Wolfram *Mathematica*® Tutorial Collection

UNCONSTRAINED 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:
Rob Knapp

Printed in the United States of America.

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

Contents

Introduction to Unconstrained Optimization

Mathematica has a collection of commands that do unconstrained optimization (`FindMinimum` and `FindMaximum`) and solve nonlinear equations (`FindRoot`) and nonlinear fitting problems (`FindFit`). All these functions work, in general, by doing a search, starting at some initial values and taking steps that decrease (or for `FindMaximum`, increase) an objective or merit function.

The search process for `FindMaximum` is somewhat analogous to a climber trying to reach a mountain peak in a thick fog; at any given point, basically all that climbers know is their position, how steep the slope is, and the direction of the fall line. One approach is always to go uphill. As long as climbers go uphill steeply enough, they will eventually reach a peak, though it may not be the highest one. Similarly, in a search for a maximum, most methods are ascent methods where every step increases the height and stops when it reaches any peak, whether it is the highest one or not.

The analogy with hill climbing can be reversed to consider descent methods for finding local minima. For the most part, the literature in optimization considers the problem of finding minima, and since this applies to most of the *Mathematica* commands, from here on, this documentation will follow that convention.

For example, the function $x \sin(x+1)$ is not bounded from below, so it has no global minimum, but it has an infinite number of local minima.

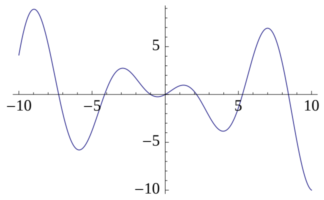
This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows a plot of the function `x Sin[x + 1]`.

```
In[2]:= Plot[x Sin[x + 1], {x, -10, 10}]
```

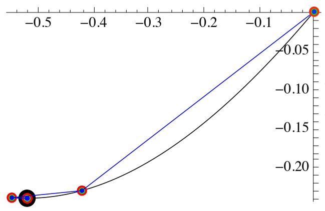
Out[2]=



This shows the steps taken by `FindMinimum` for the function `x Sin[x + 1]` starting at $x = 0$.

```
In[3]:= FindMinimumPlot[x Sin[x + 1], {x, 0}]
```

```
Out[3]= {{-0.240125, {x → -0.520269}}, {Steps → 5, Function → 6, Gradient → 6},
```



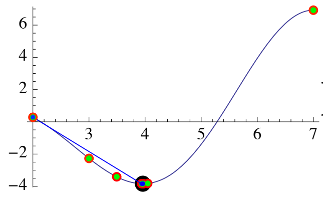
}

The `FindMinimumPlot` command is defined in the ``Optimization`UnconstrainedProblems` `` package loaded automatically by this notebook. It runs `FindMinimum`, keeps track of the function and gradient evaluations and steps taken during the search (using the `EvaluationMonitor` and `StepMonitor` options), and shows them superimposed on a plot of the function. Steps are indicated with blue lines, function evaluations are shown with green points, and gradient evaluations are shown with red points. The minimum found is shown with a large black point. From the plot, it is clear that `FindMinimum` has found a local minimum point.

This shows the steps taken by `FindMinimum` for the function `x Sin[x + 1]` starting at $x = 2$.

```
In[4]:= FindMinimumPlot[x Sin[x + 1], {x, 2}]
```

```
Out[4]= {{-3.83922, {x → 3.95976}}, {Steps → 4, Function → 9, Gradient → 9},
```



}

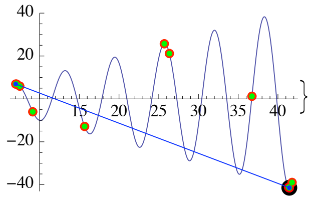
Starting at 2, `FindMinimum` heads to different local minima, at which the function is smaller than at the first minimum found.

From these two plots, you might come to the conclusion that if you start at a point where the function is sloping downward, you will always head toward the next minimum in that direction. However, this is not always the case; the steps `FindMinimum` takes are typically determined using the value of the function and its derivatives, so if the derivative is quite small, `FindMinimum` may think it has to go quite a long way to find a minimum point.

This shows the steps taken by `FindMinimum` for the function `x Sin[x + 1]` starting at `x = 7`.

```
In[5]:= FindMinimumPlot[x Sin[x + 1], {x, 7}]
```

```
Out[5]= {{-41.4236, {x → 41.4356}}, {Steps → 3, Function → 14, Gradient → 14},
```



```
}
```

When starting at `x = 7`, which is near a local maximum, the first step is quite large, so `FindMinimum` returns a completely different local minimum.

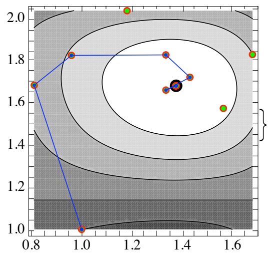
All these commands have "find" in their name because, in general, their design is to search to find any point where the desired condition is satisfied. The point found may not be the only one (in the case of roots) or even the best one (in the case of fits, minima, or maxima), or, as you have seen, not even the closest one to the starting condition. In other words, the goal is to find any point at which there is a root or a local maximum or minimum. In contrast, the function `NMinimize` tries harder to find the global minimum for the function, but `NMinimize` is also generally given constraints to bound the problem domain. However, there is a price to pay for this generality—`NMinimize` has to do much more work and, in fact, may call one of the `"Find"` functions to polish a result at the end of its process, so it generally takes much more time than the `"Find"` functions.

In two dimensions, the minimization problem is more complicated because both a step direction and step length need to be determined.

This shows the steps taken by `FindMinimum` to find a local minimum of the function $\cos(x^2 - 3y) + \sin(x^2 + y^2)$ starting at the point $\{x, y\} = \{1, 1\}$.

```
In[6]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1}, {y, 1}}]
```

```
Out[6]= {{-2., {x → 1.37638, y → 1.67868}}, {Steps → 9, Function → 13, Gradient → 13},
```

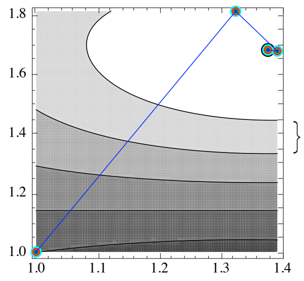


```
}
```

The `FindMinimumPlot` command for two dimensions is similar to the one-dimensional case, but it shows the steps and evaluations superimposed on a contour plot of the function. In this example, it is apparent that `FindMinimum` needed to change direction several times to get to the local minimum. You may notice that the first step starts in the direction of steepest descent (i.e., perpendicular to the contour or parallel to the gradient). Steepest descent is indeed a possible strategy for local minimization, but it often does not converge quickly. In subsequent steps in this example, you may notice that the search direction is not exactly perpendicular to the contours. The search is using information from past steps to try to get information about the curvature of the function, which typically gives it a better direction to go. Another strategy, which usually converges faster, but can be more expensive, is to use the second derivative of the function. This is usually referred to as "Newton's" method.

This shows the steps taken using Newton's method.

```
In[7]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1}, {y, 1}}, Method → Newton]

Out[7]= {{-2., {x → 1.37638, y → 1.67868}},

         {Steps → 5, Function → 6, Gradient → 6, Hessian → 6},                    }
```



In this example, it is clear that the extra information that "Newton's" method uses about the curvature of the function makes a big difference in how many steps it takes to get to the minimum. Even though Newton's method takes fewer steps, it may take more total execution time since the symbolic Hessian has to be computed once and then evaluated numerically at each step.

Usually there are tradeoffs between the rate of convergence or total number of steps taken and cost per step. Depending on the size of the problems you want to solve, you may want to pick a particular method to best match that tradeoff for a particular problem. This documentation is intended to help you understand those choices as well as some ways to get the best results from the functions in *Mathematica*. For the most part, examples will be used to illustrate the ideas, but a limited exposition on the mathematical theory behind the methods will be given so that you can better understand how the examples work.

For the most part, local minimization methods for a function f are based on a quadratic model

$$q_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p. \tag{1}$$

The subscript k refers to the k^{th} iterative step. In Newton's method, the model is based on the exact Hessian matrix, $B_k = \nabla^2 f(x_k)$, but other methods use approximations to $\nabla^2 f(x_k)$, which are typically less expensive to compute. A trial step s_k is typically computed to be the minimizer of the model, which satisfies the system of linear equations.

$$B_k s_k = -\nabla f(x_k)$$

If f is sufficiently smooth and x_k is sufficiently close to a local minimum, then with $B_k = \nabla^2 f(x_k)$, the sequence of steps $x_{k+1} = s_k + x_k$ is guaranteed to converge to the local minimum. However, in a typical search, the starting value is rarely close enough to give the desired convergence. Furthermore, B_k is often an approximation to the actual Hessian and, at the beginning of a search, the approximation is frequently quite inaccurate. Thus, it is necessary to provide additional control to the step sequence to improve the chance and rate of convergence. There are two frequently used methods for controlling the steps: line search and trust region methods.

In a "line search" method, for each trial step s_k found, a one-dimensional search is done along the direction of s_k so that $x_{k+1} = x_k + \alpha_k s_k$. You could choose α_k so that it minimizes $f(x_{k+1})$ in this direction, but this is excessive, and with conditions that require that $f(x_{k+1})$ decreases sufficiently in value and slope, convergence for reasonable approximations B_k can be proven. *Mathematica* uses a formulation of these conditions called the Wolfe conditions.

In a "trust region" method, a radius Δ_k within which the quadratic model $q_k(p)$ in equation (1) is “trusted” to be reasonably representative of the function. Then, instead of solving for the unconstrained minimum of (1), the trust region method tries to find the constrained minimum of (1) with $\|p\| \leq \Delta_k$. If the x_k are sufficiently close to a minimum and the model is good, then often the minimum lies within the circle, and convergence is quite rapid. However, near the start of a search, the minimum will lie on the boundary, and there are a number of techniques to find an approximate solution to the constrained problem. Once an approximate solution is found, the actual reduction of the function value is compared to the predicted reduction in the function value and, depending on how close the actual value is to the predicted, an adjustment is made for Δ_{k+1}.

For symbolic minimization of a univariate smooth function, all that is necessary is to find a point at which the derivative is zero and the second derivative is positive. In multiple dimensions, this means that the gradient vanishes and the Hessian needs to be positive definite. (If the Hessian is positive semidefinite, the point is a minimizer, but is not necessarily a strict one.) As a numerical algorithm converges, it is necessary to keep track of the convergence and make some judgment as to when a minimum has been approached closely enough. This is based on the sequence of steps taken and the values of the function, its gradient, and possibly its Hessian at these points. Usually, the *Mathematica* `Find`... functions will issue a message if they cannot be fairly certain that this judgment is correct. However, keep in mind that discontinuous functions or functions with rapid changes of scale can fool any numerical algorithm.

When solving "nonlinear equations", many of the same issues arise as when finding a "local minimum". In fact, by considering a so-called merit function, which is zero at the root of the equations, it is possible to use many of the same techniques as for minimization, but with the advantage of knowing that the minimum value of the function is 0. It is not always advantageous to use this approach, and there are some methods specialized for nonlinear equations.

Most examples shown will be from one and two dimensions. This is by no means because *Mathematica* is restricted to computing with such small examples, but because it is much easier to visually illustrate the main principles behind the theory and methods with such examples.

Methods for Local Minimization

Introduction to Local Minimization

The essence of most methods is in the local quadratic model

$$q_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

that is used to determine the next step. The `FindMinimum` function in *Mathematica* has five essentially different ways of choosing this model, controlled by the method option. These methods are similarly used by `FindMaximum` and `FindFit`.

`"Newton"`	use the exact Hessian or a finite difference approximation if the symbolic derivative cannot be computed
`"QuasiNewton"`	use the quasi-Newton BFGS approximation to the Hessian built up by updates based on past steps
`"LevenbergMarquardt"`	a Gauss-Newton method for least-squares problems; the Hessian is approximated by $J^T J$, where J is the Jacobian of the residual function
`"ConjugateGradient"`	a nonlinear version of the conjugate gradient method for solving linear systems; a model Hessian is never formed explicitly
`"PrincipalAxis"`	works without using any derivatives, not even the gradient, by keeping values from past steps; it requires two starting conditions in each variable

Basic method choices for `FindMinimum`.

With `Method -> Automatic`, *Mathematica* uses the "quasi-Newton" method unless the problem is structurally a sum of squares, in which case the Levenberg-Marquardt variant of the "Gauss-Newton" method is used. When given two starting conditions in each variable, the "principal axis" method is used.

Newton's Method

One significant advantage *Mathematica* provides is that it can symbolically compute derivatives. This means that when you specify `Method -> "Newton"` and the function is explicitly differentiable, the symbolic derivative will be computed automatically. On the other hand, if the function is not in a form that can be explicitly differentiated, *Mathematica* will use finite difference approximations to compute the Hessian, using structural information to minimize the number of evaluations required. Alternatively you can specify a *Mathematica* expression, which will give the Hessian with numerical values of the variables.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

In this example, `FindMinimum` computes the Hessian symbolically and substitutes numerical values for x and y when needed.

```
In[2]:= FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1}, {y, 1}}, Method -> "Newton"]
Out[2]= {-2., {x → 1.37638, y → 1.67868}}
```

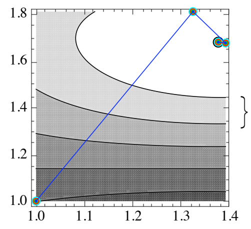
This defines a function that is only intended to evaluate for numerical values of the variables.

```
In[3]:= f[x_?NumberQ, y_?NumberQ] := Cos[x^2 - 3 y] + Sin[x^2 + y^2]
```

The derivative of this function cannot be found symbolically since the function has been defined only to evaluate with numerical values of the variables.

This shows the steps taken by `FindMinimum` when it has to use finite differences to compute the gradient and Hessian.

```
In[4]:= FindMinimumPlot[f[x, y], {{x, 1}, {y, 1}}, 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. ≫

Out[4]= {{-2., {x → 1.37638, y → 1.67867}},

  {Steps → 4, Function → 89, Gradient → 26, Hessian → 5},                }
```



When the gradient and Hessian are both computed using finite differences, the error in the Hessian may be quite large and it may be better to use a different method. In this case, `FindMinimum` does find the minimum quite accurately, but cannot be sure because of inadequate derivative information. Also, the number of function and gradient evaluations is much greater than in the example with the symbolic derivatives computed automatically because extra evaluations are required to approximate the gradient and Hessian, respectively.

If it is possible to supply the gradient (or the function is such that it can be computed automatically), the method will typically work much better. You can give the gradient using the `Gradient` option, which has several ways you can "specify derivatives".

This defines a function that returns the gradient for numerical values of x and y.

```
In[5]:= g[x_?NumberQ, y_?NumberQ] = Map[D[Cos[x^2 - 3 y] + Sin[x^2 + y^2], #] &, {x, y}]

Out[5]= {2 x Cos[x² + y²] - 2 x Sin[x² - 3 y], 2 y Cos[x² + y²] + 3 Sin[x² - 3 y]}
```

This tells `FindMinimum` to use the supplied gradient. The Hessian is computed using finite differences of the gradient.

```
In[6]:= FindMinimum[f[x, y], {{x, 1}, {y, 1}}, Gradient → g[x, y], Method → "Newton"]

Out[6]= {-2., {x → 1.37638, y → 1.67868}}
```

If you can provide a program that gives the Hessian, you can provide this also. Because the Hessian is only used by Newton's method, it is given as a method option of Newton.

This defines a function that returns the Hessian for numerical values of x and y.

```
In[7]:= h[x_?NumberQ, y_?NumberQ] =
         Outer[D[Cos[x^2 - 3 y] + Sin[x^2 + y^2], ##] &, {x, y}, {x, y}]

Out[7]= {{-4 x² Cos[x² - 3 y] + 2 Cos[x² + y²] - 2 Sin[x² - 3 y] - 4 x² Sin[x² + y²],
          6 x Cos[x² - 3 y] - 4 x y Sin[x² + y²]},
         {6 x Cos[x² - 3 y] - 4 x y Sin[x² + y²], -9 Cos[x² - 3 y] + 2 Cos[x² + y²] - 4 y² Sin[x² + y²]}}
```

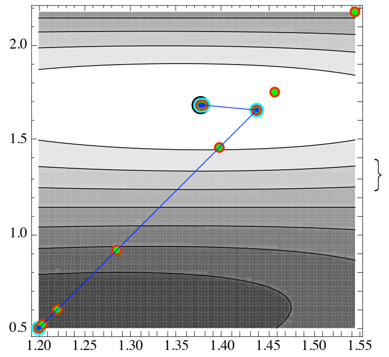
This tells `FindMinimum` to use the supplied gradient and Hessian.

```
In[8]:= FindMinimum[f[x, y], {{x, 1}, {y, 1}},
         Gradient → g[x, y], Method → {"Newton", "Hessian" → h[x, y]}]

Out[8]= {-2., {x → 1.37638, y → 1.67868}}
```

In principle, Newton's method uses the Hessian computed either by evaluating the symbolic derivative or by using finite differences. However, the convergence for the method computed this way depends on the function being convex, in which case the Hessian is always positive definite. It is common that a search will start at a location where this condition is violated, so the algorithm needs to take this possibility into account.

Here is an example where the search starts near a local maximum.

```
In[9]:=  FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
          {{x, 1.2}, {y, .5}}, Method -> "Newton"]

Out[9]=  {{-2., {x → 1.37638, y → 1.67868}},

          {Steps → 4, Function → 11, Gradient → 11, Hessian → 5},
```



```
}
```

When sufficiently near a local maximum, the Hessian is actually negative definite.

This computes the eigenvalues of the Hessian near the local maximum.

```
In[10]:=  Eigenvalues[h[1.2, .5]]

Out[10]=  {-15.7534, -6.0478}
```

If you were to only apply the Newton step formula in cases where the Hessian is not positive definite, it is possible to get a step direction that does not lead to a decrease in the function value.

This computes the directional derivative for the direction found by solving $\nabla^2 f(x_k)\, s_0 = -\nabla f\,(x_k)$. Since it is positive, moving in this direction will locally increase the function value.

```
In[11]:=  LinearSolve[h[1.2, .5], -g[1.2, .5]].g[1.2, .5]

Out[11]=  0.0172695
```

It is crucial for the convergence of line search methods that the direction be computed using a positive definite quadratic model B_k since the search process and convergence results derived from it depend on a direction with sufficient descent. See "Line Search Methods". *Mathematica* modifies the Hessian by a diagonal matrix E_k with entries large enough so that $B_k = \nabla^2 f(x_k) + E_k$ is positive definite. Such methods are sometimes referred to as modified Newton methods. The modification to B_k is done during the process of computing a Cholesky decomposition somewhat along the lines described in [GMW81], both for dense and sparse Hessians. The modification is only done if $\nabla^2 f(x_k)$ is not positive definite. This decomposition method is accessible through `LinearSolve` if you want to use it independently.

This computes the step using $B_0\, s_0 = -\nabla f\,(x_k)$, where B_0 is determined as the Cholesky factors of the Hessian are being computed.

```
In[12]:= LinearSolve[h[1.2, .5], -g[1.2, .5],
          Method → {"Cholesky", "Modification" → "Minimal"}]

Out[12]= {0.00405502, 0.0196737}
```

The computed step is in a descent direction.

```
In[13]:= %.g[1.2, .5]

Out[13]= -0.00645255
```

Besides the robustness of the (modified) Newton method, another key aspect is its convergence rate. Once a search is close enough to a local minimum, the convergence is said to be q-quadratic, which means that if x^* is the local minimum point, then

$$\|x_{k+1} - x^*\| \leq \beta\, \|x_k - x^*\|^2$$

for some constant $\beta > 0$.

At machine precision, this does not always make a substantial difference since it is typical that most of the steps are spent getting near to the local minimum. However, if you want a root to extremely high precision, Newton's method is usually the best choice because of the rapid convergence.

This computes a very high-precision solution using Newton's method. The precision is adaptively increased from machine precision (the precision of the starting point) to the maximal working precision of 100000 digits. `Reap` is used with `Sow` to save the steps taken. Counters are used to track and print the number of function evaluations and steps used.

```
In[14]:= First[Timing[Block[{e = 0, s = 0}, {{min, minpoint}, {points}} =
             Reap[FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
               {{x, 1.}, {y, 1.}}, Method -> "Newton", WorkingPrecision → 100 000,
               StepMonitor :→ (s++; Sow[{x, y}]), EvaluationMonitor :→ e++]];
           Print[s, " steps and ", e, " evaluations"]]]]

           17 steps and 27 evaluations

Out[14]= 4.56134
```

When the option `WorkingPrecision -> prec` is used, the default for the `AccuracyGoal` and `PrecisionGoal` is *prec* / 2. Thus, this example should find the minimum to at least 50000 digits.

This computes a symbolic solution for the position of the minimum which the search approaches.

```
In[15]:= exact = {x, y} /. Last[Solve[{x^2 + y^2 == 3 Pi / 2, x^2 - 3 y == -Pi}, {x, y}]]
```

$$\text{Out[15]=}\ \left\{\sqrt{-\frac{9}{2}-\pi+\frac{3}{2}\sqrt{9+10\,\pi}},\ \frac{1}{2}\left(-3+\sqrt{9+10\,\pi}\right)\right\}$$

This computes the norm of the distance from the search points at the end of each step to the exact minimum.

```
In[16]:= N[Map[Norm[exact - #] &, points]]
```

Out[16]= $\{0.140411, 0.0156607, 0.000236558, 6.09444\times10^{-8}, 3.8255\times10^{-15}, 1.59653\times10^{-29}, 3.24619\times10^{-58}, 4.8604\times10^{-108}, 1.26122\times10^{-212}, 5.865676867279906\times10^{-406}, 1.755647053247051\times10^{-791}, 4.345222958143836\times10^{-1581}, 1.099183429735576\times10^{-3141}, 1.614858677992596\times10^{-6262}, 5.998002325828813\times10^{-12514}, 1.543301971989607\times10^{-25010}, 1.131416408748486\times10^{-50010}\}$

The reason that more function evaluations were required than the number of steps is that *Mathematica* adaptively increases the precision from the precision of the initial value to the requested maximum `WorkingPrecision`. The sequence of precisions used is chosen so that as few computations are done at the most expensive final precision as possible under the assumption that the points are converging to the minimum. Sometimes when *Mathematica* changes precision, it is necessary to reevaluate the function at the higher precision.

This shows a table with the precision of each of the points with the norm of their errors.

```
In[17]:= TableForm[Transpose[{Map[Precision, points], N[Map[Norm[exact - #] &, points]]}]]
```

Out[17]//TableForm=

MachinePrecision	0.140411
MachinePrecision	0.0156607
MachinePrecision	0.000236558
MachinePrecision	6.09444×10^{-8}
24.4141	3.8255×10^{-15}
48.8283	1.59653×10^{-29}
97.6565	3.24619×10^{-58}
195.313	4.8604×10^{-108}
390.626	1.26122×10^{-212}
781.252	$5.865676867279906\times10^{-406}$
1562.5	$1.755647053247051\times10^{-791}$
3125.01	$4.345222958143836\times10^{-1581}$
6250.02	$1.099183429735576\times10^{-3141}$
12 500.	$1.614858677992596\times10^{-6262}$
25 000.1	$5.998002325828813\times10^{-12514}$
50 000.2	$1.543301971989607\times10^{-25010}$
100 000.	$1.131416408748486\times10^{-50010}$

Note that typically the precision is roughly double the scale $(\log_{10})$ of the error. For Newton's method this is appropriate since when the step is computed, the scale of the error will effectively double according to the quadratic convergence.

`FindMinimum` always starts with the precision of the starting values you gave it. Thus, if you do not want it to use adaptive precision control, you can start with values, which are exact or have at least the maximum `WorkingPrecision`.

This computes the solution using only precision 100000 throughout the computation. (Warning: this takes a very long time to complete.)

```
In[18]:= First[Timing[Block[{e = 0, s = 0}, {{min, minpoint}, {points}} =
              Reap[FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
                 {{x, 1}, {y, 1}}, Method -> "Newton", WorkingPrecision → 100 000,
                 StepMonitor :→ (s++; Sow[{x, y}]), EvaluationMonitor :→ e++]];
            Print[s, " steps and ", e, " evaluations"]]]]

           17 steps and 18 evaluations

Out[18]= 1259.84 Second
```

Even though this may use fewer function evaluations, they are all done at the highest precision, so typically adaptive precision saves a lot of time. For example, the previous command without adaptive precision takes more than 50 times as long as when starting from machine precision.

With Newton’s method, both "line search" and "trust region" step control are implemented. The default, which is used in the preceding examples, is the line search. However, any of them may be done with the trust region approach. The approach typically requires more numerical linear algebra computations per step, but because steps are better controlled, may converge in fewer iterations.

This uses the unconstrained problems package to set up the classic Rosenbrock function, which has a narrow curved valley.

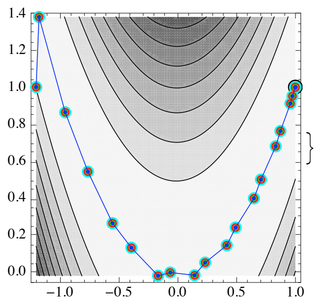
```
In[19]:= p = GetFindMinimumProblem[Rosenbrock]
```

Out[19]= FindMinimumProblem$\left[(1 - X_1)^2 + 100\left(-X_1^2 + X_2\right)^2, \{\{X_1, -1.2\}, \{X_2, 1.\}\}, \{\}, \text{Rosenbrock}, \{2, 2\}\right]$

This shows the steps taken by `FindMinimum` with a trust region Newton method for a Rosenbrock function.

```
In[20]:= FindMinimumPlot[p, Method → {"Newton", "StepControl" -> "TrustRegion"}]
```

```
Out[20]= {{2.14681×10^-26, {X₁ → 1., X₂ → 1.}},
 {Steps → 21, Function → 22, Gradient → 22, Hessian → 22}, 
```

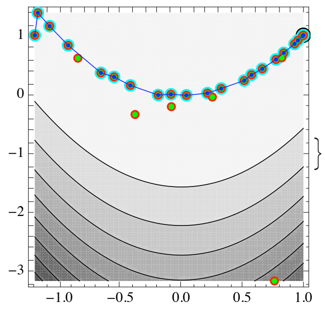


```
}
```

This shows the steps taken by `FindMinimum` with a line search Newton method for the same function.

```
In[21]:= FindMinimumPlot[p, Method → "Newton"]
```

```
Out[21]= {{4.96962×10^-18, {X₁ → 1., X₂ → 1.}},
 {Steps → 22, Function → 29, Gradient → 29, Hessian → 23}, 
```



```
}
```

You can see from the comparison of the two plots that the trust region method has kept the steps within better control as the search follows the valley and consequently converges with fewer function evaluations.

The following table summarizes the options you can use with Newton's method.

option name	*default value*	
`"Hessian"`	`Automatic`	an expression to use for computing the Hessian matrix
`"StepControl"`	`"LineSearch"`	how to control steps; options include `"LineSearch"`, `"TrustRegion"`, or `None`

Method options for `Method -> "Newton"`.

Quasi-Newton Methods

There are many variants of quasi-Newton methods. In all of them, the idea is to base the matrix B_k in the quadratic model

$$q_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

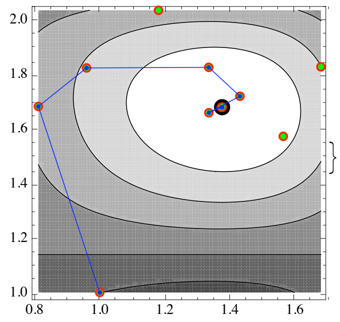
on an approximation of the Hessian matrix built up from the function and gradient values from some or all steps previously taken.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows a plot of the steps taken by the quasi-Newton method. The path is much less direct than for Newton's method. The quasi-Newton method is used by default by `FindMinimum` for problems that are not sums of squares.

```
In[2]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1}, {y, 1}}]

Out[2]= {{-2., {x → 1.37638, y → 1.67868}},

          {Steps → 9, Function → 13, Gradient → 13},                    }
```



The first thing to notice about the path taken in this example is that it starts in the wrong direction. This direction is chosen because at the first step all the method has to go by is the gradient, and so it takes the direction of steepest descent. However, in subsequent steps, it incorporates information from the values of the function and gradient at the steps taken to build up an approximate model of the Hessian.

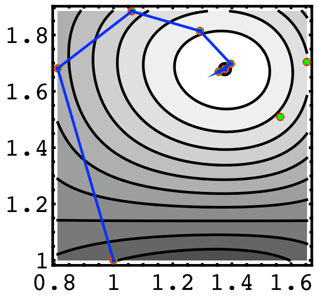
The methods used by *Mathematica* are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) updates and, for large systems, the limited-memory BFGS (L-BFGS) methods, in which the model B_k is not stored explicitly, but rather $B_k^{-1} \nabla f(x_k)$ is calculated by gradients and step directions stored from past steps.

The BFGS method is implemented such that instead of forming the model Hessian B_k at each step, Cholesky factors L_k such that $L_k.L_k^T = B_k$ are computed so that only $O(n^2)$ operations are needed to solve the system $B_k s_k = -\nabla f(x_k)$ [DS96] for a problem with n variables.

For large-scale sparse problems, the BFGS method can be problematic because, in general, the Cholesky factors (or the Hessian approximation B_k or its inverse) are dense, so the $O(n^2)$ memory and operations requirements become prohibitive compared to algorithms that take advantage of sparseness. The L-BFGS algorithm [NW99] forms an approximation to the inverse Hessian based on the last m past steps, which are stored. The Hessian approximation may not be as complete, but the memory and order of operations are limited to $O(n\,m)$ for a problem with n variables. In *Mathematica* 5, for problems over 250 variables, the algorithm is switched automatically to L-BFGS. You can control this with the method option "StepMemory" $->$ m. With $m = \infty$, the full BFGS method will always be used. Choosing an appropriate value of m is a tradeoff between speed of convergence and the work done per step. With $m < 3$, you are most likely better off using a "conjugate gradient" algorithm.

This shows the same example function with the minimum computed using L-BFGS with $m = 5$.

```
In[3]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
         {{x, 1}, {y, 1}}, Method → {"QuasiNewton", "StepMemory" → 5}]
```



```
Out[3]= {{-2., {x → 1.37638, y → 1.67868}}, {Steps → 10, Function → 13, Gradient → 13}, - ContourGraphics -}
```

Quasi-Newton methods are chosen as the default in *Mathematica* because they are typically quite fast and do not require computation of the Hessian matrix, which can be quite expensive both in terms of the symbolic computation and numerical evaluation. With an adequate "line search", they can be shown to converge superlinearly [NW99] to a local minimum where the Hessian is positive definite. This means that

$$\lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} = 0$$

or, in other words, the steps keep getting smaller. However, for very high precision, this does not compare to the q-quadratic convergence rate of "Newton's" method.

This shows the number of steps and function evaluations required to find the minimum to high precision for the problem shown.

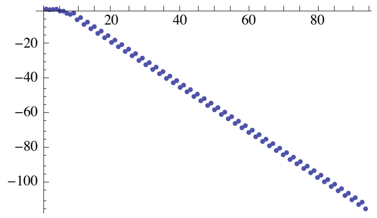
```
In[4]:= First[Timing[Block[{e = 0, s = 0}, {{min, minpoint}, {points}}  =
             Reap[FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1.}, {y, 1.}},
               Method -> "QuasiNewton", WorkingPrecision → 10 000,
               StepMonitor :→ (s ++; Sow[{x, y}]), EvaluationMonitor :→ e ++]];
           Print[s, " steps and ", e, " evaluations"]]]]

           95 steps and 106 evaluations

Out[4]= 2.79623
```

Newton's method is able to find ten times as many digits with far fewer steps because of its quadratic convergence rate. However, the convergence with the quasi-Newton method is still superlinear since the ratio of the errors is clearly going to zero.

This makes a plot showing the ratios of the errors in the computation. The ratios of the errors are shown on a logarithmic scale so that the trend can clearly be seen over a large range of magnitudes.

```
In[5]:= exact = {x, y} /. Last[Solve[{x^2 + y^2 == 3 Pi / 2, x^2 - 3 y == -Pi}, {x, y}]];
        errs = Map[Norm[N[exact - #]] &, points];
        ListPlot[Log[10, Drop[errs, 1] / Drop[errs, -1]]]
```

Out[5]=



The following table summarizes the options you can use with quasi-Newton methods.

option name	*default value*	
`"StepMemory"`	`Automatic`	the effective number of steps to "remember" in the Hessian approximation; can be a positive integer or `Automatic`
`"StepControl"`	`"LineSearch"`	how to control steps; can be `"LineSearch"` or `None`

Method options for `Method -> "QuasiNewton"`.

Gauss-Newton Methods

For minimization problems for which the objective function is a sum of squares,

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j(x)^2 = \frac{1}{2} r(x).r(x),$$

it is often advantageous to use the special structure of the problem. Time and effort can be saved by computing the residual function $r(x)$, and its derivative, the Jacobian $J(x)$. The Gauss-Newton method is an elegant way to do this. Rather than using the complete second-order Hessian matrix for the quadratic model, the Gauss-Newton method uses $B_k = J_k^T J_k$ in (1) such that the step p_k is computed from the formula

$$J_k^T J_k p_k = -\nabla f_k = -J_k^T r_k,$$

where $J_k = J(x_k)$, and so on. Note that this is an approximation to the full Hessian, which is $J^T J + \sum_{j=1}^{m} r_j \nabla^2 r_j$. In the zero residual case, where $r = 0$ is the minimum, or when r varies nearly as a linear function near the minimum point, the approximation to the Hessian is quite good and the quadratic convergence of "Newton's method" is commonly observed.

Objective functions, which are sums of squares, are quite common, and, in fact, this is the form of the objective function when `FindFit` is used with the default value of the `NormFunction` option. One way to view the Gauss-Newton method is in terms of least-squares problems. Solving the Gauss-Newton step is the same as solving a linear least-squares problem, so applying a Gauss-Newton method is in effect applying a sequence of linear least-squares fits to a nonlinear function. With this view, it makes sense that this method is particularly appropriate for the sort of nonlinear fitting that `FindFit` does.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This uses the Unconstrained Problems Package to set up the classic Rosenbrock function, which has a narrow curved valley.

```
In[2]:= p = GetFindMinimumProblem[Rosenbrock]
```

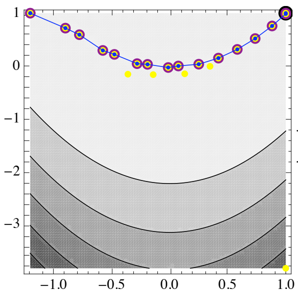
```
Out[2]= FindMinimumProblem[(1 - X1)^2 + 100 (-X1^2 + X2)^2, {{X1, -1.2}, {X2, 1.}}, {}, Rosenbrock, {2, 2}]
```

When *Mathematica* encounters a problem that is expressly a sum of squares, such as the Rosenbrock example, or a function that is the dot product of a vector with itself, the Gauss-Newton method will be used automatically.

This shows the steps taken by `FindMinimum` with the Gauss-Newton method for Rosenbrock's function using a trust region method for step control.

```
In[3]:= FindMinimumPlot[p, Method → Automatic]
```

```
Out[3]= {{0., {x1 → 1., x2 → 1.}}, {Steps → 15, Residual → 21, Jacobian → 16}, 
```



```
}
```

If you compare this with the same example done with "Newton's method", you can see that it was done with fewer steps and evaluations because the Gauss-Newton method is taking advantage of the special structure of the problem. The convergence rate near the minimum is just as good as for Newton's method because the residual is zero at the minimum.

The Levenberg-Marquardt method is a Gauss-Newton method with "trust region" step control (though it was originally proposed before the general notion of trust regions had been developed). You can request this method specifically by using the `FindMinimum` option `Method -> "LevenbergMarquardt"` or equivalently `Method -> "GaussNewton"`.

Sometimes it is awkward to express a function so that it will explicitly be a sum of squares or a dot product of a vector with itself. In these cases, it is possible to use the `"Residual"` method option to specify the residual directly. Similarly, you can specify the derivative of the residual with the `"Jacobian"` method option. Note that when the residual is specified through the `"Residual"` method option, it is not checked for consistency with the first argument of `FindMinimum`. The values returned will depend on the value given through the option.

This finds the minimum of Rosenbrock's function using the specification of the residual.

In[4]:= $\texttt{FindMinimum}\left[\frac{1}{2}\left((1-x_1)^2+100\left(-x_1^2+x_2\right)^2\right), \{\{x_1, -1.2`\}, \{x_2, 1.`\}\},\right.$
$\left.\texttt{Method} \to \left\{\text{"LevenbergMarquardt"}, \text{"Residual"} \to \left\{1-x_1, 10\left(-x_1^2+x_2\right)\right\}\right\}\right]$

Out[4]= $\{0., \{x_1 \to 1., x_2 \to 1.\}\}$

option name	*default value*	
`"Residual"`	`Automatic`	allows you to directly specify the residual r such that $f = 1/2\, r.r$
`"EvaluationMonitor"`	`Automatic`	an expression that is evaluated each time the residual is evaluated
`"Jacobian"`	`Automatic`	allows you to specify the (matrix) derivative of the residual
`"StepControl"`	`"TrustRegion"`	must be `"TrustRegion"`, but allows you to change control parameters through method options

Method options for `Method -> "LevenbergMarquardt"`.

Another natural way of setting up sums of squares problems in *Mathematica* is with `FindFit`, which computes nonlinear fits to data. A simple example follows.

Here is a model function.

```
In[5]:= fm[a_, b_, c_, x_] := a If[x > 0, Cos[b x], Exp[c x]]
```

Here is some data generated by the function with some random perturbations added.

```
In[6]:= Block[{ϵ = 0.1, a = 1.2, b = 3.4, c = 0.98},
          data = Table[{x, fm[a, b, c, x] + ϵ RandomReal[ {-.5, .5}]}, {x, -5, 5, .1}]];
```

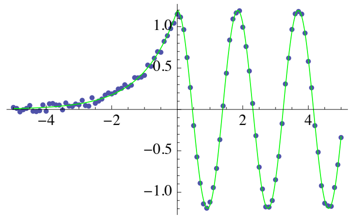
This finds a nonlinear least-squares fit to the model function.

```
In[7]:= fit = FindFit[data, fm[a, b, c, x], {{a, 1}, {b, 3}, {c, 1}}, x]
Out[7]= {a → 1.20826, b → 3.40018, c → 1.0048}
```

This shows the fit model with the data.

```
In[8]:= Show[{ListPlot[data],
          Plot[fm[a, b, c, x] /. fit, {x, -5, 5}, PlotStyle → RGBColor[0, 1, 0]]}]
```

Out[8]=



In the example, `FindFit` internally constructs a residual function and Jacobian, which are in turn used by the Gauss-Newton method to find the minimum of the sum of squares, or the nonlinear least-squares fit. Of course, `FindFit` can be used with other methods, but because a residual function that evaluates rapidly can be constructed, it is often faster than the other methods.

Nonlinear Conjugate Gradient Methods

The basis for a nonlinear conjugate gradient method is to effectively apply the linear conjugate gradient method, where the residual is replaced by the gradient. A model quadratic function is never explicitly formed, so it is always combined with a "line search" method.

The first nonlinear conjugate gradient method was proposed by Fletcher and Reeves as follows. Given a step direction p_k, use the line search to find α_k such that $x_{k+1} = x_k + \alpha_k p_k$. Then compute

$$\beta_{k+1} = \frac{\nabla f(x_{k+1}).\nabla f(x_{k+1})}{\nabla f(x_k).\nabla f(x_k)} \tag{1}$$

$$p_{k+1} = \beta_{k+1}\, p_k - \nabla f(x_{k+1}).$$

It is essential that the line search for choosing α_k satisfies the strong Wolfe conditions; this is necessary to ensure that the directions p_k are descent directions [NW99]].

An alternate method, which generally (but not always) works better in practice, is that of Polak and Ribiere, where equation (2) is replaced with

$$\beta_{k+1} = \frac{\nabla f(x_{k+1}).(\,\nabla f(x_{k+1}) - \nabla f(x_k))}{\nabla f(x_k).\nabla f(x_k)}. \tag{2}$$

In formula (3), it is possible that β_{k+1} can become negative, in which case *Mathematica* uses the algorithm modified by using $p_{k+1} = \max(\beta_{k+1},\ 0)\, p_k - \nabla f(x_{k+1})$. In *Mathematica*, the default conjugate gradient method is Polak-Ribiere, but the Fletcher-Reeves method can be chosen by using the method option

`Method → {"ConjugateGradient", Method -> "FletcherReeves"}.`

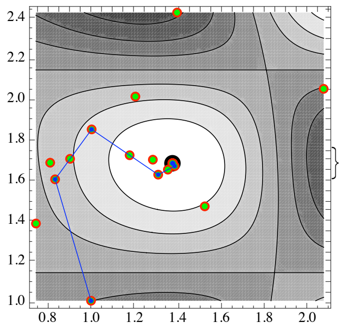
The advantage of conjugate gradient methods is that they use relatively little memory for large-scale problems and require no numerical linear algebra, so each step is quite fast. The disadvantage is that they typically converge much more slowly than "Newton" or "quasi-Newton" methods. Also, steps are typically poorly scaled for length, so the "line search" algorithm may require more iterations each time to find an acceptable step.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows a plot of the steps taken by the nonlinear conjugate gradient method. The path is much less direct than for Newton's method.

```
In[2]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
         {{x, 1}, {y, 1}}, Method -> "ConjugateGradient"]

Out[2]= {{-2., {x → 1.37638, y → 1.67868}},

        {Steps → 9, Function → 22, Gradient → 22},
```



```
}
```

One issue that arises with nonlinear conjugate gradient methods is when to restart them. As the search moves, the nature of the local quadratic approximation to the function may change substantially. The local convergence of the method depends on that of the linear conjugate gradient method, where the quadratic function is constant. With a constant quadratic function for n variables and an exact line search, the linear algorithm will converge in n or fewer iterations. By restarting (taking a steepest descent step with $\beta_{k+1} = 0$) every so often, it is possible to eliminate information from previous points, which may not be relevant to the local quadratic model at the current search point. If you look carefully at the example, you can see where the method was restarted and a steepest descent step was taken. One option is to simply restart after every k iterations, where $k <= n$. You can specify this using the method option `"RestartIterations"` $->k$. An alternative is to restart when consecutive gradients are not sufficiently orthogonal according to the test

$$\frac{|\nabla f(x_k).\nabla f(x_{k-1})|}{\nabla f(x_k).\nabla f(x_k)} < \nu,$$

with a threshold ν between 0 and 1. You can specify this using the method option `"RestartThreshold"` $->\nu$.

The table summarizes the options you can use with the conjugate gradient methods.

option name	*default value*	
`"Method"`	`"PolakRibiere"`	nonlinear conjugate gradient method can be `"PolakRibiere"` or `"FletcherReeves"`
`"RestartThreshold"`	`1/10`	threshold ν for gradient orthogonality below which a restart will be done
`"RestartIterations"`	`∞`	number of iterations after which to restart
`"StepControl"`	`"LineSearch"`	must be `"LineSearch"`, but you can use this to specify line search methods

Method options for `Method -> "ConjugateGradient"`.

It should be noted that the default method for `FindMinimum` in *Mathematica* 4 was a conjugate gradient method with a near exact line search. This has been maintained for legacy reasons and can be accessed by using the `FindMinimum` option `Method -> "Gradient"`. Typically, this will use more function and gradient evaluations than the newer `Method -> "ConjugateGradient"`, which itself often uses far more than the methods that *Mathematica* currently uses as defaults.

Principal Axis Method

"Gauss-Newton" and "conjugate gradient" methods use derivatives. When *Mathematica* cannot compute symbolic derivatives, finite differences will be used. Computing derivatives with finite differences can impose a significant cost in some cases and certainly affects the reliability of derivatives, ultimately having an effect on how good an approximation to the minimum is achievable. For functions where symbolic derivatives are not available, an alternative is to use a derivative-free algorithm, where an approximate model is built up using only values from function evaluations.

Mathematica uses the principal axis method of Brent [Br02] as a derivative-free algorithm. For an n-variable problem, take a set of search directions $u_1, u_2, \ldots, u_n$ and a point x_0. Take x_i to be the point that minimizes f along the direction u_i from x_{i-1} (i.e., do a "line search" from x_{i-1}), then replace u_i with u_{i+1}. At the end, replace u_n with $x_n - x_0$. Ideally, the new u_i should be linearly independent, so that a new iteration could be undertaken, but in practice, they are not. Brent's algorithm involves using the singular value decomposition (SVD) on the matrix $U = (u_1, u_2, \ldots u_n)$

to realign them to the principal directions for the local quadratic model. (An eigen decomposition could be used, but Brent shows that the SVD is more efficient.) With the new set of u_i obtained, another iteration can be done.

Two distinct starting conditions in each variable are required for this method because these are used to define the magnitudes of the vectors u_i. In fact, whenever you specify two starting conditions in each variable, `FindMinimum`, `FindMaximum`, and `FindFit` will use the principal axis algorithm by default.

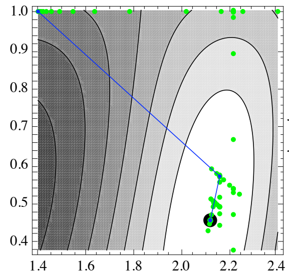
This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows the search path and function evaluations for `FindMinimum` to find a local minimum of the function $\cos(x^2 - 3y) + \sin(x^2 + y^2)$.

```
In[2]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
          {{x, 1.4, 1.5}, {y, 1, 1.1}}, Method → "PrincipalAxis"]
```

```
Out[2]= {{-2., {x → 2.12265, y → 0.454686}}, {Steps → 4, Function → 148}, 
```



```
}
```

The basics of the search algorithm can be seen quite well from the plot since the derivative-free line search algorithm requires a substantial number of function evaluations. First a line search is done in the x direction, then from that point, a line search is done in the y direction, determining the step direction. Once the step is taken, the vectors u_i are realigned appropriately to the principal directions of the local quadratic approximation and the next step is similarly computed.

The algorithm is efficient in terms of convergence rate; it has quadratic convergence in terms of steps. However, in terms of function evaluations, it is quite expensive because of the derivative-free line search required. Note that since the directions given to the line search (especially at the beginning) are not necessarily descent directions, the line search has to be able to search in both directions. For problems with many variables, the individual linear searches in all directions become very expensive, so this method is typically better suited to problems without too many variables.

Methods for Solving Nonlinear Equations

Introduction to Solving Nonlinear Equations

There are some close connections between finding a "local minimum" and solving a set of nonlinear equations. Given a set of n equations in n unknowns, seeking a solution $r(x) == 0$ is equivalent to minimizing the sum of squares $r(x).r(x)$ when the residual is zero at the minimum, so there is a particularly close connection to the "Gauss-Newton" methods. In fact, the Gauss-Newton step for local minimization and the "Newton" step for nonlinear equations are exactly the same. Also, for a smooth function, "Newton's method" for local minimization is the same as Newton's method for the nonlinear equations $\nabla f = 0$. Not surprisingly, many aspects of the algorithms are similar; however, there are also important differences.

Another thing in common with minimization algorithms is the need for some kind of "step control". Typically, step control is based on the same methods as minimization except that it is applied to a merit function, usually the smooth 2-norm squared, $r(x).r(x)$.

`"Newton"`	use the exact Jacobian or a finite difference approximation to solve for the step based on a locally linear model
`"Secant"`	work without derivatives by constructing a secant approximation to the Jacobian using n past steps; requires two starting conditions in each dimension
`"Brent"`	method in one dimension that maintains bracketing of roots; requires two starting conditions that bracket a root

Basic method choices for `FindRoot`.

Newton's Method

Newton's method for nonlinear equations is based on a linear approximation

$$r(x) = M_k(p) = r(x_k) + J(x_k)\, p,\ p = (x - x_k),$$

so the Newton step is found simply by setting $M_k(p) = 0$,

$$J(x_k)\, p_k = -r(x_k).$$

Near a root of the equations, Newton's method has q-quadratic convergence, similar to "Newton's" method for minimization. Newton's method is used as the default method for `FindRoot`.

Newton's method can be used with either "line search" or "trust region" step control. When it works, the line search control is typically faster, but the trust region approach is usually more robust.

This loads a package that contains some utility functions.

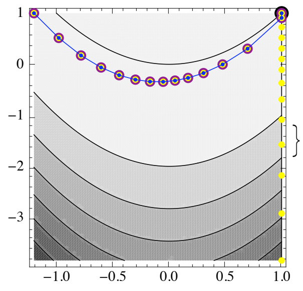
```
In[1]:= << Optimization`UnconstrainedProblems`
```

Here is the Rosenbrock problem as a `FindRoot` problem.

```
In[2]:= p = GetFindRootProblem[Rosenbrock]

Out[2]= FindRootProblem[{10 (-X1^2 + X2), 1 - X1}, {{X1, -1.2}, {X2, 1.}}, {}, Rosenbrock, {2, 2}]
```

This finds the solution of the nonlinear system using the default line search approach. (Newton's method is the default method for `FindRoot`.)

```
In[3]:= FindRootPlot[p]

Out[3]= {{X1 -> 1., X2 -> 1.}, {Steps -> 15, Residual -> 27, Jacobian -> 15},
```



```
}
```

Note that each of the line searches started along the line $x == 1$. This is a particular property of the Newton step for this particular problem.

This computes the Jacobian and the Newton step symbolically for the Rosenbrock problem.

```
In[4]:= J = Outer[D, {10 (-X1^2 + X2), 1 - X1}, {X1, X2}];
        LinearSolve[J, -{10 (-X1^2 + X2), 1 - X1}]

Out[4]= {1 - X1, 2 X1 - X1^2 - X2}
```

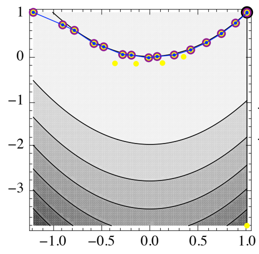
When this step is added to the point, $\{X_1, X_2\}$, it is easy to see why the steps go to the line $X_1 = 1$. This is a particular feature of this problem, which is not typical for most functions.

Because the "trust region" approach does not try the Newton step unless it lies within the region bound, this feature does not show up so strongly when the trust region step control is used.

This finds the solution of the nonlinear system using the trust region approach. The search is almost identical to the search with the "Gauss-Newton" method for the Rosenbrock objective function in `FindMinimum`.

```
In[5]:= FindRootPlot[p, Method → {"Newton", "StepControl" → "TrustRegion"}]
```

```
Out[5]= {{X1 → 1., X2 → 1.}, {Steps → 16, Residual → 21, Jacobian → 16},
```



```
}
```

When the structure of the Jacobian matrix is sparse, *Mathematica* will use `SparseArray` objects both to compute the Jacobian and to handle the necessary numerical linear algebra.

When solving nonlinear equations is used as a part of a more general numerical procedure, such as solving differential equations with implicit methods, often starting values are quite good, and complete convergence is not absolutely necessary. Often the most expensive part of computing a Newton step is finding the Jacobian and computing a matrix factorization. However, when close enough to a root, it is possible to leave the Jacobian frozen for a few steps (though this does certainly affect the convergence rate). You can do this in *Mathematica* using the method option `"UpdateJacobian"`, which gives the number of steps to go before updating the Jacobian. The default is `"UpdateJacobian" -> 1`, so the Jacobian is updated every step.

This shows the number of steps, function evaluations, and Jacobian evaluations required to find a simple square root when the Jacobian is only updated every three steps.

```
In[6]:= Block[{s = 0, e = 0, j = 0},
         {FindRoot[x^2 - 2, {{x, 1.5}}, Method → {"Newton", "UpdateJacobian" → 3},
           EvaluationMonitor :→ e++, StepMonitor :→ s++,
           Jacobian → {Automatic, EvaluationMonitor :→ j++}], s, e, j}]
Out[6]= {{x → 1.41421}, 5, 9, 2}
```

This shows the number of steps, function evaluations, and Jacobian evaluations required to find a simple square root when the Jacobian is updated every step.

```
In[7]:= Block[{s = 0, e = 0, j = 0},
         {FindRoot[x^2 - 2, {{x, 1.5}}, EvaluationMonitor :→ e++, StepMonitor :→ s++,
           Jacobian → {Automatic, EvaluationMonitor :→ j++}], s, e, j}]
Out[7]= {{x → 1.41421}, 4, 5, 4}
```

Of course for a simple one-dimensional root, updating the Jacobian is trivial in cost, so holding the update is only of use here to demonstrate the idea.

option name	*default value*	
`"UpdateJacobian"`	1	number of steps to take before updating the Jacobian
`"StepControl"`	`"LineSearch"`	method for step control, can be `"LineSearch"`, `"TrustRegion"`, or `None` (which is not recommended)

Method options for `Method -> "Newton"` in `FindRoot`.

The Secant Method

When derivatives cannot be computed symbolically, "Newton's" method will be used, but with a finite difference approximation to the Jacobian. This can have cost in terms of both time and reliability. Just as for minimization, an alternative is to use an algorithm specifically designed to work without derivatives.

In one dimension, the idea of the secant method is to use the slope of the line between two consecutive search points to compute the step instead of the derivative at the latest point. Similarly in n dimensions, differences between the residuals at n points are used to construct an approximation of sorts to the Jacobian. Note that this is similar to finite differences, but rather than trying to make the difference interval small in order to get as good a Jacobian approximation as possible, it effectively uses an average derivative just like the one-dimensional secant method. Initially, the n points are constructed from two starting points that are distinct in all n dimensions. Subsequently, as steps are taken, only the n points with the smallest merit function value are kept. It is rare, but possible, that steps are collinear and the secant approximation to the Jacobian becomes singular. In this case, the algorithm is restarted with distinct points.

The method requires two starting points in each dimension. In fact, if two starting points are given in each dimension, the secant method is the default method except in one dimension, where "Brent's" method may be chosen.

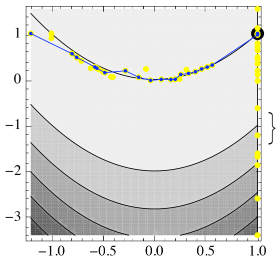
This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows the solution of the Rosenbrock problem with the secant method.

```
In[2]:= FindRootPlot[{10 (-x1^2 + x2), 1 - x1}, {{x1, -1.2, -1.}, {x2, 1., .9}}]
```

```
Out[2]= {{x1 → 1., x2 → 1.}, {Steps → 21, Residual → 70}, 
```



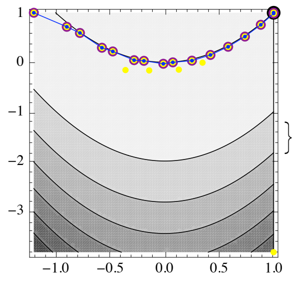
```
}
```

Note that, as compared to "Newton's" method, many more residual function evaluations are required. However, the method is able to follow the relatively narrow valley without directly using derivative information.

This shows the solution of the Rosenbrock problem with Newton's method using finite differences to compute the Jacobian.

```
In[3]:= FindRootPlot[{10 (-x1^2 + x2), 1 - x1}, {{x1, -1.2}, {x2, 1.}},
         Method → {"Newton", StepControl -> "TrustRegion"}, Jacobian -> "FiniteDifference"]
```

```
Out[3]= {{x1 → 1., x2 → 1.}, {Steps → 17, Residual → 70, Jacobian → 16}, 
```



```
}
```

However, when compared to Newton's method with finite differences, the number of residual function evaluations is comparable. For sparse Jacobian matrices with larger problems, the finite difference Newton method will usually be more efficient since the secant method does not take advantage of sparsity in any way.

Brent's Method

When searching for a real simple root of a real valued function, it is possible to take advantage of the special geometry of the problem, where the function crosses the axis from negative to positive or vice versa. Brent's method [Br02] is effectively a safeguarded secant method that always keeps a point where the function is positive and one where it is negative so that the root is always bracketed. At any given step, a choice is made between an interpolated (secant) step and a bisection in such a way that eventual convergence is guaranteed.

If `FindRoot` is given two real starting conditions that bracket a root of a real function, then Brent's method will be used. Thus, if you are working in one dimension and can determine initial conditions that will bracket a root, it is often a good idea to do so since Brent's method is the most robust algorithm available for `FindRoot`.

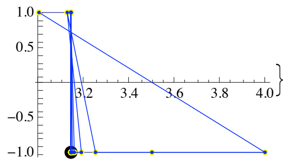
Even though essentially all the theory for solving nonlinear equations and local minimization is based on smooth functions, Brent's method is sufficiently robust that you can even get a good estimate for a zero crossing for discontinuous functions.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows the steps and function evaluations used in an attempt to find the root of a discontinuous function.

```
In[2]:= FindRootPlot[2 UnitStep[Sin[x]] - 1, {x, 3, 4}]

        FindRoot::cvmit : Failed to converge to the requested accuracy or precision within 100 iterations. >>

Out[2]= {{x → 3.14159}, {Steps → 50, Residual → 51},
```

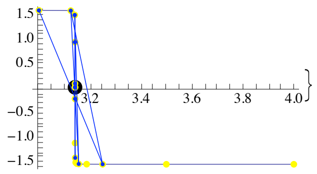


```
}
```

The method gives up and issues a message when the root is bracketed very closely, but it is not able to find a value of the function, which is zero. This robustness carries over very well to continuous functions that are very steep.

This shows the steps and function evaluations used to find the root of a function that varies rapidly near its root.

```
In[3]:= FindRootPlot[ArcTan[10 000 Sin[x] ], {x, 3, 4}, PlotRange → All]

Out[3]= {{x → 3.14159}, {Steps → 18, Residual → 19},
```



```
}
```

Step Control

Introduction to Step Control

Even with "Newton methods" where the local model is based on the actual Hessian, unless you are close to a root or minimum, the model step may not bring you any closer to the solution. A simple example is given by the following problem.

This loads a package that contains some utility functions.

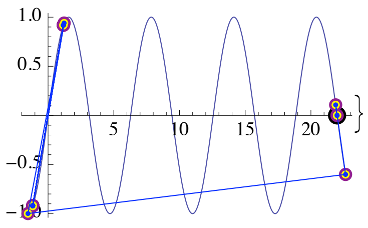
```
In[1]:= << Optimization`UnconstrainedProblems`
```

This shows a simple example for root finding with step control disabled where the iteration alternates between two points and does not converge. **Note:** On some platforms, you may see convergence. This is due to slight variations in machine-number arithmetic, which may be sufficient to break the oscillation.

```
In[2]:= FindRootPlot[Sin[x], {x, 1.1655611852072114},
         Method → {Newton,  StepControl → None}]
```

```
FindRoot::cvmit: Failed to converge to the requested accuracy or precision within 100 iterations. ≫
```

```
Out[2]= {{x → 21.9911}, {Steps → 27, Residual → 27, Jacobian → 26},
```

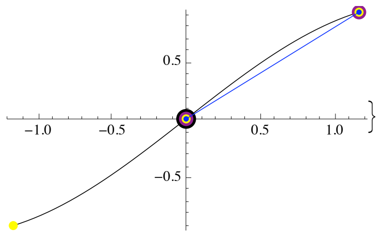


}

This shows the same example problem with step control enabled. Since the first evaluation point has not reduced the size of the function, the line search restricts the step and so the iteration converges to the solution.

```
In[3]:= FindRootPlot[Sin[x], {x, 1.1655611852072114}, Method → "Newton"]
```

```
Out[3]= {{x → 0.}, {Steps → 2, Residual → 3, Jacobian → 2},
```



}

A good step-size control algorithm will prevent repetition or escape from areas near roots or minima from happening. At the same time, however, when steps based on the model function are appropriate, the step-size control algorithm should not restrict them, otherwise the convergence rate of the algorithm would be compromised. Two commonly used step-size control algorithms are "line search" and "trust region" methods. In a line search method, the model function gives a step direction, and a search is done along that direction to find an adequate point that will lead to convergence. In a trust region method, a distance in which the model function will be trusted is updated at each step. If the model step lies within that distance, it is used; otherwise, an approximate minimum for the model function on the boundary of the trust region is used. Generally the trust region methods are more robust, but they require more numerical linear algebra.

Both step control methods were developed originally with minimization in mind. However, they apply well to finding roots for nonlinear equations when used with a merit function. In *Mathematica*, the 2-norm merit function $r(x).r(x)$ is used.

Line Search Methods

A method like "Newton's" method chooses a step, but the validity of that step only goes as far as the Newton quadratic model for the function really reflects the function. The idea of a line search is to use the direction of the chosen step, but to control the length, by solving a one-dimensional problem of minimizing

$$\phi(\alpha) == f(\alpha\, p_k + x_k),$$

where p_k is the search direction chosen from the position x_k. Note that

$$\phi'(\alpha) == \nabla f(\alpha\, p_k + x_k).p_k,$$

so if you can compute the gradient, you can effectively do a one-dimensional search with derivatives.

Typically, an effective line search only looks toward $\alpha > 0$ since a reasonable method should guarantee that the search direction is a descent direction, which can be expressed as $\phi'\, \alpha < 0$.

It is typically not worth the effort to find an exact minimum of ϕ since the search direction is rarely exactly the right direction. Usually it is enough to move closer.

One condition that measures progress is called the Armijo or sufficient decrease condition for a candidate α^*.

$$\phi(\alpha^*) \leq \phi(0) + \mu\,\phi'(0),\ 0 < \mu < 1$$

Often with this condition, methods will converge, but for some methods, Armijo alone does not guarantee convergence for smooth functions. With the additional curvature condition,

$$|\phi'(\alpha^*)| \leq \eta\,|\phi'(0)|,\ 0 < \mu \leq \eta < 1,$$

many methods can be proven to converge for smooth functions. Together these conditions are known as the strong Wolfe conditions. You can control the parameters μ and η with the `"DecreaseFactor"` $->\mu$ and `"CurvatureFactor"` $->\eta$ options of `"LineSearch"`.

The default value for `"CurvatureFactor"` $->\eta$ is $\eta = 0.9$, except for `Method` $->$ `"ConjugateGradient"` where $\eta = 0.1$ is used since the algorithm typically works better with a closer-to-exact line search. The smaller η is, the closer to exact the line search is.

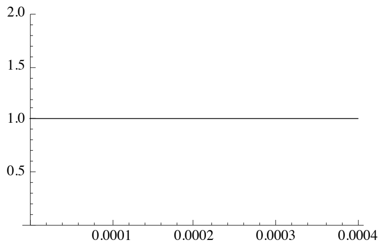
If you look at graphs showing iterative searches in two dimensions, you can see the evaluations spread out along the directions of the line searches. Typically, it only takes a few iterations to find a point satisfying the conditions. However, the line search is not always able to find a point that satisfies the conditions. Usually this is because there is insufficient precision to compute the points closely enough to satisfy the conditions, but it can also be caused by functions that are not completely smooth or vary extremely slowly in the neighborhood of a minimum.

This loads a package that contains some utility functions.

```
In[1]:= << Optimization`UnconstrainedProblems`

In[2]:= FindMinimum[x^2 / 2 + Cos[x], {x, 1}]

        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. >>

Out[2]= {1., {x → 0.000182658}}
```

This runs into problems because the real differences in the function are negligible compared to evaluation differences around the point, as can be seen from the plot.

```
In[24]:= Plot[x^2 / 2 + Cos[x], {x, 0, .0004}, PlotRange → {1 - 10^-15, 1 + 10^-15}]
```

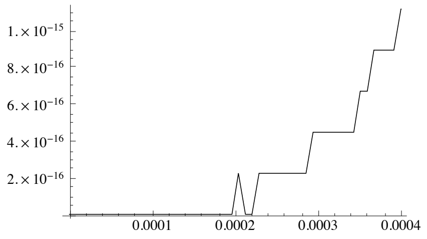
Out[24]=


Sometimes it can help to subtract out the constants so that small changes in the function are more significant.

```
In[18]:= FindMinimum[x^2 / 2 + Cos[x] - 1, {x, 1}]

        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. ≫

Out[18]= {1.11022×10^-16, {x → 0.00024197}}
```

In this case, however, the approximation is only slightly closer because the function is quite noisy near 0, as can be seen from the plot.

```
In[19]:= Plot[x^2 / 2 + Cos[x] - 1, {x, 0, .0004}]
```

Out[19]=


Thus, to get closer to the correct value of zero, higher precision is required to compute the function more accurately.

For some problems, particularly where you may be starting far from a root or a local minimum, it may be desirable to restrict steps. With line searches, it is possible to do this by using the `"MaxRelativeStepSize"` method option. The default value picked for this is designed to keep searches from going wildly out of control, yet at the same time not prevent a search from using reasonably large steps if appropriate.