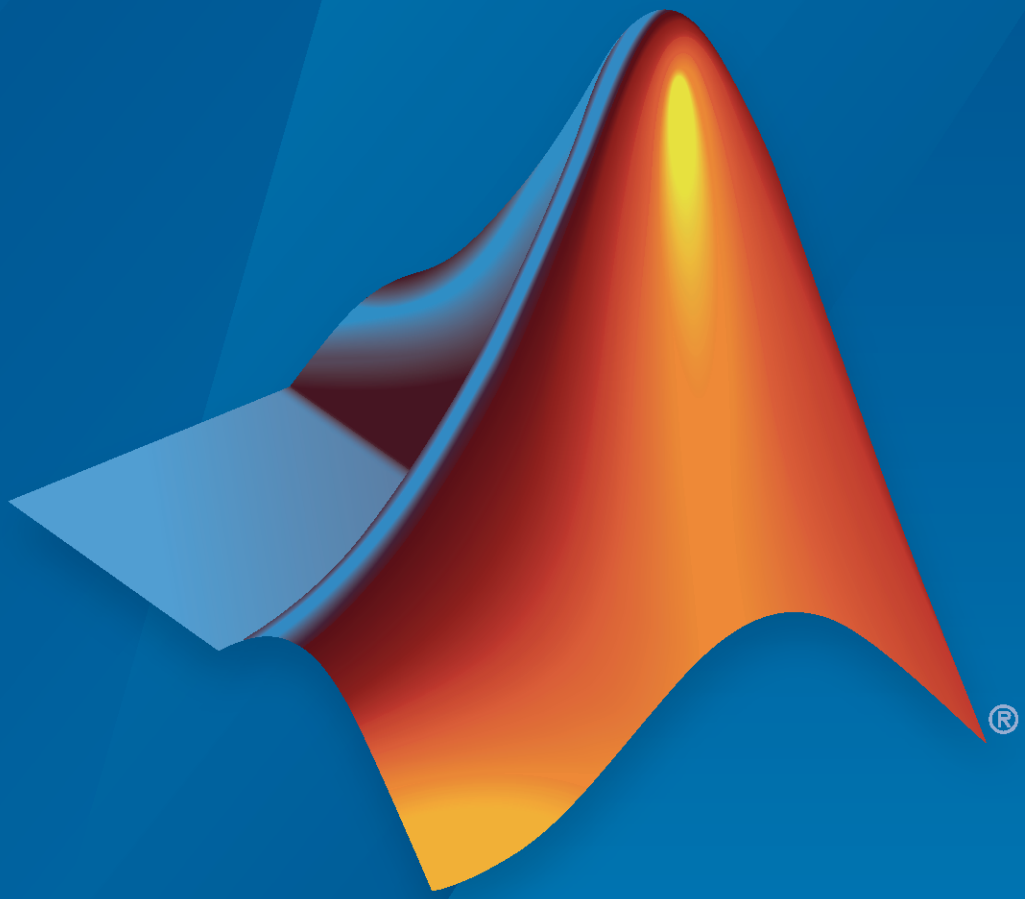
Optimization Toolbox™ Release Notes



MATLAB®

MathWorks®

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

Optimization Toolbox™ Release Notes

Contents

R2026a

R2025b

R2025a

R2024b

R2024a

R2023b

R2023a

R2022b

R2022a

R2021b

R2021a

R2026a

★Optimization Explorer App: Search for global solution to nonlinear optimization problem using multiple solver configurations

To search for a global solution to a nonlinear optimization problem, the Optimization Explorer app repeatedly runs optimization solvers from MATLAB®, Optimization Toolbox, and (if licensed) Global Optimization Toolbox on a problem. The app can choose multiple solver configurations automatically, meaning multiple solvers, options, and initial points. Or you can manually specify solver configurations to run. Save time by running the app in parallel (requires Parallel Computing Toolbox™).

The app has a variety of built-in plots for visualizing its progress and aspects of the results. You can sort and filter the results based on multiple criteria such as objective function value, solver run time, and feasibility. Export results as a table or as a function that reproduces a selected result.

For an example, see “Use Optimization Explorer App”. For details, see the Optimization Explorer app reference page.

Range constraints for linprog and intlinprog

You can now specify both upper and lower ranges for linear inequality constraints in `linprog` and `intlinprog`:

$$\text{bl} \le Ax \le b\,.$$

To do so, express the linear inequality as a two-element cell array `{bl,b}`, where `bl` is a vector of lower range constraints, and `b` is a vector of upper range constraints.

The `lambda` output of `linprog` contains a field named `ineqlinLower` that is associated with the `bl` input. The `ineqlin` field of the `lambda` output applies to the `b` input whether the input is a cell or a real vector.

For details, see the reference pages and “Linear Range Constraints”.

Sensitivity for linprog

The default `linprog` `"dual-simplex-highs"` algorithm can return sensitivity information in the sixth output.

```
[x,fval,exitflag,output,lambda,sensitivity] = linprog(...)
```

The function returns the sensitivity as a `SensitivityAnalysis` object. For an example, see “Sensitivity Analysis in Linear Programming”. For details, see the reference pages.

Run MATLAB code identical to generated code in lsqcurvefit, lsqnonlin, and fsolve

You can now use the `lsqcurvefit`, `lsqnonlin`, and `fsolve` solvers to run the identical code in MATLAB as the code they generate for target hardware. To do so, set the `UseCodegenSolver` option to `true` and the `Algorithm` option to `"levenberg-marquardt"` using `optimoptions`.

```
options = optimoptions("fsolve",Algorithm="levenberg-marquardt",...
    UseCodegenSolver=true); % Similarly for lsqcurvefit or lsqnonlin
```

These options allow you to test code in the desktop environment before generating code for target hardware. Although the generated code is identical to the MATLAB code, results can differ slightly because linked math libraries can vary. For details, see the function reference pages.

Functionality being removed or changed

intlinprog "legacy" algorithm removed

Errors

The `intlinprog` `"legacy"` algorithm has been removed. All options specific to the `"legacy"` algorithm have also been removed:

- `BranchRule`
- `CutGeneration`
- `CutMaxIterations`
- `Heuristics`
- `HeuristicsMaxNodes`
- `IntegerPreprocess`
- `IntegerTolerance`
- `LPMaxIterations`
- `LPOptimalityTolerance`
- *`LPPreprocess`*
- `NodeSelection`
- `ObjectiveImprovementThreshold`
- `RootLPAlgorithm`
- `RootLPMaxIterations`

Also, because there is only one `intlinprog` algorithm, the `Algorithm` option has been removed. Similarly, the `intlinprog` `output` structure no longer includes an `algorithm` field.

The `optimValues` structure, used in output functions or plot functions (see “intlinprog Output Function and Plot Function Syntax”), no longer includes a `phase` field because that field was used only by the `"legacy"` algorithm.

CheckGradients option removed

Errors

The `CheckGradients` option has been removed from all nonlinear gradient-based solvers:

- `fmincon`
- `fminunc`
- `fseminf`
- `fsolve`
- `lsqcurvefit`
- `lsqnonlin`

To check that your supplied gradient function is correct, use the `checkGradients` function. For example,

```
function [f,g] = rosen(x)
f = 100*(x(1) - x(2)^2)^2 + (1 - x(2))^2;
if nargout > 1
    g(1) = 200*(x(1) - x(2)^2);
    g(2) = -400*x(2)*(x(1) - x(2)^2) - 2*(1 - x(2));
end
end
% Before using the rosen function,
% you can check that the gradient is correct at a point
x0 = [2,4];
assert(checkGradients(@rosen,x0))
```

Constraint violations now include bound and variable type

The `infeasibility` and `issatisfied` functions now check the feasibility with respect to bound constraints, and take into account variable type for integer, semi-integer, and semi-continuous variables.

Variable Type	Infeasibility for Bounds and Type
Continuous	`max(0,lb - x,x - ub)`
Integer	`max(0,lb - x,x - ub,abs(x - round(x)))`
Semi-continuous	`min(max(0,lb - x,x - ub),abs(x))`
Semi-integer	`min(max(0,lb - x,x - ub,abs(x - round(x))),abs(x))`

The `issatisfied` function returns two arguments, `allsat` and `sat`.

- `allsat` is a vector with `true` values where all constraints are satisfied to within the given tolerance, including bound constraints and variable types.
- `sat` is an `OptimizationValues` array with logical `Variables` properties that reflect the bound and variable type satisfaction of each variable. Previously, the `Variables` property held the variable values in the `pts` input.

Presolve option for linprog "interior-point" algorithm

You can specify the level of presolving for the `linprog` `"interior-point"` algorithm by setting the *Presolve* option to `"auto"` (default), `"off"`, or `"on"`. This option is also available for code generation. The *Presolve* option remains a "hidden" option, meaning `optimoptions` does not show the option or its settings. See “Hidden Options”.

Change in behavior for loading options from previous releases
Behavior change

When you open `optimoptions` options from a previous software release, the resulting options can change. The current behavior is:

1. Create a new default options object for the selected solver.
2. Copy over the option settings that you specified in the old object.
   - If an option no longer exists, ignore the setting.

- If an option value is no longer available, issue a warning.

Nonlinear constraint notation change
Behavior change

Nonlinear constraint functions have been renamed from `[c,ceq]` to `[ineqnonlin,eqnonlin]` in the descriptions and examples. For example, to constrain a solution to the interior of a circle of radius one, the documentation now shows using this function as the nonlinear constraint:

```
function [ineqnonlin,eqnonlin] = circlecons(x)
eqnonlin = [];
ineqnonlin = norm(x)^2 - 1;
end
```

Previously, the function was:

```
function [c,ceq] = circlecons(x)
ceq = [];
c = norm(x)^2 - 1;
end
```

R2025b

Version: 25.2

Bug Fixes

Quality and stability improvements

R2025b delivers quality and stability improvements, building on the new features introduced in R2025a.

R2025a

★Code Generation for linprog: Generate C or C++ code for problems with linear objective functions and linear constraints

The `linprog` function now supports code generation. Also, `linprog` now supports both full and sparse input matrices for code generation, such as a linear inequality constraint matrix `A` or a lower bound matrix `lb`. For details, see Code Generation for `linprog` Background and Generate Code for `linprog`.

★Extended integer variable support in intlinprog

The `intlinprog` solver with the default `"highs"` algorithm supports semicontinuous and semi-integer variables. Semicontinuous variables can take the value `0` or any real value between the lower and upper bounds, both of which must be strictly positive and no more than `1e5`. Similarly, semi-integer variables can take the value `0` or any integer value between the lower and upper bounds, both of which must be strictly positive and no more than `1e5`. Specify the indices of the extended integer variables using the `integerConstraint` function.

For the problem-based approach, specify the variable type in `optimvar` when creating variables.

```
x = optimvar("x",Type="semi-continuous",...
    LowerBound=3,UpperBound=10)
y = optimvar("y",2,Type="semi-integer",...
    LowerBound=[3,5],UpperBound=[10,20])
```

For details, see the reference pages.

Run MATLAB Code identical to generated code

The `fmincon`, `lsqlin`, and `quadprog` solvers can now run the identical code in MATLAB as the code they generate for target hardware. To do so, set the `UseCodegenSolver` option to `true` using `optimoptions`:

```
options = optimoptions("fmincon",Algorithm="sqp",...
    UseCodegenSolver=true); % Similarly for lsqlin or quadprog
```

This option allows you to test code in the desktop environment before generating code for target hardware. Even though the generated code is identical to the MATLAB code, results can differ slightly because linked math libraries can differ. For details, see the function reference pages.

ScaleProblem Option for quadprog

The `"interior-point-convex"` algorithm of the `quadprog` solver gains the `ScaleProblem` option, which can speed the solution of some poorly-scaled problems. For details, see the `quadprog` `options` argument description.

⚠Functionality being removed or changed

linprog "dual-simplex-legacy" algorithm has been removed
Errors

The `linprog` `"dual-simplex-legacy"` algorithm has been removed. Usually, the `"dual-simplex-highs"` algorithm (default) performs better than the former `"dual-simplex-legacy"` algorithm.

Different preprocessing for linprog "interior-point" algorithm
Behavior change

The `linprog` `"interior-point"` algorithm now uses HiGHS preprocessing (also called presolve) instead of its previous preprocessing code. In most tested cases, `linprog` has improved speed with negligible changes in results when using HiGHS preprocessing.

If your problem fails to solve quickly or has a nonpositive exit flag when using HiGHS preprocessing, try loosening the `OptimalityTolerance` or `ConstraintTolerance` options. For example, try setting one or both to `1e-5` using `optimoptions`. In almost all tested cases, this adjustment allows the `linprog` `"interior-point"` algorithm to solve problems quickly and accurately.

Generated code for coneprog changes
Behavior change

`coneprog` now supports both full and sparse input matrices for code generation, such as a linear inequality constraint matrix `A`, a lower bound matrix `lb`, or sparse matrices as inputs to the `secondordercone` function.

When using sparse data, generated code for `coneprog` can return the new exit flag `-8`, meaning the solver cannot continue because it encounters a singular system. In this case, switching to full data instead of sparse can help the solver to complete successfully.

You can now update options in generated code using `optimoptions`. Still, the recommended way to update an option is to use dot notation, not `optimoptions`.

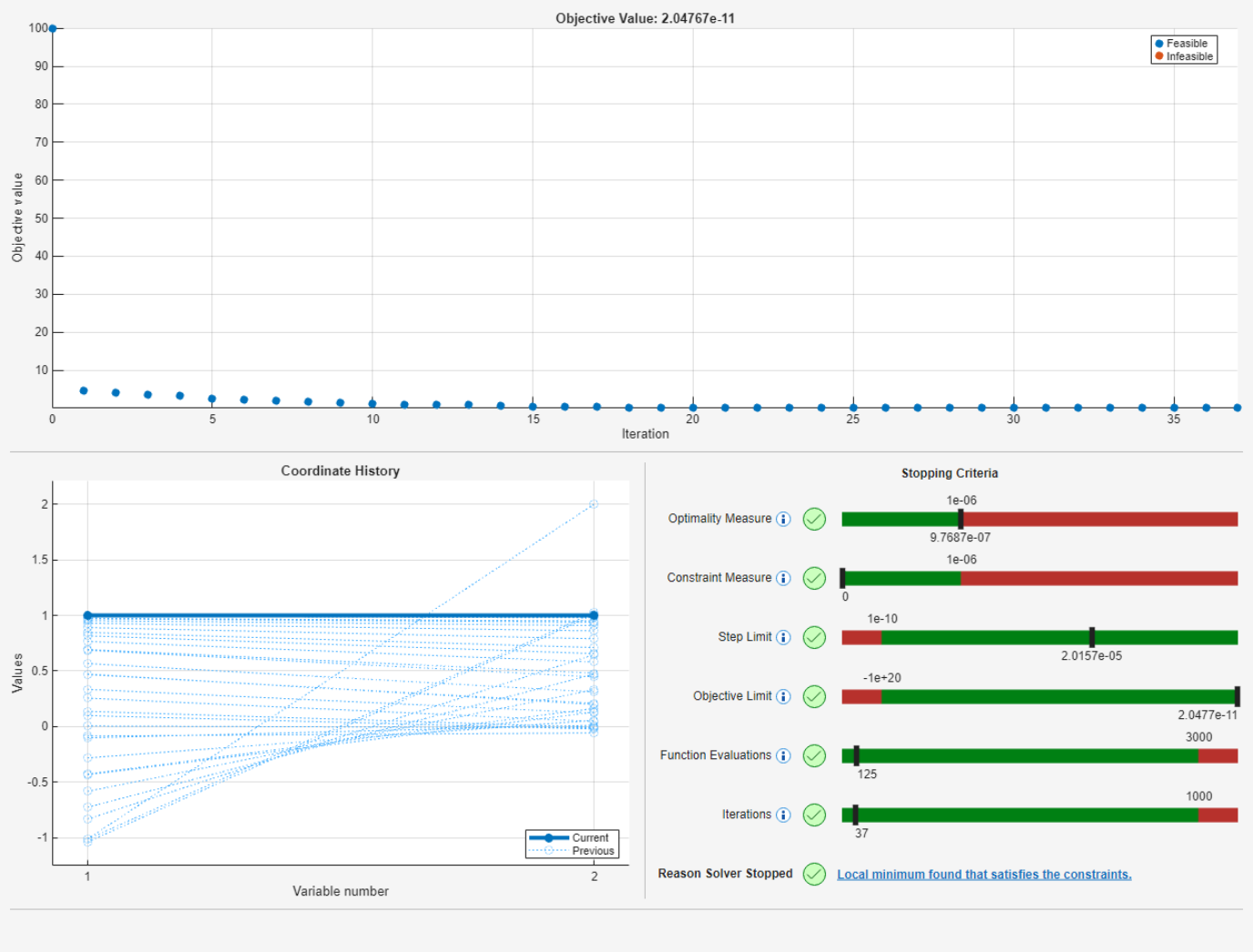
```
opts = optimoptions("coneprog",ConstraintTolerance=1e-4);
opts = optimoptions(opts,MaxIterations=1e4); % Not preferred
opts.MaxIterations = 1e4; % Recommended
```

R2024b

★Detailed plot function for fmincon

The `fmincon` `optimplot` plot function displays detailed information during an optimization. For example,

```
x0 = [-2;2];
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
lb = [-2;-2];
ub = [3;3];
options = optimoptions("fmincon",PlotFcn="optimplot");
[x,fval] = fmincon(fun,x0,[],[],[],[],lb,ub,[],options)
```



The `optimplot` plot function opens in a separate window from any other plot functions. For an example, see Monitor Solution Process with `optimplot`.

⚠★Plot Functions and Output Functions for intlinprog "highs" Algorithm: Obtain iterative information as the solver progresses

The `intlinprog` `"highs"` algorithm now supports output functions and plot functions. For example, to have `intlinprog` produce a plot while solving, specify the built-in `"optimplotmilp"` plot function.

```
options = optimoptions("intlinprog",PlotFcn="optimplotmilp");
[x,fval] = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,options)
```

For details, see `intlinprog` Output Function and Plot Function Syntax.

⚠★Compatibility Considerations

The `intlinprog` `"legacy"` algorithm will be removed in a future release.

The `optimValues` output function and plot function structure field named `lowerbound` is now named `dualbound`.

The `"highs"` algorithm now supports the `MaxFeasiblePoints` option. The `output` structure for the `"highs"` algorithm now supports the `numfeaspoints` field.

★Code Generation for coneprog: Generate C or C++ code for problems with linear objective function and cone constraints

The `coneprog` function now supports code generation. For details, see Code Generation for `coneprog` Background and Generate Code for `coneprog`.

Single-precision code generation for quadprog, lsqlin, and optimwarmstart

The `quadprog` and `lsqlin` solvers and the `optimwarmstart` function can generate code for single-precision floating point hardware. If you want to generate single-precision code, all relevant data, such as constraint matrices and bounds, must have the type `"single"`. For details, see Single-Precision Code Generation.

Note This single-precision capability is available for code generation only; it is not available for general MATLAB computations.

EquationProblem and OptimizationConstraint Evaluation: Compute equation and constraint satisfaction using issatisfied and evaluate

Compute the degree of satisfaction of nonlinear equation systems and of optimization constraints using the `evaluate` and `issatisfied` functions. For an `EquationProblem` object, `issatisfied` returns a boolean value indicating whether all equations are satisfied at a point, and returns a boolean satisfaction value for each equation in a system in the second output. `evaluate` returns the equation value at a point. For an `OptimizationConstraint` object, `evaluate` returns the values of the constraints, and `issatisfied` returns boolean values for the constraints.

The value of a constraint depends on the constraint type. An equation is equivalent to an == constraint. For expressions L and R:

Constraint Type	Value
L <= R	L – R
L >= R	R – L
L == R	abs(L – R)

For details, see the `evaluate` and `issatisfied` reference pages.

mpsread uses HiGHS software

`mpsread` now uses HiGHS open-source software to read MPS files into a form that `intlinprog` and `linprog` accept.

Compatibility Considerations

The `mpsread` display now includes a reference to HiGHS.

Green hydrogen production example

The example Optimize Green Hydrogen Production System shows how to use forecast cost and demand data to plan a hydrogen production system optimally. The example uses the problem-based optimization approach to model and solve a large linear programming problem.

Functionality being removed or changed

intlinprog "legacy" algorithm and Algorithm option will be removed in a future release
Still runs

The `intlinprog` `"legacy"` algorithm will be removed in a future release. Usually, the `"highs"` algorithm (default) performs better than the `"legacy"` algorithm. Now that `"highs"` supports output functions and plot functions, there is no loss of functionality when switching to `"highs"`.

Because `intlinprog` will have just one algorithm when the `"legacy"` algorithm is removed, at that time:

- The `Algorithm` option will be removed.
- All options that are valid only for the `"legacy"` algorithm will be removed.
- `intlinprog` will use the `"highs"` algorithm, the current default.
- The `algorithm` field will be removed from the `output` structure.

intlinprog plot function and output function optimValues field lowerbound renamed dualbound
Behavior change

The `optimValues` output function and plot function structure field named `lowerbound` is now named `dualbound`. The new name is clearer for maximization problems, where the `dualbound` is an upper bound on the objective function value. The `optimplotmilp` plot function now uses the labels `Primal Bound` and `Dual Bound` instead of `Branch/bound UB` and `Branch/bound LB`.

To avoid errors, change any custom output functions and plot functions to use the new field name.

linprog "dual-simplex-legacy" algorithm will be removed in a future release
Still runs

The `linprog` `"dual-simplex-legacy"` algorithm will be removed in a future release. Usually, the `"dual-simplex-highs"` algorithm (default) performs better than the `"dual-simplex-legacy"` algorithm.

More objects support evaluate and issatisfied

The following objects now support the `evaluate` and `issatisfied` functions:

- `EquationProblem`
- `OptimizationConstraint`
- `OptimizationEquality`
- `OptimizationInequality`

You can evaluate the values of these objects (optimization expressions or constraints or equations) on a set of points that are expressed as an `OptimizationValues` object or on a single point expressed as a structure. For an example, see Evaluate Expressions in Equation.

R2024a

HiGHS Algorithm for intlinprog: Solve mixed-integer linear programming problems faster and more reliably

The `intlinprog` solver has a new algorithm based on the open-source HiGHS code. The new algorithm is the default for `intlinprog`. Testing shows that the new algorithm often solves MILP problems faster and more reliably than before. To choose this algorithm using `optimoptions`, set the `Algorithm` option to `"highs"`.

Compatibility Considerations

To use the previous algorithm, set the `Algorithm` option to `"legacy"` using `optimoptions`. Iterative display is different than before. For details, see “intlinprog algorithm” on page 5-3.

HiGHS Algorithm for linprog: Solve linear programming problems faster and more reliably

The `linprog` solver has a new algorithm based on the open-source HiGHS code. The new algorithm is the default for `linprog`. Testing shows that the new algorithm often solves linear programming problems faster and more reliably than before. To choose this algorithm using `optimoptions`, set the `Algorithm` option to `"dual-simplex-highs"`.

Compatibility Considerations

To use the previous default algorithm, set the `Algorithm` option to `"dual-simplex-legacy"` using `optimoptions`. Iterative display is different than before. For algorithmic details, see Dual-Simplex-Highs Algorithm.

Single-precision code generation for fmincon

The `fmincon` solver can generate code for single-precision floating point hardware. To generate single-precision code, all relevant data such as constraint matrices and nonlinear function outputs must be of type `'single'`. For details, see Single-Precision Code Generation.

Note This single-precision capability is available for code generation only; it is not available for general MATLAB computations.

coneprog LinearSolver option: Accelerate solutions of problems with dense constraint matrices

The `coneprog` solver has a new `LinearSolver` algorithm, `"normal-dense"`, that has good performance on dense cone programming problems. Try this algorithm when your problem has a constraint matrix (cone constraint or linear constraint) with a large fraction of nonzero elements. Set `LinearSolver="normal-dense"` using `optimoptions`:

```
options = optimoptions("coneprog",LinearSolver="normal-dense");
```

For an example, see Compare Speeds of `coneprog` Algorithms.

Optimization expressions support 'like' syntax

You can create optimization expressions using the `'like'` syntax, which can simplify initialization of expressions in a loop. See Initialize Optimization Expressions.

Evaluate objectives and constraints in an optimization problem at a set of points

The `evaluate` function can now evaluate all objective and constraint values for an optimization problem at a set of points.

Similarly, the new `issatisfied` function can evaluate all constraints in an optimization problem at a set of points, determining feasibility to within a settable tolerance. For details and examples, see the function reference pages.

⚠More robust handling of unconstrained problems in lsqlin

The `lsqlin` solver now takes extra steps when solving an ill-conditioned unconstrained problem with a square input matrix `C`. The results can have improved accuracy and the likelihood of obtaining an `Inf` or `NaN` result is decreased.

⚠Compatibility Considerations

For unconstrained problems, the `output.algorithm` field is now `'direct'` instead of the previous `'mldivide'`.

⚠Functionality being removed or changed

intlinprog algorithm

The HiGHS algorithm (`Algorithm="highs"`) usually solves MILP problems faster and more reliably than the previous algorithm (`Algorithm="legacy"`). As a result of this change, `intlinprog` options and default display have changed.

- The following options are not available with the HiGHS algorithm.
  - `BranchRule`
  - `CutGeneration`
  - `CutMaxIterations`
  - `Heuristics`
  - `HeuristicsMaxNodes`
  - `IntegerPreprocess`
  - `IntegerTolerance`
  - `LPMaxIterations`
  - `LPOptimalityTolerance`
  - `MaxFeasiblePoints`

- `NodeSelection`
- `ObjectiveImprovementThreshold`
- `OutputFcn`
- `PlotFcn`
- `RootLPAlgorithm`
- `RootLPMaxIterations`

  Testing has shown that the HiGHS algorithm generally performs well without the removed options.
- In particular, when using the HiGHS algorithm, `intlinprog` does not use output functions or plot functions.
- Iterative display is different than before.

For algorithmic details, see HiGHS MILP Algorithm.

R2023b

Check derivatives using checkGradients

To check whether a finite-difference approximation of a first derivative matches a gradient function, use the `checkGradients` function. The `checkGradients` function applies to both objective functions and nonlinear constraint functions. `checkGradients` allows you to choose a tolerance for the accuracy of the comparison. The function optionally returns a structure showing the differences between the gradient function and finite-difference approximations at the comparison point.

Compatibility Considerations

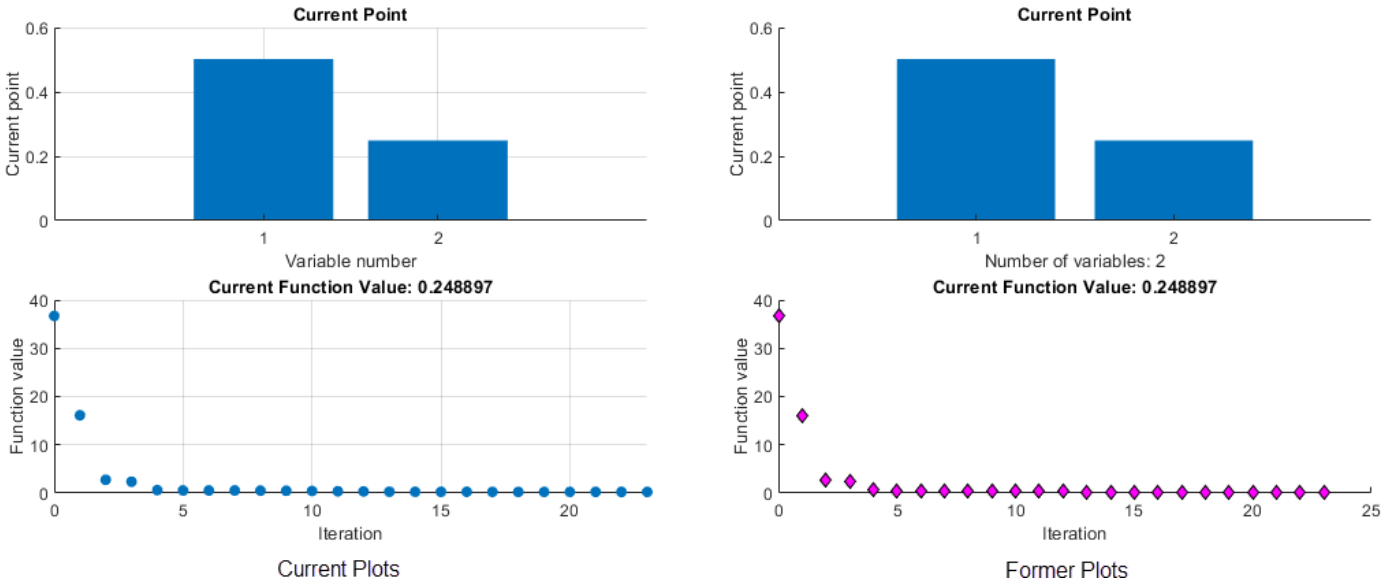
The `CheckGradients` option of the following solvers will be removed in a future release:

- `fmincon`
- `fminunc`
- `fseminf`
- `fsolve`
- `lsqcurvefit`
- `lsqnonlin`

The `CheckGradients` option will be removed because the `checkGradients` function is more flexible and provides more information.

Plot Functions Update

Plot functions, available for many solvers by using the `"PlotFcn"` option, have more consistent appearances for both Optimization Toolbox solvers and Global Optimization Toolbox solvers. The information shown is the same as before, but the markers are now uniform in appearance and use standard colors.




```
% Code to create the plots
fun = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
x0 = [-1.25,1];
```

```
A = [1,2];
b = 1;
opts = optimoptions("fmincon","PlotFcn",["optimplotx","optimplotfval"]);
x = fmincon(fun,x0,A,b,[],[],[],[],[],opts);
```

Sum optimization expressions over vector dimensions

Problem-based optimization expressions now allow the `dim` argument to be a vector of positive integers in the expression `sum(x,dim)`. For example, the following code runs as shown:

```
x = optimvar("x",3,5,6,7);
expr = sum(x,[2 4])

expr =

  3×1×6 Linear OptimizationExpression array with properties:

    IndexNames: {{}  {}  {}}
     Variables: [1×1 struct] containing 1 OptimizationVariable

  See expression formulation with show.
```

Jacobian multiply function accepts any data type

The `lsqcurvefit`, `lsqnonlin`, and `fsolve` functions can use the `JacobianMultiplyFcn` option to save memory in large, structured problems. The data named `Jinfo` in the Jacobian multiply function can now be of any type; previously, `Jinfo` was restricted to be a standard double array. This new flexibility can ease the process of programming a Jacobian multiply function.

R2023a

Linear and Nonlinear Constraints in Nonlinear Least Squares: Solve Least Squares Problems with Constraints

The `lsqnonlin` and `lsqcurvefit` solvers now accept linear and nonlinear constraints. You can use these constraints by calling the solvers directly, by solving a least-squares problem using the problem-based approach, or by using the Optimize Live Editor task. For examples, see the function reference pages.

The solvers can solve constrained problems using the new `'interior-point'` algorithm. Internally, this algorithm reformulates a least-squares problem for the `fmincon` interior-point solver. Usually, this reformulation leads to a solution more efficiently than specifying a problem for `fmincon` directly. See Compare `lsqnonlin` and `fmincon` for Constrained Nonlinear Least Squares.

If you specify linear or nonlinear constraints and do not specify an algorithm, the solvers switch to the `'interior-point'` algorithm. If you specify these constraints and specify another algorithm, the solvers throw an error.

Compatibility Considerations

- For solver-based optimization, the `'interior-point'` algorithm does not accept functions as names, but instead requires that the objective and nonlinear constraint functions are function handles. To update existing code to use the `'interior-point'` algorithm, specify the objective function using a function handle such as `@objfun` instead of a function name such as `'objfun'`.
- The default solver for problem-based least-squares with linear or nonlinear constraints has changed from `fmincon` to `lsqnonlin`. To have `solve` or `prob2struct` use `fmincon` as the solver, specify the `Solver` name-value argument, such as

  ```
  [sol,fval] = solve(prob,x0,Solver="fmincon")
  ```

Clarified plots and iterative display in problem-based maximization

The `solve` function returns more informative plots and iterative display for maximization problems. Previously, many plots and iterative display showed the negative of the objective function, because, internally, solvers maximize by minimizing the negative of the objective function. Also, linear and quadratic programming problems did not include any additive constants in the iterative display or plots. Now the plots and iterative display return the expected values for both maximization and minimization problems.

Compatibility Considerations

For problem-based maximization:

- Iterative display shows the true (not negated) objective function value.
- Custom plot functions and output functions now have the true (not negated) objective function value in the `optimValues.fval` field.
- For solvers that use `optimValues.gradient`, the gradient has its sign flipped from the previous value.
- `optimValues.searchdirection` has a flipped sign from the previous value.

Some functions have minor updates in their iterative display.

- The `fsolve` function has the header `||f(x)||^2` instead of `f(x)`.
- The `fminsearch` function has the header `f(x)` instead of `min f(x)`.
- `lsqlin` has the same headers as `quadprog` except that it uses `Resnorm` instead of `f(x)`.

fminbnd and fminsearch solvers supported in problem-based optimization

The `solve` and `prob2struct` functions accept `"fminbnd"` and `"fminsearch"` as values in the `Solver` name-value argument. The `fminbnd` solver can solve nonlinear optimization problems that have one bounded scalar variable. The `fminsearch` solver can solve unconstrained multidimensional nonlinear optimization problems. The `solvers` function, which lists the available solvers for a problem, includes these solvers for appropriate problems.

⚠Improvements in intlinprog and linprog

Internal scaling factors in `intlinprog` and in the `linprog 'dual-simplex'` and `'interior-point'` algorithms are now restricted to be powers of 2. This change avoids floating-point rounding errors. Along with other code changes, the solvers can exhibit improved reliability and performance.

⚠Compatibility Considerations

These changes can cause the solution path taken by these algorithms to differ from that of previous versions. The changes do not always lead to better performance. There is no provision for obtaining the previous behavior.

Optimal Trajectory Example: More Efficient Problem Formulation, Extension to Variable Mass

The example Discretized Optimal Trajectory, Problem-Based has the following improvements:

- Fewer problem variables. The variables representing position and velocity at each time are now inferred by integrating the equations of motion, rather than being problem variables.
- Effect of expelled mass. The object's mass decreases as it expels fuel. The example shows how to account for this effect in the trajectory, and how to choose a good starting point for the solver.

R2022b

Version: 9.4

New Features

Bug Fixes

Enhanced Analysis in fcn2optimexpr: Speed problem creation and solution

The `fcn2optimexpr` function has enhanced analysis of problems. In particular, `for` loops are faster than before, and nested `for` loops are much faster. In order to realize this speed gain, you must convert your `for` loops to functions, as described in Static Analysis of Optimization Expressions.

For more information about enhanced analysis, which is named static analysis, see Static Analysis of Optimization Expressions. For examples of converting `for` loops to functions, see Create `for` Loop for Static Analysis and Convert Constraints in `for` Loops for Static Analysis.

solvers Function: Find the default and available solvers for a problem

For a problem expressed as an `OptimizationProblem` or `EquationProblem`, the `solvers` function lists the default solver and all valid solvers for the problem. For example,

```
[defaultsolver,validsolvers] = solvers(prob)
```

You can call any valid solver by using the `Solver` name-value argument in `solve` or `prob2struct`. For example, if `fmincon` is a valid solver for `prob`:

```
[sol,fval] = solve(prob,x0,Solver="fmincon")
% or
problem = prob2struct(prob,x0,Solver="fmincon")
```

If you have Global Optimization Toolbox, the lists of default and valid solvers include Global Optimization Toolbox solvers.

optim.coder.infbound: Express infinite bounds for code generation

For code generation where the target does not accept `Inf` as a value of a bound, use `optim.coder.infbound` instead. For example,

```
lb = [-optim.coder.infbound,0]; % lb(1) = -Inf, lb(2) = 0
ub = optim.coder.infbound(1,2); % ub has two elements, all Inf
```

For details, see the function reference page.

R2022a

Problem-Based Optimize Live Editor Task: Solve optimization problems or systems of equations using a visual interface

The Optimize Live Editor task now supports the Problem-Based Optimization Workflow. Optimize also supports the problem-based approach for Global Optimization Toolbox single-objective and multiobjective solvers. Using problem-based Optimize you can:

- Create optimization variables, expressions, and problems.
- Create linear or nonlinear objectives, constraints, or equations using built-in function templates.
- Optionally, specify the solver and options.
- Run the solver directly from the task.
- Export the resulting MATLAB code.

To launch the task, create a Live Editor window, then select **Task > Optimize** or **Insert > Task > Optimize**. Then choose the problem-based approach.

For an example, see Get Started with Problem-Based Optimize Live Editor Task. For tips, see Use Problem-Based Optimize Live Editor Task Effectively.

⚠LBFGS Algorithm in fminunc: Optimize large problems

To save memory and gain speed in large `fminunc` problems, use the `"quasi-newton"` algorithm and set the `HessianApproximation` option to `"lbfgs"`. For an example of the speed gains, see Solve Nonlinear Problem with Many Variables. For algorithm details, see `fminunc quasi-newton` Algorithm.

⚠Compatibility Considerations

The option name has been renamed from *HessUpdate* to `HessianApproximation`. For details, see `fminunc options`.

⚠fcn2optimexpr Arguments: Affect conversion and obtain details of conversion

The `fcn2optimexpr` function gains two name-value arguments that can affect the conversion process and give details of the conversion:

- `Analysis` — When `"on"` (default), `fcn2optimexpr` analyzes the function being converted to see whether it is composed entirely of supported operations (see Supported Operations for Optimization Variables and Expressions). If so, `fcn2optimexpr` uses the operations to create the optimization expression, and the resulting expression supports automatic differentiation (see Automatic Differentiation Background). Furthermore, `solve` uses the most appropriate solver by default, as described in `Solver`. If the function is not composed entirely of supported operators, the resulting expression is a black box. In this case, the expression does not support automatic differentiation, and `solve` has restricted solver choice, so can fail to use the most efficient solver.
- `Display` — When `"on"`, reports the results of `Analysis`.

⚠Compatibility Considerations

When a function is composed entirely of supported operations, by default `fcn2optimexpr` returns a different expression than previously. The returned expression supports automatic differentiation and all applicable solvers.

The function analysis can take some time. This time can be noticeable for complicated functions with nested loops.

To obtain the previous behavior, set `Analysis="off"`.

⚠Functionality being removed or changed

Relative tolerance change for intlinprog
Behavior change

`intlinprog` changes the stopping condition associated with the `RelativeGapTolerance` option. Now `intlinprog` stops if the relative difference between the internally calculated upper (`U`) and lower (`L`) bounds on the objective function is less than or equal to `RelativeGapTolerance`:

`(U – L)/(|U| + 1) <= RelativeGapTolerance`.

Previously, when `L` had a large magnitude, `intlinprog` automatically modified the tolerance:

tolerance = `min(1/(1 + |L|), RelativeGapTolerance)`.

This change makes the stopping condition easier to understand and apply.

To obtain approximately the previous behavior when the objective function value has large magnitude, set the option to the value `min(1/(1 + |L|), RelativeGapTolerance)`, where `|L|` is the magnitude of the objective function for the same problem without integer constraints (`intcon = []`).

R2021b

fmincon 'interior-point' Algorithm: Obtain feasible solutions using a new feasibility mode

The `fmincon` `'interior-point'` algorithm has an updated feasibility routine. For problems where `fmincon` has difficulty reaching a feasible solution, this routine sometimes enables `fmincon` to succeed. The routine usually works better when you also set the `SubproblemAlgorithm` option to `'cg'`. To use the routine, set the new `EnableFeasibilityMode` option to `true` using `optimoptions`.

```
options = optimoptions('fmincon',...
    'Algorithm','interior-point',...
    'EnableFeasibilityMode',true,...
    'SubproblemAlgorithm','cg');
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

For details, see Feasibility Mode. For an example, see Obtain Solution Using Feasibility Mode.

⚠Functionality being removed or changed

Problem-based behavior change for norm(sum of squares)^2

Behavior change

When an optimization expression has the form `norm(expression)^2`, internal simplification routines now convert the expression to a sum of squares. Therefore, `solve` and `prob2struct` can now use the `lsqlin` and `lsqnonlin` solvers for this type of objective function when the problem constraints are compatible (linear constraints for `lsqlin` or bound constraints for `lsqnonlin`). See Write Objective Function for Problem-Based Least Squares.

Problem-based behavior change for prob2struct with integer constraints

Behavior change

When a problem has integer constraints and a nonlinear objective function, the default behavior of the `prob2struct` function now depends on whether you have Global Optimization Toolbox installed.

- If Global Optimization Toolbox is installed, `prob2struct` returns a `ga` problem.
- If Global Optimization Toolbox is not installed, `prob2struct` throws an error.

Previously, when a problem included integer constraints and a nonlinear objective function, the result was a `coneprog`, `fmincon`, `fminunc`, `lsqlin`, `lsqnonlin`, or `quadprog` problem structure. However, these solvers cannot solve problems with integer constraints.

To obtain the previous behavior, set the `Solver` name-value argument to the appropriate solver name. When you do, `prob2struct` warns that the resulting problem cannot be solved with integer constraints. For example, for a quadratic programming problem, enter

```
problem = prob2struct(prob,"Solver","quadprog");
```

For details, see Integer Constraints in Nonlinear Problem-Based Optimization.

R2021a

Automatic Differentiation: Solve nonlinear least-squares and equation problems using automatically computed derivatives in the problem-based approach

Automatic Differentiation (AD) now applies to problem-based nonlinear least-squares and systems of nonlinear equations, as well as to general nonlinear optimization problems. For added speed with these problem types, solvers can now use forward AD as well as reverse AD. New option values in `solve` and `prob2struct` allow you to choose the AD type. For the default AD type for various problems and solvers, see the `'ObjectiveDerivative'` argument.

Problem-based expressions can use more trigonometric and hyperbolic functions, including `sec`, `asec`, `sech`, and `asech`. For details, see Automatic Differentiation Background and Supported Operations on Optimization Variables and Expressions.

Warm Start quadprog and lsqlin: Solve problems faster by reusing data structures

The `quadprog` `'active-set'` algorithm and `lsqlin` `'active-set'` algorithm can update and use warm start data structures. These structures speed solving problems that are similar to previously solved problems. Warm start is available for `quadprog` and `lsqlin` code generation as well. Create a warm start object using the `optimwarmstart` function. For details, see Warm Start Best Practices, Warm Start `quadprog`, and the function reference pages.

Code Generation for Linear Least Squares: Generate C code for linear least squares problems

The `lsqlin` `'active-set'` algorithm now supports code generation. For examples and details, see Code Generation in Linear Least Squares: Background and Generate Code for `lsqlin`.

Interior-Point fmincon Algorithm: Solve constrained nonlinear problems faster

The `fmincon` `'interior-point'` algorithm has an added barrier parameter update algorithm. Set the barrier parameter algorithm using the new `BarrierParamUpdate` option. In tests, the solver is often faster for problems with inequality constraints (including bound constraints) when this option is set to `'predictor-corrector'`. To obtain the previous `fmincon` behavior, set the `BarrierParamUpdate` option to its default value, `'monotone'`.

For details, see fmincon Interior Point Algorithm.

Second-Order Cone Programming: Create and solve problems with second-order cone constraints in the problem-based approach

Problem-based optimization now uses `coneprog` internally to solve second-order cone programming problems when you call `solve` or `prob2struct`. Create a problem with a linear objective function and optionally with bounds or linear constraints. Specify a second-order cone constraint using the newly-supported `norm` function using the syntax

```
norm(linear_expression1) + constant <= linear_expression2;
```

Here, `linear_expression` represents a linear expression in optimization variables. You can also specify a cone constraint as the square root of a sum of squares of optimization variables. See Write Constraints for Problem-Based Cone Programming. For an example, see Minimize Energy of Piecewise Linear Mass-Spring System Using Cone Programming, Problem-Based.

New Algorithms in Second-Order Cone Programming: Solve problems more quickly and with greater numerical stability

The `coneprog` function has a new option named `LinearSolver` that can help you to achieve greater speed and stability in problems with large cones or large, sparse constraint matrices with some dense columns. By default, `coneprog` chooses an internal solution algorithm based on your problem. For more information, see Compare Speeds of `coneprog` Algorithms, the `coneprog` options section, and Second-Order Cone Programming Algorithm.

Compatibility Considerations

To obtain the same behavior as the previous release, set the `LinearSolver` option to `'augmented'`.

Functionality Being Removed or Changed

Optimization App Removed

Errors

The Optimization app (`optimtool`) has been removed. For a visual interface to solvers, use the Optimize Live Editor task.

prob2struct Options Name-Value Pair Removed

Errors

The `Options` name-value pair has been removed from `prob2struct`. To modify options, edit the resulting `problem` structure. For example,

```
problem.options = optimoptions('fmincon',...
    'Display','iter','MaxFunctionEvaluations',5e4);
% Or, to set just one option:
problem.options.MaxFunctionEvaluations = 5e4;
```

The `Options` name-value pair was removed because it can cause ambiguity in the presence of automatic differentiation.

mapSolution Removed

Errors

The `mapSolution` function has been removed. To obtain the indices of optimization variables when you convert a problem using `prob2struct`, use `varindex`. For an example showing how to create a solution structure from a converted problem, see Solve Problem Using Both Approaches.

Two coneprog lambda Structures Renamed

Behavior change

The `coneprog` `lambda` output argument fields `lambda.eq` and `lambda.ineq` have been renamed to `lambda.eqlin` and `lambda.ineqlin`, respectively. This change causes the `coneprog` `lambda` structure fields to have the same names as the corresponding fields in other solvers.

Also, when `coneprog` returns exit flag –2, –3, or –10, the `lambda` output argument is now empty. Previously in these cases, the `lambda` output argument was a structure with empty fields.

Optimize Bound Behavior Change
Behavior change

The `Optimize` Live Editor task no longer performs scalar expansion on a bound specified as `From workspace` and as a scalar. Previously, if your workspace variable was a scalar and the problem had multiple variables, Optimize applied the scalar bound to all problem variables. Now, Optimize treats a scalar value `From workspace` as applying to the first variable alone, with no scalar expansion.

You can have a single scalar value apply to all variables by choosing `All bounds the same` instead of `From workspace`.