

The numanal extension:

This extension provides a number of primitives for finding the roots and minima of single variable equations, the minima of multi-variable equations, and the roots of systems of n equations in n variables. Several were more or less adapted from Numerical Recipes in C: The Art of Scientific Computing, 2nd ed., 1992, by William Press, Saul A. Teukolsky, William T. Vetterling and Brian Flannery. However, several other 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. So, all the errors are indeed mine.

This extension assumes that functions can be passed to it in the form of NetLogo tasks (reporters), and therefore works only with NetLogo version 5. Several of the primitives are written to use matrix algebra and so also require the Jama Matrix class of java methods for performing linear algebra. (See <http://math.nist.gov/javanumerics/jama/>.) The Jama-1.0.2.jar file, distributed with this package, must be placed in the same directory as the numanal.jar file. (The same Jama-1.0.2.jar file is bundled with the matrix extension that comes with the NetLogo distribution.)

For single variable equations the primitives include the Brent algorithm for finding the minimum and the Brent algorithm for finding the root. For single equations in several variables the primitives include the simplex algorithm for finding the minimum. Finally, for systems of n equations in n unknowns, the primitives include the Newton and Broyden algorithms for finding the n -dimensional root. (Note that the simplex algorithm can also be used to find the n -dimensional root of a system of n equations, as described at the end of this document.)

Here are the primitives and their use.

BrentMinimize: employs the Brent algorithm to minimize a function of one variable, x . The return value is the value of x between an upper and lower bound where the function takes on its minimum value.

Usage:

```
let xAtMin numanal:BrentMinimize fcn lowBound highBound tol
```

where:

fcn is a task variable that refers to a NetLogo reporter evaluating 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.

tol is the tolerance to which the solution is taken, as a proportion of the value of x at the minimum of the function. If the minimum occurs very close to $x = 0$, the tolerance is set to a small, positive number.

BrentRoot: employs the Brent algorithm to find the root (zero) of a function of one variable, x . The return value is the value of x at the root.

Usage:

```
let xAtRoot numanal:BrentRoot fcnfn bound1 bound2 tol
```

where:

fcnfn is a task variable that refers to a NetLogo reporter evaluating 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 .

bound1 and **bound2** are the bounds between which the root is to be found. Note that if the range between the two bounds does not contain a root of the function, that is if the function evaluated at $x = \text{bound1}$ does not have a different sign than the function evaluated at $x = \text{bound2}$, then an exception is thrown. **bound1** may be less than or greater than **bound2**.

tol is the tolerance to which the solution is taken, as a proportion of the value of x at the root of the function. If the root occurs very close to $x = 0$, the tolerance is set to a small, positive number.

Examples:

Find the minimum of $y = -xe^{-x}$ between -1 and 2 with $\text{tol} = 10^{-6}$. (The minimum is at $x = 1.0$, with a y value of -0.3679)

```
let fcnfn task [function1 ?]  
let xAtMin numanal:Brent-minimize fcnfn -1 2 1.0E-6  
print xAtMin  
print function1 xAtMin  
  
to-report function1 [ x ]  
  report (- x) * exp(- x)  
end
```

Find the minimum of function1 between 2 and 3 with $\text{tol} = 10^{-6}$. (The minimum in this range is at $x = 2.0$, i.e., at the lower limit of the range.)

```
let fcnfn task [function1 ?]  
let xAtMin numanal:Brent-minimize fcnfn 2 3 1.0E-6  
print xAtMin  
print function1 xAtMin
```

Find the root of $y = \sin(x + 90)$ between 0 and 180 with $\text{tol} = 10^{-6}$. (The root is at $x = 90$.)

```
let fcnfn task [function2 ?]  
let xAtRoot numanal:Brent-root fcnfn 0 180 1.0E-6
```

```
print xAtRoot
print function2 xAtRoot

to-report function2 [ x ]
  report sin (x + 90)
end
```

simplex: finds the minimum of a multivariate function passed to it in the task variable, `fnctn`, and returns a NetLogo list of the input values at the minimum, i.e., the point at which the function is minimized. Simplex comes in two flavors: the first (simplex) places no constraint on the input values; the second (simplex-nonneg) constrains all of the input values to be greater than or equal to zero.¹ (In the two-dimensional case, that constrains the solution to the first quadrant.)

Note that the simplex procedure can also be used to find the root of a set of multivariate equations. See the discussion following the Broyden primitive, below.

Usage:

```
let xlist numanal:simplex guess fnctn tol delta
let xlist numanal:simplex-nonneg guess fnctn tol delta
```

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 a task variable that refers to a NetLogo reporter evaluating 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.

tol is the tolerance to which the solution is to be found. If any step reduces the value of `fnctn` by less than `tol`, we assume that the minimum has been found. Note that `tol` is an absolute value, not a proportion of the value of the function at its minimum.

delta 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.

The simplex and simplex-nonneg extensions contain several other parameters that affect how the procedures work. These parameters may be set by the commands

¹ Restricting solutions to positive or zero values can be useful in situations where negative values make no sense, such as when solving for quantities or prices in economics applications. This is my own hack at the simplex algorithm. I cannot guarantee that it will always work.

(numamal:simplex-set nrestarts nevals_max nevals_mod nevals_tolfactor)
 numanal:simplex-reset

where:

nrestarts specifies the desired number of restarts of the simplex or simplex-nonneg 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 one and nrestarts may be set to zero.

nevals_max 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. The default value of nevals_max is 10,000.

nevals_mod - when the number of evaluations reaches an approximate multiple of this number, the tolerance is increased by a factor of nevals_tolfactor. This allows the routine to relax the tolerance required for a solution if the number of function evaluations grows too large. By default, nevals_mod is set to (nevals_max + 1) so that the tolerance is not changed.

nevals_tolfactor is the factor by which the tolerance is multiplied each nevals_mod evaluations. nevals_tolfactor must be ≥ 1.0 . Its default value is 2.0.

numanal:simplex-set takes a variable number of arguments, but it must have at least one, nrestarts. If it has more than one argument, the whole expression must be set in parentheses. E.g.,

```
numanal:simplex-set nrestarts
(numanal:simplex-set nrestarts, nevals_max)
```

The arguments must be in the same order as above, so to set nevals_tolfactor, for instance, values must be supplies for nrestarts, nevals_max and nevals_mod as well. All the parameters may be reset to their default values by **numanal:simplex-reset**.

Examples:

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] with a return value of 0.0, while the second is minimized at [-1 -1 -1 -1 -1 -1], with a return value of 6.0. Use a tolerance of 10^{-6} and an initial delta of 10. With a single minimum, the guess is arbitrary.

```
let guess [100 100 100 100 100 100]
let fnctn3 task [function3 ?]
let fnctn4 task [function4 ?]
```

```
let xlist numanal:simplex guess fctn3 1.0E-6 10
print xlist
print function3 xlist
```

```
set xlist numanal:simplex guess fctn4 1.0E-6 10
print xlist
print function3 xlist
```

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

```
to-report function4 [ x ]
  report sum map [ (? + 1) ^ 2 ] x
end
```

Now minimize the same functions using the constrained simplex. We get the same answer for function3, while the solution for function4 is constrained to [0 0 0 0 0] with a return value of zero.

```
set xlist numanal:simplex-nonneg guess fctn3 1.0E-6 10
print xlist
print function3 xlist
```

```
set xlist numanal:simplex-nonneg guess fctn4 1.0E-6 10
print xlist
print function3 xlist
```

Finally, change the number of restarts to 3; then change nrestarts to zero, nevals_mod to 40 and nevals_tolfactor to 5; and finally reset them all back to their defaults, minimizing function3 after each change.

```
numanal:simplex-set 3
set xlist numanal:simplex guess fctn3 1.0E-6 10
(numanal:simplex-set 0 1000 40 5)
set xlist numanal:simplex guess fctn3 1.0E-6 10
numanal:simplex-reset
set xlist numanal:simplex guess fctn3 1.0E-6 10
```

Newton-root: implements the Newton algorithm for finding the root (the "zero") of a set of n nonlinear equations in the same set of n variables. It is a reporter that returns as a NetLogo list the point at which the root occurs, that is 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. **Broyden-root** will likely be significantly more efficient in those cases as it does not generally require as many calculations of the Jacobian.

usage:

```
let rootList numanal:Newton-root guess fnctn
let failed? numanal:Newton-failed?
```

where:

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 a task variable that refers to a NetLogo reporter evaluating 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, that is 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.

numanal:Newton-failed? returns true if the prior call to numanal:Newton-root resulted in a "soft" error, i.e., if it seems to have found a local or global minimum, or false if a true root has been found. A soft failure may indeed be at a root and one should check to see if that is the case.

The code for numanal:Newton-root contains many parameters that determine the accuracy with which the Newton procedure finds the root. For a complete discussion of these parameters, consult [Numerical Recipes](#) or any other presentation of the Newton method. These parameters may be set by the commands

```
(numanal:Newton-set tolf tolmin max_its stpmx epsilon alpha)
numanal:Newton-reset
```

where:

tolf is the primary criterion by which we judge whether we are close enough to the root. We are looking for the root of the series of n equations and so one way of knowing when we've 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, we've found the root. The default value of tolf is 10^{-6} .

tolx and **tolmin** are secondary criteria by which we judge whether we are close enough to the root. It is possible that we instead find a local or global minimum of f before we find the root. In that case, the change in X that is required to get f to fall will get smaller and smaller as we approach 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.

stpmx is a parameter in the calculation of the maximum step size in the `LineSearch` routine.

The default value is 100.

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} .

numanal:Newton-set takes a variable number of arguments, but it must have at least one, `tolf`. If it has more than this single argument, the whole expression must be set in parentheses.

Examples:

Find the root of the system of equations

$$\begin{aligned} 2x_1 - x_2 - x_3 - e^{-x_1} &= 0 \\ -x_1 + 3x_2 - x_3 - e^{-x_2} &= 0 \\ -x_1 - x_2 + 4x_3 - e^{-x_3} &= 0 \end{aligned}$$

(The root is approximately at [0.7828 0.6088 0.4996])

```
let fcnfn task [function5 ?]
let guess [10 20 30]
let rootList numanal:Newton-root guess fcnfn
print rootList
if numanal:Newton-failed? [print "This may not be a true root."]
print function5 rootList
```

```
to-report function5 [ x ]
  let x1 item 0 x
  let x2 item 1 x
  let x3 item 2 x
  let y []
  set y lput ((2 * x1) - x2 - x3 - exp (- x1)) y
  set y lput ((- x1) + (3 * x2) - x3 - exp(- x2)) y
  set y lput ((- x1) - x2 + (4 * x3) - exp(- x3)) y
  report y
end
```

Or `function5` could be done in parts, as shown with `function6` below. (function-list could also be a global.)

```

set fnctn task [function6 ?]
set rootList numanal:Newton-root guess fnctn
print rootList
if numanal:Newton-failed? [print "This may not be a true root."]
print function5 rootList

to-report function6 [ x ]
  let function-list (list task [function6.1 ?] task [function6.2 ?] task [function6.3 ?])
  report map [(runresult ? x)] function-list
end

to-report function6.1 [ x ]
  let x1 item 0 x
  let x2 item 1 x
  let x3 item 2 x
  report (2 * x1) - x2 - x3 - exp (- x1)
end

to-report function6.2 [ x ]
  let x1 item 0 x
  let x2 item 1 x
  let x3 item 2 x
  report (- x1) + (3 * x2) - x3 - exp(- x2)
end

to-report function6.3 [ x ]
  let x1 item 0 x
  let x2 item 1 x
  let x3 item 2 x
  report (- x1) - x2 + (4 * x3) - exp(- x3)
end

```

Finally, change `tolf` to 10^{-4} ; then change `stpmx` to 10 (keeping `tolx`, `tolmin` and `max_its` at their default values and `epsilon` and `alpha` at their current values), and finally reset all the parameters to their defaults.

```

numanal:Newton-set 1.0E-4
let rootList numanal:Newton-root guess fnctn
(numanal:Newton-set 1.0E-4 1.0E-8 1.0E-8 1000 10)
let rootList numanal:Newton-root guess fnctn
numanal:simplex-reset
let rootList numanal:Newton-root guess fnctn

```


Broyden-root: implements the Broyden algorithm for finding the root (the "zero") of a set of n nonlinear equations in the same set of n variables. It is a reporter that returns as a NetLogo list the point at which the root occurs, that is the values of the n variables that simultaneously set each of the n equations to zero. 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:Broyden-root guess fnctn
let failed? numanal:Broyden-failed?
```

where:

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 a task variable that refers to a NetLogo reporter evaluating 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, that is 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.

numanal:Broyden-failed? returns true if the prior call to numanal:Broyden-root resulted in a "soft" error, i.e., if it seems to have found a local or global minimum, or false if a true root has been found. A soft failure may indeed be at a root and one should check to see if that is the case.

The Broyden method uses the same parameters as the Newton method. Thus there are two parallel primitives that set and reset those parameters:

```
(numanal:Broyden-set tolf tolx tolmin max_its stpmx epsilon alpha)
numanal:Broyden-reset
```

The **Broyden-root**, **Broyden-failed?**, **Broyden-set** and **Broyden-reset** primitives are used exactly as the Newton primitives, so the Newton examples, above, should suffice for demonstrating their usage.

Using simplex to find a root:

Finally, 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** primitive can be used to do just that. Redefine function6 as function7.

```
to-report function7 [ x ]  
  let function-list (list task [function6.1 ?] task [function6.2 ?] task [function6.3 ?])  
  report sum map [abs (runresult ? x)] function-list  
end
```

Using

```
set rootList numanal:simplex guess task [function7 ?] 1.0E-6 10
```

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

This NetLogo extension is distributed under the same conditions as is NetLogo itself. See the NetLogo Users' Manual for details.

Charles Staelin
Department of Economics
Smith College
Northampton, MA 01063
cstaelin@smith.edu