The numanal extension:[1]

This extension provides a number of primitives for finding the roots of both single variable equations and multivariable systems of equations; for optimizing (finding the minima of) both univariate and multivariable equations; and for estimating integrals. Some were more or less adapted from Numerical Recipes in C: The Art of Scientific Computing[2]. Others are dependent upon the Apache Commons Math3 library[3] or the PAL Math[4] library

This version of the extension assumes that functions can be passed to it in the form of NetLogo anonymous reporters and therefore works only with NetLogo versions 6.0 and above. The JAMA Matrix class of java methods is used internally to the extension for performing linear algebra[5]. Jama-1.0.3.jar, commons-math3-3.6.1.jar and PalMathLibrary.jar, all distributed with this package, must be placed in the same directory as the numanal.jar file. If you use NetLogo's Extension Manager this will be done automatically when the extension is installed.

Version 4.0 of numanal is compiled against NetLogo 7.0. You can find examples of the use of (almost) all these primitives in the "Examples" directory.

---

[1] This extension arose from the need for smart agents to optimize production and pricing decisions. One thing led to another, and in casting about for the best optimization procedures for messy problems I ended up collecting them all in one extension. Take your choice!

[2] Numerical Recipes in C: The Art of Scientific Computing, 2nd ed., 1992, by William Press, Saul A. Teukolsky, William T. Vetterling and Brian Flannery. I should note that a number of other algorithm sources were also consulted and there were extensive modifications made in the translation to the Java and NetLogo environments, and in the explicit use of matrix algebra. Any mistakes, then, are entirely mine.

[3] The Apache Commons Math3 library may be found at http://www.apache.org. The software carries the following notices: *The software is licensed under the Apache Software Foundation license which grants in part "a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form." You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0.*

[4] The PAL Math library v1.4 may be found at http://iubio.bio.indiana.edu/soft/molbio/java/pal/. The PAL Math software caries the following notice: *Copyright (c) 1999-2002 by the PAL Development Core Team. This package may be distributed under the terms of the GNU Lesser General Public License (LGPL). PAL Development Core Team: Alexei Drummond, School of Biological Sciences, University of Auckland, Korbinian Strimmer, Department of Zoology, University of Oxford, Ed Buckler, Department of Genetics, North Carolina State University.*

[5] Documentation and jar files may be found at http://math.nist.gov/javanumerics/jama/. The JAMA software carries the following notice: Copyright Notice *This software is a cooperative product of The MathWorks and the National Institute of Standards and Technology (NIST) which has been released to the public domain. Neither The MathWorks nor NIST assumes any responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.*

Primitives for Finding Roots

BrentRoot

This is an implementation of the Brent algorithm to find the root (zero) of a function of a single variable, x. The return value is the value of x at the root.

Usage:

let xAtRoot numanal:BrentRoot fnctn lowBound highBound
let xAtRoot (numanal:BrentRoot fnctn lowBound highBound rtol atoll)

BrentRoot requires three inputs:

fnctn is an anonymous reporter that evaluates the function for which the root is to be found. The reporter itself should take a single argument, x, and report the value of the function evaluated at x.

lowBound and **highBound** are the bounds between which the root is to be found. Note that an exception is thrown if the range between the two bounds does not contain a root of the function, that is if the function evaluated at x = **lowBound** does not have a different sign than the function evaluated at x = **highBound**. If **lowBound** is greater than **highBound** then the bounds are reversed.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be.

rtol is the relative tolerance to which the solution is to be found. BrentRoot works by efficiently reducing the distance between values of x that bracket the root. If that distance falls below **rtol** times the absolute value of the algebraically larger of the current bounds, then the solution is assumed to have been found. If **rtol** is negative, the default value (10^{-12}) is used. **rtol** may be zero to indicate that only atol should be used, but if **atol** is also zero, the default values for both are used.

atol is the absolute tolerance to which the solution is to be found. BrentRoot works by efficiently reducing the distance between values of x that bracket the root. If that distance falls below **atol**, then the solution is assumed to have been found. If **atol** is negative, the default value (10^{-12}) is used. **atol** may be zero to indicate that only rtol should be used, but if **rtol** is also zero, the default values for both are used.

Newton-root
Broyden-root

Both these primitives implement the respective algorithms for finding the root (the "zero") of a set of n nonlinear equations in the same set of n variables. Both return as a NetLogo list the point at which the root occurs, a list of the values of the n variables that simultaneously set each of the n equations to zero. Although the Newton method is guaranteed to converge on a root, it requires that the Jacobian of the system of equations be calculated at each step, which can be quite time intensive if the equations are complex and/or if the number of equations is large. Unlike the Newton method, the Broyden algorithm attempts to update the current Jacobian with the results of past steps rather than recalculating it at each step. Only if the update does not yield a successful step is the Jacobian calculated anew. If there are many equations and/or the calculation of each equation (and thus the Jacobian) is time-consuming, this procedure will be considerably faster than the Newton method.

usage:

```
let rootList  numanal:Newton-root  guess  fnctn
let rootList  (numanal:Newton-root  guess  fnctn  stpmx  tolf  tolx  tolmin
      max_its  epsilon  alpha)

let rootList  numanal:Broyden-root  guess  fnctn
let rootList  (numanal:Newton-root  guess  fnctn  stpmx  tolf  tolx  tolmin
      max_its  epsilon  alpha)
```

Both primitives require two inputs:

guess is a NetLogo list containing an initial guess for the root. Since the set of equations may have multiple roots, the root to which Newton-root converges may be sensitive to the initial guess.

fnctn is an anonymous reporter that evaluates the set of equations for which the root is to be found. The reporter itself should take a single argument, a NetLogo list of input values, the coordinates of the n-dimensional point at which each equation is to be evaluated, and report a NetLogo list containing the value of each equation at that point.

The Newton and Broyden algorithms have many parameters that determine the accuracy with which the procedures find the root. The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. For a complete discussion of these parameters consult Numerical Recipes or any other presentation of the two methods. **A value of zero for any of these parameters indicates that the default value should be used.**

stpmx is a parameter in the calculation of the maximum step size in the LineSearch routine. The default value is 100.

tolf is the primary criterion by which the algorithm judges whether it is close enough to the root. One way of knowing when it has found it is to look at the deviation from zero of each the equations evaluated at the trial point. In particular, if F is the vector of results for any given input vector, X, we look at f = 0.5*F*F', that is half of the sum of squared values of the results. (F' is the transpose of F.) If f is small enough, i.e., less than tolf, it has found the root. The default value of tolf is 10^{-6}.

tolx and **tolmin** are secondary criteria by which the algorithm judges whether it is close enough to the root. It is possible that the algorithm will find a local or global minimum of f before it finds the root. In that case, the change in X that is required to get f to fall will get smaller and smaller as it approaches that minimum. The Newton procedure checks for that by seeing if the maximum proportionate change in the elements of X falls below tolx, or if the largest gradient in any of the X directions falls below tolmin. Of course, it is possible that the minimum is actually at the root, so the calling program may want to check on that. The default values of tolx and tolmin are both 10^{-8}. Experimentation suggests that tolx and tolmin be a couple of orders of magnitude smaller than tolf.

max_its is the maximum number of iterations (steps) allowed. If it is exceeded an ExtensionException is thrown. The default value is 1000.
epsilon is the proportional change in each x element that is used to calculate the Jacobian of the system of equations. Its default value is 10^{-4}. (If an x element is close to zero, then the calculated change may fall below the precision of java doubles. In that case, the change is set equal to $(\text{Double.MIN\_NORMAL})^{1/2}$. This can be changed in the source code.)
alpha is a parameter that ensures that the LineSearch routine has been able to find a new X vector that reduces f by a sufficient amount. The default value is 10^{-6}.

The Newton and Broyden algorithms may get stuck at local or global minimum that is not a true root. Rather than returning an error, the primitives will return that point. Two primitives can be used to check whether the solution is in fact a root.

Newton-failed? and **Broyden-failed?** return true if the prior call to **Newton-root** or **Broyden-root** resulted in a "soft" error, i.e., if it seems to have found a local or global minimum, and they return false if a true root has been found. However, a soft failure may indeed be at a root and one should check to see if that is the case before rejecting the solution.

Scarf's Fixed-Point Algorithm

scarfs-fxdpt

The Scarf Algorithm finds a fixed point of a multivariate system of equations by finding the values of the variables that map back on themselves. For instance, in economics the algorithm can be used to find a set of prices that satisfy both the supply and demand equations of an arbitrary number of goods. The algorithm was proposed by Herbert Scarf in *The Computation of Economic Equilibria*, Yale University Press, 1973, and this particular implementation was translated from a Fortran implementation authored by Frank Westhoff.

Usage:

```
let fxdPt  numanal:scarfs-fxdpt nvar fnctn
let fxdPt  (numanal:scarfs-fxdpt nvar fnctn useRelativeError
        keepIterationLog)
```

scarfs-fxdpt requires two inputs:
nVar is the dimension of the system, the number of variables.
fnctn is an anonymous reporter. The reporter should take a single argument, a list x of nVar input values, and report a list consisting of two sublists. In the case of supply and demand equations, for instance, **fnctn** takes as its input a set of nVar prices for nVar goods and evaluates the quantities demanded at those prices and the quantities supplied. It then returns a list of two lists, one with the quantities demanded and the other with the quantities supplied. The algorithm works to find a set of prices that yield the quantity supplied equal to the quantity demanded for each good. In both supply and demand, the quantities for any one good may depend on one or all of the prices.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be.

useRelativeError indicates whether the algorithm should use the maximum of the relative differences or the maximum of the absolute differences between the respective elements of the two output lists as the stopping criterion. In the supply/demand example, the algorithm will compare each good's quantity supplies and quantity demanded, and work to make the largest of those differences, in either relative or absolute terms, fall below the default tolerance. If specified, **useRelativeError** should be set to TRUE or FALSE. It's default value is TRUE. The default tolerance is 10^{-8} and is set in the code for the algorithm.

keepIterationLog indicates whether the algorithm should keep a log of each iteration, which can then be reported by **scarfs-fxdpt-info**. If specified, **keepIterationLog** should be set to TRUE or FALSE. It's default value is FALSE.

scarfs-fxdpt-info

The Scarf algorithm collects a lot of information about the solution it has found and this may be retrieved through the **scarfs-fxdpt-info** primitive.

Usage:

let fxdPtInfo numanal:scarfs-fxdpt-info

scarfs-fxdpt-info takes no arguments and returns a list with five items.

- item 0 an integer exit flag: 1 if the algorithm converged successfully; 2 if the algorithm failed to converge on a solution.
- item 1 a Boolean indicating whether or not **useRelativeError** was set.
- item 2 the total number of iterations used in finding the solution.
- item 3 the total time taken to find the solution, in milliseconds.
- item 4 a text exit message giving the tolerance to which the solution was found.

NOTE: the current version of **scarfs-fxdpt-info** collects the iteration log, but has no option for returning it.

Primitives for Optimization (Minimization)

Brent-minimize
BrentA-minimize

Both of these primitives employ the Brent algorithm to minimize a function of one variable, x. The return value is the value of x between the upper and lower bound where the function takes on its minimum value. **BrentA-minimize** uses the Apache Commons Math3 package.

Usage:

```
let xAtMin numanal:Brent-minimize fnctn lowBound highBound
let xAtMin (numanal:Brent-minimize fnctn lowBound highBound rtol atol)
let xAtMin numanal:BrentA-minimize fnctn lowBound highBound
let xAtMin (numanal:BrentA-minimize fnctn lowBound highBound rtol atol)
```

Both primitives require three inputs:

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, x, and report the value of the function evaluated at x.

lowBound and **highBound** are the x axis bounds between which the minimum is to be found. Note that if the range between the two bounds does not contain the "true" minimum of the function, the algorithm returns the bound closest to the minimum. Moreover, if **highBound** < **lowBound**, the bounds are reversed.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. NOTE, for Brent-minimize, **rtol** and **atol** are evaluated separately and whichever is achieved first will stop the search for the minimum. However, for BrentA-minimize, **rtol** and **atol** are combined to calculate a tolerance of $\mathbf{tol} = \mathbf{rtol} * |\mathbf{x}| + \mathbf{atol}$ and the minimum is found when **tol** is achieved.

rtol is the relative tolerance to which the solution is to be found. In Brent-minimize, if **rtol** is negative, the default value (10^{-12}) is used. **rtol** may be zero, but if **atol** is also zero, the default values for both are used. In BrentA-minimize, if **rtol** is non-positive, the default value is used.

atol is the absolute tolerance to which the solution is to be found. In Brent-minimize, if **atol** is negative, the default value (10^{-12}) is used. **atol** may be zero, but if **rtol** is also zero, the default values for both are used. In BrentA-minimize, if **atol** is non-positive, the default value is used.

BOBYQA-minimize

This is an implementation of Powell's BOBYQA algorithm from the Apache Commons Math3 library, which states: “This implementation is translated and adapted from the Fortran version available here. See this paper for an introduction. BOBYQA is particularly well suited for high dimensional problems where derivatives are not available. In most cases it outperforms the

PowellOptimizer significantly. Stochastic algorithms like CMAESOptimizer succeed more often than BOBYQA, but are more expensive. BOBYQA could also be considered as a replacement of any derivative-based optimizer when the derivatives are approximated by finite differences." BOBYQA-minimize can return an unbounded or a bounded solution. See the **bounds-set** primitive.

let xlist numanal:BOBYQA-minimize guess fnctn
let xlist (numanal:BOBYQA-minimize guess fnctn initialRadius
  stoppingRadius maxEvals nInterpolation)

Two inputs are required.

guess is a NetLogo list containing the initial guess for the point (input values) at which the function is minimized. Entering a guess that is closer to the global minimum is more likely to avoid the problem of getting stuck at a local minimum.

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, a NetLogo list of input values, that is the coordinates of the n-dimensional point at which the function is to be evaluated, and report the value of the function at that point.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. A value of zero for any of these inputs indicates that the default value should be used.

initialRadius the initial trust radius. The default is 10.0.

stoppingRadius the stopping radius – essentially the solution tolerance. The default is 10^{-8}.

maxEvals the maximum number of function evaluations. The default is 50,000.

nInterpolation the number of interpolation points. The default (suggested by the literature) is 2*nvar + 1, where nvar is the dimension of the problem.

CDS-minimize
CGS-minimize

These primitives implement two different minimization algorithms from the PAL math library. CDS-minimize finds the minimum without using derivatives, employing Brent's modification of a conjugate direction search method proposed by Powell. It therefore is appropriate for poorly conditioned functions. CGS-minimize, on the other hand, uses numerically calculated first and second derivatives of the multivariate function in finding the solution. It may therefore not be suitable for functions whose derivatives are poorly behaved. Both primitives can return unbounded or bounded solutions. See the **bounds-set** primitive. **NOTE** that the calls to the two primitives are somewhat different.

let xlist numanal:CDS-minimize guess fnctn illConditioned
let xlist (numanal:CDS-minimize guess fnctn illConditioned tolfx tolx
  stepSize maxEvals scaleParam)

let xlist numanal:CGS-minimize guess fnctn
let xlist (numanal:CGS-minimize guess fnctn tolfx tolx stepSize maxEvals searchType)

Three inputs are required for CDS-minimize and two for CGS-minimize.

guess is a NetLogo list containing the initial guess for the point (input values) at which the function is minimized. Entering a guess that is closer to the global minimum is more likely to avoid the problem of getting stuck at a local minimum.

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, a NetLogo list of input values, that is the coordinates of the n-dimensional point at which the function is to be evaluated, and report the value of the function at that point.

illConditioned should be set to TRUE if the problem is known to be ill-conditioned and to FALSE if it is not. This variable may be automatically set to TRUE if the problem is found to be ill-conditioned during the iterations for its solution. **This parameter is used by CDS-minimize only.**

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. Values of zero indicate the default.

tolfx the optimization stops if over a specified number of evaluations of the objective function, the value of the objective function changes by less than tolfx at each evaluation. This presumably means that the value could change in the same direction by an amount less than tolfx each time so that the lowest and highest value differ by n*tolfx, where n is a specified number of successive evaluations. (The value for n is set in the code to 4.) If tolfx is zero, only tolx is used. If both are zero the default value of 10^{-12} is used for both.

tolx the optimization stops if on two successive evaluations every element of the solution vector is within tolx of its value in the prior iteration. If tolx is zero, only tolfx is used. If both are zero the default value of 10^{-12} is used for both.

stepSize is a step length parameter and should be set equal to the expected distance between the guess the expected solution. Exceptionally small or large values of stepSize lead to slower convergence on the first few iterations. The default value for stepSize is 1.0.

maxEvals the maximum number of function evaluations. The default is no limit.

searchType determines the method for the conjugate gradient direction update. The default is Beal-Sorenson, Hestenes-Stiefel.
1 -> Fletcher-Reeves,
2 -> Polak-Ribiere,
3 -> Beale-Sorenson, Hestenes-Stiefel.
This parameter is used by CGS-minimize only.

scaleParam is a scaling parameter. 1.0 is the default and indicates no scaling. If the scales for the different variables are very different, scaleParam should be set to a value of about 10.0. **This parameter is used by CDS-minimize only.**

CMAES-minimize

This is an implantation of the CMA-ES algorithm from the Apache Commons Math3 library, which states: "An implementation of the active Covariance Matrix Adaptation Evolution Strategy (CMA-ES) for non-linear, non-convex, non-smooth, global function minimization. The CMA-Evolution Strategy (CMA-ES) is a reliable stochastic optimization method which should be applied if derivative-based methods, e.g. quasi-Newton BFGS or conjugate gradient, fail due to a rugged search landscape (e.g. noise, local optima, outlier, etc.) of the objective function. Like a quasi-Newton method, the CMA-ES learns and applies a variable metric on the underlying search space. Unlike a quasi-Newton method, the CMA-ES neither estimates nor uses gradients, making it considerably more reliable in terms of finding a good, or even close to optimal, solution. In general, on smooth objective functions the CMA-ES is roughly ten times slower than BFGS (counting objective function evaluations, no gradients provided). For up to 10 variables the derivative-free simplex direct search method (Nelder and Mead) can also be faster, but it is far less reliable than CMA-ES. The CMA-ES is particularly well suited for non-separable and/or badly conditioned problems. To observe the advantage of CMA compared to a conventional evolution strategy will usually take about 30 function evaluations. On difficult problems the complete optimization (a single run) is expected to take *roughly* between 30 and 300 N^2 function evaluations. This implementation is translated and adapted from the Matlab version of the CMA-ES algorithm as implemented in module cmaes.m version 3.51. For more information, please refer to the following links: Matlab code, Introduction to CMA-ES, Wikipedia." CMAES-minimize can return an unbounded or a bounded solution. See the **bounds-set** primitive.

```
let xlist numanal:CMAES-minimize guess fnctn sigma
let xlist (numanal:BOBYQA-minimize guess fnctn sigma lambda rtol atol
        maxEvals checkFC isActiveCMA? diagOnly)
```

Three inputs are required.

guess is a NetLogo list containing the initial guess for the point (input values) at which the function is minimized. Entering a guess that is closer to the global minimum is more likely to avoid the problem of getting stuck at a local minimum.

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, a NetLogo list of input values, that is the coordinates of the n-dimensional point at which the function is to be evaluated, and report the value of the function at that point.

sigma a list of sigma values. Sources differ on the best values, but suggest values ranging from one-third to one times the distance from the guess for a given variable to its likely value at the minimum of the function. There are no default values.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. A value of zero for any of these inputs indicates that the default value should be used.

lambda the initial population size. The commonly accepted value is $\lambda = 4 + (\text{int})(3.0 * \ln(\text{nvar})$ where nvar is the number of variables. This is the default value.

rtol relative solution tolerance. The default is 10^{-12}.

atoll absolute solution tolerance. The default is 10^{-12}.

maxEvals the maximum number of function evaluations. The default is 50,000.

checkFC Determines how often new random objective variables are generated in case they are out of bounds. The default is zero.

isActiveCMA? A Boolean specifying whether the covariance matrix update method should be used. The default is true.

diagOnly the number of initial iterations where the covariance matrix remains diagonal. The default is zero.

DES-minimize

This is an implementation of a differential evolution search (a form of genetic algorithm) to find the minimum of a multivariate function. This method employs a metaheuristic search without the need to evaluate derivatives and gradients, and is therefore appropriate for poorly conditioned functions. DES-minimize uses the DifferentialEvolution class of the PAL math library. DES-minimize can return an unbounded or a bounded solution. See the **bounds-set** primitive.

```
let xlist numanal:DES-minimize guess fnctn
let xlist (numanal:DES-minimize guess fnctn tolfx tolx popsize CR F
    maxEvals)
```

Two inputs are required.

guess is a NetLogo list containing the initial guess for the point (input values) at which the function is minimized. Entering a guess that is closer to the global minimum is more likely to avoid the problem of getting stuck at a local minimum.

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, a NetLogo list of input values, that is the coordinates of the n-dimensional point at which the function is to be evaluated, and report the value of the function at that point.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be.

tolfx the optimization stops if over a specified number of evaluations of the objective function, the value of the objective function changes by less than tolfx at each evaluation. This presumably means that the value could change in the same direction by an amount less than tolfx each time so that the lowest and highest value differ by n*tolfx, where n is a specified number of successive evaluations. (The value for n is set in the code to 4.) If tolfx is zero, only tolx is used. If both are zero the default value of 10^{-12} is used for both.

tolx the optimization stops if on two successive evaluations every element of the solution vector is within tolx of its prior value. If tolx is zero, only tolfx is used. If both are zero the default value of 10^{-12} is used for both.

popsize is the population size used by the DifferentialEvolution algorithm. If popsize is zero, the algorithm calculates and appropriate value.

CR is the crossing over factor. If CR is zero, the default value of 0.9 is used.

F is the weight factor. If F is zero, the default value of 0.7 is used.

maxEvals the maximum number of function evaluations. The default is no limit.

simplex
simplex-MD
simplex-NM

These three primitives implement different versions of the simplex method. **simplex** is a loose java translation of the standard simplex algorithm using the Nelder-Mead method, as outlined in *Numerical Recipes*. **simplex-MD** and **simplex-NM** use the simplex routines in the Apache Commons Math3 library, the first using the multi-dimensional method and the second the Nelder-Mead method. The Apache library routines are perhaps more robust, but also a bit slower. All three can return an unbounded or a bounded solution. See the **bounds-set** primitive.

Usage:

```
let xlist numanal:simplex guess fnctn
let xlist (numanal:simplex guess fnctn sideLength rtol atol maxEvals
nrestarts
        nevalsmod tolfactor)
let xlist numanal:simplex-MD guess fnctn
let xlist (numanal:simplex-MD guess fnctn sideLength rtol atol maxEvals)
let xlist numanal:simplex-NM guess fnctn
let xlist (numanal:simplex-NM guess fnctn sideLength rtol atol maxEvals)
```

All three primitives require two inputs:

guess is a NetLogo list containing the initial guess for the point (input values) at which the function is minimized. Simplex converges pretty rapidly to a minimum from any initial guess, but if there are local minima, it may get caught at one. Thus entering a guess that is closer to the global minimum is more likely to avoid the problem of getting stuck at a local minimum.

fnctn is an anonymous reporter that evaluates the function to be minimized. The reporter itself should take a single argument, a NetLogo list of input values, that is the coordinates of the n-dimensional point at which the function is to be evaluated, and report the value of the function at that point.

The following optional inputs require that the primitive and its inputs be enclosed in parentheses. Not all need be included, but all those to the left of the last one to be set must be. Note that **simplex-MD** and **simplex-NM** do not use the last three of these.

sideLength is the initial the amount by which each element of the initial guess is perturbed to form the simplex. A value that is roughly 10% of the absolute value of the smallest element of the guess is not a bad place to start, although larger values often work quite nicely. Larger values may lead to faster convergence, but may also lead to overshooting. If **sideLength** is zero, the default value (10.0) is used.

rtol is the relative tolerance to which the solution is to be found. If any step reduces the value of fnctn by a proportion smaller than **rtol**, we assume that the minimum has been found. If **rtol** is zero or negative, only the absolute tolerance is used. If **atol** is also zero or negative, then both tolerances are set to the default value (10^{-12}).

atol is the absolute tolerance to which the solution is to be found. If any step reduces the value of fnctn by an amount smaller than **atol**, we assume that the minimum has been found. If a**tol** is zero or negative, only the relative tolerance is used. If **rtol** is also zero or negative, then both tolerances are set to the default value (10^{-12}).

maxEvals insures that the procedure will not continue in an infinite loop if there is no convergence. This sets the maximum number of evaluations of the function to be minimized and throws an ExtensionException if that number is exceeded. In the case of a restart, the number of function evaluations is reset to zero. If **maxEvals** is zero, the default (10,000) is used.

nrestarts specifies the desired number of restarts of the simplex procedure. Many sources suggest a restart after the initial solution is found as the initial solution may be a false minimum. This, of course, will require more iterations, but given that it begins at the putative minimum, it should not require too many. Note that the user can specify more than one restart, although it is not clear that there is any benefit for doing so. The default number is zero. **simplex-MD** and **simplex-NM** do not use this input.

nevalsmod when the number of evaluations reaches an approximate multiple of this number, both the relative and absolute tolerances are increased by a factor of **tolfactor**. This allows the routine to relax the tolerance required for a solution if the number of function evaluations grows too large. By default, **nevalsmod** is set to (**maxEvals** + 1) so that the tolerance is not changed. **simplex-MD** and **simplex-NM** do not use this imput.

tolfactor is the factor by which the tolerances are multiplied after each **nevalsmod** evaluations. **tolfactor** must be >= 1.0. Its default value is 2.0. **simplex-MD** and **simplex-NM** do not use this imput.

Using the optimization primitives to find a root:

Remember that the Newton and Broyden methods for finding roots depend on minimizing f = 0.5*F*F', where F is the vector of deviations of each function result from zero. Therefore, directly minimizing 2f, the sum of the squared deviations, should give us the same result, as indeed should minimizing the sum of the absolute deviations. With this in mind, the **simplex** and other optimization primitives can be used to do just that.

Whether it is faster to use one of the minimization routines or **Broyden-root** will likely depend on the size of the system of equations and the complexity of finding the Jacobian. For instance, simplex normally makes many more iterations with function7 being evaluated at each iteration but never has to calculate the Jacobian.

Primitives for Integration

Romberg-integrate

This primitive uses the Romberg method (based on trapezoids) to integrate a function between the designated lower and upper bounds of the function's single argument.

Usage:

let area numanal:Romberg-integrate fnctn lowbound highbound

Romberg-integrate takes three inputs:

fnctn is an anonymous reporter that evaluates the function to be integrated. The reporter itself should take a single argument, a value of x, and report the value of the function for that x.

lowbound and **highbound** are the values between with the function should be integrated.

Primitives for Setting Bounds

bounds-set

a command which sets the upper and lower bounds for a bounded optimization.

Usage:

numanal:bounds-set lowerBounds upperBounds

bounds-set requires two inputs:

lowerBounds and upperBounds are two NetLogo lists containing respectively the lower bounds for each variable and the upper bounds for each variable. Each list must have the same number of elements as the **guess** list. The bounds can be different for each variable.

bounds-clear

A command which clears any bounds that have been set by a **bounds-set** command. Any further optimizations will be unbounded. This command takes no inputs.

Usage:

numanal:bounds-clear

bounds-get

A reporter that reports the current bounds as set by the most recent **bounds-set** command. **bounds-get** returns a list of two sublists. The first sublist (item 0) contains the current lower bounds and the second sublist (item 1) contains the current upper bounds.

Usage:

let current-bounds numanal:bounds-get

bounds-get-defaults

A reporter that returns a two-element list containing the default lower bound and the default upper bound. The default lower bound is Java's Double.NEGATIVE_INFINITY and the default upper bound is Java's Double.POSITIVE_INFINITY.

Usage:

let default-bounds numanal:bounds-get-defaults

An Example

The primitives of the numal extension are all used in the same manner as other NetLogo extensions. The extension name must be included in the extensions line of model's NetLogo code and individual primitive names prefixed by **numanal:**. Each primitive takes a required set of arguments, as noted above for each, and in many cases optional arguments. When optional arguments are specified, the whole expression, starting with **numanal:** and ending with the last argument, must be enclosed in parentheses.

Each primitive (except those related to the setting of bounds) takes as one of its arguments an anonymous reporter, a NetLogo procedure that typically takes a list of input values and returns a single number that is the result of evaluating the function to be analyzed at that point.

Here is one example. Many more are provided in the Examples directory.

Minimize $y = \sum_{i=1}^{6} x_i{}^2$ and then $y = \sum_{i=1}^{6} (x_i + 1)^2$ using the (unconstrained) simplex method. Note that first is minimized at [0 0 0 0 0 0] where the function equals 0.0, while the second is minimized at [-1 -1 -1 -1 -1 -1], where the function also equals zero. (The first function is minimized twice, first using the default values for the optional arguments and then using an absolute tolerance of 10^{-6} and an initial sidelength of 10.) Finally, we constrain the solution for the second function to the first quadrant, using all the defaults. With a single minimum, the guess is arbitrary.

```
Extensions [ numanal ]

to go
    let guess [100 100 100 100 100 100]
    let fnctn3  [[val] -> function3 val]
    let fnctn4  [[val] -> function4 val]
    let xlist numanal:simplex  guess  fnctn3
    show xlist
    set xlist (numanal:simplex  guess  fnctn3  10   0   1.0E-6)
    show xlist

    set xlist numanal:simplex  guess  fnctn4
    show xlist

    let ubound item 1 numanal:bounds-get-defaults
    numanal:bounds-set n-values 6 [0]  nvalues 6 [ubound]
    set xlist numanal:simplex guess  funcn4
    show xlist
end
```

```
to-report function3 [ x ]
    report sum map [[z] -> z ^ 2] x
end

to-report function4 [ x ]
    report sum map [[z] -> (z + 1.0) ^ 2] x
end
```


Charles Staelin
Department of Economics
Smith College
Northampton, MA 01063