GLOBAL OPTIMIZATION 6.0

GLOBAL NONLINEAR OPTIMIZATION USING *MATHEMATICA*

Loehle Enterprises

1258 Windemere Ave.

Naperville, IL 60564

USA

630-527-8402

630-416-9902 fax

info@loehleenterprises.com

www.loehleenterprises.com

Version 6.0

****

DISCLAIMER

****

NOTICE: The software product described in this manual is a research tool. No warranty is offered as to operability of this product. No guarantee can be made that any particular nonlinear equation can be solved. No safety critical or financial decisions should be based on the operation of this software. The manufacturer accepts no liability for losses resulting from its use.

****

Contents

I GETTING STARTED

I.1 Welcome

Welcome to Global Optimization. This package provides a suite of tools for solving nonlinear optimization problems, as well as a variety of other applications such as finding the roots or zeros of a nonanalytic function. The package is easier, simplier, and more robust than most optimization tools, and you will find yourself working more efficiently and more easily than before.

I.2 Registering

Before you continue please register by emailing your name and address with a notice that you have purchased Global Optimization to craigloehl@aol.com. This will enable us to reach you with free updates and other news. The developer does not receive customer information when the package is purchased from Wolfram Research or from distributors. Please include the version of Global Optimization that you purchased and the machine you are using it on.

I.3 Installation

I.3.A Recommended System Capabilities

This package is designed to work with *Mathematica*, which must be installed on your computer. The following are recommended system capabilities for installation:

- Personal computer, 600 MHz or faster
- 200Mb free RAM or more
- *Mathematica* 4.2 or higher installed

I.3.B Installation on a Hard Drive

To install the package on your hard drive on any machine, copy the contents of the disk to the *Mathematica* directory. The file may need to be unzipped. On Mac the file may be a self-extracting archive; double-click it to extract the contents. The go60v.mx file (where v denotes the version) is an encoded *Mathematica* package. This encoded file can not be read and will not work on other platforms. It needs to be placed in the main *Mathematica* directory or in the AddOns/Applications directory. An example notebook file .nb is included for illustration. The .nb file(s) may be placed anywhere.

I.4 A Quick Start and Example

Version 6.0 contains ten functions: GlobalSearch, IntervalMin, GlobalPenaltyFn, MultiStartMin, GlobalMinima, NLRegression, MaxLikelihood, InterchangeMethodMin, TabuSearchMin, and MaxAllocation. The package has been designed for easy use with *Mathematica*. It uses all valid *Mathematica* functions and syntax. The user must have *Mathematica* installed. This manual assumes basic familiarity with *Mathematica*. Input files should be in the form of *Mathematica* notebooks, with a file designation "filename.nb". This manual is executable. Timing examples in this manual are based on a 3.4 Ghz Pentium IV machine using *Mathematica* 6.0.

To begin the execution of a notebook, follow this procedure:

1. Start *Mathematica*.

2. Use the File pull-down menu to Open one of the example notebook files supplied with the disk.

3. Once this notebook is open, pull down the Kernel menu and select Evaluate Notebook.

In the examples, the package is installed with the Get (Get["go60v.mx"] or <<go60v.mx) command, with definition for the function and options given as:

```
In[2]:= Off[Syntax::"spell"];
        Off[Syntax::"spell1"];
        $HistoryLength = 0;
```

then several variables are defined, the function to be evaluated is defined, and then the problem is solved. This process should execute with no errors and should produce output that resembles closely what was in the notebook when you opened it. If not, some error may exist. The basic formats are similar to the *Mathematica* function ConstrainedMin. The following definitions define the functions in the package:

```
In[5]:= ?GlobalMinima
```

GlobalMinima finds the minimum of a constrained or unconstrained nonlinear function of n variables.GlobalMinima[expression,inequalities,{{var1name,lowbound, highbound}..},grid,tolerance,contraction,indifference,options]

In[6]:= **?GlobalSearch**

GlobalSearch finds the minimum of a nonlinear
function of n variables s.t. equality & inequality constraints.
GlobalSearch[expression,inequalities,equalities,{{var1name,lowbound,
highbound}..}, tolerance,options]

In[7]:= **?GlobalPenaltyFn**

GlobalPenaltyFn finds the minimum of a nonlinear
function of n variables s.t. equality & inequality constraints.
GlobalPenaltyFn[expression,inequalities,equalities,{{var1name,lowbound,
highbound}..}, tolerance,options]

In[598]:= **?InterchangeMethodMin**

```
InterchangeMethodMin finds the minimum of a nonlinear
  function of binary 0-1 variables using the Interchange method.
  InterchangeMethodMin[expression,inequalities,{varlist},tolerance,options]
```

In[6]:= **?IntervalMin**

```
IntervalMin finds the minimum of a
  constrained or unconstrained nonlinear function of n
  variables.IntervalMin[expression,inequalities,Null,{Interval},varlist,tolerance
  ,options]
```

In[7]:= **?MaxAllocation**

```
MaxAllocation finds the maximum of
  a nonlinear function of n variables s.t. an equality
  constraint.MaxAllocation[f[var1name,...],rhs,{varlist}..,tolerance,options]
```

In[8]:= **?MaxLikelihood**

```
MaxLikelihood performs maximum likelihood estimation on a
  function.  It has a library of functions optimized for efficiency.
  MaxLikelihood[data,expression,independent_variables,constraints,
  {{var1name,lowbound,highbound}..},tolerance,options]
```

In[9]:= **?MultiStartMin**

```
MultiStartMin finds the minimum of a nonlinear function
  of n variables s.t. equality and inequality constraints.
  MultiStartMin[expression,inequalities,equalities,{{var1name,lowbound,highbound}
  ..}, tolerance,options]
```

```
In[10]:= ?NLRegression

         NLRegression performs nonlinear regression with or without
           constraints.NLRegression[data,expression,independent_variables,inequalities,
           equalities,{{var1name,lowbound,highbound}..},tolerance,options]

In[11]:= ?TabuSearchMin

         TabuSearchMin finds the minimum of a nonlinear function of discrete
           variables.TabuSearchMin[expression,inequalities,{varlist},tolerance,options]
```

The following options are available for one or more functions. ExactEqualities and PenaltyMethod are used for GlobalPenaltyFn.

```
In[12]:= ?CompileOption

         If False, uses uncompiled version of user function.

In[175]:=
     ?EvaluateObj

         Evaluate objective function if True.

In[173]:=
     ?ExactEqualities

         Solve equalities with exact method.  Is slower. Default False.

In[174]:=
     ?PenaltyMethod

         Solve equalities with penalty method.  Default False.

In[14]:= ?FastStepping

         If True, uses fast stepping method for MultiStartMin.

In[599]:=
     ?MaxIterations

         Maximum iterations allowed.  Default 10000. More...

In[16]:= ?SensitivityPlots

         Prints sensitivity plots if True.

In[17]:= ?ShowProgress

         If True, prints intermediate results.

In[18]:= ?SimplifyOption

         Attempt to simplify objective function if True.

In[19]:= ?Starts

         Number of starting points for search.  Default minimum of 3.
```

```
In[20]:= ?StartsList

         User input of starting values.

In[21]:= ?TabuListLength

         Length of Tabu list.  Default=100.

In[22]:= ?UserMemory

         Number of megabytes available to GlobalMinima for computations.

In[23]:= ?UserResiduals

         Regression option for passing a user function for residuals.

In[24]:= ?Weights

         Weight assigned to data point i during
           regression analysis.  Does not affect sum of squares.
```

GlobalSearch[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (1)

GlobalPenaltyFn[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (2)

MultiStartMin[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (3)

InterchangeMethodMin[expression,inequalities,{varlist},tolerance,options] (4)

IntervalMin[expression,inequalities,Null,{InitialPoint__Interval},varlist,tolerance,options] (5)

TabuSearchMin[expression,inequalities,{varlist},tolerance,options] (6)

GlobalMinima[expression,inequalities,{{var1name,lowbound,highbound}..},grid,tolerance,contraction,indifference,options] (7)

MaxAllocation[f[var1name,...],rhs,{varlist}..,tolerance,options] (8)

NLRegression[data,expression,independent_variables,inequalities,{{var1name,lowbound,highbound}..},tolerance,options] (9)

MaxLikelihood[data,expression,independent_variables,inequalities,{{var1name,lowbound,highbound}..},tolerance,options] (10)

Where "**expression**" is the equation or function name for the function to be minimized, "**inequalities**" is a list of inequality constraints, and "**equalities**" is a list of equalities. Speed of execution is enhanced with *Mathematica* versions 5.1 and higher. The *Mathematica* kernel and the executable program need not reside on the same machine.

The functions are each designed for a specific type of problem. For linear problems, the user should refer to ConstrainedMin (see *Mathematica* documentation), although the functions in this package will solve linear problems. For smooth, unconstrained nonlinear problems, the user should try FindMinimum, which is faster for such problems. The functions in this package are designed for nonlinear problems with local minima, multiple minima, and/or linear or nonlinear constraints.

The function GlobalSearch is a multiple-start generalized hill-climbing algorithm designed to work with or without constraints. It is robust to noisy functions and local minima. It can handle large problems (200+ variables). It can handle linear and nonlinear equality and inequality constraints, which are assumed to be analytic. GlobalSearch is the general purpose optimizer in the package, and is the engine used for the NLRegression and MaxLikelihood functions.

The function IntervalMin is a general minimizer using Interval methods. It can handle inequality constraints, which must be analytic. It is robust to local minima, but is slower than other methods.

The function GlobalPenaltyFn is a multiple-start generalized hill-climbing algorithm designed to work with or without constraints. It handles the special case of constraints that are not Solvable for any of the variables, or that are black box. It is robust to noisy functions and local minima. It can handle large problems (200+ variables). For equality constraints, it has three options. The default attempts to work with equality constraints analytically. PenaltyMethod when True uses a penalty method. This option is useful when equality constraints are extremely complicated or non-analytic. ExactEqualities when True forces the search to always stay on the equality lines.

The function MultiStartMin is a restricted case version of GlobalSearch designed to work for problems with integer or discrete variables or that are highly nonlinear. To solve this subset of problems, it handles constraints differently and is thus slower than GlobalSearch, particularly as problems get bigger. Thus for problems with more than 15 variables, one should use GlobalSearch. It is a multiple-start generalized hill-climbing algorithm designed to work with or without constraints. It is robust to noisy functions and local minima. It can handle linear and nonlinear inequality constraints. Equality constraints must be analytic, but inequality constraints need not be and can be black box or logical. Objective functions may contain any combination of real, integer, and discrete variables.

The function GlobalMinima is designed for problems with many true or local minima or for which a region rather than a point better describes the optimum solution. It can handle linear and nonlinear inequality constraints. It is limited to smaller problems (<14 variables) but is very robust to noise and false minima. For smaller problems (<3 variables), it may be faster than MultiStartMin or GlobalSearch. GlobalMinima can find solution regions, whereas MultiStartMin and GlobalSearch are not efficient at this task.

The function MaxAllocation is designed for allocation problems, such as arise in finance. In an allocation problem, a fixed quantity, such as an investment sum, is to be distributed to a portfolio. All variables must be nonnegative and the sum of all investments must equal the total available for investing. This creates a nonlinear problem with a single equality constraint.

The function InterchangeMethodMin is designed for 0-1 integer problems such as arise in networks, transportation, and scheduling. The operation of this routine is faster than for problems with continuous variables of the same size. The function can solve vehicle routing/traveling salesman, minimum spanning tree, and capital allocation problems, among others.

The function TabuSearchMin is designed for 0-1 integer problems such as arise in networks, transportation, and scheduling. The operation of this routine is faster than for problems with continuous variables of the same size. The function can solve vehicle routing/traveling salesman,minimum spanning tree, and capital allocation problems, among others.

The function NLRegression solves nonlinear regression problems. It is particularly designed for noisy problems or those requiring constraints to achieve a good fit. L1 and L2 norms are allowed. Least-squares and chi-square options are also available. Sensitivity plots and confidence intervals on the parameters are provided as output.

The function MaxLikelihood solves maximum likelihood estimation problems using Log-likelihood estimation. A library of common distributions is provided and fit statistics are computed. Summary statistics are computed.

II TIPS FOR PERFORMANCE

A key factor for numerical optimization is the time taken to compute the objective function. This is critical because the objective function may be called thousands of times to hundreds of thousands of times. This section discusses tricks to achieve better performance.

II.1 Improving Performance

Some general tips include the removal of the reading of external files from the user function, computing constant expressions once and then using the result, and avoiding logical operations that will prevent Compile from being used. A key is to generate a function once as an explicit expression, and then pass this in to the package. If many steps are involved in generating a user function and these must be performed every time the function is called, this can be very expensive. It often happens that parts of an expression involve constants. If these can be evaluated up front, the savings can be enormous. In the following example, the function "f" evaluates numerically when Evaluate is executed during Function creation. In function "g" functions like Sin and Tan are evaluated each time the function is called:

```
In[8]:= ClearAll[f, g]

In[9]:= x = {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10};

In[10]:= f = Function[Evaluate[x], Evaluate[
            Sum[Sin[Pi / 2.1] * x[[i]] + Tan[19.4 * Pi] / x[[i]] + Cos[.1] + Sqrt[99], {i, 1, 10}]]]
```

Out[10]=

$$\text{Function}\left[\{x1, x2, x3, x4, x5, x6, x7, x8, x9, x10\},\right.$$
$$109.449 + \frac{3.07768}{x1} + 0.997204\, x1 + \frac{3.07768}{x10} + 0.997204\, x10 + \frac{3.07768}{x2} + 0.997204\, x2 +$$
$$\frac{3.07768}{x3} + 0.997204\, x3 + \frac{3.07768}{x4} + 0.997204\, x4 + \frac{3.07768}{x5} + 0.997204\, x5 + \frac{3.07768}{x6} +$$
$$\left. 0.997204\, x6 + \frac{3.07768}{x7} + 0.997204\, x7 + \frac{3.07768}{x8} + 0.997204\, x8 + \frac{3.07768}{x9} + 0.997204\, x9\right]$$

```
In[11]:= g[p_] := Module[{d}, {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10} = p;
            r = Sum[Sin[Pi / 2.1] * x[[i]] + Tan[19.4 * Pi] / x[[i]] + Cos[.1] + Sqrt[99], {i, 1, 10}];
            Return[r]]

In[12]:= dat = Table[i, {i, 1, 10}]

Out[12]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[15]:= Apply[f, dat]

Out[15]= 173.309
```

```
In[16]:= g[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]

Out[16]= 173.309

In[17]:= Do[Apply[f, dat], {i, 1, 500}] // Timing

Out[17]= {0.031, Null}

In[18]:= Do[g[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}], {i, 1, 500}] // Timing

Out[18]= {0.297, Null}
```

Because the constants in f were Evaluated before execution (during Function definition), f was very fast to execute, even without Compile. In contrast, g, which computes Sin etc. every time it is called, took 46 times longer to compute.

II.2 The Compile Function

The *Mathematica* function Compile is used to speed up program execution. The user input function and constraints are both Compiled. The advantage of the Compile function is greatest with more complex expressions and larger problems. For small problems, it can be difficult to even detect a benefit, but there is little cost to the Compile. The range of speed improvement due to Compile can range from 30% to 30x. Compilation does not work for all *Mathematica* expressions. For example, if "Apply" or "If" are used in the function definition, the function will not Compile. It is suggested that if a complex expression involving *Mathematica* special functions is to be used, the user attempt to Compile the expression to test it. The CompileOption can be set to False (CompileOption->False) in the function call if the user function is not compilable. It may also be set to False if the user Compiles the expression before passing it in. It is possible to get errors during the Compile step. These will be highlighed in blue and will indicate that they are compilation errors. It is also possible that execution errors can result if non-machine numbers result from some computation of the compiled function. *Mathematica* usually reverts to the uncompiled expression in such cases, but the user may wish to run with CompileOption->False in this case to obtain the most accuracy and best speed.

We can see the benefit of the Compile in the following example:

```
In[19]:= ClearAll[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]

In[20]:= x = {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10};

In[21]:= f = Sum[(1 + x[[i]])^2, {i, 1, 10}]

Out[21]= (1 + x1)^2 + (1 + x10)^2 + (1 + x2)^2 + (1 + x3)^2 +
          (1 + x4)^2 + (1 + x5)^2 + (1 + x6)^2 + (1 + x7)^2 + (1 + x8)^2 + (1 + x9)^2

In[22]:= g1 = Function[Evaluate[x], Evaluate[f]];

In[23]:= g2 = Compile[Evaluate[x], Evaluate[f]];

In[24]:= Do[Apply[g1, {.1, 2., 3., 4., 3., .3, .3, .3, .5, .5}], {i, 1, 10 000}] // Timing

Out[24]= {0.296, Null}

In[25]:= Do[Apply[g2, {.1, 2., 3., 4., 3., .3, .3, .3, .5, .5}], {i, 1, 10 000}] // Timing

Out[25]= {0.079, Null}
```

We see that Compile speeded up the execution by a factor of more than 4. On some functions, the speedup can be a factor of 20 or more. Setting up a function as a Module prevents Compile from working on the objective function, and should be avoided. Some *Mathematica* functions, particularly those that are math functions, can be included in a Compiled function. Examples include Abs, Sqrt, Max, Min, Sin, etc. Others, such as FindRoot, can not.

II.3 User Functions: Working with Named Functions

The objective (goal) function in all the functions in this package can be defined in several ways. The function must in some cases be passed in differently, depending on the function format. In the simplest case, the expression is itself passed in. Here we find the minimum of the given function, which is zero.

```
In[9]:= ClearAll[x]

In[10]:= GlobalSearch[x^2 + y^2, , , {{x, 38, 40}, {y, 38, 40}}, .00000001, Starts → 1]

Out[10]= {{{x → 0., y → 0.}, 0.}}
```

We can also pass in a function by name:

```
In[11]:= f := x^2 + y^2

In[12]:= GlobalSearch[f, , , {{x, 38, 40}, {y, 38, 40}}, .00000001, Starts → 1]

Out[12]= {{{x → 0., y → 4.44089 × 10^-16}, 1.97215 × 10^-31}}
```

In order to Compile or use the objective function, the programs in this package will attempt to Evaluate the expression. If there are certain operations involved in the user function, such as obtaining the Inverse or Determinant of a matrix, *Mathematica* will attempt to Evaluate these symbolically when the function is used in the program. It will then work with the Evaluated function. If the user expression is complex, however, this may become impossible. For example, a symbolic inverse of a matrix with 10 variables may take a very long time. The user can test this by trying to Evaluate the objective function before passing it in to any of the functions of this package. If it fails to Evaluate, then use the parameter EvaluateObj->False to prevent evaluation. Another time when EvaluateObj should be set to False is when any logical operations are performed in the objective function, as in the following example:

```
In[13]:= f = Function[{x, y}, If[x + y < 0, tem = 0, tem = y + x]; tem];

In[14]:= Apply[f, {-1, -1}]

Out[14]= 0

In[15]:= Apply[f, {1, 1}]

Out[15]= 2
```

Clearly, the function is minimized for any set with x + y negative or for x==-y. Because of the conditional logic in the expression, it should not be symbolically evaluated. This means that it should not be simplified (SimplifyOption->False) or Evaluated (EvaluateObj->False):

```
In[33]:= GlobalSearch[f, {}, , {{x, 38, 40}, {y, 38, 40}}, .000001,
           Starts → 1, EvaluateObj → False, SimplifyOption → False] // Timing

Out[33]= {0.016, {{{x → -125.745, y → 38.013}, 0}}}
```

If, in contrast, we set up a function as a Module, the function can not be Compiled or Evaluated without giving an incorrect

answer. However, this problem is checked for internally and does not cause a problem, though it does cause a problem in *Mathematica* functions such as NMinimize.

```
In[16]:= g[x_, y_] := Module[{d}, If[x < 0 && y < 0, tem = 0, tem = 1 + Abs[y] + Abs[x]]; Return[tem]];

In[17]:= GlobalSearch[g, , , {{x, 38, 40}, {y, 38, 40}}, .000001, Starts → 1] // Timing

Out[17]= {0.031, {{{x → -2.49755, y → -2.93525}, 0}}}
```

III GENERAL NONLINEAR OPTIMIZATION: GlobalSearch, GlobalPenaltyFn, IntervalMin AND MultiStartMin

III.1 Introduction

Five general nonlinear solvers are included in this package. They each are designed for a particular class of problems. These functions provide tools for global optimization. Traditional gradient (local) approaches require the user to know how many optima are being sought for the function to be solved, and roughly where the optima are, so that a good initial guess can be made. The user, however, rarely can make a good initial guess and usually has no information about the existence of multiple solutions. In the absence of such information, existing algorithms will generally converge to an answer, but this may be only a local solution that may not be globally optimal. Furthermore, even global optima may not be unique. The functions in this package can solve black-box objective functions, objective functions that are nondifferentiable, differential equation models, and functions with discrete steps in them. The initial bounds on the parameters given on input do not need to bound the true solution (except for the GlobalMinima function), but only help get the algorithms started in a good region.

GlobalSearch approaches the difficult problem of finding a global optimum with several techniques. A generalized hill climbing technique is used that is based on Newton's method but using a generalized gradient rather than a derivative, and allowing for constraints. Constraints must be analytic, but can be linear or nonlinear. Multiple starts are used to test for the existence of multiple solutions. The multiple starts are generated randomly from the region defined by the range of parameter values input by the user. Feasible starting regions are not needed, but it is assumed that objective function values in this region are Real. When a step can not be made that improves the solution by at least "tolerance", the program terminates. If a problem uses only discrete 0-1 variables, the functions InterchangeMethodMin and TabuSearchMin should be used. GlobalSearch is the solver used by the regression and maximum likelihood functions. The function is defined by

GlobalSearch[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (11)

GlobalPenaltyFn addresses the problem of constraints that are nonanalytic. Nonanalytic functions can be algorithmic, can have conditional logic, or can be too complex for Solve to separate variables. If GlobalSearch fails on the constraints, use GlobalPenaltyFn. A generalized hill climbing technique is used that is based on Newton's method but using a generalized gradient rather than a derivative, and allowing for constraints. An adaptive penalty method is used that adjusts the penalties for constraint violation depending on the degree of violation of constraints. Multiple starts are used to test for the existence of multiple solutions. The multiple starts are generated randomly from the region defined by the range of parameter values input by the user. Feasible starting regions are not needed, but it is assumed that objective function values in this region are Real. When a step can not be made that improves the solution by at least "tolerance", the program terminates. If a problem uses only discrete 0-1 variables, the functions InterchangeMethodMin and TabuSearchMin should be used. Two methods are available for solving equality constraints. An analytic method is default. To override this method, used PenaltyMethod->True. For the PenaltyMethod, the default (ExactEqualities->True) uses a numerical method to stay on equality constraint lines at all times, and the alternate method is a penalty method for both inequalities and equalities. An

optional 4th parameter in the variable bounds list can be used to define a variable to be Integer (as in **{{var1name,lowbound,highbound, Integer}..}**). The function is defined by

GlobalPenaltyFn[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (12)

IntervalMin uses Interval methods to find a solution. Intervals are computed in *Mathematica* based on concepts of limits. An initial Interval is input by the user. It is beneficial for this Interval to cover the true solution point, but this is not necessary as long as it is not orders of magnitude distant. Both the input objective function and the constraints must be analytic. IntervalMin is robust to local minima, but is not as fast as GlobalSearch because Compile can not be used with Interval variables. IntervalMin can not solve problems with equality constraints at this time. We illustrate Intervals below:

```
In[51]:= x^2 /. x → Interval[{-1, 2}]

Out[51]= Interval[{0, 4}]
```

In this case, the true minimum of the function is given by the lower bound of the Interval result. In simple or separable cases, this will pertain. In other cases it will not:

```
In[19]:= 1.` - 2.` x + x^2 + 100.` x^4 - 200.` x^2 y + 100.` y^2 /.
          {x → Interval[{0, 2}], y → Interval[{0, 2}]}

Out[19]= Interval[{-1603., 2005.}]
```

The true minimum of this function is 0 at {1,1}. The true minimum is within the Interval of the result in this case. Note that when IntervalMin is used, both the estimated parameters and the function value are given in terms of an Interval. The format for the function is given by:

IntervalMin[expression,inequalities,Null,{InitialPoint__Interval},varlist,tolerance,options] (13)

MultiStartMin addresses the problem of inequality constraints that are nonanalytic. MultiStartMin is effective for highly nonlinear functions with fewer than 15 variables. MultiStartMin may have trouble with multiple constraints if the solution lies in the corner of two constraints. In such cases, use IntervalMin or GlobalPenaltyFn. A generalized hill climbing technique is used that is based on Newton's method but using a generalized gradient rather than a derivative, and allowing for constraints. Inequality constraints are treated as hard boundaries. Equality constraints must be analytic. Integer variables should not be used in Equality constraints unless the equality constraints are linear. All combinations of 2 or more variables at a time (number defined by option **SearchDirections**) are tested to find the best search direction. Multiple starts are used to test for the existence of multiple solutions. The multiple starts are generated randomly from the region defined by the range of parameter values input by the user. Feasible starting regions are not needed, but it is assumed that objective function values in this region are Real. Mixtures of Real, Integer, and Discrete (defined by a List) variables are allowed. If a problem uses only discrete 0-1 variables, the functions InterchangeMethodMin and TabuSearch-Min should be used. The function is defined by

MultiStartMin[expression,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (14)

The GlobalMinima function uses an adaptive grid refinement technique. This technique is robust, and can find multiple solutions in a single run. Because it's operation is so different, it is documented in it's own section, later.

**GlobalMinima[expression,inequalities,{{var1name,lowbound,highbound}..},grid,tolerance
,
contraction,indifference,options]**

III.2 Example

A simple example is to minimize a square function subject to a constraint, as follows. The constraint

```
x > 1 && y > 1
```

must be converted to standard form as a list:

```
-x < -1 && -y < -1
```

```
-x + 1 < 0 && -y + 1 < 0
```

```
In[20]:= c = {-x + 1, -y + 1}
Out[20]= {1 - x, 1 - y}
```

```
In[21]:= GlobalSearch[x^2 + y^2, c, , {{x, -5, 5}, {y, -5, 5}}, .000001, Starts → 1] // Timing
Out[21]= {0.031, {{{x → 1, y → 1}, 2.}}}
```

The output shows three solutions from random initial points, the default, with the {x,y} value and the function value in each list element. Initial feasible points are not needed. In this case, the function is smooth, so all three solutions are the same, and there is really no need to use multiple starts.

III.3 Program Operation

III.3.A Parameters and Default Values

The tolerance (T) defines the amount of improvement (absolute, not relative) in the function value required at each iteration. If at least this much improvement is not found, the program stops. The tolerance can be set to very small values, such as 10^{-15}, to achieve a highly accurate estimate of the minimum. If the user sets tolerance = 0, this is an error.

The functions have seven options. The defaults are MaxIterations->10000, CompileOption->True, ShowProgress->True, StartsList->{}, EvaluateObj->True, SimplifyOption->True, and Starts->3. MaxIterations prevents the program from running away to infinity when a mistake is made in the function (Min[x] for example). For very computationally expensive problems (like those with many parameters), it is useful to use Starts->1 to evaluate the time a run takes. For general usage, Starts->3 (the default) is good. The results of this short run can then be used to define further, more directed runs based on whether all 3 starts found the same solution. A list of starting values can be input instead of letting the program find them with random search (e.g., StartsList->{{1.,2.},{2.,1.}}). The CompileOption determines whether the user function and constraints will be compiled. While Compile reduces execution time, some functions can not be compiled, in which case CompileOption->False should be used. If the objective function should not be Evaluated, use EvaluateObj->False. SimplifyOption attempts to simplify the objective function. If this should not be done or will not improve the solution, this should be set to False. The MultiStartMin function also has a SearchDirections (default 2) option. More search directions help solve highly nonlinear problems, but increase the execution time exponentially.

III.3.B Bounds

Whereas most algorithms require input of an initial guess or starting point (which must be feasible) to initiate the search, GlobalSearch requires only bounds on the variables. Bounds are generally easier to define than are feasible starting points. Physical, logical, or economic considerations often provide guidance on realistic bounds for a variable. If the optimization problem is wing design and one variable is wing thickness, a certain range of thicknesses is sensible to consider. For a financial problem, the range of values for an investment mix across a product line is clearly bounded between zero and the total budget, and the upper limit can generally be bounded below a much smaller value than this. If narrower bounds are known, then convergence will require fewer function calls, but even quite wide bounds are acceptable. If erroneously narrow bounds are input, then initial step size will be too small, and the function will act more like a local solver. It is better to err on the side of wider bounds. Upper and lower bounds must be input by the user, as illustrated below. These bounds do not restrict the searching of GlobalSearch, MultiStartMin, IntervalMin, or GlobalPenaltyFn (as they do for GlobalMinima) but merely provide an initial guide for generating feasible starts. Initial bounds do not need to produce feasible solutions, the program can find initial feasible points, but the values within these bounds are assumed to be Real. If hard bounds on variables are necessary (such as positivity restrictions), they can be entered as constraints.

III.3.C Constraints

Constraints are entered as *Mathematica* functions, which may be linear or nonlinear. Equality constraints are entered in the third position in standard form. For example, positivity restrictions on x and y would be represented by the list:

```
{-x, -y}
```

The following problem is a typical LP problem, with the solution at the intersection of two inequality constraints. The solution is {2/3,10/3,0}).

```
In[22]:= GlobalSearch[-x1 - 2 x2 + x3,
           {x1 + x2 + x3 - 4, -x1 + 2 x2 - 2 x3 - 6, 2 x1 + x2 - 5, -x1, -x2, -x3}, ,
           {{x1, 0, 1}, {x2, 0, 1}, {x3, 0, 1}}, .000000001, Starts → 1] // Timing

Out[22]= {0.375, {{{x1 → 0.666667, x2 → 3.33333, x3 → 0}, -7.33333}}}

In[23]:= IntervalMin[-x1 - 2 x2 + x3,
           {x1 + x2 + x3 - 4, -x1 + 2 x2 - 2 x3 - 6, 2 x1 + x2 - 5, -x1, -x2, -x3}, ,
           {Interval[{0., 4.}], Interval[{0., 4.}], Interval[{0., 1.}]},
           {x1, x2, x3}, .0000001] // Timing

Out[23]= {4.625, {{x1 → Interval[{0.666667, 0.666667}], x2 → Interval[{3.33333, 3.33333}],
            x3 → Interval[{5.55112 × 10^-17, 9.15934 × 10^-16}]}, Interval[{-7.33333, -7.33333}]}}
```

We see that IntervalMin is much slower, but there are problems where it performs better. We next test GlobalPenaltyFn.

```
In[24]:= GlobalPenaltyFn[-x1 - 2 x2 + x3,
           {x1 + x2 + x3 - 4, -x1 + 2 x2 - 2 x3 - 6, 2 x1 + x2 - 5, -x1, -x2, -x3}, ,
           {{x1, 0, 3}, {x2, 0, 3}, {x3, 0, 1}}, .000001, Starts → 1] // Timing

Out[24]= {1.734, {{{x1 → 0.666667, x2 → 3.33333, x3 → 0}, -7.33333}}}
```

We see above that GlobalPenaltyFn is slower than GlobalSearch because it can take nonanalytic constraints. MultiStartMin is able to solve this problem.

```
In[25]:= MultiStartMin[-x1 - 2 x2 + x3,
           {x1 + x2 + x3 - 4, -x1 + 2 x2 - 2 x3 - 6, 2 x1 + x2 - 5, -x1, -x2, -x3}, ,
           {{x1, 0, 3}, {x2, 0, 3}, {x3, 0, 1}}, .0000001, Starts → 1] // Timing

Out[25]= {0.125, {{{x1 → 0.666666, x2 → 3.33333, x3 → 1.58683 × 10^-7}, -7.33333}}}
```

It is useful to illustrate a function in which the constraints are nonanalytic. Any function for which Solve will not work falls in this category. We take as the example:

```
Min[y^2 + x]  s.t.  x == Sin[y]
```

The above equality can not be Solved for y uniquely. GlobalSearch and GlobalPenaltyFn nevertheless succeed.

```
In[26]:= res = GlobalSearch[y^2 + Sin[y], , , {{y, 0, 1}}, .000000001, Starts → 1] // Timing

Out[26]= {0.015, {{{y → -0.450184}, -0.232466}}}
```

Where x is then

```
In[27]:= x = N[Sin[y]] /. res[[2, 1, 1]]

Out[27]= -0.435131

In[28]:= ClearAll[x]

In[29]:= res = GlobalSearch[y^2 + x, , {x - Sin[y]},
           {{x, 0, 1}, {y, 0, 1}}, .00000001, Starts → 1, ShowProgress → False] // Timing

Out[29]= {0.015, {{{x → -0.435131, y → -0.450184}, -0.232466}}}
```

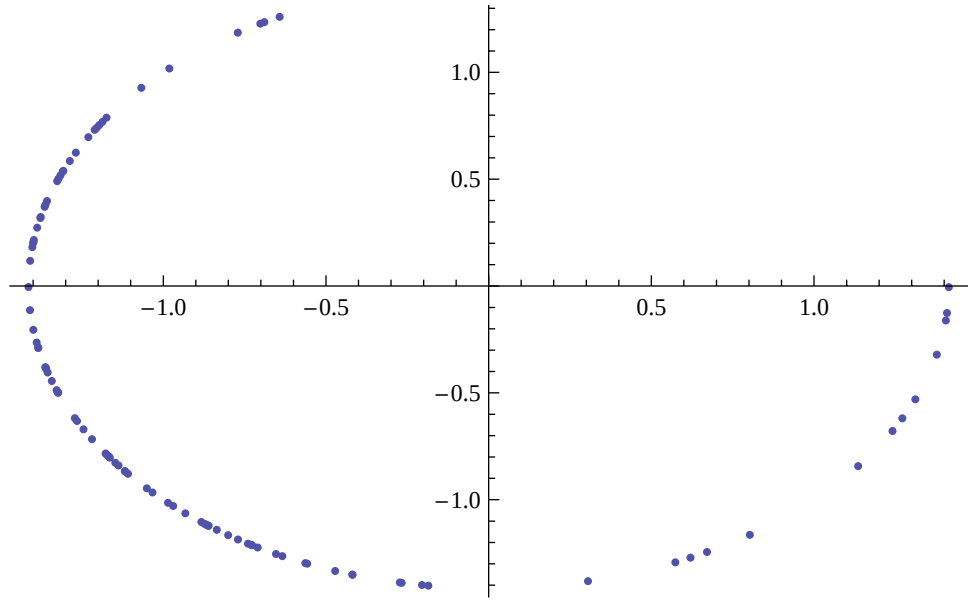
In this problem, GlobalPenaltyFn can solve it.

```
In[30]:= res = GlobalPenaltyFn[y^2 + x, , {x - Sin[y]},
           {{x, -1, 1}, {y, 0, 1}}, .000000001, Starts → 1, ShowProgress → False] // Timing

Out[30]= {0.031, {{{x → -0.435131, y → -0.450184}, -0.232466}}}
```

In the next example, we see how a nonlinear constraint can be defined that creates a whole set of solution points, an approximation to which can be obtained with enough starts. In this example, the solution must lie outside the circle of radius Sqrt[2], but the unconstrained solution is at the origin. This means that all points on this circle are optimal.

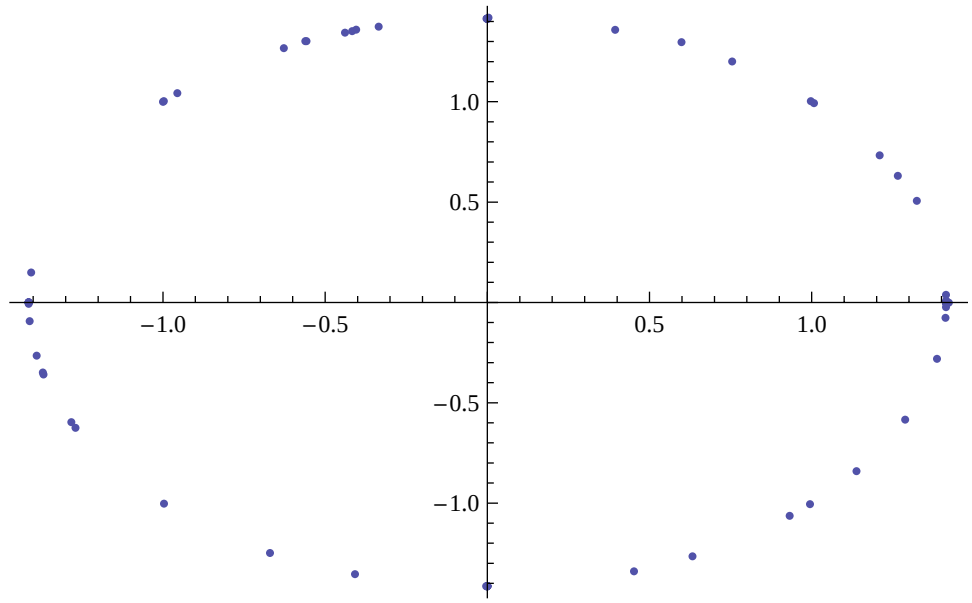
```
In[31]:= ClearAll[x]

In[32]:= b = GlobalSearch[x^2 + y^2, {2 - (x^2 + y^2)}, ,
           {{x, -10, 10}, {y, -10, 10}}, .0001, Starts → 100, CompileOption -> True];

In[33]:= b[[1]]

Out[33]= {{x → -1.24498, y → -0.670847}, 2.}

In[34]:= c = {}; Do[AppendTo[c, {x, y} /. b[[ii, 1]]], {ii, 1, 100}]
```

```
In[35]:= ListPlot[c]
```



In this case, the program finds only a subset of the solutions because of the order in which multiple roots are processed. MultiStartMin finds a better sample of points around the circle in this case.

```
In[36]:= b = MultiStartMin[x^2 + y^2, {2 - (x^2 + y^2)}, ,
             {{x, -10, 10}, {y, -10, 10}}, .0001, Starts → 100, CompileOption -> True];

In[37]:= b[[1]]

Out[37]= {{x → -1.38914, y → -0.265158}, 2.00001}

In[38]:= c = {}; Do[AppendTo[c, {x, y} /. b[[ii, 1]]], {ii, 1, 80}]

In[39]:= ListPlot[c]
```



Equality constraints are input as a list of equality statements in the third parameter position. In the following problem, the solution is {1/3,1/3,1/3}, with f=1/3.

```
In[40]:= GlobalSearch[x^2 + y^2 + z^2, {}, {x + y + z - 1},
           {{x, 0, 1}, {y, 0, 1}, {z, 0, 1}}, .00000001, Starts → 1] // Timing

Out[40]= {0.031, {{{x → 0.333333, y → 0.333333, z → 0.333333}, 0.333333}}}

In[41]:= MultiStartMin[x^2 + y^2 + z^2, {}, {x + y + z - 1},
           {{x, 0, 1}, {y, 0, 1}, {z, 0, 1}}, .00000001, Starts → 1] // Timing

Out[41]= {0.016, {{{x → 0.333326, y → 0.333372, z → 0.333302}, 0.333333}}}

In[42]:= GlobalPenaltyFn[x^2 + y^2 + z^2, {}, {x + y + z - 1},
           {{x, 0, 1}, {y, 0, 1}, {z, 0, 1}}, .00000001, Starts → 1] // Timing

Out[42]= {0.016, {{{x → 0.333333, y → 0.333333, z → 0.333333}, 0.333333}}}
```

The options PenaltyMethod→True,ExactEqualities→False are useful for complex equality constraints. The following problem needs these options.

```
In[43]:= vars = {x1, x2, x3}; nvars = Length[vars]; varlist = {{x1, 0, 1}, {x2, 0, 1}, {x3, 0, 1}};
         (* randomly generated solution to test problem *)sol = Table[Random[], {nvars}];

         objf = (x1 - sol[[1]])^2 + 2 * (x2 - sol[[2]])^2 + 3 * (x3 - sol[[3]])^2;

         eqs = {Sin[5 * (x1 - sol[[1]])] - 12 * (x3 - sol[[3]]) * (x2 - sol[[2]]),
            Exp[(x1 - sol[[1]]) * (x2 - sol[[2]]) * (x3 - sol[[3]])] - 1};

         ineqs = {((x1 - sol[[1]]) * (x2 - sol[[2]]) * (x3 - sol[[3]]))^2 +
             Sin[15 * (x1 * x2 * x3 - sol[[1]] * sol[[2]] * sol[[3]])] - 0.05};

In[47]:= sol

Out[47]= {0.624533, 0.146612, 0.193704}

In[48]:= res = GlobalPenaltyFn[objf, ineqs, eqs,
           varlist, .0000000001, Starts → 1, PenaltyMethod → True] // Timing

Out[48]= {0.469, {{{x1 → 0.624533, x2 → 0.146621, x3 → 0.193699}, 2.42804 × 10^-10}}}

In[49]:= res = GlobalPenaltyFn[objf, ineqs, eqs, varlist, .0000000001,
           Starts → 1, PenaltyMethod → True, ExactEqualities → True] // Timing

Out[49]= {0.578, {{{x1 → 0.624533, x2 → 0.14661, x3 → 0.193704}, 6.45615 × 10^-12}}}
```

In some cases, the solution to a problem occurs at the intersection of constraints. These problems are difficult but can be solved. In the following problem, the solution is exactly at the intersection of the line and the circle:

```
In[50]:= r = Solve[{x1 + x2 - 1.2 == 0, x1^2 + x2^2 == 1}]

Out[50]= {{x1 → 0.225834, x2 → 0.974166}, {x1 → 0.974166, x2 → 0.225834}}
```

```
In[51]:= (x1 - .1)^2 + (x2 - .1)^2 + x3^2 /. r[[1]]

Out[51]= 0.78 + x3^2
```

In this problem, some of the solutions are not optimal.

```
In[54]:= GlobalSearch[(x1 - .1)^2 + (x2 - .1)^2 + x3^2,
           {x1 + x2 - 1.2}, {x1^2 + x2^2 - 1}, {{x1, 0, 1}, {x2, 0, 1}, {x3, 5, 10}},
           .00000001, Starts → 1, ShowProgress → False] // Timing

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       General::stop : Further output of
          CompiledFunction::cfn will be suppressed during this calculation. >>

       LessEqual::nord : Invalid comparison with -0.129711 - 0.381468 i attempted. >>

       LessEqual::nord : Invalid comparison with -0.129711 - 0.381468 i attempted. >>

       LessEqual::nord : Invalid comparison with -0.169711 - 0.247981 i attempted. >>

       General::stop :
        Further output of LessEqual::nord will be suppressed during this calculation. >>

Out[54]= {0.312, {{{x1 → 0.974166, x2 → 0.225834, x3 → -0.00823202}, 0.780068}}}
```

We see that GlobalSearch found one of the solutions with 1 starts. GlobalPenaltyFn can also find a solution but is slower.

```
In[53]:= GlobalPenaltyFn[(x1 - .1)^2 + (x2 - .1)^2 + x3^2,
           {-x1, -x2, x1 + x2 - 1.2}, {x1^2 + x2^2 - 1},
           {{x1, 0, 1}, {x2, 0, 1}, {x3, 5, 10}}, .0000000001, Starts → 1] // Timing

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 8; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       General::stop : Further output of
          CompiledFunction::cfn will be suppressed during this calculation. >>

Out[53]= {1.328, {{{x1 → 0.974166, x2 → 0.225834, x3 → -2.52968 × 10^-6}, 0.78}}}
```

MultiStartMin also solves it correctly.

```
In[57]:= MultiStartMin[(x1 - .1)^2 + (x2 - .1)^2 + x3^2, {x1 + x2 - 1.2}, {x1^2 + x2^2 - 1},
           {{x1, 0, .5}, {x2, 0, .5}, {x3, 5, 10}}, .0000001, Starts → 1] // Timing

       CompiledFunction::cfn : Numerical error encountered
          at instruction 12; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 12; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 12; proceeding with uncompiled evaluation. >>

       General::stop : Further output of
          CompiledFunction::cfn will be suppressed during this calculation. >>

       LessEqual::nord : Invalid comparison with -0.121512 - 0.403902 i attempted. >>

       LessEqual::nord : Invalid comparison with 0.00489123 - 0.672133 i attempted. >>

       LessEqual::nord : Invalid comparison with -0.18917 - 0.14757 i attempted. >>

       General::stop :
        Further output of LessEqual::nord will be suppressed during this calculation. >>

Out[57]= {0.141, {{{x1 → 0.974166, x2 → 0.225834, x3 → 0.0000161832}, 0.78}}}
```

In the following example, the constraint forces a solution outside the unit circle. The solution is 0.737157 at {.707,.707}, which is found by GlobalSearch and GlobalPenaltyFn. MultiStartMin only succeeds on this problem with multiple starts. IntervalMin gets pretty close.

```
In[58]:= MultiStartMin[(x - .1)^2 + (y - .1)^2, {-(x^2 + y^2 - 1)},
           {}, {{x, -4, 5}, {y, -4, 5}}, .0000001, Starts → 3] // Timing

Out[58]= {0.156, {{{x → 0.861922, y → 0.507044}, 0.74621}}}

In[59]:= GlobalSearch[(x - .1)^2 + (y - .1)^2, {-(x^2 + y^2 - 1)},
           {}, {{x, -4, 5}, {y, -4, 5}}, .000000001, Starts → 1] // Timing

Out[59]= {0.109, {{{x → 0.707147, y → 0.707066}, 0.737157}}}

In[60]:= IntervalMin[(x - .1)^2 + (y - .1)^2, {-(x^2 + y^2 - 1)}, ,
           {Interval[{0., 1.}], Interval[{0., 1.}]}, {x, y}, .0001] // Timing

Out[60]= {1.219, {{x → Interval[{0.695487, 0.695545}], y → Interval[{0.718552, 0.718576}]},
           Interval[{0.737212, 0.73731}]}}

In[61]:= GlobalPenaltyFn[(x - .1)^2 + (y - .1)^2, {-(x^2 + y^2 - 1)}, ,
           {{x, 0, 1}, {y, 0, 1}}, .000000001, Starts → 2, ShowProgress → False] // Timing

Out[61]= {6.141, {{{x → 0.705179, y → 0.70903}, 0.737158}}}
```

WARNING: Constraints can cause difficulties for GlobalSearch. Constraints can be mistakenly formulated such that no feasible space exists. In this case, GlobalSearch will stop with an error message:

```
In[62]:= ClearAll[x, y, z]
```

```
In[63]:= GlobalSearch[x^2 + y^2 + z^2, {x - 1, -x + 2}, {},
           {{x, 0, 1}, {y, 0, 1}, {z, 0, 1}}, .00000001, Starts → 1] // Timing

         Error: feasible solution not found for starting value

Out[63]= {0.062, {{$Failed}}}
```

WARNING: Equality constraints should not be used with Integer variables. The result of doing so is that the optimal solution is not found.

```
In[64]:= GlobalPenaltyFn[(x1 - .1)^2 + (x2 - .1)^2 + x3^2, {x1 + x2 - 1.2},
           {x1^2 + x2^2 - 1}, {{x1, 0, 1, Integer}, {x2, 0, .5}, {x3, 5, 10}},
           .0000001, Starts → 1, ShowProgress → False] // Timing

       CompiledFunction::cfn : Numerical error encountered
          at instruction 23; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 9; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 9; proceeding with uncompiled evaluation. >>

       General::stop : Further output of
          CompiledFunction::cfn will be suppressed during this calculation. >>

Out[64]= {0.344, {{{x1 → 1, x2 → 0, x3 → -8.10327 × 10^-9}, 0.82}}}
```

III.4 Output

Following solution, a list is returned that contains the solution. For example, two points in a 2 parameter problem would give the list {{{1.1,1.2},-5},{{1.2,1.21},-4.9}}. Intermediate output can be printed out by setting ShowProgress->True.

```
In[65]:= GlobalSearch[x^2 + y^2, {-x}, , {{x, -5, 5}, {y, -5, 5}},
           .000000001, Starts → 1, ShowProgress → True]

         Global Optimization, Version 5.2

         number of variables = 2

         tolerance = 1. × 10^-9

         number of starts = 1

         Initial point 1  {{x → 0, y → 0.166722}, 0.0277963}

         Vector of search:

         {{x → 1.27086 × 10^-7, y → -7.4504 × 10^-9}, 1.62063 × 10^-14}

Out[65]= {{{x → 1.27086 × 10^-7, y → -7.4504 × 10^-9}, 1.62063 × 10^-14}}
```

Following solution, a list is returned that contains the solution as a list of replacement rules with corresponding solutions.

III.5 Maximization

To find a maximum instead of a minimum, simply call GlobalSearch with the negative of the function, keeping in mind that a negative function value will be printed out.

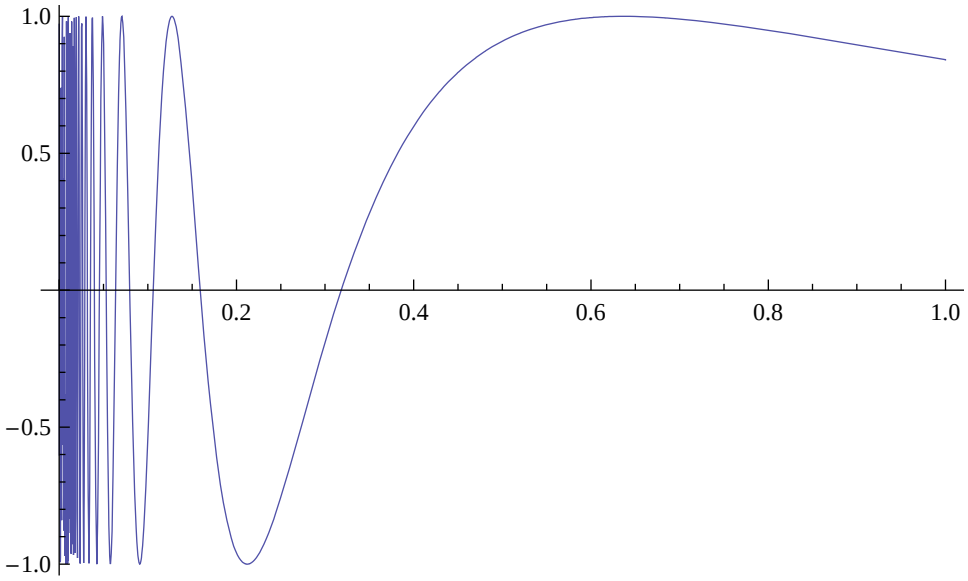
III.6 Limitations

No optimization algorithm can solve all conceivable problems. Consider a response surface in two dimensions that is flat everywhere except for a very small but very deep pit at one location. Any hill-climbing algorithm will fail completely to solve this problem, because for almost all starting points there is no gradient.

Some functions have infinitely many solutions. Consider Sin(1/x) between 0 and 1. As the function approaches zero, the minima (with z = -1) get closer and closer together, as we can see from the plot following:

```
In[66]:=
Plot[Sin[1/x], {x, 0, 1}]
```

Out[66]=



Although it is fundamentally impossible to list all solutions when there are infinitely many, GlobalSolve can show that there are very many solutions. If we run the program with 40 starts on the interval [0, 0.6], we obtain many points with values close to -1.0 (shown below). This result demonstrates that there are many solutions and that the density of solutions increases with an approach to zero. This is thus an approximate solution to the true situation. x is bounded away from 0 to prevent underflow.

```
In[67]:= GlobalSearch[Sin[1 / x], {-x + .00001, x - .6}, ,
          {{x, 0.0000001, .6}}, .0001, CompileOption → False, Starts → 20]

Out[67]= {{{x → 0.211722}, -0.999942}, {{x → 0.212207}, -1.},
          {{x → 0.212207}, -1.}, {{x → 0.212759}, -0.999925}, {{x → 0.212207}, -1.},
          {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.},
          {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.}, {{x → 0.00347879}, -1.},
          {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.}, {{x → 0.0909457}, -1.},
          {{x → 0.0909457}, -1.}, {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.},
          {{x → 0.212207}, -1.}, {{x → 0.212207}, -1.}, {{x → 0.0578745}, -1.}}
```

By restricting the search region, we find more solutions:

```
In[68]:= GlobalSearch[Sin[1 / x], {-x + .000000000001, x - .2}, ,
          {{x, 0.000001, .2}}, .00001, Starts -> 20]

Out[68]= {{{x → 0.0909457}, -1.}, {{x → 0.0909457}, -1.},
          {{x → 1. × 10^-12}, -0.999988}, {{x → 0.0424413}, -1.}, {{x → 0.0909457}, -1.},
          {{x → 0.0025774}, -0.999999}, {{x → 0.0909457}, -1.},
          {{x → 0.0909457}, -1.}, {{x → 0.0424413}, -1.}, {{x → 0.0909457}, -1.},
          {{x → 0.0909457}, -1.}, {{x → 0.0205361}, -1.}, {{x → 0.0909457}, -1.},
          {{x → 0.0578745}, -1.}, {{x → 0.00290693}, -1.}, {{x → 0.0424413}, -1.},
          {{x → 0.0909457}, -1.}, {{x → 0.0424413}, -1.}, {{x → 0.0909457}, -1.}}
```

These examples illustrate that not all problems can be solved. Some are not solvable by any algorithm. Some may be ill-posed. However, for many problems that cause other algorithms to fail, GlobalSolve either succeeds, has a probability of succeeding, or provides a partial or approximate solution. We may thus say that the set of unsolvable problems is much smaller than it is for other solution methods. This is particularly so with respect to the inclusion of constraints.

The optimization algorithm fundamentally assumes real valued functions. It is possible for some parameter values that are tested to return values that are complex or otherwise not Real. The program assumes that these are illegal values and treats them like values that fail a constraint test. This means that it can solve a problem where values can be complex. The problem below works even if the input range does not include positive numbers.

```
In[71]:= GlobalSearch[x^.5, {}, {}, {{x, -2, -1}}, .000000000001, Starts → 1]

       CompiledFunction::cfn : Numerical error encountered
          at instruction 3; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 3; proceeding with uncompiled evaluation. >>

       CompiledFunction::cfn : Numerical error encountered
          at instruction 3; proceeding with uncompiled evaluation. >>

       General::stop : Further output of
          CompiledFunction::cfn will be suppressed during this calculation. >>

Out[71]= {{{x → 4.98279 × 10^-25}, 7.05889 × 10^-13}}
```

III.7 Error Messages

An effort has been made to trap as many errors as possible. Many user errors will prompt *Mathematica* error messages. On input, only the order of parameters is evaluated in the passing function call. If a valid value for tolerance (> 0.0) is not input, the following error message is printed, and the program stops:

Error: tolerance must be > 0

If the upper bound of the variable definitions is smaller than the lower bound, the program will also stop with an error message.

If a parameter is omitted from the calling sequence, *Mathematica* cannot continue. To indicate the problem, it echoes back the names for functions and values for constants in the function call, and then it stops. This type of echoed-back output means that a format error exists in the calling sequence. Some syntax errors are trapped within the Global Optimization package.

After input, the program compiles the user input function and the constraints, speeding execution. If these functions are improperly formatted, this step may produce a compilation error message, which will terminate the run. If a function is defined such that it does not return numeric values, this will cause error messages from *Mathematica*, followed by termination of GlobalSearch.

When constraints are used, there may be no feasible region, particularly if the problem has many constraints. If this occurs, the following error message is printed, and the program terminates:

Error: no valid initial points found

In this case, check the constraints. Constraints can be in conflict if care is not taken.

A common mistake is to maximize when the intention was to minimize. This error can be recognized because the solution will tend to run into one of the parameter bounds and will have an illogical value. For example, the optimal airplane wing thickness for a design problem might come back as zero.

An unbounded solution may result from an improperly formulated problem (e.g., min(-1/x) over {-1, 1} which becomes -inf at x=0). Because the function z continues to get larger the closer one gets to zero at an increasing rate, the program will never report convergence. ShowProgress->True can be used to check for such problems during a run.

The user function passed in to GlobalSearch is Compiled. This may cause compilation errors. The result is usually to stop the execution of the program, but it is necessary for the user to realize the source of the error. Compiled functions can also generate errors at run time if the user function generates non-machine numbers such as high precision numbers or infinity. *Mathematica* will usually revert to the uncompiled function and continue running.

If the user defines variables in his program by mistake that should be parameters of the function, this will cause the program to stop or malfunction. For example, if pp=5 is defined by mistake, this will cause GlobalSearch to detect an error which can be understood from looking at the error output which shows an invalid parameter list:

```
In[72]:= pp = 5;
```

```
In[73]:= GlobalSearch[(pp - 30)^2 (y - 60)^2, , , {{pp, 0., 100.}, {y, 1, 5}}, 0.001]

          Error: Symbol expected in parameter list, variable 1

          {{5, 0., 100.}, {y, 1, 5}}

Out[73]= $Failed
```

Note how the program echoes back the parameter input, showing the mistake.

If the function to be solved is by mistake unbounded, then the parameter MaxIterations will stop the program from running forever.

```
In[74]:= GlobalSearch[x + y, {}, {}, {{x, -5, 5}, {y, -5, 5}},
          .0000001, Starts → 1, MaxIterations → 4]

          Warning: MaxIterations exceeded, execution terminated

          Result for this starting value not necessarily optimal

Out[74]= {{{x → -1.00543 × 10^36, y → -1.00543 × 10^36}, -2.01087 × 10^36}}
```

MaxIterations can also be used to make a preliminary run to test out execution.

```
In[75]:= r = GlobalSearch[100. * (x^2 - y)^2 + (1. - x)^2, {}, {},
           {{x, -5, 5}, {y, -10., 10.}}, .0000001, Starts → 1, MaxIterations → 1]

          Warning: MaxIterations exceeded, execution terminated

          Result for this starting value not necessarily optimal

Out[75]= {{{x → 1., y → 1.}, 1.2326 × 10^-30}}
```

We now wish to restart the program with the given ending value.

```
In[76]:= rr = r[[1]]

Out[76]= {{x → 1., y → 1.}, 1.2326 × 10^-30}

In[77]:= s = {x, y} /. rr[[1]]

Out[77]= {1., 1.}

In[78]:= GlobalSearch[100. * (x^2 - y)^2 + (1. - x)^2, {}, {}, {{x, -5, 5}, {y, -10., 10.}},
           .0000001, StartsList → {s}, ShowProgress → False] // Timing

Out[78]= {0.031, {{{x → 1., y → 1.}, 1.2326 × 10^-30}}}
```

III.8 Performance/Speed

For larger problems, speed can become a limiting factor. It is suggested that the objective function and constraints be formulated such that they can be Compiled (see sec. II) which can speed up operation by 2x to 20x. If a function is very complex, it can be helpful to pull out subexpressions that do not need to be computed each time, and compute them once in advance. Simplify or FullSimplify may also speed up the function. Some large problems are run next to illustrate performance. Either GlobalSearch or GlobalPenaltyFn can be run for large unconstrained problems, for which they use the same basic algorithm. MultiStartMin can only be used on large problems if SearchDirections->1 is used, and FastStepping->True is recommended for such cases. Using MultiStartMin will not be advantageous for problems with highly nonlinear functions, but may be the only option for problems with integer or discrete variables.

```
In[79]:= vars = Table[ToExpression[StringJoin["x", ToString[i]]], {i, 1, 100}];

In[80]:= fg = Evaluate[Sum[vars[[i]]^2, {i, 1, 100}]];

In[81]:= vardefs = Table[{vars[[i]], -10, 11}, {i, 1, 100}];

In[82]:= GlobalSearch[fg, , , vardefs, .00000001, Starts → 1, ShowProgress → False] // Timing

Out[82]= {0.391, {{{x1 → 0., x2 → 0., x3 → 0., x4 → 0., x5 → 8.88178 × 10^-16, x6 → 0., x7 → 0., x8 → 0.,
      x9 → 0., x10 → 0., x11 → 0., x12 → 8.88178 × 10^-16, x13 → 0., x14 → 0., x15 → 0.,
      x16 → 0., x17 → 0., x18 → -8.88178 × 10^-16, x19 → 0., x20 → 0., x21 → 8.88178 × 10^-16,
      x22 → 0., x23 → 0., x24 → -8.88178 × 10^-16, x25 → 0., x26 → 2.22045 × 10^-16, x27 → 0.,
      x28 → 0., x29 → 0., x30 → -8.88178 × 10^-16, x31 → 0., x32 → 0., x33 → 0., x34 → 0.,
      x35 → 0., x36 → 0., x37 → 0., x38 → 0., x39 → 0., x40 → 0., x41 → 0., x42 → 0., x43 → 0.,
      x44 → 0., x45 → -8.88178 × 10^-16, x46 → 0., x47 → 0., x48 → 0., x49 → 0., x50 → 0.,
      x51 → 0., x52 → 0., x53 → 0., x54 → 0., x55 → -4.44089 × 10^-16, x56 → 0., x57 → 0.,
      x58 → 0., x59 → 0., x60 → 0., x61 → 0., x62 → 0., x63 → 0., x64 → 0., x65 → 0., x66 → 0.,
      x67 → 0., x68 → 0., x69 → 0., x70 → -8.88178 × 10^-16, x71 → 0., x72 → 4.44089 × 10^-16,
      x73 → 0., x74 → 0., x75 → 0., x76 → 0., x77 → 0., x78 → 0., x79 → 0., x80 → 0., x81 → 0.,
      x82 → 0., x83 → 0., x84 → 0., x85 → 0., x86 → 0., x87 → 0., x88 → -5.55112 × 10^-17,
      x89 → 0., x90 → 0., x91 → 0., x92 → 0., x93 → 0., x94 → 0., x95 → 0., x96 → 0.,
      x97 → 0., x98 → 0., x99 → 0., x100 → -8.88178 × 10^-16}, 7.54656 × 10^-30}}}

In[83]:= GlobalPenaltyFn[fg, , , vardefs, .0000001, Starts → 1, ShowProgress → False] // Timing

Out[83]= {0.328,
  {{{x1 → 0, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 0, x7 → 0, x8 → 0, x9 → 0, x10 → 0, x11 → 0,
      x12 → 0, x13 → 0, x14 → 0, x15 → 0, x16 → 0, x17 → 0, x18 → 0, x19 → 0, x20 → 0, x21 → 0,
      x22 → 0, x23 → 0, x24 → 0, x25 → 0, x26 → 0, x27 → 0, x28 → 0, x29 → 0, x30 → 0, x31 → 0,
      x32 → 0, x33 → 0, x34 → 0, x35 → 0, x36 → 0, x37 → 0, x38 → 0, x39 → 0, x40 → 0, x41 → 0,
      x42 → 0, x43 → 0, x44 → 0, x45 → 0, x46 → 0, x47 → 0, x48 → 0, x49 → 0, x50 → 0, x51 → 0,
      x52 → 0, x53 → 0, x54 → 0, x55 → 0, x56 → 0, x57 → 0, x58 → 0, x59 → 0, x60 → 0, x61 → 0,
      x62 → 0, x63 → 0, x64 → 0, x65 → 0, x66 → 0, x67 → 0, x68 → 0, x69 → 0, x70 → 0, x71 → 0,
      x72 → 0, x73 → 0, x74 → 0, x75 → 0, x76 → 0, x77 → 0, x78 → 0, x79 → 0, x80 → 0, x81 → 0,
      x82 → 0, x83 → 0, x84 → 0, x85 → 0, x86 → 0, x87 → 0, x88 → 0, x89 → 0, x90 → 0, x91 → 0,
      x92 → 0, x93 → 0, x94 → 0, x95 → 0, x96 → 0, x97 → 0, x98 → 0, x99 → 0, x100 → 0}, 0.}}}

In[84]:= vars = Table[ToExpression[StringJoin["x", ToString[i]]], {i, 1, 400}];

In[85]:= fg = Evaluate[Sum[vars[[i]]^2, {i, 1, 400}]];
```

```
In[86]:= vardefs = Table[{vars[[i]], -10, 11}, {i, 1, 400}];

In[87]:= GlobalSearch[fg, , , vardefs, .0000001, Starts → 1] // Timing
```

Out[87]=

$$\{3.875, \{\{\{x1 \to 0., x2 \to 0., x3 \to 0., x4 \to 0., x5 \to 0., x6 \to 0., x7 \to 2.22045 \times 10^{-16}, x8 \to 0.,$$
$$x9 \to 4.44089 \times 10^{-16}, x10 \to 8.88178 \times 10^{-16}, x11 \to 0., x12 \to 8.88178 \times 10^{-16},$$
$$x13 \to -2.22045 \times 10^{-16}, x14 \to 0., x15 \to 0., x16 \to 4.33681 \times 10^{-19}, x17 \to 0.,$$
$$x18 \to 0., x19 \to 0., x20 \to 0., x21 \to 0., x22 \to -8.88178 \times 10^{-16}, x23 \to 0.,$$
$$x24 \to 0., x25 \to 8.88178 \times 10^{-16}, x26 \to 0., x27 \to 0., x28 \to 0., x29 \to 0., x30 \to 0.,$$
$$x31 \to 0., x32 \to 0., x33 \to 2.22045 \times 10^{-16}, x34 \to 0., x35 \to 0., x36 \to 0., x37 \to 0.,$$
$$x38 \to -8.88178 \times 10^{-16}, x39 \to 0., x40 \to 0., x41 \to 0., x42 \to 0., x43 \to 0., x44 \to 0.,$$
$$x45 \to 0., x46 \to 0., x47 \to 0., x48 \to 0., x49 \to 0., x50 \to 0., x51 \to 0., x52 \to 0., x53 \to 0.,$$
$$x54 \to 0., x55 \to 0., x56 \to 8.88178 \times 10^{-16}, x57 \to -4.44089 \times 10^{-16}, x58 \to 0., x59 \to 0.,$$
$$x60 \to -8.88178 \times 10^{-16}, x61 \to 0., x62 \to 0., x63 \to 0., x64 \to 0., x65 \to -8.88178 \times 10^{-16},$$
$$x66 \to 0., x67 \to 0., x68 \to 0., x69 \to 0., x70 \to 0., x71 \to 0., x72 \to 0., x73 \to 0.,$$
$$x74 \to 0., x75 \to -8.88178 \times 10^{-16}, x76 \to 0., x77 \to 0., x78 \to 0., x79 \to 0., x80 \to 0.,$$
$$x81 \to 0., x82 \to 0., x83 \to 0., x84 \to 0., x85 \to 8.88178 \times 10^{-16}, x86 \to 0., x87 \to 0.,$$
$$x88 \to 0., x89 \to 0., x90 \to 0., x91 \to 0., x92 \to 0., x93 \to 8.88178 \times 10^{-16}, x94 \to 0.,$$
$$x95 \to 0., x96 \to 0., x97 \to 4.44089 \times 10^{-16}, x98 \to 0., x99 \to 0., x100 \to 0., x101 \to 0.,$$
$$x102 \to 0., x103 \to 0., x104 \to 4.44089 \times 10^{-16}, x105 \to 0., x106 \to 0., x107 \to 0.,$$
$$x108 \to 0., x109 \to 0., x110 \to 0., x111 \to 0., x112 \to 0., x113 \to 0., x114 \to 0.,$$
$$x115 \to 0., x116 \to -8.88178 \times 10^{-16}, x117 \to 0., x118 \to 0., x119 \to 0., x120 \to 0.,$$
$$x121 \to -8.88178 \times 10^{-16}, x122 \to -4.44089 \times 10^{-16}, x123 \to 4.44089 \times 10^{-16}, x124 \to 0.,$$
$$x125 \to 0., x126 \to 0., x127 \to 0., x128 \to 8.88178 \times 10^{-16}, x129 \to 0., x130 \to 0., x131 \to 0.,$$
$$x132 \to 0., x133 \to 4.44089 \times 10^{-16}, x134 \to 0., x135 \to 0., x136 \to 2.22045 \times 10^{-16}, x137 \to 0.,$$
$$x138 \to 0., x139 \to 0., x140 \to 0., x141 \to 0., x142 \to 0., x143 \to -1.11022 \times 10^{-16}, x144 \to 0.,$$
$$x145 \to 0., x146 \to 0., x147 \to 0., x148 \to 0., x149 \to -2.77556 \times 10^{-17}, x150 \to 0.,$$
$$x151 \to -2.22045 \times 10^{-16}, x152 \to 8.88178 \times 10^{-16}, x153 \to 0., x154 \to 0., x155 \to 0.,$$
$$x156 \to 0., x157 \to 0., x158 \to 0., x159 \to 0., x160 \to 0., x161 \to 8.88178 \times 10^{-16}, x162 \to 0.,$$
$$x163 \to 0., x164 \to 0., x165 \to 0., x166 \to -8.88178 \times 10^{-16}, x167 \to 0., x168 \to 0.,$$
$$x169 \to 0., x170 \to 0., x171 \to 0., x172 \to -4.44089 \times 10^{-16}, x173 \to -4.44089 \times 10^{-16},$$
$$x174 \to 0., x175 \to 4.44089 \times 10^{-16}, x176 \to 0., x177 \to 0., x178 \to 0., x179 \to 0.,$$
$$x180 \to 0., x181 \to 0., x182 \to 0., x183 \to 0., x184 \to 0., x185 \to 0., x186 \to 0., x187 \to 0.,$$
$$x188 \to 0., x189 \to 0., x190 \to 0., x191 \to 1.38778 \times 10^{-17}, x192 \to 0., x193 \to 0.,$$
$$x194 \to 0., x195 \to 0., x196 \to 0., x197 \to 0., x198 \to 0., x199 \to 0., x200 \to 0., x201 \to 0.,$$
$$x202 \to 0., x203 \to 0., x204 \to 0., x205 \to -4.44089 \times 10^{-16}, x206 \to 0., x207 \to 0.,$$
$$x208 \to 0., x209 \to 0., x210 \to 0., x211 \to 0., x212 \to 0., x213 \to 0., x214 \to 0., x215 \to 0.,$$
$$x216 \to -4.44089 \times 10^{-16}, x217 \to 0., x218 \to 0., x219 \to 0., x220 \to 0., x221 \to 0.,$$
$$x222 \to 0., x223 \to 0., x224 \to 0., x225 \to 0., x226 \to 0., x227 \to 0., x228 \to 0.,$$
$$x229 \to 0., x230 \to 0., x231 \to 0., x232 \to 8.88178 \times 10^{-16}, x233 \to 0., x234 \to 0.,$$
$$x235 \to 8.88178 \times 10^{-16}, x236 \to 0., x237 \to 0., x238 \to 0., x239 \to 0., x240 \to 0., x241 \to 0.,$$
$$x242 \to 0., x243 \to -1.11022 \times 10^{-16}, x244 \to 0., x245 \to 0., x246 \to 0., x247 \to 0.,$$
$$x248 \to 0., x249 \to 0., x250 \to 0., x251 \to 0., x252 \to 0., x253 \to 0., x254 \to 0., x255 \to 0.,$$
$$x256 \to 0., x257 \to 0., x258 \to 0., x259 \to 0., x260 \to 0., x261 \to 0., x262 \to 0., x263 \to 0.,$$
$$x264 \to 0., x265 \to 8.88178 \times 10^{-16}, x266 \to 0., x267 \to 0., x268 \to 0., x269 \to 0.,$$
$$x270 \to 8.88178 \times 10^{-16}, x271 \to 0., x272 \to 0., x273 \to 0., x274 \to 2.77556 \times 10^{-17},$$
$$x275 \to 0., x276 \to 0., x277 \to 0., x278 \to 0., x279 \to 0., x280 \to 0., x281 \to 0., x282 \to 0.,$$
$$x283 \to 0., x284 \to 0., x285 \to 4.44089 \times 10^{-16}, x286 \to 4.44089 \times 10^{-16}, x287 \to 0.,$$
$$x288 \to 8.88178 \times 10^{-16}, x289 \to 0., x290 \to 0., x291 \to -8.88178 \times 10^{-16}, x292 \to 0.,$$

```
x293 → 0., x294 → 0., x295 → 0., x296 → 0., x297 → 0., x298 → 0., x299 → 0., x300 → 0.,
x301 → 0., x302 → 0., x303 → 0., x304 → 0., x305 → 0., x306 → 0., x307 → 0., x308 → 0.,
x309 → 0., x310 → 0., x311 → 0., x312 → 0., x313 → 0., x314 → -8.88178×10^-16,
x315 → 0., x316 → 0., x317 → 0., x318 → 0., x319 → 0., x320 → 0., x321 → 0., x322 → 0.,
x323 → -4.44089×10^-16, x324 → -1.11022×10^-16, x325 → 0., x326 → 0., x327 → 0.,
x328 → 0., x329 → 0., x330 → 0., x331 → 8.88178×10^-16, x332 → 0., x333 → 0.,
x334 → 0., x335 → 0., x336 → 0., x337 → 0., x338 → 0., x339 → 0., x340 → 0., x341 → 0.,
x342 → 0., x343 → 0., x344 → 0., x345 → -4.44089×10^-16, x346 → 0., x347 → 0.,
x348 → -4.44089×10^-16, x349 → 0., x350 → 4.44089×10^-16, x351 → 0., x352 → 0.,
x353 → 0., x354 → -4.44089×10^-16, x355 → -4.44089×10^-16, x356 → 0., x357 → 0.,
x358 → -8.88178×10^-16, x359 → 0., x360 → 0., x361 → 0., x362 → 0., x363 → 0.,
x364 → 0., x365 → 0., x366 → 0., x367 → 0., x368 → 1.38778×10^-17, x369 → 0.,
x370 → 0., x371 → 0., x372 → 0., x373 → 0., x374 → 0., x375 → 0., x376 → 0., x377 → 0.,
x378 → 0., x379 → 8.88178×10^-16, x380 → 0., x381 → 0., x382 → 0., x383 → 0.,
x384 → 0., x385 → 4.44089×10^-16, x386 → 0., x387 → 0., x388 → 0., x389 → 0.,
x390 → 0., x391 → 8.88178×10^-16, x392 → 0., x393 → 0., x394 → 0., x395 → 0.,
x396 → 0., x397 → 0., x398 → 0., x399 → -1.11022×10^-16, x400 → 0.}, 2.65274×10^-29}}}
```

Note that for large problems with constraints or lots of data, execution will be slower.

III.9 Testing and Examples

In this section, the general nonlinear functions are tested. The test criteria are the ability to obtain accurate answers, time required to execute, size of problem, ability to handle noisy functions, and ability to find multiple solutions. In the first example, we test for the ability to solve to arbitrary precision:

```
In[88]:= GlobalSearch[(x - 30)^2 + (y - 60)^2, , ,
          {{x, 0., 10.}, {y, 1, 5}}, 0.000000000000000001, Starts -> 1]

Out[88]= {{{x → 30., y → 60.}, 0.}}
```

The following problem is the difficult Rosenbrock function, but it is easily solved.

```
In[89]:= GlobalSearch[100. * (x^2 - y)^2 + (1. - x)^2, {}, {}, {{x, -5, 5}, {y, -10., 10.}},
           .00000000001, Starts → 1, ShowProgress → False] // Timing

Out[89]= {0.047, {{{x → 1., y → 1.}, 0.}}}

In[90]:= MultiStartMin[100. * (x^2 - y)^2 + (1. - x)^2, {}, {},
           {{x, -5, 5}, {y, -10., 10.}}, .00000001, Starts → 1, ShowProgress → False] // Timing

Out[90]= {0.062, {{{x → 0.999992, y → 0.999967}, 3.02457×10^-8}}}

In[91]:= GlobalPenaltyFn[100. * (x^2 - y)^2 + (1. - x)^2, {}, {},
           {{x, -5, 5}, {y, -10., 10.}}, .00000001, Starts → 1, ShowProgress → False] // Timing

Out[91]= {0.032, {{{x → 1., y → 1.}, 0.}}}
```

```
In[92]:= IntervalMin[100. * (x^2 - y) ^2 + (1. - x) ^2, , ,
           {Interval[{0., 3.1}], Interval[{0., 3.2}]}, {x, y}, .0000001] // Timing

Out[92]= {18.796, {{x → Interval[{0.992646, 0.992652}], y → Interval[{0.985332, 0.985338}]},
           Interval[{0.0000540026, 0.0000541503}]}}
```

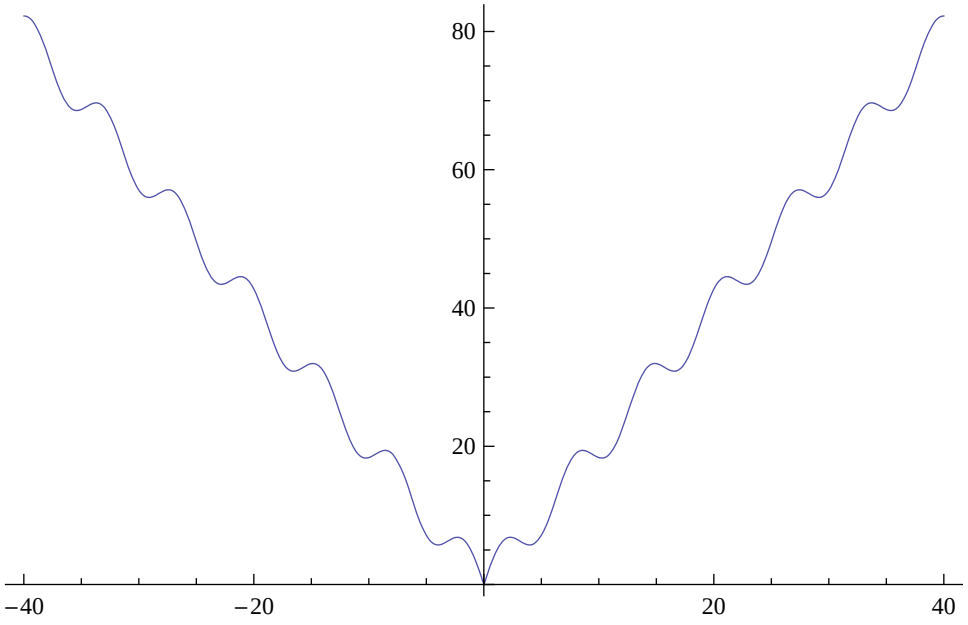
Such extremely flat functions can be slow for IntervalMin to solve. In this case, IntervalMin did pretty well, but did not get the exact answer. If we restart with narrower Intervals around the above solution, we get a better result:

```
In[304]:=
IntervalMin[100. * (x^2 - y) ^2 + (1. - x) ^2, , ,
  {Interval[{.98, 1.1}], Interval[{.97, 1.1}]}, {x, y}, .00000001] // Timing

Out[304]=
{54.7 Second, {{x → Interval[{1.00203, 1.00203}], y → Interval[{1.00407, 1.00407}]},
  Interval[{4.12126 × 10^-6, 4.13963 × 10^-6}]}}
```

Of particular interest is how the function performs on highly irregular functions. The following irregular function is tested:

```
In[93]:= Plot [Abs[2 x + 3 Sin[x]], {x, -40, 40}]
```

Out[93]=

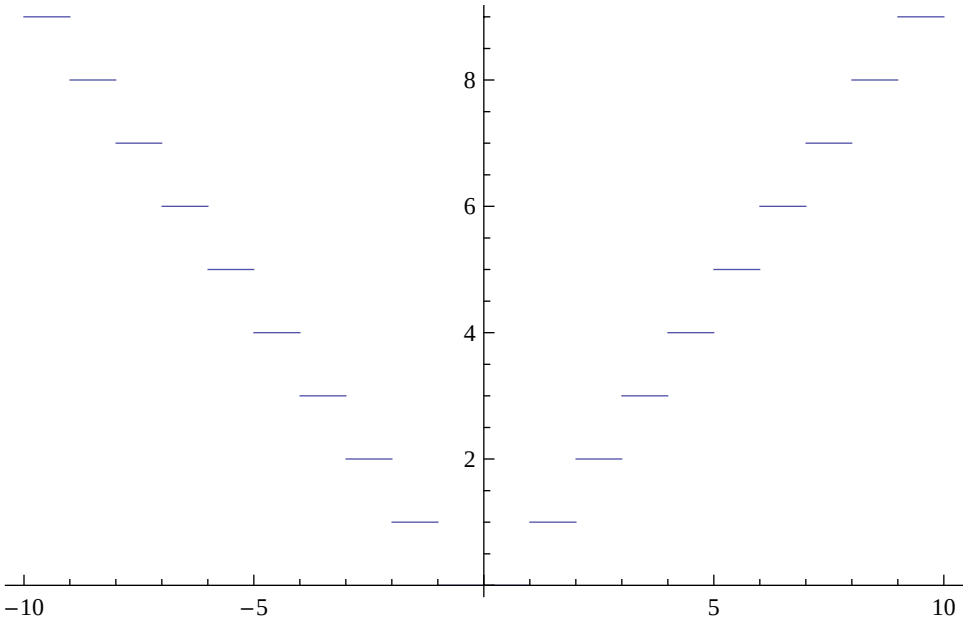



```
In[94]:= GlobalSearch[Abs[2 x + 3 Sin[x]], , , {{x, -40, 40}}, .0000001, Starts -> 1] // Timing

Out[94]= {0.015, {{{x → -1.93805 × 10^-9}, 9.69023 × 10^-9}}}
```

All 10 starting values found the solution to this problem. In the next problem, a function is tested that is step-wise discontinuous:

```
In[95]:= Plot[Abs[IntegerPart[i]], {i, -10, 10}]
```

Out[95]=




```
In[96]:= GlobalSearch[Abs[IntegerPart[x]], , ,
          {{x, 0, 100}}, .1, Starts → 1, CompileOption → False]

Out[96]= {{{x → 0.989767}, 0}}
```

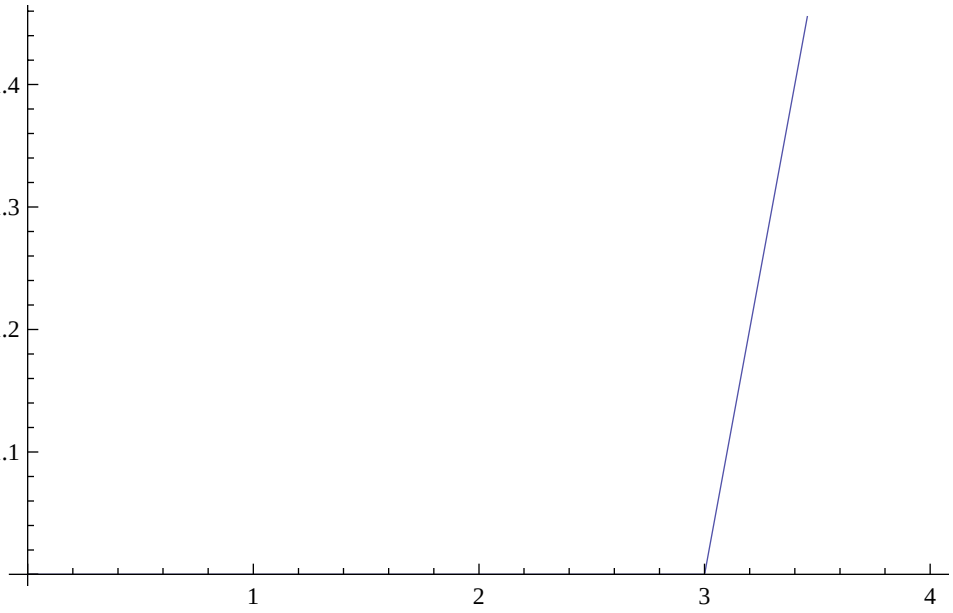
The solution is 0 between -1 and 1, so a valid solution was found. We can also solve this using an integer solution:

```
In[97]:= MultiStartMin[Abs[x], , , {{x, 0, 100, Integer}}, .1, Starts → 1]

Out[97]= {{{x → 0}, 0.}}
```

Problems can be solved that cause FindMinimum to fail. In the following, any number below 3 has the minimum function value of 1.

```
In[98]:= Plot[Max[1, x - 2], {x, 0, 4}]
```

Out[98]=



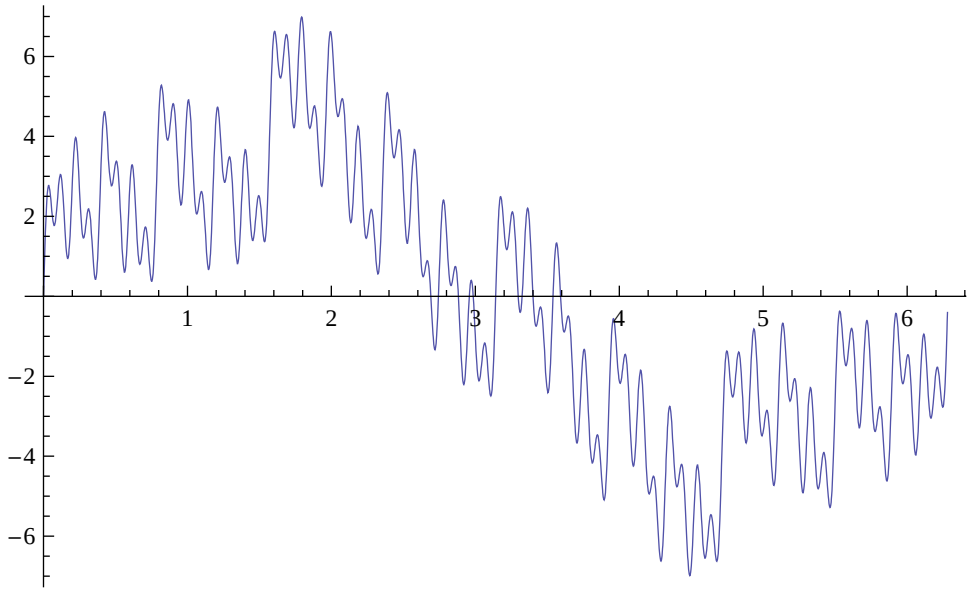

```
In[99]:= FindMinimum[Max[1, x - 2], {x, 5, 6}]

Out[99]= FindMinimum[Max[1, x - 2], {x, 5, 6}]

In[101]:=
GlobalSearch[Max[1, x - 2], , , {{x, 5, 9}}, .1, Starts → 1, CompileOption → False]

Out[101]=
{{{x → 2.51486}, 1}}
```

In the next problem, a more irregular function is tested:

In[102]:=

```
Plot [4 Sin[x] + Sin[4 x] + Sin[8 x] + Sin[16 x] + Sin[32 x] + Sin[64 x], {x, 0, 6.28}]
```

Out[102]=



In[103]:=

```
GlobalSearch[4 Sin[x] + Sin[4 x] + Sin[8 x] + Sin[16 x] + Sin[32 x] + Sin[64 x],
  {-x, x - 6}, , {{x, 0, 6.28}}, .000001, Starts -> 10] // Timing
```

Out[103]=

```
{0.359, {{{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759},
  {{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759},
  {{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759}}}
```

Many of the starts found the minimum (-6.98759) and the rest were quite close. We see next that IntervalMin fails on this problem because the way terms are handled separately does not allow the algorithm to narrow down the range.

In[104]:=

```
IntervalMin[4 Sin[x] + Sin[4 x] + Sin[8 x] + Sin[16 x] + Sin[32 x] + Sin[64 x],
  {}, , {Interval[{0., 6.28}]}, {x}, .000001] // Timing
```

Out[104]=

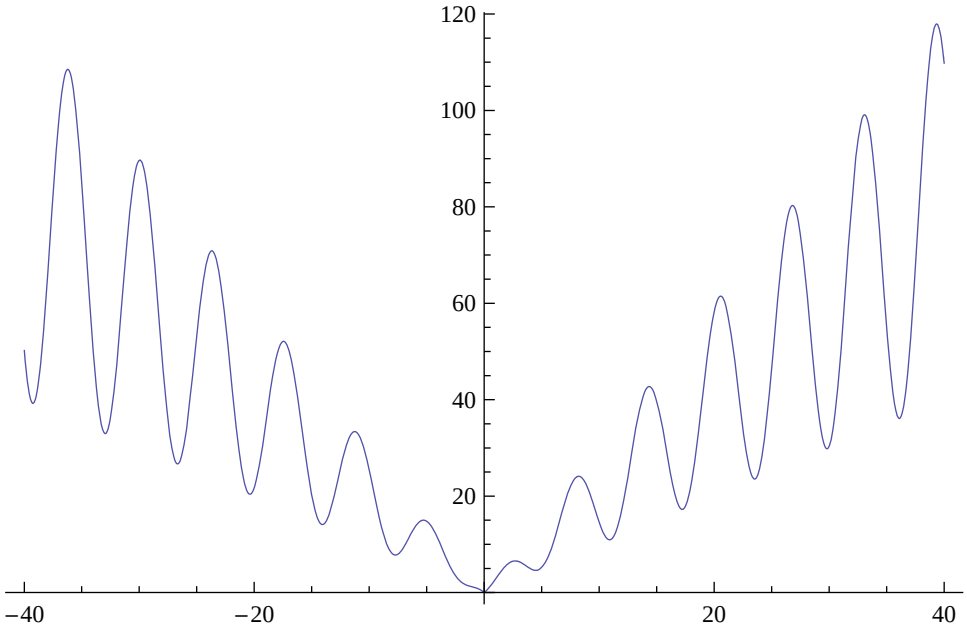
```
{0.016, {{x → Interval[{-2.22507 × 10^-308, 6.28}]}, Interval[{-9, 9}]}}
```

In the next problem, the scale of the fluctuations makes solution even more difficult than in the above problem. Nevertheless, all ten starts find the solution.

In[105]:=

```
Plot [Abs[2 x + x * Sin[x]], {x, -40, 40}]
```

Out[105]=



In[106]:=

```
GlobalSearch[Abs[2 x + x * Sin[x]], , , {{x, -40, 40}}, .00000001, Starts -> 10] // Timing
```

Out[106]=

$$\{0.125, \{\{\{x \to -6.70087 \times 10^{-9}\}, 1.34017 \times 10^{-8}\}, \{\{x \to -2.93322 \times 10^{-9}\}, 5.86643 \times 10^{-9}\},$$
$$\{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\}, \{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\},$$
$$\{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\}, \{\{x \to -1.84432 \times 10^{-9}\}, 3.68865 \times 10^{-9}\},$$
$$\{\{x \to -6.54427 \times 10^{-9}\}, 1.30885 \times 10^{-8}\}, \{\{x \to -3.12961 \times 10^{-9}\}, 6.25921 \times 10^{-9}\},$$
$$\{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\}, \{\{x \to -2.81907 \times 10^{-9}\}, 5.63814 \times 10^{-9}\}\}\}$$

All 10 solutions are correct. With a smaller range for the initial guess, the problem is still solved correctly for most of the starts.

In[107]:=

```
GlobalSearch[Abs[2 x + x * Sin[x]], , , {{x, 38., 39}}, .000001, Starts -> 10] // Timing
```

Out[107]=

$$\{0.125, \{\{\{x \to -7.10874 \times 10^{-9}\}, 1.42175 \times 10^{-8}\}, \{\{x \to -1.61639 \times 10^{-9}\}, 3.23278 \times 10^{-9}\},$$
$$\{\{x \to -6.99283 \times 10^{-9}\}, 1.39857 \times 10^{-8}\}, \{\{x \to -5.65239 \times 10^{-9}\}, 1.13048 \times 10^{-8}\},$$
$$\{\{x \to -8.2679 \times 10^{-11}\}, 1.65358 \times 10^{-10}\}, \{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\},$$
$$\{\{x \to -1.82016 \times 10^{-10}\}, 3.64033 \times 10^{-10}\}, \{\{x \to -1.22585 \times 10^{-10}\}, 2.4517 \times 10^{-10}\},$$
$$\{\{x \to -5.01469 \times 10^{-9}\}, 1.00294 \times 10^{-8}\}, \{\{x \to -7.45058 \times 10^{-9}\}, 1.49012 \times 10^{-8}\}\}\}$$

However, with a small enough range for the initial guess, the problem may not be solved because the solver may be converted to a local solver. We see next that IntervalMin actually solves this problem quickly and correctly brackets zero.

```
In[108]:=
IntervalMin[Abs[2 x + x * Sin[x]], {}, , {Interval[{-40., 40}]}, {x}, .0000001] // Timing
```

```
Out[108]=
{0.047, {{x → Interval[{-6.66134 × 10^-16, 6.10623 × 10^-16}]}, Interval[{0, 1.33227 × 10^-15}]}}
```

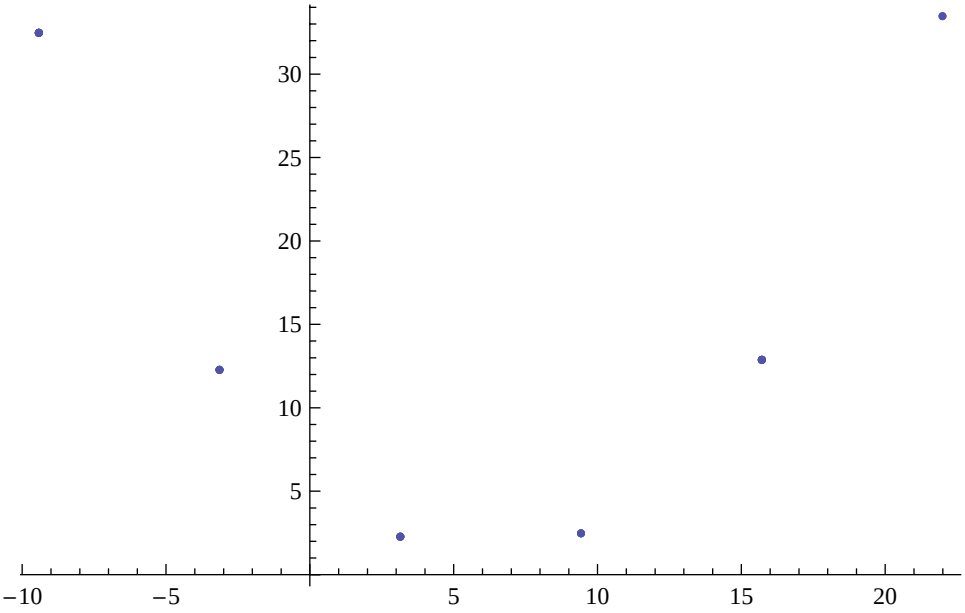
In the next test, a problem with multiple minima, the Branin rcos function, is solved.

```
In[109]:=
b = GlobalSearch[(y - x^2 * 5.1 / (4. * π^2) + 5. * x / π - 6.)^2 +
    10. * (1. - 1. / (8. * π)) * Cos[x] + 10., {-40 - x, x - 40, -40 - y, y - 40}, {},
   {{x, -40, 40}, {y, -40, 40}}, .000001, Starts -> 30, CompileOption -> True];
```

```
In[110]:=
c = {}; Do[AppendTo[c, {x, y} /. b[[ii, 1]]], {ii, 1, 30}]
```

```
In[111]:=
ListPlot[c]
```

```
Out[111]=
```



All six solutions to this problem were found with 30 starts. In the next test, the Csendes function is tested, which has a highly dissected but flat region around the optimum, shaped like an ashtray (see next section for illustrative figures).

```
In[112]:=
GlobalSearch[x^6 * (2.0 + Sin[1.0 / x]) + y^6 * (2.0 + Sin[1.0 / y]), , ,
  {{x, -1., 1.}, {y, -1., 1.}}, 0.0000000000000000001] // Timing
```

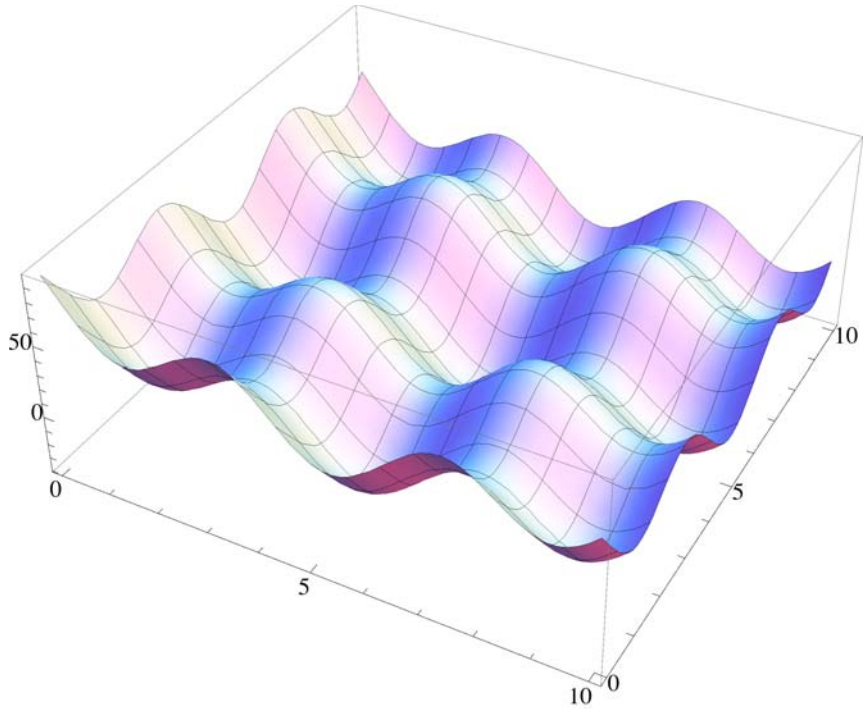
```
Out[112]=
{0.156, {{{x → 0.0000575029, y → -0.0000435469}, 5.67993 × 10^-26},
  {{x → -0.00036275, y → -0.000500678}, 4.92 × 10^-20},
  {{x → -0.000417382, y → 0.000259364}, 6.12527 × 10^-21}}}
```

We can see that the minimum point {0,0} is being approached with arbitrary accuracy. In the test of this function with the GlobalMinima function below, we see that GlobalMinima can be used to define the flat region around the minimum.

The following function has many local minima:

```
In[113]:=
Plot3D[20 Sin[Pi/2 (x - 2 Pi)] + 20 Sin[Pi/2 (y - 2 Pi)] + (x - 2 Pi)^2 + (y - 2 Pi)^2,
 {x, 0, 10}, {y, 0, 10}]
```

Out[113]=



We can see that this could be difficult to find. It is known that the solution is -38 at {5.322,5.322}.

```
In[114]:=
GlobalSearch[20 Sin[Pi/2 (x - 2 Pi)] + 20 Sin[Pi/2 (y - 2 Pi)] + (x - 2 Pi)^2 + (y - 2 Pi)^2,
  {}, , {{x, 0, 10}, {y, 0, 10}}, .000000001, Starts → 4] // Timing
```

```
Out[114]=
{0.094, {{{x → 5.32216, y → 5.32216}, -38.0779}, {{x → 5.32216, y → 5.32216}, -38.0779}}}
```

GlobalSearch is able to find the true minimum when several starts are used. IntervalMin solves this problem reliably:

```
In[115]:=
IntervalMin[20 Sin[Pi/2 (x - 2 Pi)] + 20 Sin[Pi/2 (y - 2 Pi)] + (x - 2 Pi)^2 + (y - 2 Pi)^2, , ,
  {Interval[{0., 10.}], Interval[{0., 10.}]}, {x, y}, .0001] // Timing
```

```
Out[115]=
{2.687, {{x → Interval[{5.32211, 5.32214}], y → Interval[{5.32211, 5.32214}]},
  Interval[{-38.078, -38.0778}]}}
```

IV THE GlobalMinima FUNCTION

IV.1 Introduction

This function represents a robust approach to global optimization. Traditional gradient (local) approaches require the user to know how many optima are being sought for the function to be solved, and roughly where the optima are, so that a good initial guess can be made. The user, however, rarely can make a good initial guess and has no information about the existence of multiple minima. In the absence of such information, existing algorithms will generally converge to an answer, but this is only a local solution that may not be globally optimal. Furthermore, even global optima may not be unique. In addition, if the region around the optimum is very flat, most algorithms will report difficulty with convergence. In reality, such a flat region, for real-world problems, may indicate the existence of an indifference zone of equally good solutions.

This function is designed to overcome these difficulties. It uses the Adaptive Grid Refinement (AGR) algorithm (an n-dimensional adaptation of Newton's method related to adaptive mesh generation) which finds multiple optima and defines optimal regions in a single run. Because the user does not need to know much about the function, the AGR algorithm saves considerable time for the user. It is not necessary to know how many optima exist or where they are; the AGR algorithm will generally find them all. Feasible starting points do not need to be provided. The AGR algorithm does this at some cost in computer time by making more function calls, and is thus limited to smaller problems. The AGR algorithm also increases the user's certainty that all useful solutions have been found. The genetic algorithm method can find multiple optima, but only stochastically, it does not allow constraints, it is difficult to use, and does not define optimal regions. Other global methods that are available are either difficult to use, require special conditions for the user function, or do not allow constraints. These global methods also do not generally provide information on solution regions, but rather assume that only optimal points are being sought. Compared to existing algorithms, the AGR algorithm is the easiest to use, is the most general, and provides the most information. It is not feasible for larger problems, however, in which case GlobalSearch, GlobalPenaltyFn, IntervalMin, or MultiStartMin should be used.

IV.2 The Grid Refinement Approach to Nonlinear Optimization

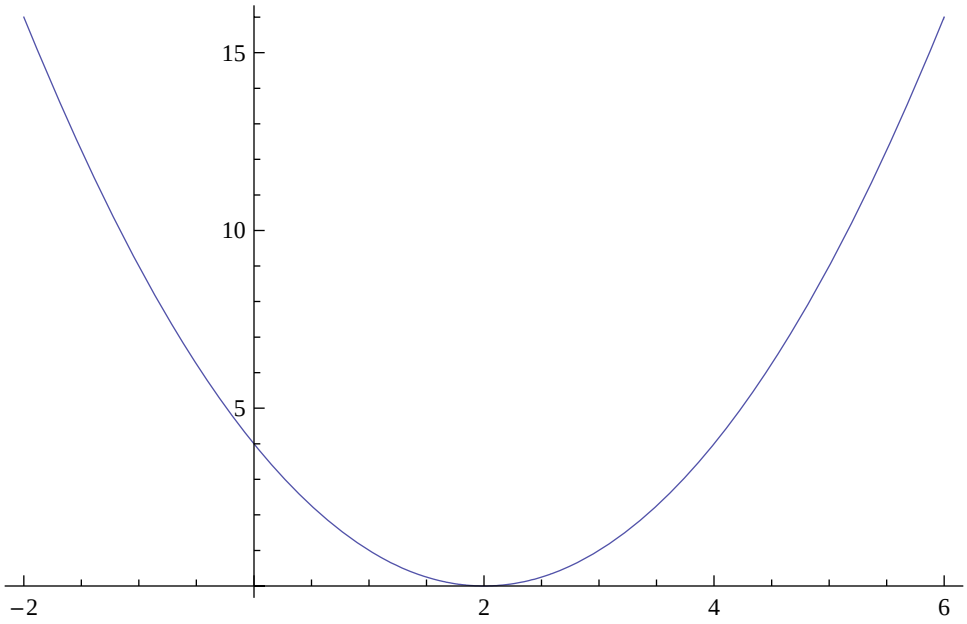
The AGR algorithm works by a grid refinement process. Consider a function

(x-2)^2 (16)

where we wish to minimize z. In this case, the answer is obviously z= 0 at x= 2, as can be seen from the plot:

```
In[116]:=
     Plot[(x - 2)^2, {x, -2, 6}]
```

Out[116]=



The Adaptive Grid Refinement (AGR) algorithm works as follows. The interval to be searched for a solution (say, -100 to 100) is gridded in this case into n initial grid points (with n = 11 being a useful initial grid density) at a distance d = 18.18 units apart. At each x point, z is evaluated. The best points (with the lowest z values) are kept, and the rest are thrown away. At each x kept, new points (daughters) are evaluated on each side at one-third the distance between the first set of points, and z is again evaluated. This one-third trisection prevents duplication of points during daughter point generation. This process of grid refinement continues until a user stopping criterion is met, when all optimal points are displayed. The same procedure is used for any number of dimensions. In two dimensions, each point generates eight daughter points. In three dimensions, 26 points are generated as the corners of a hypercube. The grid refinement algorithm is in essence a generalized-descent approach. At each iteration, the population of points in the working set is moving downhill, but over multiple regions and directions. Consider

$$\mathbf{min}\,(z) = \sum_{i=1}^{2} x_i^{\,2} \tag{17}$$

for a two-dimensional (2D) problem. For an initial grid that is symmetric around 0, points in all four quadrants will remain at each iteration and will converge toward 0 from all four directions, very much like four steepest-descent vectors.

The AGR algorithm is derivative-free and uses no search vectors. It is therefore more numerically stable than other algorithms. For example, discontinuities (such as step functions) do not cause problems, nor do evaluations in the vicinity of a constraint boundary. The algorithm uses very little computational time in operations other than function calls. The size of the problem (the number of dimensions) is limited only by execution time. The transparency of the algorithm's operation makes this tool ideal for teaching concepts of optimization to introductory college classes or high school math students. No advanced mathematical training is required to understand and use it.

The AGR algorithm is similar to a variety of interval methods that have been formally shown to converge (Pinter,

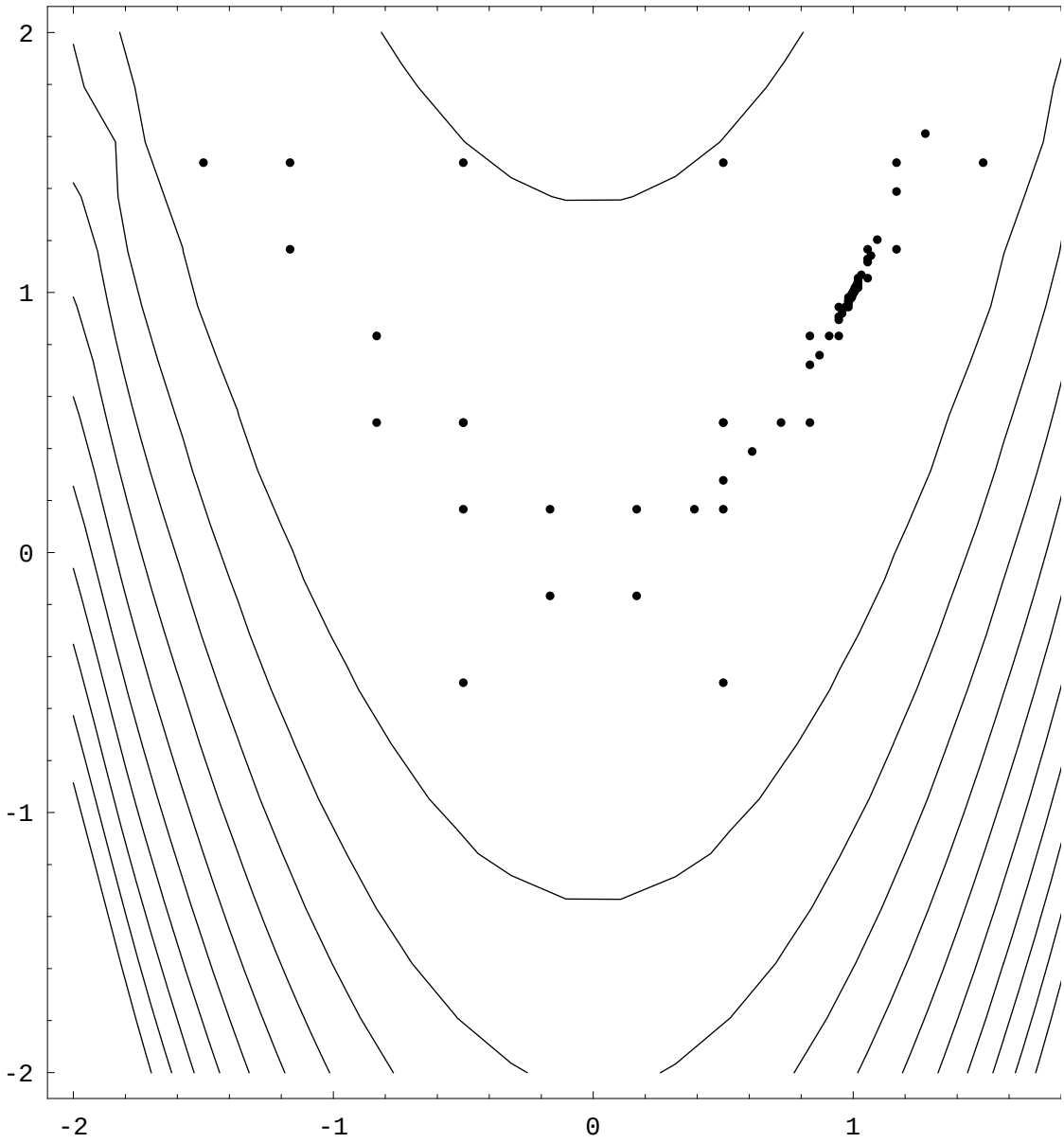
1996). AGR also has certain similarities to Taboo Search (Cvijovic and Klinowski, 1995) and Jones' algorithm (Jones et al., 1993).

In the worst case, with the optimum falling exactly halfway between two initial grid points, the error estimate of the optimum will decrease as

$$\frac{1}{2 * 3^{i}} (x_{max} - x_{min}) / n \qquad (18)$$

where i is the number of iterations, the min and max terms reflect the bounds on the search region for the choice parameter x, and n is the number of initial grid points. Thus, if the initial grid is at 10 unit intervals, after 6 iterations we can estimate the optimal parameter value to within ±0.000686. After 10 iterations, we can achieve ±0.00000847. On average, a point will fall away from this worst case scenario, and convergence will be slightly faster.

We can illustrate the grid refinement procedure with the Rosenbrock function. In the following figure, the contours of the function are shown. The points represent the values kept after the initial gridding and after each subsequent grid refinement step. We can see the rapid convergence to the solution at {1,1}.



Grid refinement algorithms tend to increase their computational costs exponentially and can use too much memory space. In the AGR algorithm, memory constraints are overcome by judicious memory management and by repeated pruning of the working solution set during algorithm operation. Execution time presents more of a problem. On a fast PC, small problems run in seconds, problems with up to 7 variables run in minutes, problems from 8 to 11 variables run in up to hours, and problems of 12 to 15 variables run overnight. This is because millions of function calls may need to be made in higher dimensions. The AGR algorithm is therefore not suitable for functions whose solution is itself time consuming, such as optimizing the output of a simulation model. On a fast Unix machine, up to 20 variables might be feasible. Although the practical upper limit of 20 variables prevents this algorithm from being universal, its ease of use and ability to find all optima and optimal regions certainly recommend it for the large subset of problems that fall within its domain.

IV.3 Program Operation

IV.3.A Parameters and Default Values

The user must input several parameters. The initial grid density defines the initial gridding on each x dimension. Thus, for a 2D problem, an initial grid density of 11 x 11 will give 11^2 = 121 initial function calls. As the dimensionality increases, it is advisable to reduce the initial grid density to reduce execution time. On the other hand, the more (true or false) minima there are likely to be and the more poorly behaved the function, the larger the initial grid density should be (15 x 15, 20 x 20, or even higher). Otherwise, minima may be missed. A high grid density with a high dimensionality may not be feasible, since initial grid exploration requires a function call at the number of grid points raised to the power of the number of dimensions.

$$n = grid^d \tag{19}$$

The second input parameter, the contraction coefficient (C), defines the degree of contraction of the grid region on each iteration. For the largest (z_l) and smallest (z_s) function values at the current iteration, a point z_i will be kept for further refinement only if

$$z_i < z_s + (z_l - z_s)\, C \tag{20}$$

That is, for C = 0.1, the only points that will be kept are less than 10% of the range larger than the current minimum. If local minima are expected to be a problem, values of 0.25 to 0.5+ will help prevent the true minima from being missed. The parameter C is cut in half twice as the program proceeds.

The tolerance (T) defines the amount of improvement (absolute, not relative) in the smallest z required at each iteration. If at least this much improvement is not found, the program stops. The tolerance can be set to very small values, such as 10^{-15}, to achieve a highly accurate estimate of the minimum. If the user sets tolerance = 0, the program will stop if the minimum remains the same for two iterations, but if there is a very gradual slope near the minimum and slight improvement is achieved with each iteration, the program will not terminate. Thus, tolerance values of zero are not advisable. To select a good tolerance value, the user can make an initial run with a large tolerance to see if multiple optima exist, and then make a second run with a smaller tolerance and bounds that are closer around the rough estimate of the optimum (or around each solution region if there are several).

A second solution criterion is provided by the indifference (I). When the tolerance criterion causes the program to stop, all points from intermediate iterations with values between z_s and z_s + I are considered part of the optimal region(s). This is because the user is often indifferent to extremely small differences in z, but may be interested in the existence of multiple solutions that are (within I) equally good. Thus, for a steel mill, there may be a range of furnace temperatures within which the output steel quality is approximately the same. There is no point in trying to control the furnace temperature closer than this target zone; in fact, such control is likely to be costly or impossible. In general, for real-world optimization problems the existence of zones of indifference is extremely important information, because it is far easier to control a process within some range than to keep it exactly at some "optimal" point. Only the AGR algorithm provides information on zones of indifference to guide real-world decision making. Consider the function

$$z = x^4 \tag{21}$$

with I = 0.1, which is minimized at x = 0. All values of x between -0.562 and +0.562 produce a z less than I, and the user is thus indifferent to these differences. The "answer" is therefore the range -0.562 < x < 0.562 and not just the point x = 0. The AGR algorithm provides an approximation to this range that improves as initial grid density improves If only the

absolute minimum point is being sought, I can be set to zero. If I is set very large, the solution set may define a large region containing many thousands of points.

For higher-dimensional problems, initial grid density may be so coarse that a full picture of the indifference region is not found. This situation will be evident if there are few points in the indifference region relative to the dimensionality. The AGR algorithm can be reapplied to a smaller domain around a particular optimum by setting the ranges for x_i closer to this zone than in the initial run in order to better define the indifference region.

The AGR algorithm is implemented in *Mathematica* via the GlobalMinima command, which has eight options. The defaults are MaxIterations->25, CompileOption->True, ShowProgress->True, StartsList->{}, EvaluateObj->True, SimplifyOption->True, UserMemory->32, and Starts->3. MaxIterations prevents the program from running away to infinity when a mistake is made in the function (Min[x] for example). For very computationally expensive problems (like those with many parameters), it is useful to use Starts->1 to evaluate the time a run takes. For general usage, Starts->3 (the default) is good. The results of this short run can then be used to define further, more directed runs based on whether all 3 starts found the same solution. A list of starting values can be input instead of letting the program find them with random search (e.g. StartsList->{{1.,2.},{2.,1.}}). The CompileOption determines whether the user function and constraints will be compiled. While Compile reduces execution time, some functions can not be compiled, in which case CompileOption->False should be used. If the objective function should not be Evaluated, use EvaluateObj->False. SimplifyOption attempts to simplify the objective function. If this should not be done or will not improve the solution, this should be set to False. UserMemory, in units of Megabytes, is a user input defining the free memory available on their machine. GlobalMinima has several ways of executing, depending on how much memory is available. If more memory is available, it runs faster.

IV.3.B Bounds

Whereas local algorithms require input of an initial guess or starting point (which must be feasible) to initiate the search, GlobalMinima requires only bounds on the variables. Bounds are generally easier to define than are feasible starting points. Physical, logical, or economic considerations often provide guidance on realistic bounds for a variable. If the optimization problem is wing design and one variable is wing thickness, a certain range of thicknesses is sensible to consider. For a financial problem, the range of values for an investment mix across a product line is clearly bounded between zero and the total budget, and the upper limit can generally be bounded below a much smaller value than this. If narrower bounds are known, then convergence will require fewer function calls, but even quite wide bounds are acceptable. GlobalMinima can not search outside of the bounds, and if the solution found runs right against an input bound, then the true solution may lie outside the bounds and another run may be required if the bounds do not actually represent a constraint.

IV.3.C Constraints

Constraints are entered as *Mathematica* functions, which may be linear or nonlinear. Constraints may be any logical function that defines a region or regions of space. Constraints are entered as a list. The terms of the inequalities can be any valid *Mathematica* inequality expression. For example, we can define a 3D, doughnut-shaped search region (a sphere with a hollow middle) as follows:

```
In[66]:= 1. < x^2 + y^2 + z^2 < 10.;
```

In contrast to gradient search algorithms, for which constraints are a real problem, GlobalMinima handles constraints easily. In fact, constraints, by limiting the space to be searched, may greatly reduce the number of function calls. When constraints are involved, it is advisable to increase the initial grid density somewhat. Any number of constraints is allowed, but equality constraints are not allowed. Equality constraints can be approximated by a bounding constraint such as:

```
In[67]:= 0.9 < x1 + x2 + x3 < 1.1;
```

which becomes in standard form:

```
In[67]:= {x1 + x2 + x3 - 1.1, -(x1 + x2 + x3) + 0.9};
```

but in this case, a higher initial grid density is needed to obtain points that fall in the narrow feasible zone.

Note that simple bounds on the parameters are handled by the variable definition terms {x, x_{min}, x_{max}}. Duplicating these bounds as constraints is valid but is unnecessary and will slow program execution. To be specific, no nonnegativity constraints are needed, as these are handled by the bounds inputs.

WARNING: Constraints can cause difficulties for GlobalMinima. Problems are caused by the creation of unreachable regions of the search space by the constraints. In an unconstrained problem, the gridding is such that the entire search space can be reached by iterating from the initial grid configuration. This may not be true when constraints are present. This can happen in several ways. First, the feasible space may be defined such that no initial grid points fall in the feasible region. For example, for bounds {-100,100} with a gridding of 5, a constraint {.1<x<.2} will be such a small space that the initial grid won't fall into it except with a very high initial gridding. If this happens, GlobalMinima will exit with an error message. Second, a small unreachable space can be created right against the constraint boundary. For example, for initial grid points {-5,0,5}, the point 0 provides access to the region -2.5<x<2.5. If a constraint is entered {x>1}, the region {1>x>2.5} is no longer reachable, since only the grid point {5} is available for grid refinement. Thus in this case, the smallest value of x that can be found is 2.5 if the minimization tends to push the solution up against the constraint. The solution to this difficulty is to rerun the problem with tighter bounds focused around the solution found, with a higher grid density and/or a shift in the grid. For example a shift from 4 to 5 grid points will alter the intersection between the gridding and the constraint region, as will a zoom in or an alteration in the left or right bounds.

IV.3.D Memory Usage

The algorithm used by the program is able to trade memory for speed. This means that if more memory is available it will run faster. The default for the option UserMemory is 32 (in units of Megabytes). If your machine has more memory than this, setting UserMemory to the higher value will likely make it run faster. If the kernel runs out of memory, try quitting all other applications. If the kernel still runs out of memory, it may be necessary to restrict UserMemory to smaller values to force GlobalMinima to use the computational path that requires less memory but takes longer.

IV.4 Output

During program execution, output showing the size of the current solution set and the current minimum is optionally printed at each iteration. ShowProgress->True will print out these intermediate results, which provide information for evaluating convergence and whether the contraction coefficient is too tight or too loose. Especially for higher dimensional problems, this intermediate output should always be printed. Following solution, a list is returned that contains the solution. For example, two points in a 2 parameter problem would give the list {{{1.1,1.2},-5},{{1.2,1.21},-4.9}}. The total number of function calls is also printed. Output is illustrated below.

IV.5 Maximization

The algorithm assumes that z is being minimized. If a function is to be maximized, simply minimize -z. The results will then be in terms of the correct parameter values, but the z values will be negatives of their true values.

IV.6 Limitations

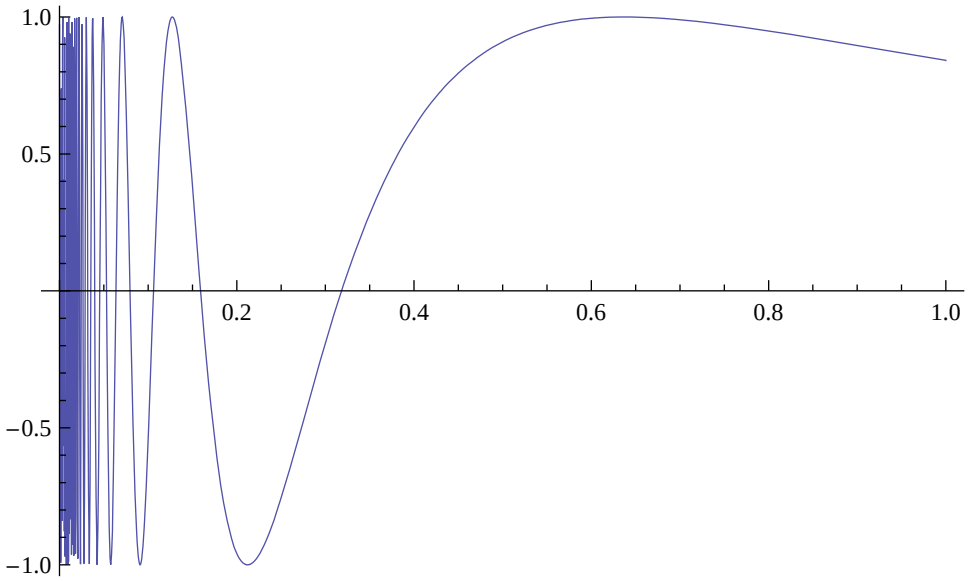
No optimization algorithm can solve all conceivable problems. Consider a response surface in two dimensions that is flat everywhere except for a very small but very deep pit at one location. Steepest-descent algorithms will fail completely to solve this problem, because for almost all starting points there is no gradient. The algorithms will just stop and say that the gradient is zero. GlobalMinima will stop after two iterations because no reduction has been made in the minimum, and it will identify points across the entire region as valid. This result tells the user that the surface is very flat. The next step that can be taken, if there is reason to believe that an optimum does exist somewhere, is to increase the grid density and run GlobalMinima again. Initial densities > 1000 are feasible for problems with few dimensions and may by chance find the narrow optimal zone. Once this zone is identified, convergence is rapid. There is thus at least a chance that GlobalMinima can solve such an "impossible" problem.

Some functions have infinitely many solutions. Consider Sin(1/x) between 0 and 1. As the function approaches zero, the minima (with z = -1) get closer and closer together, as we can see from the plot following:

In[117]:=

```
Plot[Sin[1/x], {x, 0, 1}]
```

Out[117]=



Although it is fundamentally impossible to list all solutions when there are infinitely many, GlobalMinima can show that there are very many solutions. If we try an initial grid density of 20 on the interval [0, 1], with contraction = 0.4, we obtain many points with values close to -1.0 (see section IV for details). This result demonstrates that there are many solutions and that the density of solutions increases with an approach to zero. This is thus an approximate solution to the true situation. One can determine that this is an approximate solution by increasing the grid density to find more solutions close to zero. Making two runs allows us to extrapolate the result that there are very many solutions close to zero, perhaps infinitely many (as in this case).

These examples illustrate that not all problems can be solved. Some are not solvable by any algorithm. Some may be ill-posed. However, for many problems that cause other algorithms to fail, the AGR algorithm either succeeds, has a probability of succeeding, or provides a partial or approximate solution. We may thus say that the set of unsolvable problems is much smaller for the AGR algorithm than for other solution methods. This is particularly so with respect to the inclusion of constraints.

IV.7 Efficiency

Although the number of function calls is a typical criterion for comparing the speed of optimization algorithms, this measure does not tell the entire story. Many algorithms do considerable computing for things such as derivatives and gradients and inverting matrices, whereas the AGR algorithm does very little such secondary computing. The AGR algorithm is thus faster than the number of function calls alone would indicate. A second important point is that the literature comparing algorithms often indicates that an algorithm had only some probability of finding a solution (for a random initial starting point). For real problems where the "correct" answer is not known, this means that the problem must be solved many times with different initial conditions to start the algorithm; sometimes hundreds or thousands of runs are necessary. Genetic algorithms and simulated annealing behave in this way (Ingber and Rosen, 1992). Thus, the actual efficiency is a function of the total number of runs, not just the single-run function call count or the execution time. This requirement for making multiple runs creates substantial work for the user. With GlobalMinima, a single run is sufficient to generate reliable solutions. In addition, it is easier to define feasible bounds on the domain of a problem than to generate hundreds of feasible starting points for multiple runs, especially if constraints are involved.

It is obvious that a grid refinement approach like that used here will require function calls as an exponential function of the number of dimensions. The initial gridding can be slow if too great a grid density is input. For example, for a five-dimensional problem with a grid density of 7, the initial number of grid points is $7^5 = 16,807$. After the initial grid calculation, execution time is governed by the generation of daughter points in the process of grid refinement. The formula for the number of daughters d generated by a point in the current solution set is:

$$\mathbf{d = 3^n - 1} \tag{22}$$

This gives d = {2, 8, 26, 80, 242 ...} for i = {1, 2, 3, 4, 5 ...}. With this formulation, large problems take too long to run. One approach to reducing execution time is to make the contraction coefficient smaller. For problems with only one minimum, this economy can cut execution time by 0.5 to 0.9. However, this strategy will cause multiple optima to be missed. The grid-based approach used here is thus obviously not suitable for very large problems, but does allow for constraints, even nonlinear constraints. Therefore, a contraction coefficient much below 0.05 is not recommended unless the function is likely to be smooth. For large problems or problems with complex constraints, the MultiStartMin function is recommended.

IV.8 Error Messages

An effort has been made to trap as many errors as possible. Much of the setup for running GlobalMinima uses *Mathematica* formats. Many user errors will prompt *Mathematica* error messages. On input, only the order of parameters is evaluated in the passing function call, not their names. The user is thus free to use any names, though the names used here and in the example notebooks are the most obvious and will help prevent mistakes. Therefore using "c" for "contraction" is not an error. If valid values for input parameters (tolerance >= 0.0, indifference >= 0, 0.0 < contraction < 0.8) are not input, the following error message is printed, and the program stops:

Error found by GlobalMinima

Error: Input parameter outside range

If a parameter is omitted from the calling sequence, *Mathematica* cannot continue. To indicate the problem, it echoes back the names for functions and values for constants in the function call, and then it stops. This type of echoed-back output means that a format error exists in the calling sequence.

After input, the program compiles the user input function and the constraints, speeding execution by a factor of about six. If these functions are improperly formatted, this step may produce a compilation error message, which will terminate the run. If a function is defined such that it does not return numeric values, this will cause error messages from *Mathematica*, followed by termination of GlobalMinima.

If a completely flat function is mistakenly defined, every point that is tried will be kept, and the solution set will keep growing. The program traps this type of error and exits with a message that the response surface is completely flat after the initial gridding. If a flat function is input for a large problem with a high initial grid, all available memory could be used up before GlobalMinima can detect that the function is flat.

When constraints are used, there may be no feasible region, particularly if the problem has many constraints. If this occurs, the following error message is printed, and the program terminates:

Error found by GlobalMinima

Error: No valid grid points generated

In this case, check the constraints and their relation to the bounds on the parameters. Constraints and bounds can be in conflict if care is not taken. In some cases, the constraints are valid, but a higher initial grid density is needed to obtain feasible points.

A common mistake is to maximize when the intention was to minimize. This error can be recognized because the solution will tend to run into one of the parameter bounds and will have an illogical z value. For example, the optimal airplane wing thickness for a design problem might come back as zero.

An unbounded solution may result from an improperly formulated problem (e.g., min(-1/x) over {-1, 1} which becomes -inf at x=0). Because the function z continues to get larger the closer one gets to zero at an increasing rate, the program will never report convergence. If this mistake has been made, GlobalMinima will terminate after MaxIterations and will print out results showing a very large (negative) function value.

The user function passed in to GlobalMinima is Compiled. This may cause compilation errors. The result is usually to stop the execution of the program, but it is necessary for the user to realize the source of the error. Compiled

functions can also generate errors at run time if the user function generates non-machine numbers such as high precision numbers or infinity. *Mathematica* will usually revert to the uncompiled function and continue running.

If the user defines variables in his program by mistake that should be parameters of the function, this will cause the program to stop or malfunction. For example, if pp=5 is defined by mistake, this will cause *Mathematica* to detect an error but GlobalMinima to attempt to execute:

```
In[118]:=
     pp = 5;

In[119]:=
     GlobalMinima[(pp - 30)^2 (y - 60)^2, , {{pp, 0., 100.}, {y, 1, 5}}, 20, 0.001, 0.3, 0.01]

        Error: Symbol expected in parameter list, variable 1

        Warning: memory is restricted for initial grid and performance will be affected.

        Please increase UserMemory if possible and close other applications.

        fatal error encountered, job stopped

Out[119]=
     $Failed
```

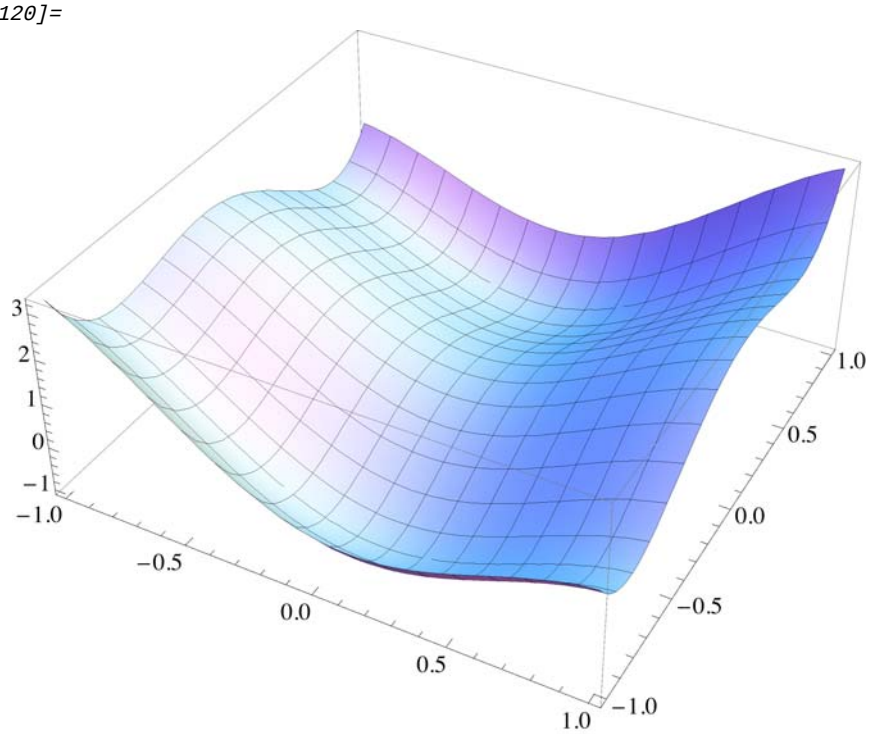
In this example, the parameter list {pp,0.,100.} has become {5,0.,100.} which does not match the function input sequence, causing the program to stop.

IV.9 A STEP-BY-STEP EXAMPLE

Our example is a function with multiple optima, the camelback function (Griewank, 19__):

```
In[120]:=
Plot3D[4*x^2-2.1*x^4+x^6/3+x*y-4*y^2+4*y^4, {x, -1, 1}, {y, -1, 1}]
```

Out[120]=



This function has two minima, which can be seen in the figure. GlobalMinima easily finds the two solutions. Even with an initial grid = 7 points and contraction = 0.3, GlobalMinima finds min = -1.293439 at the two solution points {0.4867, -0.74074} and {-0.4867, 0.74074}. We now illustrate how this solution is obtained.

GlobalMinima is a *Mathematica* package and must be installed by using the Get command. The function to be minimized must be defined; in this case, it is the camelback function depicted above. This is the first parameter for the function call. The second parameter is the constraints, input as a list of inequality functions separated by commas. If there are no constraints, an empty list must be entered as {}. When a set of constraints is entered, *Mathematica* returns a True or False value. GolobalMinima tests each potential grid point against the constraint equation set. The third parameter is the variable names and their bounds, in any order, each defined as a list. Each variable used in the function must be defined. The fourth parameter, the indifference parameter, defines the range of values larger than the best solution found that we will consider essentially equivalent and therefore equally good. We must define initial grid density (Integer) and tolerance (Real), where tolerance specifies the improvement in the solution necessary to continue iterating (i.e., if (old min - min) > tolerance, keep going). Note that this is an absolute rather than a relative tolerance.

The contraction coefficient is defined for input where the range is typically 0.5 to 0.1. The contraction coefficient is cut in half after the first iteration and in half again after the second. This is because the initial grid narrows the solution space to regions where faster winnowing of points is possible.

We can now execute GlobalMinima. Note that parameter names are arbitrary but that order is fixed. The option ShowProgress causes intermediate output to be printed. In this first example, no constraints are being used, so an empty list

{} is entered. By setting indifference very small, we only get the best solutions:

```
In[121]:=
     toler = 0.001;
     grid = 7;
     contraction = 0.25;
     indif = 0.001;
```

In[125]:=

```
GlobalMinima[4.*a1^2*a1^4+(1./3.)*a1^6+a1*a2-4.*a2^2+4.*a2^4, ,
 {{a1, -1., 1.}, {a2, -1, 1.}}, grid, toler, contraction, indif, ShowProgress -> True]
```

```
Global Optimization, Version 5.2

dimension= 2

grid density= 7

contraction= 0.25

indifference= 0.001

tolerance= 0.001

no constraints in use

full efficiency, memory needed: 147/50000

creating initial grid

max=-0.53242 min=-1.1186 number of nodes=16

daughters generated per node = 9

contraction coefficient cut=0.125

iteration 1

max=-1.21774 min=-1.28637 number of nodes=12

contraction coefficient cut=0.0625

iteration 2

max=-1.29207 min=-1.29278 number of nodes=4

iteration 3

max=-1.29338 min=-1.29344 number of nodes=4

iteration 4

max=-1.29369 min=-1.29369 number of nodes=2

iteration 5

max=-1.29371 min=-1.29371 number of nodes=2

total function calls=353

final results:
```

Out[125]=

```
{{{a1 → -0.4903, a2 → 0.736038}, -1.29371}, {{a1 → -0.4903, a2 → 0.737213}, -1.29369},
 {{a1 → -0.486772, a2 → 0.730159}, -1.29338}, {{a1 → -0.486772, a2 → 0.740741}, -1.29344},
 {{a1 → -0.47619, a2 → 0.730159}, -1.29278}, {{a1 → 0.47619, a2 → -0.730159}, -1.29278},
 {{a1 → 0.486772, a2 → -0.740741}, -1.29344}, {{a1 → 0.486772, a2 → -0.730159}, -1.29338},
 {{a1 → 0.4903, a2 → -0.737213}, -1.29369}, {{a1 → 0.4903, a2 → -0.736038}, -1.29371}}
```

GlobalMinima in this example provided stepwise reports on its progress. After echoing back the input parameters, it reports on the memory needed for computations. If more memory is available, the program will run faster unless it is already printing out "full efficiency". At each iteration, it reported the current minimum and maximum and the number of points in the solution set. For large problems, the execution time can become extremely high if too many points are being kept. In this case, it can be useful to make the contraction coefficient somewhat smaller. If a problem is known to be convex and well-behaved (e.g., $z = x^2$), a very small contraction (<0.05) will increase speed dramatically. A helpful type of printed output is a 2D plot of indifference-region values, for cases with two variables only. These plots help provide insight into program operation and the process of grid refinement and convergence.

The program can be run with more compact output by omitting the ShowProgress option, which then allows the default False to take precendence.

The camelback problem has two optimal solutions. A constraint that selects the lower right quadrant as a valid region finds just one of the two solutions, as shown below.

```
In[126]:=
     toler = 0.00001;
     grid = 7;
     contraction = 0.3;
     indif = 0.0001;

In[130]:=
     GlobalMinima[4. * a1^2 * a1^4 + (1. / 3.) * a1^6 + a1 * a2 - 4. * a2^2 + 4. * a2^4,
       {-a1, a2}, {{a1, -1., 1.}, {a2, -1, 1.}}, grid, toler,
       contraction, indif, ShowProgress → False] // Timing

Out[130]=
     {0.016, {{{a1 → 0.490169, a2 → -0.735907}, -1.29371},
       {{a1 → 0.490213, a2 → -0.735951}, -1.29371}, {{a1 → 0.4903, a2 → -0.737213}, -1.29369},
       {{a1 → 0.4903, a2 → -0.736038}, -1.29371}, {{a1 → 0.4903, a2 → -0.733686}, -1.29366}}}
```

We can see that only the solutions in one quadrant were found, as desired.

IV.10 Testing

No numerical algorithm can be guaranteed not to fail. However, the approach taken here is far more robust than other approaches. The algorithm has been tested with a number of both simple and complex functions and has been shown to perform well for a number of test functions that exhibit false minima, multiple minima, and minimal regions. Test functions have included positive polynomials with multiple roots, poorly behaved functions with large numbers of false minima, the Rosenbrock function, a number of multiple-minima problems, and many others. The only difficulty encountered is that for multiple-minima problems, some solutions could be missed if the initial grid density is too low. The algorithm should be stable for achieving high-precision results, because no derivatives are used, and testing has shown that high accuracy can be achieved easily. *Mathematica* uses high precision computations, and stores numbers as exact ratios of whole numbers whenever possible. Thus very little error is expected from truncation. Nondifferentiable functions, such as step functions, were tested and caused no numerical difficulties. The algorithm has no trouble with problems having up to dozens of true optima. For pathological functions with thousands of optima, the program may be able to find only scores of solutions in a particular run.

This package is not a tool for pure mathematics but rather is designed for applied problems. In the Csendes function discussed below, a box around the origin contains only solutions less than some value. Near the origin, values within the box may be 10^{-33} or less. GlobalMinima is designed to define this box and to converge near the best minimum. Conver-

gence to the absolute minimum (in this case, zero at x = 0) is of less interest, because all locations within the defined box region are essentially equally good (to 10^{-33}). It is thus difficult to compare the AGR algorithm directly to other algorithms, because other approaches seek out the analytical optimum point whereas the AGR algorithm seeks out the practically optimal region(s). What we can say is that although the AGR algorithm may not always find the analytical minimum of a function, it usually gets very close, and it can also find multiple minima and optimal regions, which other algorithms cannot.

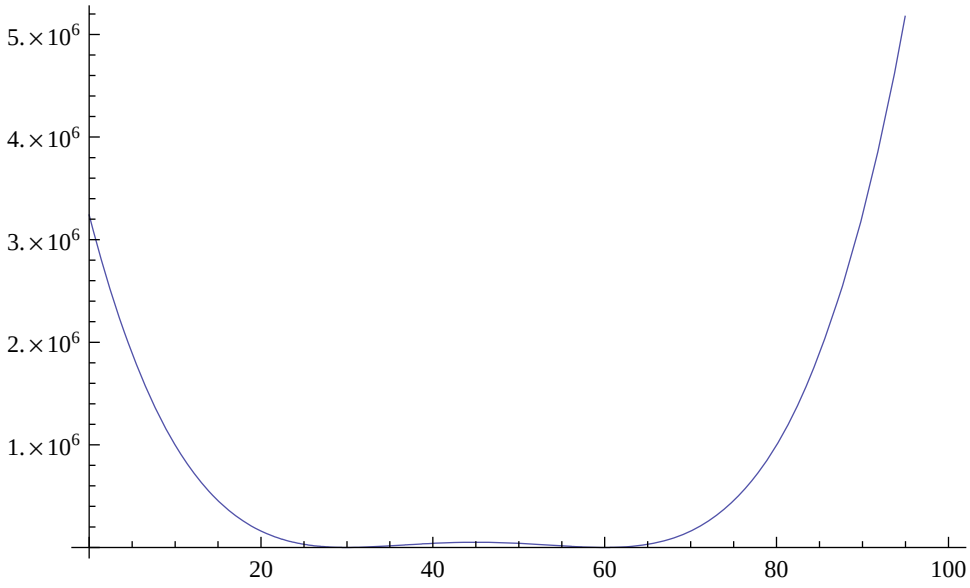
IV.10.A A Simple Polynomial.

The use of the package is best illustrated with examples. We begin with some simple cases. The function $\sum_{i=1}^{n} x^2$ is symmetric and for any number of dimensions has a global minimum at the origin with z = 0. This problem meets the classic criteria of gradient descent algorithms, which converge to the solution from any starting point for functions with no saddles, discontinuities, or false minima. GlobalMinima converges to the correct answer easily. Although the AGR algorithm is not as fast as a steepest-descent algorithm, it is quite fast for such well-behaved and steep problems.

A simple modification to this function creates a problem with multiple solutions:

```
In[132]:=
Plot[(x - 30)^2 (x - 60)^2, {x, 0, 100}]
```

Out[132]=



Here, z = 0 at x = 30 and at x = 60. When we run GlobalMinima over the interval {0, 100}, we need an initial grid density of 20 points to find both solutions:

```
In[133]:=
GlobalMinima[(x - 30)^2 (x - 60)^2, , {{x, 0., 100.}}, 20, 0.00001, 0.3, 0.0001] // Timing
```

```
Out[133]=
{0.016, {{{x → 29.9999}, 0.0000145192}, {{x → 30.}, 1.61324 × 10^-6},
  {{x → 30.}, 1.79248 × 10^-7}, {{x → 30.}, 1.99165 × 10^-8}, {{x → 30.}, 2.21294 × 10^-9},
  {{x → 30.}, 2.21294 × 10^-9}, {{x → 30.}, 1.99165 × 10^-8}, {{x → 30.}, 1.79248 × 10^-7},
  {{x → 30.}, 1.61323 × 10^-6}, {{x → 30.0001}, 0.000014519}, {{x → 59.9999}, 0.000014519},
  {{x → 60.}, 1.61323 × 10^-6}, {{x → 60.}, 1.79248 × 10^-7}, {{x → 60.}, 1.99165 × 10^-8},
  {{x → 60.}, 2.21294 × 10^-9}, {{x → 60.}, 2.21294 × 10^-9}, {{x → 60.}, 1.99165 × 10^-8},
  {{x → 60.}, 1.79248 × 10^-7}, {{x → 60.}, 1.61324 × 10^-6}, {{x → 60.0001}, 0.0000145192}}}
```

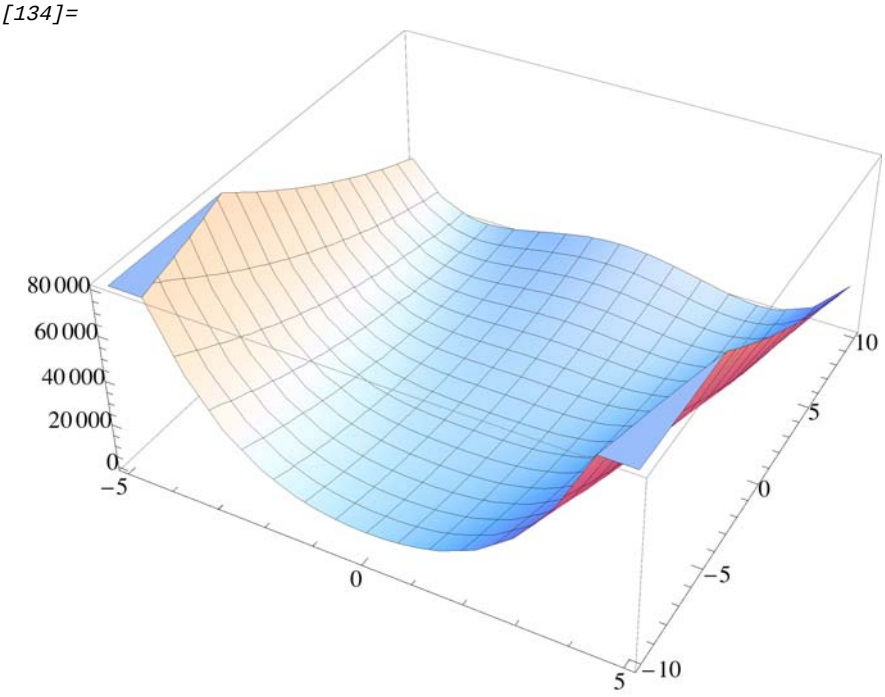
Note that for small problems like this, the solution is very fast. While some of the values look like duplicates, note that they must not be because otherwise the values of the function would be identical. This results from significant digits that *Mathematica* does not show on output. Traditional methods would find one or the other of these solutions, depending on the initial guess given to the algorithm. In this case, the maximum and minimum coordinates of the indifference region overstate the size of the region, which is really just two widely separated clusters of points.

IV.10.B The Rosenbrock Function

An interesting problem is the Rosenbrock function (Ingber and Rosen, 1992) a classic test of convergence of optimization algorithms. The plot of this function is:

```
In[134]:=
Plot3D[100. * (x^2 - y)^2 + (1 - x)^2, {x, -5, 5}, {y, -10, 10}]
```

```
Out[134]=
```



This function becomes quite shallow as the optimal region is approached, and some algorithms experience convergence difficulties with it. We can see that GlobalMinima does not experience difficulties with it and finds the global minimum z

= 0 at {1, 1}, even with an initial grid = 5 and contraction = $0.\overline{2}$, so long as the tolerance is small (< $10^{-\overline{7}}$). The execution time is faster than when using GlobalSearch.

```
In[135]:=
GlobalMinima[100. * (x^2 - y)^2 + (1. - x)^2, ,
  {{x, -5, 5.}, {y, -10., 10.}}, 8, 0.00000001, 0.2, 0.000000000001] // Timing
```

```
Out[135]=
{0.188, {{{x → 0.999997, y → 0.999994}, 0},
  {{x → 0.999997, y → 0.999994}, 0}, {{x → 0.999997, y → 0.999994}, 0},
  {{x → 0.999997, y → 0.999994}, 0}, {{x → 0.999997, y → 0.999995}, 0},
  {{x → 0.999997, y → 0.999994}, 0}, {{x → 0.999997, y → 0.999994}, 0},
  {{x → 0.999997, y → 0.999995}, 0}, {{x → 0.999997, y → 0.999995}, 0},
  {{x → 0.999997, y → 0.999996}, 0}, {{x → 0.999998, y → 0.999995}, 0},
  {{x → 0.999998, y → 0.999996}, 0}, {{x → 0.999998, y → 0.999996}, 0},
  {{x → 0.999998, y → 0.999995}, 0}, {{x → 0.999998, y → 0.999996}, 0},
  {{x → 0.999998, y → 0.999996}, 0}, {{x → 0.999999, y → 0.999998}, 0},
  {{x → 0.999999, y → 0.999999}, 0}, {{x → 0.999999, y → 0.999998}, 0},
  {{x → 0.999999, y → 0.999999}, 0}, {{x → 0.999999, y → 0.999999}, 0},
  {{x → 1., y → 0.999998}, 0}, {{x → 1., y → 0.999999}, 0}, {{x → 1., y → 0.999999}, 0},
  {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.}, 0},
  {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.}, 0},
  {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.}, 0}, {{x → 1., y → 1.00001}, 0},
  {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0},
  {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0},
  {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0},
  {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0}, {{x → 1., y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00001}, 0},
  {{x → 1.00001, y → 1.00001}, 0}, {{x → 1.00001, y → 1.00002}, 0}}}
```

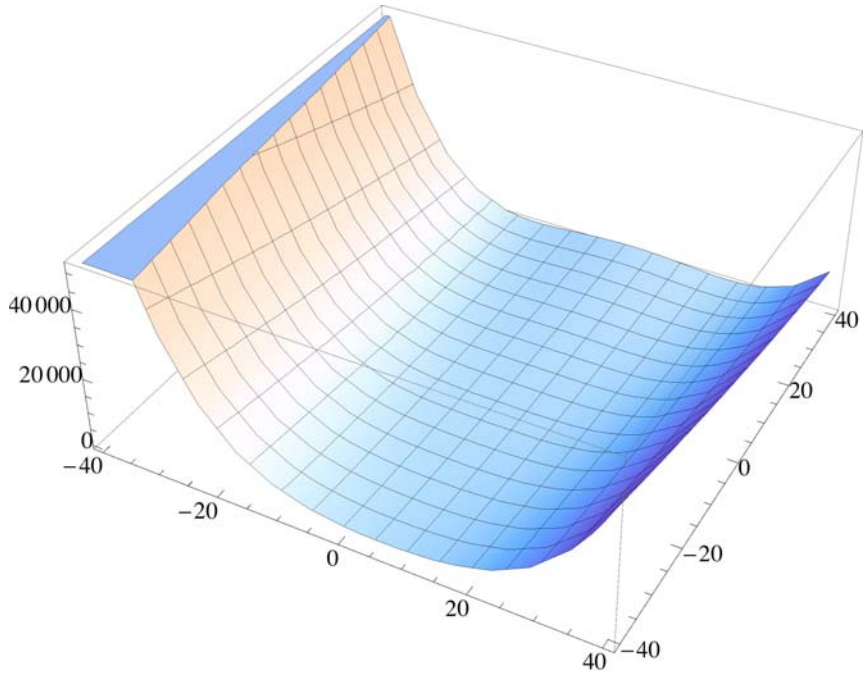
IV.10.C The Branin rcos Function

A related function, the Branin rcos function (Dixon and Szego, 1978), has a more complex surface:

In[136]:=

```
Plot3D[(y - x^2 * 5.1 / (4. * 3.14159^2) + 5. * x / 3.14159 - 6.)^2 +
  10. * (1. - 1. / (8. * 3.14159)) * Cos[x] + 10., {x, -40, 40}, {y, -40, 40}]
```

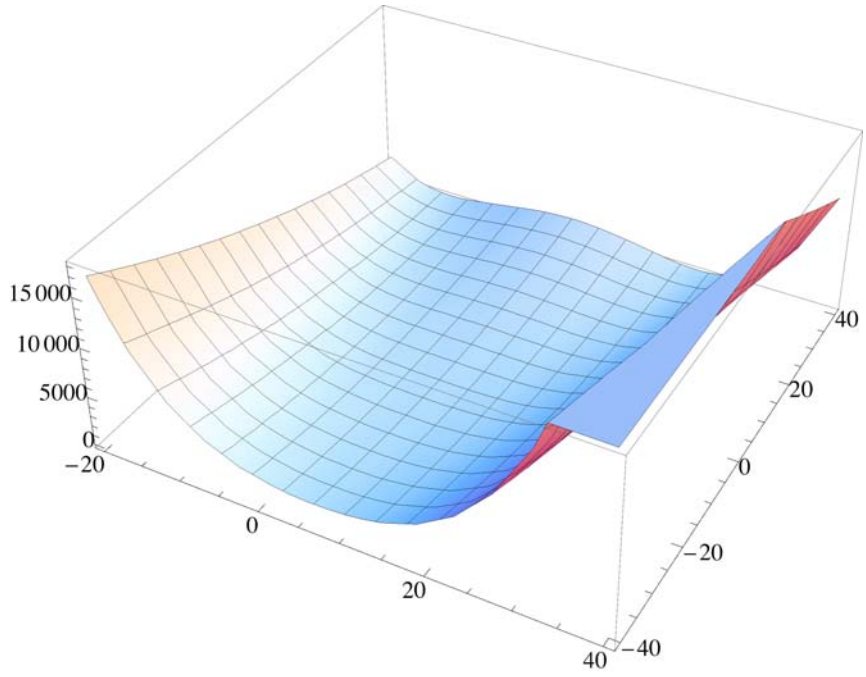
Out[136]=



In the region {-40, 40}, there are six equivalent minima. With traditional algorithms, it would be very difficult to know what initial guesses to use. The number of possible minima is certainly not clear from inspection of the equation or the plot. Zooming in a little on the flat region, we can see a horseshoe-shaped valley:

```
In[137]:=
Plot3D[(y - x^2 * 5.1 / (4. * 3.14159^2) + 5. * x / 3.14159 - 6.)^2 +
  10. * (1. - 1. / (8. * 3.14159)) * Cos[x] + 10., {x, -20, 40}, {y, -40, 40}]
```

Out[137]=



However, we still do not have much information about the solutions. GlobalMinima provides a solution:

```
In[138]:=
ans = GlobalMinima[(y - x^2 * 5.1 / (4. * 3.14159^2) + 5. * x / 3.14159 - 6.)^2 +
   10. * (1. - 1. / (8. * 3.14159)) * Cos[x] + 10., ,
  {{x, -40, 40}, {y, -40, 40}}, 11, 0.001, 0.3, 0.01, ShowProgress -> True]
```

Here, the semicolon was used to suppress output of the list of points, which are instead stored in "ans". From this example, we can see that GlobalMinima finds all six optima with the given parameters and that the indifference parameter defines regions around these optima.

IV.10.D The Csendes Function

A very difficult problem is the Csendes function (Csendes, 1985; Courrieu, 1997):

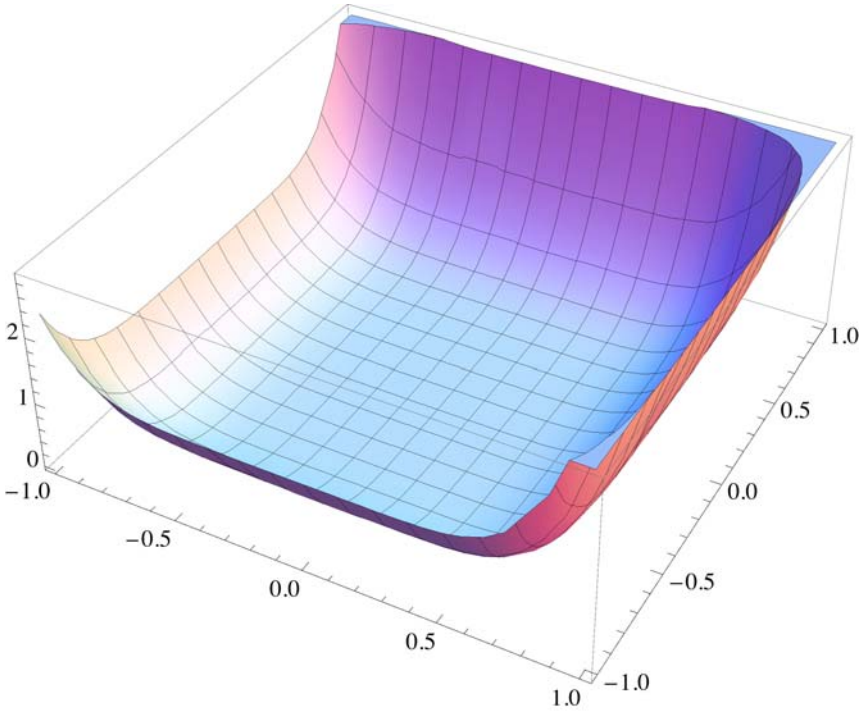
$$C_n(S) = \sum_{i=1}^{n} x_i^6 \left(2 + \sin\frac{1}{x_i}\right) \qquad -1 < x_i < 1 \tag{23}$$

This function has a unique global minimum (0) at x = 0, an oscillation frequency that approaches infinity as one approaches the origin, and a countable infinity of local minima, making local searches impractical. For a 2D version, we can plot the function:

In[139]:=

```
Plot3D[x^6 * (2.0 + Sin[1.0 / x]) + y^6 * (2.0 + Sin[1.0 / y]), {x, -1., 1.}, {y, -1., 1.}]
```

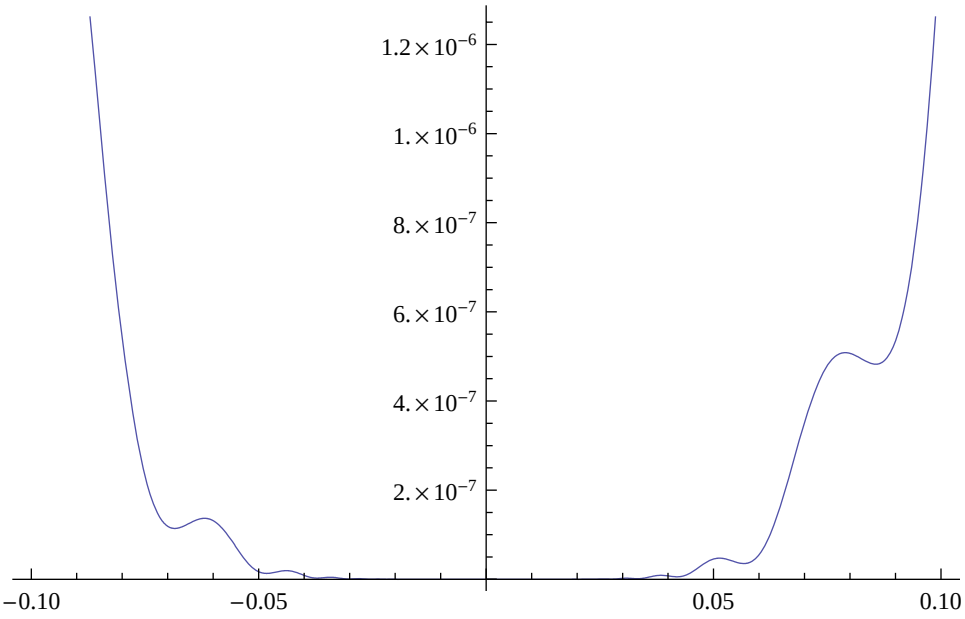
Out[139]=



At this scale, the plot appears smooth, but in cross section and closer to the origin it gets increasingly wavy as we approach zero:

```
In[140]:=
     Plot [x^6 * (2. + Sin[1. / x]), {x, - .1, .1}]
```

Out[140]=



Although the waviness increases close to zero, the absolute magnitude of the oscillations goes to zero as we approach the origin. We can try to solve this problem as follows, with grid = 10 and tolerance = 0.0000001.

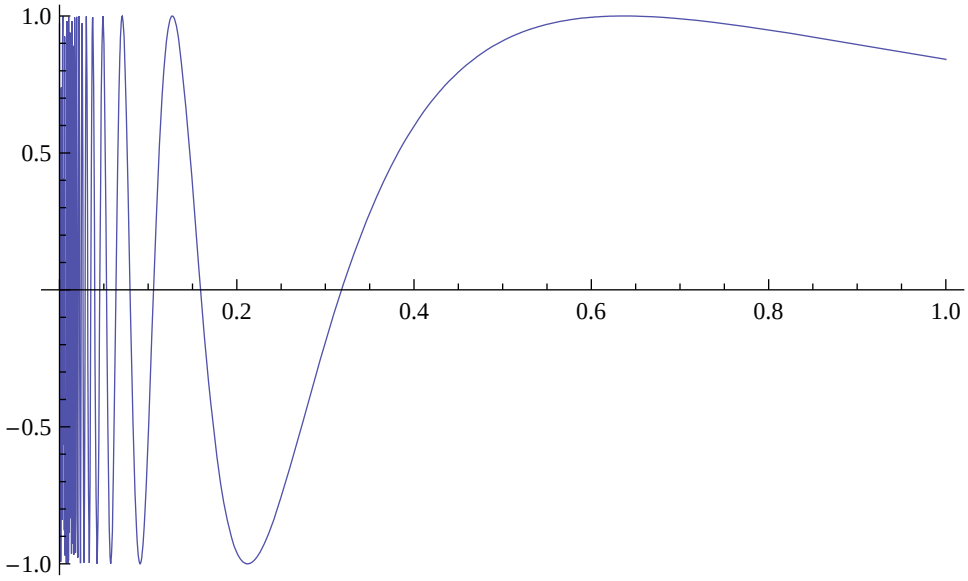
```
GlobalMinima[
 x^6 * (2.0 + Sin[1.0 / x]) + y^6 * (2.0 + Sin[1.0 / y]), ,
 {{x, -1., 1.}, {y, -1., 1.}}, 10, 0.00001, 0.2, 0.0000001]
```
(24)

The solution to this is not shown here because many pages of printout result from the identification of the indifference region. It was found that the best solution is 10^{-15} at {0.0037, 0.0037}, but that the entire region within the box ±0.056 in both dimensions is less than 0.00001 (based on the indifference zone criterion) and that the region within the box -0.167 to 0.21 in both dimensions is less than 0.0001. Zooming in around the best solution and restarting with a new grid does yield some improvement (z = 10^{-30} at {9e-06, 9e-06}), but we have to consider whether improvement from 10^{-15} could possibly have significance.

We can also turn the problem around and ask how bad the local solution can be in the optimal neighborhood. In the region within ±0.056, we find that the worst solution found is around 0.0000001. This enables us to put bounds on the set of points in the optimal region. If these bounds are wide, then very large Z values might be adjacent to very small ones, in which case it is not advisable to view the region full of optimal solutions as truly optimal, because the region is also full of bad solutions. From a purely mathematical viewpoint, this is irrelevant; however, from a practical viewpoint, we should not try to implement an optimal policy or design if very bad solutions are very close to it, because our ability to implement a solution is always imperfect. An obvious example of this difficulty occurs with sin(1/x), whose plot looks like

```
In[141]:=
Plot[Sin[1/x], {x, 0, 1}]
```

```
Out[141]=
```



GlobalMinima with 20 initial grid points yields z = -0.999 at points {0.0033,0.0335,0.0424,0.058,0.09}. This is not a complete set, but it does show that many equivalent solutions exist. Solving for the maxima on this same interval, we find that z = -0.999 at {0.022,0.025,0.03,0.037,0.049,0.07}. We can see that maxima alternate with and are very close to minima over the region. Unless our implementation is perfect, which is quite unlikely, we cannot in the real world implement the optimal solution for such a problem without great risk of actually implementing a worst possible solution. Thus, when multiple optima exist, a combined minimization-maximization is useful to define the feasibility of the solutions.

IV.10.E Wavy Functions, the W Function

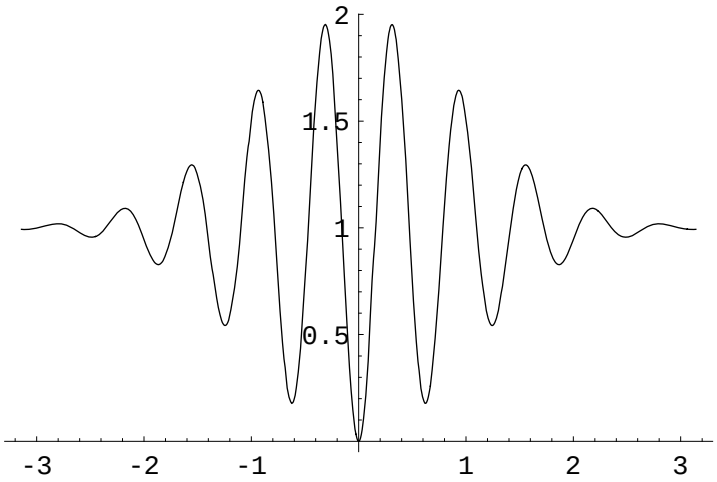
Another "difficult" function is the W function (Courrieu,1997):

$$W_{n,k}(x) = \frac{1}{n}\sum_{i=1}^{n1} -\cos(kx_i)\exp\left(-x_i^2/2\right) \qquad (25)$$

for $-\pi<x<\pi$. W functions have their unique global minimum (0) at x = 0. The number of local minima in the search domain is kn (for k odd) or (k + 1)n (for k even). Here we used only k = 10, which gave 121 local minima for n = 2 and more than 2.59×10^{10} local minima for n = 10. The function oscillates between two hulls of constant mean (= 1) whose distance from each other is maximal in the neighborhood of the solution.

A one-dimensional slice across the W function shows why this is a difficult function for optimization algorithms.

```
In[16]:= Plot[1 - Cos[10. * x] Exp[-x^2 / 2], {x, - 3.14159, 3.14159}];
```



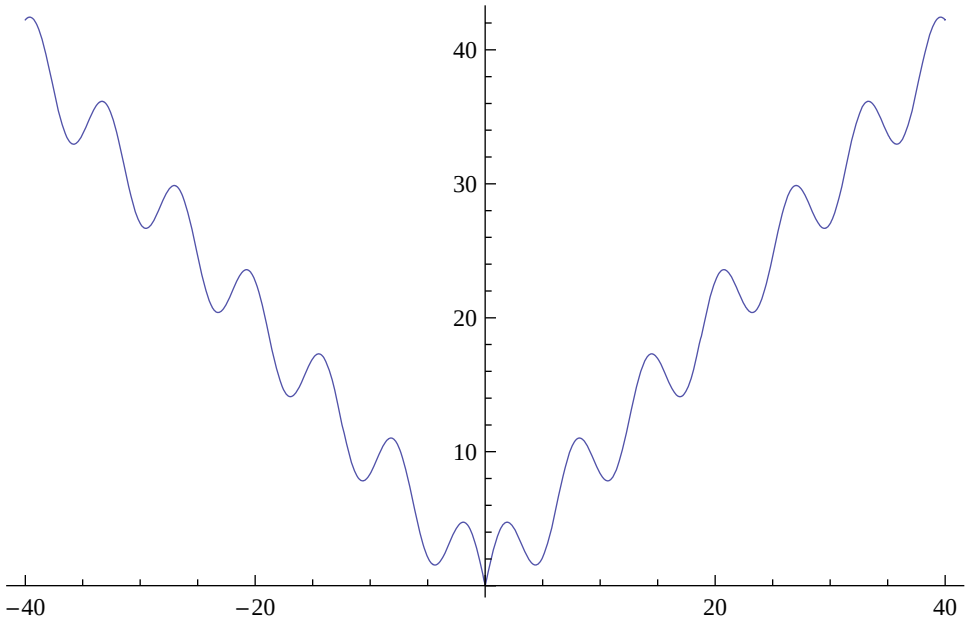
The solution of this function with the AGR algorithm depends on having sufficient grid density so that an initial point falls into the inner oscillation that goes to zero. With an initial grid density of 40 points for a 2D problem, GlobalMinima finds the solution to arbitrary accuracy, depending on Tolerance. Again for such a problem in real-world applications, it is important to know whether bad solutions are close to good ones. Here we can zoom in on the best solution and, over the interval {-1, 1} on each dimension, find min(z) = -W. When we do this, GlobalMinima finds the worst solution z = 1.95 at {-0.311, -0.311} and three other symmetrically located points, all very close to the global minimum (as can also be seen in the one dimensional figure above). This is a potentially unstable situation. We must then evaluate whether we can implement a solution to this problem in the real world with sufficient accuracy to ensure that neither noise nor error will lead to the adverse solutions rather than the desired optimal one.

IV.10.F More Wavy Functions

Some further examples of wavy functions are instructive. First, we consider the function Sin_1:

$$z = |x + 3\,\mathrm{Sin}(x)| \tag{26}$$

```
In[142]:=
     Plot[Abs[x + 3 Sin[x]], {x, -40, 40}]
Out[142]=
```



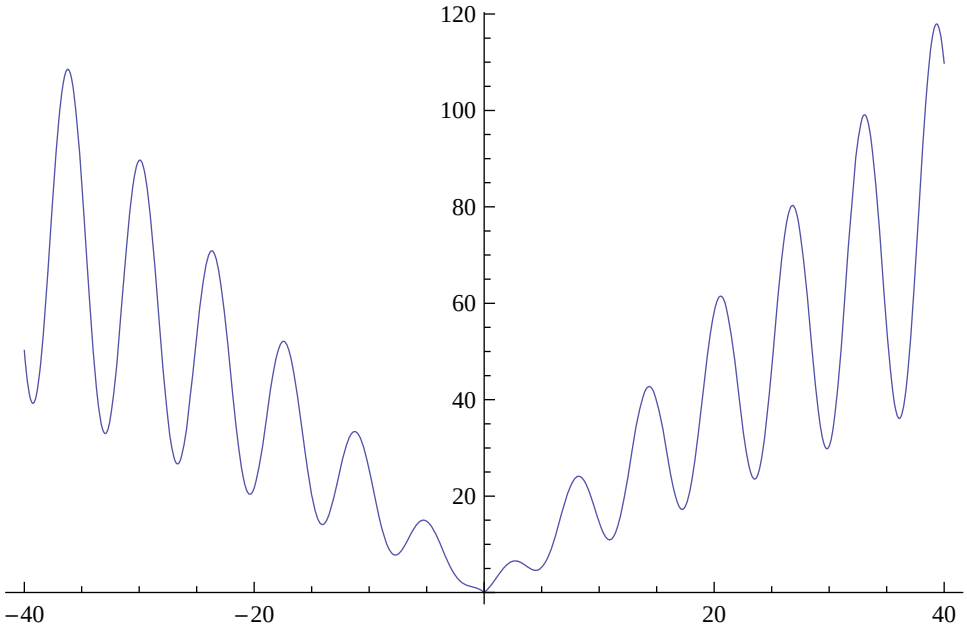
When solved by solvers such as IMSL or the *Mathematic*a FindMinimum function, this function yields local minima unless the initial guess is near the origin. GlobalMinima solved this to arbitrary accuracy (on {-40, 40}) with 26 initial grid points. An accuracy of 10^{-12} was achieved with 150 function calls.

The related waxy function Sin_2:

$$\mathbf{z = |\, 2x + xSinx\,|} \tag{27}$$

```
In[143]:=
     Plot [Abs [2 x + x * Sin [x]], {x, -40, 40}]
```

Out[143]=



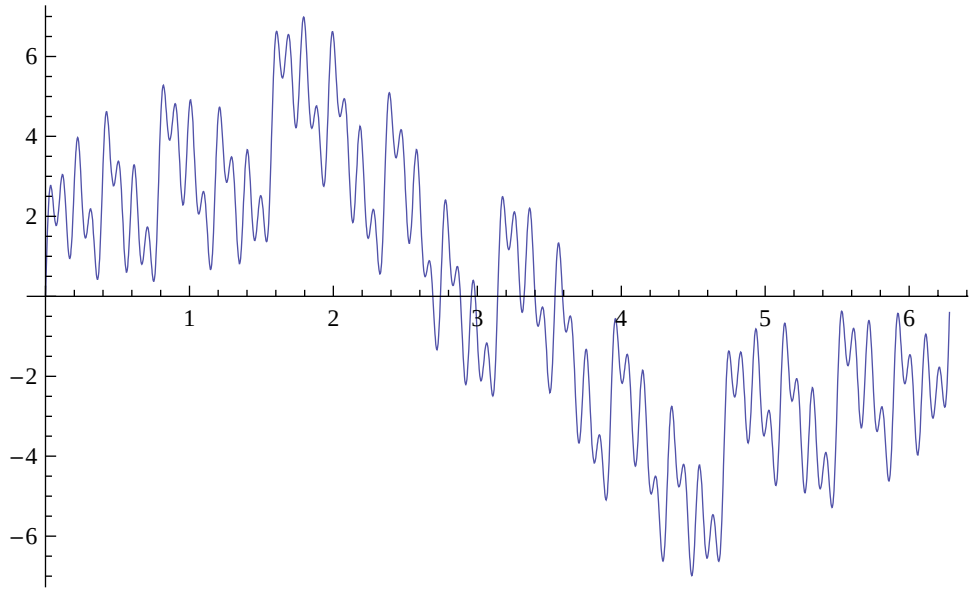
with 12 initial grid points GlobalMinima attained an accuracy of 10^{-12} with 152 function calls.

IV.10.G Wavy Fractals

As an example of a very wavy function, a fractal (nondifferentiable) figure on the line was constructed:

```
In[144]:=
Plot[4 Sin[x] + Sin[4 x] + Sin[8 x] + Sin[16 x] + Sin[32 x] + Sin[64 x], {x, 0, 6.28}]
```

```
Out[144]=
```



(Here this figure was iterated to just six levels and evaluated between 0 and 2π.) With just 10 initial grid points, GlobalMinima found the solution z = -6.98759 at x = 4.48989. Even with as few as 5 initial grid points, the optimal solution was found.

```
In[145]:=
GlobalMinima[4 Sin[x] + Sin[4 x] + Sin[8 x] + Sin[16 x] + Sin[32 x] + Sin[64 x], ,
  {{x, 0., 6.28}}, 10, 0.00001, 0.2, 0.0000001] // Timing
```

```
Out[145]=
{0.016, {{{x → 4.48984}, -6.98759}, {{x → 4.48984}, -6.98759}}}
```

Note that this is much faster than using MultiStartMin or GlobalSearch and also more reliable on this very wavy function. GlobalMinima will be faster for small problems (<3 variables without flat regions).

V NONLINEAR REGRESSION: THE NLRegression FUNCTION

V.1 Introduction

NLRegression is a function that performs nonlinear regression using nonlinear least-squares. It is able to incorporate constraints which may represent physical limits on the parameters being estimated, such as that a growth rate can not be negative.

The optimizer underlying NLRegression is GlobalSearch. All options used by GlobalSearch are accepted by NLRegression and passed down to it except StartsList. See the GlobalSearch section for its options. The only other option accepted is Weights, which is the weight to be assigned to each data point. Weighting might be employed when some of the data have a greater reliability than others. The format for the function call is

NLRegression[data,expression,independent_variables,inequalities,equalities,{{var1name,lowbound,highbound}..},tolerance,options] (28)

The output of the NLRegression procedure is a table of confidence intervals on the parameters, sensitivity plots, the fit parameter values, and the fit statistics. The sensitivity plot can be very useful for detecting parameters that are redundant.

The typical regression problem involves fitting data to a nonlinear model. If the model is nonanalytic (black box), the problem can not be solved by NLRegression, but instead must be solved by setting up the least-squares error function which is then passed in to GlobalSearch. If the function returns values that are below machine accuracy or can return complex results, CompileOption->False should be used. There are four options: FitStatistic, SensitivityPlots, Weights and Norm:

```
In[146]:=
?FitStatistic
```

Fit criterion for NLRegression.
LeastSquare default. ChiSquare also available option.

```
In[147]:=
?Weights
```

Weight assigned to data point i during
regression analysis. Does not affect sum of squares.

```
In[148]:=
?Norm
```

Norm for fitting in NLRegression. L2 is default. L1 also available. ≫

```
In[149]:=
?SensitivityPlots
```

Prints sensitivity plots if True.

```
In[150]:=
?UserResiduals
```

Regression option for passing a user function for residuals.

Let's consider a simple regression problem.

```
In[152]:=
p = Table[i, {i, 0, 20}]
```

```
Out[152]=
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

```
In[153]:=
ClearAll[a, b, x]
```

```
In[154]:=
eqn = a + b * x^2
```

```
Out[154]=
a + b x^2
```

Where a and b are parameters, and x is the independent variable.

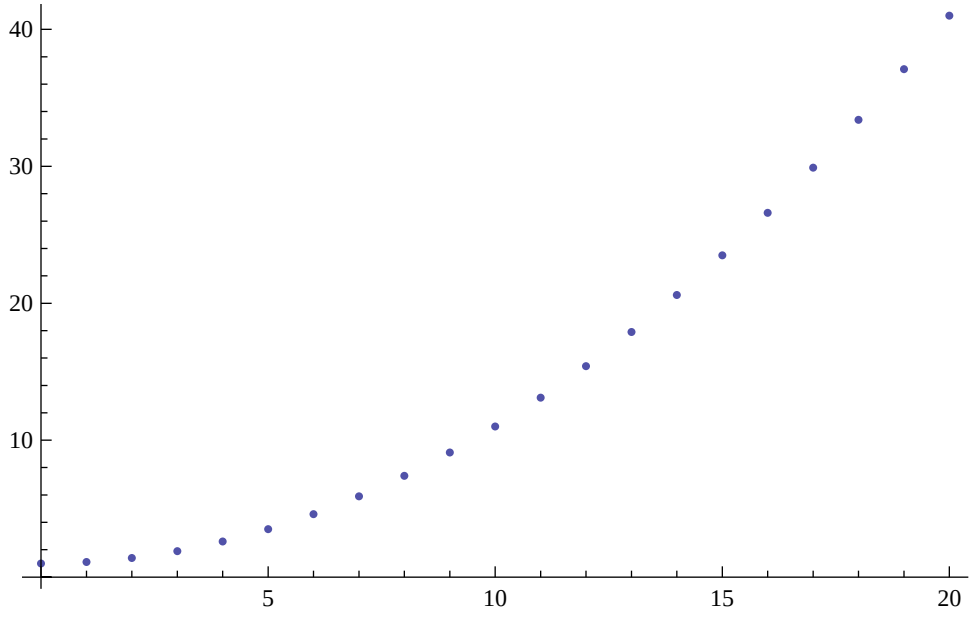
```
In[155]:=
data = {}; Do[AppendTo[data, {x, eqn /. {a → 1, b → .1}}], {x, 0, 20}]
```

```
In[156]:=
data
```

```
Out[156]=
{{0, 1}, {1, 1.1}, {2, 1.4}, {3, 1.9}, {4, 2.6}, {5, 3.5}, {6, 4.6}, {7, 5.9},
 {8, 7.4}, {9, 9.1}, {10, 11.}, {11, 13.1}, {12, 15.4}, {13, 17.9}, {14, 20.6},
 {15, 23.5}, {16, 26.6}, {17, 29.9}, {18, 33.4}, {19, 37.1}, {20, 41.}}
```

```
In[157]:=
ListPlot[data]
```

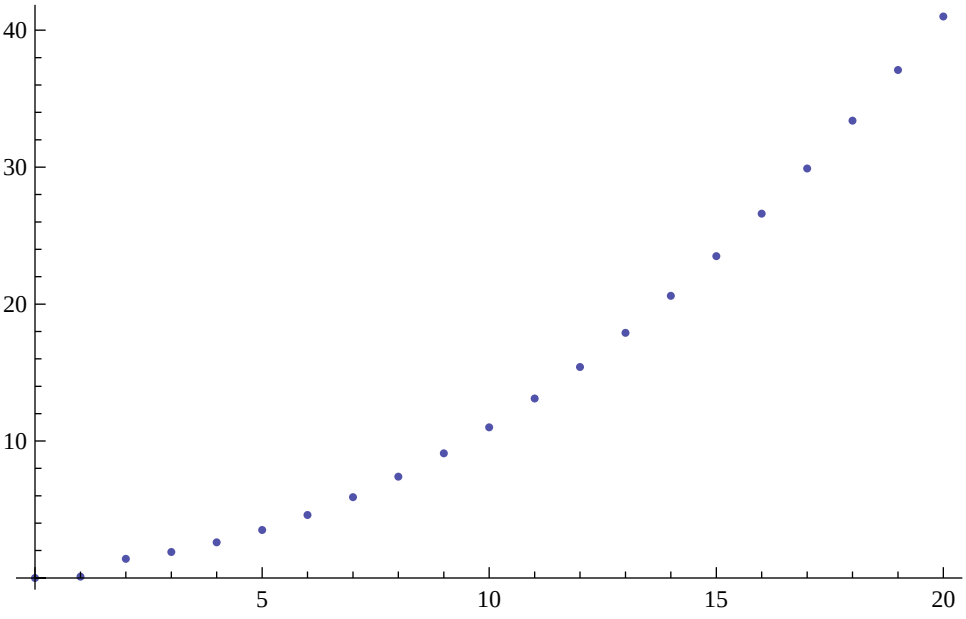
```
Out[157]=
```



```
In[158]:=
r = NLRegression[data, eqn, {x}, {}, {}, {{a, 0, 3}, {b, 0, 1}},
   .0000001, SensitivityPlots → False, StartsList → {{2, .5}}] // Timing

Out[158]=
{2.547, {DegreesOfFreedom → 19, TotalSS → 3304.33, ParameterEstimate → {a → 1., b → 0.1},
  ResidualSS → 1.13652 × 10^-14, RSquare → 1., ConfidenceIntervals →
   {a → Interval[{0.997817, 1.00218}], b → Interval[{0.0999882, 0.100012}]}}}
```

We recover exactly our original parameters. It is possible that the best fit to data violates some constraint such as physical feasibility. In this case, it is necessary to constrain the optimization problem. We introduce some measurement error into the data.

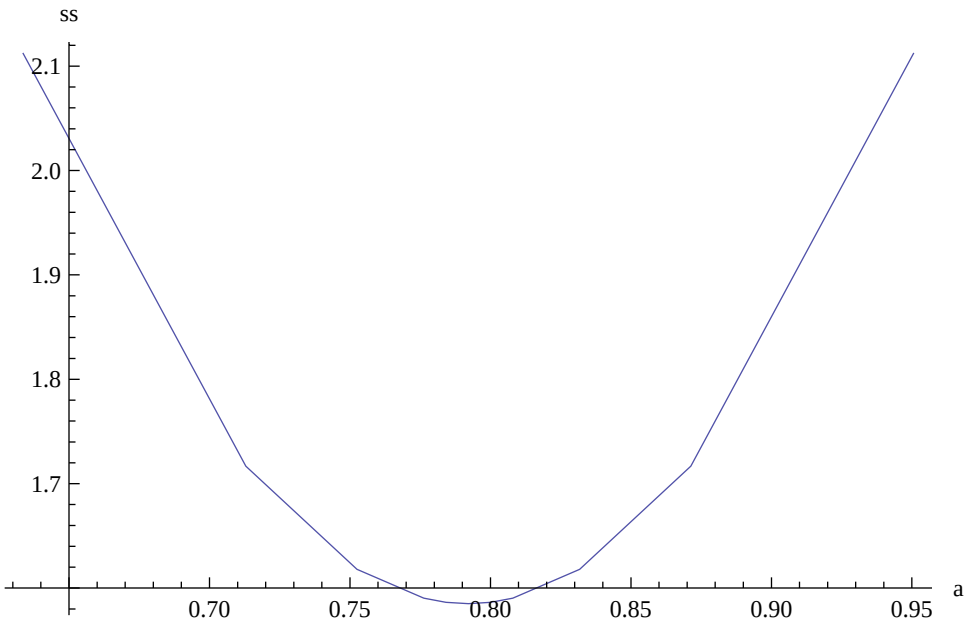
```
In[159]:=
data[[1, 2]] = data[[1, 2]] - 1

Out[159]=
0

In[160]:=
data[[2, 2]] = data[[2, 2]] - 1

Out[160]=
0.1

In[161]:=
data

Out[161]=
{{0, 0}, {1, 0.1}, {2, 1.4}, {3, 1.9}, {4, 2.6}, {5, 3.5}, {6, 4.6}, {7, 5.9},
 {8, 7.4}, {9, 9.1}, {10, 11.}, {11, 13.1}, {12, 15.4}, {13, 17.9}, {14, 20.6},
 {15, 23.5}, {16, 26.6}, {17, 29.9}, {18, 33.4}, {19, 37.1}, {20, 41.}}

In[162]:=
ListPlot[data]

Out[162]=
```



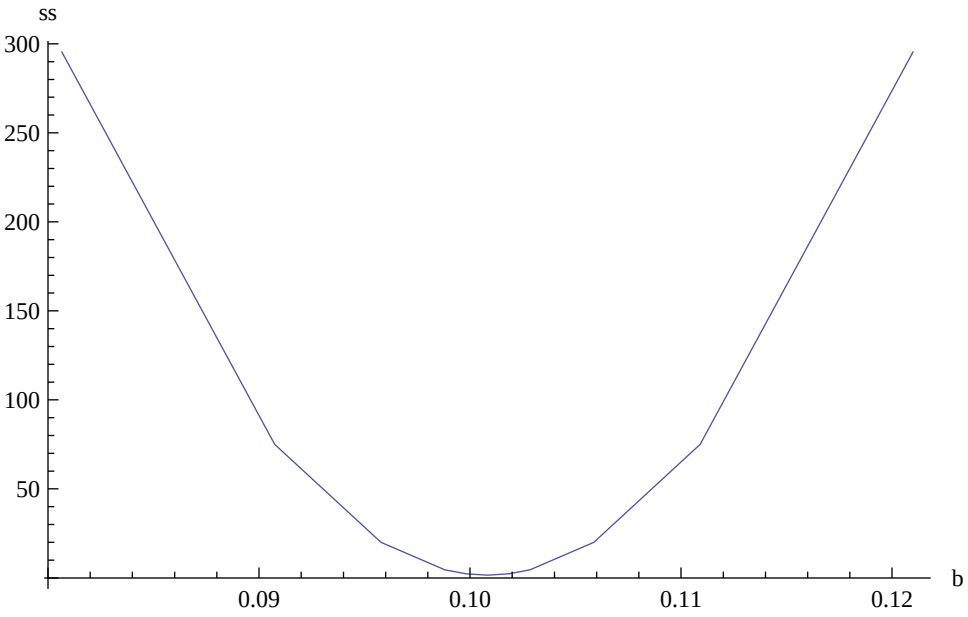
In[163]:=

```
r = NLRegression[data, eqn, {x}, {}, {}, {{a, 0, 3}, {b, 0, 1}}, .0000001] // Timing
```

Sensitivity plot variable a



Sensitivity plot variable b



Out[163]=

```
{0.703, {DegreesOfFreedom → 19,
  TotalSS → 3360.6, ParameterEstimate → {a → 0.792125, b → 0.100824},
  ResidualSS → 1.58507, RSquare → 0.999528, ConfidenceIntervals →
   {a → Interval[{0.505452, 1.0788}], b → Interval[{0.0992788, 0.10237}]}}}
```

We see above that the error added to the data causes "a" to fall down to .792. If we know that parameter a can not fall

below .9 for some reason, we may add this constraint to the problem. a>.9 is converted to standard form as {-a+.9}. We add this to the function call.

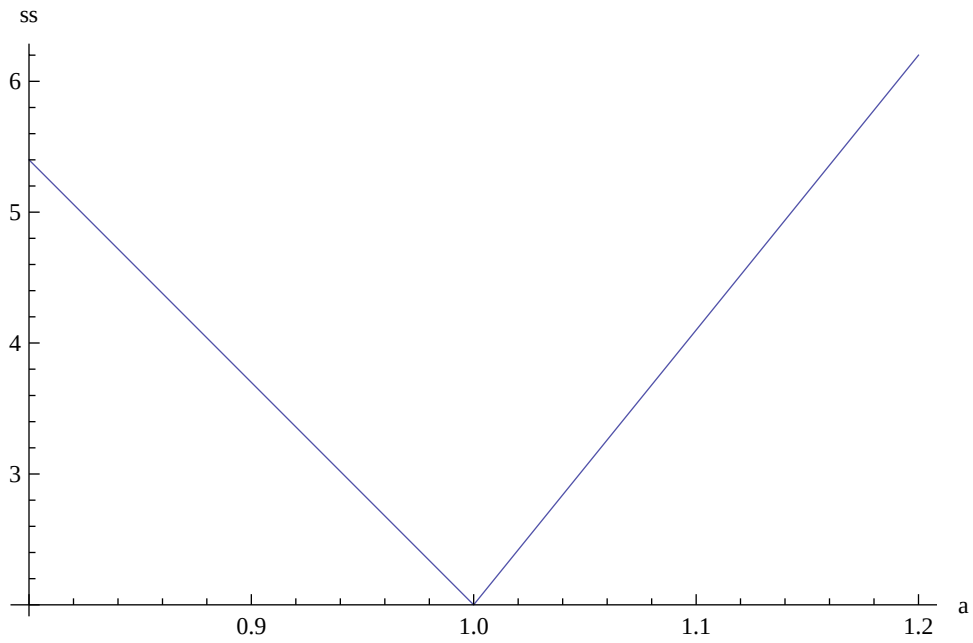
```
In[164]:=
     r = NLRegression[data, eqn, {x}, {-a + .9}, {}, {{a, 0, 3}, {b, 0, 1}}, .0000001] // Timing

         Warning: C.I. and sens. plots not defined with constraints

Out[164]=
     {0.062, {DegreesOfFreedom → 19, TotalSS → 3360.6,
       ParameterEstimate → {a → 0.9, b → 0.100396}, ResidualSS → 1.69681, RSquare → 0.999495}}
```

The addition of the constraint satisfies the restrictions on the problem. Note that with constraints the confidence intervals and sensitivity plots are not defined and can not be printed. Another way to approach this problem is to use the L1 norm, which reduces the influence of outliers. We try this next, without constraints.

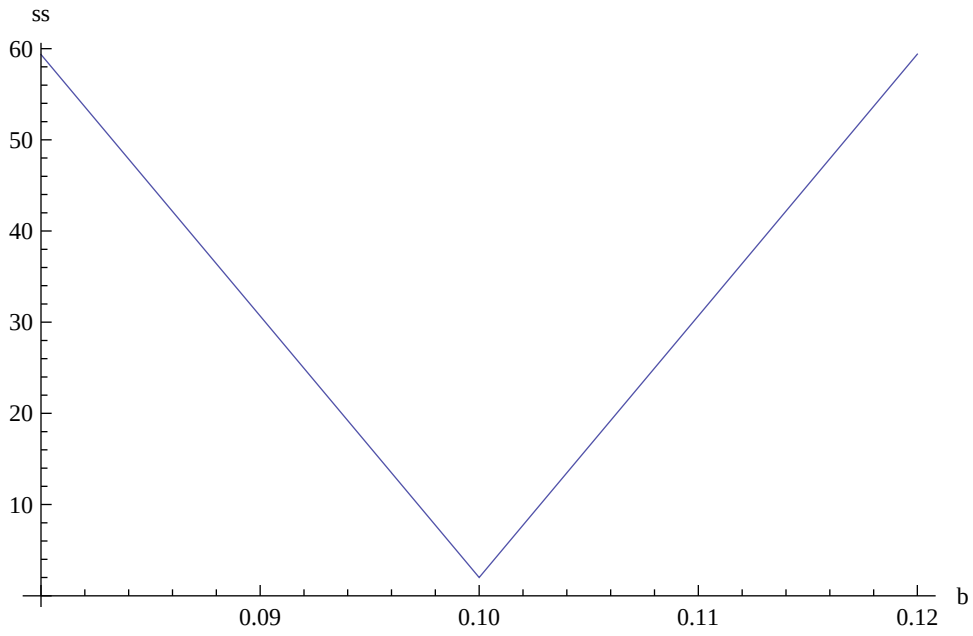
In[165]:=

```
r =
 NLRegression[data, eqn, {x}, {}, {}, {{a, 0, 3}, {b, 0, 1}}, .0000001, Norm → L1] // Timing
```

Sensitivity plot variable a



Sensitivity plot variable b



Out[165]=

```
{1.094, {DegreesOfFreedom → 19, TotalSS → 3360.6, ParameterEstimate → {a → 1., b → 0.1},
  ResidualSS → 2., RSquare → 0.999405, ConfidenceIntervals →
   {a → Interval[{1.22315, 1.22317}], b → Interval[{0.0983608, 0.101636}]}}}
```

We note two things immediately. First, the sensitivity plots are now linear rather than quadratic, because of the different norm. Second, the L1 norm recovers the true parameters in spite of the error in the data and without the use of constraints.

V.2 Utilizing Chi-Square fit Criteria

It is sometimes useful to use a chi-square fit statistic:

In[178]:=
```
FitStatistic → ChiSquare
```

Out[178]=
```
FitStatistic → ChiSquare
```

V.3 Multiple Independent Variables in Regression Problems

In the next example,the fitting of a function to data is illustrated when there are multiple independent variables. A common example of such a problem is when several experiments are done under different conditions and all data from all experiments are to be used to estimate certain parameters.

In[179]:=
```
p = Table[i, {i, 1, 20}]
```

Out[179]=
```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

In[180]:=
```
ClearAll[x, a, b, c, g]
```

In[181]:=
```
y1 = d (a * x^g + b) + c * (2 * b + a * x^g)
```

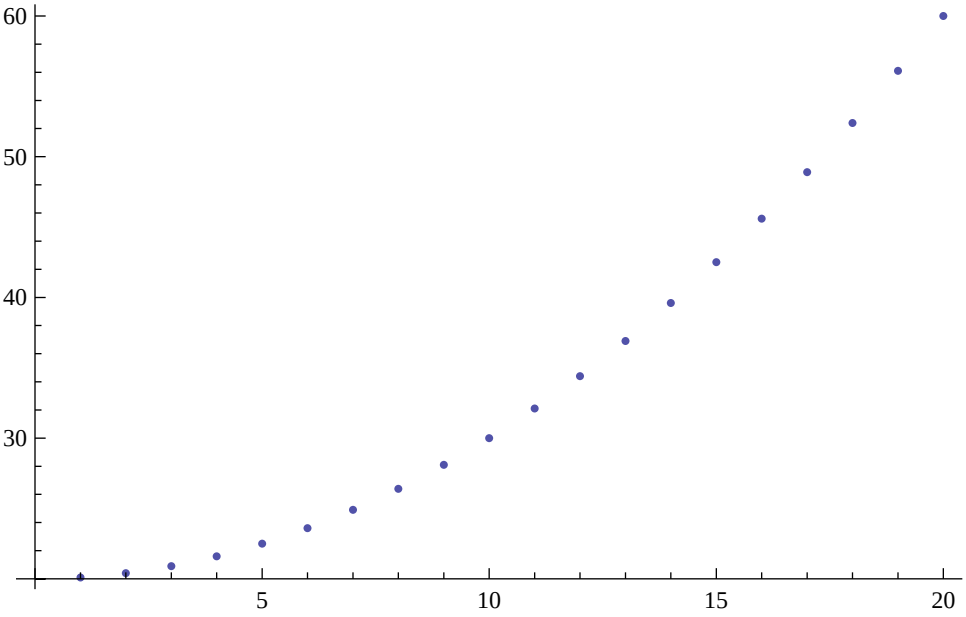
Out[181]=
```
d (b + a x^g) + c (2 b + a x^g)
```

In[182]:=
```
c = 1; d = 0;
dat1 = {}; curve1 = {}; curve2 = {}; Do[k = y1 /. {x -> p[[i]], a → .1, b → 10., g → 2};
 AppendTo[dat1, {c, d, p[[i]], k}]; AppendTo[curve1, {p[[i]], k}];, {i, 1, 20}];
c = 0; d = 1;
Do[k = y1 /. {x -> p[[i]], a → .1, b → 10., g → 2};
 AppendTo[dat1, {c, d, p[[i]], k}]; AppendTo[curve2, {p[[i]], k}];, {i, 1, 20}]
```

In[186]:=

```
p1 = ListPlot[curve1]
```

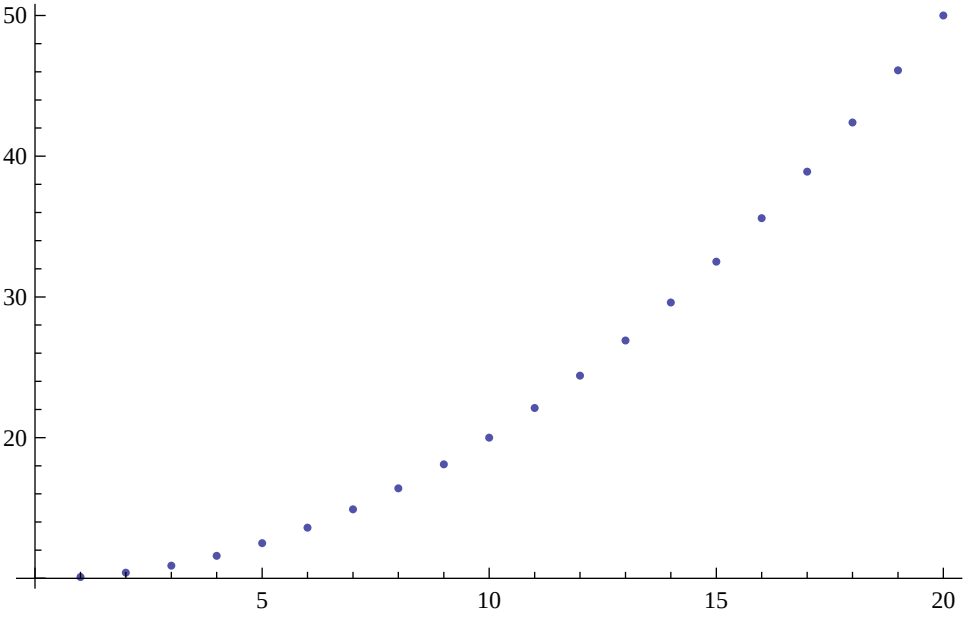
Out[186]=



In[187]:=

```
p2 = ListPlot[curve2]
```

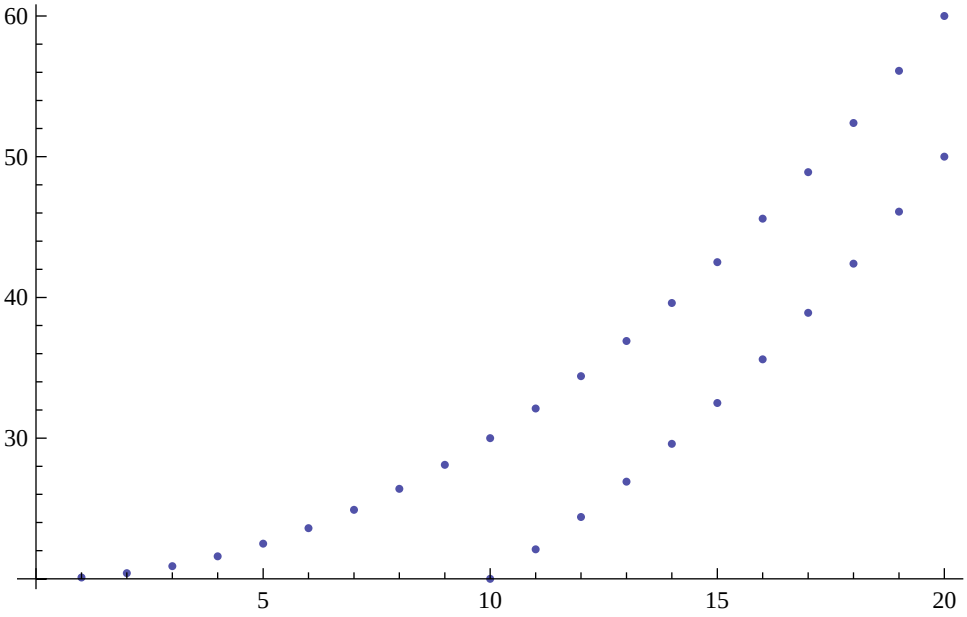
Out[187]=



In[188]:=

```
Show[p1, p2]
```

Out[188]=



In[189]:=

```
ClearAll[c, d, x]
```

In[190]:=

```
indepvars = {c, d, x}
```

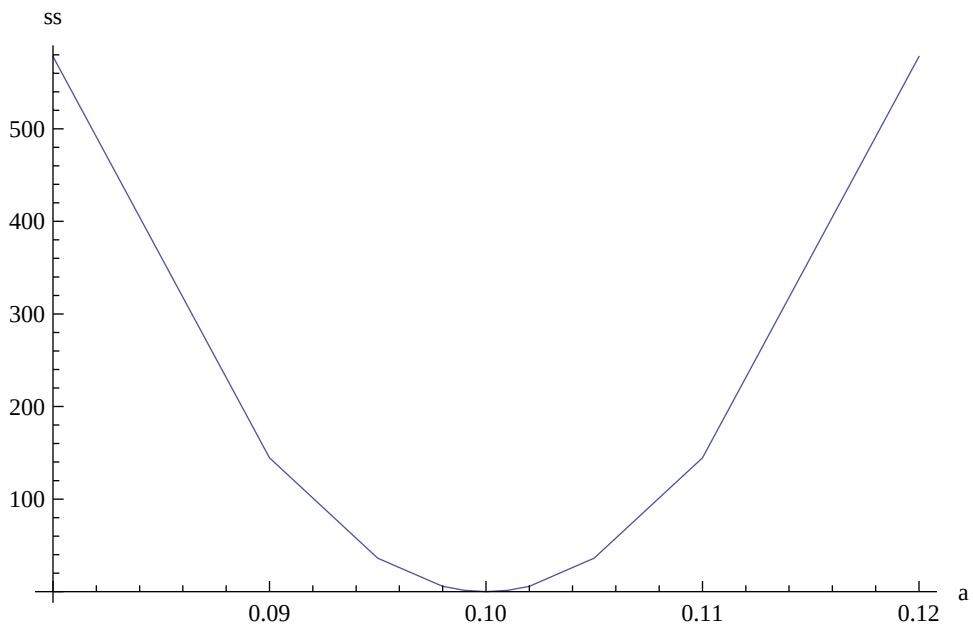
Out[190]=

```
{c, d, x}
```

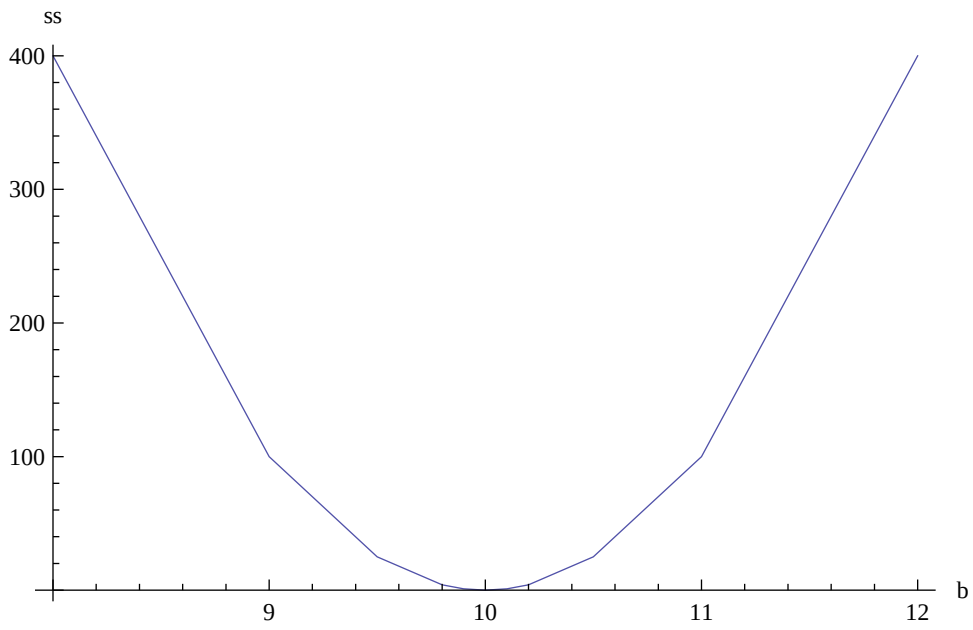
In[191]:=

```
r = NLRegression[dat1, y1, indepvars,
   {}, {}, {{a, 0, 1}, {b, 0, 20}, {g, 1, 3}}, .0000001] // Timing
```

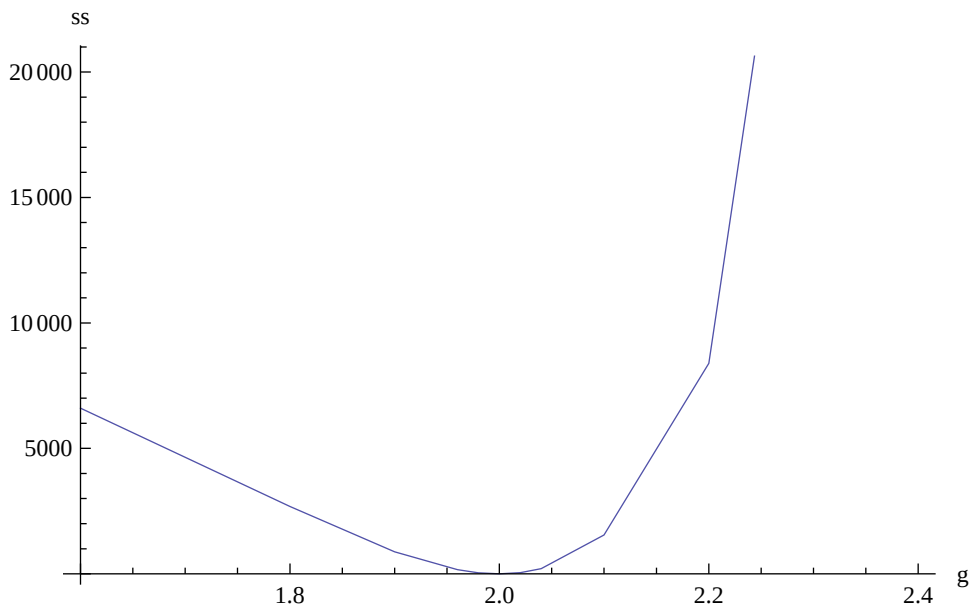
```
   Sensitivity plot variable a
```



Sensitivity plot variable b



Sensitivity plot variable g



Out[191]=

$$\{11.781, \{\text{DegreesOfFreedom} \to 37, \text{TotalSS} \to 7216.42,$$
$$\text{ParameterEstimate} \to \{a \to 0.1, b \to 10., g \to 2.\}, \text{ResidualSS} \to 6.34686 \times 10^{-25},$$
$$\text{RSquare} \to 1., \text{ConfidenceIntervals} \to \{a \to \text{Interval}[\{0.0999917, 0.100008\}],$$
$$b \to \text{Interval}[\{9.999, 10.001\}], g \to \text{Interval}[\{1.99997, 2.00003\}]\}\}\}$$

In[192]:=

```
rr = r[[2]]
```

Out[192]=

$$\{\text{DegreesOfFreedom} \to 37, \text{TotalSS} \to 7216.42,$$
$$\text{ParameterEstimate} \to \{a \to 0.1, b \to 10., g \to 2.\}, \text{ResidualSS} \to 6.34686 \times 10^{-25},$$
$$\text{RSquare} \to 1., \text{ConfidenceIntervals} \to \{a \to \text{Interval}[\{0.0999917, 0.100008\}],$$
$$b \to \text{Interval}[\{9.999, 10.001\}], g \to \text{Interval}[\{1.99997, 2.00003\}]\}\}$$

In[193]:=

```
rrr = rr[[3]]
```

Out[193]=

$$\text{ParameterEstimate} \to \{a \to 0.1, b \to 10., g \to 2.\}$$

In[194]:=

```
pars = rrr[[2]]
```

Out[194]=

$$\{a \to 0.1, b \to 10., g \to 2.\}$$

In[195]:=

```
y1
```

Out[195]=

$$d\,(b + a\,x^g) + c\,(2\,b + a\,x^g)$$

In[196]:=

```
pars2 = Join[pars, {c → 1, d → 0}]
```

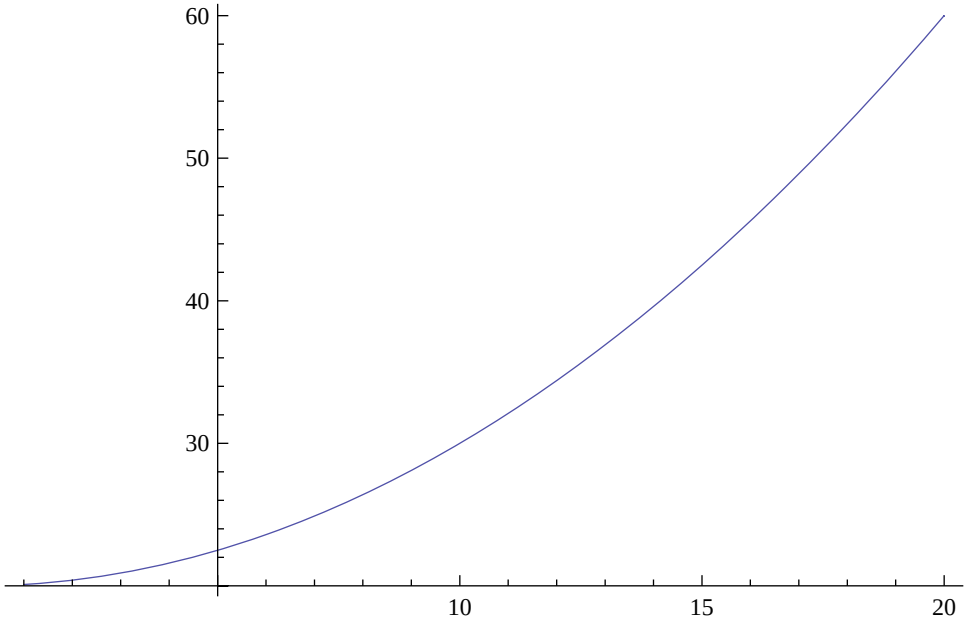
Out[196]=

```
{a → 0.1, b → 10., g → 2., c → 1, d → 0}
```

In[197]:=

```
p3 = Plot[y1 /. pars2, {x, 1, 20}]
```

Out[197]=



In[198]:=

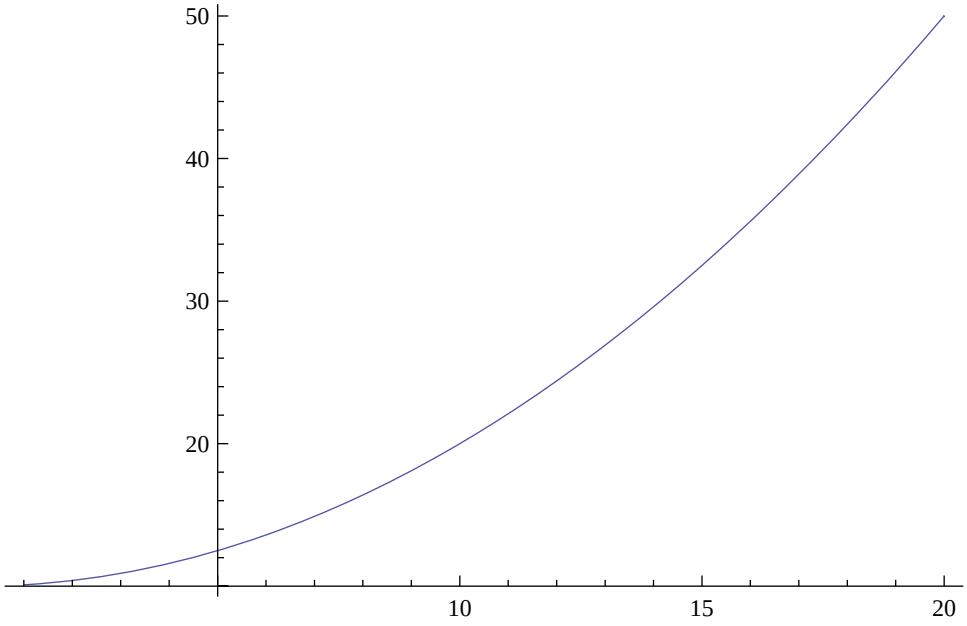
```
pars3 = Join[pars, {c → 0, d → 1}]
```

Out[198]=

```
{a → 0.1, b → 10., g → 2., c → 0, d → 1}
```

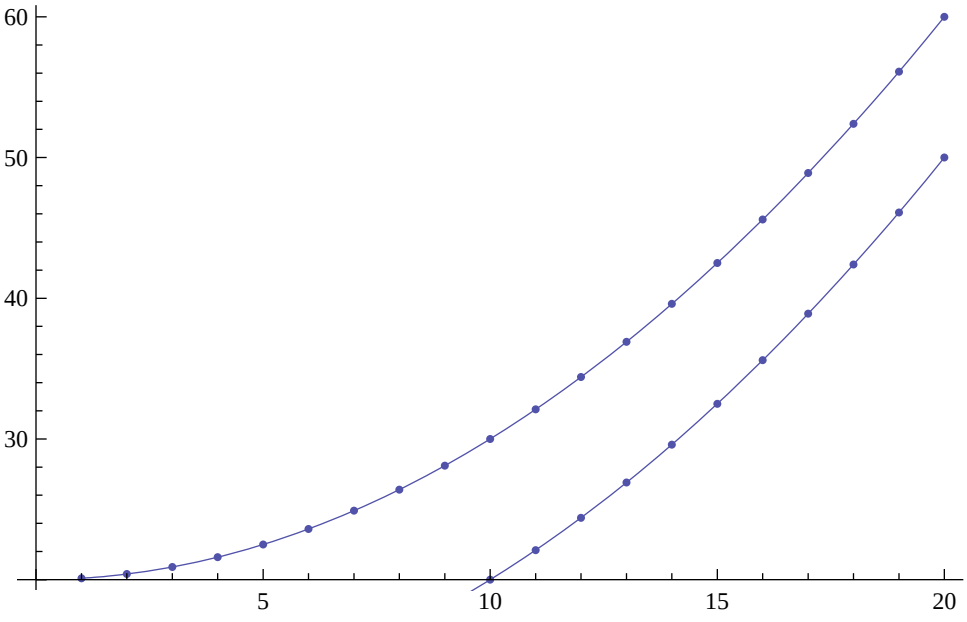
```
In[199]:=
p4 = Plot[y1 /. pars3, {x, 1, 20}]
```

Out[199]=




```
In[200]:=
Show[p1, p2, p3, p4]
```

Out[200]=



We see that we have simultaneously fit both curves. Problems with multiple independent variables can be fit even if they can not be graphed in the manner above.

VI MAXIMUM LIKELIHOOD ESTIMATION: THE

MaxLikelihood FUNCTION

VI.1 Introduction

In this section, the **MaxLikelihood** function is described. This function is an estimation procedure which attempts to maximize the log-likelihood of a function with respect to a data set. It is a useful alternative to nonlinear regression (the NLRegression function). The statistics used are based on Crooke et al. (1999). The function uses GlobalSearch as the minimizing function, which is needed because some functions require constraints to be solved properly. It thus accepts all GlobalSearch options, except CompileOption and SimplifyOption. This is because Compile does not work well with Log[], and because a sum of Log terms will generally not simplify. It is assumed that the function to be solved is algebraic, and black box functions are not allowed. Weighting is not yet implemented. Special functions are pre-programmed for efficiency and ease of use. The pre-programmed functions can be ascertained as follows:

```
In[419]:=
?LikelihoodModels

    Predefined models for MaxLikelihood
      function:NormalModel,PoissonModel,BetaModel,GammaModel,LogNormalModel.
```

The syntax for the function is as follows:

MaxLikelihood[data,expression,independent_variables,constraints,{{var1name,lowbound, highbound}..},tolerance,options] (29)

VI.2 Examples and Built-In Functions

We first illustrate use of the **MaxLikelihood** function for determining the mean μ and standard deviation σ from a random sample that has been generated by the Normal distribution:

$$f(x;\,\mu,\,\sigma) = \frac{e^{-\frac{(x+\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}},\ x \in \mathbb{R}.$$

This probability distribution density is defined as **NormalDistribution**.

For demonstration purposes, we draw a set of pseudo random numbers from the Normal distribution with $\mu = 10$ and $\sigma = 1$.

```
In[201]:=
T = 100;
exact = {μ -> 10, σ -> 1};
data = Table[Random[NormalDistribution[μ, σ] /. exact], {T}];
```

Next, we define the probability density that we would like to fit as a pure function.

```
In[204]:=
Clear[f];
f = PDF[NormalDistribution[μ, σ], #] &;
```

In[206]:=

```
g = f[x]
```

Out[206]=

$$\frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\ \sigma}$$

In[207]:=

```
mle = MaxLikelihood[data, g, {x}, {},
   {{μ, 8, 12}, {σ, .8, 1.5}}, .000001, CompileOption → True] // Timing
```

```
CompiledFunction::cfn : Numerical error encountered
   at instruction 24; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 24; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 24; proceeding with uncompiled evaluation. >>

General::stop : Further output of
   CompiledFunction::cfn will be suppressed during this calculation. >>

   -----Maximum Likelihood with Gradient Variance Computation-----

   Number of Observations: 100

   Log(L): -144.247
```

	Parameter	Standard Error	z-statistic
μ	10.05	0.1075	93.49
σ	1.024	0.06432	15.92

Out[207]=

```
{0.687, {ParameterEstimate → {μ → 10.0456, σ → 1.02381}, LogLikelihood → -144.247,
  vcov → {{0.0115462, -0.00209818}, {-0.00209818, 0.00413659}}}}
```

We see that we have recovered the true parameters pretty well. A large sample size would be needed for a better estimate. We can compare the results using the built-in function NormalModel:

```
In[208]:=
mle = MaxLikelihood[data, NormalModel, {x}, {},
   {{μ, 8, 12}, {σ, .8, 1.5}}, .000001, CompileOption → True] // Timing

CompiledFunction::cfn : Numerical error encountered
   at instruction 17; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 17; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 17; proceeding with uncompiled evaluation. >>

General::stop : Further output of
   CompiledFunction::cfn will be suppressed during this calculation. >>

   -----Maximum Likelihood with Gradient Variance Computation-----

   Number of Observations: 100

   Log(L): -144.247

     | Parameter  Standard Error  z-statistic
   ──┼──────────────────────────────────────
   μ | 10.05      0.1075          93.49
   σ | 1.024      0.06432         15.92
```

```
Out[208]=
{0.078, {ParameterEstimate → {μ → 10.0456, σ → 1.02381}, LogLikelihood → -144.247,
  vcov → {{0.0115462, -0.00209818}, {-0.00209818, 0.00413659}}}}
```

We see that the exact same result is obtained, but the run is about 4 times faster. Note that the order of the parameters, but not their names, is assumed to match the {mean,variance} parameters of the normal model. In this example, since the data is a sample from the distribution, we do not get back exactly the parameters we used to generate the data unless we draw a very large sample data set.

```
In[209]:=
T = 200;
exact = {μ -> 10, σ -> 1};
data = Table[Random[NormalDistribution[μ, σ] /. exact], {T}];
```

```
In[212]:=
mle =
 MaxLikelihood[data, NormalModel, {x}, {}, {{μ, 8, 12}, {σ, .8, 1.5}}, .000001] // Timing

   -----Maximum Likelihood with Gradient Variance Computation-----

   Number of Observations: 200

   Log(L): -296.231

     | Parameter  Standard Error  z-statistic
   ──┼──────────────────────────────────────
   μ | 10.03      0.0759          132.1
   σ | 1.064      0.06104         17.43
```

```
Out[212]=
{0.078, {ParameterEstimate → {μ → 10.0277, σ → 1.06419}, LogLikelihood → -296.231,
  vcov → {{0.00576139, -0.000606893}, {-0.000606893, 0.00372613}}}}
```

We see above that a larger sample size gave a much better estimate of the parameters. Note also that the execution time is nearly unaffected by the size of the data set for the built in model, although time required goes up linearly with data set size

for the standard function.

Next we consider the estimation of the mean λ for a sample data drawn from the Poisson distribution:

$$f(x; \lambda) = \frac{e^{-\lambda}\lambda^x}{x!}, \; x = 0, 1, 2, \ldots .$$

We use the same approach to illustrate the method, generating a sample with a known mean and then estimating the mean from the sample. We draw sample data from a distribution with $\lambda = 10$.

In[213]:=

```
Clear[f];
T = 100;
exact = λ -> 10;
data = Table[Random[PoissonDistribution[λ] /. exact], {T}];
f = PDF[PoissonDistribution[λ], #] &;
```

In[218]:=

```
g = f[x]
```

Out[218]=

$$\frac{e^{-\lambda}\lambda^x}{x!}$$

In[220]:=

```
mle = MaxLikelihood[data, PoissonModel, {x}, {}, {{λ, 6, 9}}, .000001] // Timing
```

```
-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 100

Log(L): -267.396

  | Parameter  Standard Error  z-statistic
λ | 9.56       0.2727          35.05
```

Out[220]=

```
{0.031, {ParameterEstimate → {λ → 9.56}, LogLikelihood → -267.396, vcov → {{0.074386}}}}
```

Once again, the preprogrammed model is faster. Next we evaluate the Beta function.

Our next example is the Beta Distribution

$$f(x; \alpha, \beta) = \frac{\Gamma(\alpha+\beta)\,(1-x)^{q-1}\,x^{p-1}}{\Gamma(\alpha)\,\Gamma(\beta)}, \; 0 \le x \le 1$$

where $\alpha, \beta > -1$ are parameters and Γ denotes the gamma function. As above, we draw sample data from the specified distribution.

```
In[235]:=
Clear[f];
T = 200;
exact = {α -> 2, β -> 5};
data = Table[Random[BetaDistribution[α, β] /. exact], {T}];
f = PDF[BetaDistribution[α, β], #] &;

In[240]:=
g = f[x]

Out[240]=
(1 - x)^(-1+β) x^(-1+α)
---------------------
Beta[α, β]
```

In this case, we must add constraints or the function goes to negative infinity during the solution.

```
In[241]:=
mle = MaxLikelihood[data, g, {x}, {-α + 1, -β}, {{α, 1, 5}, {β, 3, 8}}, .000001] // Timing

-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 200

Log(L): 91.717

  | Parameter  Standard Error  z-statistic
--+-------------------------------------
α | 2.107      0.1899          11.09
β | 4.878      0.4828          10.1

Out[241]=
{1.313, {ParameterEstimate → {α → 2.10727, β → 4.87844},
  LogLikelihood → 91.717, vcov → {{0.0360752, 0.0768617}, {0.0768617, 0.233123}}}}

In[242]:=
mle = MaxLikelihood[data, BetaModel,
  {x}, {-α + 1, -β + 1}, {{α, 1, 5}, {β, 3, 8}}, .000001] // Timing

-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 200

Log(L): 91.717

  | Parameter  Standard Error  z-statistic
--+-------------------------------------
α | 2.107      0.1899          11.09
β | 4.878      0.4828          10.1

Out[242]=
{0.125, {ParameterEstimate → {α → 2.10727, β → 4.87844},
  LogLikelihood → 91.717, vcov → {{0.0360752, 0.0768617}, {0.0768617, 0.233123}}}}
```

Next we illustrate the Gamma distribution. Note that the order of parameters is assumed to match that in the Gamma built in function.

In[243]:=

```
Clear[f];
T = 100;
exact = {α → 1, λ → 1};
data = Table[Random[GammaDistribution[α, λ] /. exact], {T}];
f = PDF[GammaDistribution[α, λ], #] &;
```

In[248]:=

```
g = f[x]
```

Out[248]=

$$\frac{e^{-\frac{x}{\lambda}}\, x^{-1+\alpha}\, \lambda^{-\alpha}}{\text{Gamma}[\alpha]}$$

Note that we must constrain the parameters to be positive to prevent the function running away to -Infinity. In addition, lambda must be constrained away from 0 to prevent underflow during computations, even though the final answer is not near zero.

In[249]:=

```
mle = MaxLikelihood[data, g, {x},
   {-α + .0001, α - 5, -λ + .0001, λ - 8}, {{α, .2, 5}, {λ, 3, 8}}, .000001] // Timing
```

```
-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 100

Log(L): -103.626
```

	Parameter	Standard Error	z-statistic
α	0.8749	0.09857	8.876
λ	1.192	0.1975	6.036

Out[249]=

```
{0.672, {ParameterEstimate → {α → 0.87492, λ → 1.19244}, LogLikelihood → -103.626,
  vcov → {{0.00971663, -0.014305}, {-0.014305, 0.0390238}}}}
```

In[250]:=

```
mle = MaxLikelihood[data, GammaModel, {x},
   {-α + .0001, α - 5, -λ + .0001, λ - 8}, {{α, .2, 5}, {λ, 3, 8}}, .000001] // Timing
```

```
-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 100

Log(L): -103.626
```

	Parameter	Standard Error	z-statistic
α	0.8749	0.09857	8.876
λ	1.192	0.1975	6.036

Out[250]=

```
{0.063, {ParameterEstimate → {α → 0.87492, λ → 1.19244}, LogLikelihood → -103.626,
  vcov → {{0.00971663, -0.014305}, {-0.014305, 0.0390238}}}}
```

In this case we get a huge speedup (10 times faster). Next, we test the LogNormal predefined model.

In[251]:=

```
Clear[f];
T = 100;
exact = {μ -> 20, σ -> 5};
data = Table[Random[LogNormalDistribution[μ, σ] /. exact], {T}];
f = PDF[LogNormalDistribution[μ, σ], #] &;
```

In[256]:=

```
g = f[x]
```

Out[256]=

$$\frac{e^{-\frac{(-\mu+\text{Log}[x])^2}{2\sigma^2}}}{\sqrt{2\pi}\ x\ \sigma}$$

In[257]:=

```
mle = MaxLikelihood[data, g, {x}, {}, {{μ, 8, 12}, {σ, .8, 10}}, .000001] // Timing
```

```
-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 100

Log(L): -2296.65
```

	Parameter	Standard Error	z-statistic
μ	19.92	0.5117	38.93
σ	5.086	0.371	13.71

Out[257]=

```
{0.579, {ParameterEstimate → {μ → 19.9211, σ → 5.08585},
  LogLikelihood → -2296.65, vcov → {{0.261864, -0.0210043}, {-0.0210043, 0.137646}}}}
```

In[258]:=

```
mle = MaxLikelihood[data, LogNormalModel,
   {x}, {}, {{μ, 8, 12}, {σ, .8, 10}}, .000001] // Timing
```

```
-----Maximum Likelihood with Gradient Variance Computation-----

Number of Observations: 100

Log(L): -2296.65
```

	Parameter	Standard Error	z-statistic
μ	19.92	0.5117	38.93
σ	5.086	0.371	13.71

Out[258]=

```
{0.047, {ParameterEstimate → {μ → 19.9211, σ → 5.08585},
  LogLikelihood → -2296.65, vcov → {{0.261864, -0.0210043}, {-0.0210043, 0.137646}}}}
```

VII DISCRETE VARIABLE PROBLEMS: THE InterchangeMethodMin & TabuSearchMin FUNCTIONS

V.1I Introduction

InterchangeMethodMin and TabuSearchMin are functions which maximize a linear or nonlinear function of integer 0-1 variables either with or without constraints. This type of problem comes up often in operations research (e.g., network, scheduling, and allocation applications). Integer 0-1 problems are not generally solvable with an analytic approach such as Linear Programming. The method used here is the Interchange method (Densham and Rushton, 1992; Goldberg and Paz, 1991; Lin and Kernighan, 1973; Teitz and Bart, 1968), which is similar to the author's SWAP algorithm (Loehle, 2000). It has been shown that the Interchange method is guaranteed to get close to the optimum solution, but for complex problems may not find the exact solution. On the other hand, it is much faster than Simulated Annealing (Murray and Church, 1995). The Interchange algorithm begins with a feasible start generated randomly. It then makes changes in the configuration at each iteration until no more progress is possible. Note that while it is possible to input this type of problem into MultiStartMin because it accepts discrete variables, MultiStartMin assumes that only a few discrete variables are involved, and may generate a large set of possible moves and run slowly for large problems. TabuSearchMin operates with a basic Interchange framework, but then adds a tabu list feature. The tabu list is the list of n previous moves that will not be revisited in looking for the next good move. This feature adds efficiency because bad moves are not checked repeatedly.

The functions are defined by

InterchangeMethodMin[expr,ineqs,{varlist},tolerance,options] (30)

TabuSearchMin[expr,ineqs,{varlist},tolerance,options] (31)

where expr is the function to be minimized, which need not be linear, and ineqs is the inequalities, which are optional and are in standard form. There are three options for the program. Starts defines the number of random starting points to test (default 5). Input starting points can be input with the StartsList option. CompileOption can be set to False if the function is not compilable. ShowProgress shows intermediate stages of the computation. Program operation is shown next in the following examples of applications. Note that the range of applications is not exhausted by these examples.

VII.2 Applications

VII.2.A Capital Allocation

In many investment problems, the goal is to allocate a fixed amount of money across a series of investments. When the items to be allocated are discrete, and must be allocated to one or another use in their entirety, then the problem must be solved with integer programming. In the following example, the variables {x1,x2,x3,x4} are 0-1. There are several constraints. The objective function is given by

```
Max (z) = .2 x1 + .3 x2 + .5 x3 + .1 x4
```
(32)

This problem may be solved as follows. We minimize the negative of z to maximize, and enter all the constraints, which

are less than inequalities:

```
In[259]:=
ClearAll[x1, x2, x3, x4]

In[260]:=
InterchangeMethodMin[-(.2 x1 + .3 x2 + .5 x3 + .1 x4),
  {.5 x1 + x2 + 1.5 x3 + .1 x4 - 3.1, .3 x1 + .8 x2 + 1.5 x3 + .4 x4 - 2.5,
   .2 x1 + .2 x2 + .3 x3 + .1 x4 - .4}, {x1, x2, x3, x4}, .001, Starts → 4] // Timing

Out[260]=
{0.187, {{{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6},
  {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}}}
```

Seven random starts were used, but only 2 unique starts were used due to duplications. Two starts found the correct answer {0,0,1,1}. On larger problems, performance might not be quite this good. TabuSearch also can solve this problem correctly:

```
In[261]:=
TabuSearchMin[-(.2 x1 + .3 x2 + .5 x3 + .1 x4), {.5 x1 + x2 + 1.5 x3 + .1 x4 - 3.1,
   .3 x1 + .8 x2 + 1.5 x3 + .4 x4 - 2.5, .2 x1 + .2 x2 + .3 x3 + .1 x4 - .4},
  {x1, x2, x3, x4}, .001, Starts → 4, TabuListLength → 1] // Timing

Out[261]=
{0.047, {{{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6},
  {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}}}
```

Note that Tabu Search may perform worse than Interchange on small problems. Note also that a small tabu list size is needed for this problem. If we specify a tabu list too long, a warning results:

```
In[262]:=
TabuSearchMin[-(.2 x1 + .3 x2 + .5 x3 + .1 x4), {.5 x1 + x2 + 1.5 x3 + .1 x4 - 3.1,
   .3 x1 + .8 x2 + 1.5 x3 + .4 x4 - 2.5, .2 x1 + .2 x2 + .3 x3 + .1 x4 - .4},
  {x1, x2, x3, x4}, .001, Starts → 4, TabuListLength → 100] // Timing

   Warning: TabuListLength too long, will cause gridlock

      TabuListLength shortened to 2

Out[262]=
{0.062, {{{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6},
  {{x1 → 1, x2 → 1, x3 → 0, x4 → 0}, -0.5}, {{x1 → 0, x2 → 0, x3 → 1, x4 → 1}, -0.6}}}
```

VII.2.B Vehicle Routing/Travelling Salesman

A very common problem in operations research is vehicle routing. This problem arises in delivery service, airline, and other transportation contexts. To illustrate, define a set of 8 cities with coordinates:

c={{1,1},{2,0.1},{3,0.1},{4,1},{1,2},{2,2},{3,2},{4,2}} (33)

Where the objective is to minimize the sum of the links between cities. We define the objective function as the sum of the distance*existence of the link over all possible links between cities. Self-loops (e.g., x11) are removed from the lists. Constraints are added that force every city to have at least 2 links (for entry and exit). Finally, the sum of all edges must be at least 8. We really want exactly 8 links, but the algorithm does not accept equality constraints. The minimization of the sum of link distances will force the solution to exactly 8. Links are defined between cities such that there are no repeated links (e.g., x13 but no x31) to make it faster.

```
In[263]:=
lis = {x12, x13, x23, x14, x24, x34, x15, x25, x35, x45, x16, x26, x36,
   x46, x56, x17, x27, x37, x47, x57, x67, x18, x28, x38, x48, x58, x68, x78}

Out[263]=
{x12, x13, x23, x14, x24, x34, x15, x25, x35, x45, x16, x26, x36,
 x46, x56, x17, x27, x37, x47, x57, x67, x18, x28, x38, x48, x58, x68, x78}

In[264]:=
InterchangeMethodMin[
  2 (1.3453624047073711` x12 + 2.193171219946131` x13 + 3 x14 + x15 + √2 x16 + √5 x17 +
      √10 x18 + 1.` x23 + 2.193171219946131` x24 + 2.1470910553583886` x25 + 1.9` x26 +
      2.1470910553583886` x27 + 2.758622844826744` x28 + 1.3453624047073711` x34 +
      2.758622844826744` x35 + 2.1470910553583886` x36 + 1.9` x37 + 2.1470910553583886`
       x38 + √10 x45 + √5 x46 + √2 x47 + x48 + x56 + 2 x57 + 3 x58 + x67 + 2 x68 + x78) +
   Abs[x12 + x13 + x14 + x15 + x16 + x17 + x18 - 2] + Abs[x12 + x23 + x24 + x25 + x26 + x27 + x28 - 2] +
   Abs[x13 + x23 + x34 + x35 + x36 + x37 + x38 - 2] +
   Abs[x14 + x24 + x34 + x45 + x46 + x47 + x48 - 2] +
   Abs[x15 + x25 + x35 + x45 + x56 + x57 + x58 - 2] +
   Abs[x16 + x26 + x36 + x46 + x56 + x67 + x68 - 2] +
   Abs[x17 + x27 + x37 + x47 + x57 + x67 + x78 - 2] +
   Abs[x18 + x28 + x38 + x48 + x58 + x68 + x78 - 2],
  {- (x12 + x13 + x14 + x15 + x16 + x17 + x18 + x23 + x24 + x25 + x26 + x27 + x28 + x34 + x35 +
       x36 + x37 + x38 + x45 + x46 + x47 + x48 + x56 + x57 + x58 + x67 + x68 + x78) + 8},
  lis, .01, Starts → 5, CompileOption → True, ShowProgress → False] // Timing

Out[264]=
{5.016,
 {{{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0, x45 → 0,
    x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0, x57 → 0,
    x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814}}}
```

We see that the function found the solution in 4 out of 5 starts, which is a loop around the points in a rough circle. Even the starts that failed got quite close to the best solution and had the proper number of links defined. The timing is quite good. The problem of subgraphs that are not linked to form a complete vehicle route can occur in this problem but is unlikely unless the subgraphs produce exactly the same total travel distance as the linked route.

```
In[265]:=
TabuSearchMin[
  2 (1.3453624047073711` x12 + 2.193171219946131` x13 + 3 x14 + x15 + Sqrt[2] x16 + Sqrt[5] x17 +
     Sqrt[10] x18 + 1.` x23 + 2.193171219946131` x24 + 2.1470910553583886` x25 + 1.9` x26 +
     2.1470910553583886` x27 + 2.758622844826744` x28 + 1.3453624047073711` x34 +
     2.758622844826744` x35 + 2.1470910553583886` x36 + 1.9` x37 + 2.1470910553583886` x38 +
     Sqrt[10] x45 + Sqrt[5] x46 + Sqrt[2] x47 + x48 + x56 + 2 x57 + 3 x58 + x67 + 2 x68 + x78) +
   Abs[x12 + x13 + x14 + x15 + x16 + x17 + x18 - 2] + Abs[x12 + x23 + x24 + x25 + x26 + x27 + x28 - 2] +
   Abs[x13 + x23 + x34 + x35 + x36 + x37 + x38 - 2] +
   Abs[x14 + x24 + x34 + x45 + x46 + x47 + x48 - 2] +
   Abs[x15 + x25 + x35 + x45 + x56 + x57 + x58 - 2] +
   Abs[x16 + x26 + x36 + x46 + x56 + x67 + x68 - 2] +
   Abs[x17 + x27 + x37 + x47 + x57 + x67 + x78 - 2] +
   Abs[x18 + x28 + x38 + x48 + x58 + x68 + x78 - 2],
  {-(x12 + x13 + x14 + x15 + x16 + x17 + x18 + x23 + x24 + x25 + x26 + x27 + x28 + x34 + x35 +
      x36 + x37 + x38 + x45 + x46 + x47 + x48 + x56 + x57 + x58 + x67 + x68 + x78) + 8},
  lis, .01, Starts → 5, TabuListLength → 60] // Timing

Out[265]=
{0.859,
 {{{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0, x45 → 0,
    x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 0, x17 → 0, x27 → 0, x37 → 0, x47 → 0, x57 → 1,
    x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 21.3814},
  {{x12 → 0, x13 → 0, x23 → 1, x14 → 0, x24 → 1, x34 → 0, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 1, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 1,
    x57 → 0, x67 → 0, x18 → 0, x28 → 0, x38 → 1, x48 → 0, x58 → 0, x68 → 0, x78 → 1}, 22.3374},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 0, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 1,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 19.5192},
  {{x12 → 0, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 1, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 19.5192},
  {{x12 → 1, x13 → 0, x23 → 1, x14 → 0, x24 → 0, x34 → 1, x15 → 1, x25 → 0, x35 → 0,
    x45 → 0, x16 → 0, x26 → 0, x36 → 0, x46 → 0, x56 → 1, x17 → 0, x27 → 0, x37 → 0, x47 → 0,
    x57 → 0, x67 → 1, x18 → 0, x28 → 0, x38 → 0, x48 → 1, x58 → 0, x68 → 0, x78 → 1}, 17.3814}}}
```

We see that the TabuSearchMin function found the solution in 1 out of 5 starts, which is a loop around the points in a rough circle, but in 1/3 the time of the InterchangeMethod. Even the starts that failed got quite close to the best solution and had the proper number of links defined. The timing is quite good. The problem of subgraphs that are not linked to form a complete vehicle route can occur in this problem but is unlikely unless the subgraphs produce exactly the same total travel distance as the linked route.

VII.2.C Minimum Spanning Tree

The construction of a minimum spanning tree is a problem that comes up in problems of delivery from fixed warehouses, utility network design, and other applications. We begin with the weighted graph, with nodes (a,b,c,d,e,f), and with edges and weights defined by:

{xab,xac,xae,xbc,xbd,xcd,xce,xcf,xdf,xef} (34)

{7, 2, 2, 1, 3, 3, 4, 4, 5, 6} (35)

This can be solved with the interchange method. The objective function is to minimize the sum of the weighted edges. This produces the minimum tree we desire. We know that we want 5 edges as the final solution (#nodes -1), so we add an inequality that the sum of the edges must be greater than 5. This allows more than 5 initially, but then edges are trimmed as the algorithm proceeds. If an equality constraint is put in, the program will fail. Additional constraints force every node to have at least one edge touching it.

```
In[266]:=
InterchangeMethodMin[7 xab + 2 xac + 2 xae + 1 xbc + 3 xbd + 3 xcd + 4 xce + 4 xcf + 5 xdf + 6 xef,
  {-(xab + xac + xae + xbc + xbd + xcd + xce + xcf + xdf + xef) + 5,
   -(xab + xac + xae) + 1, -(xab + xbc + xbd) + 1, -(xac + xbc + xcd + xce + xcf) + 1,
   -(xbd + xcd + xdf) + 1, -(xae + xce + xef) + 1, -(xcf + xdf + xef) + 1},
  {xab, xac, xae, xbc, xbd, xcd, xce, xcf, xdf, xef}, .1,
  Starts → 1, CompileOption → True] // Timing

Out[266]=
{0.156, {{{xab → 0, xac → 1, xae → 1,
    xbc → 1, xbd → 0, xcd → 1, xce → 0, xcf → 1, xdf → 0, xef → 0}, 12.}}}
```

We see that the function returns the correct solution {ac,ae,bc,bd,cf} with a very fast execution time. This reflects the fast computational time for binary variables. Large problems can thus be solved.

```
In[267]:=
TabuSearchMin[7 xab + 2 xac + 2 xae + 1 xbc + 3 xbd + 3 xcd + 4 xce + 4 xcf + 5 xdf + 6 xef,
  {-(xab + xac + xae + xbc + xbd + xcd + xce + xcf + xdf + xef) + 5,
   -(xab + xac + xae) + 1, -(xab + xbc + xbd) + 1, -(xac + xbc + xcd + xce + xcf) + 1,
   -(xbd + xcd + xdf) + 1, -(xae + xce + xef) + 1, -(xcf + xdf + xef) + 1},
  {xab, xac, xae, xbc, xbd, xcd, xce, xcf, xdf, xef}, .1, Starts → 1,
  TabuListLength → 10, CompileOption → True] // Timing

Out[267]=
{0.031, {{{xab → 0, xac → 1, xae → 1,
    xbc → 1, xbd → 0, xcd → 1, xce → 0, xcf → 1, xdf → 0, xef → 0}, 12.}}}
```

We see that the function returns the correct solution {ac,ae,bc,bd,cf} with a very fast execution time. For such smaller problems, the Tabu Search approach does not provide superior speed.

VIII THE MaxAllocation FUNCTION

VIII.1 Introduction

MaxAllocation is a function to maximize a nonlinear function subject to a linear constraint.

$$\text{Max } (f(x_i)) \text{ s.t } .0 < x_i < \text{rhs}, \sum_{i=1}^{n} x_i = \text{rhs} \qquad (36)$$

This type of problem comes up often in finance, economics, and investment. Traditional algorithms have a great deal of trouble with such a problem, though apparently simple. A new algorithm enables a very efficient solution of this type of problem. The algorithm is based on the idea of path following. If one can stay on the constraint line, then the solution stays feasible at all times. An approach to path following is based on the SWAP algorithm (Loehle, 2001). We first discretize the interval of the rhs. This discretization is made proportional to the number of variables, N, for convenience. The initial 2N pieces are placed onto all the variables equally (each variable is given a value 2*rhs/N). At iteration 1, each variable is tested to see if swapping a single piece from that variable would improve the solution. This performs a rectangular move in N space. After no more pieces can be moved from variable 1 (either all have been moved or no improvement is obtained), testing proceeds to variable 2 to see if any pieces can be moved, and so on. Note that some of the pieces moved from 1 to 2 might be redistributed at this point. After the algorithm has tested all the variables in order for the possibility of making a swap, the number of pieces on each variable is doubled (with half the length) for the next iteration. This enables a finer resolution of the function.

The function is defined by

```
MaxAllocation[function, varlist, rhs, tolerance, options]
```
(37)

where rhs is the right hand side value (from eqn (1)) and tolerance defines the stopping criterion. There are three options for the program. MaxIterations will stop the program at the specified number of iterations if desired. CompileOption is useful if the function is not compilable. ShowProgress shows intermediate stages of the computation.

Operation of the function is shown in the following example. This example is a sum of diminishing return terms. This is a function with no nonlinear interaction terms. It can be shown that the algorithm converges to the analytic optimum in this case.

```
In[268]:=
ClearAll[x2]

In[269]:=
varlist = {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15,
   x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30,
   x31, x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42, x43, x44, x45,
   x46, x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60};

In[270]:=
v[x_] := Sum[i * (1.0 - E^(-3. * x[[i]])), {i, 1, 60}]
```

Note the technique used here to take advantage of indexed variables. The function v is defined with a tensor as an argument. Then function vvv is defined to accept a list and turn it into a tensor by passing it down to v. This function can be

compiled, and is in the following example.

```
In[271]:=
vvv = Function[{x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15,
    x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30,
    x31, x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42, x43, x44, x45,
    x46, x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60},
   v[{x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16,
     x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30, x31,
     x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42, x43, x44, x45, x46,
     x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60}]];

In[272]:=
t = Table[1, {i, 1, 60}]

Out[272]=
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

In[273]:=
v[t]

Out[273]=
1738.89

In[274]:=
Apply[vvv, t]

Out[274]=
1738.89

In[275]:=
MaxAllocation[vvv[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15,
   x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30,
   x31, x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42, x43, x44, x45,
   x46, x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60],
  varlist, 1.0, .1, CompileOption → True, ShowProgress → False] // Timing

Out[275]=
{1.547, {{x1 → 0, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 0, x7 → 0, x8 → 0, x9 → 0, x10 → 0, x11 → 0,
   x12 → 0, x13 → 0, x14 → 0, x15 → 0, x16 → 0, x17 → 0, x18 → 0, x19 → 0, x20 → 0, x21 → 0,
   x22 → 0, x23 → 0, x24 → 0, x25 → 0, x26 → 0, x27 → 0, x28 → 0, x29 → 0, x30 → 0,
   x31 → 0, x32 → 0, x33 → 0, x34 → 0, x35 → 0, x36 → 0, x37 → 0, x38 → 0, x39 → 0,
   x40 → 0, x41 → 0, x42 → 0, x43 → 0, x44 → 0.00416667, x45 → 0.0125, x46 → 0.01875,
   x47 → 0.0270833, x48 → 0.0333333, x49 → 0.0395833, x50 → 0.0479167, x51 → 0.0541667,
   x52 → 0.0604167, x53 → 0.0666667, x54 → 0.0729167, x55 → 0.0791667, x56 → 0.0854167,
   x57 → 0.0916667, x58 → 0.0958333, x59 → 0.102083, x60 → 0.108333}, 146.317}}
```

Next, we can compare the value of compiling the function in terms of timing.

```
In[276]:=
MaxAllocation[vvv[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15,
   x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30,
   x31, x32, x33, x34, x35, x36, x37, x38, x39, x40, x41, x42, x43, x44, x45,
   x46, x47, x48, x49, x50, x51, x52, x53, x54, x55, x56, x57, x58, x59, x60],
  varlist, 1.0, .1, CompileOption -> False] // Timing

Out[276]=
{5.109, {{x1 → 0, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 0, x7 → 0, x8 → 0, x9 → 0, x10 → 0, x11 → 0,
   x12 → 0, x13 → 0, x14 → 0, x15 → 0, x16 → 0, x17 → 0, x18 → 0, x19 → 0, x20 → 0, x21 → 0,
   x22 → 0, x23 → 0, x24 → 0, x25 → 0, x26 → 0, x27 → 0, x28 → 0, x29 → 0, x30 → 0,
   x31 → 0, x32 → 0, x33 → 0, x34 → 0, x35 → 0, x36 → 0, x37 → 0, x38 → 0, x39 → 0,
   x40 → 0, x41 → 0, x42 → 0, x43 → 0, x44 → 0.00416667, x45 → 0.0125, x46 → 0.01875,
   x47 → 0.0270833, x48 → 0.0333333, x49 → 0.0395833, x50 → 0.0479167, x51 → 0.0541667,
   x52 → 0.0604167, x53 → 0.0666667, x54 → 0.0729167, x55 → 0.0791667, x56 → 0.0854167,
   x57 → 0.0916667, x58 → 0.0958333, x59 → 0.102083, x60 → 0.108333}, 146.317}}
```

We can see that the compiled version runs 5 times faster.

It would be useful to have a proof that this algorithm works. At this time, no proof is available. The algorithm has been tested on a suite of problems with known solutions, and gives good results. The algorithm has been designed to guard against infinite loops. It is possible that a problem could be devised with local minima that would prevent a global optimum from being reached, but no such problem has yet been encountered. The initial equitable distribution of pieces across the variables guards against interaction (e.g., x1*x2) type terms which can not be improved by rectangular moves if they are initialized to zero.

The above problem with 250 variables took 10 minutes to run on a Pentium III 600 MHz machine. Because the execution time goes up nonlinearly with the number of variables, it is likely that the upper limit for an overnight run is 350-1000 variables, depending on the machine used.

VIII.2 Applications

VIII.2.A Investment Allocation Problems

In many investment problems, the goal is to allocate a fixed amount of money across a series of investments. We need not invest in all the options available. A typical problem to solve is the following, (from A.K. Dixit, Optimization in Economic Theory): A capital sum C is available for allocation among n investment projects. The expected return from a portfolio of x_j projects is

$$\sum_{j=1}^{n} \left[\alpha_j x_j - 0.5\, \beta_j x_j^2\right] \tag{38}$$

subject to:

$$\sum_{j=1}^{nx_j} = C \tag{39}$$

This problem may be solved as follows:

```
In[277]:=
a = {1000, 2000, 3000};

In[278]:=
rhs = 10.0;

In[279]:=
b = {100, 500, 1000.};

In[280]:=
h[x_] := Sum[a[[i]] * x[[i]] - .5 * b[[i]] * x[[i]]^2, {i, 1, 3}];

In[281]:=
fff = Function[{a1, a2, a3}, h[{a1, a2, a3}]];

In[282]:=
MaxAllocation[fff[a1, a2, a3], {a1, a2, a3}, rhs, .1, ShowProgress → True]

   Global Optimization, Version 5.2

   number of variables = 3

   constraint rhs = 10.

   tolerance = 0.1

   {{a1 → 4.66667, a2 → 2.83333, a3 → 2.5}, 11612.5}

   {{a1 → 4.58333, a2 → 2.91667, a3 → 2.5}, 11614.6}

   {{a1 → 4.625, a2 → 2.91667, a3 → 2.45833}, 11615.4}

   {{a1 → 4.625, a2 → 2.91667, a3 → 2.45833}, 11615.4}

Out[282]=
{{a1 → 4.625, a2 → 2.91667, a3 → 2.45833}, 11615.4}
```

Here we find the solution which is an interesting mix of investments. This problem can be extended to 350 to 1000 variable cases for overnight runs, depending on machine speed.

VIII.2.B Derivative Hedging with Quadratic Programming

A common problem in finance is derivative construction to hedge an investment. One technique for derivative construction is to construct a target payoff value (T) for the instrument, and then to allocate investment to a series of instruments with return R to get as close to the payoff value as possible using a quadratic objective function (for 3 instruments here):

$$\sum_{i=1}^{3} \left[T_i - \sum_{j=1}^{3} x_j \, R_{i,j} \right]^2 \tag{40}$$

$$\sum_{j=1}^{3} x_j \, p_j \leq C \tag{41}$$

This can be solved with MaxAllocation. First, for a problem with 3 possible investments, we make a dummy variable (index 4 here) to take up the slack in the investment (to represent the part of C NOT invested). Also, the equation is rearranged so that x represents the dollar amount invested in each instrument, so x is divided by the price p in eqn. (3). Then C represents our rhs and the problem is given by:

```
In[283]:=
    p = {1, 1, 1};

In[284]:=
    targetpayoff = {1000, 2000, 3000};

In[285]:=
    payoff = {{1000, 0, 0}, {0, 2000, 0}, {0, 0, 3000}};

In[286]:=
    fy[x_] := -Sum[(targetpayoff[[i]] - Sum[(x[[j]] / p[[j]]) * payoff[[i, j]],
                                {j, 1, 3}])^2, {i, 1, 3}];

In[287]:=
    fff = Function[{a1, a2, a3, a4}, fy[{a1, a2, a3}]];

In[288]:=
    MaxAllocation[fff[a1, a2, a3, a4], {a1, a2, a3, a4}, 6., .1]


Out[288]=
    {{a1 → 0.999976, a2 → 0.999976, a3 → 0.999976, a4 → 3.00007}, -0.00834465}
```

In this example, we know the exact answer, and the algorithm converged to the exact answer quickly. Such derivative construction problems can be solved with hundreds of variables.

IX APPLICATIONS

IX.1 Zeros of a Function/Roots of a Polynomial

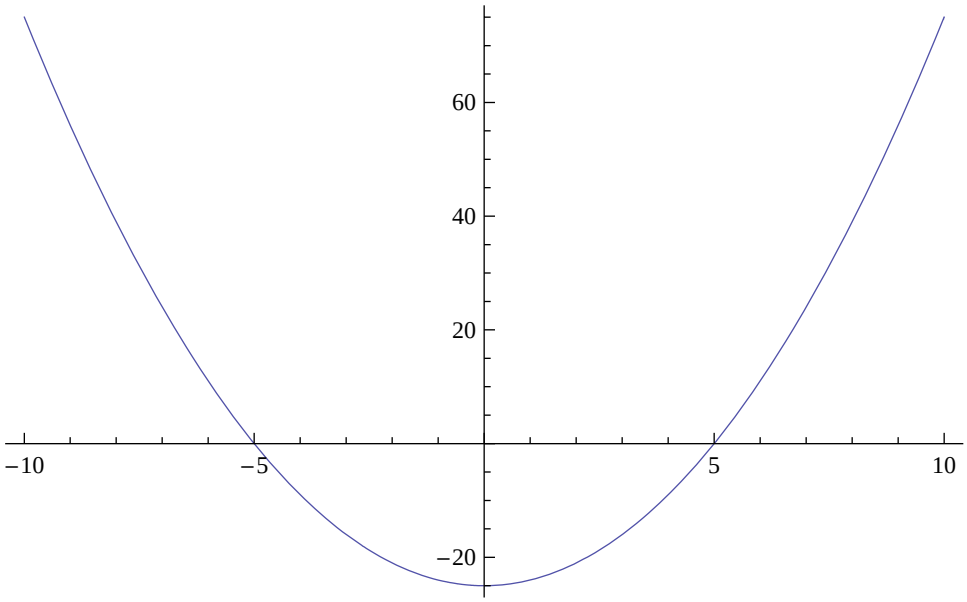
Although the *Mathematica* function FindRoot can find the zeros of a function, it can occur that a function is not analytic, in which case FindRoot will not work. For example, a function might need to be calculated recursively, algorithmically, or by simulation. FindRoot also will only find a single root at a time from a given starting point. In addition, we may invoke the need for identifying the indifference zone: we may want all the solutions that are within some indifference region of the zero or root. It is a simple matter to use either GlobalMinima or MultiStartMin to find roots of a function. Consider the function

$$x^2 - 25 \tag{42}$$

where the roots or zeros are obviously +5 and -5. As formulated, the function looks like

```
In[289]:=
Plot[x^2 - 25, {x, -10, 10}]
```

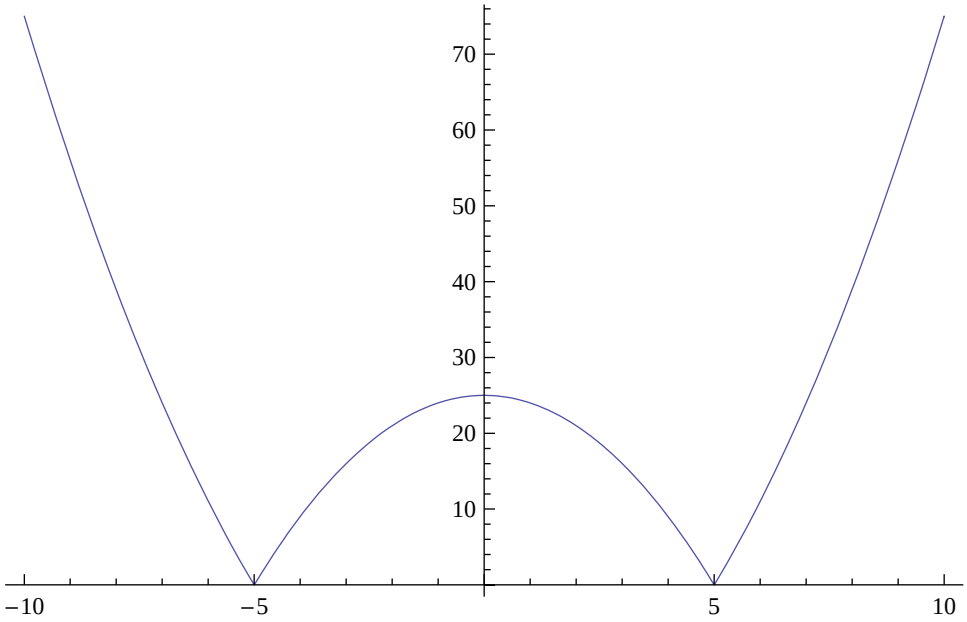
```
Out[289]=
```



and the minimum is at -25. If, however, we evaluate the absolute value of the function

```
In[290]:=
Plot[Abs[x^2 - 25], {x, -10, 10}]
```

```
Out[290]=
```



then we can simply minimize the function as in all the above examples and we will find the roots or zeros of the function,

which are now the minima. In cases where there is a region of solutions with values near zero, this procedure will likewise allow us to define these regions.

```
In[291]:=
GlobalMinima[Abs[x^2 - 25], , {{x, -10, 10}}, 10, 0.00001, 0.1, 0.0000001] // Timing

Out[291]=
{2.94209 × 10^-15, {{{x → -5}, 0}, {{x → 5}, 0}}}

In[292]:=
MultiStartMin[Abs[x^2 - 25], {}, {}, {{x, -10, 10}}, .0000001, Starts → 4] // Timing

Out[292]=
{0.047, {{{x → 5.}, 0.0000489897}, {{x → 5.}, 0.0000171186},
  {{x → 5.}, 0.0000425679}, {{x → 5.}, 5.81469 × 10^-6}}}

In[293]:=
GlobalSearch[Abs[x^2 - 25], {}, {}, {{x, -10, 10}}, .0000001, Starts → 4] // Timing

Out[293]=
{0.063, {{{x → -5.}, 1.5376 × 10^-7}, {{x → 5.}, 3.72529 × 10^-7},
  {{x → 5.}, 3.06218 × 10^-7}, {{x → 5.}, 7.96277 × 10^-8}}}
```

We can see that all three functions find the two solutions but on this small problem GlobalMinima is faster. The other 2 functions also need multiple starts to find both solutions.

IX.2 Integer Programming: the Knapsack Problem

As discussed above, MultiStartMin allows integer variables. This implies that it can solve integer programming problems. Optimization problems with integer variables are difficult because they violate the assumptions of continuous methods such as LP or gradient descent. Various methods have been applied to integer programming problems, including heuristic search. The method used here for integer variables is a generalized descent with discrete step sizes, combined with limited interchange. This method is illustrated for the knapsack problem. In the knapsack problem, the optimum return on packing of discrete variables is desired. This could occur when packing a truck, where different packages have different shipping values. On the other hand, the size of each package may differ, and the total space available is constrained. We next test such a problem. Because we wish to maximize, we put a negative sign in front of the objective function. To use GlobalPenaltyFn on this problem, it is necessary to multiply 100 times the positivity constraints to prevent it from going negative. Only the best solutions are shown in the output. The GlobalPenaltyFn solution can be quite close.

```
In[294]:=
    GlobalPenaltyFn[-(x1 + 2 x2 + 4 x3 + 15 x4 + 80 x5 + 100 x6), {-100 x1, -100 x2, -100 x3,
       -100 x4, -100 x5, -100 x6, 1 x1 + 3 x2 + 15 x3 + 29 x4 + 70 x5 + 59 x6 - 300}, ,
      {{x1, 0, 5, Integer}, {x2, 0, 5, Integer}, {x3, 0, 5, Integer},
       {x4, 0, 5, Integer}, {x5, 0, 5, Integer}, {x6, 0, 5, Integer}},
      .00001, ShowProgress → False, Starts → 20] // Timing

Out[294]=
    {9.766, {{{x1 → 4, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 4, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.},
      {{x1 → 2, x2 → 1, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -504.}}}

In[295]:=
    MultiStartMin[-(x1 + 2 x2 + 4 x3 + 15 x4 + 80 x5 + 100 x6),
      {-x1, -x2, -x3, -x4, -x5, -x6, 1 x1 + 3 x2 + 15 x3 + 29 x4 + 70 x5 + 59 x6 - 300}, ,
      {{x1, 0, 2, Integer}, {x2, 0, 2, Integer}, {x3, 0, 2, Integer},
       {x4, 0, 2, Integer}, {x5, 0, 2, Integer}, {x6, 0, 4, Integer}},
      .00001, ShowProgress → False, Starts → 10] // Timing

Out[295]=
    {0.766, {{{x1 → 20, x2 → 5, x3 → 0, x4 → 1, x5 → 0, x6 → 4}, -445.}}}
```

We see that the correct solution {{5,0,0,0,0,5},-505.} was not found out of 10 or 20 starts with either method. A better approach is to first solve the continuous problem, and then pass the result back to the discrete problem:

```
In[296]:=
    res = GlobalSearch[-(x1 + 2 x2 + 4 x3 + 15 x4 + 80 x5 + 100 x6),
       {-x1, -x2, -x3, -x4, -x5, -x6, 1 x1 + 3 x2 + 15 x3 + 29 x4 + 70 x5 + 59 x6 - 300}, ,
       {{x1, 0, 2}, {x2, 0, 2}, {x3, 0, 2}, {x4, 0, 2}, {x5, 0, 2}, {x6, 0, 4}},
       .00001, ShowProgress → False, Starts → 1] // Timing

Out[296]=
    {0.547, {{{x1 → 0., x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 5.08475}, -508.475}}}
```

We next pass this result, after we make it Integer, back to MultiStartMin or GlobalPenaltyFn, obtaining the correct solution:

```
In[297]:=
    s = {x1, x2, x3, x4, x5, x6} /. res[[2, 1, 1]]

Out[297]=
    {0., 0, 0, 0, 0, 5.08475}

In[298]:=
    s = Round[s]

Out[298]=
    {0, 0, 0, 0, 0, 5}
```

```
In[299]:=
MultiStartMin[-(x1 + 2 x2 + 4 x3 + 15 x4 + 80 x5 + 100 x6),
  {-x1, -x2, -x3, -x4, -x5, -x6, 1 x1 + 3 x2 + 15 x3 + 29 x4 + 70 x5 + 59 x6 - 300}, ,
  {{x1, 0, 2, Integer}, {x2, 0, 2, Integer}, {x3, 0, 2, Integer},
   {x4, 0, 2, Integer}, {x5, 0, 2, Integer}, {x6, 0, 4, Integer}},
  .00001, ShowProgress → False, StartsList → {s}] // Timing

Out[299]=
{0.078, {{{x1 → 5, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -505.}}}

In[300]:=
GlobalPenaltyFn[-(x1 + 2 x2 + 4 x3 + 15 x4 + 80 x5 + 100 x6),
  {-x1, -x2, -x3, -x4, -x5, -x6, 1 x1 + 3 x2 + 15 x3 + 29 x4 + 70 x5 + 59 x6 - 300}, ,
  {{x1, 0, 2, Integer}, {x2, 0, 2, Integer}, {x3, 0, 2, Integer},
   {x4, 0, 2, Integer}, {x5, 0, 2, Integer}, {x6, 0, 4, Integer}},
  .00001, ShowProgress → False, StartsList → {s}] // Timing

Out[300]=
{0.469, {{{x1 → 3, x2 → 0, x3 → 0, x4 → 0, x5 → 0, x6 → 5}, -503.}}}
```

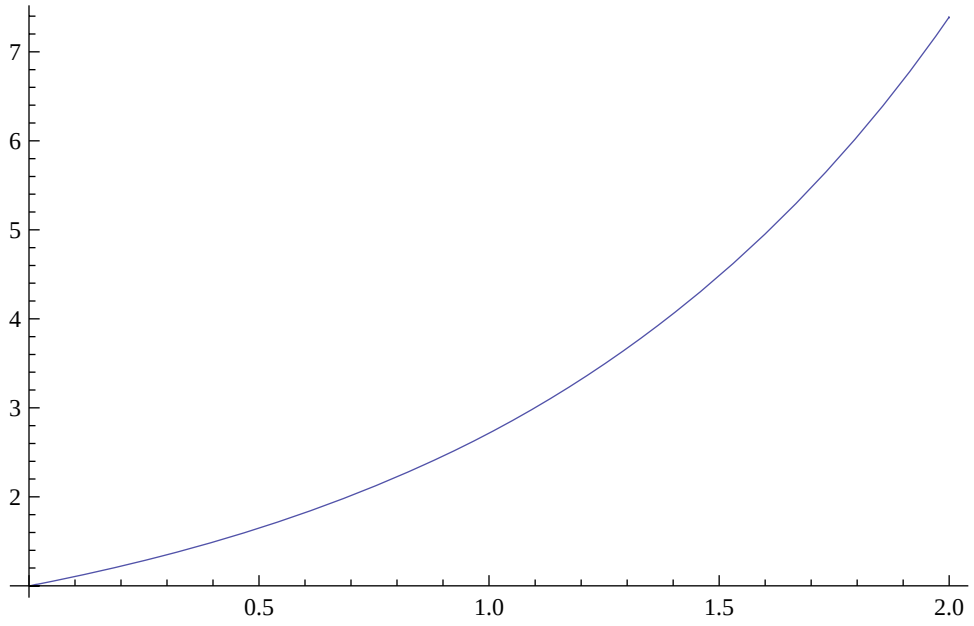
IX.3 Differential Equation Models

The following example shows how to fit parameters in a differential equation model.

```
In[301]:=
sol = NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 0, 2}]

Out[301]=
{{y → InterpolatingFunction[{{0., 2.}}, <>]}}

In[302]:=
y[1.5] /. sol

Out[302]=
{4.48169}
```

```
In[303]:=
Plot[y[t] /. sol, {t, 0, 2}]
```

```
Out[303]=
```




```
In[304]:=
fn = Function[{a},
  so = NDSolve[{y'[x] == y[x], y[0] == a}, y, {x, 0, 2}]; j = y[1.5] /. so; j[[1]]]
```

```
Out[304]=
Function[{a}, so = NDSolve[{y'[x] == y[x], y[0] == a}, y, {x, 0, 2}]; j = y[1.5] /. so; j[[1]]]
```

```
In[305]:=
fn[1]
```

```
Out[305]=
4.48169
```

```
In[306]:=
gg[x_] := Abs[fn[x] - fn[1]]
```

```
In[307]:=
GlobalSearch[gg, {}, , {{x, 0, 2}}, .00001,
  CompileOption → False, SimplifyOption → False, Starts → 1] // Timing
```

```
Out[307]=
{1.563, {{{x → 1.}, 5.47942 × 10^-6}}}
```

In this example we see that we recovered the original parameter a=1. Note that CompileOption->False was necessary because a *Mathematica* function was passed in, which prohibits compilation. SimplifyOption->False was also needed to prevent error messages (although the correct answer is still obtained without it). It is also possible to fit models in cases where we want to fit the entire time trajectory to the data, in which case the function is evaluated at many points and some measure of fit computed.

IX.4 Constraint Equations

One of the hurdles to solving nonlinear optimization problems is finding a feasible starting point. This is particularly true when there are a large number of constraints. In the following example, it is clear that the feasible region is the positive quarter of a sphere.

```
In[308]:=
ClearAll[z]

In[309]:=
constraints = {-x, -y, -z, x^2 + y^2 + z^2 - 2}

Out[309]=
{-x, -y, -z, -2 + x^2 + y^2 + z^2}
```

It is possible to use optimization to find the feasible region. We set up the problem to solve for a constant objective function, so that the program stops when it finds feasible initial points.

```
In[310]:=
MultiStartMin[2, constraints, , {{x, -5, 5}, {y, -5, 5}, {z, -5, 5}},
  .001, Starts → 10, ShowProgress → False] // Timing

Out[310]=
{0.281, {{{x → 0.0483366, y → 0.53029, z → 1.13155}, 2},
  {{x → 0.09089, y → 0.52393, z → 0.421732}, 2},
  {{x → 1.10698, y → 0.0178877, z → 0.36338}, 2},
  {{x → 0.545995, y → 1.02937, z → 0.0676439}, 2},
  {{x → 0.47304, y → 0.109673, z → 0.254154}, 2},
  {{x → 0.251385, y → 0.464905, z → 0.297733}, 2},
  {{x → 0.199679, y → 0.72498, z → 0.116262}, 2},
  {{x → 0.172754, y → 0.45899, z → 0.258331}, 2},
  {{x → 0.206245, y → 0.773349, z → 0.158459}, 2},
  {{x → 0.00838564, y → 1.14862, z → 0.119339}, 2}}}

In[311]:=
GlobalPenaltyFn[2, constraints, , {{x, -5, 5}, {y, -5, 5}, {z, -5, 5}},
  .001, Starts → 10, ShowProgress → False] // Timing

Out[311]=
{1.032, {{{x → 1.1668, y → 0.174565, z → 0.186763}, 2},
  {{x → 0.490512, y → 0.275311, z → 0.040621}, 2},
  {{x → 0.036959, y → 0.65716, z → 0.162351}, 2},
  {{x → 0.869512, y → 0.237804, z → 0.103923}, 2},
  {{x → 0.579303, y → 1.08266, z → 0.17406}, 2},
  {{x → 0.294281, y → 0.611031, z → 0.0758022}, 2},
  {{x → 0.150908, y → 0.0202404, z → 0.275322}, 2},
  {{x → 0.986036, y → 0.493321, z → 0.11452}, 2},
  {{x → 1.38911, y → 0.155131, z → 0.154396}, 2},
  {{x → 0.000117562, y → 0.0162639, z → 0.469868}, 2}}}
```

```
In[312]:=
GlobalSearch[2, constraints, , {{x, -5, 5}, {y, -5, 5}, {z, -5, 5}},
  .001, Starts → 10, ShowProgress → False] // Timing

CompiledFunction::cfn : Numerical error encountered
   at instruction 11; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 11; proceeding with uncompiled evaluation. >>

CompiledFunction::cfn : Numerical error encountered
   at instruction 11; proceeding with uncompiled evaluation. >>

General::stop : Further output of
   CompiledFunction::cfn will be suppressed during this calculation. >>

Out[312]=
{0.328, {{{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2},
  {{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2},
  {{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2},
  {{x → 0, y → 0, z → 0}, 2}, {{x → 0, y → 0, z → 0}, 2}}}
```

The above result gives a set of feasible points, which gives a basis for defining the feasible region.

X. Literature Cited

Cadzow, J. A. and Martens, H.R., 1970. Discrete-Time and Computer Control Systems, Prentice-Hall, Inc.

Courrieu, P. 1997. The hyperbell algorithm for global optimization: A random walk using Cauchy densities. J. Global Optimization 10:37-55.

Crooke, P., L. Froeb, and S. Tschantz. 1999. Maximum likelihood estimation. *Mathematica* in Education and Research 8:17-23.

Csendes, T. 1985. A simple but hard-to-solve global optimization test problem. IIASA Workshop on Global Optimization, Sopron, Hungary.

Cvijovic, D. and J. Klinowski. 1995. Taboo search: an approach to the multiple minima problem. Science 267:664-666.

Densham, P., and G. Rushton. 1992. A more efficient heuristic for solving large p-nedian problems. *Pap. Reg. Sci.* 71:307-329.

Dixon, L.C.W., and G.P. Szego. 1978. The global optimization problem: An introduction. *Towards Global Optimization 2*, North-Holland, Amsterdam.

Goldberg, J. and L. Paz. 1991. Locating emergency vehicle bases when service time depends on call location. *Transportation Science* 25:264-280.

Griewank, A.O., 19 ___. Generalized descent for global optimization.

Ingber, L., and B. Rosen. 1992. Genetic algorithms and very fast simulated annealing: A comparison. Mathematical and Computer Modelling 16:87-100.

Jones, D.R., C.D. Perttunen, and B.E. Stuckman. 1993. Lipschitzian optimization without the Lipschitz constant. J. Optimization Theory and Applications 79:157-181.

Lin, S. and B. Kernighan. 1973. An effective heuristic algorithm for the traveling salesman problem. *Operations Research* 21:498-516.

Loehle, C. 2000. Optimal control of spatially distributed process models. Ecological Modelling 131:79-95.

Murray, A.T. and R.L. Church. 1995. Heuristic solution approaches to operational forest planning problems. OR Spektrum 17:193-203.

Nelson, J. and J.D. Brodie. 1990. Comparison of random search algorithm and mixed integer programming for solving area-based forest plans. *Canadian J. Forest Research* 20:934-942.

Pinter, J.D. 1996. Global Optimization in Action. Kluwer Academic Publishers, Boston.

Phillips, C. L. and Nagle, H. T., 1995. Digital Control System Analysis and Design, 3rd edition, Prentice-Hall, Inc.

Shahian, B. and Hassul, M., 1993. Control System Design Using MATLAB, Prentice-Hall, Inc.

Teitz, M. and P. Bart. 1968. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research* 16:955-961.