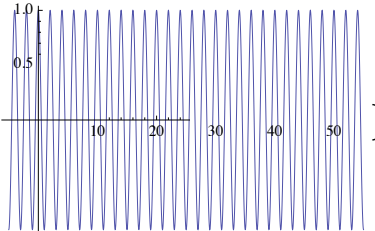


This is an example of a problem where the Newton step is very large because the starting point is at a position where the Jacobian (derivative) is nearly singular. The step size is (not severely) limited by the option.

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

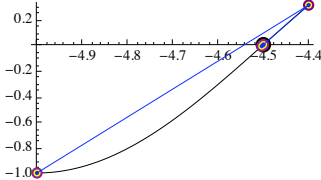
```
Out[3]= {{x → 2.5}, {Steps → 2, Residual → 5, Jacobian → 2},
```



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}}]
```

```
Out[4]= {{x → -4.5}, {Steps → 5, Residual → 5, Jacobian → 5},
```



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.

<i>option name</i>	<i>default value</i>	
"Method"	Automatic	method to use for executing the line search; can be Automatic, "MoreThuente", "Backtracking", or "Brent"
"CurvatureFactor"	Automatic	factor $\eta$ in the Wolfe conditions, between 0 and 1; smaller values of $\eta$ result in a more exact line search
"DecreaseFactor"	1/10 000	factor $\mu$ in the Wolfe conditions, between 0 and $\eta$
"MaxRelativeStepSize"	10	largest step that will be taken relative to the norm of the current search point, can be any positive number or $\infty$ 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]
```

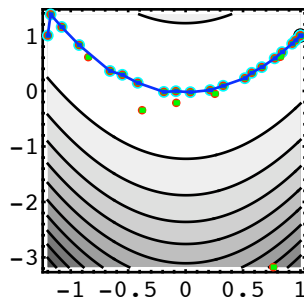
```
Out[5]= FindMinimumProblem[(1 - X1)2 + 100 (-X12 + X2)2, {{X1, -1.2}, {X2, 1.}}, {}, Rosenbrock, {2, 2}]
```

## 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 × 10-18, {X1 -> 1., X2 -> 1.}},  
{Steps -> 22, Function -> 29, Gradient -> 29, Hessian -> 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.

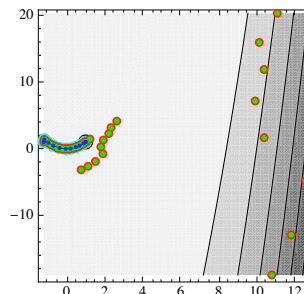
Decreasing the curvature factor, which means that the line search ends nearer to the exact minimum, decreases the number of steps taken by Newton's method but increases the total number of function and gradient evaluations.

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.

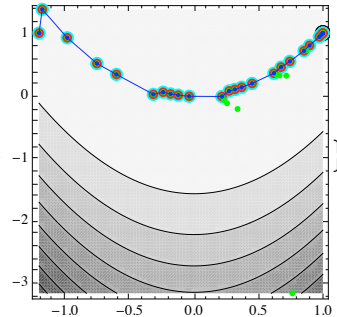
### Backtracking

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 × 10-30, {X1 -> 1., X2 -> 1.}},
```

```
{Steps -> 25, Function -> 34, Gradient -> 26, Hessian -> 25},
```



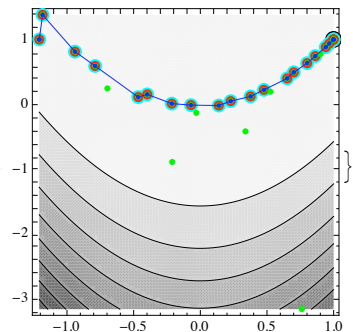
Each backtracking step is taken by doing a polynomial interpolation and finding the minimum point for the interpolant. This point  $\alpha_k$  is used as long as it lies between  $c_1 \alpha_{k-1}$  and  $c_2 \alpha_{k-1}$ , where  $\alpha_{k-1}$  is the previous value of the parameter  $\alpha$  and  $0 < c_1 \leq c_2 < 1$ . By default,  $c_1 = 0.1$  and  $c_2 = 0.5$ , but they can be controlled by the method option "BacktrackFactors"  $\rightarrow \{c_1, c_2\}$ . 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 × 10-21, {X1 -> 1., X2 -> 1.}},
```

```
{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  $\rightarrow$  "Backtracking".



## Brent

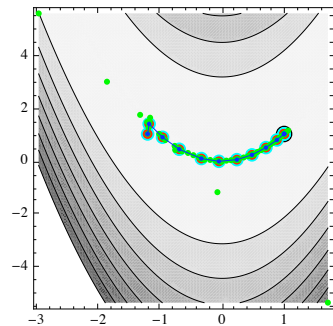
This uses the derivative-free univariate method of Brent [Br02] for the line search. It attempts to find the minimum of  $\phi \alpha$  to within tolerances, regardless of the decrease and curvature factors. In effect, it has two phases. First, it tries to bracket the root, then it uses "Brent's" combined 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  $\alpha$ . 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

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

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  $\|Dp\| \leq \Delta$ .  $D$  is a diagonal scaling (often taken from the diagonal of the approximate Hessian) and  $\Delta$  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  $\rho_k$  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  $\rho_k$  is too small, the region is decreased in size. When  $\rho_k$  is below a threshold,  $\eta$ , the step is rejected and recomputed. You can control this threshold with the method option "AcceptableStepRatio"  $\rightarrow \eta$ . Typically the value of  $\eta$  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  $\eta$  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  $\Delta$  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  $\Delta_0$  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.

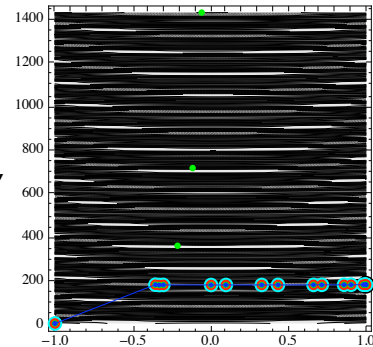
```
In[1]:= << Optimization`UnconstrainedProblems`
```

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}},
```



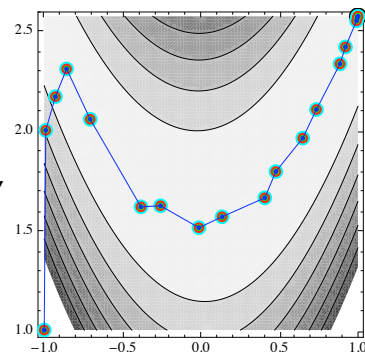
The plot looks quite bad because the search has extended over such a large region that the fine structure of the function cannot really be seen on that scale.

This shows the steps and evaluations for the same function, but with a restricted initial trust region radius  $\Delta_0$ . 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"  $\rightarrow \Delta_{max}$  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

`\Unconstrained Problems`` 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.

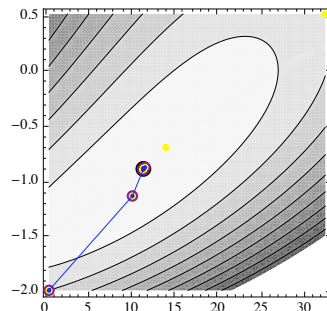
```
In[5]:= FindMinimumPlot[pfr]
```

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}},
```

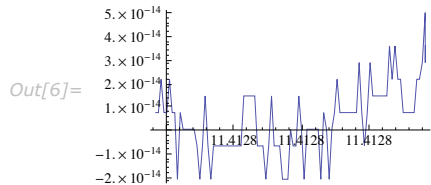
```
{Steps → 16, Residual → 35, Jacobian → 17}},
```



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[{ $\epsilon = 10^{-7}$ , 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 -  $\epsilon$ , x1f +  $\epsilon$ }]]
```

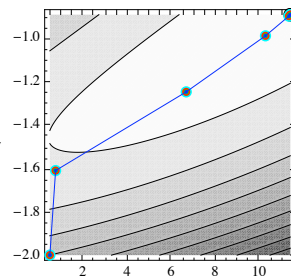


The plot along one direction makes it fairly clear why no more improvement is possible. Part of the reason the Levenberg-Marquardt method gets into trouble in this situation is that convergence is relatively slow because the residual is nonzero at the minimum. With "Newton's" method, the convergence 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},
```



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

option name	default value	
"AcceptableStepRatio"	1/10 000	the threshold $\eta$ , such that when the actual to prediction reduction $\rho_k \geq \eta$ , the search is moved to the computed step
"MaxScaledStepSize"	$\infty$	the value $\Delta_{max}$ , such that the trust region size $\Delta_k < \Delta_{max}$ for all steps
"StartingScaledStepSize"	Automatic	the initial trust region size $\Delta_0$

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.

<code>Automatic</code>	find a symbolic derivative for the function and use finite difference approximations if a symbolic derivative cannot be found
<code>Symbolic</code>	same as <code>Automatic</code> , but gives a warning message if finite differences are to be used
<code>FiniteDifference</code>	use finite differences to approximate the derivative
<code>expression</code>	use the given <i>expression</i> 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.

<i>option name</i>	<i>default value</i>	
"EvaluationMonitor"	None	expression to evaluate with local values of the variables every time the derivative is evaluated, usually specified with <code>:&gt;</code> instead of <code>-&gt;</code> to prevent symbolic evaluation
"Sparse"	<code>Automatic</code>	sparse structure for the derivative; can be <code>Automatic</code> , <code>True</code> , <code>False</code> , or a pattern <code>SparseArray</code> 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::synd:

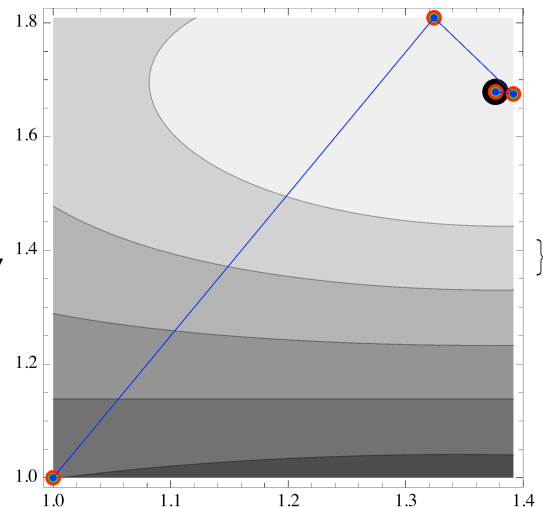
Unable to automatically compute the symbolic derivative of  $f[x, y]$  with respect to the arguments  $\{x, y\}$ . Numerical approximations to derivatives will be used instead. >>

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},
```



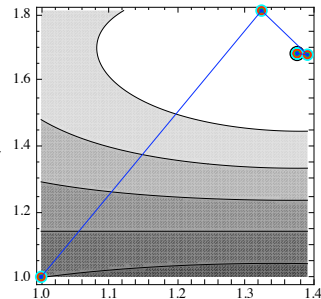
The following describes how you can use the gradient option to specify the derivative.

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},
```



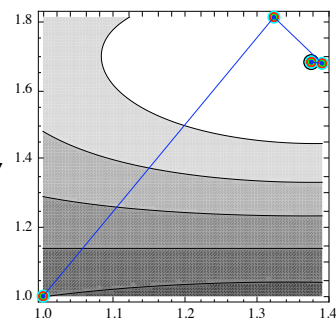
Symbolic derivatives are not always available. If you need extra accuracy from finite differences, you can increase the difference order from the default of 1 at the cost of extra function evaluations.

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.

<code>FindMinimum [f, vars]</code>	find a local minimum of $f$ with respect to the variables given in $vars$
<code>FindMaximum [f, vars]</code>	find a local maximum of $f$ with respect to the variables given in $vars$
<code>FindRoot [f, vars]</code>	find a root $f = 0$ with respect to the variables given in $vars$
<code>FindRoot [eqns, vars]</code>	find a root of the equations $eqns$ with respect to the variables given in $vars$
<code>FindFit [data, expr, pars, vars]</code>	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.

<code>{var, st}</code>	variable $var$ has starting value $st$
<code>{var, st<sub>1</sub>, st<sub>2</sub>}</code>	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
<code>{var, st, rl, ru}</code>	variable $var$ has starting value $st$ ; the search will be terminated when the value of $var$ goes outside of the interval $[rl, ru]$
<code>{var, st<sub>1</sub>, st<sub>2</sub>, rl, ru}</code>	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 =  $\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix};$ 
```

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  $\lambda$  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[{{x11, x12}, {x21, x22}}]` gives a symbolic representation of the function for scalar  $x_{11}, \dots$

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 → {{0.9999996, 0.9999996, 0.9999996, 0.9999996, 0.9999996, 0.9999996, 0.9999996, 0.9999996,
0.9999996, 0.9999996}, {0.9999991, 0.9999991, 0.9999991, 0.9999991,
0.9999991, 0.9999991, 0.9999991, 0.9999991, 0.9999991, 0.9999991}}}}
```

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.

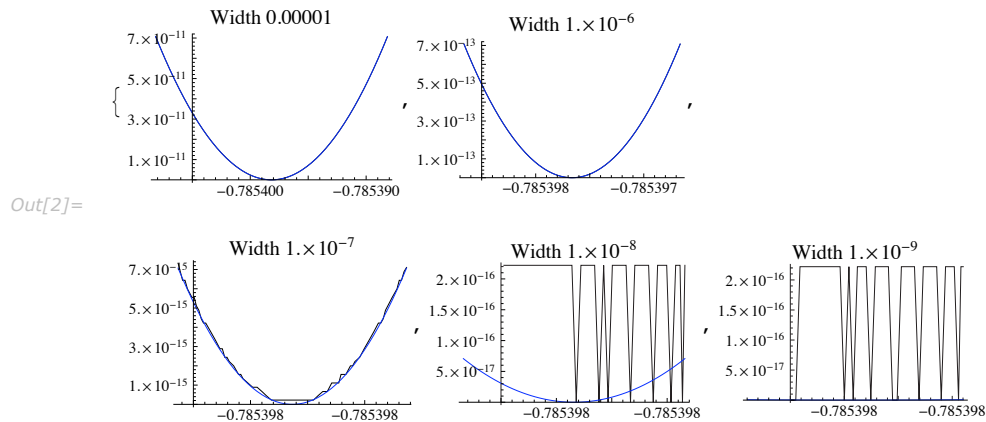
Tolerances affect how close a search will try to get to a root or local minimum before terminating the search. Assuming that the function itself has some error (as is typical when it is computed with numerical values), it is not typically possible to locate the position of a minimum much better than 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[{ $\epsilon = 10.^{-k}$ },
  Show[{Plot[Sin[x] - Cos[x] + Sqrt[2], {x, - $\pi/4 - \epsilon$ , - $\pi/4 + \epsilon$ }, PlotStyle -> Black],
  Plot[Sin[x] - Cos[x] + Sqrt[2], {x, - $\pi/4 - \epsilon$ , - $\pi/4 + \epsilon$ }, PlotStyle -> Blue,
  WorkingPrecision -> 100}], PlotLabel -> Row[{"Width ",  $\epsilon$ ]}], {k, 5, 9}]
```



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  $\rightarrow ag$  and PrecisionGoal  $\rightarrow pg$ , this defines tolerances  $tol_a = 10^{-ag}$  and  $tol_r = 10^{-pg}$ .



Given  $\text{tol}_a$  and  $\text{tol}_r$ , `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  $\infty$  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  $\infty$  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.

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

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]
```

```
Out[12]//Short= FindMinimumProblem[ $\left( \left( -e^{1/5} + X_1 + \frac{X_2}{5} \right)^2 + \left( -\cos\left[\frac{1}{5}\right] + X_3 + \sin\left[\frac{1}{5}\right] X_4 \right)^2 \right)^2 +$   

 $\left( \left( -e^{2/5} + X_1 + \frac{2 X_2}{5} \right)^2 + \left( -\cos\left[\frac{2}{5}\right] + X_3 + \sin\left[\frac{2}{5}\right] X_4 \right)^2 \right)^2 +$   

 $\llbracket 17 \rrbracket + \left( \left( -e^4 + X_1 + 4 X_2 \right)^2 + \left( -\cos[4] + X_3 + \sin[4] X_4 \right)^2 \right)^2,$   

 $\{\{X_1, 25.\}, \{X_2, 5.\}, \{X_3, -5.\}, \{X_4, -1.\}\}, \{\}, \text{BrownDennis}, \{4, 20\}$ ]
```

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. >>
```

```
Out[13]= {105.443., {X1 -> -7.35071, X2 -> 11.7365, X3 -> -0.60436, X4 -> 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. >>
```

```
Out[44]= {85.822.2, {X1 -> -11.5944, X2 -> 13.2036, X3 -> -0.403439, X4 -> 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 <code>Block</code> and <code>Table</code> , add rules to the variables so that any assignments given to them will not affect your <i>Mathematica</i> 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). <b>Note:</b> this is a change which was instituted in <i>Mathematica</i> 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 <code>PatternTest</code> (?), as in $f[x_? \text{NumberQ}] := \text{definition}.$
Preprocess the function	analyze the function to help determine the algorithm to use (e.g., sum of squares $\rightarrow$ Levenberg-Marquardt); optimize and compile the function for faster numerical evaluation if possible: for <code>FindRoot</code> 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::nnum:
  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::nnum:
  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, Pi}]
```

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.

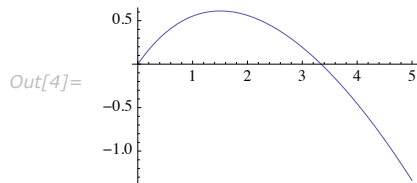
This defines a function that returns the value `x[1]` as a function of a numerical value for `x' [t]` at `t = -1`.

```
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}]
```



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`.

<code>FindMinimumPlot[f, {x, xst}, opts]</code>	plot the steps and the points at which the function $f$ and any of its derivatives that were evaluated in <code>FindMinimum[f, {x, xst}]</code> superimposed on a plot of $f$ versus $x$ ; <i>opts</i> may include options from both <code>FindMinimum</code> and <code>Plot</code>
<code>FindMinimumPlot[f, {{x, xst}, {y, yst}}, opts]</code>	plot the steps and the points at which the function $f$ and any of its derivatives that were evaluated in <code>FindMinimum[f, {{x, xst}, {y, yst}}]</code> superimposed on a contour plot of $f$ as a function of $x$ and $y$ ; <i>opts</i> may include options from both <code>FindMinimum</code> and <code>ContourPlot</code>
<code>FindRootPlot[f, {x, xst}, opts]</code>	plot the steps and the points at which the function $f$ and any of its derivatives which were evaluated in <code>FindRoot[f, {x, xst}]</code> superimposed on a plot of $f$ versus $x$ ; <i>opts</i> may include options from both <code>FindRoot</code> and <code>Plot</code>
<code>FindRootPlot[f, {{x, xst}, {y, yst}}, opts]</code>	plot the steps and the points at which the function $f$ and any of its derivatives that were evaluated in <code>FindRoot[f, {{x, xst}, {y, yst}}]</code> superimposed on a contour plot of the merit function $f$ as a function of $x$ and $y$ ; <i>opts</i> may include options from both <code>FindRoot</code> and <code>ContourPlot</code>

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`
```

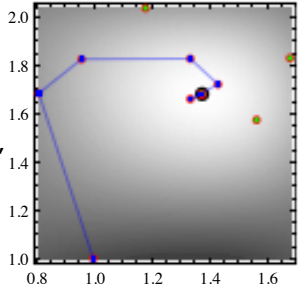
This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function  $\cos(x^2 - 3y) + \sin(x^2 + y^2)$  starting at the point  $\{x, y\} = \{1, 1\}$ . Options are given to `ContourPlot` so that no contour lines are shown and the function value is indicated 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]
```

```
{{- 2.1 x -> 1.37638, y -> 1.67868} ,
```

```
Out[2]=
```

```
{ Steps -> 9, Function -> 13, Gradient -> 13 ,
```



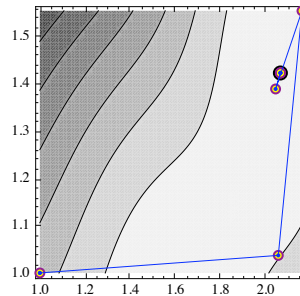
```
}
```

This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function  $(x^2 - 3y)^2 + \sin^2(x^2 + y^2)$  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},
```

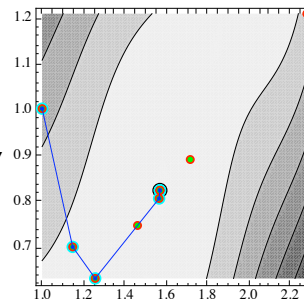


This shows in two dimensions the steps and evaluations used by `FindMinimum` to find a local minimum of the function  $(x^2 - 3y)^2 + \sin^2(x^2 + y^2)$  starting at the point  $\{x, y\} = \{1, 1\}$  using "Newton's" method. Points at which the function, gradient, and Hessian were all evaluated are shown 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}},
```

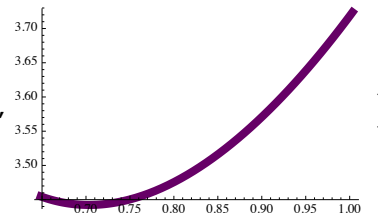
```
{Steps → 6, Function → 10, Gradient → 10, Hessian → 7},
```



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]}
```

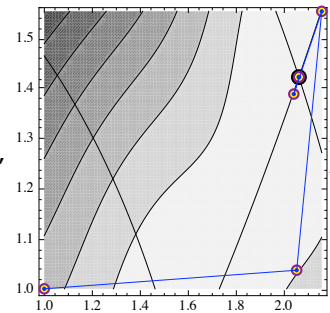
```
Out[5]= {{3.44228, {x -> 0.703467}}, {Steps -> 6, Function -> 14},
```



This shows in two dimensions the steps and evaluations used by `FindRoot` to find a root of the function  $\{x^2 - 3y, \sin(x^2 + y^2)\} = \{0, 0\}$  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  $(x^2 - 3y)^2 + \sin^2(x^2 + y^2)$  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`.

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

Accessing `FindMinimum` problems.

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

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.

<i>option name</i>	<i>default value</i>	
<code>Variables</code>	<code>X::&amp;</code>	a function that is applied to the integers $1, \dots, n$ to generate the variables for a problem with <i>n</i> variables or a list of length <i>n</i> 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]
```

```
Out[2]= FindMinimumProblem[ $\left[\left(\frac{3}{2} - x_1 (1 - x_2)\right)^2 + \left(\frac{9}{4} - x_1 (1 - x_2^2)\right)^2 + \left(\frac{21}{8} - x_1 (1 - x_2^3)\right)^2\right]$ ,  
{x1, 1.}, {x2, 1.}, {}, Beale, {2, 3}]
```

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

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

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

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

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

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 $-\text{Log}[10, \ error\ in\ f\ ]$
"FunctionPrecision"	the precision of the function value $-\text{Log}[10, \ relative\ error\ in\ f\ ]$
"SpatialAccuracy"	the accuracy in the position of the minimizer or root $-\text{Log}[10, \ error\ in\ x\ ]$
"SpatialPrecision"	the precision in the position of the minimizer or root $-\text{Log}[10, \ relative\ error\ in\ x\ ]$
"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= FindMinimumProblem[ $\frac{1}{81} (-9 + 2 X_1 + 2 X_2 + 2 X_3 + 2 X_4 + 2 X_5 + 2 X_6 + 2 X_7 + 2 X_8 + 2 X_9)^2 +$   
 $\frac{1}{81} (-3 (-1 + 2 X_1) + 4 (-1 + 2 X_1)^3 - 3 (-1 + 2 X_2) + \ll 21 \gg + 4 (-1 + 2 X_9)^3)^2 +$   
 $\frac{1}{81} (\ll 35 \gg + 16 \ll 1 \gg^5)^2 + \frac{1}{81} (\ll 1 \gg)^2 + \frac{1}{81} \ll 1 \gg^2 + (\ll 1 \gg + \ll 1 \gg)^2 +$   
 $\left( \frac{1}{15} + \frac{1}{9} (\ll 1 \gg) \right)^2 + \left( \frac{1}{35} + \frac{1}{9} (-9 + \ll 35 \gg + 32 (-1 + \ll 1 \gg)^6) \right)^2 +$   
 $\left( \frac{1}{63} + \frac{1}{9} (9 - 32 (-1 + 2 X_1)^2 + \ll 51 \gg + 128 (-1 + 2 X_9)^8) \right)^2, \ll 3 \gg, \{9, 9\}]$ 
```

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]]  
Automatic 0.0288897 20.0663 9.94666  
QuasiNewton 0.0317216 17.1785 8.3777  
QuasiNewton  
StepMemory → 10 0.0323488 16.4119 7.47304  
Out[11]//TableForm= 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]]
```

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
Out[13]//TableForm= 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. Hillstom. "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 Press, 1992.
- [Rhein98] Rheinboldt, W. C. *Methods for Solving Systems of Nonlinear Equations*. SIAM, 1998 (Original volume 1974).

