that are Discriminatory

To visualize which features have been selected by the genetic algorithm, the data is plotted with peak positions marked with red vertical lines.

```

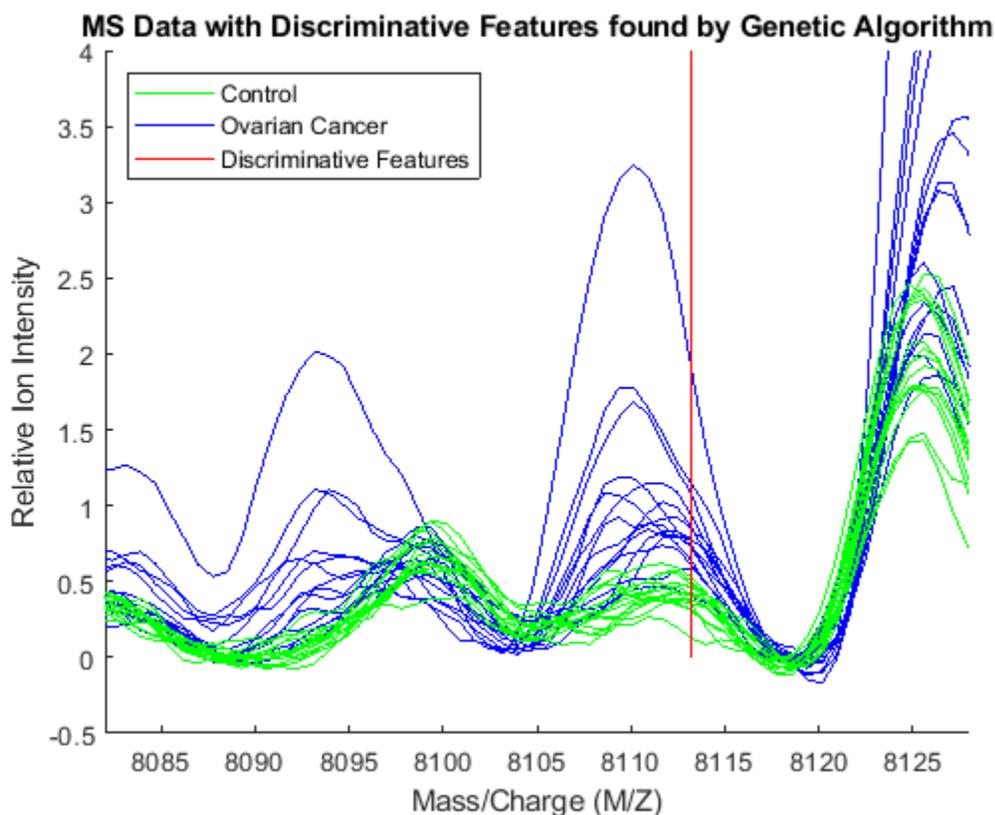
xAxisLabel = 'Mass/Charge (M/Z)';      % x label for plots
yAxisLabel = 'Relative Ion Intensity'; % y label for plots
figure; hold on;
hC = plot(MZ,Y(:,1:15), 'b');
hN = plot(MZ,Y(:,141:155), 'g');
hG = plot(MZ(feats(ceil((1:3*nVars)/3))), repmat([0 100 NaN],1,nVars), 'r');
xlabel(xAxisLabel); ylabel(yAxisLabel);
axis([1900 12000 -1 40]);
legend([hN(1),hC(1),hG(1)], {'Control', 'Ovarian Cancer', 'Discriminative Features'}, ...
      'Location', 'NorthWest');
title('MS Data with Discriminative Features found by Genetic Algorithm');

```



Observe the interesting peak around 8100 Da., which seems to be shifted to the right on healthy samples.

```
axis([8082 8128 -.5 4])
```



References

- [1] Conrads, T P, V A Fusaro, S Ross, D Johann, V Rajapakse, B A Hitt, S M Steinberg, et al. "High-Resolution Serum Proteomic Features for Ovarian Cancer Detection." *Endocrine-Related Cancer*, June 2004, 163-78.
- [2] Petricoin, Emanuel F, Ali M Ardekani, Ben A Hitt, Peter J Levine, Vincent A Fusaro, Seth M Steinberg, Gordon B Mills, et al. "Use of Proteomic Patterns in Serum to Identify Ovarian Cancer." *The Lancet* 359, no. 9306 (February 2002): 572-77.

See Also

msnorm

Related Examples

- "Batch Processing of Spectra Using Sequential and Parallel Computing" on page 6-77
- "Identifying Significant Features and Classifying Protein Profiles" on page 6-38

Batch Processing of Spectra Using Sequential and Parallel Computing

This example shows how you can use a single computer, a multicore computer, or a cluster of computers to preprocess a large set of mass spectrometry signals. Note: Parallel Computing Toolbox™ and MATLAB® Parallel Server™ are required for the last part of this example.

Introduction

This example shows the required steps to set up a batch operation over a group of mass spectra contained in one or more directories. You can achieve this sequentially, or in parallel using either a multicore computer or a cluster of computers. Batch processing adapts to the single-program multiple-data (SPMD) parallel computing model, and it is best suited for Parallel Computing Toolbox™ and MATLAB® Parallel Server™.

The signals to preprocess come from protein surface-enhanced laser desorption/ionization-time of flight (SELDI-TOF) mass spectra. The data in this example are from the FDA-NCI Clinical Proteomics Program Databank. In particular, the example uses the high-resolution ovarian cancer data set that was generated using the WCX2 protein array. For a detailed description of this data set, see [1] and [2].

Setting the Repository for the Data

This example assumes that you have downloaded and uncompressed the data sets into your repository. Ideally, you should place the data sets in a network drive. If the workers all have access to the same drives on the network, they can access needed data that reside on these shared resources. This is the preferred method for sharing data, as it minimizes network traffic.

First, get the name and full path to the data repository. Two strings are defined: the path from the local computer to the repository and the path required by the cluster computers to access the same directory. Change both accordingly to your network configuration.

```
local_repository = 'C:/Examples/MassSpecRepository/OvarianCD_PostQAQC/';
worker_repository = 'L:/Examples/MassSpecRepository/OvarianCD_PostQAQC/';
```

For this particular example, the files are stored in two subdirectories: 'Normal' and 'Cancer'. You can create lists containing the files to process using the DIR command,

```
cancerFiles = dir([local_repository 'Cancer/*.txt'])
normalFiles = dir([local_repository 'Normal/*.txt'])
```

```
cancerFiles =
```

```
121x1 struct array with fields:
```

```
name
folder
date
bytes
isdir
datenum
```

```
normalFiles =  
  
    95x1 struct array with fields:  
  
        name  
        folder  
        date  
        bytes  
        isdir  
        datenum
```

and put them into a single variable:

```
files = [ strcat('Cancer/',{cancerFiles.name}) ...  
         strcat('Normal/',{normalFiles.name})];  
N = numel(files) % total number of files
```

```
N =  
  
    216
```

Sequential Batch Processing

Before attempting to process all the files in parallel, you need to test your algorithms locally with a for loop.

Write a function with the sequential set of instructions that need to be applied to every data set. The input arguments are the path to the data (depending on how the machine that will actually do the work sees them) and the list of files to process. The output arguments are the preprocessed signals and the M/Z vector. Because the M/Z vector is the same for every spectrogram after the preprocessing, you need to store it only once. For example:

type `msbatchprocessing`

```
function [MZ,Y] = msbatchprocessing(repository,files)  
% MSBATCHPROCESSING Example function for BIODISTCOMPDEMO  
%  
% [MZ,Y] = MSBATCHPROCESSING(REPOSITORY,FILES) Preprocesses the  
% spectrogram in files FILES and returns the mass/charge (MZ) and ion  
% intensities (Y) vectors.  
%  
% Hard-coded parameters in the preprocessing steps have been adjusted to  
% deal with the high-resolution spectrograms of the example.  
  
% Copyright 2004-2012 The MathWorks, Inc.  
  
K = numel(files);  
Y = zeros(15000,K); % need to preset the size of Y for memory performance  
MZ = zeros(15000,1);  
  
parfor k = 1:K  
  
    file = [repository files{k}];
```

```

% read the two-column text file with mass-charge and intensity values
fid = fopen(file,'r');
ftext = textscan(fid, '%f%f');
fclose(fid);
signal = ftext{1};
intensity = ftext{2};

% resample the signal to 15000 points between 2000 and 11900
mzout = (sqrt(2000)+(0:(15000-1))*diff(sqrt([2000,11900])))/15000).^2;
[mz,YR] = msresample(signal,intensity,mzout);

% align the spectrograms to two good reference peaks
P = [3883.766 7766.166];
YA = msalign(mz,YR,P,'WIDTH',2);

% estimate and adjust the background
YB = msbackadj(mz,YA,'STEP',50,'WINDOW',50);

% reduce the noise using a nonparametric filter
Y(:,k) = msLowess(mz,YB,'SPAN',5);

% the mass/charge vector is the same for all spectra after the resample
if k==1
    MZ(:,k) = mz;
end
end
end

```

Note inside the function `MSBATCHPROCESSING` the intentional use of `PARFOR` instead of `FOR`. Batch processing is generally implemented by tasks that are independent between iterations. In such case, the statement `FOR` can indifferently be changed to `PARFOR`, creating a sequence of MATLAB® statements (or program) that can run seamlessly on a sequential computer, a multicore computer, or a cluster of computers, without having to modify it. In this case, the loop executes sequentially because you have not created a Parallel Pool (assuming that in the Parallel Computing Toolbox™ Preferences the checkbox for automatically creating a Parallel Pool is not checked, otherwise MATLAB will execute in parallel anyways). For the example purposes, only 20 spectrograms are preprocessed and stored in the `Y` matrix. You can measure the amount of time MATLAB® takes to complete the loop using the `TIC` and `TOC` commands.

```

tic
repository = local_repository;
K = 20; % change to N to do all

[MZ,Y] = msbatchprocessing(repository,files(1:K));

disp(sprintf('Sequential time for %d spectrograms: %f seconds',K,toc))

Sequential time for 20 spectrograms: 7.725275 seconds

```

Parallel Batch Processing with Multicore Computers

If you have Parallel Computing Toolbox™, you can use local workers to parallelize the loop iterations. For example, if your local machine has four-cores, you can start a Parallel Pool with four workers using the default 'local' cluster profile:

```
P00L = parpool('local',4);
```

```
tic
repository = local_repository;
K = 20; % change to N to do all

[MZ,Y] = msbatchprocessing(repository,files(1:K));

disp(sprintf('Parallel time with four local workers for %d spectrograms: %f seconds',K,toc))
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
Parallel time with four local workers for 20 spectrograms: 3.549382 seconds
```

Stop the local worker pool:

```
delete(P00L)
```

Parallel Batch Processing with Distributed Computing

If you have Parallel Computing Toolbox™ and MATLAB® Parallel Server™ you can also distribute the loop iterations to a larger number of computers. In this example, the cluster profile 'compbio_config_01' links to 6 workers. For information about setting up and selecting parallel configurations, see "Cluster Profiles and Computation Scaling" in the Parallel Computing Toolbox™ documentation.

Note that if you have written your own batch processing function, you should include it in the respective cluster profile by using the Cluster Profile Manager. This will ensure that MATLAB® properly transmits your new function to the workers. You access the Cluster Profile Manager using the Parallel pull-down menu on the MATLAB® desktop.

```
P00L = parpool('compbio_config_01',6);
```

```
tic
repository = worker_repository;
K = 20; % change to N to do all

[MZ,Y] = msbatchprocessing(repository,files(1:K));

disp(sprintf('Parallel time with 6 remote workers for %d spectrograms: %f seconds',K,toc))
```

```
Starting parallel pool (parpool) using the 'compbio_config_01' profile ...
Connected to the parallel pool (number of workers: 6).
Parallel time with 6 remote workers for 20 spectrograms: 3.541660 seconds
```

Stop the cluster pool:

```
delete(P00L)
```

Asynchronous Parallel Batch Processing

The execution schemes described above all operate synchronously, that is, they block the MATLAB® command line until their execution is completed. If you want to start a batch process job and get access to the command line while the computations run asynchronously (async), you can manually distribute the parallel tasks and collect the results later. This example uses the same cluster profile as before.

Create one job with one task (MSBATCHPROCESSING). The task runs on one of the workers, and its internal PARFOR loop is distributed among all the available workers in the parallel configuration.

Note that if N (number of spectrograms) is much larger than the number of available workers in your parallel configuration, Parallel Computing Toolbox™ automatically balances the work load, even if you have a heterogeneous cluster.

```
tic % start the clock
repository = worker_repository;
K = N; % do all spectrograms
CLUSTER = parcluster('compbio_config_01');
JOB = createCommunicatingJob(CLUSTER, 'NumWorkersRange', [6 6]);
TASK = createTask(JOB, @msbatchprocessing, 2, {repository, files(1:K)});

submit(JOB)
```

When the job is submitted, your local MATLAB® prompt returns immediately. Your parallel job starts once the parallel resources become available. Meanwhile, you can monitor your parallel job by inspecting the TASK or JOB objects. Use the WAIT method to programmatically wait until your task finishes:

```
wait(TASK)
TASK.OutputArguments
```

```
ans =
```

```
1x2 cell array
    {15000x1 double}    {15000x216 double}
```

```
MZ = TASK.OutputArguments{1};
Y = TASK.OutputArguments{2};
destroy(JOB) % done retrieving the results
disp(sprintf('Parallel time (asynchronous) with 6 remote workers for %d spectrograms: %f seconds', 216, 68.368132))
```

```
Parallel time (asynchronous) with 6 remote workers for 216 spectrograms: 68.368132 seconds
```

Postprocessing

After collecting all the data, you can use it locally. For example, you can apply group normalization:

```
Y = msnorm(MZ, Y, 'QUANTILE', 0.5, 'LIMITS', [3500 11000], 'MAX', 50);
```

Create a grouping vector with the type for each spectrogram as well as indexing vectors. This "labelling" will aid to perform further analysis on the data set.

```
grp = [repmat({'Cancer'}, size(cancerFiles)); ...
       repmat({'Normal'}, size(normalFiles))];
cancerIdx = find(strcmp(grp, 'Cancer'));
numel(cancerIdx) % number of files in the "Cancer" subdirectory
```

```
ans =
```

```
121
```

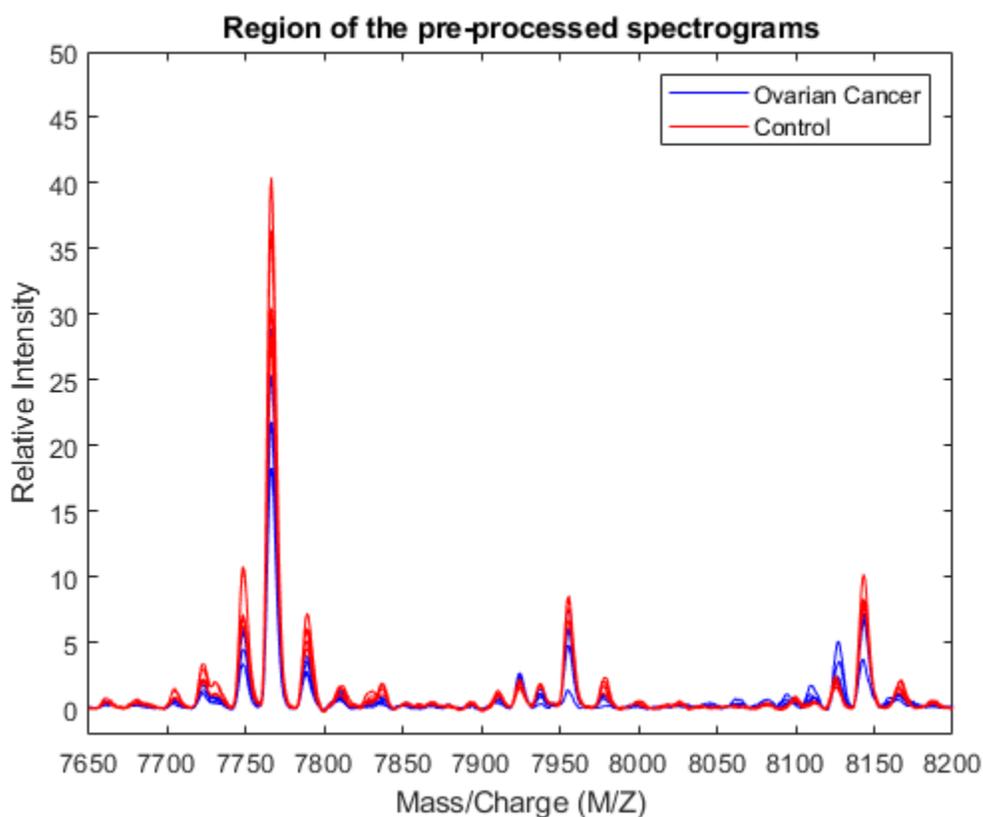
```
normalIdx = find(strcmp(grp, 'Normal'));
numel(normalIdx) % number of files in the "Normal" subdirectory
```

```
ans =
```

```
95
```

Once the data is labelled, you can display some spectrograms of each class using a different color (the first five of each group in this example).

```
h = plot(MZ,Y(:,cancerIdx(1:5)), 'b',MZ,Y(:,normalIdx(1:5)), 'r');  
axis([7650 8200 -2 50])  
xlabel('Mass/Charge (M/Z)');ylabel('Relative Intensity')  
legend(h([1 end]),{'Ovarian Cancer','Control'})  
title('Region of the pre-processed spectrograms')
```



Save the preprocessed data set, because it will be used in the examples “Identifying Significant Features and Classifying Protein Profiles” on page 6-38 and “Genetic Algorithm Search for Features in Mass Spectrometry Data” on page 6-71.

```
save OvarianCancerQAQCdataset.mat Y MZ grp
```

Disclaimer

TIC - TOC timing is presented here as an example. The sequential and parallel execution time will vary depending on the hardware you use.

References

- [1] Conrads, T P, V A Fusaro, S Ross, D Johann, V Rajapakse, B A Hitt, S M Steinberg, et al. "High-Resolution Serum Proteomic Features for Ovarian Cancer Detection." *Endocrine-Related Cancer*, June 2004, 163-78.
- [2] Petricoin, Emanuel F, Ali M Ardekani, Ben A Hitt, Peter J Levine, Vincent A Fusaro, Seth M Steinberg, Gordon B Mills, et al. "Use of Proteomic Patterns in Serum to Identify Ovarian Cancer." *The Lancet* 359, no. 9306 (February 2002): 572-77.

See Also

`msnorm` | `msresample` | `msbackadj` | `mslowess` | `msalign`

