Wolfram *Mathematica*® Tutorial Collection

CONSTRAINED OPTIMIZATION

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Brett Champion, Adam Strzebonski

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Introduction to Constrained Optimization in *Mathematica*

Optimization Problems

Constrained optimization problems are problems for which a function $f(x)$ is to be minimized or maximized subject to constraints $\Phi(x)$. Here $f:\mathbb{R}^n \to \mathbb{R}$ is called the objective function and $\Phi(x)$ is a Boolean-valued formula. In *Mathematica* the constraints $\Phi(x)$ can be an arbitrary Boolean combination of equations $g(x)=0$, weak inequalities $g(x)\geq 0$, strict inequalities $g(x)>0$, and $x\in\mathbb{Z}$ statements. The following notation will be used.

$$\begin{array}{l}\text{Min } f(x)\\ s.t.\ \ \Phi(x)\end{array} \tag{1}$$

stands for "minimize $f(x)$ subject to constraints $\Phi(x)$", and

$$\begin{array}{l}\text{Max } f(x)\\ s.t.\ \ \Phi(x)\end{array} \tag{2}$$

stands for "maximize $f(x)$ subject to constraints $\Phi(x)$".

You say a point $u\in\mathbb{R}^n$ satisfies the constraints Φ if $\Phi(u)$ is true.

The following describes constrained optimization problems more precisely, restricting the discussion to minimization problems for brevity.

Global Optimization

A point $u\in\mathbb{R}^n$ is said to be a global minimum of f subject to constraints Φ if u satisfies the constraints and for any point v that satisfies the constraints, $f(u)\leq f(v)$.

A value $a\in\mathbb{R}\bigcup\{-\infty,\infty\}$ is said to be the global minimum value of f subject to constraints Φ if for any point v that satisfies the constraints, $a\leq f(v)$.

The global minimum value a exists for any f and Φ. The global minimum value a is attained if there exists a point u such that $\Phi(u)$ is true and $f(u)==a$. Such a point u is necessarily a global minimum.

If f is a continuous function and the set of points satisfying the constraints Φ is compact (closed and bounded) and nonempty, then a global minimum exists. Otherwise a global minimum may or may not exist.

Here the minimum value is not attained. The set of points satisfying the constraints is not closed.

```
In[1]:= Minimize[{x, x^2 + y^2 < 1}, {x, y}]
```

```
Minimize::wksol :
  Warning: There is no minimum in the region described by the constraints; returning a result
      on the boundary. >>
```

```
Out[1]= {-1, {x → -1, y → 0}}
```

Here the set of points satisfying the constraints is closed but unbounded. Again, the minimum value is not attained.

```
In[3]:= Minimize[{x^2, x y == 1}, {x, y}]
```

```
Minimize::natt : The minimum is not attained at any point satisfying the given constraints. >>
```

```
Out[3]= {0, {x → Indeterminate, y → Indeterminate}}
```

The minimum value may be attained even if the set of points satisfying the constraints is neither closed nor bounded.

```
In[4]:= Minimize[{x^2 + (y - 1)^2, y > x^2}, {x, y}]
Out[4]= {0, {x → 0, y → 1}}
```

Local Optimization

A point $u \in \mathbb{R}^n$ is said to be a local minimum of f subject to constraints Φ if u satisfies the constraints and, for some $r > 0$, if v satisfies $|v - u| < r \bigwedge \Phi(v)$, then $f(u) \le f(v)$.

A local minimum may not be a global minimum. A global minimum is always a local minimum.

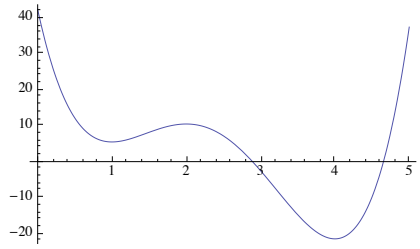
Here `FindMinimum` finds a local minimum that is not a global minimum.

```
In[18]:= FindMinimum[3 x^4 - 28 x^3 + 84 x^2 - 96 x + 42, {x, 1.1}]
Out[18]= {5., {x → 1.}}
```

```
In[19]:= Minimize[3 x^4 - 28 x^3 + 84 x^2 - 96 x + 42, {x}]
Out[19]= {-22, {x → 4}}
```

```
In[20]:= Plot[3 x^4 - 28 x^3 + 84 x^2 - 96 x + 42, {x, 0, 5}]
```

Out[20]=



Solving Optimization Problems

The methods used to solve local and global optimization problems depend on specific problem types. Optimization problems can be categorized according to several criteria. Depending on the type of functions involved there are linear and nonlinear (polynomial, algebraic, transcendental, ...) optimization problems. If the constraints involve $x \in \mathbb{Z}$, you have integer and mixed integer-real optimization problems. Additionally, optimization algorithms can be divided into numeric and symbolic (exact) algorithms.

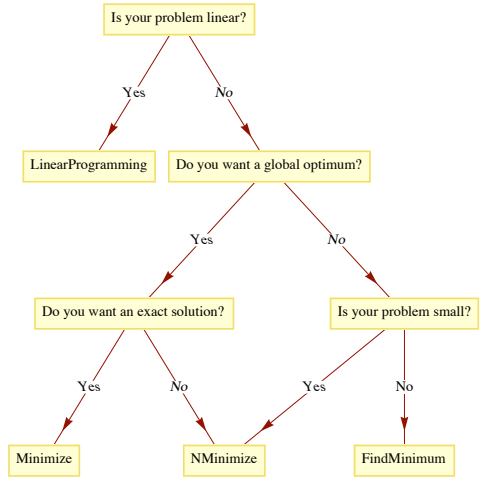
Mathematica functions for constrained optimization include `Minimize`, `Maximize`, `NMinimize` and `NMaximize` for global constrained optimization, `FindMinimum` for local constrained optimization, and `LinearProgramming` for efficient and direct access to linear programming methods. The following table briefly summarizes each of the functions.

Function	*Solves*	*Algorithms*
`FindMinimum , FindMaximum`	numeric local optimization	linear programming methods, nonlinear interior point algorithms, utilize second derivatives
`NMinimize , NMaximize`	numeric global optimization	linear programming methods, Nelder-Mead, differential evolution, simulated annealing, random search
`Minimize , Maximize`	exact global optimization	linear programming methods, cylindrical algebraic decomposition, Lagrange multipliers and other analytic methods, integer linear programming
`LinearProgramming`	linear optimization	linear programming methods (simplex, revised simplex, interior point)

Summary of constrained optimization functions.

Here is a decision tree to help in deciding which optimization function to use.

Out[2]=



Linear Programming

Introduction

Linear programming problems are optimization problems where the objective function and constraints are all linear.

Mathematica has a collection of algorithms for solving linear optimization problems with real variables, accessed via `LinearProgramming`, `FindMinimum`, `FindMaximum`, `NMinimize`, `NMaximize`, `Minimize`, and `Maximize`. `LinearProgramming` gives direct access to linear programming algorithms, provides the most flexibility for specifying the methods used, and is the most efficient for large-scale problems. `FindMinimum`, `FindMaximum`, `NMinimize`, `NMaximize`, `Minimize`, and `Maximize` are convenient for solving linear programming problems in equation and inequality form.

This solves a linear programming problem

$$\begin{array}{ll} \text{Min} & x + 2y \\ s.t. & -5x + y = 7 \\ & x + y \geq 26 \\ & x \geq 3,\ y \geq 4 \end{array}$$

using `Minimize`.

```
In[1]:= Minimize[{x + 2 y, -5 x + y == 7 && x + y ≥ 26 && x ≥ 3 && y ≥ 4}, {x, y}]
```

$$\text{Out[1]}= \left\{\frac{293}{6}, \left\{x \to \frac{19}{6}, y \to \frac{137}{6}\right\}\right\}$$

This solves the same problem using `NMinimize`. `NMinimize` returns a machine-number solution.

```
In[2]:= NMinimize[{x + 2 y, -5 x + y == 7 && x + y ≥ 26 && x ≥ 3 && y ≥ 4}, {x, y}]
Out[2]= {48.8333, {x → 3.16667, y → 22.8333}}
```

This solves the same problem using `LinearProgramming`.

```
In[3]:= LinearProgramming[{1, 2}, {{-5, 1}, {1, 1}},
          {{7, 0}, {26, 1}}, {{3, Infinity}, {4, Infinity}}]
```

$$\text{Out[3]}= \left\{\frac{19}{6}, \frac{137}{6}\right\}$$

The LinearProgramming Function

`LinearProgramming` is the main function for linear programming with the most flexibility for specifying the methods used, and is the most efficient for large-scale problems.

The following options are accepted.

option name	*default value*	
`Method`	`Automatic`	method used to solve the linear optimization problem
`Tolerance`	`Automatic`	convergence tolerance

Options for `LinearProgramming`.

The `Method` option specifies the algorithm used to solve the linear programming problem. Possible values are `Automatic`, `"Simplex"`, `"RevisedSimplex"`, and `"InteriorPoint"`. The default is `Automatic`, which automatically chooses from the other methods based on the problem size and precision.

The `Tolerance` option specifies the convergence tolerance.

Examples

Difference between Interior Point and Simplex and/or Revised Simplex

The simplex and revised simplex algorithms solve a linear programming problem by moving along the edges of the polytope defined by the constraints, from vertices to vertices with successively smaller values of the objective function, until the minimum is reached. *Mathematica*'s implementation of these algorithm uses dense linear algebra. A unique feature of the implementation is that it is possible to solve exact/extended precision problems. Therefore these methods are suitable for small-sized problems for which non-machine-number results are needed, or a solution on the vertex is desirable.

Interior point algorithms for linear programming, loosely speaking, iterate from the interior of the polytope defined by the constraints. They get closer to the solution very quickly, but unlike the simplex/revised simplex algorithms, do not find the solution exactly. *Mathematica*'s implementation of an interior point algorithm uses machine precision sparse linear algebra. Therefore for large-scale machine-precision linear programming problems, the interior point method is more efficient and should be used.

This solves a linear programming problem that has multiple solutions (any point that lies on the line segment between `{1, 0}` and `{1, 0}` is a solution); the interior point algorithm gives a solution that lies in the middle of the solution set.

```
In[6]:= LinearProgramming[{-1., -1}, {{1., 1.}}, {{1., -1}}, Method → "InteriorPoint"]
Out[6]= {0.5, 0.5}
```

Using `Simplex` or `RevisedSimplex`, a solution at the boundary of the solution set is given.

```
In[7]:= LinearProgramming[{-1., -1}, {{1., 1.}}, {{1., -1}}, Method → "RevisedSimplex"]
Out[7]= {1., 0.}
```

This shows that interior point method is much faster for the following random sparse linear programming problem of 200 variables and gives similar optimal value.

```
In[43]:= m = SparseArray[RandomChoice[{0.1, 0.9} → {1., 0.}, {50, 200}]];
         xi = LinearProgramming[Range[200], m, Table[0, {50}],
          Method → "InteriorPoint"]; // Timing
Out[44]= {0.012001, Null}

In[45]:= xs = LinearProgramming[Range[200], m, Table[0, {50}],
          Method → "Simplex"]; // Timing
Out[45]= {0.576036, Null}

In[46]:= Range[200].xi - Range[200].xs
Out[46]= 2.14431×10^-7
```

Finding Dual Variables

Given the general linear programming problem

$$\begin{array}{lll} \text{Min} & c^T x & (P) \\ s.t. & A_1 x = b_1 & \\ & A_2 x \geq b_2 & \\ & l \leq x \leq u\,, & \end{array}$$

its dual is

$$\begin{array}{lll} \text{Max} & b^T y + l^T z - u^T w & (D) \\ s.t. & A^T y + z - w = c & \\ & y_2 \geq 0,\ z,\ w \geq 0 & \end{array}$$

It is useful to know solutions for both for some applications.

The relationship between the solutions of the primal and dual problems is given by the following table.

if the primal is	*then the dual problem is*
feasible	feasible
unbounded	infeasible or unbounded
infeasible	unbounded or infeasible

When both problems are feasible, then the optimal values of (P) and (D) are the same, and the following complementary conditions hold for the primal solution `x`, and dual solution `y, z`.

$$(A_2\, x - b_2)^T\, y_2 = 0$$
$$(l - x^*)^T\, z^* = (u - x^*)^T\, w^* = 0$$

`DualLinearProgramming` returns a list `{x, y, z, w}`.

This solves the primal problem

$$\begin{array}{ll} \text{Min} & 3\,x_1 + 4\,x_2 \\ s.t. & x_1 + 2\,x_2 \geq 5 \\ & 1 \leq x_1 \leq 4,\ 1 \leq x_2 \leq 4, \end{array}$$

as well as the dual problem

$$\begin{array}{ll} \text{Max} & 5\,y_1 + z_1 + z_2 - 4\,w_1 - 4\,w_2 \\ s.t. & y_1 + z_1 - w_1 = 3 \\ & 2\,y_1 + z_2 - w_2 = 4 \\ & y_1,\ z_1,\ z_2,\ w_1,\ w_2 \geq 0 \end{array}$$

```
In[14]:= {x, y, z, w} = DualLinearProgramming[{3, 4}, {{1, 2}}, {5}, {{1, 4}, {1, 4}}]
Out[14]= {{1, 2}, {2}, {1, 0}, {0, 0}}
```

This confirms that the primal and dual give the same objective value.

```
In[15]:= {3, 4}.x
Out[15]= 11
```

```
In[16]:= {5, 1, 1, -4, -4}.Flatten[{y, z, w}]
Out[16]= 11
```

The dual of the constraint is `y = {2.}`, which means that for one unit of increase in the right-hand side of the constraint, there will be 2 units of increase in the objective. This can be confirmed by perturbing the right-hand side of the constraint by 0.001.

```
In[17]:= {x, y, z, w} = DualLinearProgramming[{3, 4}, {{1, 2}}, {5 + 0.001}, {{1, 4}, {1, 4}}]
Out[17]= {{1., 2.0005}, {2.}, {1., 0.}, {0., 0.}}
```

Indeed the objective value increases by twice that amount.

```
In[18]:= {3, 4}.x - 11
Out[18]= 0.002
```

Dealing with Infeasibility and Unboundedness in the Interior Point Method

The primal-dual interior point method is designed for feasible problems; for infeasible/unbounded problems it will diverge, and with the current implementation, it is difficult to find out if the divergence is due to infeasibility, or unboundedness.

A heuristic catches infeasible/unbounded problems and issues a suitable message.

```
In[19]:= LinearProgramming[{1., 1}, {{1, 1}, {1, 1}},
          {{1, -1}, {2, 1}}, Method → "InteriorPoint"]

   LinearProgramming::lpsnf : No solution can be found that satisfies the constraints. ≫

Out[19]= LinearProgramming[{1., 1}, {{1, 1}, {1, 1}}, {{1, -1}, {2, 1}}, Method → InteriorPoint]
```

Sometimes the heuristic cannot tell with certainty if a problem is infeasible or unbounded.

```
In[20]:= LinearProgramming[{-1., -1.}, {{1., 1.}}, {1.}, Method → "InteriorPoint"]

   LinearProgramming::lpdinf :
     The dual of this problem is infeasible, which implies that this problem is either
       unbounded or infeasible. Setting the option Method –> Simplex should give a
       more definite answer, though large problems may take longer computing time. ≫

Out[20]= LinearProgramming[{-1., -1.}, {{1., 1.}}, {1.}, Method → InteriorPoint]
```

Using the `Simplex` method as suggested by the message shows that the problem is unbounded.

```
In[21]:= LinearProgramming[{-1., -1.}, {{1., 1.}}, {1.}, Method → "Simplex"]

   LinearProgramming::lpsub : This problem is unbounded.

Out[21]= {Indeterminate, Indeterminate}
```

The Method Options of "InteriorPoint"

`"TreatDenseColumn"` is a method option of `"InteriorPoint"` that decides if dense columns are to be treated separately. Dense columns are columns of the constraint matrix that have many nonzero elements. By default, this method option has the value `Automatic`, and dense columns are treated separately.

Large problems that contain dense columns typically benefit from dense column treatment.

```
In[95]:= A = SparseArray[{{i_, i_} → 1., {i_, 1} → 1.}, {300, 300}];
         c = Table[1, {300}];
         b = A.Range[300];
```

```
In[98]:= {x1 = LinearProgramming[c, A, b, Method → "InteriorPoint"]; // Timing,
          x2 = LinearProgramming[c, A, b,
               Method → {"InteriorPoint", "TreatDenseColumns" → False}]; // Timing}
Out[98]= {{0.028001, Null}, {0.200013, Null}}

In[99]:= x1.c - x2.c
Out[99]= 4.97948 × 10^-11
```

Importing Large Datasets and Solving Large-Scale Problems

A commonly used format for documenting linear programming problems is the Mathematical Programming System (MPS) format. This is a text format consisting of a number of sections.

Importing MPS Formatted Files in Equation Form

Mathematica is able to import `MPS` formatted files. By default, `Import` of `MPS` data returns a linear programming problem in equation form, which can then be solved using `Minimize` or `NMinimize`.

This solves the linear programming problem specified by `MPS` file `"afiro.mps"`.

```
In[25]:= p = Import["Optimization/Data/afiro.mps"]
```

```
Out[25]= {{-0.4 X02_MPS - 0.32 X14_MPS - 0.6 X23_MPS - 0.48 X36_MPS + 10. X39_MPS,
    -1. X01_MPS + 1. X02_MPS + 1. X03_MPS == 0. && -1.06 X01_MPS + 1. X04_MPS == 0. && 1. X01_MPS ≤ 80. &&
     -1. X02_MPS + 1.4 X14_MPS ≤ 0. && -1. X06_MPS - 1. X07_MPS - 1. X08_MPS - 1. X09_MPS + 1. X14_MPS + 1. X15_MPS == 0. &&
     -1.06 X06_MPS - 1.06 X07_MPS - 0.96 X08_MPS - 0.86 X09_MPS + 1. X16_MPS == 0. && 1. X06_MPS - 1. X10_MPS ≤ 80. &&
     1. X07_MPS - 1. X11_MPS ≤ 0. && 1. X08_MPS - 1. X12_MPS ≤ 0. && 1. X09_MPS - 1. X13_MPS ≤ 0. &&
     -1. X22_MPS + 1. X23_MPS + 1. X24_MPS + 1. X25_MPS == 0. && -0.43 X22_MPS + 1. X26_MPS == 0. && 1. X22_MPS ≤ 500. &&
     -1. X23_MPS + 1.4 X36_MPS ≤ 0. && -0.43 X28_MPS - 0.43 X29_MPS - 0.39 X30_MPS - 0.37 X31_MPS + 1. X38_MPS == 0. &&
     1. X28_MPS + 1. X29_MPS + 1. X30_MPS + 1. X31_MPS - 1. X36_MPS + 1. X37_MPS + 1. X39_MPS == 44. &&
     1. X28_MPS - 1. X32_MPS ≤ 500. && 1. X29_MPS - 1. X33_MPS ≤ 0. && 1. X30_MPS - 1. X34_MPS ≤ 0. &&
     1. X31_MPS - 1. X35_MPS ≤ 0. && 2.364 X10_MPS + 2.386 X11_MPS + 2.408 X12_MPS + 2.429 X13_MPS - 1. X25_MPS +
       2.191 X32_MPS + 2.219 X33_MPS + 2.249 X34_MPS + 2.279 X35_MPS ≤ 0. && -1. X03_MPS + 0.109 X22_MPS ≤ 0. &&
     -1. X15_MPS + 0.109 X28_MPS + 0.108 X29_MPS + 0.108 X30_MPS + 0.107 X31_MPS ≤ 0. &&
     0.301 X01_MPS - 1. X24_MPS ≤ 0. && 0.301 X06_MPS + 0.313 X07_MPS + 0.313 X08_MPS + 0.326 X09_MPS - 1. X37_MPS ≤ 0. &&
     1. X04_MPS + 1. X26_MPS ≤ 310. && 1. X16_MPS + 1. X38_MPS ≤ 300. && X01_MPS ≥ 0 && X02_MPS ≥ 0 && X03_MPS ≥ 0 &&
     X04_MPS ≥ 0 && X06_MPS ≥ 0 && X07_MPS ≥ 0 && X08_MPS ≥ 0 && X09_MPS ≥ 0 && X10_MPS ≥ 0 && X11_MPS ≥ 0 &&
     X12_MPS ≥ 0 && X13_MPS ≥ 0 && X14_MPS ≥ 0 && X15_MPS ≥ 0 && X16_MPS ≥ 0 && X22_MPS ≥ 0 && X23_MPS ≥ 0 &&
     X24_MPS ≥ 0 && X25_MPS ≥ 0 && X26_MPS ≥ 0 && X28_MPS ≥ 0 && X29_MPS ≥ 0 && X30_MPS ≥ 0 && X31_MPS ≥ 0 &&
     X32_MPS ≥ 0 && X33_MPS ≥ 0 && X34_MPS ≥ 0 && X35_MPS ≥ 0 && X36_MPS ≥ 0 && X37_MPS ≥ 0 && X38_MPS ≥ 0 && X39_MPS ≥ 0},
   {X01_MPS, X02_MPS, X03_MPS, X04_MPS, X06_MPS, X07_MPS, X08_MPS, X09_MPS, X10_MPS, X11_MPS, X12_MPS,
    X13_MPS, X14_MPS, X15_MPS, X16_MPS, X22_MPS, X23_MPS, X24_MPS, X25_MPS, X26_MPS, X28_MPS, X29_MPS,
    X30_MPS, X31_MPS, X32_MPS, X33_MPS, X34_MPS, X35_MPS, X36_MPS, X37_MPS, X38_MPS, X39_MPS}}
```

```
In[26]:= NMinimize @@ p
```

```
Out[26]= {-464.753, {X01_MPS → 80., X02_MPS → 25.5, X03_MPS → 54.5, X04_MPS → 84.8, X06_MPS → 18.2143, X07_MPS → 0.,
    X08_MPS → 0., X09_MPS → 0., X10_MPS → 0., X11_MPS → 0., X12_MPS → 0., X13_MPS → 0., X14_MPS → 18.2143,
    X15_MPS → 0., X16_MPS → 19.3071, X22_MPS → 500., X23_MPS → 475.92, X24_MPS → 24.08, X25_MPS → 0.,
    X26_MPS → 215., X28_MPS → 0., X29_MPS → 0., X30_MPS → 0., X31_MPS → 0., X32_MPS → 0., X33_MPS → 0.,
    X34_MPS → 0., X35_MPS → 0., X36_MPS → 339.943, X37_MPS → 383.943, X38_MPS → 0., X39_MPS → 0.}}
```

Large-Scale Problems: Importing in Matrix and Vector Form

For large-scale problems, it is more efficient to import the MPS data file and represent the linear programming using matrices and vectors, then solve using LinearProgramming.

This shows that for MPS formatted data, the following 3 elements can be imported.

```
In[101]:= p = Import["Optimization/Data/ganges.mps", "Elements"]
Out[101]= {ConstraintMatrix, Equations, LinearProgrammingData}
```

This imports the problem "ganges", with 1309 constraints and 1681 variables, in a form suitable for LinearProgramming.

```
In[102]:= {c, A, b, bounds} =
            Import["Optimization/Data/ganges.mps", "LinearProgrammingData"];
```

This solves the problem and finds the optimal value.

```
In[103]:= x = LinearProgramming[c, A, b, bounds];

In[104]:= c.x
Out[104]= -109586.
```

The "ConstraintMatrix" specification can be used to get the sparse constraint matrix only.

```
In[105]:= p = Import["Optimization/Data/ganges.mps", "ConstraintMatrix"]
Out[105]= SparseArray[<6912>, {1309, 1681}]
```

Free Formatted MPS Files

Standard MPS formatted files use a fixed format, where each field occupies a strictly fixed character position. However some modeling systems output MPS files with a free format, where fields are positioned freely. For such files, the option "FreeFormat" -> True can be specified for Import.

This string describes an MPS file in free format.

```
In[122]:= txt =
            "NAME TESTPROB\nROWS\n N COST\n L CON1\n G CON2\n E CON3 \nCOLUMNS\n x COST
              1 CON1 2\n x CON2 3\n y COST 4 CON1 5\n y CON3 6\n Z COST
              7 CON2 8\n Z CON3 9\nRHS\n RHS1 CON1 10 CON2 11\n RHS1 CON3
              12\nBOUNDS\n UP BND1 x 13\n LO BND1 y 14\n UP BND1 y 15\nENDATA\n";
```

This gets a temporary file name, and exports the string to the file.

```
In[123]:= file = Close[OpenWrite[]];
          Export[file, txt, "Text"];
```

This imports the file, using the "FreeFormat" -> True option.

```
In[126]:= Import[file, "MPS", "FreeFormat" → True]
```

```
Out[126]= {{1. x_MPS + 4. y_MPS + 7. z_MPS, 2. x_MPS + 5. y_MPS ≤ 10. && 3. x_MPS + 8. z_MPS ≥ 11. && 6. y_MPS + 9. z_MPS == 12. &&
             x_MPS ≥ 0 && x_MPS ≤ 13. && y_MPS ≥ 14. && y_MPS ≤ 15. && z_MPS ≥ 0}, {x_MPS, y_MPS, z_MPS}}
```

Linear Programming Test Problems

Through the ExampleData function, all NetLib linear programming test problems can be accessed.

This finds all problems in the Netlib set.

```
In[12]:= ExampleData["LinearProgramming"]
```

```
Out[12]= {{LinearProgramming, 25fv47}, {LinearProgramming, 80bau3b},
          {LinearProgramming, adlittle}, {LinearProgramming, afiro}, {LinearProgramming, agg},
          {LinearProgramming, agg2}, {LinearProgramming, agg3}, {LinearProgramming, bandm},
          {LinearProgramming, beaconfd}, {LinearProgramming, blend}, {LinearProgramming, bnl1},
          {LinearProgramming, bnl2}, {LinearProgramming, boeing1}, {LinearProgramming, boeing2},
          {LinearProgramming, bore3d}, {LinearProgramming, brandy}, {LinearProgramming, capri},
          {LinearProgramming, cre-a}, {LinearProgramming, cre-b}, {LinearProgramming, cre-c},
          {LinearProgramming, cre-d}, {LinearProgramming, cycle}, {LinearProgramming, czprob},
          {LinearProgramming, d2q06c}, {LinearProgramming, d6cube}, {LinearProgramming, degen2},
          {LinearProgramming, degen3}, {LinearProgramming, dfl001}, {LinearProgramming, e226},
          {LinearProgramming, etamacro}, {LinearProgramming, fffff800}, {LinearProgramming, finnis},
          {LinearProgramming, fit1d}, {LinearProgramming, fit1p}, {LinearProgramming, fit2d},
          {LinearProgramming, fit2p}, {LinearProgramming, forplan}, {LinearProgramming, ganges},
          {LinearProgramming, gfrd-pnc}, {LinearProgramming, greenbea}, {LinearProgramming, greenbeb},
          {LinearProgramming, grow15}, {LinearProgramming, grow22}, {LinearProgramming, grow7},
          {LinearProgramming, infeas/bgdbg1}, {LinearProgramming, infeas/bgetam},
          {LinearProgramming, infeas/bgindy}, {LinearProgramming, infeas/bgprtr},
          {LinearProgramming, infeas/box1}, {LinearProgramming, infeas/ceria3d},
          {LinearProgramming, infeas/chemcom}, {LinearProgramming, infeas/cplex1},
          {LinearProgramming, infeas/cplex2}, {LinearProgramming, infeas/ex72a},
          {LinearProgramming, infeas/ex73a}, {LinearProgramming, infeas/forest6},
          {LinearProgramming, infeas/galenet}, {LinearProgramming, infeas/gosh},
          {LinearProgramming, infeas/gran}, {LinearProgramming, infeas/greenbea},
          {LinearProgramming, infeas/itest2}, {LinearProgramming, infeas/itest6},
          {LinearProgramming, infeas/klein1}, {LinearProgramming, infeas/klein2},
          {LinearProgramming, infeas/klein3}, {LinearProgramming, infeas/mondou2},
          {LinearProgramming, infeas/pang}, {LinearProgramming, infeas/pilot4i},
          {LinearProgramming, infeas/qual}, {LinearProgramming, infeas/reactor},
          {LinearProgramming, infeas/refinery}, {LinearProgramming, infeas/vol1},
          {LinearProgramming, infeas/woodinfe}, {LinearProgramming, israel}, {LinearProgramming, kb2},
          {LinearProgramming, ken-07}, {LinearProgramming, ken-11}, {LinearProgramming, ken-13},
          {LinearProgramming, ken-18}, {LinearProgramming, lotfi}, {LinearProgramming, maros},
          {LinearProgramming, maros-r7}, {LinearProgramming, modszk1}, {LinearProgramming, nesm},
          {LinearProgramming, osa-07}, {LinearProgramming, osa-14}, {LinearProgramming, osa-30},
          {LinearProgramming, osa-60}, {LinearProgramming, pds-02}, {LinearProgramming, pds-06},
          {LinearProgramming, pds-10}, {LinearProgramming, pds-20}, {LinearProgramming, perold},
          {LinearProgramming, pilot}, {LinearProgramming, pilot4}, {LinearProgramming, pilot87},
          {LinearProgramming, pilot.ja}, {LinearProgramming, pilotnov}, {LinearProgramming, pilot.we},
          {LinearProgramming, recipe}, {LinearProgramming, sc105}, {LinearProgramming, sc205},
          {LinearProgramming, sc50a}, {LinearProgramming, sc50b}, {LinearProgramming, scagr25},
```

```
{LinearProgramming, scagr7}, {LinearProgramming, scfxm1}, {LinearProgramming, scfxm2},
{LinearProgramming, scfxm3}, {LinearProgramming, scorpion}, {LinearProgramming, scrs8},
{LinearProgramming, scsd1}, {LinearProgramming, scsd6}, {LinearProgramming, scsd8},
{LinearProgramming, sctap1}, {LinearProgramming, sctap2}, {LinearProgramming, sctap3},
{LinearProgramming, seba}, {LinearProgramming, share1b}, {LinearProgramming, share2b},
{LinearProgramming, shell}, {LinearProgramming, ship04l}, {LinearProgramming, ship04s},
{LinearProgramming, ship08l}, {LinearProgramming, ship08s}, {LinearProgramming, ship12l},
{LinearProgramming, ship12s}, {LinearProgramming, sierra}, {LinearProgramming, stair},
{LinearProgramming, standata}, {LinearProgramming, standgub}, {LinearProgramming, standmps},
{LinearProgramming, stocfor1}, {LinearProgramming, stocfor2}, {LinearProgramming, tuff},
{LinearProgramming, vtp.base}, {LinearProgramming, wood1p}, {LinearProgramming, woodw}}
```

This imports the problem "afiro" and solves it.

In[8]:=
```
ExampleData[{"LinearProgramming", "afiro"}]
```

Out[8]=
```
{{0, -0.4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.32, 0, 0, 0, -0.6, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, -0.48, 0, 0, 10.}, SparseArray[<83>, {27, 32}],
 {{0., 0}, {0., 0}, {80., -1}, {0., -1}, {0., 0}, {0., 0}, {80., -1},
  {0., -1}, {0., -1}, {0., -1}, {0., 0}, {0., 0}, {500., -1}, {0., -1},
  {0., 0}, {44., 0}, {500., -1}, {0., -1}, {0., -1}, {0., -1}, {0., -1},
  {0., -1}, {0., -1}, {0., -1}, {0., -1}, {310., -1}, {300., -1}},
 {{0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞},
  {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞},
  {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}, {0, ∞}}}
```

In[9]:=
```
LinearProgramming @@ %
```

Out[9]=
```
{80., 25.5, 54.5, 84.8, 18.2143, 0., 0., 0., 0., 0., 0., 0., 18.2143, 0., 19.3071, 500.,
 475.92, 24.08, 0., 215., 0., 0., 0., 0., 0., 0., 0., 0., 339.943, 383.943, 0., 0.}
```

This shows other properties that can be imported for the "afiro" problem.

In[10]:=
```
ExampleData[{"LinearProgramming", "afiro"}, "Properties"]
```

Out[10]=
```
{Collection, ConstraintMatrix, Dimensions, Equations, LinearProgrammingData, Name, Source}
```

This imports "afiro" in equation form.

In[11]:=
```
ExampleData[{"LinearProgramming", "afiro"}, "Equations"]
```

Out[11]=
$$\begin{aligned}
&\{\{-0.4\,X02_{MPS} - 0.32\,X14_{MPS} - 0.6\,X23_{MPS} - 0.48\,X36_{MPS} + 10.\,X39_{MPS},\\
&\quad -1.\,X01_{MPS} + 1.\,X02_{MPS} + 1.\,X03_{MPS} == 0.\ \&\&\ -1.06\,X01_{MPS} + 1.\,X04_{MPS} == 0.\ \&\&\ 1.\,X01_{MPS} \le 80.\ \&\&\\
&\quad -1.\,X02_{MPS} + 1.4\,X14_{MPS} \le 0.\ \&\&\ -1.\,X06_{MPS} - 1.\,X07_{MPS} - 1.\,X08_{MPS} - 1.\,X09_{MPS} + 1.\,X14_{MPS} + 1.\,X15_{MPS} == 0.\ \&\&\\
&\quad -1.06\,X06_{MPS} - 1.06\,X07_{MPS} - 0.96\,X08_{MPS} - 0.86\,X09_{MPS} + 1.\,X16_{MPS} == 0.\ \&\&\ 1.\,X06_{MPS} - 1.\,X10_{MPS} \le 80.\ \&\&\\
&\quad 1.\,X07_{MPS} - 1.\,X11_{MPS} \le 0.\ \&\&\ 1.\,X08_{MPS} - 1.\,X12_{MPS} \le 0.\ \&\&\ 1.\,X09_{MPS} - 1.\,X13_{MPS} \le 0.\ \&\&\\
&\quad -1.\,X22_{MPS} + 1.\,X23_{MPS} + 1.\,X24_{MPS} + 1.\,X25_{MPS} == 0.\ \&\&\ -0.43\,X22_{MPS} + 1.\,X26_{MPS} == 0.\ \&\&\ 1.\,X22_{MPS} \le 500.\ \&\&\\
&\quad -1.\,X23_{MPS} + 1.4\,X36_{MPS} \le 0.\ \&\&\ -0.43\,X28_{MPS} - 0.43\,X29_{MPS} - 0.39\,X30_{MPS} - 0.37\,X31_{MPS} + 1.\,X38_{MPS} == 0.\ \&\&\\
&\quad 1.\,X28_{MPS} + 1.\,X29_{MPS} + 1.\,X30_{MPS} + 1.\,X31_{MPS} - 1.\,X36_{MPS} + 1.\,X37_{MPS} + 1.\,X39_{MPS} == 44.\ \&\&\\
&\quad 1.\,X28_{MPS} - 1.\,X32_{MPS} \le 500.\ \&\&\ 1.\,X29_{MPS} - 1.\,X33_{MPS} \le 0.\ \&\&\ 1.\,X30_{MPS} - 1.\,X34_{MPS} \le 0.\ \&\&\\
&\quad 1.\,X31_{MPS} - 1.\,X35_{MPS} \le 0.\ \&\&\ 2.364\,X10_{MPS} + 2.386\,X11_{MPS} + 2.408\,X12_{MPS} + 2.429\,X13_{MPS} - 1.\,X25_{MPS} +\\
&\qquad 2.191\,X32_{MPS} + 2.219\,X33_{MPS} + 2.249\,X34_{MPS} + 2.279\,X35_{MPS} \le 0.\ \&\&\ -1.\,X03_{MPS} + 0.109\,X22_{MPS} \le 0.\ \&\&\\
&\quad -1.\,X15_{MPS} + 0.109\,X28_{MPS} + 0.108\,X29_{MPS} + 0.108\,X30_{MPS} + 0.107\,X31_{MPS} \le 0.\ \&\&\\
&\quad 0.301\,X01_{MPS} - 1.\,X24_{MPS} \le 0.\ \&\&\ 0.301\,X06_{MPS} + 0.313\,X07_{MPS} + 0.313\,X08_{MPS} + 0.326\,X09_{MPS} - 1.\,X37_{MPS} \le 0.\ \&\&\\
&\quad 1.\,X04_{MPS} + 1.\,X26_{MPS} \le 310.\ \&\&\ 1.\,X16_{MPS} + 1.\,X38_{MPS} \le 300.\ \&\&\ X01_{MPS} \ge 0\ \&\&\ X02_{MPS} \ge 0\ \&\&\ X03_{MPS} \ge 0\ \&\&\\
&\quad X04_{MPS} \ge 0\ \&\&\ X06_{MPS} \ge 0\ \&\&\ X07_{MPS} \ge 0\ \&\&\ X08_{MPS} \ge 0\ \&\&\ X09_{MPS} \ge 0\ \&\&\ X10_{MPS} \ge 0\ \&\&\ X11_{MPS} \ge 0\ \&\&\\
&\quad X12_{MPS} \ge 0\ \&\&\ X13_{MPS} \ge 0\ \&\&\ X14_{MPS} \ge 0\ \&\&\ X15_{MPS} \ge 0\ \&\&\ X16_{MPS} \ge 0\ \&\&\ X22_{MPS} \ge 0\ \&\&\ X23_{MPS} \ge 0\ \&\&\\
&\quad X24_{MPS} \ge 0\ \&\&\ X25_{MPS} \ge 0\ \&\&\ X26_{MPS} \ge 0\ \&\&\ X28_{MPS} \ge 0\ \&\&\ X29_{MPS} \ge 0\ \&\&\ X30_{MPS} \ge 0\ \&\&\ X31_{MPS} \ge 0\ \&\&\\
&\quad X32_{MPS} \ge 0\ \&\&\ X33_{MPS} \ge 0\ \&\&\ X34_{MPS} \ge 0\ \&\&\ X35_{MPS} \ge 0\ \&\&\ X36_{MPS} \ge 0\ \&\&\ X37_{MPS} \ge 0\ \&\&\ X38_{MPS} \ge 0\ \&\&\ X39_{MPS} \ge 0\},\\
&\{X01_{MPS}, X02_{MPS}, X03_{MPS}, X04_{MPS}, X06_{MPS}, X07_{MPS}, X08_{MPS}, X09_{MPS}, X10_{MPS}, X11_{MPS}, X12_{MPS},\\
&\ X13_{MPS}, X14_{MPS}, X15_{MPS}, X16_{MPS}, X22_{MPS}, X23_{MPS}, X24_{MPS}, X25_{MPS}, X26_{MPS}, X28_{MPS}, X29_{MPS},\\
&\ X30_{MPS}, X31_{MPS}, X32_{MPS}, X33_{MPS}, X34_{MPS}, X35_{MPS}, X36_{MPS}, X37_{MPS}, X38_{MPS}, X39_{MPS}\}\}
\end{aligned}$$

Application Examples of Linear Programming

L1-Norm Minimization

It is possible to solve an l_1 minimization problem

$$\text{Min} \qquad |Ax - b|_1$$

by turning the system into a linear programming problem

$$\begin{aligned} \text{Min} \qquad & z^T e \\ & z \geq Ax - b \\ & z \geq -Ax + b \end{aligned}$$

This defines a function for solving an l_1 minimization problem.

In[35]:=
```
L1Minimization[A_, b_] := Module[
  {B, c, All, ball, x, lb, AT},
  {m, n} = Dimensions[A];
  AT = Transpose[A];
  B = SparseArray[{{i_, i_} → 1}, {m, m}];
  Aall = Join[Transpose[Join[B, -AT]], Transpose[Join[B, AT]]];
  ball = Join[-b, b];
  c = Join[Table[1, {m}], Table[0, {n}]];
  lb = Table[-Infinity, {m + n}];
  x = LinearProgramming[c, Aall, ball, lb];
  x = Drop[x, m]
 ]
```

The following is an over-determined linear system.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 5 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

A simple application of `LinearSolve` would fail.

In[36]:=
```
A = SparseArray[{{1, 2, 3}, {5, 6, 7}, {7, 8, 9}, {10, 11, 12}}];
b = {1, 2, 3, 4};
LinearSolve[A, b]
```

LinearSolve::nosol : Linear equation encountered that has no solution. ≫

Out[38]=
```
LinearSolve[SparseArray[<12>, {4, 3}], {1, 2, 3, 4}]
```

This finds the l_1 minimization solution.

```
In[39]:= x = L1Minimization[A, b]
```

$$\text{Out[39]}= \left\{0,\ 0,\ \frac{1}{3}\right\}$$

```
In[40]:= {Norm[A.x - b, 1], Norm[A.x - b, 2]} // N
Out[40]= {0.333333, 0.333333}
```

The least squares solution can be found using `PseudoInverse`. This gives a large l_1 norm, but a smaller l_2 norm.

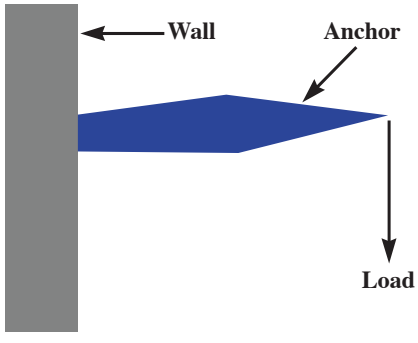
```
In[41]:= x2 = PseudoInverse[A].b
```

$$\text{Out[41]}= \left\{\frac{4}{513},\ \frac{58}{513},\ \frac{112}{513}\right\}$$

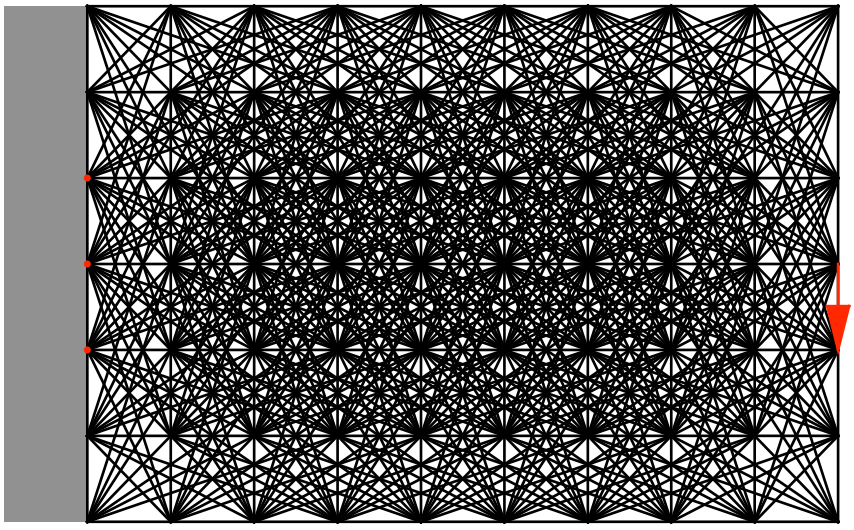
```
In[42]:= {Norm[A.x2 - b, 1], Norm[A.x2 - b, 2]} // N
Out[42]= {0.491228, 0.286132}
```

Design of an Optimal Anchor

The example is adopted from [2]. The aim is to design an anchor that uses as little material as possible to support a load.



This problem can be modeled by discretizing and simulating it using nodes and links. The modeling process is illustrated using the following figure. Here a grid of 7×10 nodes is generated. Each node is then connected by a link to all other nodes that are of Manhattan distance of less than or equal to three. The three red nodes are assumed to be fixed to the wall, while on all other nodes, compression and tension forces must balance.



Each link represents a rigid rod that has a thickness, with its weight proportional to the force on it and its length. The aim is to minimize the total material used, which is

$$\text{Material needed to construct a link} = \text{force} * \text{link\_length}$$

$$\text{subject to force balance on each node except the fixed ones.}$$

Hence mathematically this is a linearly constrained minimization problem, with objective function a sum of absolute values of linear functions.

$$\text{Minimize} \sum \left| \text{force} * \text{link\_length} \right|$$

$$\text{subject to force balance on every unanchored node.}$$

The absolute values $\left|\text{force} * \text{link\_length}\right|$ in the objective function can be replaced by breaking down `force` into a combination of compression and tension forces, with each non-negative. Thus assume `E` is the set of links, `V` the set of nodes, $\texttt{l}_{\texttt{ij}}$ the length of link between nodes `i` and `j`, $\texttt{c}_{\texttt{ij}}$and $\texttt{t}_{\texttt{ij}}$ the compression and tension forces on the link; then the above model can be converted to a linear programming problem

$$\text{Minimize} \sum_{\{i,j\} \in E} \left(c_{\text{ij}} + t_{\text{ij}}\right) l_{\text{ij}}$$

$$\text{subject to} \sum_{\{i,k\} \in E} \left(t_{\text{ik}} - c_{\text{ik}}\right) = \text{load}_i,\ t_{\text{ij}},\ c_{\text{ij}} \geq 0,\ \text{for all } i \in V \text{ and } \{i, j\} \in E.$$

The following sets up the model, solves it, and plots the result; it is based on an AMPL model [2].

In[1]:=
```
OptimalAnchorDesign[X_, Y_, ANCHORS_, forcepoints_, dist_: 3] :=
  Module[{a, c, ldist, p, NODES, UNANCHORED, setx, sety, length, xload, yload,
    nnodes, getarcs, ARCS, comp, comps, tensions, const1, const2, lengths, volume,
    inedges, outedges, nodebalance, const3, vars, totalf, maxf, res, tens,
    setInOutEdges, consts, sol, f, xii, yii, xjj, yjj, t, rhs, ma, obj, m, n},
   Clear[comp, tensions, tens, vars];

   (* need at least 2 nchor points *)
   If[Length[Union[ANCHORS]] == 1, Return[{}]];

   (* A lattice of Nodes *)
   NODES = Partition[Flatten[Outer[List, X, Y]], 2];

   (* these are the nodes near the wall that will be anchored *)
   UNANCHORED = Complement[NODES, ANCHORS];
   (* the many linked exist in the structure that we try to optimize away *)
   setx[{x_, y_}] := (xload[x, y] = 0);
   sety[{x_, y_}] := (yload[x, y] = 0);
   Map[setx, UNANCHORED];
   Map[sety, UNANCHORED];
   Map[(yload[#[[1]], #[[2]]] = -1) &, forcepoints];

   (* get the edges that link nodes with neighbors of distance ≤ 3 *)
   nnodes = Length[NODES];
   getarcs =
    Compile[{{NODES, _Integer, 2}}, Module[{xi, yi, xj, yj, i, j, nn = 0, NN},
      (* we use a two sweep strategy as a nexted list
       would not be allowed to compile by Compile *)
      Do[Do[{xi, yi} = NODES[[i]];
        {xj, yj} = NODES[[j]];
        If[Abs[xj - xi] ≤ dist && Abs[yj - yi] ≤ dist &&
          Abs[GCD[xj - xi, yj - yi]] == 1 && (xi > xj || (xi == xj && yi > yj)), nn++],
        {j, Length[NODES]}], {i, Length[NODES]}];
      NN = Table[{1, 1, 1}, {nn}];
      nn = 1;
      Do[Do[{xi, yi} = NODES[[i]];
        {xj, yj} = NODES[[j]];
        If[Abs[xj - xi] ≤ dist && Abs[yj - yi] ≤ dist &&
          Abs[GCD[xj - xi, yj - yi]] == 1 && (xi > xj || (xi == xj && yi > yj)),
         NN[[nn++]] = {xi, yi, xj, yj}], {j, Length[NODES]}], {i, Length[NODES]}];
      NN]];
   ARCS = Partition[Flatten[getarcs[NODES]], {4}];
   length[{xi_, yi_, xj_, yj_}] := Sqrt[(xj - xi)^2 + (yj - yi)^2] // N;
   (* the variables: compression and tension forces *)
   comps = Map[(comp @@ #) &, ARCS];
   tensions = Map[(tens @@ #) &, ARCS];
   const1 = Thread[Greater[comps, 0]];
   const2 = Thread[Greater[tensions, 0]];
   lengths = Map[(length[#]) &, ARCS] // N;

   (* objective function *)
   volume = lengths.(comps + tensions);

   Map[(inedges[#] = False) &, NODES];
   Map[(outedges[#] = False) &, NODES];
   setInOutEdges[{xi_, yi_, xj_, yj_}] := Module[{},
     If[outedges[{xj, yj}] === False, outedges[{xj, yj}] = {xi, yi, xj, yj},
      outedges[{xj, yj}] = {outedges[{xj, yj}], {xi, yi, xj, yj}}];
     If[inedges[{xi, yi}] === False, inedges[{xi, yi}] = {xi, yi, xj, yj},
```

```
    inedges[{xi, yi}] = {inedges[{xi, yi}], {xi, yi, xj, yj}}];];
Map[(setInOutEdges[#]) &, ARCS];
Map[(inedges[#] = Partition[Flatten[{inedges[#]}], {4}]) &, NODES];
Map[(outedges[#] = Partition[Flatten[{outedges[#]}], {4}]) &, NODES];

nodebalance[{x_, y_}] :=
 Module[{Inedges, Outedges, xforce, yforce}, Inedges = inedges[{x, y}];
  Outedges = outedges[{x, y}];
  xforce[{xi_, yi_, xj_, yj_}] := ((xj - xi) / length[{xi, yi, xj, yj}]) *
    (comp[xi, yi, xj, yj] - tens[xi, yi, xj, yj]);
  yforce[{xi_, yi_, xj_, yj_}] := ((yj - yi) / length[{xi, yi, xj, yj}]) *
    (comp[xi, yi, xj, yj] - tens[xi, yi, xj, yj]);
  (* constraints *)
  {Total[Map[xforce, Inedges]] - Total[Map[xforce, Outedges]] == xload[x, y],
   Total[Map[yforce, Inedges]] - Total[Map[yforce, Outedges]] == yload[x, y]}
 ];
const3 = Flatten[Map[nodebalance[#] &, UNANCHORED]];
(* assemble the variables and constraints, and solve *)


vars = Union[Flatten[{comps, tensions}]];
{rhs, ma} = CoefficientArrays[const3, vars];
obj = CoefficientArrays[volume, vars][[2]];
{m, n} = Dimensions[ma];
Print["Number of variables = ", n, " number of constraints = ", m];

(* solve *)
t = Timing[sol = LinearProgramming[obj, ma,
      Transpose[{-rhs, Table[0, {m}]}], Table[{0, Infinity}, {n}]];];
Print["CPU time = ", t[[1]], " Seconds"];
Map[Set @@ # &, Transpose[{vars, sol}]];

(* Now add up the total force on all links,
and scale them to be between 0 and 1. *)
maxf = Max[comps + tensions];
Evaluate[Map[totalf[#] &, ARCS]] = (comps + tensions) / maxf;

(* Now we plot the links that has a force at least 0.001 and
 get the optimal design of the anchor. We color code the drawing
 so that red means a large force and blue a small one. Also,
links with large forces are drawn thinker than those with small forces. *)

res = {EdgeForm[Black], White,
  Polygon[{{0, 0}, {0, Length[Y]}, {1, Length[Y]}, {1, 0}}],
  Map[({xii, yii, xjj, yjj} = #; f = totalf[{xii, yii, xjj, yjj}];
      If[f > 0.001, {Hue[.7 * (1 - f)], Thickness[.02 Sqrt[f]],
        Line[{{xii, yii}, {xjj, yjj}}]}, {}]) &, ARCS], GrayLevel[.5],
  PointSize[0.04], {Black, Map[{Arrow[{#, # + {0, -4}}]} &, forcepoints]},
  Map[Point, ANCHORS]};
Graphics[res]
];
```

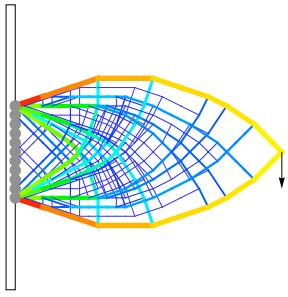
This solves the problem by placing 30 nodes in the horizontal and vertical directions.

In[2]:=
```
m = 30;(* y direction. *)
n = 30;(* x direction. *)
X = Table[i, {i, 0, n}];
Y = Table[i, {i, 0, m}];
res = OptimalAnchorDesign[X, Y,
  Table[{1, i}, {i, Round[m / 3], Round[m / 3 * 2]}], {{n, m / 2}}, 3]
```

```
Number of variables = 27 496 number of constraints = 1900

CPU time = 4.8123 Seconds
```

Out[6]=



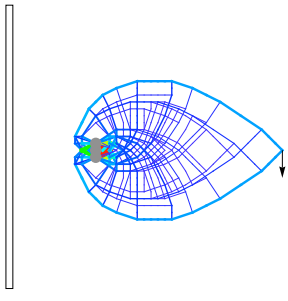
If, however, the anchor is fixed not on the wall, but on some points in space, notice how the results resemble the shape of some leaves. Perhaps the structure of leaves is optimized in the process of evolution.

In[7]:=
```
m = 40;(*must be even*)
n = 40;
X = Table[i, {i, 0, n}];
Y = Table[i, {i, 0, m}];
res = OptimalAnchorDesign[X, Y,
  Table[{Round[n / 3], i}, {i, Round[m / 2] - 1, Round[m / 2] + 1}], {{n, m / 2}}, 3]
```

```
Number of variables = 49 456 number of constraints = 3356

CPU time = 9.83262 Seconds
```

Out[11]=



Algorithms for Linear Programming

Simplex and Revised Simplex Algorithms

The simplex and revised simplex algorithms solve linear programming problems by constructing a feasible solution at a vertex of the polytope defined by the constraints, and then moving along the edges of the polytope to vertices with successively smaller values of the objective function until the minimum is reached.

Although the sparse implementation of simplex and revised algorithms are quite efficient in practice, and are guaranteed to find the global optimum, they have a poor worst-case behavior: it is possible to construct a linear programming problem for which the simplex or revised simplex method takes a number of steps exponential in the problem size.

Mathematica implements simplex and revised simplex algorithms using dense linear algebra. The unique feature of this implementation is that it is possible to solve exact/extended precision problems. Therefore these methods are more suitable for small-sized problems for which non-machine number results are needed.

This sets up a random linear programming problem with 20 constraints and 200 variables.

In[12]:=
```
SeedRandom[123];
{m, n} = {20, 200};
c = Table[RandomInteger[{1, 10}], {n}];
A = Table[RandomInteger[{-100, 100}], {m}, {n}];
b = A.Table[1, {n}];
bounds = Table[{-10, 10}, {n}];
```

This solves the problem. Typically, for a linear programming problem with many more variables than constraints, the revised simplex algorithm is faster. On the other hand, if there are many more constraints than variables, the simplex algorithm is faster.

In[25]:=
```
t = Timing[x = LinearProgramming[c, A, b, bounds, Method → "Simplex"];];
Print["time = ", t[[1]], " optimal value = ", c.x, " or ", N[c.x]]
```

```
time = 14.7409 optimal value =
  1 151 274 037 058 983 869 972 777 363
 -------------------------------------- or -10 935.
     105 283 309 229 356 027 027 010
```

In[26]:=
```
t = Timing[x = LinearProgramming[c, A, b, bounds, Method → "RevisedSimplex"];];
Print["time = ", t[[1]], " optimal value = ", c.x, " or ", N[c.x]]
```

```
time = 6.3444 optimal value =
   1 151 274 037 058 983 869 972 777 363
 - ------------------------------------ or -10 935.
     105 283 309 229 356 027 027 010
```

If only machine-number results are desired, then the problem should be converted to machine numbers, and the interior point algorithm should be used.

In[20]:=
```
t = Timing[
  x = LinearProgramming[N[c], N[A], N[b], N[bounds], Method → "InteriorPoint"];];
Print["time = ", t, " optimal value = ", c.x]
```

```
time = {0.036002, Null} optimal value = -10 935.
```

Interior Point Algorithm

Although the simplex and revised simplex algorithms can be quite efficient on average, they have a poor worst-case behavior. It is possible to construct a linear programming problem for which the simplex or revised simplex methods take a number of steps exponential in the problem size. The interior point algorithm, however, has been proven to converge in a number of steps that are polynomial in the problem size.

Furthermore, the *Mathematica* simplex and revised simplex implementation use dense linear algebra, while its interior point implementation uses machine-number sparse linear algebra. Therefore for large-scale, machine-number linear programming problems, the interior point method is more efficient and should be used.

Interior Point Formulation

Consider the standardized linear programming problem

$$\text{Min } c^T x,\ s.t.\ A x = b,\ x \geq 0,$$

where $c,\ x \in R^n$, $A \in R^{m \times n}$, $b \in R^m$. This problem can be solved using a barrier function formulation to deal with the positive constraints

$$\text{Min } c^T x - t \sum_{i=1}^{n} \ln(x_i),\ s.t.\ A x = b,\ x \geq 0,\ t > 0,\ t \to 0$$

The first-order necessary condition for the above problem gives

$$c - t X^{-1} e = A^T y, \text{ and } A x = b,\ x \geq 0$$

Let X denote the diagonal matrix made of the vector x, and $z = t\,X^{-1}\,e$.

$$\begin{aligned} x\,z &= t\,e \\ A^T\,y + z &= c \\ A\,x &= b \\ x,\ z &\geq 0 \end{aligned}$$

This is a set of $2\,m + n$ linear/nonlinear equations with constraints. It can be solved using Newton's method

$$(x,\ y,\ z) := (x,\ y,\ z) + (\Delta\mathrm{x}, \Delta\mathrm{y},\ \Delta\mathrm{z})$$

with

$$\begin{pmatrix} X & Z & 0 \\ I & 0 & A^T \\ 0 & A & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathrm{z} \\ \Delta\mathrm{x} \\ \Delta\mathrm{y} \end{pmatrix} = \begin{pmatrix} t\,e - x\,z \\ c - A^T\,y - z \\ b - A\,x \end{pmatrix}.$$

One way to solve this linear system is to use Gaussian elimination to simplify the matrix into block triangular form.

$$\begin{pmatrix} X & Z & 0 \\ I & 0 & A^T \\ 0 & A & 0 \end{pmatrix} \rightarrow \begin{pmatrix} X & Z & 0 \\ 0 & X^{-1}\,Z & A^T \\ 0 & A & 0 \end{pmatrix} \rightarrow \begin{pmatrix} X & Z & 0 \\ 0 & X^{-1}\,Z & A^T \\ 0 & 0 & A\,Z^{-1}\,X\,A^T \end{pmatrix}$$

To solve this block triangular matrix, the so-called normal system can be solved, with the matrix in this normal system

$$B = A\,Z^{-1}\,X\,A^T$$

This matrix is positive definite, but becomes very ill-conditioned as the solution is approached. Thus numerical techniques are used to stabilize the solution process, and typically the interior

point method can only be expected to solve the problem to a tolerance of about $\sqrt{\texttt{\$MachineEpsilon}}$, with tolerance explained in "Convergence Tolerance". *Mathematica* uses Mehrotra's predictor-corrector scheme [1].

Convergence Tolerance

General Linear Programming problems are first converted to the standard form

$$\begin{aligned} &\text{Min } c^T x \\ &\text{subject to } A x = b \\ &\qquad\qquad x \geq 0 \end{aligned}$$

with the corresponding dual

$$\begin{aligned} &\text{Max } b^T y \\ &\text{subject to } A^T y + z = c \\ &\qquad\qquad z \geq 0 \end{aligned}$$

The convergence criterion for the interior point algorithm is

$$\frac{\| b - A x \|}{\max(1, \ \| b \|)} + \frac{\| c - A^T y - z \|}{\max(1, \ \| c \|)} + \frac{\| c^T x - b^T y \|}{\max\left(1, \ \| c^T x \|, \ \| b^T y \|\right)} \leq \text{tolerance}$$

with the tolerance set, by default, to $\sqrt{\texttt{\$MachineEpsilon}}$.

References

[1] Vanderbei, R. *Linear Programming: Foundations and Extensions*. Springer-Verlag, 2001.

[2] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization* 2 (1992): 575-601.

Numerical Nonlinear Local Optimization

Introduction

Numerical algorithms for constrained nonlinear optimization can be broadly categorized into gradient-based methods and direct search methods. Gradient search methods use first derivatives (gradients) or second derivatives (Hessians) information. Examples are the sequential quadratic programming (SQP) method, the augmented Lagrangian method, and the (nonlinear) interior point method. Direct search methods do not use derivative information. Examples are Nelder-Mead, genetic algorithm and differential evolution, and simulated annealing. Direct search methods tend to converge more slowly, but can be more tolerant to the presence of noise in the function and constraints.

Typically, algorithms only build up a local model of the problems. Furthermore, to ensure convergence of the iterative process, many such algorithms insist on a certain decrease of the objective function or of a merit function which is a combination of the objective and constraints. Such algorithms will, if convergent, only find the local optimum, and are called local optimization algorithms. In *Mathematica* local optimization problems can be solved using `FindMinimum`.

Global optimization algorithms, on the other hand, attempt to find the global optimum, typically by allowing decrease as well as increase of the objective/merit function. Such algorithms are usually computationally more expensive. Global optimization problems can be solved exactly using `Minimize` or numerically using `NMinimize`.

This solves a nonlinear programming problem,

$$\begin{aligned} &\text{Min} \quad x - y \\ &s.t \quad -3x^2 + 2xy - y^2 \geq -1 \end{aligned}$$

using `Minimize`, which gives an exact solution.

```
In[1]:= Minimize[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[1]= {-1, {x → 0, y → 1}}
```

This solves the same problem numerically. `NMinimize` returns a machine-number solution.

```
In[2]:= NMinimize[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[2]= {-1., {x → -3.57514×10^-17, y → 1.}}
```

`FindMinimum` numerically finds a local minimum. In this example the local minimum found is also a global minimum.

```
In[3]:= FindMinimum[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[3]= {-1., {x → 2.78301×10^-17, y → 1.}}
```

The FindMinimum Function

`FindMinimum` solves local unconstrained and constrained optimization problems. This document only covers the constrained optimization case. See "Unconstrained Optimization" for details of `FindMinimum` for unconstrained optimization.

This solves a nonlinear programming problem,

$$\begin{array}{ll} \text{Min} & -\frac{100}{(x-1)^2+(y-1)^2+1}-\frac{200}{(x+1)^2+(y+2)^2+1} \\ s.t & x^2+y^2>3 \end{array}$$

using `FindMinimum`.

```
In[4]:= FindMinimum[{-100/((x - 1)^2 + (y - 1)^2 + 1) - 200/((x + 1)^2 + (y + 2)^2 + 1), x^2 + y^2 > 3}, {x, y}]
Out[4]= {-207.16, {x → -0.994861, y → -1.99229}}
```

This provides `FindMinimum` with a starting value of 2 for `x`, but uses the default starting point for `y`.

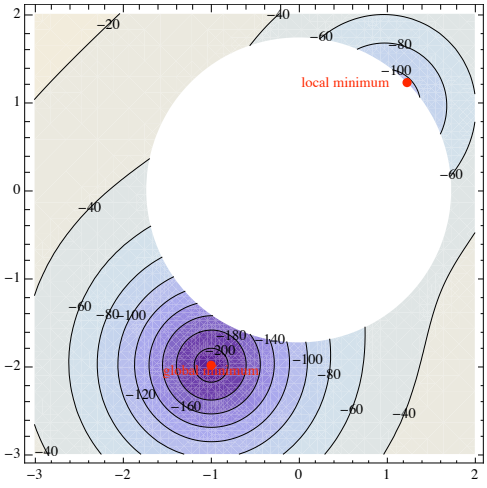
```
In[5]:= FindMinimum[{-100/((x - 1)^2 + (y - 1)^2 + 1) - 200/((x + 1)^2 + (y + 2)^2 + 1), x^2 + y^2 > 3}, {{x, 2}, y}]
Out[5]= {-103.063, {x → 1.23037, y → 1.21909}}
```

The previous solution point is actually a local minimum. `FindMinimum` only attempts to find a local minimum.

This contour plot of the feasible region illustrates the local and global minima.

In[6]:=
```
ContourPlot[-100/((x-1)^2+(y-1)^2+1) - 200/((x+1)^2+(y+2)^2+1), {x, -3, 2},
 {y, -3, 2}, RegionFunction → (#1^2+#2^2 > 3 &), Contours → 10,
 Epilog → ({Red, PointSize[.02], Text["global minimum", {-.995, -2.092}],
    Point[{-.995, -1.992}], Text["local minimum", {0.5304, 1.2191}],
    Point[{1.2304, 1.2191}]}), ContourLabels → True]
```

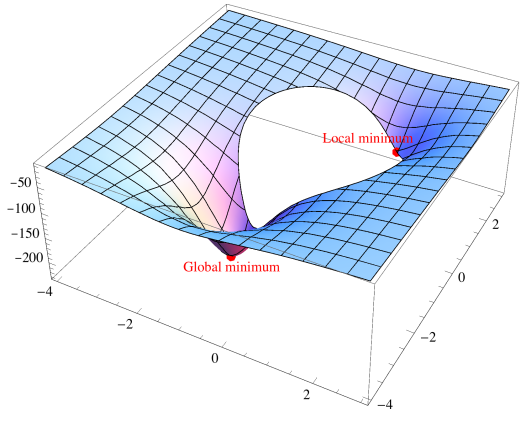
Out[6]=



This is a 3D visualization of the function in its feasible region.

In[21]:=
```
Show[{Plot3D[-100/((x-1)^2+(y-1)^2+1) - 200/((x+1)^2+(y+2)^2+1), {x, -4, 3},
   {y, -4, 3}, RegionFunction → (#1^2+#2^2 > 3 &), PlotRange → All],
  Graphics3D[{Red, PointSize[.02], Text["Global minimum", {-.995, -2.092, -230}],
    Point[{-.995, -2.092, -207}], Text["Local minimum", {0.5304, 1.2191, -93.4}],
    Point[{1.23, 1.22, -103.}]}]}]
```

Out[21]=



Options for FindMinimum

FindMinimum accepts these options.

option name	default value	
AccuracyGoal	Automatic	the accuracy sought
Compiled	Automatic	whether the function and constraints should automatically be compiled
EvaluationMonitor	Automatic	expression to evaluate whenever f is evaluated
Gradient	Automatic	the list of gradient functions {D[f, x], D[f, y], ...}
MaxIterations	Automatic	maximum number of iterations to use
Method	Automatic	method to use
PrecisionGoal	Automatic	the precision sought
StepMonitor	None	expression to evaluate whenever a step is taken
WorkingPrecision	Automatic	the precision used in internal computations

The Method option specifies the method to use to solve the optimization problem. Currently, the only method available for constrained optimization is the interior point algorithm.

This specifies that the interior point method should be used.

```
In[8]:= FindMinimum[{x^2 + y^2, (x - 1)^2 + 2 (y - 1)^2 > 5}, {x, y}, Method → "InteriorPoint"]
Out[8]= {0.149239, {x → -0.150959, y → -0.355599}}
```

MaxIterations specifies the maximum number of iterations that should be used. When machine precision is used for constrained optimization, the default MaxIterations -> 500 is used.

When StepMonitor is specified, it is evaluated once every iterative step in the interior point algorithm. On the other hand, EvaluationMonitor, when specified, is evaluated every time a function or an equality or inequality constraint is evaluated.

This demonstrates that 19 iterations are not sufficient to solve the following problem to the default tolerance. It collects points visited through the use of `StepMonitor`.

In[9]:=
```
pts =
 Reap[sol = FindMinimum[{-100/((x-1)^2+(y-1)^2+1) - 200/((x+1)^2+(y+2)^2+1), x^2+y^2>3},
   {{x, 1.5}, {y, -1}}, MaxIterations → 19, StepMonitor :→ (Sow[{x, y}])];]; sol
```

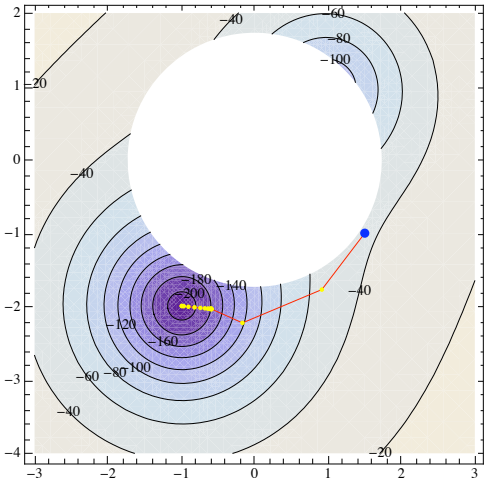
FindMinimum::eit: The algorithm does not converge to the tolerance of 4.806217383937354`*^-6 in 19 iterations. The best estimated solution, with {feasibility residual, KKT residual, complementary residual} of $\{0.000305478, 0.0173304, 1.1484\times10^{-12}\}$, is returned. >>

Out[9]=
```
{-207.16, {x → -0.994818, y → -1.9923}}
```

The points visited are shown using `ContourPlot`. The starting point is blue, the rest yellow.

In[10]:=
```
ContourPlot[-100/((x-1)^2+(y-1)^2+1) - 200/((x+1)^2+(y+2)^2+1), {x, -3, 3},
 {y, -4, 2}, RegionFunction → (#1^2+#2^2>3 &), Contours → 10,
 Epilog → ({Red, PointSize[.01], Line[pts[[2, 1]]], Yellow, Point /@ pts[[2, 1]],
   Blue, PointSize[.02], Point[pts[[2, 1, 1]]]}), ContourLabels → True]
```

Out[10]=



`WorkingPrecision -> prec` specifies that all the calculation in `FindMinimum` is to be carried out at precision *prec*. By default, *prec* = `MachinePrecision`. If *prec* > `MachinePrecision`, a fixed precision of *prec* is used through the computation.

`AccuracyGoal` and `PrecisionGoal` options are used in the following way. By default, `AccuracyGoal -> Automatic`, and is set to *prec* / 3. By default, `PrecisionGoal -> Automatic` and is set to `-Infinity`. `AccuracyGoal -> ga` is the same as `AccuracyGoal -> {-Infinity, ga}`.

Suppose `AccuracyGoal -> {a, ga}` and `PrecisionGoal -> p`, then `FindMinimum` attempts to drive the residual, which is a combination of the feasibility and the satisfaction of the Karush-Kuhn-Tucker (KKT) and complementary conditions, to be less than or equal to $tol = 10^{-ga}$. In addition, it requires the difference between the current and next iterative point, x and x^+, to satisfy $\| x^+ - x \| <= 10^{-a} + 10^{-p} \| x \|$, before terminating.

This computes a solution using a `WorkingPrecision` of 100.

```
In[11]:= sol = FindMinimum[{- 100/((x - 1)^2 + (y - 1)^2 + 1) - 200/((x + 1)^2 + (y + 2)^2 + 1), x^2 + y^2 > 3},
           {x, y}, WorkingPrecision → 100]
```

```
Out[11]= {-207.15989698200872850175933169003419200503000506959008313458370052141625851558970840050348225
           93961079, {x →
           -0.994861334736009401495655384594446803199030436322982571709856118064758198290815887716129
             969329515966, y →
           -1.992292002104014102243483076891670204944778559261548493163022824035671533601903494300753
             03273288575}}
```

The exact optimal value is computed using `Minimize`, and compared with the result of `FindMinimum`.

```
In[12]:= solExact =
           Minimize[{- 100/((x - 1)^2 + (y - 1)^2 + 1) - 200/((x + 1)^2 + (y + 2)^2 + 1), x^2 + y^2 > 3}, {x, y}];
```

```
In[13]:= sol[[1]] - solExact[[1]]
```

```
Out[13]= 8.587396168751353852650510 20 × 10^-71
```

Examples of FindMinimum

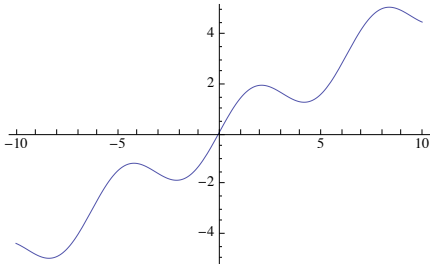
Finding a Global Minimum

If a global minimum is needed, `NMinimize` or `Minimize` should be used. However since each run of `FindMinimum` tends to be faster than `NMinimize` or `Minimize`, sometimes it may be more efficient to use `FindMinimum`, particularly for relatively large problems with a few local minima. `FindMinimum` attempts to find a local minimum, therefore it will terminate if a local minimum is found.

This shows a function with multiple minima within the feasible region of $-10 \le x \le 10$.

```
In[14]:= Plot[Sin[x] + .5 x, {x, -10, 10}]
```

Out[14]=



With the automatic starting point, `FindMinimum` converges to a local minimum.

```
In[15]:= FindMinimum[{Sin[x] + .5 x, -10 <= x <= 10}, {x}]
Out[15]= {-1.91322, {x → -2.0944}}
```

If the user has some knowledge of the problem, a better starting point can be given to `FindMinimum`.

```
In[16]:= FindMinimum[{Sin[x] + .5 x, -10 <= x <= 10}, {{x, -5}}]
Out[16]= {-5.05482, {x → -8.37758}}
```

Alternatively, the user can tighten the constraints.

```
In[17]:= FindMinimum[{Sin[x] + .5 x, -10 <= x <= 10 && x < -5}, {x}]
Out[17]= {-5.05482, {x → -8.37758}}
```

Finally, multiple starting points can be used and the best resulting minimum selected.

```
In[18]:= SeedRandom[7919];
         Table[
          FindMinimum[{Sin[x] + .5 x, -10 <= x <= 10}, {{x, RandomReal[{-10, 10}]}}], {10}]
Out[19]= {{-1.91322, {x → -2.09439}}, {-5.05482, {x → -8.37758}}, {-5.05482, {x → -8.37758}},
          {-5.05482, {x → -8.37758}}, {-1.91322, {x → -2.0944}}, {-5.05482, {x → -8.37758}},
          {1.22837, {x → 4.18879}}, {1.22837, {x → 4.18879}}, {1.22837, {x → 4.18879}}, {4.45598, {x → 10.}}}
```

Multiple starting points can also be done more systematically via `NMinimize`, using the `"RandomSearch"` method with an interior point as the post-processor.

```
In[1]:= NMinimize[{Sin[x] + .5 x, -10 <= x <= 10}, {x},
         Method → {"RandomSearch", "PostProcess" → "InteriorPoint"}]
Out[1]= {-5.05482, {x → -8.37758}}
```

Solving Minimax Problems

The minimax (also known as minmax) problem is that of finding the minimum value of a function defined as the maximum of several functions, that is,

$$\begin{aligned}&\text{Min Max}_i\, f_{i\in\{1,2,\ldots,m\}}(x)\\&s.t\; g(x)\geq 0,\; h(x)=0\end{aligned}$$

While this problem can often be solved using general constrained optimization technique, it is more reliably solved by reformulating the problem into one with smooth objective function. Specifically, the minimax problem can be converted into the following

$$\begin{aligned}&\text{Min } z\\&s.t\; z\geq f_i(x),\; i\in\{1,\,2,\,\ldots,\,m\},\; g(x)\geq 0,\; h(x)=0\end{aligned}$$

and solved using either `FindMinimum` or `NMinimize`.

This defines a function `FindMinMax`.

```
In[9]:= (*FindMinMax[{Max[{f1,f2,..}],constraints},vars]*)
SetAttributes[FindMinMax, HoldAll];
FindMinMax[{f_Max, cons_}, vars_, opts___?OptionQ] :=
  With[{res = iFindMinMax[{f, cons}, vars, opts]}, res /; ListQ[res]];
iFindMinMax[{ff_Max, cons_}, vars_, opts___?OptionQ] :=
  Module[{z, res, f = List @@ ff},
 res = FindMinimum[{z, (And @@ cons) && (And @@ Thread[z >= f])},
     Append[Flatten[{vars}, 1], z], opts];
   If[ListQ[res], {z /. res[[2]], Thread[vars -> (vars /. res[[2]])]}]];
```

This solves an unconstrained minimax problem with one variable.

```
In[19]:= FindMinMax[{Max[{x^2, (x - 1)^2}], {}}, {x, y}]
Out[19]= {0.25, {x → 0.5, y → 1.}}
```

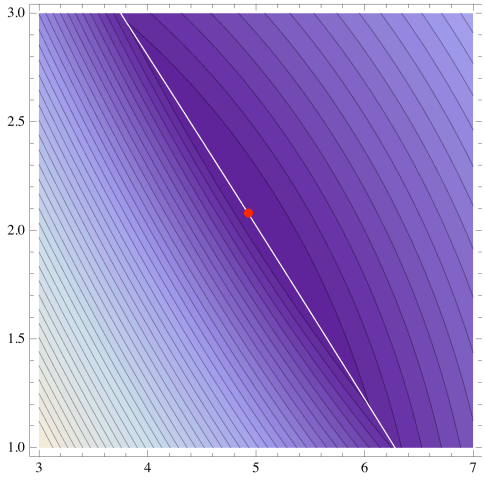
This solves an unconstrained minimax problem with two variables.

```
In[12]:= FindMinMax[{Max[{Abs[2 x^2 + y^2 - 48 x - 40 y + 304], Abs[-x^2 - 3 y^2],
      Abs[x + 3 y - 18], Abs[-x - y], Abs[ x + y - 8]}], {}}, {x, y}]
Out[12]= {37.2356, {x → 4.92563, y → 2.07956}}
```

This shows the contour of the objective function, and the optimal solution.

In[14]:=
```
ContourPlot[Max[{Abs[2 x^2 + y^2 - 48 x - 40 y + 304], Abs[-x^2 - 3 y^2],
   Abs[x + 3 y - 18], Abs[-x - y], Abs[ x + y - 8]}], {x, 3, 7}, {y, 1, 3},
 Contours → 40, Epilog → {Red, PointSize[0.02], Point[{4.93, 2.08}]}]
```

Out[14]=



This solves a constrained minimax problem.

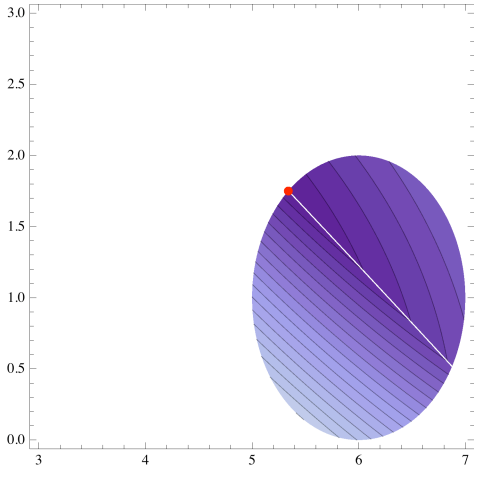
In[16]:=
```
FindMinMax[
 {Max[{Abs[2 x^2 + y^2 - 48 x - 40 y + 304], Abs[-x^2 - 3 y^2], Abs[x + 3 y - 18],
    Abs[-x - y], Abs[ x + y - 8]}], {(x - 6) ^2 + (y - 1) ^2 ≤ 1}}, {x, y}]
```

Out[16]=
```
{37.7192, {x → 5.34014, y → 1.75139}}
```

This shows the contour of the objective function within the feasible region, and the optimal solution.

In[18]:=
```
ContourPlot[Max[{Abs[2 x^2 + y^2 - 48 x - 40 y + 304],
   Abs[-x^2 - 3 y^2], Abs[x + 3 y - 18], Abs[-x - y], Abs[ x + y - 8]}],
 {x, 3, 7}, {y, 0, 3}, RegionFunction → (((#1 - 6) ^2 + (#2 - 1) ^2 <= 1) &),
 Contours → 40, Epilog → {Red, PointSize[0.02], Point[{5.34, 1.75}]}]
```

Out[18]=



Multiobjective Optimization: Goal Programming

Multiobjective programming involves minimizing several objective functions, subject to constraints. Since a solution that minimizes one function often does not minimize the others at the same time, there is usually no unique optimal solution.

Sometimes the decision maker has in mind a goal for each objective. In that case the so-called goal programming technique can be applied.

There are a number of variants of how to model a goal-programming problem. One variant is to order the objective functions based on priority, and seek to minimize the deviation of the most important objective function from its goal first, before attempting to minimize the deviations of the less important objective functions from their goals. This is called lexicographic or preemptive goal programming.

In the second variant, the weighted sum of the deviation is minimized. Specifically, the following constrained minimization problem is to be solved.

$$\begin{aligned}&\mathrm{Min}_x\, w_1\left(f_1(x) - \mathrm{goal}_1\right)^+ + w_2\left(f_2(x) - \mathrm{goal}_2\right)^+ + \ldots + w_m\left(f_m(x) - \mathrm{goal}_m\right)^+ \\ &s.t\ g(x) \geq 0,\ h(x) = 0\end{aligned}$$

Here a^+ stands for the positive part of the real number a. The weights w_i reflect the relative importance, and normalize the deviation to take into account the relative scales and units. Possible values for the weights are the inverse of the goals to be attained. The previous problem can be reformulated to one that is easier to solve.

$$\begin{aligned}&\mathrm{Min}\ w_1\, z_1 + w_2\, z_2 + \ldots + w_m\, z_m \\ &s.t\ z_1 \geq f_1(x) - \mathrm{goal}_1,\ z_2 \geq f_2(x) - \mathrm{goal}_2,\ \ldots,\ z_m \geq f_m(x) - \mathrm{goal}_m,\ z_1, z_2, \ldots, z_m \geq 0 \\ &g(x) \geq 0,\ h(x) = 0\end{aligned}$$

The third variant, Chebyshev goal programming, minimizes the maximum deviation, rather than the sum of the deviations. This balances the deviation of different objective functions. Specifically, the following constrained minimization problem is to be solved.

$$\begin{aligned}&\mathrm{Min\ Max}_i\, w_i\left(f_i(x) - \mathrm{goal}_i\right) \\ &s.t\ g(x) \geq 0,\ h(x) = 0\end{aligned}$$

This can be reformulated as

$$\begin{aligned}&\text{Min } z\\&s.t\ z \geq w_i\left(f_i(x) - \text{goal}_i\right),\ i = 1, 2, \ldots, m\\&g(x) \geq 0,\ h(x) = 0\end{aligned}$$

This defines a function `GoalProgrammingWeightedAverage` that solves the goal programming model by minimizing the weighted sum of the deviation.

```
(* GoalProgrammingWeightedAverage[{{{f1,goal1,weight1},...},cons},vars]]*)
GoalProgrammingWeightedAverage[
   {fg : {{_, _} ..}, cons_}, vars_, opts___?OptionQ] := With[
   {res = Catch[iGoalProgrammingWeightedAverage[{Map[(Append @@ # &),
          Thread[{fg, ConstantArray[1, {Length[fg]}]}]], cons}, vars]]},
   res /; ListQ[res]
  ];
GoalProgrammingWeightedAverage[
   {fg : {{_, _, _} ..}, cons_}, vars_, opts___?OptionQ] := With[
   {res = Catch[iGoalProgrammingWeightedAverage[{fg, cons}, vars]]},
   res /; ListQ[res]
  ];
iGoalProgrammingWeightedAverage[
   {fg : {{_, _, _} ..}, cons_}, vars_, opts___?OptionQ] := Module[
   {fs, goals, zs, z, res, ws},
   {fs, goals, ws} = Transpose[fg];
   If[! VectorQ[ws, (# >= 0 &)], Throw[$Failed]];
   If[! VectorQ[goals, ((NumericQ[#] && Head[#] =!= Complex) &)], Throw[$Failed]];
   zs = Array[z, Length[fs]];
   res = FindMinimum[{ws.zs, (And @@ Flatten[{cons}, 1]) && (And @@ Thread[zs ≥ 0]) &&
        (And @@ Thread[zs ≥ fs - goals ])}, Join[Flatten[{vars}, 1], zs], opts];
   If[ListQ[res], {fs /. res[[2]], Thread[vars → (vars /. res[[2]])]}]
  ];
```

This defines a function `GoalProgrammingChebyshev` that solves the goal programming model by minimizing the maximum deviation.

```
(* syntax GoalAttainment[{{{f1,goal1,weight1},...},cons},vars]]*)
GoalProgrammingChebyshev[
   {fg : {{_, _} ..}, cons_}, vars_, opts___?OptionQ] := With[
   {res = Catch[iGoalProgrammingChebyshev[{Map[(Append @@ # &),
          Thread[{fg, ConstantArray[1, {Length[fg]}]}]], cons}, vars]]},
   res /; ListQ[res]
  ];
GoalProgrammingChebyshev[
   {fg : {{_, _, _} ..}, cons_}, vars_, opts___?OptionQ] := With[
   {res = Catch[iGoalProgrammingChebyshev[{fg, cons}, vars]]},
   res /; ListQ[res]
  ];
iGoalProgrammingChebyshev[
   {fg : {{_, _, _} ..}, cons_}, vars_, opts___?OptionQ] := Module[
   {fs, goals, y, res, ws},
   {fs, goals, ws} = Transpose[fg];
   If[! VectorQ[ws, (# >= 0 &)], Throw[$Failed]];
   If[! VectorQ[goals, ((NumericQ[#] && Head[#] =!= Complex) &)], Throw[$Failed]];
   res = FindMinimum[
     {y, (And @@ Flatten[{cons}, 1]) && (And @@ Thread[y ≥ ws * (fs - goals) ])},
     Append[Flatten[{vars}, 1], y], opts];
   If[ListQ[res], {fs /. res[[2]], Thread[vars → (vars /. res[[2]])]}]
  ];
```

This solves a goal programming problem with two objective functions and one constraint using `GoalProgrammingWeightedAverage` with unit weighting, resulting in deviations from the goal of 13.12 and 33.28, thus a total deviation of 37, and a maximal deviation of 33.28.

```
res1 = GoalProgrammingWeightedAverage[
   {{{x^2 + y^2, 0}, {4 (x - 2)^2 + 4 (y - 2)^2, 0}}, y - x == -4}, {x, y}]
```

```
{{13.12, 33.28}, {x → 3.6, y → -0.4}}
```

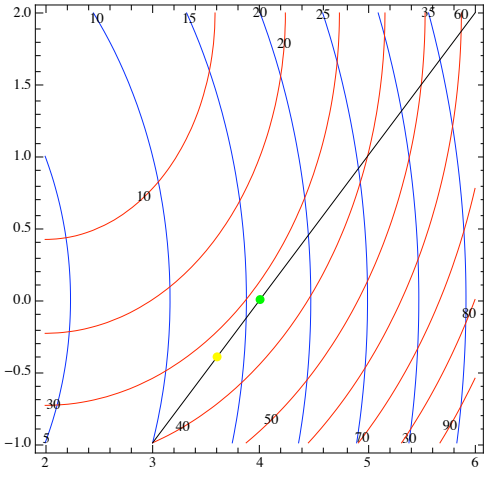
This solves a goal programming problem with two objective functions and one constraint using `GoalProgrammingChebyshev` with unit weighting, resulting in deviations from the goal of 16 and 32, thus a maximal deviation of 32, but a total deviation of 38.

```
res2 = GoalProgrammingChebyshev[
   {{{x^2 + y^2, 0}, {4 (x - 2)^2 + 4 (y - 2)^2, 0}}, y - x == -4}, {x, y}]
```

$$\left\{\{16., 32.\}, \left\{x \to 4., y \to -4.55071 \times 10^{-9}\right\}\right\}$$

This shows the contours for the first (blue) and second (red) objective functions, the feasible region (the black line), and the optimal solution found by `GoalProgrammingWeightedAverage` (yellow point) and by `GoalProgrammingChebyshev` (green point).

```
g1 = ContourPlot[x^2 + y^2, {x, 2, 6}, {y, -1, 2},
    ContourShading → False, ContourStyle → Blue, ContourLabels → Automatic];
g2 = ContourPlot[4 (x - 2)^2 + 4 (y - 2)^2, {x, 2, 6}, {y, -1, 2},
    ContourShading → False, ContourStyle → Red, ContourLabels → Automatic];
Show[{g1, g2}, Epilog → {Line[{{3, -1}, {6, 2}}], PointSize[0.02], Yellow,
    Point[{x, y} /. res1[[2]]], Green, Point[{x, y} /. res2[[2]]]}]
```



An Application Example: Portfolio Optimization

A powerful tool in managing investments is to spread the risk by investing in assets that have few or no correlations. For example, if asset A goes up 20% one year and is down 10% the next, asset B goes down 10% one year and is up 20% the next, and up years for A are down years for B, then holding both in equal amounts would result in a 10% increase every year, without any risk. In reality such assets are rarely available, but the concept remains a useful one.

In this example, the aim is to find the optimal asset allocation so as to minimize the risk, and achieve a preset level of return, by investing in a spread of stocks, bonds, and gold.

Here are the historical returns of various assets between 1973 and 1994. For example, in 1973, S&P 500 lost `1 - 0.852 = 14.8 %`, while gold appreciated by 67.7%.

	"3m Tbill"	"long Tbond"	"SP500"	"Wilt.5000"	"Corp. Bond"	"NASDQ"	"EAFE"	"Gold"
1973	1.075	0.942	0.852	0.815	0.698	1.023	0.851	1.677
1974	1.084	1.02	0.735	0.716	0.662	1.002	0.768	1.722
1975	1.061	1.056	1.371	1.385	1.318	0.123	1.354	0.76
1976	1.052	1.175	1.236	1.266	1.28	1.156	1.025	0.96
1977	1.055	1.002	0.926	0.974	1.093	1.03	1.181	1.2
1978	1.077	0.982	1.064	1.093	1.146	1.012	1.326	1.295
1979	1.109	0.978	1.184	1.256	1.307	1.023	1.048	2.212
1980	1.127	0.947	1.323	1.337	1.367	1.031	1.226	1.296
1981	1.156	1.003	0.949	0.963	0.99	1.073	0.977	0.688
1982	1.117	1.465	1.215	1.187	1.213	1.311	0.981	1.084
1983	1.092	0.985	1.224	1.235	1.217	1.08	1.237	0.872
1984	1.103	1.159	1.061	1.03	0.903	1.15	1.074	0.825
1985	1.08	1.366	1.316	1.326	1.333	1.213	1.562	1.006
1986	1.063	1.309	1.186	1.161	1.086	1.156	1.694	1.216
1987	1.061	0.925	1.052	1.023	0.959	1.023	1.246	1.244
1988	1.071	1.086	1.165	1.179	1.165	1.076	1.283	0.861
1989	1.087	1.212	1.316	1.292	1.204	1.142	1.105	0.977
1990	1.08	1.054	0.968	0.938	0.83	1.083	0.766	0.922
1991	1.057	1.193	1.304	1.342	1.594	1.161	1.121	0.958
1992	1.036	1.079	1.076	1.09	1.174	1.076	0.878	0.926
1993	1.031	1.217	1.1	1.113	1.162	1.11	1.326	1.146
1994	1.045	0.889	1.012	0.999	0.968	0.965	1.078	0.99
average	1.078	1.093	1.120	1.124	1.121	1.046	1.141	1.130

This is the annual return data.

In[2]:=
```
R = {{1.075`, 1.084`, 1.061`, 1.052`, 1.055`, 1.077`,
     1.109`, 1.127`, 1.156`, 1.117`, 1.092`, 1.103`, 1.08`, 1.063`,
     1.061`, 1.071`, 1.087`, 1.08`, 1.057`, 1.036`, 1.031`, 1.045`},
    {0.942`, 1.02`, 1.056`, 1.175`, 1.002`, 0.982`, 0.978`, 0.947`,
     1.003`, 1.465`, 0.985`, 1.159`, 1.366`, 1.309`, 0.925`,
     1.086`, 1.212`, 1.054`, 1.193`, 1.079`, 1.217`, 0.889`},
    {0.852`, 0.735`, 1.371`, 1.236`, 0.926`, 1.064`, 1.184`, 1.323`, 0.949`,
     1.215`, 1.224`, 1.061`, 1.316`, 1.186`, 1.052`, 1.165`, 1.316`,
     0.968`, 1.304`, 1.076`, 1.1`, 1.012`}, {0.815`, 0.716`, 1.385`, 1.266`,
     0.974`, 1.093`, 1.256`, 1.337`, 0.963`, 1.187`, 1.235`, 1.03`, 1.326`,
     1.161`, 1.023`, 1.179`, 1.292`, 0.938`, 1.342`, 1.09`, 1.113`, 0.999`},
    {0.698`, 0.662`, 1.318`, 1.28`, 1.093`, 1.146`, 1.307`, 1.367`, 0.99`,
     1.213`, 1.217`, 0.903`, 1.333`, 1.086`, 0.959`, 1.165`, 1.204`, 0.83`,
     1.594`, 1.174`, 1.162`, 0.968`}, {1.023`, 1.002`, 0.123`, 1.156`,
     1.03`, 1.012`, 1.023`, 1.031`, 1.073`, 1.311`, 1.08`, 1.15`, 1.213`,
     1.156`, 1.023`, 1.076`, 1.142`, 1.083`, 1.161`, 1.076`, 1.11`, 0.965`},
    {0.851`, 0.768`, 1.354`, 1.025`, 1.181`, 1.326`, 1.048`, 1.226`, 0.977`,
     0.981`, 1.237`, 1.074`, 1.562`, 1.694`, 1.246`, 1.283`, 1.105`, 0.766`,
     1.121`, 0.878`, 1.326`, 1.078`}, {1.677`, 1.722`, 0.76`, 0.96`, 1.2`,
     1.295`, 2.212`, 1.296`, 0.688`, 1.084`, 0.872`, 0.825`, 1.006`, 1.216`,
     1.244`, 0.861`, 0.977`, 0.922`, 0.958`, 0.926`, 1.146`, 0.99`}};
```

Here are the expected returns over this 22-year period for the eight assets.

In[3]:=
```
{n, nyear} = Dimensions[R];
```

In[5]:=
```
ER = Mean[Transpose@R]
```

Out[5]=
```
{1.07814, 1.09291, 1.11977, 1.12364, 1.12132, 1.04632, 1.14123, 1.12895}
```

Here is the covariant matrix, which measures how the assets correlate to each other.

In[11]:=
```
Covariants = Covariance[Transpose[R]];
```

This finds the optimal asset allocation by minimizing the standard deviation of an allocation, subject to the constraints that the total allocation is 100% (`Total[vars] == 1`), the expected return is over 12% (*vars*`.ER ≥ 1.12`), and the variables must be non-negative, thus each asset is allocated a non-negative percentage (thus no shorting). The resulting optimal asset allocation suggests 15.5% in 3-month treasury bills, 20.3% in gold, and the rest in stocks, with a resulting standard deviation of 0.0126.

In[18]:=
```
vars = Map[Subscript[x, #] &, {"3m T-bill", "long T-bond", "SP500",
     "Wiltshire 5000", "Corporate Bond", "NASDQ", "EAFE", "Gold"}];
vars = Map[Subscript[x, #] &, {"3m T-bill", "long T-bond", "SP500",
     "Wiltshire 5000", "Corporate Bond", "NASDQ", "EAFE", "Gold"}];
FindMinimum[{
  vars.Covariants.vars,
  Total[vars] == 1 && vars.ER ≥ 1.12 && Apply[And, Thread[Greater[vars, 0]]]}, vars]
```

Out[20]=
```
{0.0126235, {x3m T-bill → 0.154632, xlong T-bond → 0.0195645, xSP500 → 0.354434, xWiltshire 5000 → 0.0238249,
  xCorporate Bond → 0.000133775, xNASDQ → 0.0000309191, xEAFE → 0.24396, xGold → 0.203419}}
```

This trades less return for smaller volatility by asking for an expected return of 10%. Now we have 55.5% in 3-month treasury bills, 10.3% in gold, and the rest in stocks.

```
In[16]:= vars = Map[Subscript[x, #] &, {"3m T-bill", "long T-bond", "SP500",
           "Wiltshire 5000", "Corporate Bond", "NASDQ", "EAFE", "Gold"}];
         FindMinimum[{
           vars.Covariants.vars,
           Total[vars] == 1 && vars.ER ≥ 1.10 && Apply[And, Thread[Greater[vars, 0]]]}, vars]
```

$$\text{Out[17]= } \{0.00365995, \{x_{\text{3m T-bill}} \to 0.555172, x_{\text{long T-bond}} \to 0.0244205, x_{\text{SP500}} \to 0.156701, x_{\text{Wiltshire 5000}} \to 0.0223812, x_{\text{Corporate Bond}} \to 0.00017454, x_{\text{NASDQ}} \to 0.0000293021, x_{\text{EAFE}} \to 0.13859, x_{\text{Gold}} \to 0.102532\}\}$$

Limitations of the Interior Point Method

The implementation of the interior point method in `FindMinimum` requires first and second derivatives of the objective and constraints. Symbolic derivatives are first attempted, and if they fail, finite difference will be used to calculate the derivatives. If the function or constraints are not smooth, particularly if the first derivative at the optimal point is not continuous, the interior point method may experience difficulty in converging.

This shows that the interior point method has difficulty in minimizing this nonsmooth function.

```
In[29]:= FindMinimum[{Abs[x - 3], 0 ≤ x ≤ 5}, {x}]
```

FindMinimum::eit: The algorithm does not converge to the tolerance of 4.806217383937354`*^-6 in 500 iterations. The best estimated solution, with {feasibility residual, KKT residual, complementary residual} of $\{4.54827 \times 10^{-6}, 0.0402467, 2.27414 \times 10^{-6}\}$, is returned. ≫

$$\text{Out[29]= } \{8.71759 \times 10^{-6}, \{x \to 2.99999\}\}$$

This is somewhat similar to the difficulty experienced by an unconstrained Newton's method.

```
In[30]:= FindMinimum[{Abs[x - 3]}, {x}, Method → Newton]
```

FindMinimum::lstol:
The line search decreased the step size to within tolerance specified by AccuracyGoal and PrecisionGoal but was unable to find a sufficient decrease in the function. You may need more than MachinePrecision digits of working precision to meet these tolerances. ≫

$$\text{Out[30]= } \{8.06359 \times 10^{-6}, \{x \to 2.99999\}\}$$

Numerical Algorithms for Constrained Local Optimization

The Interior Point Algorithm

The interior point algorithm solves a constrained optimization by combining constraints and the objective function through the use of the barrier function. Specifically, the general constrained optimization problem is first converted to the standard form

$$\begin{aligned} &\text{Min } f(x) \\ &s.t. h(x) = 0,\ x \geq 0. \end{aligned} \tag{3}$$

The non-negative constraints are then replaced by adding a barrier term to the objective function

$$\begin{aligned} &\text{Min } \psi_\mu(x) := f(x) - \mu \sum\nolimits_i \ln(x_i) \\ &s.t. h(x) = 0, \end{aligned}$$

where $\mu > 0$ is a barrier parameter.

The necessary KKT condition (assuming, for example, that the gradient of h is linearly independent) is

$$\begin{aligned} &\nabla \psi_\mu(x) - y^T A(x) = 0 \\ &h(x) = 0, \end{aligned}$$

where $A(x) = (\nabla h_1(x), \nabla h_2(x), \ldots, \nabla h_m(x))^T$ is of dimension $m \times n$. Or

$$\begin{aligned} &g(x) - \mu X^{-1} e - y^T A(x) = 0 \\ &h(x) = 0. \end{aligned}$$

Here X is a diagonal matrix, with the diagonal element i of x_i if $i \in I$, or 0. Introducing dual variables $z = \mu X^{-1} e$, then

$$\begin{aligned} &g(x) - z - y^T A(x) = 0 \\ &h(x) = 0 \\ &Z X e = \mu e. \end{aligned} \tag{4}$$

This nonlinear system can be solved with Newton's method. Let $L(x, y) = f(x) - h(x)^T y$ and $H(x, y) = \nabla^2 L(x, y) = \nabla^2 f(x) - \sum_{i=1}^{m} y_i \nabla^2 h_i(x)$; the Jacobi matrix of the above system (4) is

$$\begin{pmatrix} H(x, y) & -A(x)^T & -I \\ -A(x) & 0 & 0 \\ Z & 0 & X \end{pmatrix} \begin{pmatrix} \delta x \\ \delta y \\ \delta z \end{pmatrix} = -\begin{pmatrix} g(x) - z - y^T A(x) \\ -h(x) \\ Z X e - \mu e \end{pmatrix} = -\begin{pmatrix} d_\psi \\ -d_h \\ d_{xz} \end{pmatrix}.$$

Eliminating δz, $\delta z = -X^{-1}(Z \delta x + d_{xz})$, then $\left(H(x, y) + X^{-1} Z\right) \delta_x - A(x)^T \delta y = -d_\psi - X^{-1} d_{xz}$, thus

$$\begin{pmatrix} H(x, y) + X^{-1} Z & -A(x)^T \\ -A(x) & 0 \end{pmatrix} \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} = -\begin{pmatrix} d_\psi + X^{-1} d_{xz} \\ -d_h \end{pmatrix} = -\begin{pmatrix} g(x) - A(x)^T y - \mu X^{-1} e \\ -h(x) \end{pmatrix}. \quad (5)$$

Thus the nonlinear constrained problem can be solved iteratively by

$$x := x + \delta x, y := y + \delta y, z := z + \delta z \quad (6)$$

with the search direction $\{\delta x, \delta y, \delta z\}$ given by solving the previous Jacobi system (5).

To ensure convergence, you need to have some measure of success. One way of doing this is to use a merit function, such as the following augmented Langrangian merit function.

Augmented Langrangian Merit Function

This defines an augmented Langrangian merit function

$$\phi(x, \beta) = f(x) - \mu \sum_i \ln(x_i) - h(x)^T \lambda + \beta \| h(x) \|^2 . \quad (7)$$

Here $\mu > 0$ is the barrier parameter and $\beta > 0$ a penalty parameter. It can be proved that if the matrix $N(x, y) = H(x, y) + X^{-1} Z$ is positive definite, then either the search direction given by (6) is a decent direction for the above merit function (7), or $\{x, y, z, \mu\}$ satisfied the KKT condition (4). A line search is performed along the search direction, with the initial step length chosen to be as close to 1 as possible, while maintaining the positive constraints. A backtracking procedure is then used until the Armijo condition is satisfied on the merit function, $\phi(x + t \delta x, \beta) \le \phi(x, \beta) + \gamma t \nabla \phi(x, \beta)^T \delta x$ with $\gamma \in (0, 1/2]$.

Convergence Tolerance

The convergence criterion for the interior point algorithm is

$$|| g(x) - z - y^T A(x) || + || h(x) || + || Z X e - \mu e || \leq \text{tol}$$

with `tol` set, by default, to `10^-MachinePrecision/3`.

Numerical Nonlinear Global Optimization

Introduction

Numerical algorithms for constrained nonlinear optimization can be broadly categorized into *gradient-based methods* and *direct search methods.* Gradient-based methods use first derivatives (gradients) or second derivatives (Hessians). Examples are the sequential quadratic programming (SQP) method, the augmented Lagrangian method, and the (nonlinear) interior point method. Direct search methods do not use derivative information. Examples are Nelder-Mead, genetic algorithm and differential evolution, and simulated annealing. Direct search methods tend to converge more slowly, but can be more tolerant to the presence of noise in the function and constraints.

Typically, algorithms only build up a local model of the problems. Furthermore, many such algorithms insist on certain decrease of the objective function, or decrease of a merit function which is a combination of the objective and constraints, to ensure convergence of the iterative process. Such algorithms will, if convergent, only find local optima, and are called *local optimization algorithms*. In *Mathematica* local optimization problems can be solved using `FindMinimum`.

Global optimization algorithms, on the other hand, attempt to find the global optimum, typically by allowing decrease as well as increase of the objective/merit function. Such algorithms are usually computationally more expensive. Global optimization problems can be solved exactly using `Minimize` or numerically using `NMinimize`.

This solves a nonlinear programming problem,

$$\begin{aligned} &\text{Min} \quad x - y \\ &s.t. \quad -3x^2 + 2xy - y^2 \geq -1 \end{aligned}$$

using Minimize, which gives an exact solution.

```
In[1]:= Minimize[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[1]= {-1, {x → 0, y → 1}}
```

This solves the same problem numerically. NMinimize returns a machine-number solution.

```
In[2]:= NMinimize[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[2]= {-1., {x → 1.90701×10^-6, y → 1.}}
```

FindMinimum numerically finds a local minimum. In this example the local minimum found is also a global minimum.

```
In[3]:= FindMinimum[{x - y, -3 x^2 + 2 x y - y^2 ≥ -1}, {x, y}]
Out[3]= {-1., {x → 2.78301×10^-17, y → 1.}}
```

The NMinimize Function

NMinimize and NMaximize implement several algorithms for finding constrained global optima. The methods are flexible enough to cope with functions that are not differentiable or continuous and are not easily trapped by local optima.

Finding a global optimum can be arbitrarily difficult, even without constraints, and so the methods used may fail. It may frequently be useful to optimize the function several times with different starting conditions and take the best of the results.

This finds the maximum of $\sin(x+y) - x^2 - y^2$.

```
In[46]:= NMaximize[Sin[x + y] - x^2 - y^2, {x, y}]
Out[46]= {0.400489, {x → 0.369543, y → 0.369543}}
```

This finds the minimum of $\left(y - \frac{1}{2}\right)^2 + x^2$ subject to the constraints $y \geq 0$ and $y \geq x + 1$.

```
In[47]:= NMinimize[{x^2 + (y - .5)^2, y ≥ 0 && y ≥ x + 1}, {x, y}]
Out[47]= {0.125, {x → -0.25, y → 0.75}}
```

The constraints to NMinimize and NMaximize may be either a list or a logical combination of equalities, inequalities, and domain specifications. Equalities and inequalities may be nonlinear.

Any strong inequalities will be converted to weak inequalities due to the limits of working with approximate numbers. Specify a domain for a variable using `Element`, for example, `Element[x, Integers]` or $x \in$ `Integers`. Variables must be either integers or real numbers, and will be assumed to be real numbers unless specified otherwise. Constraints are generally enforced by adding penalties when points leave the feasible region.

Constraints can contain logical operators like `And`, `Or`, and so on.

```
In[3]:= NMinimize[{x^2 + y^2, x ≥ 1 || y ≥ 2}, {x, y}]
Out[3]= {1., {x → 1., y → 0.}}
```

Here x is restricted to being an integer.

```
In[4]:= NMinimize[{(x - 1 / 3)^2 + (y - 1 / 3)^2, x ∈ Integers}, {x, y}]
Out[4]= {0.111111, {x → 0, y → 0.333333}}
```

In order for `NMinimize` to work, it needs a rectangular initial region in which to start. This is similar to giving other numerical methods a starting point or starting points. The initial region is specified by giving each variable a finite upper and lower bound. This is done by including $a \le x \le b$ in the constraints, or $\{x, a, b\}$ in the variables. If both are given, the bounds in the variables are used for the initial region, and the constraints are just used as constraints. If no initial region is specified for a variable x, the default initial region of $-1 \le x \le 1$ is used. Different variables can have initial regions defined in different ways.

Here the initial region is taken from the variables. The problem is unconstrained.

```
In[5]:= NMinimize[x^2, {{x, 3, 4}}]
Out[5]= {0., {x → 0.}}
```

Here the initial region is taken from the constraints.

```
In[6]:= NMinimize[{x^2, 3 ≤ x ≤ 4}, {x}]
Out[6]= {9., {x → 3.}}
```

Here the initial region for x is taken from the constraints, the initial region for y is taken from the variables, and the initial region for z is taken to be the default. The problem is unconstrained in y and z, but not x.

```
In[7]:= NMinimize[{x^2 + y^2 + z^2, 3 ≤ x ≤ 4}, {x, {y, 2, 5}, z}]
Out[7]= {9., {x → 3., y → 0., z → 0.}}
```

The polynomial $4x^4 - 4x^2 + 1$ has global minima at $x \to \pm\frac{\sqrt{2}}{2}$. `NelderMead` finds one of the minima.

```
In[48]:= NMinimize[4 x^4 - 4 x^2 + 1, x, Method -> "NelderMead"]
Out[48]= {0., {x → 0.707107}}
```

The other minimum can be found by using a different `RandomSeed`.

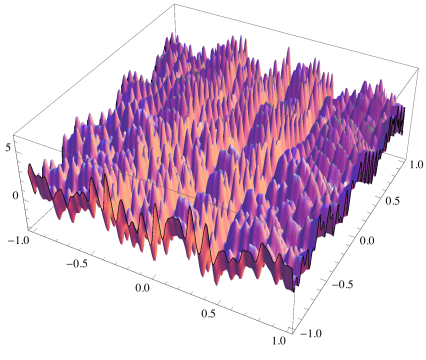
```
In[50]:= NMinimize[4 x^4 - 4 x^2 + 1, x, Method → {"NelderMead", "RandomSeed" → 111}]
Out[50]= {0., {x → -0.707107}}
```

`NMinimize` and `NMaximize` have several optimization methods available: `Automatic`, `"DifferentialEvolution"`, `"NelderMead"`, `"RandomSearch"`, and `"SimulatedAnnealing"`. The optimization method is controlled by the `Method` option, which either takes the method as a string, or takes a list whose first element is the method as a string and whose remaining elements are method-specific options. All method-specific option, left-hand sides should also be given as strings.

The following function has a large number of local minima.

```
In[51]:= Clear[f];
f =
  e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + 1/4 (x^2 + y^2);
Plot3D[f, {x, -1, 1}, {y, -1, 1}, PlotPoints → 50, Mesh → False]
```

```
Out[53]=
```



Use `RandomSearch` to find a minimum.

```
In[54]:= NMinimize[f, {x, y}, Method → "RandomSearch"]
Out[54]= {-2.85149, {x → 0.449094, y → 0.291443}}
```

Use RandomSearch with more starting points to find the global minimum.

```
In[55]:= NMinimize[f, {x, y}, Method → {"RandomSearch", "SearchPoints" → 250}]
Out[55]= {-3.30687, {x → -0.0244031, y → 0.210612}}
```

With the default method, NMinimize picks which method to use based on the type of problem. If the objective function and constraints are linear, LinearProgramming is used. If there are integer variables, or if the head of the objective function is not a numeric function, differential evolution is used. For everything else, it uses Nelder-Mead, but if Nelder-Mead does poorly, it switches to differential evolution.

Because the methods used by NMinimize may not improve every iteration, convergence is only checked after several iterations have occurred.

Numerical Algorithms for Constrained Global Optimization

Nelder-Mead

The Nelder-Mead method is a direct search method. For a function of n variables, the algorithm maintains a set of $n+1$ points forming the vertices of a polytope in n-dimensional space. This method is often termed the "simplex" method, which should not be confused with the well-known simplex method for linear programming.

At each iteration, $n+1$ points $x_1, x_2, \ldots, x_{n+1}$ form a polytope. The points are ordered so that $f(x_1) \le f(x_2) \le \ldots \le f(x_{n+1})$. A new point is then generated to replace the worst point x_{n+1}.

Let c be the centroid of the polytope consisting of the best n points, $c = \frac{1}{n}\sum_{i=1}^{n} x_i$. A trial point x_t is generated by reflecting the worst point through the centroid, $x_t = c + \alpha(c - x_{n+1})$, where $\alpha > 0$ is a parameter.

If the new point x_t is neither a new worst point nor a new best point, $f(x_1) \le f(x_t) \le f(x_n)$, x_t replaces x_{n+1}.

If the new point x_t is better than the best point, $f(x_t) < f(x_1)$, the reflection is very successful and can be carried out further to $x_e = c + \beta(x_t - r)$, where $\beta > 1$ is a parameter to expand the polytope. If the expansion is successful, $f(x_e) < f(x_t)$, x_e replaces x_{n+1}; otherwise the expansion failed, and x_t replaces x_{n+1}.

If the new point x_t is worse than the second worst point, $f(x_t) \geq f(x_n)$, the polytope is assumed to be too large and needs to be contracted. A new trial point is defined as

$$x_c = \begin{cases} c + \gamma(x_{n+1} - c), \text{ if } f(x_t) \geq f(x_{n+1}), \\ c + \gamma(x_t - c), \text{ if } f(x_t) < f(x_{n+1}), \end{cases}$$

where $0 < \gamma < 1$ is a parameter. If $f(x_c) < \text{Min}(f(x_{n+1}), f(x_t))$, the contraction is successful, and x_c replaces x_{n+1}. Otherwise a further contraction is carried out.

The process is assumed to have converged if the difference between the best function values in the new and old polytope, as well as the distance between the new best point and the old best point, are less than the tolerances provided by `AccuracyGoal` and `PrecisionGoal`.

Strictly speaking, Nelder-Mead is not a true global optimization algorithm; however, in practice it tends to work reasonably well for problems that do not have many local minima.

option name	*default value*	
`"ContractRatio"`	`0.5`	ratio used for contraction
`"ExpandRatio"`	`2.0`	ratio used for expansion
`"InitialPoints"`	`Automatic`	set of initial points
`"PenaltyFunction"`	`Automatic`	function applied to constraints to penalize invalid points
`"PostProcess"`	`Automatic`	whether to post-process using local search methods
`"RandomSeed"`	`0`	starting value for the random number generator
`"ReflectRatio"`	`1.0`	ratio used for reflection
`"ShrinkRatio"`	`0.5`	ratio used for shrinking
`"Tolerance"`	`0.001`	tolerance for accepting constraint violations

`NelderMead` specific options.

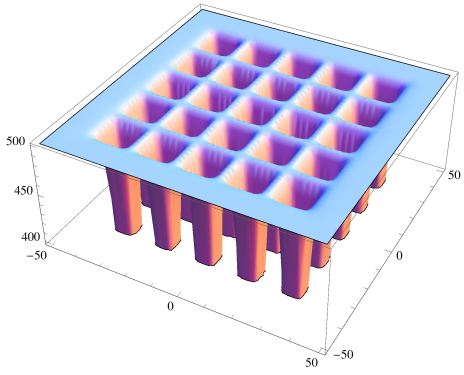
Here the function inside the unit disk is minimized using `NelderMead`.

```
In[82]:= NMinimize[{100 (y - x^2)^2 + (1 - x)^2, x^2 + y^2 ≤ 1}, {x, y}, Method → "NelderMead"]
Out[82]= {0.0456748, {x → 0.786415, y → 0.617698}}
```

Here is a function with several local minima that are all different depths.

```
In[83]:= Clear[a, f];
a = Reverse /@ Distribute[{{-32, -16, 0, 16, 32}, {-32, -16, 0, 16, 32}}, List];
f = 1 / (0.002 + Plus @@ MapIndexed[1 / (#2[[1]] + Plus @@ (({x, y} - #1)^6)) &, a]);
Plot3D[f, {x, -50, 50}, {y, -50, 50}, Mesh → None, PlotPoints → 25]
```

Out[83]=



With the default parameters, `NelderMead` is too easily trapped in a local minimum.

```
In[116]:= Do[Print[NMinimize[f, {{x, -50, 50}, {y, -50, 50}},
    Method → {"NelderMead", "RandomSeed" → i}]], {i, 5}]

{3.96825, {x → 15.9816, y → -31.9608}}

{12.6705, {x → 0.02779, y → 1.57394 × 10^-6}}

{10.7632, {x → -31.9412, y → 0.0253465}}

{1.99203, {x → -15.9864, y → -31.9703}}

{16.4409, {x → -15.9634, y → 15.9634}}
```

By using settings that are more aggressive and less likely to make the simplex smaller, the results are better.

```
In[117]:= Do[Print[NMinimize[f, {{x, -50, 50}, {y, -50, 50}},
    Method → {"NelderMead", "ShrinkRatio" → 0.95, "ContractRatio" → 0.95,
      "ReflectRatio" → 2, "RandomSeed" → i}]], {i, 5}]

{3.96825, {x → 15.9816, y → -31.9608}}

{2.98211, {x → -0.0132362, y → -31.9651}}

{1.99203, {x → -15.9864, y → -31.9703}}

{16.4409, {x → -15.9634, y → 15.9634}}

{0.998004, {x → -31.9783, y → -31.9783}}
```

Differential Evolution

Differential evolution is a simple stochastic function minimizer.

The algorithm maintains a population of m points, $\{x_1, x_2, \ldots, x_j, \ldots, x_m\}$, where typically $m \gg n$, with n being the number of variables.

During each iteration of the algorithm, a new population of m points is generated. The j^{th} new point is generated by picking three random points, x_u, x_v and x_w, from the old population, and forming $x_s = x_w + s(x_u - x_v)$, where s is a real scaling factor. Then a new point x_{new} is constructed from x_j and x_s by taking the i^{th} coordinate from x_s with probability ρ and otherwise taking the coordinate from x_j. If $f(x_{\text{new}}) < f(x_j)$, then x_{new} replaces x_j in the population. The probability ρ is controlled by the `"CrossProbability"` option.

The process is assumed to have converged if the difference between the best function values in the new and old populations, as well as the distance between the new best point and the old best point, are less than the tolerances provided by `AccuracyGoal` and `PrecisionGoal`.

The differential evolution method is computationally expensive, but is relatively robust and tends to work well for problems that have more local minima.

option name	*default value*	
`"CrossProbability"`	`0.5`	probability that a gene is taken from x_i
`"InitialPoints"`	`Automatic`	set of initial points
`"PenaltyFunction"`	`Automatic`	function applied to constraints to penalize invalid points
`"PostProcess"`	`Automatic`	whether to post-process using local search methods
`"RandomSeed"`	`0`	starting value for the random number generator
`"ScalingFactor"`	`0.6`	scale applied to the difference vector in creating a mate
`"SearchPoints"`	`Automatic`	size of the population used for evolution
`"Tolerance"`	`0.001`	tolerance for accepting constraint violations

`DifferentialEvolution` specific options.

Here the function inside the unit disk is minimized using `DifferentialEvolution`.

```
In[125]:= NMinimize[{100 (y - x^2)^2 + (1 - x)^2, x^2 + y^2 ≤ 1},
           {x, y}, Method → "DifferentialEvolution"]
Out[125]= {0.0456748, {x → 0.786415, y → 0.617698}}
```

The following constrained optimization problem has a global minimum of `7.66718`.

```
In[126]:= Clear[f, c, v, x1, x2, y1, y2, y3]

In[127]:= f = 2 x1 + 3 x2 + 3 y1 / 2 + 2 y2 - y3 / 2;
          c = {x1 ^ 2 + y1 == 5 / 4, x2 ^ (3 / 2) + 3 y2 / 2 == 3,
              x1 + y1 ≤ 8 / 5, 4 x2 / 3 + y2 ≤ 3, y3 ≤ y1 + y2, 0 ≤ x1 ≤ 10, 0 ≤ x2 ≤ 10,
              0 ≤ y1 ≤ 1, 0 ≤ y2 ≤ 1, 0 ≤ y3 ≤ 1, {y1, y2, y3} ∈ Integers
             };
          v = {x1, x2, y1, y2, y3};
```

With the default settings for `DifferentialEvolution`, an unsatisfactory solution results.

```
In[130]:= NMinimize[{f, c}, v, Method → "DifferentialEvolution"]
Out[130]= {7.93086, {x1 → 0.499931, x2 → 1.31033, y1 → 1, y2 → 1, y3 → 1}}
```

By adjusting `ScalingFactor`, the results are much better. In this case, the increased `ScalingFactor` gives `DifferentialEvolution` better mobility with respect to the integer variables.

```
In[131]:= NMinimize[{f, c}, v, Method → {"DifferentialEvolution", "ScalingFactor" → 1}]
Out[131]= {7.66718, {x1 → 1.11803, x2 → 1.31037, y1 → 0, y2 → 1, y3 → 1}}
```

Simulated Annealing

Simulated annealing is a simple stochastic function minimizer. It is motivated from the physical process of annealing, where a metal object is heated to a high temperature and allowed to cool slowly. The process allows the atomic structure of the metal to settle to a lower energy state, thus becoming a tougher metal. Using optimization terminology, annealing allows the structure to escape from a local minimum, and to explore and settle on a better, hopefully global, minimum.

At each iteration, a new point, x_{new}, is generated in the neighborhood of the current point, x. The radius of the neighborhood decreases with each iteration. The best point found so far, x_{best}, is also tracked.

If $f(x_{\text{new}}) \le f(x_{\text{best}})$, x_{new} replaces x_{best} and x. Otherwise, x_{new} replaces x with a probability $e^{b(i,\Delta f,f_0)}$. Here b is the function defined by `BoltzmannExponent`, i is the current iteration, Δf is the change in the objective function value, and f_0 is the value of the objective function from the previous iteration. The default function for b is $\frac{-\Delta f \log(i+1)}{10}$.

Like the `RandomSearch` method, `SimulatedAnnealing` uses multiple starting points, and finds an optimum starting from each of them.

The default number of starting points, given by the option `SearchPoints`, is $\min(2\,d, 50)$, where d is the number of variables.

For each starting point, this is repeated until the maximum number of iterations is reached, the method converges to a point, or the method stays at the same point consecutively for the number of iterations given by `LevelIterations`.

option name	*default value*	
`"BoltzmannExponent"`	`Automatic`	exponent of the probability function
`"InitialPoints"`	`Automatic`	set of initial points
`"LevelIterations"`	`50`	maximum number of iterations to stay at a given point
`"PenaltyFunction"`	`Automatic`	function applied to constraints to penalize invalid points
`"PerturbationScale"`	`1.0`	scale for the random jump
`"PostProcess"`	`Automatic`	whether to post-process using local search methods
`"RandomSeed"`	`0`	starting value for the random number generator
`"SearchPoints"`	`Automatic`	number of initial points
`"Tolerance"`	`0.001`	tolerance for accepting constraint violations

`SimulatedAnnealing` specific options.

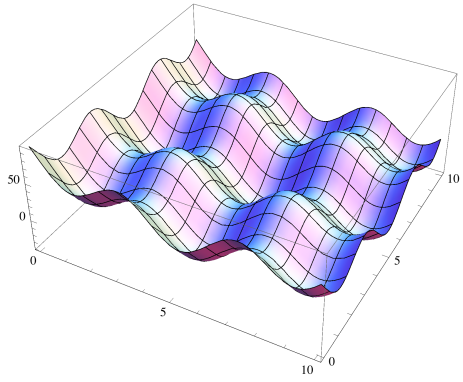
Here a function in two variables is minimized using `SimulatedAnnealing`.

```
In[62]:= NMinimize[{100 (y - x^2)^2 + (1 - x)^2, -2.084 ≤ x ≤ 2.084 && -2.084 ≤ y ≤ 2.084},
          {x, y}, Method → "SimulatedAnnealing"]
Out[62]= {0., {x → 1., y → 1.}}
```

Here is a function with many local minima.

```
In[63]:= Clear[f]
         f[x_, y_] := 20 Sin[π / 2 (x - 2 π)] + 20 Sin[π / 2 (y - 2 π)] + (x - 2 π)^2 + (y - 2 π)^2;
         Plot3D[f[x, y], {x, 0, 10}, {y, 0, 10}]
```

Out[65]=



By default, the step size for `SimulatedAnnealing` is not large enough to escape from the local minima.

```
In[68]:= NMinimize[f[x, y], {x, y}, Method → "SimulatedAnnealing"]
Out[68]= {8.0375, {x → 1.48098, y → 1.48098}}
```

By increasing `PerturbationScale`, larger step sizes are taken to produce a much better solution.

```
In[69]:= NMinimize[f[x, y], {x, y}, Method → {"SimulatedAnnealing", "PerturbationScale" → 3}]
Out[69]= {-38.0779, {x → 5.32216, y → 5.32216}}
```

Here `BoltzmannExponent` is set to use an exponential cooling function that gives faster convergence. (Note that the modified `PerturbationScale` is still being used as well.)

```
In[70]:= NMinimize[f[x, y], {x, y}, Method → {"SimulatedAnnealing", "PerturbationScale" → 3,
             "BoltzmannExponent" → Function[{i, df, f0}, -df / (Exp[i / 10])]}]
Out[70]= {-38.0779, {x → 5.32216, y → 5.32216}}
```

Random Search

The random search algorithm works by generating a population of random starting points and uses a local optimization method from each of the starting points to converge to a local minimum. The best local minimum is chosen to be the solution.

The possible local search methods are `Automatic` and `"InteriorPoint"`. The default method is `Automatic`, which uses `FindMinimum` with unconstrained methods applied to a system with penalty terms added for the constraints. When `Method` is set to `"InteriorPoint"`, a nonlinear interior-point method is used.

The default number of starting points, given by the option `SearchPoints`, is $\min(10\,d, 100)$, where d is the number of variables.

Convergence for `RandomSearch` is determined by convergence of the local method for each starting point.

`RandomSearch` is fast, but does not scale very well with the dimension of the search space. It also suffers from many of the same limitations as `FindMinimum`. It is not well suited for discrete problems and others where derivatives or secants give little useful information about the problem.

option name	*default value*	
`"InitialPoints"`	`Automatic`	set of initial points
`"Method"`	`Automatic`	which method to use for minimization
`"PenaltyFunction"`	`Automatic`	function applied to constraints to penalize invalid points
`"PostProcess"`	`Automatic`	whether to post-process using local search methods
`"RandomSeed"`	`0`	starting value for the random number generator
`"SearchPoints"`	`Automatic`	number of points to use for starting local searches
`"Tolerance"`	`0.001`	tolerance for accepting constraint violations

`RandomSearch` specific options.

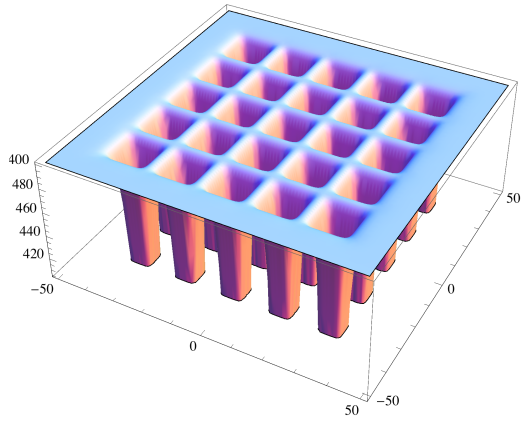
Here the function inside the unit disk is minimized using `RandomSearch`.

```
In[71]:= NMinimize[{100 (y - x^2)^2 + (1 - x)^2, x^2 + y^2 ≤ 1}, {x, y}, Method → "RandomSearch"]
Out[71]= {0.0456748, {x → 0.786415, y → 0.617698}}
```

Here is a function with several local minima that are all different depths and are generally difficult to optimize.

In[72]:=
```
Clear[a, f];
a = Reverse /@ Distribute[{{-32, -16, 0, 16, 32}, {-32, -16, 0, 16, 32}}, List];
f = 1 / (0.002 + Plus @@ MapIndexed[1 / (#2[[1]] + Plus @@ (({x, y} - #1)^6)) &, a]);
Plot3D[f, {x, -50, 50}, {y, -50, 50}, Mesh → None,
 NormalsFunction → "Weighted", PlotPoints → 50]
```

Out[72]=



With the default number of `SearchPoints`, sometimes the minimum is not found.

In[73]:=
```
Do[Print[NMinimize[f, {{x, -50, 50}, {y, -50, 50}},
    Method → {"RandomSearch", "RandomSeed" → i}]], {i, 5}]
```

```
{1.99203, {x → -15.9864, y → -31.9703}}

{1.99203, {x → -15.9864, y → -31.9703}}

{0.998004, {x → -31.9783, y → -31.9783}}

{1.99203, {x → -15.9864, y → -31.9703}}

{0.998004, {x → -31.9783, y → -31.9783}}
```

Using many more `SearchPoints` produces better answers.

In[74]:=
```
Do[Print[NMinimize[f, {{x, -50, 50}, {y, -50, 50}},
    Method → {"RandomSearch", "SearchPoints" → 100, "RandomSeed" → i}]], {i, 5}]
```

```
{0.998004, {x → -31.9783, y → -31.9783}}

{0.998004, {x → -31.9783, y → -31.9783}}

{0.998004, {x → -31.9783, y → -31.9783}}

{0.998004, {x → -31.9783, y → -31.9783}}

{0.998004, {x → -31.9783, y → -31.9783}}
```

Here points are generated on a grid for use as initial points.

```
In[75]:= NMinimize[f, {{x, -50, 50}, {y, -50, 50}}, Method → {"RandomSearch",
           "InitialPoints" → Flatten[Table[{i, j}, {i, -45, 45, 5}, {j, -45, 45, 5}], 1]}]
Out[75]= {0.998004, {x → -31.9783, y → -31.9783}}
```

This uses nonlinear interior point methods to find the minimum of a sum of squares.

```
In[76]:= n = 10;
         f = Sum[(x[i] - Sin[i])^2, {i, 1, n}];
         c = Table[-0.5 < x[i] < 0.5, {i, n}];
         v = Array[x, n];
         Timing[NMinimize[{f, c}, v, Method → {"RandomSearch", Method → "InteriorPoint"}]]
Out[80]= {8.25876, {0.82674, {x[1] → 0.5, x[2] → 0.5, x[3] → 0.14112, x[4] → -0.5,
           x[5] → -0.5, x[6] → -0.279415, x[7] → 0.5, x[8] → 0.5, x[9] → 0.412118, x[10] → -0.5}}}
```

For some classes of problems, limiting the number of `SearchPoints` can be much faster without affecting the quality of the solution.

```
In[81]:= Timing[NMinimize[{f, c}, v,
           Method → {"RandomSearch", Method → "InteriorPoint", "SearchPoints" → 1}]]
Out[81]= {0.320425, {0.82674, {x[1] → 0.5, x[2] → 0.5, x[3] → 0.14112, x[4] → -0.5,
           x[5] → -0.5, x[6] → -0.279415, x[7] → 0.5, x[8] → 0.5, x[9] → 0.412118, x[10] → -0.5}}}
```

Exact Global Optimization

Introduction

Exact global optimization problems can be solved exactly using `Minimize` and `Maximize`.

This computes the radius of the circle, centered at the origin, circumscribed about the set $x^4 + 3\,y^4 \le 7$.

```
In[1]:= Maximize[{Sqrt[x^2 + y^2], x^4 + 3 y^4 ≤ 7}, {x, y}]
Out[1]= {Sqrt[2] (7/3)^(1/4), {x → Root[-21 + 4 #1^4 &, 1], y → Root[-7 + 12 #1^4 &, 1]}}
```

This computes the radius of the circle, centered at the origin, circumscribed about the set $a\,x^2 + b\,y^2 \le 1$ as a function of the parameters a and b.

```
In[2]:= Maximize[{Sqrt[x^2 + y^2], a x^2 + b y^2 <= 1}, {x, y}]
```

```
Out[2]= {{Sqrt[1/a]      (b > 0 && a == b) || (b > 0 && 0 < a < b)
          Sqrt[1/(a-b)]  b > 0 && a == 2 b
          Sqrt[1/b]      (b > 0 && a > 2 b) || (b > 0 && b < a < 2 b)
          ∞              True                                                  ,
  {x -> {0               (b > 0 && a == 2 b) || (b > 0 && a > 2 b) || (b > 0 && b < a < 2 b)
         -Sqrt[1/a]      b > 0 && 0 < a < b
         -Sqrt[1/a]/2    b > 0 && a == b                                       ,
         Indeterminate   True
  y -> {0               b > 0 && 0 < a < b
        -Sqrt[3]/2 Sqrt[1/a]  b > 0 && a == b
        -Sqrt[1/(a-b)]  b > 0 && a == 2 b                                     }}
        -Sqrt[1/b]      (b > 0 && a > 2 b) || (b > 0 && b < a < 2 b)
        Indeterminate   True
```

Algorithms

Depending on the type of problem, several different algorithms can be used.

The most general method is based on the cylindrical algebraic decomposition (CAD) algorithm. It applies when the objective function and the constraints are real algebraic functions. The method can always compute global extrema (or extremal values, if the extrema are not attained). If parameters are present, the extrema can be computed as piecewise-algebraic functions of the parameters. A downside of the method is its high, doubly exponential complexity in the number of variables. In certain special cases not involving parameters, faster methods can be used.

When the objective function and all constraints are linear with rational number coefficients, global extrema can be computed exactly using the simplex algorithm.

For univariate problems, equation and inequality solving methods are used to find the constraint solution set and discontinuity points and zeros of the derivative of the objective function within the set.

Another approach to finding global extrema is to find all the local extrema, using the Lagrange multipliers or the Karush-Kuhn-Tucker (KKT) conditions, and pick the smallest (or the greatest). However, for a fully automatic method, there are additional complications. In addition to solving the necessary conditions for local extrema, it needs to check smoothness of the objective function and smoothness and nondegeneracy of the constraints. It also needs to check for extrema at the boundary of the set defined by the constraints and at infinity, if the set is unbounded. This in general requires exact solving of systems of equations and inequalities over the reals, which may lead to CAD computations that are harder than in the optimization by CAD algorithm. Currently *Mathematica* uses Lagrange multipliers only for equational constraints within a bounded box. The method also requires that the number of stationary points and the number of singular points of the constraints be finite. An advantage of this method over the CAD-based algorithm is that it can solve some transcendental problems, as long as they lead to systems of equations that *Mathematica* can solve.

Optimization by Cylindrical Algebraic Decomposition

Examples

This finds the point on the cubic curve $x^3 - x + y^2 = \frac{1}{4}$ which is closest to the origin.

```
In[3]:= Minimize[{x^2 + y^2, x^3 + y^2 - x == 1/4}, {x, y}]
Out[3]= {Root[-1 + 16 #1 - 32 #1^2 + 16 #1^3 &, 1], {x → Root[-1 - 4 #1 + 4 #1^3 &, 2], y → 0}}
```

This finds the point on the cubic curve $x^3 - x + y^2 = a$ which is closest to the origin, as a function of the parameter a.

```
In[4]:= min = Minimize[{x^2 + y^2, x^3 + y^2 - x == a}, {x, y}]
Out[4]= {{1/9                                  a == 8/27
         {1/27 (-5 + 27 a)                     8/27 < a ≤ 80/27,
         Root[-a^2 + #1 - 2 #1^2 + #1^3 &, 1]  True
```

$$\left\{x \to \begin{cases} -\frac{1}{3} & a == \frac{8}{27} \;||\; \frac{8}{27} < a \leq \frac{80}{27} \\ \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 1\right] & a > \frac{80}{27} \;||\; a < -\frac{2}{3\sqrt{3}} \\ \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 2\right] & \text{True} \end{cases}, y \to \right.$$

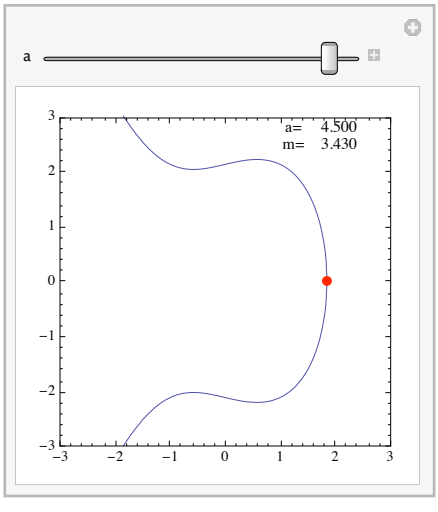
$$\left\{ \begin{cases} 0 & a == \frac{8}{27} \\ -\sqrt{-\frac{8}{27} + a} & \frac{8}{27} < a \leq \frac{80}{27} \\ -\sqrt{\left(a + \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 1\right] - \right.} & a > \frac{80}{27} \;||\; a < -\cdot \\ \quad \left. \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 1\right]^3\right) & \quad 3 \\ -\sqrt{\left(a + \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 2\right] - \right.} & \text{True} \\ \quad \left. \text{Root}\left[-a + \text{Root}\left[-a^2 + \#1 - 2\,\#1^2 + \#1^3 \,\&,\, 1\right] - \#1 - \#1^2 + \#1^3 \,\&,\, 2\right]^3\right) & \end{cases} \right\}$$

This visualization shows the point on the cubic curve $x^3 - x + y^2 = a$ which is closest to the origin, and the distance m of the point from the origin. The value of parameter a can be modified using the slider. The visualization uses the minimum computed by `Minimize`.

In[5]:=

```
plot[a_] := ContourPlot[x^3 + y^2 - x == a,
    {x, -3, 3}, {y, -3, 3}, PlotRange -> {{-3, 3}, {-3, 3}}];
minval[a_] := Evaluate[min[[1]]]
minpt[a_] := Evaluate[min[[2]]]
mmark = Graphics[Text[Style["m=", 10], {1.25, 2.5}]];
mvalue[a_] :=
  Graphics[Text[Style[PaddedForm[minval[a], {5, 3}], 10], {2, 2.5}]];
amark = Graphics[Text[Style["a=", 10], {1.25, 2.8}]];
avalue[a_] := Graphics[Text[Style[PaddedForm[a, {5, 3}], 10], {2, 2.8}]];
mpoint[a_] := Graphics[{PointSize[0.03], Red, Point[Re[{x, y} /. minpt[a]]]}];
Manipulate[Show[{plot[a], amark, avalue[a], mmark, mvalue[a], mpoint[a]}],
 {{a, 4.5}, -5, 5}, SaveDefinitions → True]
```

Out[13]=



Customized CAD Algorithm for Optimization

The cylindrical algebraic decomposition (CAD) algorithm is available in *Mathematica* directly as `CylindricalDecomposition`. The algorithm is described in more detail in "Real Polynomial Systems". The following describes how to customize the CAD algorithm to solve the global optimization problem.

Reduction to Minimizing a Coordinate Function

Suppose it is required to minimize an algebraic function $f(x, t)$ over the solution sets of algebraic constraints $\Phi(x, t)$, where x is a vector of variables and t is a vector of parameters. Let y be a new variable. The minimization of f over the constraints Φ is equivalent to the minimization of the coordinate function y over the semialgebraic set given by $y = f(x, t) \bigwedge \Phi(x, t)$.

If f happens to be a monotonic function of one variable x_1, a new variable is not needed, as x_1 can be minimized instead.

The Projection Phase of CAD

The variables are projected, with x first, then the new variable y, and then the parameters t.

If algebraic functions are present, they are replaced with new variables; equations and inequalities satisfied by the new variables are added. The variables replacing algebraic functions are projected first. They also require special handling in the lifting phase of the algorithm.

Projection operator improvements by Hong, McCallum, and Brown can be used here, including the use of equational constraints. Note that if a new variable needs to be introduced, there is at least one equational constraint, namely $y == f$.

The Lifting Phase of CAD

The parameters t are lifted first, constructing the algebraic function equation and inequality description of the cells. If there are constraints that depend only on parameters and you can determine that Φ is identically false over a parameter cell, you do not need to lift this cell further.

When lifting the minimization variable y, you start with the smallest values of y, and proceed (lifting the remaining variables in the depth-first manner) until you find the first cell for which the constraints are satisfied. If this cell corresponds to a root of a projection polynomial in y, the root is the minimum value of f, and the coordinates corresponding to x of any point in the cell give a point at which the minimum is attained. If parameters are present, you get a parametric description of a point in the cell in terms of roots of polynomials bounding the cell. If there are no parameters, you can simply give the sample point used by the CAD algorithm. If the first cell satisfying the constraints corresponds to an interval (r, s), where r is a root of a projection polynomial in y, then r is the infimum of values of f, and the infimum value is not attained. Finally, if the first cell satisfying the constraints corresponds to an interval $(-\infty, s)$, f is unbounded from below.

Strict Inequality Constraints

If there are no parameters, all constraints are strict inequalities, and you only need the extremum value, then a significantly simpler version of the algorithm can be used. (You can safely make inequality constraints strict if you know that $C \subseteq \overline{\text{int}(C)}$, where C is the solution set of the constraints.) In this case many lower-dimensional cells can be disregarded; hence, the projection may only consist of the leading coefficients, the resultants, and the discriminants. In the lifting phase, only full-dimensional cells need be constructed; hence, there is no need for algebraic number computations.

`Experimental`Infimum[{`*f*`,`*cons*`},{`*x*`,`*y*`,...}]`

find the infimum of values of f on the set of points satisfying the constraints *cons*.

`Experimental`Supremum[{`*f*`,`*cons*`},{`*x*`,`*y*`,...}]`

find the supremum of values of f on the set of points satisfying the constraints *cons*.

Finding extremum values.

This finds the smallest ball centered at the origin which contains the set bounded by the surface $x^4 - y\,z\,x + 2\,y^4 + 3\,z^4 = 1$. A full `Maximize` call with the same input did not finish in 10 minutes.

```
In[14]:= Experimental`Supremum[{x^2 + y^2 + z^2, x^4 + 2 y^4 + 3 z^4 - x y z < 1}, {x, y, z}] // Timing
Out[14]= {4.813, -Root[-1 341 154 819 099 - 114 665 074 208 #1 + 4 968 163 024 164 #1^2 +
      288 926 451 967 #1^3 - 7 172 215 018 940 #1^4 - 240 349 978 752 #1^5 + 5 066 800 071 680 #1^6 +
      69 844 008 960 #1^7 - 1 756 156 133 376 #1^8 - 2 717 908 992 #1^9 + 239 175 991 296 #1^10 &, 1]}
```

Linear Optimization

When the objective function and all constraints are linear, global extrema can be computed exactly using the simplex algorithm.

This solves a random linear minimization problem with ten variables.

In[15]:=
```
SeedRandom[1]; n = 10;
A = Table[RandomInteger[{-1000, 1000}], {n / 2}, {n}];
B = Table[RandomInteger[{-1000, 1000}], {n / 2}, {n}];
α = Table[RandomInteger[{-1000, 1000}], {n / 2}];
β = Table[RandomInteger[{-1000, 1000}], {n / 2}];
γ = Table[RandomInteger[{-1000, 1000}], {n}];
X = x /@ Range[n];
Minimize[{γ.X, And @@ Thread[A.X == α] && And @@ Thread[β ≤ B.X ≤ β + 100]}, X]
```

Out[22]=
$$\left\{\frac{6\,053\,416\,204\,117\,714\,679\,590\,329\,859\,484\,149}{1\,194\,791\,208\,768\,786\,909\,167\,074\,679\,920},\right.$$
$$\left\{x[1] \to \frac{1\,231\,164\,669\,474\,551\,725\,622\,041\,404\,999}{1\,194\,791\,208\,768\,786\,909\,167\,074\,679\,920}, x[2] \to -\frac{1\,324\,409\,686\,130\,055\,761\,704\,674\,699\,781}{597\,395\,604\,384\,393\,454\,583\,537\,339\,960},\right.$$
$$x[3] \to \frac{33\,103\,498\,981\,835\,356\,980\,792\,655\,092}{74\,674\,450\,548\,049\,181\,822\,942\,167\,495}, x[4] \to -\frac{859\,057\,104\,531\,672\,755\,759\,277\,109\,213}{597\,395\,604\,384\,393\,454\,583\,537\,339\,960},$$
$$x[5] \to -\frac{1\,101\,359\,025\,510\,393\,235\,751\,743\,044\,237}{1\,194\,791\,208\,768\,786\,909\,167\,074\,679\,920}, x[6] \to \frac{681\,114\,758\,987\,787\,242\,569\,015\,281\,099}{597\,395\,604\,384\,393\,454\,583\,537\,339\,960},$$
$$x[7] \to \frac{3\,008\,784\,898\,283\,435\,639\,647\,867\,743}{14\,934\,890\,109\,609\,836\,364\,588\,433\,499}, x[8] \to \frac{656\,889\,989\,559\,037\,679\,422\,691\,779\,229}{597\,395\,604\,384\,393\,454\,583\,537\,339\,960},$$
$$\left.x[9] \to \frac{1\,769\,243\,029\,064\,640\,615\,513\,519\,505\,823}{597\,395\,604\,384\,393\,454\,583\,537\,339\,960}, x[10] \to \frac{699\,425\,860\,731\,183\,550\,585\,590\,812\,579}{1\,194\,791\,208\,768\,786\,909\,167\,074\,679\,920}\right\}\Bigg\}$$

Optimization problems where the objective is a fraction of linear functions and the constraints are linear (linear fractional programs) reduce to linear optimization problems. This solves a random linear fractional minimization problem with ten variables.

In[23]:=
```
SeedRandom[2]; n = 10;
A = Table[RandomInteger[{-1000, 1000}], {n / 2}, {n}];
B = Table[RandomInteger[{-1000, 1000}], {n / 2}, {n}];
α = Table[RandomInteger[{-1000, 1000}], {n / 2}];
β = Table[RandomInteger[{-1000, 1000}], {n / 2}];
γ = Table[RandomInteger[{-1000, 1000}], {n}];
δ = Table[RandomInteger[{-1000, 1000}], {n}];
X = x /@ Range[n];
Minimize[{γ.X / δ.X, And @@ Thread[A.X == α] && And @@ Thread[β ≤ B.X ≤ β + 100]}, X]
```

Out[31]=
$$\left\{-\frac{1\,286\,274\,653\,702\,415\,809\,313\,525\,025\,452\,519}{437\,743\,412\,320\,661\,916\,541\,674\,600\,912\,158},\right.$$
$$\left\{x[1] \to \frac{611\,767\,491\,996\,227\,433\,062\,183\,883\,923}{599\,276\,957\,533\,098\,032\,663\,796\,688\,622}, x[2] \to \frac{2\,665\,078\,586\,976\,600\,235\,350\,409\,286\,849}{1\,198\,553\,915\,066\,196\,065\,327\,593\,377\,244},\right.$$
$$\left.x[3] \to -\frac{215\,391\,679\,158\,483\,849\,611\,061\,030\,533}{299\,638\,478\,766\,549\,016\,331\,898\,344\,311}, x[4] \to \frac{1\,477\,394\,491\,589\,036\,027\,204\,142\,993\,013}{599\,276\,957\,533\,098\,032\,663\,796\,688\,622},\right.$$

$$x[5] \to \frac{473\,657\,331\,854\,113\,835\,444\,689\,628\,600}{299\,638\,478\,766\,549\,016\,331\,898\,344\,311},\ x[6] \to -\frac{955\,420\,726\,065\,204\,315\,229\,251\,112\,109}{599\,276\,957\,533\,098\,032\,663\,796\,688\,622},$$
$$x[7] \to \frac{265\,603\,080\,958\,760\,324\,085\,018\,021\,123}{1\,198\,553\,915\,066\,196\,065\,327\,593\,377\,244},\ x[8] \to -\frac{447\,840\,634\,450\,080\,124\,431\,365\,644\,067}{599\,276\,957\,533\,098\,032\,663\,796\,688\,622},$$
$$x[9] \to -\frac{2\,155\,930\,215\,697\,442\,604\,517\,040\,669\,063}{1\,198\,553\,915\,066\,196\,065\,327\,593\,377\,244},\ x[10] \to \frac{18\,201\,652\,848\,869\,287\,002\,844\,477\,177}{299\,638\,478\,766\,549\,016\,331\,898\,344\,311}\Big\}\Big\}$$

Univariate Optimization

For univariate problems, equation and inequality solving methods are used to find the constraint solution set and discontinuity points and zeros of the derivative of the objective function within the set.

This solves a univariate optimization problem with a transcendental objective function.

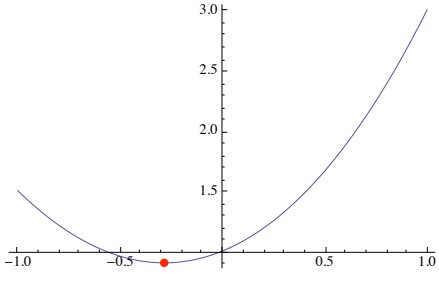
In[32]:=
```
m = Minimize[x^2 + 2^x, x]
```

Out[32]=
$$\left\{2^{-\frac{\text{ProductLog}\left[\frac{\text{Log}[2]^2}{2}\right]}{\text{Log}[2]}} + \frac{\text{ProductLog}\left[\frac{\text{Log}[2]^2}{2}\right]^2}{\text{Log}[2]^2},\ \left\{x \to -\frac{\text{ProductLog}\left[\frac{\text{Log}[2]^2}{2}\right]}{\text{Log}[2]}\right\}\right\}$$

Here is a visualization of the minimum found.

In[33]:=
```
Show[{Plot[x^2 + 2^x, {x, -1, 1}],
  Graphics[{PointSize[0.02], Red, Point[N[{x /. m[[2]], m[[1]]}]]}]}]
```

Out[33]=



Here *Mathematica* recognizes that the objective functions and the constraints are periodic.

In[34]:=
```
Minimize[{Tan[2 x - π/2]^2, -1/2 ≤ Sin[x] ≤ 1/2}, x]
```

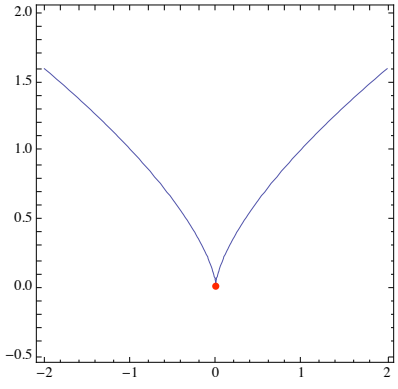
Out[34]=
$$\left\{\frac{1}{3},\ \left\{x \to \frac{\pi}{6}\right\}\right\}$$

Optimization by Finding Stationary and Singular Points

Here is an example where the minimum is attained at a singular point of the constraints.

In[35]:=
```
m = Minimize[{y, y^3 == x^2 && -2 ≤ x ≤ 2 && -2 ≤ y ≤ 2}, {x, y}]
```
Out[35]=
```
{0, {x → 0, y → 0}}
```

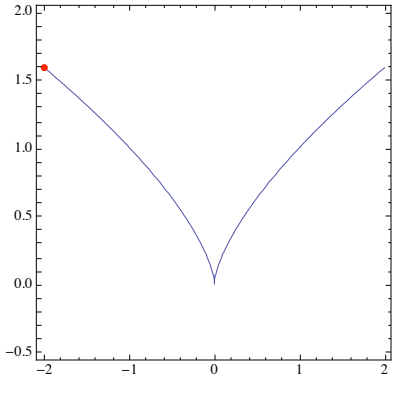
In[36]:=
```
Show[{ContourPlot[y^3 == x^2, {x, -2, 2}, {y, -0.5, 2}],
  Graphics[{PointSize[0.02], Red, Point[{x, y} /. m[[2]]]}]}]
```
Out[36]=



The maximum of the same objective function is attained on the boundary of the set defined by the constraints.

In[37]:=
```
m = Maximize[{y, y^3 == x^2 && -2 ≤ x ≤ 2 && -2 ≤ y ≤ 2}, {x, y}]
```
Out[37]=
```
{Root[-4 + #1^3 &, 1], {x → -2, y → Root[-4 + #1^3 &, 1]}}
```

In[38]:=
```
Show[{ContourPlot[y^3 == x^2, {x, -2, 2}, {y, -0.5, 2}],
  Graphics[{PointSize[0.02], Red, Point[{x, y} /. m[[2]]]}]}]
```
Out[38]=



There are no stationary points in this example.

```
In[39]:= Reduce[y^3 == x^2 && -2 x λ == 0 && 1 + 3 y^2 λ == 0, {x, y, λ}]
Out[39]= False
```

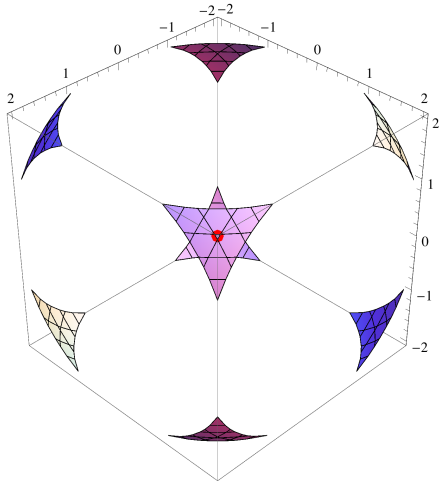
Here is a set of 3-dimensional examples with the same constraints. Depending on the objective function, the maximum is attained at a stationary point of the objective function on the solution set of the constraints, at a stationary point of the restriction of the objective function to the boundary of the solution set of the constraints, and at the boundary of the boundary of the solution set of the constraints.

Here the maximum is attained at a stationary point of the objective function on the solution set of the constraints.

```
In[40]:= m = Maximize[x + y + z, x^2 + y^2 + z^2 == 9 && -2 ≤ x ≤ 2 && -2 ≤ y ≤ 2 && -2 ≤ z ≤ 2, {x, y, z}]
Out[40]= {3 Sqrt[3], {x → Sqrt[3], y → Sqrt[3], z → Sqrt[3]}}
```

```
In[41]:= Show[{ContourPlot3D[x^2 + y^2 + z^2 - 9 == 0, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}],
   Graphics3D[{PointSize[0.03], Red, Point[{x, y, z} /. m[[2]]]}],
  ViewPoint → {3, 3, 3}]
```

Out[41]=

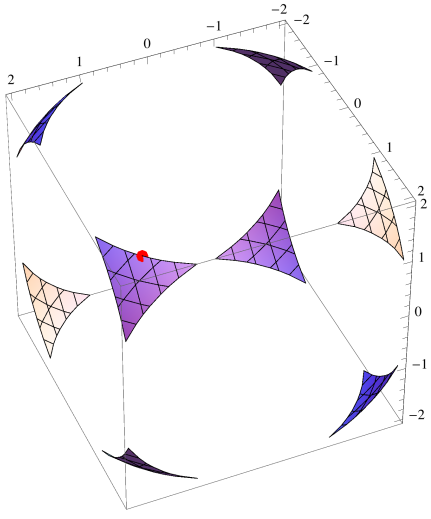


Here the maximum is attained at a stationary point of the restriction of the objective function to the boundary of the solution set of the constraints.

```
In[42]:= m = Maximize[x + y + 2 z, x^2 + y^2 + z^2 == 9 && -2 ≤ x ≤ 2 && -2 ≤ y ≤ 2 && -2 ≤ z ≤ 2, {x, y, z}]
Out[42]= {4 + Sqrt[10], {x → Sqrt[5/2], y → Sqrt[5/2], z → 2}}
```

In[43]:=
```
Show[{ContourPlot3D[x^2 + y^2 + z^2 - 9 == 0, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}],
  Graphics3D[{PointSize[0.03], Red, Point[{x, y, z} /. m[[2]]]}],
 ViewPoint → {3, 7, 7}]
```

Out[43]=



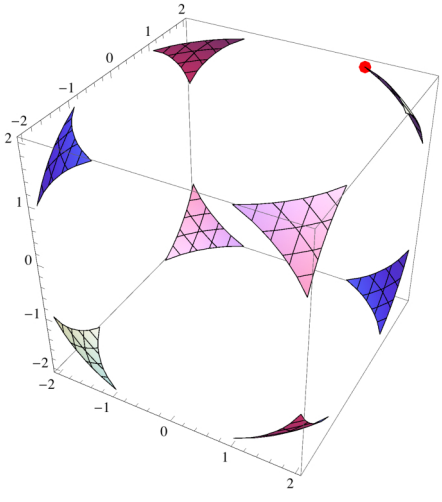
Here the maximum is attained at the boundary of the boundary of the solution set of the constraints.

In[44]:=
```
m = Maximize[x + 2 y + 2 z, x^2 + y^2 + z^2 == 9 && -2 ≤ x ≤ 2 && -2 ≤ y ≤ 2 && -2 ≤ z ≤ 2, {x, y, z}]
```

Out[44]=
```
{9, {x → 1, y → 2, z → 2}}
```

In[45]:=
```
Show[{ContourPlot3D[x^2 + y^2 + z^2 - 9 == 0, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}],
  Graphics3D[{PointSize[0.03], Red, Point[{x, y, z} /. m[[2]]]}]}]
```

Out[45]=



The Lagrange multiplier method works for some optimization problems involving transcendental functions.

In[46]:=
```
Minimize[{y + Sin[10 x], y^3 == Cos[5 x] && -5 ≤ x ≤ 5 && -5 ≤ y ≤ 5}, {x, y}]
```

```
Minimize::ztest: Unable to decide whether numeric quantities
    {Sin[4 (π - ArcTan[AlgebraicNumber[≪2≫]])] - Sin[4 (2 π - ArcTan[≪1≫])], ≪5≫, Sin[4 (π - ArcTan[
          AlgebraicNumber[≪2≫]])] + Sin[≪1≫]}
    are equal to zero. Assuming they are.
```

Out[46]=
$$\left\{\text{AlgebraicNumber}\left[\text{Root}\left[43\,046\,721 - 95\,659\,380\,\#1^2 - 59\,049\,\#1^3 + 78\,653\,268\,\#1^4 - 32\,805\,\#1^5 - 29\,052\,108\,\#1^6 - 7290\,\#1^7 + 4\,763\,286\,\#1^8 - 810\,\#1^9 - 358\,668\,\#1^{10} - 45\,\#1^{11} + 11\,988\,\#1^{12} - \#1^{13} - 180\,\#1^{14} + \#1^{16}\,\&,\, 6\right],\right.\right.$$
$$\left\{0, \frac{2825}{256}, \frac{1}{81}, -\frac{10\,645}{768}, \frac{1271}{186\,624}, \frac{117\,277}{20\,736}, \frac{421}{279\,936}, -\frac{177\,851}{186\,624}, \frac{157}{944\,784}, \frac{13\,523}{186\,624}, \frac{625}{68\,024\,448}, -\frac{36\,749}{15\,116\,544}, \frac{83}{408\,146\,688}, \frac{4975}{136\,048\,896}, 0, -\frac{83}{408\,146\,688}\right\}\Big] - \text{Sin}\Big[$$
$$4\left(\pi - \text{ArcTan}\left[\text{AlgebraicNumber}\left[\text{Root}\left[43\,046\,721 - 95\,659\,380\,\#1^2 - 59\,049\,\#1^3 + 78\,653\,268\,\#1^4 - 32\,805\,\#1^5 - 29\,052\,108\,\#1^6 - 7290\,\#1^7 + 4\,763\,286\,\#1^8 - 810\,\#1^9 - 358\,668\,\#1^{10} - 45\,\#1^{11} + 11\,988\,\#1^{12} - \#1^{13} - 180\,\#1^{14} + \#1^{16}\,\&,\, 6\right],\, \left\{0, \frac{1}{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right\}\right]\right]\right),$$
$$\left\{x \to -\frac{2}{5}\left(\pi - \text{ArcTan}\left[\text{AlgebraicNumber}\left[\text{Root}\left[43\,046\,721 - 95\,659\,380\,\#1^2 - 59\,049\,\#1^3 + 78\,653\,268\,\#1^4 - 32\,805\,\#1^5 - 29\,052\,108\,\#1^6 - 7290\,\#1^7 + 4\,763\,286\,\#1^8 - 810\,\#1^9 - 358\,668\,\#1^{10} - 45\,\#1^{11} + 11\,988\,\#1^{12} - \#1^{13} - 180\,\#1^{14} + \#1^{16}\,\&,\, 6\right],\, \left\{0, \frac{1}{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right\}\right]\right]\right),$$
$$y \to \text{AlgebraicNumber}\left[\text{Root}\left[43\,046\,721 - 95\,659\,380\,\#1^2 - 59\,049\,\#1^3 + 78\,653\,268\,\#1^4 - 32\,805\,\#1^5 - 29\,052\,108\,\#1^6 - 7290\,\#1^7 + 4\,763\,286\,\#1^8 - 810\,\#1^9 - 358\,668\,\#1^{10} - 45\,\#1^{11} + 11\,988\,\#1^{12} - \#1^{13} - 180\,\#1^{14} + \#1^{16}\,\&,\, 6\right],\right.$$
$$\left.\left.\left\{0, \frac{2825}{256}, \frac{1}{81}, -\frac{10\,645}{768}, \frac{1271}{186\,624}, \frac{117\,277}{20\,736}, \frac{421}{279\,936}, -\frac{177\,851}{186\,624}, \frac{157}{944\,784}, \frac{13\,523}{186\,624}, \frac{625}{68\,024\,448}, -\frac{36\,749}{15\,116\,544}, \frac{83}{408\,146\,688}, \frac{4975}{136\,048\,896}, 0, -\frac{83}{408\,146\,688}\right\}\right]\right\}\right\}$$

In[47]:=
```
N[%, 20]
```

Out[47]=
```
{-1.9007500346675151230, {x → -0.77209298024514961134, y → -0.90958837944086038552}}
```

Optimization over the Integers

Integer Linear Programming

An integer linear programming problem is an optimization problem in which the objective function is linear, the constraints are linear and bounded, and the variables range over the integers.

To solve an integer linear programming problem *Mathematica* first solves the equational constraints, reducing the problem to one containing inequality constraints only. Then it uses lattice reduction techniques to put the inequality system in a simpler form. Finally, it solves the simplified optimization problem using a branch-and-bound method.

This solves a randomly generated integer linear programming problem with 7 variables.

```
In[48]:= SeedRandom[1];
         A = Table[RandomInteger[{-1000, 1000}], {3}, {7}];
         α = Table[RandomInteger[{-1000, 1000}], {3}];
         B = Table[RandomInteger[{-1000, 1000}], {3}, {7}];
         β = Table[RandomInteger[{-1000, 1000}], {3}];
         γ = Table[RandomInteger[{-1000, 1000}], {7}];
         X = x /@ Range[7];
         eqns = And @@ Thread[A.X == α];
         ineqs = And @@ Thread[B.X ≤ β];
         bds = And @@ Thread[X ≥ 0] && Total[X] ≤ 10^100;
         Minimize[{γ.X, eqns && ineqs && bds && X ∈ Integers}, X]
Out[58]= {448 932, {x[1] → 990, x[2] → 1205, x[3] → 219, x[4] → 60, x[5] → 823, x[6] → 137, x[7] → 34}}
```

Optimization over the Reals Combined with Integer Solution Finding

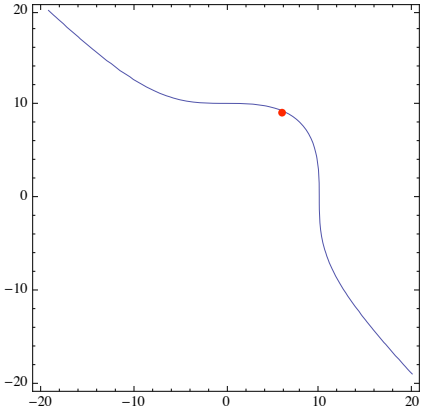
Suppose a function $f \in \mathbb{Z}[x]$ needs to be minimized over the integer solution set of constraints $\Phi(x)$. Let m be the minimum of f over the real solution set of $\Phi(x)$. If there exists an integer point satisfying $f(x) = \lceil m \rceil \wedge \Phi(x)$, then $\lceil m \rceil$ is the minimum of f over the integer solution set of Φ. Otherwise you try to find an integer solution of $f(x) = \lceil m \rceil + 1 \wedge \Phi(x)$, and so on. This heuristic works if you can solve the real optimization problem and all the integer solution finding problems, and you can find an integer solution within a predefined number of attempts. (By default *Mathematica* tries 10 candidate optimum values. This can be changed using the `IntegerOptimumCandidates` system option.)

This finds a point with integer coordinates maximizing $x + y$ among the points lying below the cubic $x^3 + y^3 = 1000$.

```
In[59]:= m = Maximize[{x + y, x^3 + y^3 ≤ 1000 && (x | y) ∈ Integers}, {x, y}]
Out[59]= {15, {x → 6, y → 9}}
```

```
In[60]:= Show[{ContourPlot[x^3 + y^3 == 1000, {x, -20, 20}, {y, -20, 20}],
    Graphics[{PointSize[0.02], Red, Point[{x, y} /. m[[2]]]}]}]
```

Out[60]=



Comparison of Constrained Optimization Functions

`NMinimize`, `NMaximize`, `Minimize` and `Maximize` employ global optimization algorithms, and are thus suitable when a global optimum is needed.

`Minimize` and `Maximize` can find exact global optima for a class of optimization problems containing arbitrary polynomial problems. However, the algorithms used have a very high asymptotic complexity and therefore are suitable only for problems with a small number of variables.

`Maximize` always finds a global maximum, even in cases that are numerically unstable. The left-hand side of the constraint here is $\left(x^2 + y^2 - 10^{10}\right)^2 \left(x^2 + y^2\right)$.

```
In[1]:= Maximize[{x + y,
    100 000 000 000 000 000 000 x^2 - 20 000 000 000 x^4 + x^6 + 100 000 000 000 000 000 000 y^2 -
      40 000 000 000 x^2 y^2 + 3 x^4 y^2 - 20 000 000 000 y^4 + 3 x^2 y^4 + y^6 ≤ 1}, {x, y}] // N[#, 20] &
Out[1]= {141 421.35623730957559, {x → 70 710.678118654787795, y → 70 710.678118654787795}}
```

This input differs from the previous one only in the twenty-first decimal digit, but the answer is quite different, especially the location of the maximum point. For an algorithm using 16 digits of precision both problems look the same, hence it cannot possibly solve them both correctly.

```
In[2]:= Maximize[{x + y,
          100 000 000 000 000 000 001 x^2 - 20 000 000 000 x^4 + x^6 + 100 000 000 000 000 000 000 y^2 -
            40 000 000 000 x^2 y^2 + 3 x^4 y^2 - 20 000 000 000 y^4 + 3 x^2 y^4 + y^6 <= 1}, {x, y}] // N[#, 20] &
Out[2]= {100 000.99999500000000, {x → 1.0000000000000000000, y → 99 999.999995000000000}}
```

NMaximize, which by default uses machine-precision numbers, is not able to solve either of the problems.

```
In[3]:= NMaximize[
          {x + y, 100 000 000 000 000 000 000 x^2 - 20 000 000 000 x^4 + x^6 + 100 000 000 000 000 000 000 y^2 -
            40 000 000 000 x^2 y^2 + 3 x^4 y^2 - 20 000 000 000 y^4 + 3 x^2 y^4 + y^6 <= 1}, {x, y}]

        NMaximize::incst:
          NMaximize was unable to generate any initial points satisfying the inequality constraints
            {-1 + 10000000000000000000 x^2 - 20000000000 x^4 + x^6 + 10000000000000000000 y^2 - <<1>> +
              3 x^4 y^2 - 20000000000 y^4 + 3 x^2 y^4 + y^6 <= 0}. The initial
            region specified may not contain any feasible points. Changing the initial
            region or specifying explicit initial points may provide a better solution. >>

Out[3]= {1.35248×10^-10, {x → 4.69644×10^-11, y → 8.82834×10^-11}}

In[4]:= NMaximize[
          {x + y, 100 000 000 000 000 000 001 x^2 - 20 000 000 000 x^4 + x^6 + 100 000 000 000 000 000 000 y^2 -
            40 000 000 000 x^2 y^2 + 3 x^4 y^2 - 20 000 000 000 y^4 + 3 x^2 y^4 + y^6 <= 1}, {x, y}]

        NMaximize::incst:
          NMaximize was unable to generate any initial points satisfying the inequality constraints
            {-1 + 10000000000000000001 x^2 - 20000000000 x^4 + x^6 + 10000000000000000000 y^2 - <<1>> +
              3 x^4 y^2 - 20000000000 y^4 + 3 x^2 y^4 + y^6 <= 0}. The initial
            region specified may not contain any feasible points. Changing the initial
            region or specifying explicit initial points may provide a better solution. >>

Out[4]= {1.35248×10^-10, {x → 4.69644×10^-11, y → 8.82834×10^-11}}
```

FindMinimum only attempts to find a local minimum, therefore is suitable when a local optimum is needed, or when it is known in advance that the problem has only one optimum or only a few optima that can be discovered using different starting points.

Even for local optimization, it may still be worth using NMinimize for small problems. NMinimize uses one of the four direct search algorithms (Nelder-Mead, differential evolution, simulated annealing, and random search), then finetunes the solution by using a combination of KKT solution, the interior point, and a penalty method. So if efficiency is not an issue, NMinimize should be more robust than FindMinimum, in addition to being a global optimizer.

This shows the default behavior of `NMinimize` on a problem with four variables.

```
In[5]:= Clear[f, x, y, z, t];
        f = -Log[x] - 2 Log[y] - 3 Log[y] - 3 Log[t];
        cons = {200 x^2 + y^2 + z^2 + t^2 == 100, x > 0, y > 0, z > 0, t > 0};
        vars = {x, y, z, t};
        sol = NMinimize[{f, cons}, vars]
Out[9]= {-13.8581, {t → 5.7735, x → 0.235702, y → 7.45356, z → 0.00177238}}
```

This shows that two of the post-processors, KKT and `FindMinimum`, do not give the default result. Notice that for historical reasons, the name `FindMinimum`, when used as an option value of `PostProcess`, stands for the process where a penalty method is used to convert the constrained optimization problem into unconstrained optimization methods and then solved using (unconstrained) `FindMinimum`.

```
In[10]:= sol = NMinimize[{f, cons}, vars, Method → {NelderMead, PostProcess → KKT}]

NMinimize::nosat: Obtained solution does not satisfy the following
    constraints within Tolerance -> 0.001`: {100 - t^2 - 200 x^2 - y^2 - z^2 == 0}. >>
Out[10]= {0.759899, {t → 9.98441, x → 0.0103018, y → 0.539287, z → 0.0246594}}

In[11]:= sol = NMinimize[{f, cons}, vars, Method → {NelderMead, PostProcess → FindMinimum}]
Out[11]= {-13.8573, {t → 5.84933, x → 0.233007, y → 7.41126, z → 0.00968789}}
```

However, if efficiency is important, `FindMinimum` can be used if you just need a local minimum, or you can provide a good starting point, or you know the problem has only one minimum (e.g., convex), or your problem is large/expensive. This uses `FindMinimum` and `NMinimize` to solve the same problem with seven variables. The constraints are relatively expensive to compute. Clearly `FindMinimum` in this case is much faster than `NMinimize`.

```
In[12]:= Clear[f, cons, vars, x];
         {f, cons, vars} =
           {20 x[2] x[6] / (x[1]^2 x[4] x[5]^2) + 15 x[3] x[4] / (x[1] x[2]^2 x[5] x[7]^0.5`) +
             10 x[1] x[4]^2 x[7]^0.125` / (x[2] x[6]^3) +
             25 x[1]^2 x[2]^2 x[5]^0.5` x[7] / (x[3] x[6]^2),
            {0.1` ≤ x[1] ≤ 10, 0.1` ≤ x[2] ≤ 10, 0.1` ≤ x[3] ≤ 10,
             0.1` ≤ x[4] ≤ 10, 0.1` ≤ x[5] ≤ 10, 0.1` ≤ x[6] ≤ 10, 0.01` ≤ x[7] ≤ 10,
             1 - 0.2` x[3] x[6]^(2/3) x[7]^0.25` / (x[2] x[4]^0.5`) - 0.7` x[1]^3 x[2] x[6] x[7]^0.5` / x[3]^2 -
               0.5` x[1]^0.5` x[7] / (x[3] x[6]^2) ≥ 0,
             1 - 3.1` x[2]^0.5` x[6]^(1/3) / (x[1] x[4]^2 x[5]) - 1.3` x[2] x[6] / (x[1]^0.5` x[3] x[5]) -
               0.8` x[3] x[6]^2 / (x[4] x[5]) ≥ 0,
```

$$1 - \frac{x[2]\,x[3]^{0.5`}\,x[5]}{x[1]} - \frac{0.1`\,x[2]\,x[5]}{x[3]^{0.5`}\,x[6]\,x[7]^{0.5`}} - \frac{2\,x[1]\,x[5]\,x[7]^{1/3}}{x[3]^{1.5`}\,x[6]} - \frac{0.65`\,x[3]\,x[5]\,x[7]}{x[2]^2\,x[6]} \geq 0,\ 1 - \frac{0.3`\,x[1]^{0.5`}\,x[2]^2\,x[3]\,x[4]^{1/3}\,x[7]^{0.25`}}{x[5]^{2/3}} - \frac{0.2`\,x[2]\,x[5]^{0.5`}\,x[7]^{1/3}}{x[1]^2\,x[4]} - \frac{0.5`\,x[4]\,x[7]^{0.5`}}{x[3]^2} - \frac{0.4`\,x[3]\,x[5]\,x[7]^{0.75`}}{x[1]^3\,x[2]^2} \geq 0,$$

$$\frac{20\,x[2]\,x[6]}{x[1]^2\,x[4]\,x[5]^2} + \frac{15\,x[3]\,x[4]}{x[1]\,x[2]^2\,x[5]\,x[7]^{0.5`}} + \frac{10\,x[1]\,x[4]^2\,x[7]^{0.125`}}{x[2]\,x[6]^3} + \frac{25\,x[1]^2\,x[2]^2\,x[5]^{0.5`}\,x[7]}{x[3]\,x[6]^2} \geq 100,\ \frac{20\,x[2]\,x[6]}{x[1]^2\,x[4]\,x[5]^2} + \frac{15\,x[3]\,x[4]}{x[1]\,x[2]^2\,x[5]\,x[7]^{0.5`}} + \frac{10\,x[1]\,x[4]^2\,x[7]^{0.125`}}{x[2]\,x[6]^3} + \frac{25\,x[1]^2\,x[2]^2\,x[5]^{0.5`}\,x[7]}{x[3]\,x[6]^2} \leq 3000\Big\},$$

$$\{x[1], x[2], x[3], x[4], x[5], x[6], x[7]\}\Big\};$$

In[14]:=
```
FindMinimum[{f, cons}, vars] // Timing
```

Out[14]=
```
{0.541, {911.881, {x[1] → 3.89625, x[2] → 0.809359, x[3] → 2.66439,
    x[4] → 4.30091, x[5] → 0.853555, x[6] → 1.09529, x[7] → 0.0273105}}}
```

In[15]:=
```
NMinimize[{f, cons}, vars] // Timing
```

NMinimize::incst: NMinimize was unable to generate any initial points satisfying the inequality constraints $\left\{-1 + \frac{3.1\,x[2]^{0.5}\,x[6]^{1/3}}{x[1]\,x[4]^2\,x[5]} + \frac{1.3\,x[2]\,x[6]}{x[1]^{0.5}\,x[3]\,x[5]} + \frac{0.8\,x[3]\,x[6]^2}{x[4]\,x[5]} \leq 0, \ll 4 \gg, -1 + \ll 4 \gg \leq 0\right\}$. The initial region specified may not contain any feasible points. Changing the initial region or specifying explicit initial points may provide a better solution. ≫

NMinimize::incst: NMinimize was unable to generate any initial points satisfying the inequality constraints $\left\{-1 + \frac{3.1\,x[2]^{0.5}\,x[6]^{1/3}}{x[1]\,x[4]^2\,x[5]} + \frac{1.3\,x[2]\,x[6]}{x[1]^{0.5}\,x[3]\,x[5]} + \frac{0.8\,x[3]\,x[6]^2}{x[4]\,x[5]} \leq 0, \ll 4 \gg, -1 + \ll 4 \gg \leq 0\right\}$. The initial region specified may not contain any feasible points. Changing the initial region or specifying explicit initial points may provide a better solution. ≫

Out[15]=
```
{8.151, {911.881, {x[1] → 3.89625, x[2] → 0.809359, x[3] → 2.66439,
    x[4] → 4.30091, x[5] → 0.853555, x[6] → 1.09529, x[7] → 0.0273105}}}
```

Constrained Optimization in *Mathematica*—References

[1] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization* 2 (1992): 575-601.

[2] Nelder, J.A. and R. Mead. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (1965): 308-313.

[3] Ingber, L. "Simulated Annealing: Practice versus Theory." *Mathematical Computer Modelling* 18, no. 11 (1993): 29-57.

[4] Price, K. and R. Storn. "Differential Evolution." *Dr. Dobb's Journal* 264 (1997): 18-24.