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

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematic*a software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, “look and feel,” programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and web*Mathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

Contents

Introduction 1

Methods for Local Minimization

- **Introduction** 7
- **Newton's Method** 8
- **Quasi-Newton Methods** 15
- **Gauss-Newton Methods** 18
- **Nonlinear Conjugate Gradient Methods** 21
- **Principal Axis Method** 23

Methods for Solving Nonlinear Equations

- **Introduction** 25
- **Newton’s Method** 25
- **The Secant Method** 28
- **Brent’s Method** 29

Step Control

- **Introduction** 31
- **Line Search Methods** 32
- **Trust Region Methods** 39

Setting Up Optimization Problems in *Mathematica*

- **Specifying Derivatives** 44
- **Variables and Starting Conditions** 49
- **Termination Conditions** 53
- **Symbolic Evaluation** 59

UnconstrainedProblems Package

- **Plotting Search Data** 62
- **Test Problems** 65

References 71

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.

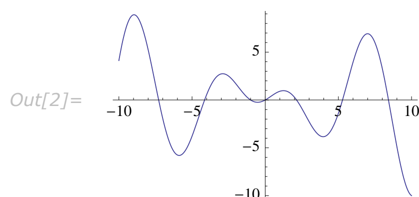
This loads a package that contains some utility functions.

```
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}]
```

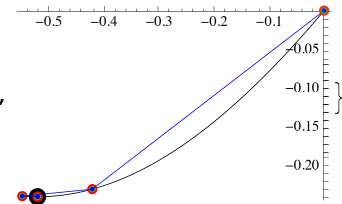
Out[2]=



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},
```



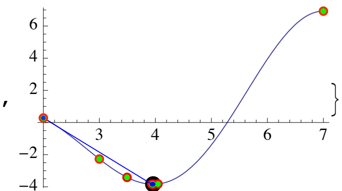
}

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},
```



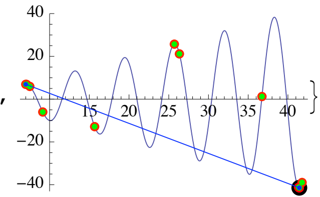
}

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},
```


`}`

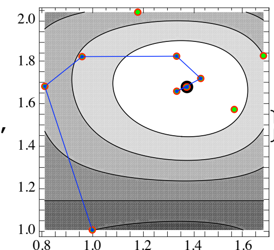
When starting at `x = 7`, which is near a local maximum, the first step is quite large, so `FindMinimum` returns a completely different local minimum.

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}}]
```

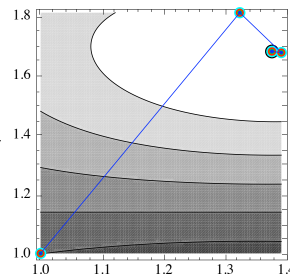
```
Out[6]= {{-2., {x → 1.37638, y → 1.67868}}, {Steps → 9, Function → 13, Gradient → 13},
```


`}`

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 kth 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}}
```

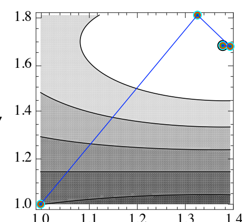
This defines a function that is only intended to evaluate for numerical values of the variables.

```
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[x^2 + y^2] - 2 x Sin[x^2 - 3 y], 2 y Cos[x^2 + y^2] + 3 Sin[x^2 - 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 x^2 Cos[x^2 - 3 y] + 2 Cos[x^2 + y^2] - 2 Sin[x^2 - 3 y] - 4 x^2 Sin[x^2 + y^2],
          6 x Cos[x^2 - 3 y] - 4 x y Sin[x^2 + y^2]},
         {6 x Cos[x^2 - 3 y] - 4 x y Sin[x^2 + y^2], -9 Cos[x^2 - 3 y] + 2 Cos[x^2 + y^2] - 4 y^2 Sin[x^2 + y^2]}}
```

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

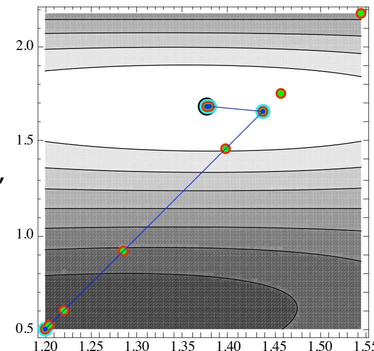
```
In[8]:= FindMinimum[f[x, y], {{x, 1}, {y, 1}},
         Gradient → g[x, y], Method → {"Newton", "Hessian" → h[x, y]}]
Out[8]= {-2., {x → 1.37638, y → 1.67868}}
```

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

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

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 → 100000,
              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}]]
```

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 → 100000,
               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.

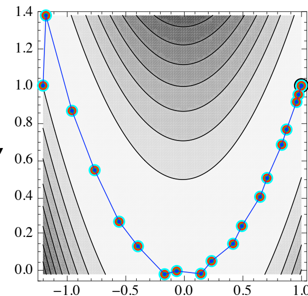
This uses the unconstrained problems package to set up the classic Rosenbrock function, which has a narrow curved valley.

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

Out[19]= FindMinimumProblem[(1 - X1)^2 + 100 (-X1^2 + X2)^2, {{X1, -1.2}, {X2, 1.}}, {}, Rosenbrock, {2, 2}]
```

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"}]
```

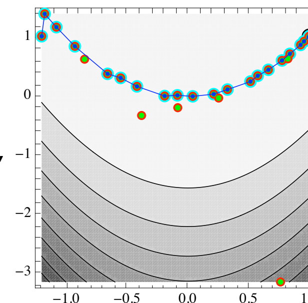
Out[20]= $\{\{2.14681\times10^{-26}, \{X_1 \to 1., X_2 \to 1.\}\},$

$\{Steps \to 21, Function \to 22, Gradient \to 22, Hessian \to 22\},$  $\}$

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]= $\{\{4.96962\times10^{-18}, \{X_1 \to 1., X_2 \to 1.\}\},$

$\{Steps \to 22, Function \to 29, Gradient \to 29, Hessian \to 23\},$  $\}$

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.

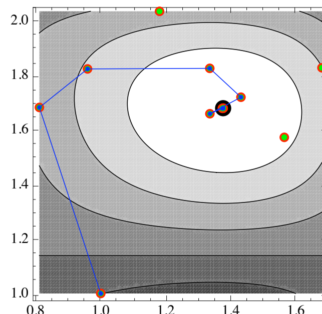
This loads a package that contains some utility functions.

```
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.

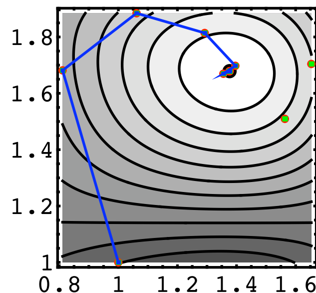
The methods used by *Mathematica* are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) updates and, for large systems, the limited-memory BFGS (L-BFGS) methods, in which the model B_k is not stored explicitly, but rather $B_k^{-1} \nabla f(x_k)$ is calculated by gradients and step directions stored from past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

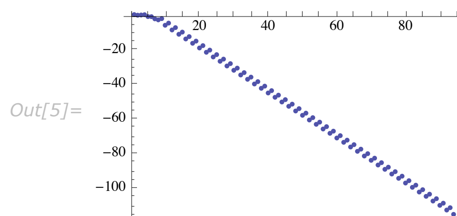
```
In[4]:= First[Timing[Block[{e = 0, s = 0}, {{min, minpoint}, {points}} =
          Reap[FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2], {{x, 1.}, {y, 1.}},
            Method -> "QuasiNewton", WorkingPrecision → 10 000,
            StepMonitor :→ (s++; Sow[{x, y}]), EvaluationMonitor :→ e++]];
    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

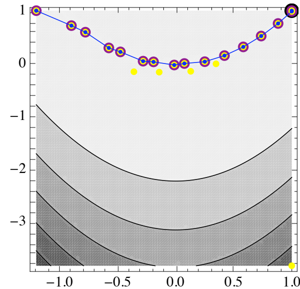
```
Out[2]= FindMinimumProblem[(1 - X1)^2 + 100 (-X1^2 + X2)^2, {{X1, -1.2}, {X2, 1.}}, {}, Rosenbrock, {2, 2}]
```

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]:= $\text{FindMinimum}\left[\frac{1}{2}\left((1-X_1)^2+100\left(-X_1^2+X_2\right)^2\right), \{\{X_1, -1.2`\}, \{X_2, 1.`\}\}, \text{Method} \to \left\{\text{"LevenbergMarquardt"}, \text{"Residual"} \to \left\{1-X_1, 10\left(-X_1^2+X_2\right)\right\}\right\}\right]$

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}]];
```

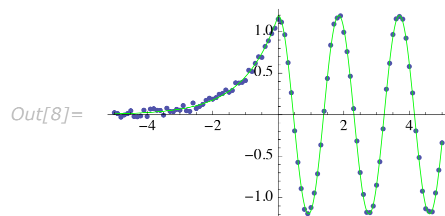
This finds a nonlinear least-squares fit to the model function.

```
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"}.
```

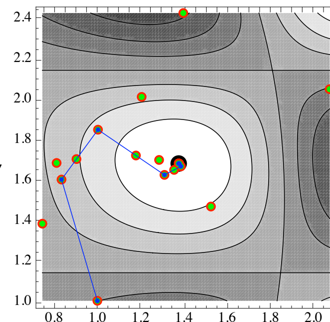
The advantage of conjugate gradient methods is that they use relatively little memory for large-scale problems and require no numerical linear algebra, so each step is quite fast. The disadvantage is that they typically converge much more slowly than "Newton" or "quasi-Newton" methods. Also, steps are 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"` $->$ ν.

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.

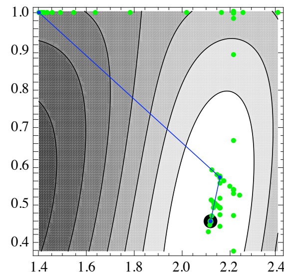
This loads a package that contains some utility functions.

```
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.

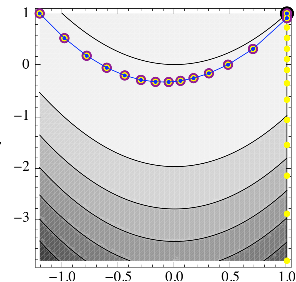
```
In[1]:= << Optimization`UnconstrainedProblems`
```

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}
```

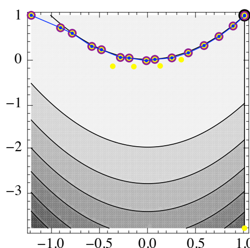
When this step is added to the point, $\{X_1, X_2\}$, it is easy to see why the steps go to the line $X_1 = 1$. This is a particular feature of this problem, which is not typical for most functions.

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.

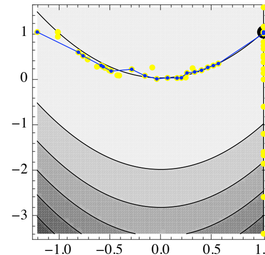
This loads a package that contains some utility functions.

```
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}, 
```



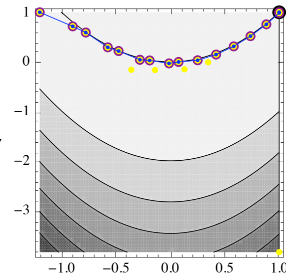
```
}
```

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.

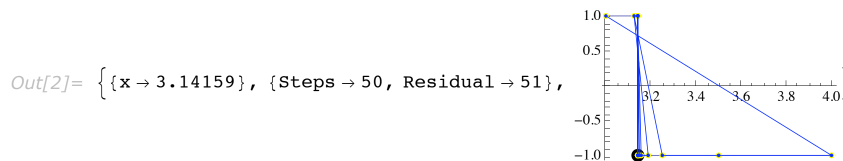
This loads a package that contains some utility functions.

```
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}]
```

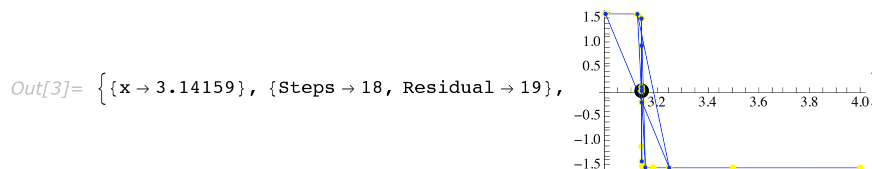
FindRoot::cvmit : Failed to converge to the requested accuracy or precision within 100 iterations. ≫



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]
```



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.

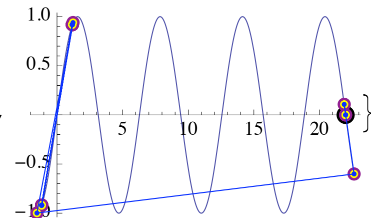
```
In[1]:= << Optimization`UnconstrainedProblems`
```

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. ≫

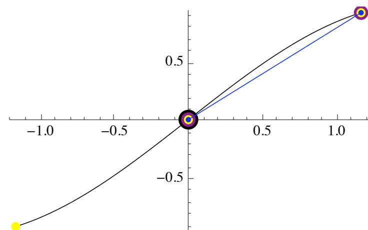
```
Out[2]= {{x → 21.9911}, {Steps → 27, Residual → 27, Jacobian → 26},
```



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" -> μ` and `"CurvatureFactor" -> η` options of `"LineSearch"`.

The default value for `"CurvatureFactor" -> η` 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.

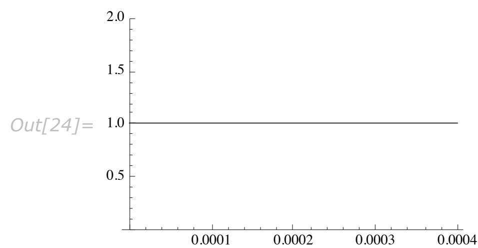
```
In[1]:= << Optimization`UnconstrainedProblems`
```

```
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}]
```

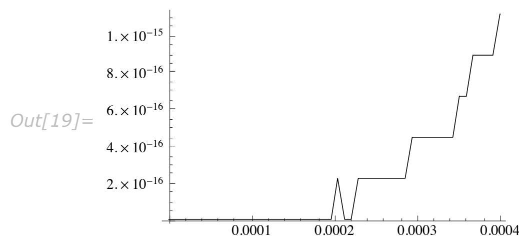
FindMinimum::lstol :
The line search decreased the step size to within tolerance specified by AccuracyGoal and PrecisionGoal but was unable to find a sufficient decrease in the function. You may need more than MachinePrecision digits of working precision to meet these tolerances. ≫

Out[18]= $\{1.11022\times 10^{-16}, \{x \to 0.00024197\}\}$

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]=

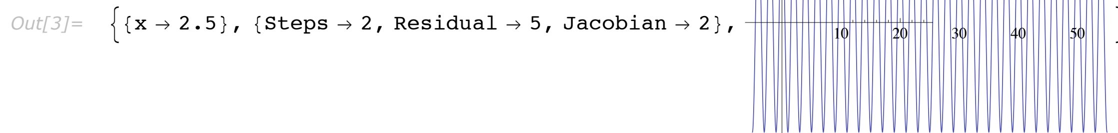


Thus, to get closer to the correct value of zero, higher precision is required to compute the function more accurately.

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.

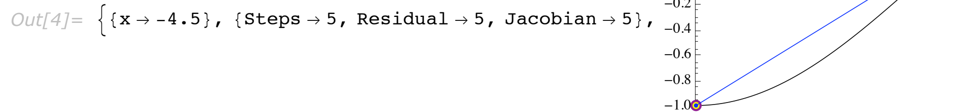
This is an example of a problem where the Newton step is very large because the starting point is at a position where the Jacobian (derivative) is nearly singular. The step size is (not severely) limited by the option.

```
In[3]:= FindRootPlot[Cos[x Pi], {{x, -5}}]
```



This shows the same example but with a more rigorous step-size limitation, which finds the root near the starting condition.

```
In[4]:= FindRootPlot[Cos[x Pi], {{x, -5}},
         Method → {"Newton", "StepControl" → {"LineSearch", "MaxRelativeStepSize" → .1}}]
```



Note that you need to be careful not to set the "MaxRelativeStepSize" option too small, or it will affect convergence, especially for minima and roots near zero.

The following table shows a summary of the options, which can be used to control line searches.

option name	*default value*	
"Method"	Automatic	method to use for executing the line search; can be Automatic, "MoreThuente", "Backtracking", or "Brent"
"CurvatureFactor"	Automatic	factor η in the Wolfe conditions, between 0 and 1; smaller values of η result in a more exact line search
"DecreaseFactor"	1/10 000	factor μ in the Wolfe conditions, between 0 and η
"MaxRelativeStepSize"	10	largest step that will be taken relative to the norm of the current search point, can be any positive number or ∞ for no restriction

Method options for "StepControl" → "LineSearch".

The following sections will describe the three line search algorithms implemented in *Mathematica*. Comparisons will be made using the Rosenbrock function.

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

```
In[5]:= p = GetFindMinimumProblem[Rosenbrock]
```

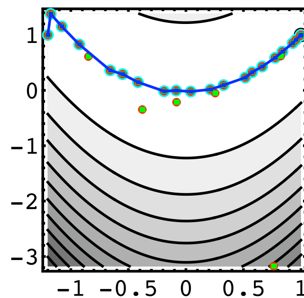
Out[5]= FindMinimumProblem$\left[(1 - X_1)^2 + 100\left(-X_1^2 + X_2\right)^2, \{\{X_1, -1.2\}, \{X_2, 1.\}\}, \{\}, \text{Rosenbrock}, \{2, 2\}\right]$

MoreThuente

The default line search used by `FindMinimum`, `FindMaximum`, and `FindFit` is one described by More and Thuente in [MT94]. It tries to find a point that satisfies both the decrease and curvature conditions by using bracketing and quadratic and cubic interpolation.

This shows the steps and evaluations done with Newton's method with the default line search parameters. Points with just red and green are where the function and gradient were evaluated in the line search, but the Wolfe conditions were not satisfied so as to take a step.

```
In[10]:= FindMinimumPlot[p, Method → Newton]
```



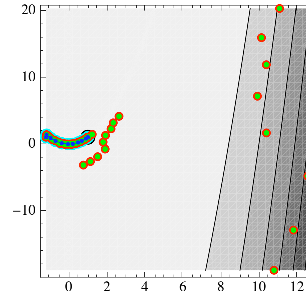
Out[10]= $\{\{4.96962\times10^{-18}, \{X_1 \to 1., X_2 \to 1.\}\},$
$\{\text{Steps} \to 22, \text{Function} \to 29, \text{Gradient} \to 29, \text{Hessian} \to 23\}$, - ContourGraphics -}

The points at which only the function and gradient were evaluated were the ones attempted in the line search phase that did not satisfy both conditions. Unless restricted by `"MaxRelativeStepSize"`, the line search always starts with the full step length ($\alpha = 1$), so that if the full (in this case Newton) step satisfies the line search criteria, it will be taken, ensuring a full convergence rate close to a minimum.

Decreasing the curvature factor, which means that the line search ends nearer to the exact minimum, decreases the number of steps taken by Newton's method but increases the total number of function and gradient evaluations.

This shows the steps and evaluations done with Newton's method with a curvature factor in the line search parameters that is smaller than the default. Points with just red and green are where the function and gradient were evaluated in the line search, but the Wolfe conditions were not satisfied so as to take a step.

```
In[31]:= FindMinimumPlot[p,
          Method → {"Newton", "StepControl" → {"LineSearch", CurvatureFactor → .1}}]

Out[31]= {{5.54946×10^-22, {X1 → 1., X2 → 1.}},

          {Steps → 14, Function → 61, Gradient → 61, Hessian → 15},
```



}

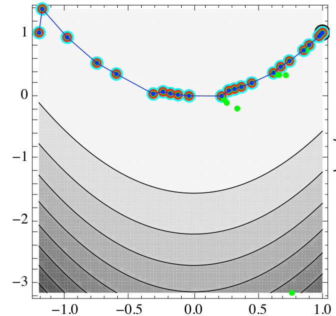
This example demonstrates why a more exact line search is not necessarily better. When the line search takes the step to the right at the bottom of the narrow valley, the Newton step is based on moving along the valley without seeing its curvature (the curvature of the valley is beyond quadratic order), so the Newton steps end up being far too long, even though the direction is better. On the other hand, some methods, such as the conjugate gradient method, need a better line search to improve convergence.

Backtracking

This is a simple line search that starts from the given step size and backtracks toward a step size of 0, stopping when the sufficient decrease condition is met. In general with only backtracking, there is no guarantee that you can satisfy the curvature condition, even for nice functions, so the convergence properties of the methods are not assured. However, the backtracking line search also does not need to evaluate the gradient at each point, so if gradient evaluations are relatively expensive, this may be a good choice. It is used as the default line search in `FindRoot` because evaluating the gradient of the merit function involves computing the Jacobian, which is relatively expensive.

```
In[32]:= FindMinimumPlot[p,
          Method → {"Newton", "StepControl" → {"LineSearch", Method → "Backtracking"}}]
```

Out[32]= $\{\{1.2326\times10^{-30}, \{X_1\to1., X_2\to1.\}\},$

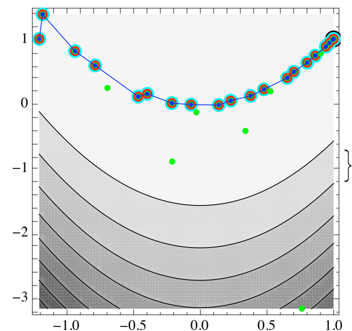
{Steps → 25, Function → 34, Gradient → 26, Hessian → 25},  }

Each backtracking step is taken by doing a polynomial interpolation and finding the minimum point for the interpolant. This point α_k is used as long as it lies between $c_1\,\alpha_{k-1}$ and $c_2\,\alpha_{k-1}$, where α_{k-1} is the previous value of the parameter α and $0 < c_1 \le c_2 < 1$. By default, $c_1 = 0.1$ and $c_2 = 0.5$, but they can be controlled by the method option "BacktrackFactors" -> $\{c_1, c_2\}$. If you give a single value for the factors, this sets $c_1 = c_2$, and no interpolation is used. The value 1/2 gives bisection.

In this example, the effect of the relatively large backtrack factor is quite apparent.

```
In[33]:= FindMinimumPlot[p, Method → {"Newton", "StepControl" →
              {"LineSearch", Method → {"Backtracking", "BacktrackFactors" → 1 / 2}}}]
```

Out[33]= $\{\{3.74398\times10^{-21}, \{X_1\to1., X_2\to1.\}\},$

{Steps → 21, Function → 29, Gradient → 22, Hessian → 22},  }

option name	*default value*	
"BacktrackFactors"	{1/10, 1/2}	determine the minimum and maximum factor by which the attempted step length must shrink between backtracking steps

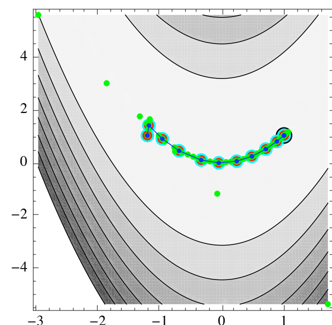
Method option for line search Method -> "Backtracking".

Brent

This uses the derivative-free univariate method of Brent [Br02] for the line search. It attempts to find the minimum of $\phi\alpha$ to within tolerances, regardless of the decrease and curvature factors. In effect, it has two phases. First, it tries to bracket the root, then it uses "Brent's" combined interpolation/golden section method to find the minimum. The advantage of this line search is that it does not require, as the other two methods do, that the step be in a descent direction, since it will look in both directions in an attempt to bracket the minimum. As such it is very appropriate for the derivative-free "principal axis" method. The downside of this line search is that it typically uses many function evaluations, so it is usually less efficient than the other two methods.

This example shows the effect of using the Brent method for line search. Note that in the phase of bracketing the root, it may use negative values of α. Even though the number of Newton steps is relatively small in this example, the total number of function evaluations is much larger than for other line search methods.

```
In[34]:= FindMinimumPlot[p,
          Method → {"Newton", "StepControl" → {"LineSearch", Method → "Brent"}}]

Out[34]= {{1.01471×10^-23, {X1 → 1., X2 → 1.}},

          {Steps → 13, Function → 188, Gradient → 14, Hessian → 14}, 
```



}

Trust Region Methods

A trust region method has a region around the current search point, where the quadratic model

$$q_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p \tag{3}$$

for "local minimization" is "trusted" to be correct and steps are chosen to stay within this region. The size of the region is modified during the search, based on how well the model agrees with actual function evaluations.

Very typically, the trust region is taken to be an ellipse such that $\|D\,p\| \leq \Delta$. D is a diagonal scaling (often taken from the diagonal of the approximate Hessian) and Δ is the trust region radius, which is updated at each step.

When the step based on the quadratic model alone lies within the trust region, then, assuming the function value gets smaller, that step will be chosen. Thus, just as with "line search" methods, the step control does not interfere with the convergence of the algorithm near to a minimum where the quadratic model is good. When the step based on the quadratic model lies outside the trust region, a step just up to the boundary of the trust region is chosen, such that the step is an approximate minimizer of the quadratic model on the boundary of the trust region.

Once a step p_k is chosen, the function is evaluated at the new point, and the actual function value is checked against the value predicted by the quadratic model. What is actually computed is the ratio of actual to predicted reduction.

$$\rho_k = \frac{f(x_k) - f(x_{k+}\,p_k)}{q_k(0) - q_k(p_k)} = \frac{\text{actual reduction of } f}{\text{predicted model reduction of } f}$$

If ρ_k is close to 1, then the quadratic model is quite a good predictor and the region can be increased in size. On the other hand, if ρ_k is too small, the region is decreased in size. When ρ_k is below a threshold, η, the step is rejected and recomputed. You can control this threshold with the method option `"AcceptableStepRatio"` $\rightarrow \eta$. Typically the value of η is quite small to avoid rejecting steps that would be progress toward a minimum. However, if obtaining the quadratic model at a point is quite expensive (e.g., evaluating the Hessian takes a relatively long time), a larger value of η will reduce the number of Hessian evaluations, but it may increase the number of function evaluations.

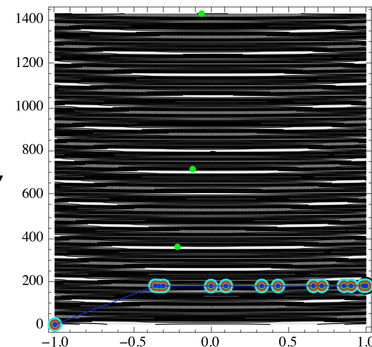
To start the trust region algorithm, an initial radius Δ needs to be determined. By default *Mathematica* uses the size of the step based on the model (1) restricted by a fairly loose relative step size limit. However, in some cases, this may take you out of the region you are primarily interested in, so you can specify a starting radius Δ_0 using the option `"StartingScaledStepSize"` $\rightarrow \Delta_0$. The option contains `Scaled` in its name because the trust region radius works through the diagonal scaling D, so this is not an absolute step size.

This loads a package that contains some utility functions.

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

This shows the steps and evaluations taken during a search for a local minimum of a function similar to Rosenbrock's function, using Newton's method with trust region step control.

```
In[2]:= FindMinimumPlot[(x - 1)^2 + 100 Sin[x^2 - y], {{x, -1}, {y, 1}},
         Method → {"Newton", "StepControl" -> "TrustRegion"}, MaxRecursion → 0]

Out[2]= {{-100., {x → 1., y → 178.5}},
         {Steps → 16, Function → 20, Gradient → 17, Hessian → 16}, 
```

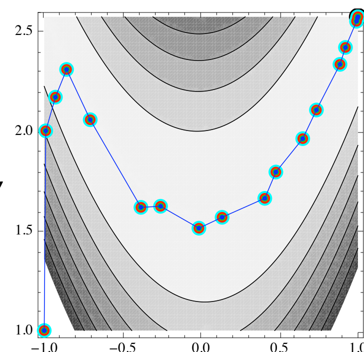


}

The plot looks quite bad because the search has extended over such a large region that the fine structure of the function cannot really be seen on that scale.

This shows the steps and evaluations for the same function, but with a restricted initial trust region radius Δ_0. Here the search stays much closer to the initial condition and follows the narrow valley.

```
In[3]:= FindMinimumPlot[(x - 1)^2 + 100 Sin[x^2 - y], {{x, -1}, {y, 1}}, Method →
          {"Newton", "StepControl" → {"TrustRegion", "StartingScaledStepSize" → 1}}]

Out[3]= {{-100., {x → 1., y → 2.5708}},
         {Steps → 18, Function → 20, Gradient → 19, Hessian → 19}, 
```



}

It is also possible to set an overall maximum bound for the trust region radius by using the option `"MaxScaledStepSize" -> ` Δ_{max} so that for any step, $\Delta_k \leq \Delta_{max}$.

Trust region methods can also have difficulties with functions which are not smooth due to problems with numerical roundoff in the function computation. When the function is not sufficiently smooth, the radius of the trust region will keep getting reduced. Eventually, it will get to the point at which it is effectively zero.

This gets the Freudenstein-Roth test problem from the Optimization`UnconstrainedProblems` package in a form where it can be solved by `FindMinimum`. (See "Test Problems".)

```
In[4]:= pfr = GetFindMinimumProblem[FreudensteinRoth]
```

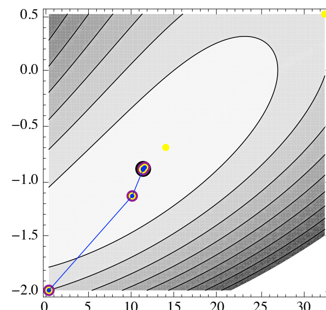
```
Out[4]= FindMinimumProblem[(-13 + X1 + X2 (-2 + (5 - X2) X2))^2 + (-29 + X1 + X2 (-14 + X2 (1 + X2)))^2,
  {{X1, 0.5}, {X2, -2.}}, {}, FreudensteinRoth, {2, 2}]
```

This finds a local minimum for the function using the default method. The default method in this case is the (trust region) Levenberg-Marquardt method since the function is a sum of squares.

```
In[5]:= FindMinimumPlot[pfr]
```

```
FindMinimum::sszero :
  The step size in the search has become less than the tolerance prescribed by the PrecisionGoal
    option, but the gradient is larger than the tolerance specified
    by the AccuracyGoal option. There is a possibility that the
    method has stalled at a point which is not a local minimum. >>
```

```
Out[5]= {{48.9843, {X1 -> 11.4128, X2 -> -0.896805}},
  {Steps -> 16, Residual -> 35, Jacobian -> 17},
```



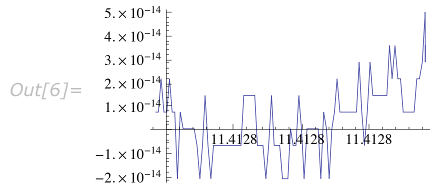
}

The message means that the size of the trust region has become effectively zero relative to the size of the search point, so steps taken would have negligible effect. **Note:** On some platforms, due to subtle differences in machine arithmetic, the message may not show up. This is because the reasons leading to the message have to do with numerical uncertainty, which can vary between different platforms.

This makes a plot of the variation function along the X_1 direction at the final point found.

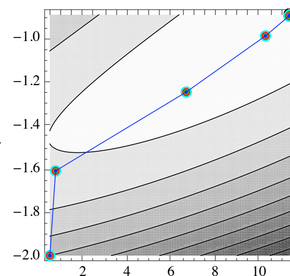
```
In[6]:= Block[{ϵ = 10^-7, x1f = 11.412778991937346, x2f = -0.8968052550911878, min},
  min = (-13 + X1 + X2 (-2 + (5 - X2) X2))^2 + (-29 + X1 + X2 (-14 + X2 (1 + X2)))^2 /.
    {X1 → x1f, X2 → x2f};
  Plot[((-13 + X1 + X2 (-2 + (5 - X2) X2))^2 + (-29 + X1 + X2 (-14 + X2 (1 + X2)))^2) - min /.
    X2 → x2f, {X1, x1f - ϵ, x1f + ϵ}]]
```

Out[6]=



The plot along one direction makes it fairly clear why no more improvement is possible. Part of the reason the Levenberg-Marquardt method gets into trouble in this situation is that convergence is relatively slow because the residual is nonzero at the minimum. With "Newton's" method, the convergence is faster, and the full quadratic model allows for a better estimate of step size, so that `FindMinimum` can have more confidence that the default tolerances have been satisfied.

```
In[52]:= FindMinimumPlot[pfr, Method → {"Newton", StepControl → "TrustRegion"}]

Out[52]= {{48.9843, {X1 → 11.4128, X2 → -0.896805}},

  {Steps → 6, Function → 7, Gradient → 7, Hessian → 7}, [plot]}
```



The following table summarizes the options for controlling trust region step control.

option name	default value	
"AcceptableStepRatio"	1/10 000	the threshold η, such that when the actual to prediction reduction $\rho_k \geq \eta$, the search is moved to the computed step
"MaxScaledStepSize"	∞	the value Δ_{max}, such that the trust region size $\Delta_k < \Delta_{max}$ for all steps
"StartingScaledStepSize"	Automatic	the initial trust region size Δ_0

Method options for "StepControl" -> "TrustRegion".

Setting Up Optimization Problems in Mathematica

Specifying Derivatives

The function `FindRoot` has a `Jacobian` option; the functions `FindMinimum`, `FindMaximum`, and `FindFit` have a `Gradient` option; and the "Newton" method has a method option `Hessian`. All these derivatives are specified with the same basic structure. Here is a summary of ways to specify derivative computation methods.

`Automatic`	find a symbolic derivative for the function and use finite difference approximations if a symbolic derivative cannot be found
`Symbolic`	same as `Automatic`, but gives a warning message if finite differences are to be used
`FiniteDifference`	use finite differences to approximate the derivative
expression	use the given *expression* with local numerical values of the variables to evaluate the derivative

Methods for computing gradient, Jacobian, and Hessian derivatives.

The basic specification for a derivative is just the method for computing it. However, all of the derivatives take options as well. These can be specified by using a list {*method*, *opts*}. Here is a summary of the options for the derivatives.

option name	*default value*	
`"EvaluationMonitor"`	`None`	expression to evaluate with local values of the variables every time the derivative is evaluated, usually specified with `:>` instead of `->` to prevent symbolic evaluation
`"Sparse"`	`Automatic`	sparse structure for the derivative; can be `Automatic`, `True`, `False`, or a pattern `SparseArray` giving the nonzero structure
`"DifferenceOrder"`	`1`	difference order to use when finite differences are used to compute the derivative

Options for computing gradient, Jacobian, and Hessian derivatives.

A few examples will help illustrate how these fit together.

This loads a package that contains some utility functions.

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

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

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

With just `Method → "Newton"`, `FindMinimum` issues an `lstol` message because it was not able to resolve the minimum well enough due to lack of good derivative information.

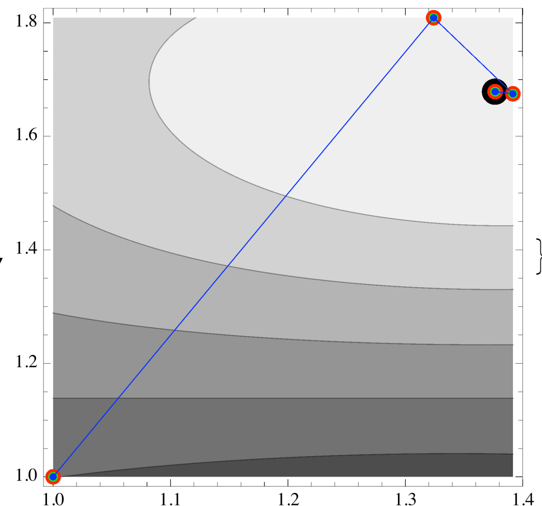
This shows the steps taken by `FindMinimum` when it has to use finite differences to compute the gradient and Hessian.

```
In[3]:= FindMinimumPlot[f[x, y], {{x, 1}, {y, 1}}, Method -> "Newton"]
```

FindMinimum::symd :
Unable to automatically compute the symbolic derivative of f[x, y] with respect to the arguments {x, y}. Numerical approximations to derivatives will be used instead. ≫

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[3]= {{-2., {x → 1.37638, y → 1.67867}},
 {Steps → 4, Function → 89, Gradient → 26}, 
```



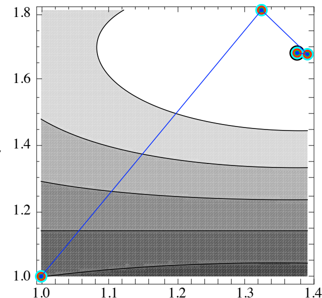
}

The following describes how you can use the gradient option to specify the derivative.

This computes the minimum of f[x, y] using a symbolic expression for its gradient.

```
In[4]:= FindMinimumPlot[f[x, y], {{x, 1}, {y, 1}},
         Gradient → {2 x Cos[x^2 + y^2] - 2 x Sin[x^2 - 3 y], 2 y Cos[x^2 + y^2] + 3 Sin[x^2 - 3 y]},
         Method → "Newton"]
```

```
Out[4]= {{-2., {x → 1.37638, y → 1.67868}},
         {Steps → 5, Function → 6, Gradient → 6, Hessian → 6}, 
```



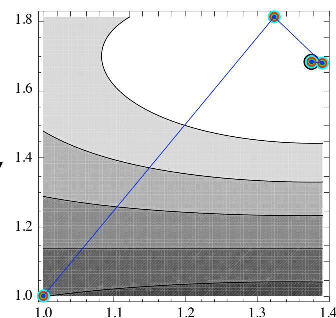

```
}
```

Symbolic derivatives are not always available. If you need extra accuracy from finite differences, you can increase the difference order from the default of 1 at the cost of extra function evaluations.

This computes the minimum of f[x, y] using a second-order finite difference to compute the gradient.

```
In[5]:= FindMinimumPlot[f[x, y], {{x, 1}, {y, 1}},
         Gradient → {Automatic, "DifferenceOrder" → 2}, Method → "Newton"]
```

```
Out[5]= {{-2., {x → 1.37638, y → 1.67868}},
         {Steps → 5, Function → 102, Gradient → 24, Hessian → 6}, 
```




```
}
```

Note that the number of function evaluations is much higher because function evaluations are used to compute the gradient, which is used to approximate the Hessian in turn. (The Hessian is computed with finite differences since no symbolic expression for it can be computed from the information given.)

The information given from `FindMinimumPlot` about the number of function, gradient, and Hessian evaluations is quite useful. The `EvaluationMonitor` options are what make this possible. Here is an example that simply counts the number of each type of evaluation. (The plot is made using `Reap` and `Sow` to collect the values at which the evaluations are done.)

This computes the minimum with counters to keep track of the number of steps and the number of function, gradient, and Hessian evaluations.

```
In[6]:= Block[{s = 0, e = 0, g = 0, h = 0},
          {FindMinimum[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
             {{x, 1}, {y, 1}}, StepMonitor :→ s++, EvaluationMonitor :→ e++,
             Gradient → {Automatic, EvaluationMonitor :→ g++}, Method →
              {"Newton", "Hessian" → {Automatic, EvaluationMonitor :→ h++}}], s, e, g, h}]

Out[6]= {{-2., {x → 1.37638, y → 1.67868}}, 5, 6, 6, 6}
```

Using such diagnostics can be quite useful for determining what methods and/or method parameters may be most successful for a class of problems with similar characteristics.

When *Mathematica* can access the symbolic structure of the function, it automatically does a structural analysis of the function and its derivatives and uses `SparseArray` objects to represent the derivatives when appropriate. Since subsequent numerical linear algebra can then use the sparse structures, this can have a profound effect on the overall efficiency of the search. When *Mathematica* cannot do a structural analysis, it has to assume, in general, that the structure is dense. However, if you know what the sparse structure of the derivative is, you can specify this with the `"Sparse"` method option and gain huge efficiency advantages, both in computing derivatives (with finite differences, the number of evaluations can be reduced significantly) and in subsequent linear algebra. This issue is particularly important when working with vector-valued variables. A good example for illustrating this aspect is the extended Rosenbrock problem, which has a very simple sparse structure.

This gets the extended Rosenbrock function with 1000 variables in symbolic form ready to be solved with `FindRoot` using the ``UnconstrainedProblems` `` package.

```
In[7]:= n = 1000; Short[pex = GetFindRootProblem[ExtendedRosenbrock, n], 20]

Out[7]//Short= FindRootProblem[{10 (-X1^2 + X2), 1 - X1, <<997>>, 1 - X999}, {<<1>>}, {}, <<18>>, {1000, 1000}]
```

This solves the problem using the symbolic form of the function.

```
In[8]:= Timing[Norm[1 - (Array[X# &, n] /. ProblemSolve[pex])]]

Out[8]= {0.321984, 0.}
```

For a function with simple form like this, it is easy to write a vector form of the function, which can be evaluated much more quickly than the symbolic form can, even with automatic compilation.

This defines a vector form of the extended Rosenbrock function, which evaluates very efficiently.

```
In[9]:= ExtendedRosenbrockResidual[X_List] := Module[{x1, x2},
          x1 = Take[X, {1, -1, 2}];
          x2 = Take[X, {2, -1, 2}];
          Flatten[Transpose[{10 (x2 - x1^2), 1 - x1}]]]
```

This extracts the starting point as a vector from the problem structure.

```
In[10]:= Short[start = pex[[2, All, 2]]]

Out[10]//Short= {-1.2, 1., -1.2, 1., -1.2, 1., -1.2, <<986>>, 1., -1.2, 1., -1.2, 1., -1.2, 1.}
```

This solves the problem using a vector variable and the vector function for evaluation.

```
In[11]:= Timing[Norm[1 - (X /. FindRoot[ExtendedRosenbrockResidual[X], {X, start}])]]

Out[11]= {12.2235, 0.}
```

The solution with the function, which is faster to evaluate, winds up being slower overall because the Jacobian has to be computed with finite differences since the `x_List` pattern makes it opaque to symbolic analysis. It is not so much the finite differences that are slow as the fact that it needs to do 100 function evaluations to get all the columns of the Jacobian. With knowledge of the structure, this can be reduced to two evaluations to get the Jacobian. For this function, the structure of the Jacobian is quite simple.

This defines a pattern `SparseArray`, which has the structure of nonzeros for the Jacobian of the extended Rosenbrock function. (By specifying _ for the values in the rules, the `SparseArray` is taken to be a template of the `Pattern` type as indicated in the output form.)

```
In[12]:= sparsity = SparseArray[
           Flatten[Table[{{i, i} → _, {i, i + 1} → _, {i + 1, i} → _}, {i, 1, n - 1, 2}]]]

Out[12]= SparseArray[<1500>, {1000, 1000}, Pattern]
```

This solves the problem with the knowledge of the actual Jacobian structure, showing a significant cost savings.

```
In[13]:= Timing[Norm[1 - (X /. FindRoot[ExtendedRosenbrockResidual[X], {X, start},
               Method → {"Newton"}, Jacobian → {Automatic, Sparse → sparsity}])]]

Out[13]= {0.031138, 0.}
```

When a sparse structure is given, it is also possible to have the value computed by a symbolic expression that evaluates to the values corresponding to the positions given in the sparse structure template. Note that the values must correspond directly to the positions as ordered in the `SparseArray` (the ordering can be seen using `ArrayRules`). One way to get a consistent ordering of indices is to transpose the matrix twice, which results in a `SparseArray` with indices in lexicographic order.

This transposes the nonzero structure matrix twice to get the indices sorted.

```
In[14]:= sparsity = Transpose[Transpose[sparsity]]

Out[14]= SparseArray[<1500>, {1000, 1000}, Pattern]
```

This defines a function that will return the nonzero values in the Jacobian corresponding to the index positions in the nonzero structure matrix.

```
In[15]:= ERJValues[X_List] := Module[{x1, zero},
           x1 = Take[X, {1, -1, 2}];
           zero = 0. x1;
           Flatten[Transpose[{-20 x1, 10. + zero, -1. + zero}]]]
```

This solves the problem with the resulting sparse symbolic Jacobian.

```
In[16]:= Timing[Norm[1 - (X /. FindRoot[ExtendedRosenbrockResidual[X], {X, start},
                Method → {"Newton"}, Jacobian → {ERJValues[X], Sparse → sparsity}])]]

Out[16]= {0.025614, 0.}
```

In this case, using the sparse Jacobian is not significantly faster because the Jacobian is so sparse that a finite difference approximation can be found for it in only two function evaluations and because the problem is well enough defined near the minimum that the extra accuracy in the Jacobian does not make any significant difference.

Variables and Starting Conditions

All the functions `FindMinimum`, `FindMaximum`, and `FindRoot` take variable specifications of the same form. The function `FindFit` uses the same form for its parameter specifications.

`FindMinimum`[f, $vars$]	find a local minimum of f with respect to the variables given in $vars$
`FindMinimum`[f, $vars$]	find a local maximum of f with respect to the variables given in $vars$
`FindRoot`[f, $vars$]	find a root $f = 0$ with respect to the variables given in $vars$
`FindRoot`[$eqns$, $vars$]	find a root of the equations $eqns$ with respect to the variables given in $vars$
`FindFit`[$data$, $expr$, $pars$, $vars$]	find values of the parameters $pars$ that make $expr$ give a best fit to $data$ as a function of $vars$

Variables and parameters in the "`Find`" functions.

The list $vars$ ($pars$ for `FindFit`) should be a list of individual variable specifications. Each variable specification should be of the following form.

{var, st}	variable var has starting value st
{var, st_1, st_2}	variable var has two starting values st_1 and st_2; the second starting condition is only used with the principal axis and secant methods
{var, st, rl, ru}	variable var has starting value st; the search will be terminated when the value of var goes outside of the interval $[rl, ru]$
{var, st_1, st_2, rl, ru}	variable var has two starting values st_1 and st_2; the search will be terminated when the value of var goes outside of the interval $[rl, ru]$

Individual variable specifications in the "`Find`" functions.

The specifications in $vars$ all need to have the same number of starting values. When region bounds are not specified, they are taken to be unbounded, that is, $rl = -\infty$, $ru = \infty$.

Vector- and Matrix-Valued Variables

The most common use of variables is to represent numbers. However, the variable input syntax supports variables that are treated as vectors, matrices, or higher-rank tensors. In general, the "`Find`" commands, with the exception of `FindFit`, which currently only works with scalar variables, will consider a variable to take on values with the same rectangular structure as the starting conditions given for it.

Here is a matrix.

```
In[1]:= A = (0 1 2
             3 4 5
             6 7 8);
```

This uses `FindRoot` to find an eigenvalue and corresponding normalized eigenvector for `A`.

```
In[2]:= FindRoot[{A.x == λ x, x.x == 1}, {{λ, 1}, {x, {1, 2, 3}}}]
Out[2]= {λ → 13.3485, x → {0.164764, 0.505774, 0.846785}}
```

Of course, this is not the best way to compute the eigenvalue, but it does show how the variable dimensions are picked up from the starting values. Since λ has a starting value of 1, it is taken to be a scalar. On the other hand, x is given a starting value, which is a vector of length 3, so it is always taken to be a vector of length 3.

If you use multiple starting values for variables, it is necessary that the values have consistent dimensions and that each component of the starting values is distinct.

This finds a different eigenvalue using two starting conditions for each variable.

```
In[3]:= FindRoot[{A.x == λ x, x.x == 1}, {{λ, -2, -1}, {x, {-1, 0, 0}, {0, 1, 1}}}]
Out[3]= {λ → -1.34847, x → {-0.7997, -0.104206, 0.591288}}
```

One advantage of variables that can take on vector and matrix values is that they allow you to write functions, which can be very efficient for larger problems and/or handle problems of different sizes automatically.

This defines a function that gives an objective function equivalent to the `ExtendedRosenbrock` problem in the `UnconstrainedProblems` package. The function expects a value of x which is a matrix with two rows.

```
In[4]:= ExtendedRosenbrockObjective[x_ /; ((Length[x] == 2) && MatrixQ[x])] :=
         Module[{x1, x2},
          {x1, x2} = x;
          x2 -= x1^2;
          x1 -= 1;
          x1.x1 + 100 x2.x2]
```

Note that since the value of the function would be meaningless unless x had the correct structure, the definition is restricted to arguments with that structure. For example, if you defined the function for any pattern `x_`, then evaluating with an undefined symbol `x` (which is what `FindMinimum` does) gives meaningless unintended results. It is often the case that when working with functions for vector-valued variables, you will have to restrict the definitions. Note that

the definition above does not rule out symbolic values with the right structure. For example, `ExtendedRosenbrockObjective[{{`x_{11}`, `x_{12}`}, {`x_{21}`, `x_{22}`}}]` gives a symbolic representation of the function for scalar x_{11}, ….

This uses `FindMinimum` to solve the problem given a generic value for the problem size. You can change the value of n without changing anything else to solve problems of different size.

```
In[5]:= n = 10;
start = {Table[-1.2, {n}], Table[1., {n}]};
FindMinimum[ExtendedRosenbrockObjective[x], {x, start}]
```

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[7]= {2.00081×10^-10,
 {x→{{0.999996, 0.999996, 0.999996, 0.999996, 0.999996, 0.999996, 0.999996, 0.999996,
     0.999996, 0.999996}, {0.999991, 0.999991, 0.999991, 0.999991,
     0.999991, 0.999991, 0.999991, 0.999991, 0.999991, 0.999991}}}}
```

The solution did not achieve the default tolerances due to the fact that *Mathematica* was not able to get symbolic derivatives for the function, so it had to fall back on finite differences that are not as accurate.

A disadvantage of using vector- and matrix-valued variables is that *Mathematica* cannot currently compute symbolic derivatives for them. Sometimes it is not difficult to develop a function that gives the correct derivative. (Failing that, if you really need greater accuracy, you can use higher-order finite differences.)

This defines a function that returns the gradient for the `ExtendedRosenbrockObjective` function. Note that the gradient is a vector obtained by flattening the matrix corresponding to the variable positions.

```
In[8]:= ExtendedRosenbrockGradient[x_ /; ((Length[x] == 2) && MatrixQ[x])] :=
 Module[{x1, x2},
  {x1, x2} = x;
  x2 -= x1^2;
  Flatten[{2 (x1 - 1) - 400 x1 x2, 200 x2}]]
```

This solves the problem using the symbolic value of the gradient.

```
In[9]:= n = 10;
start = {Table[-1.2, {n}], Table[1., {n}]};
FindMinimum[ExtendedRosenbrockObjective[x],
 {x, start}, Gradient → ExtendedRosenbrockGradient[x]]
```

```
Out[11]= {3.00886×10^-20,
 {x→{{1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}, {1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}}}}
```

Jacobian and Hessian derivatives are often sparse. You can also specify the structural sparsity of these derivatives when appropriate, which can reduce overall solution complexity by quite a bit.

Termination Conditions

Mathematically, sufficient conditions for a local minimum of a smooth function are quite straightforward: x^* is a local minimum if $\nabla f(x^*) = 0$ and the Hessian $\nabla^2 f(x^*)$ is positive definite. (It is a necessary condition that the Hessian be positive semidefinite.) The conditions for a root are even simpler. However, when the function f is being evaluated on a computer where its value is only known, at best, to a certain precision, and practically only a limited number of function evaluations are possible, it is necessary to use error estimates to decide when a search has become close enough to a minimum or a root, and to compute the solution only to a finite tolerance. For the most part, these estimates suffice quite well, but in some cases, they can be in error, usually due to unresolved fine scale behavior of the function.

Tolerances affect how close a search will try to get to a root or local minimum before terminating the search. Assuming that the function itself has some error (as is typical when it is computed with numerical values), it is not typically possible to locate the position of a minimum much better than to half of the precision of the numbers being worked with. This is because of the quadratic nature of local minima. Near the bottom of a parabola, the height varies quite slowly as you move across from the minimum. Thus, if there is any error noise in the function, it will typically mask the actual rise of the parabola over a width roughly equal to the square root of the noise. This is best seen with an example.

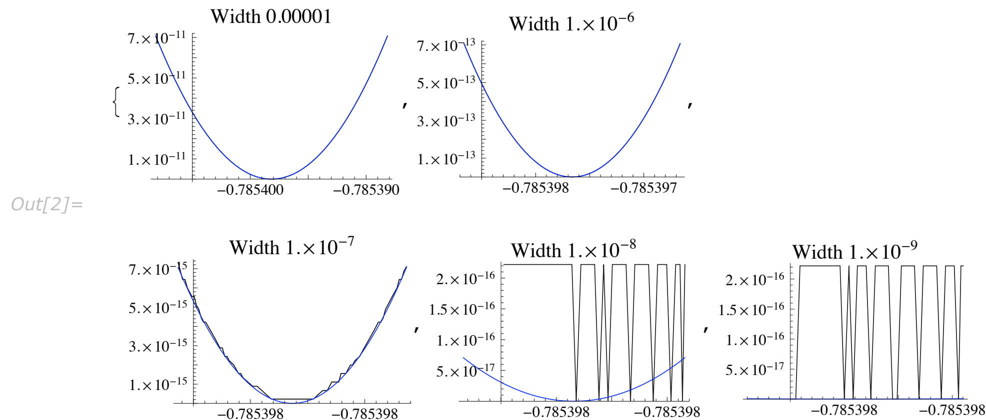
This loads a package that contains some utility functions.

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

The following command displays a sequence of plots showing the minimum of the function $\sin(x) - \cos(x) + \sqrt{2}$ over successively smaller ranges. The curve computed with machine numbers is shown in black; the actual curve (computed with 100 digits of precision) is shown in blue.

In[2]:=
```
Table[Block[{ϵ = 10.^-k},
  Show[{Plot[Sin[x] - Cos[x] + Sqrt[2], {x, -π / 4 - ϵ, -π / 4 + ϵ}, PlotStyle → Black],
    Plot[Sin[x] - Cos[x] + Sqrt[2], {x, -π / 4 - ϵ, -π / 4 + ϵ}, PlotStyle → Blue,
     WorkingPrecision → 100]}, PlotLabel → Row[{"Width ", ϵ}]]], {k, 5, 9}]
```

Out[2]=



From the sequence of plots, it is clear that for changes of order 10^{-8}, which is about half of machine precision and smaller, errors in the function are masking the actual shape of the curve near the minimum. With just sampling of the function at that precision, there is no way to be sure if a given point gives the smallest local value of the function or not to any closer tolerance.

The value of the derivative, if it is computed symbolically, is much more reliable, but for the general case, it is not sufficient to rely only on the value of the derivative; the search needs to find a local minimal value of the function where the derivative is small to satisfy the tolerances in general. Note also that if symbolic derivatives of your function cannot be computed and finite differences or a derivative-free method is used, the accuracy of the solution may degrade further.

Root finding can suffer from the same inaccuracies in the function. While it is typically not as severe, some of the error estimates are based on a merit function, which does have a quadratic shape.

For the reason of this limitation, the default tolerances for the `Find` functions are all set to be half of the final working precision. Depending on how much error the function has, this may or may not be achievable, but in most cases it is a reasonable goal. You can adjust the tolerances using the `AccuracyGoal` and `PrecisionGoal` options. When `AccuracyGoal` -> ag and `PrecisionGoal` -> pg, this defines tolerances $\text{tol}_a = 10^{-ag}$ and $\text{tol}_r = 10^{-pg}$.

Given tol_a and tol_r `FindMinimum` tries to find a value x_k such that $\|x_k - x^*\| \leq \max(\text{tol}_a, \|x_k\| \, \text{tol}_r)$. Of course, since the exact position of the minimum, x^*, is not known, the quantity $\|x_k - x^*\|$ is estimated. This is usually done based on past steps and derivative values. To match the derivative condition at a minimum, the additional requirement $\|\nabla f(x_k)\| \leq \text{tol}_a$ is imposed. For `FindRoot`, the corresponding condition is that just the residual be small at the root: $\|f\| \leq \text{tol}_a$.

This finds the $\sqrt{2}$ to at least 12 digits of accuracy, or within a tolerance of 10^{-12}. The precision goal of ∞ means that $\text{tol}_r = 0$, so it does not have any effect in the formula. (**Note:** you cannot similarly set the accuracy goal to ∞ since that is always used for the size of the residual.)

```
In[3]:= FindRoot[x^2 - 2, {x, 1}, AccuracyGoal → 12, PrecisionGoal → ∞]

Out[3]= {x → 1.41421}
```

This shows that the result satisfied the requested error tolerances.

```
In[4]:= {x - Sqrt[2], x^2 - 2} /. %

Out[4]= {0., 4.44089×10^-16}
```

This tries to find the minimum of the function $\sin(x) - \cos(x)$ to 8 digits of accuracy. `FindMinimum` gives a warning message because of the error in the function as seen in the plots.

```
In[5]:= FindMinimum[Sin[x] - Cos[x], {x, 0},
         Method -> "Newton", AccuracyGoal → 8, PrecisionGoal → ∞]

    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[5]= {-1.41421, {x → -0.785398}}
```

This shows that though the value at the minimum was found to be basically machine epsilon, the position was only found to the order of 10^{-8} or so.

```
In[6]:= {Sqrt[2] + %[[1]], π / 4 + x /. %[[2]]}

Out[6]= {2.22045×10^-16, -1.26022×10^-8}
```

In multiple dimensions, the situation is even more complicated since there can be more error in some directions than others, such as when a minimum is found along a relatively narrow valley, as in the Freudenstein-Roth problem. For searches such as this, often the search parameters are scaled, which in turn affects the error estimates. Nonetheless, it is still typical that the quadratic shape of the minimum affects the realistically achievable tolerances.

When you need to find a root or minimum beyond the default tolerances, it may be necessary to increase the final working precision. You can do this with the `WorkingPrecision` option. When you use `WorkingPrecision -> prec`, the search starts at the precision of the starting values and is adaptively increased up to *prec* as the search converges. By default, `WorkingPrecision -> MachinePrecision`, so machine numbers are used, which are usually much faster. Going to higher precision can take significantly more time, but can get you much more accurate results if your function is defined in an appropriate way. For very high-precision solutions, "Newton's" method is recommended because its quadratic convergence rate significantly reduces the number of steps ultimately required.

It is important to note that increasing the setting of the `WorkingPrecision` option does no good if the function is defined with lower-precision numbers. In general, for `WorkingPrecision -> prec` to be effective, the numbers used to define the function should be exact or at least of precision *prec*. When possible, the precision of numbers in the function is artificially raised to *prec* using `SetPrecision` so that convergence still works, but this is not always possible. In any case, when the functions and derivatives are evaluated numerically, the precision of the results is raised to *prec* if necessary so that the internal arithmetic can be done with *prec* digit precision. Even so, the actual precision or accuracy of the root or minimum and its position is limited by the accuracy in the function. This is especially important to keep in mind when using `FindFit`, where data is usually only known up to a certain precision.

Here is a function defined using machine numbers.

```
In[7]:= f[x_?NumberQ] := Sin[1. x] - Cos[1. x];
```

Even with higher working precision, the minimum cannot be resolved better because the actual function still has the same errors as shown in the plots. The derivatives were specified to keep other things consistent with the computation at machine precision shown previously.

```
In[8]:= FindMinimum[f[x], {x, 0}, Gradient → {Cos[1. x] + Sin[1. x]},
         Method → {"Newton", Hessian → {{Cos[1. x] - Sin[1. x]}}},
         AccuracyGoal → 8, PrecisionGoal → ∞, WorkingPrecision → 20]

   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 20.` digits of working precision to meet these tolerances. ≫

Out[8]= {-1.4142135623730949234, {x → -0.78539817599970194669}}
```

Here is the computation done with 20-digit precision when the function does not have machine numbers.

```
In[9]:= FindMinimum[Sin[x] - Cos[x], {x, 0}, Method -> "Newton",
         AccuracyGoal → 8, PrecisionGoal → ∞, WorkingPrecision → 20]

Out[9]= {-1.4142135623730950488, {x → -0.78539816339744830962}}
```

If you specify `WorkingPrecision` -> *prec*, but do not explicitly specify the `AccuracyGoal` and `PrecisionGoal` options, then their default settings of `Automatic` will be taken to be `AccuracyGoal` -> *prec* / 2 and `PrecisionGoal` -> *prec* / 2. This leads to the smallest tolerances that can realistically be expected in general, as discussed earlier.

Here is the computation done with 50-digit precision without an explicitly specified setting for the `AccuracyGoal` or `PrecisionGoal` options.

```
In[10]:= FindMinimum[Sin[x] - Cos[x], {x, 0}, Method -> "Newton", WorkingPrecision → 50]

Out[10]= {-1.4142135623730950488016887242096980785696718753769,
          {x → -0.78539816339744830961566084581987572104929234984378}}
```

This shows that though the value at the minimum was actually found to be even better than the default 25-digit tolerances.

```
In[11]:= {Sqrt[2] + %[[1]], π / 4 + x /. %[[2]]}

Out[11]= {0.×10^-50, 0.×10^-51}
```

The following table shows a summary of the options affecting precision and tolerance.

option name	*default value*	
`WorkingPrecision`	`MachinePrecision`	the final working precision, *prec*, to use; precision is adaptively increased from the smaller of *prec* and the precision of the starting conditions to *prec*
`AccuracyGoal`	`Automatic`	setting *ag* determines an absolute tolerance by $tol_a = 10^{-ag}$; when `Automatic`, $ag = prec / 2$
`PrecisionGoal`	`Automatic`	setting *pg* determines an absolute tolerance by $tol_r = 10^{-pg}$; when `Automatic`, $pg = prec / 2$

Precision and tolerance options in the `"Find"` functions.

A search will sometimes converge slowly. To prevent slow searches from going on indefinitely, the `Find` commands all have a maximum number of iterations (steps) that will be allowed

before terminating. This can be controlled with the option `MaxIterations` that has the default value `MaxIterations -> 100`. When a search terminates with this condition, the command will issue the `cvmit` message.

This gets the Brown-Dennis problem from the `Optimization`UnconstrainedProblems`` package.

```
In[12]:= Short[bd = GetFindMinimumProblem[BrownDennis], 5]
```

$$\text{Out[12]//Short=}\ \text{FindMinimumProblem}\Big[\Big(\Big(-e^{1/5}+X_1+\frac{X_2}{5}\Big)^2+\Big(-\text{Cos}\Big[\frac{1}{5}\Big]+X_3+\text{Sin}\Big[\frac{1}{5}\Big]X_4\Big)^2\Big)^2+\Big(\Big(-e^{2/5}+X_1+\frac{2X_2}{5}\Big)^2+\Big(-\text{Cos}\Big[\frac{2}{5}\Big]+X_3+\text{Sin}\Big[\frac{2}{5}\Big]X_4\Big)^2\Big)^2+\ll 17\gg+\Big(\big(-e^4+X_1+4X_2\big)^2+\big(-\text{Cos}[4]+X_3+\text{Sin}[4]X_4\big)^2\Big)^2,$$
$$\{\{X_1,25.\},\{X_2,5.\},\{X_3,-5.\},\{X_4,-1.\}\},\{\},\text{BrownDennis},\{4,20\}\Big]$$

This attempts to solve the problem with the default method, which is the Levenberg-Marquardt method, since the function is a sum of squares.

```
In[13]:= ProblemSolve[bd]
```

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

$$\text{Out[13]=}\ \{105\,443.,\{X_1\to -7.35071,X_2\to 11.7365,X_3\to -0.60436,X_4\to 0.168396\}\}$$

The Levenberg-Marquardt method is converging slowly on this problem because the residual is nonzero near the minimum and the second-order part of the Hessian is needed. While the method eventually does converge in just under 400 steps, perhaps a better option is to use a method which may converge faster.

```
In[44]:= ProblemSolve[bd, Method → QuasiNewton]
```

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. ≫

$$\text{Out[44]=}\ \{85\,822.2,\{X_1\to -11.5944,X_2\to 13.2036,X_3\to -0.403439,X_4\to 0.236779\}\}$$

In a larger calculation, one possibility when hitting the iteration limit is to use the final search point, which is returned, as a starting condition for continuing the search, ideally with another method.

Symbolic Evaluation

The functions `FindMinimum`, `FindMaximum`, and `FindRoot` have the `HoldAll` attribute and so have special semantics for evaluation of their arguments. First, the variables are determined from the second argument, then they are localized. Next, the function is evaluated symbolically, then processed into an efficient form for numerical evaluation. Finally, during the execution of the command, the function is repeatedly evaluated with different numerical values. Here is a list showing these steps with additional description.

Determine variables	process the second argument; if the second argument is not of the correct form (a list of variables and starting values), it will be evaluated to get the correct form
Localize variables	in a manner similar to `Block` and `Table`, add rules to the variables so that any assignments given to them will not affect your *Mathematica* session beyond the scope of the `"Find"` command and so that previous assignments do not affect the value (the variable will evaluate to itself at this stage)
Evaluate the function	with the locally undefined (symbolic) values of the variables, evaluate the first argument (function or equations). **Note:** this is a change which was instituted in *Mathematica* 5, so some adjustments may be necessary for code that ran in previous versions. If your function is such that symbolic evaluation will not keep the function as intended or will be prohibitively slow, you should define your function so that it only evaluates for numerical values of the variables. The simplest way to do this is by defining your function using `PatternTest` (`?`), as in *f*[*x*_?`NumberQ`] := *definition*.
Preprocess the function	analyze the function to help determine the algorithm to use (e.g., sum of squares -> Levenberg-Marquardt); optimize and compile the function for faster numerical evaluation if possible: for `FindRoot` this first involves going from equations to a function
Compute derivatives	compute any needed symbolic derivatives if possible; otherwise, do preprocessing needed to compute derivatives using finite differences
Evaluate numerically	repeatedly evaluate the function (and derivatives when required) with different numerical values

Steps in processing the function for the `"Find"` commands.

`FindFit` does not have the `HoldAll` attribute, so its arguments are all evaluated before the commands begin. However, it uses all of the stages described above, except instead of evaluating the function, it constructs a function to minimize from the model function, variables, and provided data.

You will sometimes want to prevent symbolic evaluation, most often when your function is not an explicit formula, but a value derived through running through a program. An example of what happens and how to prevent the symbolic evaluation is shown.

This attempts to solve a simple boundary value problem numerically using shooting.

```
In[1]:= FindRoot[
         First[x[1] /. NDSolve[{x''[t] + (x[t] UnitStep[t] + 1) x[t] == 0, x[-1] == 0,
             x'[-1] == xp}, x, {t, -1, 1}]], {xp, Pi}]

   NDSolve::ndinnt : Initial condition xp is not a number or a rectangular array of numbers. >>

   ReplaceAll::reps :
     {NDSolve[{x[t] (1 + Times[<<2>>]) + x''[t] == 0, x[-1] == 0, x'[-1] == xp}, x, {t, -1, 1}]} is neither a list of
           replacement rules nor a valid dispatch table, and so cannot be used for replacing. >>

   FindRoot::nlnum :
     The function value {x[1.]} is not a list of numbers with dimensions {1} at {xp} = {3.14159}. >>

   NDSolve::ndinnt : Initial condition xp is not a number or a rectangular array of numbers. >>

   ReplaceAll::reps :
     {NDSolve[{x[t] (1 + Times[<<2>>]) + x''[t] == 0, x[-1] == 0, x'[-1] == xp}, x, {t, -1, 1}]} is neither a list of
           replacement rules nor a valid dispatch table, and so cannot be used for replacing. >>

   FindRoot::nlnum :
     The function value {x[1.]} is not a list of numbers with dimensions {1} at {xp} = {3.14159}. >>

Out[1]= FindRoot[First[x[1] /.
           NDSolve[{x''[t] + (x[t] UnitStep[t] + 1) x[t] == 0, x[-1] == 0, x'[-1] == xp}, x, {t, -1, 1}]], {xp, π}]
```

The command fails because of the symbolic evaluation of the function. You can see what happens when you evaluate it inside of `Block`.

This evaluates the function given to `FindRoot` with a local (undefined) value of `xp`.

```
In[2]:= Block[{xp},
         First[x[1] /. NDSolve[{x''[t] + (x[t] UnitStep[t] + 1) x[t] == 0, x[-1] == 0,
             x'[-1] == xp}, x, {t, -1, 1}]]]

   NDSolve::ndinnt : Initial condition xp is not a number or a rectangular array of numbers. >>

   ReplaceAll::reps :
     {NDSolve[{x[t] (1 + Times[<<2>>]) + x''[t] == 0, x[-1] == 0, x'[-1] == xp}, x, {t, -1, 1}]} is neither a list of
           replacement rules nor a valid dispatch table, and so cannot be used for replacing. >>

Out[2]= x[1]
```

Of course, this is not at all what was intended for the function; it does not even depend on xp. What happened is that without a numerical value for xp, NDSolve fails, so ReplaceAll (/.) fails because there are no rules. First just returns its first argument, which is x[1]. Since the function is meaningless unless xp has numerical values, it should be properly defined.

This defines a function that returns the value x[1] as a function of a numerical value for x'[t] at t = -1.

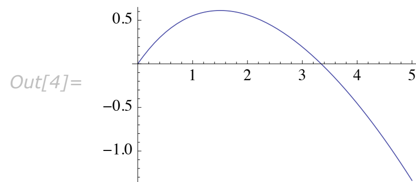
```
In[3]:= fx1[xp_?NumberQ] :=
         First[x[1] /. NDSolve[{x''[t] + (x[t] UnitStep[t] + 1) x[t] == 0,
              x[-1] == 0, x'[-1] == xp}, x, {t, -1, 1}]]
```

An advantage of having a simple function definition outside of FindRoot is that it can independently be tested to make sure that it is what you really intended.

This makes a plot of fx1.

```
In[4]:= Plot[fx1[xp], {xp, 0, 5}]
```

Out[4]=



From the plot, you can deduce two bracketing values for the root, so it is possible to take advantage of "Brent's" method to quickly and accurately solve the problem.

This solves the shooting problem.

```
In[5]:= FindRoot[fx1[xp], {xp, 3, 4}]
Out[5]= {xp → 3.34372}
```

It may seem that symbolic evaluation just creates a bother since you have to define the function specifically to prevent it. However, without symbolic evaluation, it is hard for *Mathematica* to take advantage of its unique combination of numerical and symbolic power. Symbolic evaluation means that the commands can consistently take advantage of benefits that come from symbolic analysis, such as algorithm determination, automatic computation of derivatives, automatic optimization and compilation, and structural analysis.

UnconstrainedProblems Package

Plotting Search Data

The utility functions `FindMinimumPlot` and `FindRootPlot` show search data for `FindMinimum` and `FindRoot` for one- and two-dimensional functions. They work with essentially the same arguments as `FindMinimum` and `FindRoot` except that they additionally take options, which affect the graphics functions they call to provide the plots, and they do not have the `HoldAll` attribute as do `FindMinimum` and `FindRoot`.

`FindMinimumPlot[`f`,{`x`,`x_{st}`},`*opts*`]`	plot the steps and the points at which the function f and any of its derivatives that were evaluated in `FindMinimum[`f`, {`x`, `x_{st}`}]` superimposed on a plot of f versus x; *opts* may include options from both `FindMinimum` and `Plot`
`FindMinimumPlot[`f`, {{`x`,`x_{st}`},{`y`,`y_{st}`}},`*opts*`]`	plot the steps and the points at which the function f and any of its derivatives that were evaluated in `FindMinimum[`f`, {{`x`, `x_{st}`}, {`y`, `y_{st}`}}]` superimposed on a contour plot of f as a function of x and y; *opts* may include options from both `FindMinimum` and `ContourPlot`
`FindRootPlot[`f`,{`x`,`x_{st}`},`*opts*`]`	plot the steps and the points at which the function f and any of its derivatives which were evaluated in `FindRoot[`f`, {`x`, `x_{st}`}]` superimposed on a plot of f versus x; *opts* may include options from both `FindRoot` and `Plot`
`FindRootPlot[`f`, {{`x`,`x_{st}`},{`y`,`y_{st}`}},`*opts*`]`	plot the steps and the points at which the function f and any of its derivatives that were evaluated in `FindRoot[`f`, {{`x`, `x_{st}`}, {`y`, `y_{st}`}}]` superimposed on a contour plot of the merit function f as a function of x and y; *opts* may include options from both `FindRoot` and `ContourPlot`

Plotting search data.

Note that to simplify processing and reduce possible confusion about the function f, `FindRootPlot` does not accept equations; it finds a root $f = 0$.

Steps and evaluation points are color coded for easy detection as follows:

- Steps are shown with blue lines and blue points.
- Function evaluations are shown with green points.
- Gradient evaluations are shown with red points.
- Hessian evaluations are shown with cyan points.
- Residual function evaluations are shown with yellow points.
- Jacobian evaluations are shown with purple points.
- The search termination is shown with a large black point.

`FindMinimumPlot` and `FindRootPlot` return a list containing {*result*, *summary*, *plot*}, where:

- *result* is the result of `FindMinimum` or `FindRoot`.
- *summary* is a list of rules showing the numbers of steps and evaluations of the function and its derivatives.
- *plot* is the graphics object shown.

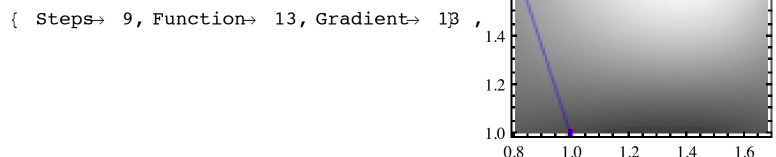
This loads the package.

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

This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function $\cos(x^2 - 3y) + \sin(x^2 + y^2)$ starting at the point $\{x, y\} = \{1, 1\}$. Options are given to `ContourPlot` so that no contour lines are shown and the function value is indicated by grayscale. Since `FindMinimum` by default uses the "quasi-Newton" method, there are only evaluations of the function and gradient that occur at the same points, indicated by the red circles with green centers.

```
In[2]:= FindMinimumPlot[Cos[x^2 - 3 y] + Sin[x^2 + y^2],
         {{x, 1}, {y, 1}}, Contours → 100, ContourLines → False]
```

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

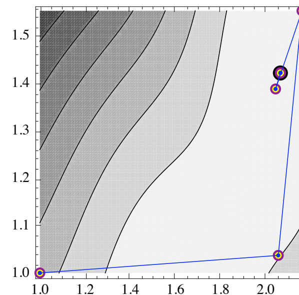


```
 {Steps → 9, Function → 13, Gradient → 13}, [plot]}
```

This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function $(x^2 - 3\,y)^2 + \sin^2(x^2 + y^2)$ starting at the point $\{x, y\} = \{1, 1\}$. Since the problem is a sum of squares, `FindMinimum` by default uses the "Gauss-Newton"/Levenberg-Marquardt method that derives a residual function and only evaluates it and its Jacobian. Points at which the residual function is evaluated are shown with yellow dots. The yellow dots surrounded by a large purple circle are points at which the Jacobian was evaluated as well.

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

Out[3]=
```
{{2.27472×10^-28, {x → 2.06482, y → 1.42116}},
 {Steps → 6, Residual → 7, Jacobian → 7}, 
```

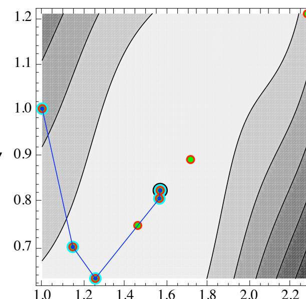



```
}
```

This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function $(x^2 - 3\,y)^2 + \sin^2(x^2 + y^2)$ starting at the point $\{x, y\} = \{1, 1\}$ using "Newton's" method. Points at which the function, gradient, and Hessian were all evaluated are shown by concentric green, red, and cyan circles. Note that in this example, all of the Newton steps satisfied the Wolfe conditions, so there were no points where the function and gradient were evaluated separately from the Hessian, which is not always the case. Note also that Newton's method finds a different local minimum than the default method.

In[4]:=
```
FindMinimumPlot[(x^2 - 3 y)^2 + Sin[x^2 + y^2]^2,
 {{x, 1}, {y, 1}}, Method → Newton]
```

Out[4]=
```
{{4.03019×10^-29, {x → 1.57033, y → 0.82198}},
 {Steps → 6, Function → 10, Gradient → 10, Hessian → 7}, 
```

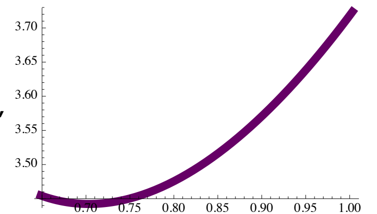



```
}
```

This shows the steps and evaluations used by `FindMinimum` to find a local minimum of the function $e^x + \frac{1}{x}$ with two starting values superimposed on the plot of the function. Options are given to `Plot` so that the curve representing the function is thick and purple. With two starting values, `FindMinimum` uses the derivative-free principal axis method, so there are only function evaluations, indicated by the green dots.

```
In[5]:= FindMinimumPlot[Exp[x] + 1 / x, {x, 1, 1.1},
         PlotStyle → {Thickness[.025], RGBColor[.4, 0, .4]}]
```

```
Out[5]= {{3.44228, {x → 0.703467}}, {Steps → 6, Function → 14},
```

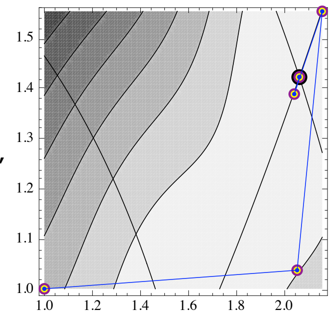


```
}
```

This shows in two dimensions the steps and evaluations used by `FindRoot` to find a root of the function $\{x^2 - 3\,y, \sin(x^2 + y^2)\} = \{0, 0\}$ starting at the point $\{x, y\} = \{1, 1\}$. As described earlier, the function is a residual, and the default method in `FindRoot` evaluates the residual and its Jacobian as shown by the yellow dots and purple circles. Note that this plot is nearly the same as the one produced by `FindMinimumPlot` with the default method for the function $(x^2 - 3\,y)^2 + \sin^2(x^2 + y^2)$ since the residual is the same. `FindRootPlot` also shows the zero contour of each component of the residual function in red and green.

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

```
Out[6]= {{x → 2.06482, y → 1.42116}, {Steps → 7, Residual → 7, Jacobian → 7},
```



```
}
```

Test Problems

All the test problems presented in [MGH81] have been coded into *Mathematica* in the `` Optimization`UnconstrainedProblems` `` package. A data structure is used so that the problems can be processed for solution and testing with `FindMinimum` and `FindRoot` in a seamless way. The lists of problems for `FindMinimum` and `FindRoot` are in `$FindMinimumProblems` and `$FindRootProblems`, respectively, and a problem can be accessed using `GetFindMinimumProblem` and `GetFindRootProblem`.

`$FindMinimumProblems`	list of problems that are appropriate for `FindMinimum`
`GetFindMinimumProblem[`*prob*`]`	get the problem *prob* using the default size and starting values in a `FindMinimumProblem` data structure
`GetFindMinimumProblem[`*prob*`,{`*n*`,`*m*`}]`	get the problem *prob* with *n* variables such that it is a sum of *m* squares in a `FindMinimumProblem` data structure
`GetFindMinimumProblem[`*prob*`,`*size*`,`*start*`]`	get the problem *prob* with given *size* and starting value *start* in a `FindMinimumProblem` data structure
`FindMinimumProblem[`*f*`,`*vars*`,`*opts*`,`*prob*`,`*size*`]`	a data structure that contains a minimization problem to be solved by `FindMinimum`

Accessing `FindMinimum` problems.

`$FindRootProblems`	list of problems that are appropriate for `FindRoot`
`GetFindRootProblem[`*prob*`]`	get the problem *prob* using the default size and starting values in a `FindRootProblem` data structure
`GetFindRootProblem[`*prob*`,`*n*`]`	get the problem *prob* with *n* variables (and *n* equations) in a `FindRootProblem` data structure
`GetFindRootProblem[`*prob*`,`*n*`,`*start*`]`	get the problem *prob* with size *n* and starting value *start* in a `FindRootProblem` data structure
`FindRootProblem[`*f*`,`*vars*`,`*opts*`,`*prob*`,`*size*`]`	a data structure that contains a minimization problem to be solved by `FindRoot`

Accessing `FindRoot` problems.

`GetFindMinimumProblem` and `GetFindRootProblem` are both pass options to be used by other commands. They also accept the option `Variables ->` *vars* which is used to specify what variables to use for the problems.

option name	*default value*	
`Variables`	`X#&`	a function that is applied to the integers $1, \ldots n$ to generate the variables for a problem with n variables or a list of length n containing the variables

Specifying variable names.

This loads the package.

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

This gets the Beale problem in a `FindMinimumProblem` data structure.

```
In[2]:= beale = GetFindMinimumProblem[Beale]
```

$$\text{Out[2]= FindMinimumProblem}\left[\left(\frac{3}{2} - X_1 (1 - X_2)\right)^2 + \left(\frac{9}{4} - X_1 \left(1 - X_2^2\right)\right)^2 + \left(\frac{21}{8} - X_1 \left(1 - X_2^3\right)\right)^2, \{\{X_1, 1.\}, \{X_2, 1.\}\}, \{\}, \text{Beale}, \{2, 3\}\right]$$

This gets the Powell singular function problem in a `FindRootProblem` data structure.

```
In[3]:= ps = GetFindRootProblem[PowellSingular, Variables → {x, y, z, w}]
```

$$\text{Out[3]= FindRootProblem}\left[\left\{x + 10\, y, \sqrt{5}\, (-w + z), (y - 2\, z)^2, \sqrt{10}\, (-w + x)^2\right\}, \{\{x, 3.\}, \{y, -1\}, \{z, 0.\}, \{w, 1.\}\}, \{\}, \text{PowellSingular}, \{4, 4\}\right]$$

Once you have a `FindMinimumProblem` or `FindRootProblem` object, in addition to simply solving the problem, there are various tests that you can run.

`ProblemSolve[`*p*`,`*opts*`]`	solve the problem in *p*, giving the same output as `FindMinimum` or `FindRoot`
`ProblemStatistics[`*p*`,`*opts*`]`	solve the problem, giving a list `{`*sol*`, `*stats*`}`, where *sol* is the output of `ProblemSolve[`*p*`]` and *evals* is a list of rules indicating the number of steps and evaluations used
`ProblemTime[`*p*`,`*opts*`]`	solve the problem giving a list `{`*sol*`, Time -> `*time*`}`, where *sol* is the output of `ProblemSolve[`*p*`]` and *time* is time taken to solve the problem; if *time* is less than a second, the problem will be solved multiple times to get an average timing
`ProblemTest[`*p*`,`*opts*`]`	solve the problem, giving a list of rules including the step and evaluation statistics and time from `ProblemStatistics[`*p*`]` and `ProblemTime[`*p*`]` along with rules indicating the accuracy and precision of the solution as compared with a reference solution
`FindMinimumPlot[`*p*`,`*opts*`]`	plot the steps and evaluation points for solving a `FindMinimumProblem` *p*
`FindRootPlot[`*p*`,`*opts*`]`	plot the steps and evaluation points for solving a `FindRootProblem` *p*

Operations with `FindMinimumProblem` and `FindRootProblem` data objects.

Any of the previous commands shown can take options that are passed on directly to `FindMinimum` or `FindRoot` and override any options for these functions which may have been specified when the problem was set up.

This uses `FindRoot` to solve the Powell singular function problem and gives the root.

```
In[4]:= ProblemSolve[ps]

Out[4]= {x → 8.86974×10^-9, y → -8.86974×10^-10, z → 1.41916×10^-9, w → 1.41916×10^-9}
```

This does the same as the previous example, but includes statistics on steps and evaluations required.

```
In[5]:= ProblemStatistics[ps]

Out[5]= {x → 8.86974×10^-9, y → -8.86974×10^-10, z → 1.41916×10^-9,
  w → 1.41916×10^-9, {Steps → 28, Function → 29, Jacobian → 28}}
```

This uses `FindMinimum` to solve the Beale problem and averages the timing over several trials to get the average time it takes to solve the problem.

```
In[6]:= ProblemTime[beale]

Out[6]= {{2.63792×10^-19, {X1 → 3., X2 → 0.5}}, Time → 0.00201428 Second}
```

This uses `FindMinimum` to solve the Beale problem, compares the result with a reference solution, and gives a list of rules indicating the results of the test.

```
In[7]:= ProblemTest[beale]

Out[7]= {FunctionAccuracy → 18.5787, FunctionPrecision → Indeterminate,
  SpatialAccuracy → 9.7438, SpatialPrecision → 9.85325,
  Time → 0.00202963 Second, Steps → 6, Residual → 8, Jacobian → 7, Messages → {}}
```

`ProblemTest` gives a way to easily compare two different methods for the same problem.

This uses `FindMinimum` to solve the Beale problem using "Newton's" method, compares the result with a reference solution, and gives a list of rules indicating the results of the test.

```
In[8]:= ProblemTest[beale, Method -> "Newton"]

Out[8]= {FunctionAccuracy → 25.5581, FunctionPrecision → Indeterminate,
  SpatialAccuracy → 12.384, SpatialPrecision → 12.6444, Time → 0.00297526 Second,
  Steps → 8, Function → 9, Gradient → 9, Hessian → 9, Messages → {}}
```

Most of the rules returned by these functions are self-explanatory, but a few require some description. Here is a table clarifying those rules.

`"FunctionAccuracy"`	the accuracy of the function value `-Log[10, `‖*error in f*‖`]`
`"FunctionPrecision"`	the precision of the function value `-Log[10, `‖*relative error in f*‖`]`
`"SpatialAccuracy"`	the accuracy in the position of the minimizer or root `-Log[10, `‖*error in x*‖`]`
`"SpatialPrecision"`	the precision in the position of the minimizer or root `-Log[10, `‖*relative error in x*‖`]`
`"Messages"`	a list of messages issued during the solution of the problem

A very useful comparison is to see how a list of methods affect a particular problem. This is easy to do by setting up a `FindMinimumProblem` object and mapping a problem test over a list of methods.

This gets the Chebyquad problem. The output has been abbreviated to save space.

```
In[9]:= Short[cq = GetFindMinimumProblem[Chebyquad], 5]
```

Out[9]//Short=

$$\text{FindMinimumProblem}\Big[\frac{1}{81}(-9+2X_1+2X_2+2X_3+2X_4+2X_5+2X_6+2X_7+2X_8+2X_9)^2+\frac{1}{81}\left(-3(-1+2X_1)+4(-1+2X_1)^3-3(-1+2X_2)+\ll 21\gg+4(-1+2X_9)^3\right)^2+\frac{1}{81}\left(\ll 35\gg+16\ll 1\gg^5\right)^2+\frac{1}{81}(\ll 1\gg)^2+\frac{1}{81}\ll 1\gg^2+(\ll 1\gg+\ll 1\gg)^2+\left(\frac{1}{15}+\frac{1}{9}(\ll 1\gg)\right)^2+\left(\frac{1}{35}+\frac{1}{9}\left(-9+\ll 35\gg+32(-1+\ll 1\gg)^6\right)\right)^2+\left(\frac{1}{63}+\frac{1}{9}\left(9-32(-1+2X_1)^2+\ll 51\gg+128(-1+2X_9)^8\right)\right)^2,\ \ll 3\gg,\ \{9,9\}\Big]$$

Here is a list of possible methods.

```
In[10]:= methods = {Automatic, "QuasiNewton", {"QuasiNewton", "StepMemory" → 10},
    "Newton", {"Newton", "StepControl" -> "TrustRegion"}, "ConjugateGradient"};
```

This makes a table comparing the different methods in terms of accuracy and computation time.

```
In[11]:= TableForm[Map[Join[{#}, {"Time", "FunctionAccuracy", "SpatialAccuracy"} /.
      ProblemTest[cq, Method → #]] &, methods]]
```

Out[11]//TableForm=

Automatic	0.0288897	20.0663	9.94666
QuasiNewton	0.0317216	17.1785	8.3777
QuasiNewton StepMemory → 10	0.0323488	16.4119	7.47304
Newton	0.0769076	20.025	9.34314
Newton StepControl → TrustRegion	0.0761128	21.8281	10.6614
ConjugateGradient	0.0388904	15.7931	7.72219

It is possible to generate tables of how a particular method affects a variety of problems by mapping over the names in `$FindMinimumProblems` or `$FindRootProblems`.

This sets up a function that tests a problem with `FindMinimum` using its default settings except with a large setting for `MaxIterations` so that the default (Levenberg-Marquardt) method can run to convergence.

In[12]:=
```
TestDefault[problem_] := Join[{"Name" → problem},
  ProblemTest[GetFindMinimumProblem[problem, MaxIterations → 1000]]]
```

This makes a table showing some of the results from testing all the problems in `$FindMinimumProblems`. It may take several minutes to run.

In[13]:=
```
TableForm[Map[({"Name", "Time", "Residual", "Jacobian", "FunctionAccuracy",
     "SpatialAccuracy"} /. TestDefault[#]) &, $FindMinimumProblems]]
```

Out[13]//TableForm=

Rosenbrock	0.00284034	21	16	15.9546	15.9546
FreudensteinRoth	0.00442559	35	17	14.1484	8.4797
PowellBadlyScaled	0.00276841	18	17	29.9092	12.4303
BrownBadlyScaled	0.00182188	10	10	20.5345	16.2673
Beale	0.00199867	8	7	18.5787	9.7438
JennrichSampson	0.00828054	34	20	13.3703	8.87261
HelicalValley	0.00218182	11	9	32.0055	17.2046
Bard	0.00673732	7	7	16.9157	8.00751
Gauss	0.00786546	3	3	21.1019	11.0733
Meyer	0.0264677	126	116	11.5089	9.95814
Gulf	0.0120229	89	17	31.109	13.543
Box3D	0.00715045	6	6	18.9447	8.68579
PowellSingular	0.0034851	28	28	30.3044	7.73816
Wood	0.00791268	69	64	23.5366	13.0536
KowalikOsborne	0.010429	36	35	18.6639	8.33507
BrownDennis	0.0899279	412	375	9.13811	6.11409
Osborne1	0.0224698	20	17	17.4797	9.3597
BiggsExp6	0.0231614	50	36	30.2266	14.4925
Osborne2	0.121583	20	17	17.1587	7.90304
Watson	0.0736547	11	9	18.8178	6.68865
ExtendedRosenbrock	0.0954113	21	16	29.9092	15.9546
ExtendedPowell	0.123236	27	27	29.9092	7.21075
PenaltyFunctionI	0.0249084	117	94	18.1356	6.96613
PenaltyFunctionII	0.0271926	109	72	15.9546	7.62089
VariablyDimensionedFunction	0.130756	17	17	15.9546	15.9546
TrigonometricFunction	0.00774007	7	7	28.0238	14.6546
BrownAlmostLinear	0.00557332	14	13	29.1488	0.668059
DiscreteBoundaryValue	0.00547087	4	4	30.5195	14.2959
DiscreteIntegralEquation	0.0105878	4	4	29.3985	14.8825
BroydenTridiagonal	0.00479374	5	5	17.9475	9.44685
BroydenBanded	0.00825598	8	7	28.0567	15.503
LinearFullRank	0.00370734	2	2	14.7505	14.6348
LinearRank1	0.00938284	55	2	15.0515	ERROR
LinearRank1Z	0.00742234	37	2	15.0515	ERROR
Chebyquad	0.0280148	11	9	20.0663	9.94666

The two cases where the spatial accuracy is shown as `ERROR` are for linear problems, which do not have an isolated minimizer. The one case, which has a spatial accuracy that is quite poor, has multiple minimizers, and the method goes to a different minimum than the reference one. Many of these functions have multiple local minima, so be aware that the error may be reported as large only because a method went to a different minimum than the reference one.

References

[AN96] Adams, L. and J. L. Nazareth, eds. *Linear and Nonlinear Conjugate Gradient-Related Methods*. SIAM, 1996.

[Br02] Brent, R. P. *Algorithms for Minimization without Derivatives*. Dover, 2002 (Original volume 1973).

[DS96] Dennis, J. E. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization*. SIAM, 1996 (Original volume 1983).

[GMW81] Gill, P. E., W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1981.

[MW93] More, J. J. and S. J. Wright. *Optimization Software Guide*. SIAM, 1993.

[MT94] More, J. J. and D. J. Thuente. "Line Search Algorithms with Guaranteed Sufficient Decrease." *ACM Transactions on Mathematical Software* 20, no. 3 (1994): 286-307.

[MGH81] More, J. J., B. S. Garbow, and K. E. Hillstrom. "Testing Unconstrained Optimization Software." *ACM Transactions on Mathematical Software* 7, no. 1 (1981): 17-41.

[NW99] Nocedal, J. and S. J. Wright. *Numerical Optimization*. Springer, 1999.

[PTVF92] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*, 2nd ed. Cambridge University Pressn, 1992.

[Rhein98] Rheinboldt, W. C. *Methods for Solving Systems of Nonlinear Equations*. SIAM, 1998 (Original volume 1974).