10 Minimization and maximization of functions

Functions:

mnbrak: bracket a minimum of a function
golden: find a minimum of a function by golden section
brent: find a minumum of a function by Brent's method
dbrent: find a minumum using derivative information
amoeba: minimum of a function, multidimensional downhill simplex
powell: minimum of a function, multidimensional Powel's method
rprmn: minimum of a function, multidimensional, conjugage gradient
dfpmin: minimum of a function, multidimensions, variable metric
simplex: linear programming simplex method
anneal: travelling salesman by simulated annealing

---

mnbrak

Description:

This function brackets a minimum of a Senac function.

mnbrak(ax, bx, func) $\longrightarrow$ [ax, bx, cx, fa, fb, fc]

Where:

ax is a real number on input representing an initial point and on output bracketing a minimum,
bx is a real number distinct from ax on input representing an initial point and on output bracketing a minimum,
func is the name of a Senac function of one variable,
cx is a floating point number bracketing a mini mum,
fa is the value of $func$ at ax,
fb is the value of $func$ at bx,
fc is the value of $func$ at cx.

Pre-variables:

mnbrak_glimit [100d0] a floating point number representing the maximum magnification allowed for a parabolic fit step,
mnbrak_tiny [1.0d-20] a floating point number.

Example:

The double root of the quadratic function $f(x) = (x-2)^2$ is bracketed. Then the same is done for the function $g(x) = sin(\pi x)$.

```
-> f(x) := (x-2)^2;
-> mnbrak(4,5,f)?
[2.3819660, 2.0000000, 1.3819660, 0.1458980, 0.0000000, 0.3819660]
-> pi : float(%pi)?
3.1415927
-> g(x) := sin(pi*x);
-> mnbrak(1/2,1/8,g)?
[0.1250000, -0.4817628, -1.4635255, 0.3826834, -0.9983591, 0.9934420]
```

golden

Description:

This function finds a minimum of a Senac function using golden section search.

golden(ax, bx, cx, f) ⟶ [value, xmin]

Where:

a, bx, cx are three real numbers which bracket f in that bx is between ax and cx and $f(bx)$ is less than both $f(ax)$ and $f(cx)$.
f is a Senac function of one real variable,
xmin is a floating point number such that f has a minumum at the corresponding point,
value is minimum value of the function.

Pre-variables:

golden_tol [1.0d-5] a floating point number determining the relative accuracy of the computation.

Example:

The minimum of the quadratic function $f(x) = (x-2)^2$ is found using `mnbrak` and `golden`. The same is then done for $g(x) = sin(\pi x)$.

```
-> f(x) := (x-2)^2;
-> bracket : mnbrak(4,5,f)?
[2.3819660, 2.0000000, 1.3819660, 0.1458980, 0.0000000, 0.3819660]
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> golden(ax,bx,cx,f)?
[0.0000000, 2.0000000]
-> pi : float(%pi);
-> g(x) := sin(pi*x);
-> bracket : mnbrak(1/2,1/8,g)?
[0.1250000, -0.4817628, -1.4635255, 0.3826834, -0.9983591, 0.9934420]
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> golden(ax,bx,cx,g)?
[-1.0000000, -0.5000012]
```

brent

Description:

This function finds a minimum of a Senac function using Brent's method.

brent(ax, bx, cx, f) ⟶ [value, xmin]

Where:

a, bx, cx are three real numbers which bracket f in that bx is between ax and cx and $f(bx)$ is less than both $f(ax)$ and $f(cx)$.
f is a Senac function of one real variable,
value is a floating point number representing a minumum value of f,
xmin is a floating point number representing the point where the minimum occured.

Pre-variables:

brent_tol [1.0d-4] a floating point number representing the relative accuracy,
brent_itmax [100] the maximum allowed number of iterations,
brent_zeps [1.0d-10] a floating point number which protects against attempting to achieve fractional accuracy for a minumum which happens to be exactly zero.

Example:

The critical point and minimum value of the quadratic function `f` and trigonometric function `g` are determined using `mnbrak` and `brent`.

```
-> f(x) := (x-2)^2;
-> bracket : mnbrak(4,5,f);
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> brent(ax,bx,cx,f)?
[0.0000000, 2.0000000]
-> pi : float(%pi);
-> g(x) := sin(pi*x);
-> bracket : mnbrak(1/2,1/8,g);
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> brent(ax,bx,cx,g)?
[-1.0000000, -0.4999995]
```

dbrent

Description:

This function finds a minimum of a Senac function using Brent's method employing function and derivative values.

dbrent(ax, bx, cx, f, df) $\longrightarrow$ **[value, xmin]**

Where:

a, bx, cx are three real numbers which bracket f in that bx is between ax and cx and $f(bx)$ is less than both $f(ax)$ and $f(cx)$.
f is a Senac function of one real variable representing the function to be minimized,
df is a Senac function of one real variable representing the functions derivative,
value is a floating point number representing a minumum value of f,
xmin is a floating point number representing the point where the minimum occured.

Pre-variables:

dbrent_tol [1.0d-4] a floating point number representing the relative accuracy,
dbrent_itmax [100] the maximum allowed number of iterations,
dbrent_zeps [1.0d-10] a floating point number which protects against attempting to achieve fractional accuracy for a minumum which happens to be exactly zero.

Example:

Minimums of `f(x)=(x-2)^2` and of `g(x)=sin(\pi x)` are found using `mnbrak` and `dbrent`.

```
-> f(x) := (x-2)^2;
-> df(x) := 2*(x-2);
-> bracket : mnbrak(4,5,f);
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> dbrent(ax,bx,cx,f,df)?
[0.0000000, 2.0000000]
-> pi : float(%pi);
-> g(x) := sin(pi*x);
-> dg(x) := pi*cos(pi*x);
-> bracket : mnbrak(1/2,1/8,g);
-> ax : first(bracket); bx : proj(bracket,2); cx : proj(bracket,3);
-> dbrent(ax,bx,cx,g,dg)?
[-1.0000000, -0.4999975]
```

amoeba

Description:

This function finds a minumum of a Senac function of many variables using the downhill-simplex method.

amoeba(p, y, func) $\longrightarrow$ **[p, y, iter]**

Where:

p is a $ndim + 1$ by $ndim$ dimensional real matrix such that on input the rows represent an initial simplex and on output a simplex all of whose points are within $ftol$ of a minumum of $func$,
y is a real list of length $ndim + 1$ representing the values of the function $func$ at the vertices of the simplex on input and output,
func is a Senac function of $ndim$ variables,
iter is the number of iterations taken.

Pre-variables:

amoeba_itmax [500] the maximum allowd number of iterations,
amoeba_ftol [1.0d-5] a floating point number representing the accuracy.

Example:

A function `f` of three variables, defined in terms of the Bessel function `bessj0`, is minimized using `amoeba`.

```
-> func(x,y,z) := 0.6 - bessj0((x-0.5)^2 + (y-0.6)^2 + (z-0.7)^2);
-> p : matrix([0,0,0],
   [1,0,0],
   [0,1,0],
   [0,0,1]);
-> y : [func(0,0,0),func(1,0,0),func(0,1,0),func(0,0,1)]?
[-0.1196220, -0.1196220, -0.2075238, -0.2812008]
-> amoeba(p,y,func)?
  | 0.4513889  0.6391782  0.6929977 |
  |                                 |
  | 0.4490741  0.6221065  0.6672454 |
[|                                  |, [-0.3999961, -0.3999957, -0.3999931,
  | 0.4469522  0.6484857  0.6904417 |
  |                                 |
  | 0.4351852  0.6114005  0.6860532 |
-0.3999949], 10]
-> func(0.5,0.6,0.7)?
-0.4000000
```

powell

Description:

This function finds a minimum of a function of many variables using Powell's method.

powell(p, xi, func) ⟶ [p, xi, iter, fret]

Where:

p is a real list representing on input an initial guess and on output the best point found,
xi is a real matrix representing on input an initial set of directions and on output the current direction set,
func is the name of a Senac function of many variables,
iter is the number of iterations which have been performed,
fret is a floating point number representing the value of $func$ at the final value of p.

Pre-variables:

powell_ftol [1.0d-5] a floating point number representing the required accuracy,
powell_itmax [200] the maximum number of allowed iterations.

Example:

A function `f`, defined in terms of the Bessel function `bessj0` and having a minimum at (1,2,3), is minimized using `powell`.

```
-> func(x,y,z) := 1/2 - bessj0((x-1)^2 + (y-2)^2 + (z-3)^2);
-> p : [3/2, 3/2, 5/2];
-> xi : matrix([1,0,0],
    [0,1,0],
    [0,0,1]);
-> powell(p,xi,func)?
                                       | 1.0000000  0.0000000  0.0000000 |
                                       |                                 |
[[1.0000000, 2.0000093, 2.9999429], | 0.0000000  1.0000000  0.0000000 |, 2,
                                       |                                 |
                                       | 0.0000000  0.0000000  1.0000000 |
-0.5000000]
-> func(1,2,3)?
-0.5000000
```

frprmn

Description:

This function finds a minum of a function of many variables using the conjugate gradient method.

frprmn(p, func, dfunc) $\longrightarrow$ **[p, fret, iter]**

Where:

p is a real list which on input represents an initial point and on output the position of the minumum,
func is the name of a Senac function of many variables,
dfunc is a list of the names of Senac functions representing the gradient of $func$,
fret is a floating point number representing the minumum value of the function,
iter is the number of iterations.

Pre-variables:

frprmn_ftol [1.0d-5] a floating point number representing the convergence tolerance on the function value,
frprmn_eps [1.0d-10] a small floating point number to allow for the special case where the function value at the minimum is zero,
frprmn_itmax 200] the maximum allowed number of iterations.

Example:

```
-> Take sample function: #
-> func(x,y,z) := 1.0 - bessj0(x-0.5)*bessj0(y-0.5)*bessj0(z-0.5);
-> which has min. at (0.5, 0.5, 0.5) #
-> dfuncx(x,y,z) := bessj1(x-0.5)*bessj0(y-0.5)*bessj0(z-0.5);
-> dfuncy(x,y,z) := bessj0(x-0.5)*bessj1(y-0.5)*bessj0(z-0.5);
-> dfuncz(x,y,z) := bessj0(x-0.5)*bessj0(y-0.5)*bessj1(z-0.5);
-> p : [1.0, 0.0, 0.0];
-> frprmn(p,func,[dfuncx,dfuncy,dfuncz])?
[[0.499999984, 0.500000016, 0.500000016], -0.000000008, 2]
-> p : [2.0*cos(pi/4), 2.0*sin(pi/4), 1.0];
-> frprmn(p,func,[dfuncx,dfuncy,dfuncz])?
[[0.500000000, 0.500000000, 0.500000000], -0.000000008, 4]
```

dfpmin

Description:

This function finds a minumum of a a Senac function of many variables using a variable metric method.

dfpmin(p, func, dfunc) ⟶ [p, fret, iter]

Where:

p is a real list which on input represents an initial point and on output the position of the minumum,
func is the name of a Senac function of many variables,
dfunc is a list of the names of Senac functions representing the gradient of $func$,
fret is a floating point number representing the minumum value of the function,
iter is the number of iterations.

Pre-variables:

dfpmin_ftol [1.0d-5] a floating point number representing the convergence tolerance on the function value,
dfpmin_eps [1.0d-10] a small floating point number to allow for the special case where the function value at the minimum is zero,
dfpmin_itmax 200] the maximum allowed number of iterations.

Example:

```
-> Take sample function: #
-> func(x,y,z) := 1.0 - bessj0(x-0.5)*bessj0(y-0.5)*bessj0(z-0.5);
-> which has min. at (0.5, 0.5, 0.5) #
-> dfuncx(x,y,z) := bessj1(x-0.5)*bessj0(y-0.5)*bessj0(z-0.5);
-> dfuncy(x,y,z) := bessj0(x-0.5)*bessj1(y-0.5)*bessj0(z-0.5);
-> dfuncz(x,y,z) := bessj0(x-0.5)*bessj0(y-0.5)*bessj1(z-0.5);
-> p : [1.0, 0.0, 0.0];
-> frprmn(p,func,[dfuncx,dfuncy,dfuncz])?
[[0.500000009, 0.499999991, 0.499999991], -0.000000008, 3]
-> pi:float(%pi);
-> p : [2.0*cos(pi/4), 2.0*sin(pi/4), 1.0];
-> dfpmin(p,func,[dfuncx,dfuncy,dfuncz])?
[[0.500000000, 0.500000000, 0.500000000], -0.000000008, 4]
-> p : [2.0*cos(3*pi/8), 2.0*sin(3*pi/8), 1.0];
-> dfpmin(p,func,[dfuncx,dfuncy,dfuncz])?
[[0.499999991, 0.500000001, 0.499999997], -0.000000008, 6]
```

simplex

Description:

This function implements the simplex algorithm for the minimization of a linear function of many variables over a domain described by equality, upper bound and lower bound linear constraints.

simplex(objective, ub_constraints, lb_constraints, eq_constraints) $\longrightarrow$ [min, b, icase]

Where:

objective is a real list of length n representing the linear objective function with the i'th element being the coefficient of the i'th variable,
ub_constraints is an $m1$ by $n+1$ matrix representing the upper bound constraints where the first column is the upper bounds and the remaining colums the coefficients of the variables,
lb_constraints is an $m2$ by $n+1$ matrix representing the lower bound constraints where the first column is the lower bounds and the remaining colums the coefficients of the variables,
eq_constraints is an $m3$ by $n+1$ matrix representing the equality constraints where the first column is the equality values and the remaining colums the coefficients of the variables,
min if $icase = 0$ is a floating point number representing the solution value,
b if $icase = 0$ is a floating point list representing the values of the variables at the solution,
icase is an integer having the value 0 if a solution is obtained, 1 if the objective is unbounded and -1 if no feasible point can be computed.

Pre-variables:

simplex_nmax 100] an integer upper bound for the number of variables,
simplex_eps [1.0d-8] a floating point number represnting accuracy.

Example:

A linear programming problem having upper bound, lower bound and equality constraints, is solved using `simplex`.

```
-> Take sample function: #
-> objective : [1.0, 1.0, 3.0, -0.5];
-> and sample upper-bound constraints: #
-> ub : matrix([740.0, 1.0, 0.0, 2.0,  0.0],
    [  0.0, 0.0, 2.0, 0.0, -7.0]);
-> lower-bound constraints: #
-> lb : matrix([0.5, 0.0, 1.0, -1.0, 2.0]);
-> equality constraints: #
-> eq : matrix([9.0, 1.0, 1.0, 1.0, 1.0]);
-> simplex(objective,ub,lb,eq)?
[17.0250000, [0.0000000, 3.3250000, 4.7250000, 0.9500000], [1, 6, 8, 7],
[5, 2, 4, 3], 0]
```

anneal

Description:

This function solves the travelling salesman problem by the method of simulated annealing (it should be generalized to solve a wider class of problems).

anneal(x, y, iorder) ⟶ iorder

Where:

x is a real list of length n representing the x-coordinates of the cities,
y is a real list representing the y-coordinates of the cities,
iorder is an integer list which on input is some permutation of the numbers 1 through n and on output represents the best alternative path that can be found.

Example:

This example uses `anneal` to find a path though 8 points which is an approximation to the path of minimum length. The points are numbered one through 8 starting with (0,0). If the user is trying this example set the Senac global variable `verbose` to false to check the progress of this time consuming algorithm.

```
-> x : [0,1,2,3,0,1,2,3];
-> y : [0,0,0,0,1,1,1,1];
-> iorder : listn(1,8);
-> anneal(x,y,iorder)?
[4, 3, 8, 7, 1, 2, 5, 6]
```
