# **Matrix factorization and systems of linear equations.**

**LU and LDL factorization.** At the end of Gaussian elimination the matrix multiplication:

has changed into .

The original matrix has become upper triangular: the number highlighted in bold red in the final matrix are called the *pivots*. Now, notice how the original matrix can be represented as the product of a *lower triangular matrix* and the *upper triangular matrix* derived from Gaussian elimination:

the number in bold red in the lower triangular matrix are the *row multipliers* we have used in the Gaussian elimination. The factorization of a matrix ***A*** into the product of two triangular (**L**ower and **U**pper) matrices is called the **LU** factorization. This LU factorization is often used instead of the Gaussian elimination for the solution of large systems of linear equations. For example, suppose the matrix ***A*** for the linear system ***Ax = b*** can be factorized into an ***LU*** product; then the solution to ***L***(***Ux***) ***= b*** can be obtained by defining ***y = Ux*** and by solving two systems in sequence:

1. ***Ly = b*** (solve for **y** by forward substitution)

2. ***Ux = y*** (solve for **x** by backward substitution)

The solution to the equation: ***L y = b***

becomes:

Forward substitution phase (***Ly = b***):

***L y = b***

***y***

Backward substitution phase (***Ux = y***):

***U x = y***

The advantage of the **LU** decomposition over the Gauss elimination is that once ***A*** is decomposed into ***LU***, we can solve ***Ax = b*** for as many ***b*** vectors as we like. It is also normal practice to store the elements of both ***U*** and ***L*** in a single matrix ***L\U*** since it is understood that each of the diagonal elements of ***L*** is 1. For example, the ***L\U*** matrix for ***A*** is:

We provide two MATLAB functions lu\_fact.m (LU factorization of a square matrix without row exchanges) and lu\_solv.m (LU based solver of systems of linear equations) to solve the matrix equation ***Ax*** *=* ***b***. The solution of the system is obtained by running the two functions in sequence:

LU = lu\_fact(A)

x = lu\_solv(LU,b)

Alternatively we can use MATLAB intrinsic function *lu*:

[L,U] = lu(sym(A)) % symbolic factorization

[L,U,P] = lu(A) % numerical factorization

P\*A , L\*U

inv(P)\*L\*U

where ***P*** is a permutation matrix such that the product ***PA*** is equal to the product ***LU***. The solution to ***Ax*** *=* ***b*** is normally obtained with matrix division using the *backslash operator*:

x = A\b

However, this operator and also the more complex function *linsolve* actually work by solving the two triangular systems:

y = L\(P\*b)

x = U\y

x = U\(L\(P\*b));

MATLAB offers also the function *ilu* for *incomplete LU factorization*. This function is most often used in the implementation of *iterative methods*, but is also useful to obtain a factorization without permutation of the rows:

setup.type = 'nofill';

[L,U] = ilu(sparse(A),setup)

U = full(U), L = full(L)

y = L\b

x = U\y

x = U\(L\b);

Very often matrices that arise in biochemical application are symmetric (i.e., the *covariance matrix* (CHAPTER 10)). If a matrix ***A*** is symmetric, then the **LU** factorization can be represented in the form:

***A*** *=* ***LU*** *=* ***LDLT***

where ***D*** is a diagonal matrix. For example consider the matrix ***A***:

in this case:

Notice how ***U*** can be obtained directly by left multiplying the transpose of ***L*** by a diagonal matrix ***D*** containing the pivots:

***U*** ***D***  ***LT***

Thus, the **LDLT** factorization yields:

***L*** ***D LT***

Based on this information, we can easily obtain the **LDLT** factorization of the symmetric matrix:

A = [2 3 1;3 1 1;1 1 2];

using the *lu\_fact* function and the following matlab commands:

LU = lu\_fact(A); U = triu(LU);

L = tril(lu\_fact(A),-1)+eye(3);

D = diag(diag(U));

or directly using matlab *ldl* function:

[L,D] = ldl(A)

Clearly, we can use the LDL factorization to solve the matrix equation ***Ax*** *=* ***b***. In this case the solution to ***L***(***DLTx***) ***= b*** can be obtained by defining ***DLTx = y*** and by solving two systems in sequence:

1. ***Ly = b*** (solve for **y** by forward substitution)

2. ***DLTx = y*** ⇒ ***x = (DLT)-1y*** ⇒ ***x =*** (***LT***)***-1D-1y*** (solve for ***x*** by backward substitution)

These two steps correspond to the matlab commands:

y = inv(L)\*b

y = L\b

x = inv(L')\*(inv(D)\*y)

x = L'\(D\(L\b))

**SPECIAL TOPIC**

**Iterative methods.** When the system of linear equations ***Ax = b*** becomes too large for ordinary Gaussian elimination a new class of methods becomes useful to find the solution: these are the *iterative methods*. In general, these methods consist of two steps:

1. Preconditioning of ***A***: we look for a matrix ***S*** which is close to ***A***, but simpler to work with.

2. Iteration: knowing the matrix difference ***T = S-A*** ⇒ ***S-T = A*** we replace ***A*** with ***S-T***, so we can rewrite the system of equations as:

We start with a guess (i.e., a vector of 0s or 1s) and we solve iteratively:

⇓

⇓ ... and so on

until the *residual* ***r*** becomes less than an *error* we are willing to tolerate (ideally 0):

Of course an essential part of the method success is that the *preconditioner* ***S*** should allow an easy solution of the modified system. Some popular choices for ***S*** are:

***J*** = diagonal part of ***A*** (this is the *Jacobi* method).

***GS*** = lower triangular part of ***A*** including the diagonal (this is the *Gauss-Seidel* method).

***ILU*** = (approximate lower triangular) x (approximate upper triangular): this is the *incomplete LU* method based on the function *ilu*. The incomplete **LU** factorization produces ***L*** and ***U*** *sparse* matrices in which the small terms are replaced with 0s.

In general these methods work best for *symmetric* matrices in which the elements on the diagonal are larger than the non-diagonal elements, and may fail for *non-symmetric* matrices. As an example, consider the matrix equation:

**A x** = **b**

A = [2 3 1;3 1 1;1 1 2]; A = A'\*A;

b = [3 2 1]'; x = A\b

and apply the simple functions *gauss\_seidel* or ilu\_iter\_solv:

function [ x,r,S,T,iter ] = gauss\_seidel( A,x0,b,tol )

S = tril(A);

T = S-A;

r = b-A\*x0;

iter = 0;

while norm(r) > tol

iter = iter + 1;

x1 = S\(T\*x0+b);

r = b-A\*x1;

x0 = x1;

end

x = x1;

end

tol = eps^(1/3); x0 = [0 0 0]';

[ x,r,S,T,iter ] = gauss\_seidel(A,x0,b,tol)

[ x,r,S,T,iter ] = ilu\_iter\_solv(A,x0,b,tol,0.1)

While useful to understand the general principle behind *iterative methods* the *Gauss-Seidel* and *incomplete LU* methods are considered obsolete to solve large systems of linear equations. MATLAB has several advanced functions implementing *iterative methods*, that are optimized for different types of matrices:

*Conjugate gradients* (symmetric *positive definite* matrices):

[x,flag,relres,iter,resvec] = cgs(A,b)

*Biconjugate gradients* (symmetric *positive definite* matrices):

[x,flag,relres,iter,resvec] = bicg(A,b)

*Generalized minimum residual method* (square non-symmetric matrices):

[x,flag,relres,iter,resvec] = gmres(A,b)

*Minimum residual method* (symmetric but not necessarily *positive definite* matrices):

[x,flag,relres,iter,resvec] = minres(A,b)

Because of their importance in current computational strategies to find the *minimum* of functions, we describe here in more detail the *conjugate gradients* methods, which are specifically designed to work with *positive definite* matrices. This is a very important class of *symmetric* matrices that will be discussed in a later chapter, but for now it suffices to say that one important features of these matrices is that upon Gaussian elimination (or LU factorization) all the *pivots* are *positive*, and the product is a *positive scalar* for any vector:

Consider the *quadratic* scalar function:

where ***A*** is *symmetric* and *positive definite*.

We want to find the value of for which the function acquires its minimum value. Differentiating with respect to ***x*** we obtain the *gradient* of :

Because is at its minimum when (*First Order Optimality Condition*, FOC), *minimizing* is the same as solving.

Here we obtain a positive definite matrix ***A***:

A = [2 3 1;3 1 1;1 1 2]; A = A'\*A

b = [3 2 1]'

x = A\b

Here we represent as isosurfaces:

x1 = [-.3:.05:0.6]; x2 = [-.4:.05:0.4]; x3 = [-.5:.05:0.5];

npoints1 = length(x1);

npoints2 = length(x2);

npoints3 = length(x3);

X = zeros(3,npoints1\*npoints2\*npoints3);

E = zeros(1,npoints1\*npoints2\*npoints3);

x and y number of points are inverted in this array to match the ordering produced by meshgrid later on

E3 = zeros(npoints2,npoints1,npoints3);

n = 0;

for i = 1:npoints1

x = x1(i);

for j = 1:npoints2

y = x2(j);

for k = 1:npoints3

z = x3(k);

n = n+1;

X(:,n) = [x;y;z];

E(n) = 0.5\*X(:,n)'\*A\*X(:,n) -b'\*X(:,n);

i and j indices are inverted here to match the ordering produced by meshgrid

E3(j,i,k) = E(n);

end

end

end

Isosurface plot

Energy\_Function = figure;

set(gcf,'Unit','Normalized','Position',[0 0 0.6 0.8]);

[Xg, Yg, Zg] = meshgrid(x1,x2,x3);

p0 = patch(isosurface(Xg,Yg,Zg,E3,1.8));

isonormals(Xg,Yg,Zg,E3,p0)

p0.FaceColor = 'red';

p0.EdgeColor = 'none';

p0.FaceAlpha = 0.05;

grid on

box on

hold on

[Xg, Yg, Zg] = meshgrid(x1,x2,x3);

p1 = patch(isosurface(Xg,Yg,Zg,E3,.9));

isonormals(Xg,Yg,Zg,E3,p1)

p1.FaceColor = 'red';

p1.EdgeColor = 'none';

p1.FaceAlpha = 0.075;

p2 = patch(isosurface(Xg,Yg,Zg,E3,.3));

isonormals(Xg,Yg,Zg,E3,p2)

p2.FaceColor = 'red';

p2.EdgeColor = 'none';

p2.FaceAlpha = 0.15;

p3 = patch(isosurface(Xg,Yg,Zg,E3,-.1));

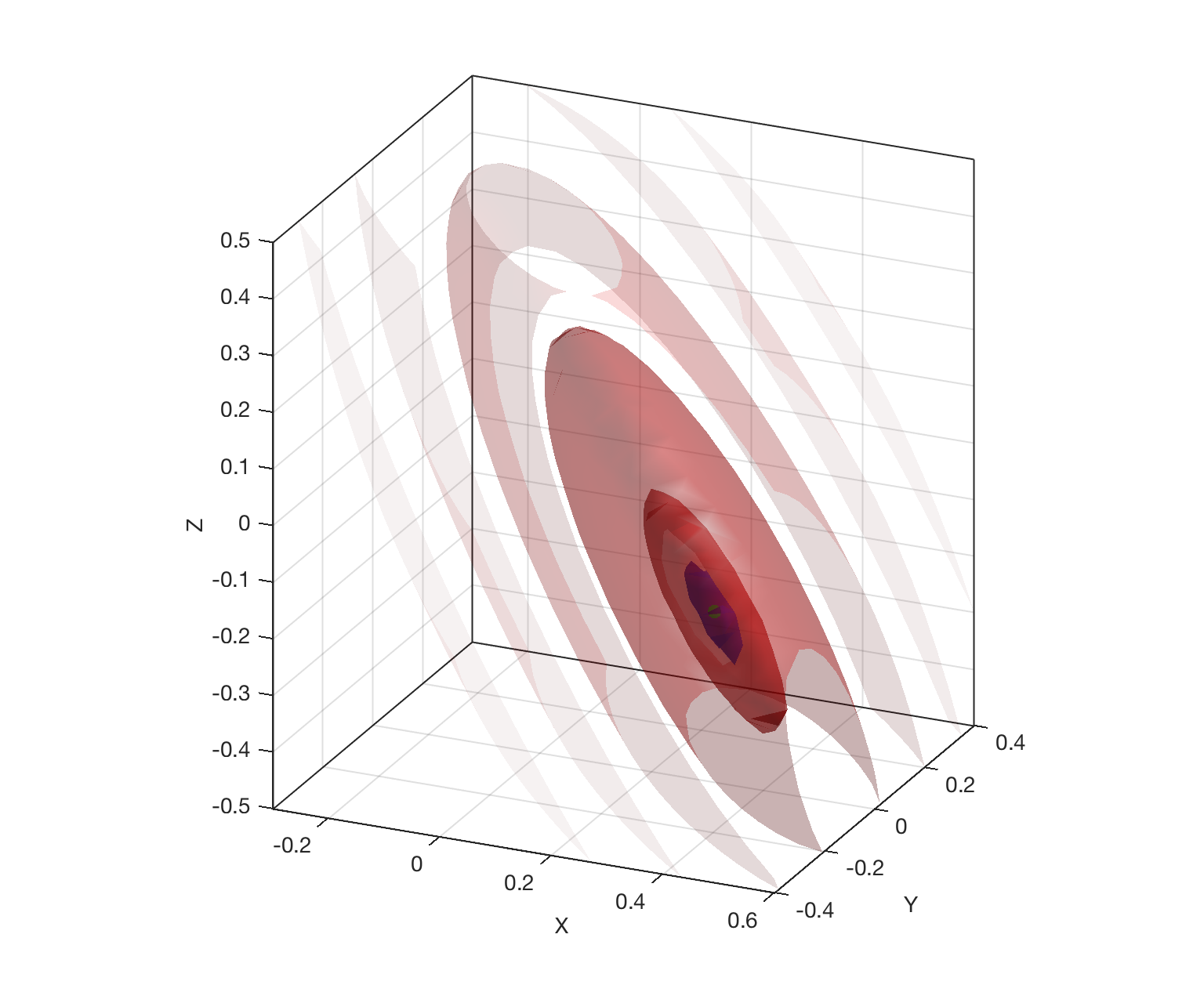
isonormals(Xg,Yg,Zg,E3,p3)

p3.FaceColor = 'red';

p3.EdgeColor = 'none';

p3.FaceAlpha = 0.3;

p4 = patch(isosurface(Xg,Yg,Zg,E3,-.32));

isonormals(Xg,Yg,Zg,E3,p4)

p4.FaceColor = 'red';

p4.EdgeColor = 'none';

p4.FaceAlpha = 0.4;

p5 = patch(isosurface(Xg,Yg,Zg,E3,-.36));

isonormals(Xg,Yg,Zg,E3,p5)

p5.FaceColor = 'b';

p5.EdgeColor = 'none';

p5.FaceAlpha = 0.45;

daspect([1,1,1])

view(24.1,22); axis tight

camlight

lighting gouraud

xlabel('X');ylabel('Y');zlabel('Z');

Function minimum

Emin = 0.5\*x'\*A\*x -b'\*x;

scatter3(x(1),x(2),x(3),80,'g','filled');

Gradient methods start with an initial vector and compute a refined solution at each cycle:

⇓

⇓ ... and so on

The *step length* is chosen so that minimizes in the *search direction* . This means that must *tend to* satisfy:

Introducing the *residual*:

and pre-multiplying both sides by we find :

The obvious choice for the search direction is the negative of the gradient:

This choice corresponds to the method of *steepest descent* (as it wipes out the residual for the current value of ). However, while providing the correct result, this method converges very slowly, because consecutive searches tend to be in approximately the same direction. To avoid this problem, the *conjugate gradient* method uses a modified search direction:

The constant is chosen so that the two successive search directions are *not interfering* with each other; these directions are called *conjugate* or ***A*** *normal* because:

Substituting for we obtain:

The following is a flowchart of the *conjugate gradients* algorithm:

Start with any ***x0***

**⇓**

Use the *steepest descent* as 1st step

**⇓**

**⇓**

Loop with *k = 1 to n* where *n*

is the number of variables*.*

**⇓**

If exit loop (convergence criterion = error tolerance is met)

which can be easily implemented as a small function:

function [ x ] = conj\_grad(A,x,b,tol)

if nargin<4

tol = 1e-9;

end

n = length(b);

r = b - A\*x;

s = r;

for i = 1:n

As = A\*s;

alpha = (s'\*r)/(s'\*As);

x = x + alpha\*s;

r = b - A\*x;

if norm(r) <= tol

return

else

beta = -(r'\*As)/(s'\*As);

s = r + beta\*s;

end

end

x0 = [0 0 0]'

[x,r,i] = conj\_grad(A,x0,b)

It can be shown that not only the search direction ***s****1,* ***s****2,* ***s****3, . . .* ***s****n* are *conjugate*, but that the residual vectors ***r****1,* ***r****2,* ***r****3, . . .* ***r****n,* are *mutually orthogonal*:

In terms of speed, the conjugate gradients method is not as efficient as direct methods like Gaussian elimination in the solution of small sets of equations. However, it becomes unbeatable with large, sparse systems (where most of the elements of ***A*** are zero). Furthermore, since it reaches the exact solution in *n* cycles, it's not really an iterative method. In practice, with large systems of equations, and if the starting vector is not too far from the solution, convergence is usually achieved in even less than *n* iterations.

We will return to the *conjugate gradients* method in CHAPTER 17, when we discuss current strategies for unconstrained minimization of non-linear functions.

**PRACTICE**

**1.** Use the two provided MATLAB functions lu\_fact.m (LU factorization of a square matrix) and lu\_solv.m (LU based solver of systems of linear equation) in sequence to solve the matrix equation Ax = b, where:

A = [3 7 8;1 4 7;2 2 4];

b = [59 34 22]';

**2.** Carry out the LDL' factorization of the symmetric matrix:

A = [2 3 1;3 1 1;1 1 2];

and show that L\*U and L\*D\*L' give indeed the same result.

**3.** Using code lines from lu\_solv.m write a MATLAB program that will solve the matrix equation Ax = b with:

b = [5 7 3]';

using the LDL' factorization. Remember that in this case the logical steps of the solution are going to be:

1. Ly = b

2. DL'x = y => L'x = y/D

These correspond to the MATLAB commands:

y = inv(L)\*b

x = inv(L')\*(inv(D)\*y)