Math 480 - May 12, 2008 - 1-dimensional root finding in Sage

Remark related to last time. If we get rid of (most) loop overhead and variable name lookup by using the Cython compiler and a C loop, then RDF is over 10-15 times faster than RIF, etc. See below.

Thus if you code things up tightly it is reasonable to expect over a 10 times slowdown when using interval arithmetic. Deal with it.

```
%cython

def foo(int n, a):
   cdef int i
   for i from 0 <= i < n:
       b = a*a
```

__Users_wa..._code_sage617_spyx.c __Users_wa...age617_spyx.pyx.html

```
time foo(10^7, RDF(pi))
```

```
CPU time: 0.45 s,  Wall time: 0.45 s
```

```
time foo(10^6, RDF(pi))
```

```
CPU time: 0.05 s,  Wall time: 0.05 s
```

```
time foo(10^6, float(pi))
```

```
CPU time: 0.04 s,  Wall time: 0.04 s
```

```
time foo(10^6, CDF(pi,pi))
```

```
CPU time: 0.17 s,  Wall time: 0.18 s
```

```
time foo(10^6, complex(pi,pi))
```

```
CPU time: 0.06 s,  Wall time: 0.06 s
```

```
time foo(10^6, RIF(pi))
```

```
CPU time: 0.73 s,  Wall time: 0.73 s
```

```
time foo(10^6, RR(pi))
```

```
CPU time: 0.39 s,  Wall time: 0.39 s
```

```
time foo(10^6, RealField(1000)(pi))
```

```
CPU time: 1.83 s,  Wall time: 1.84 s
```

```
time foo(10^6, RealIntervalField(300)(pi))
```

```
CPU time: 1.15 s,  Wall time: 1.16 s
```

```
time foo(10^6, RQDF(pi))
```

```
CPU time: 0.48 s,  Wall time: 0.49 s
```

1-Dimension Root Finding in Sage

Numerical Root Finding: Bisections, Newton's Method, Polynomials

Fast Float

Sage has a really nice **fast float** function that given a symbolic expression returns a very fast callable object. This is really useful to know about when implementing root finding or if you call any functions from scipy. At present, _fast_float_ can easily be 1000 times faster than not using it! (Which means we probably need to start automatically implicitly using it!) Anyway, here are some examples.

```
var('x')
f = cos(x)^2 - x^3 + x - 5
parent(f)
```

```
Symbolic Ring
```

```
a = float(e); a
```

```
2.7182818284590451
```

```
f(a)
```

```
-21.535996363355856
```

```
type(f(a))
```

```
    <class 'sage.calculus.calculus.SymbolicConstant'>
timeit('float(f(a))')
    625 loops, best of 3: 412 Âµs per loop
g = f._fast_float_(x)

g
    <sage.ext.fast_eval.FastDoubleFunc object at 0x27882a78>
g(a)
    -21.535996363355856
timeit('g(a)')
    625 loops, best of 3: 510 ns per loop
g.op_list()

# Robert Bradshaw makes some interesting remarks about what
_fast_float_ does and how it works.


# Works with n variables as well

var('x,y'); f = x^3 + y^2 - cos(x-y)^3
    (x, y)
a
    2.7182818284590451
time timeit('f(a,a)')
    5 loops, best of 3: 3.03 s per loop
    CPU time: 6.17 s,  Wall time: 61.80 s
g = f._fast_float_(x,y)

time timeit('g(a,a)')
    625 loops, best of 3: 589 ns per loop
    CPU time: 0.00 s,  Wall time: 0.00 s
```

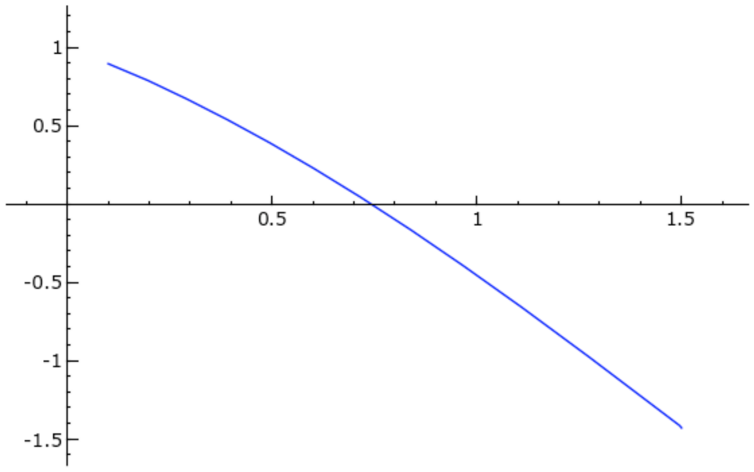
Wait this is about a million times faster... !

```
3.03/(589*10^(-9))
    5.14431239388795e6
```

Basics of Root Finding

Suppose $f(x)$ is a continuous real-valued function on an interval $[a, b]$ and that $f(a)f(b) < 0$, i.e., there is a sign change. Then by calculus f has a zero on this interval.

```
f = cos(x)-x
show(plot(f, 0.1,1.5),xmin=0)
```



The main goal today is to understand various ways of numerically finding a point $c \in [a, b]$ such that $f(c)$ is very close to 0.

Bisection

Perhaps the simplest method is *bisection*. Given $\epsilon > 0$, the following algorithm find a value $c \in [a, b]$ such that $|f(c)| < \epsilon$

1. If $|f((a+b)/2)| < \epsilon$ output $c = (a+b)/2$.
2. Chop the interval $[a, b]$ in two parts $[a, c] \cup [c, b]$
3. If f has a sign change on $[a, c]$ replace $[a, b]$ by $[a, c]$. Otherwise, replace $[a, b]$ by $[c, b]$. Then go to step 2.

At every step in the algorithm there is a point c in the interval so that $f(c) = 0$. Since the width of the interval halves each time and each interval contains a zero of f, the sequence of intervals will converge to some root of f. Since f is continuous, after a finite number of steps we will find a c such that $|f(c)| < \epsilon$

```
def bisect_method(f, a, b, eps):
   try:
       f = f._fast_float_(f.variables()[0])
   except AttributeError:
       pass
   intervals = [(a,b)]
   two = float(2); eps = float(eps)
   while True:
       c = (a+b)/two
       fa = f(a); fb = f(b); fc = f(c)
       if abs(fc) < eps: return c, intervals
       if fa*fc < 0:
           a, b = a, c
       elif fc*fb < 0:
           a, b = c, b
       else:
           raise ValueError, "f must have a sign change in the
interval (%s,%s)"%(a,b)
       intervals.append((a,b))

html("<h1>Double Precision Root Finding Using Bisection</h1>")
@interact
def _(f = cos(x) - x, a = float(0), b = float(1), eps=(-3,(-16..-
1))):
    eps = 10^eps
    print "eps = %s"%float(eps)
    try:
        time c, intervals = bisect_method(f, a, b, eps)
    except ValueError:
        print "f must have opposite sign at the endpoints of the
interval"
```

```
        show(plot(f, a, b, color='red'), xmin=a, xmax=b)
    else:
        print "root =", c
        print "f(c) = %r"%f(c)
        print "iterations =", len(intervals)
        P = plot(f, a, b, color='red')
        h = (P.ymax() - P.ymin())/ (1.5*len(intervals))
        L = sum(line([(c,h*i), (d,h*i)]) for i, (c,d) in 
enumerate(intervals) )
        L += sum(line([(c,h*i-h/4), (c,h*i+h/4)]) for i, (c,d) in 
enumerate(intervals) )
        L += sum(line([(d,h*i-h/4), (d,h*i+h/4)]) for i, (c,d) in 
enumerate(intervals) )
        show(P + L, xmin=a, xmax=b)
```

Double Precision Root Finding Using Bisection

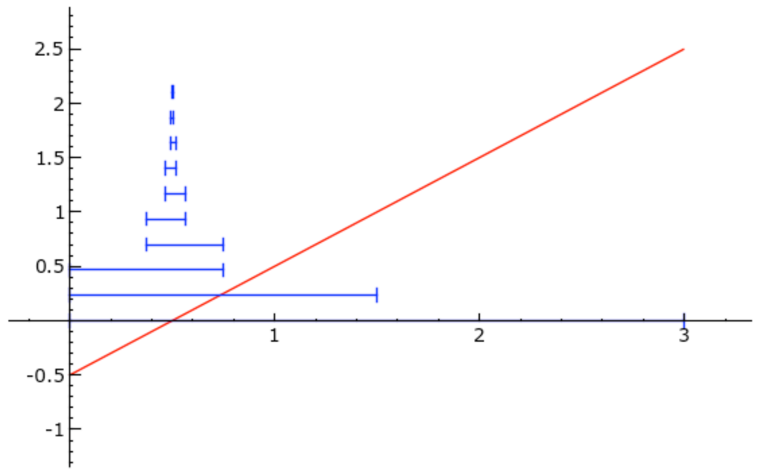
f cos(x) - x

a 0.0

b 1.0

eps

```
eps = 0.001
Time: CPU 0.02 s, Wall: 1.02 s
root = 0.5009765625
f(c) = 0.000976562500000000
iterations = 10
```



```
import scipy.optimize
```

```
help(scipy.optimize)
```

```
Help on package scipy.optimize in scipy: NAME scipy.optimize FILE
/Users/was/build/build/sage-3.0.1/local/lib/python2.5/site-
packages/scipy/optimize/__init__.py DESCRIPTION Optimization Tools
================== A collection of general-purpose optimization routines.
fmin -- Nelder-Mead Simplex algorithm (uses only function calls) fmin_powell
-- Powell's (modified) level set method (uses only function calls) fmin_cg -
- Non-linear (Polak-Ribiere) conjugate gradient algorithm (can use function
and gradient). fmin_bfgs -- Quasi-Newton method (Broydon-Fletcher-Goldfarb-
Shanno); (can use function and gradient) fmin_ncg -- Line-search Newton
Conjugate Gradient (can use function, gradient and Hessian). leastsq --
Minimize the sum of squares of M equations in N unknowns given a starting
estimate. Constrained Optimizers (multivariate) fmin_l_bfgs_b -- Zhu, Byrd,
and Nocedal's L-BFGS-B constrained optimizer (if you use this please quote
their papers -- see help) fmin_tnc -- Truncated Newton Code originally
written by Stephen Nash and adapted to C by Jean-Sebastien Roy. fmin_cobyla
-- Constrained Optimization BY Linear Approximation Global Optimizers anneal
-- Simulated Annealing brute -- Brute force searching optimizer Scalar
function minimizers fminbound -- Bounded minimization of a scalar function.
brent -- 1-D function minimization using Brent method. golden -- 1-D
function minimization using Golden Section method bracket -- Bracket a
minimum (given two starting points) Also a collection of general-purpose
root-finding routines. fsolve -- Non-linear multi-variable equation solver.
Scalar function solvers brentq -- quadratic interpolation Brent method
```

```
brenth -- Brent method (modified by Harris with hyperbolic extrapolation)
ridder -- Ridder's method bisect -- Bisection method newton -- Secant method
or Newton's method fixed_point -- Single-variable fixed-point solver. A
collection of general-purpose nonlinear multidimensional solvers. broyden1 -
- Broyden's first method - is a quasi-Newton-Raphson method for updating an
approximate Jacobian and then inverting it broyden2 -- Broyden's second
method - the same as broyden1, but updates the inverse Jacobian directly
broyden3 -- Broyden's second method - the same as broyden2, but instead of
directly computing the inverse Jacobian, it remembers how to construct it
using vectors, and when computing inv(J)*F, it uses those vectors to compute
this product, thus avoding the expensive NxN matrix multiplication.
broyden_generalized -- Generalized Broyden's method, the same as broyden2,
but instead of approximating the full NxN Jacobian, it construct it at every
iteration in a way that avoids the NxN matrix multiplication. This is not as
precise as broyden3. anderson -- extended Anderson method, the same as the
broyden_generalized, but added w_0^2*I to before taking inversion to improve
the stability anderson2 -- the Anderson method, the same as anderson, but
formulated differently Utility Functions line_search -- Return a step that
satisfies the strong Wolfe conditions. check_grad -- Check the supplied
derivative using finite difference techniques. PACKAGE CONTENTS _cobyla
_lbfgsb _minpack _zeros anneal cobyla info lbfgsb linesearch minpack
minpack2 moduleTNC nonlin optimize setup tnc zeros FUNCTIONS anderson(F,
xin, iter=10, alpha=0.10000000000000001, M=5, w0=0.01, verbose=False)
Extended Anderson method. Computes an approximation to the inverse Jacobian
from the last M interations. Avoids NxN matrix multiplication, it only has
MxM matrix multiplication and inversion. M=0 .... linear mixing M=1 ....
Anderson mixing with 2 iterations M=2 .... Anderson mixing with 3 iterations
etc. optimal is M=5 anderson2(F, xin, iter=10, alpha=0.10000000000000001,
M=5, w0=0.01, verbose=False) Anderson method. M=0 .... linear mixing M=1
.... Anderson mixing with 2 iterations M=2 .... Anderson mixing with 3
iterations etc. optimal is M=5 anneal(func, x0, args=(), schedule='fast',
full_output=0, T0=None, Tf=9.9999999999999998e-13, maxeval=None,
maxaccept=None, maxiter=400, boltzmann=1.0, learn_rate=0.5,
feps=9.9999999999999995e-07, quench=1.0, m=1.0, n=1.0, lower=-100,
upper=100, dwell=50) Minimize a function using simulated annealing. Schedule
is a schedule class implementing the annealing schedule. Available ones are
'fast', 'cauchy', 'boltzmann' Inputs: func -- Function to be optimized x0 --
Parameters to be optimized over args -- Extra parameters to function
schedule -- Annealing schedule to use (a class) full_output -- Return
optional outputs T0 -- Initial Temperature (estimated as 1.2 times the
largest cost-function deviation over random points in the range) Tf -- Final
goal temperature maxeval -- Maximum function evaluations maxaccept --
Maximum changes to accept maxiter -- Maximum cooling iterations learn_rate -
- scale constant for adjusting guesses boltzmann -- Boltzmann constant in
acceptance test (increase for less stringent test at each temperature). feps
-- Stopping relative error tolerance for the function value in last four
coolings. quench, m, n -- Parameters to alter fast_sa schedule lower, upper
-- lower and upper bounds on x0 (scalar or array). dwell -- The number of
times to search the space at each temperature. Outputs: (xmin, {Jmin, T,
feval, iters, accept,} retval) xmin -- Point giving smallest value found
retval -- Flag indicating stopping condition: 0 : Cooled to global optimum 1
: Cooled to final temperature 2 : Maximum function evaluations 3 : Maximum
cooling iterations reached 4 : Maximum accepted query locations reached Jmin
-- Minimum value of function found T -- final temperature feval -- Number of
function evaluations iters -- Number of cooling iterations accept -- Number
of tests accepted. See also: fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg
-- multivariate local optimizers leastsq -- nonlinear least squares
minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained multivariate
optimizers anneal, brute -- global optimizers fminbound, brent, golden,
bracket -- local scalar minimizers fsolve -- n-dimenstional root-finding
brentq, brenth, ridder, bisect, newton -- one-dimensional root-finding
fixed_point -- scalar fixed-point finder approx_fprime(xk, f, epsilon,
*args) bisect(f, a, b, args=(), xtol=9.9999999999999998e-13,
rtol=4.4408920985e-16, maxiter=100, full_output=False, disp=False) Find root
of f in [a,b] Basic bisection routine to find a zero of the function f
between the arguments a and b. f(a) and f(b) can not have the same signs.
Slow but sure. f : Python function returning a number. a : Number, one end
of the bracketing interval. b : Number, the other end of the bracketing
interval. xtol : Number, the routine converges when a root is known to lie
within xtol of the value return. Should be >= 0. The routine modifies this
to take into account the relative precision of doubles. maxiter : Number, if
convergence is not achieved in maxiter iterations, and error is raised. Must
be >= 0. args : tuple containing extra arguments for the function f. f is
called by apply(f,(x)+args). If full_output is False, the root is returned.
If full_output is True, the return value is (x, r), where x is the root, and
r is a RootResults object containing information about the convergence. In
particular, r.converged is True if the the routine converged. See also:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder bisection(func, a, b, args=(), xtol=1e-10, maxiter=400)
Bisection root-finding method. Given a function and an interval with func(a)
* func(b) < 0, find the root between a and b. See also: fmin, fmin_powell,
fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq --
nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla --
constrained multivariate optimizers anneal, brute -- global optimizers
fminbound, brent, golden, bracket -- local scalar minimizers fsolve -- n-
```

```
dimenstional root-finding brentq, brenth, ridder, bisect, newton -- one-
dimensional root-finding fixed_point -- scalar fixed-point finder
bracket(func, xa=0.0, xb=1.0, args=(), grow_limit=110.0, maxiter=1000) Given
a function and distinct initial points, search in the downhill direction (as
defined by the initital points) and return new points xa, xb, xc that
bracket the minimum of the function: f(xa) > f(xb) < f(xc). It doesn't
always mean that obtained solution will satisfy xa<=x<=xb :Parameters: func
-- objective func xa, xb : number bracketing interval args -- additional
arguments (if present) grow_limit : number max grow limit maxiter : number
max iterations number :Returns: xa, xb, xc, fa, fb, fc, funcalls xa, xb, xc
: number bracket fa, fb, fc : number objective function values in bracket
funcalls : number number of function evaluations brent(func, args=(),
brack=None, tol=1.48e-08, full_output=0, maxiter=500) Given a function of
one-variable and a possible bracketing interval, return the minimum of the
function isolated to a fractional precision of tol. :Parameters: func -
objective func args - additional arguments (if present) brack - triple
(a,b,c) where (a< func(a),func(c). If bracket is two numbers (a,c) then they
are assumed to be a starting interval for a downhill bracket search (see
bracket); it doesn't always mean that obtained solution will satisfy
a<=x<=c. full_output : number 0 - return only x (default) 1 - return all
output args (xmin, fval, iter, funcalls) :Returns: xmin : ndarray optim
point fval : number optim value iter : number number of iterations funcalls
: number number of objective function evaluations :SeeAlso: fmin,
fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers
leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla -- constrained multivariate optimizers anneal, brute -- global
optimizers fminbound, brent, golden, bracket -- local scalar minimizers
fsolve -- n-dimenstional root-finding brentq, brenth, ridder, bisect, newton
-- one-dimensional root-finding fixed_point -- scalar fixed-point finder
Notes ---------------------------- Uses inverse parabolic interpolation when
possible to speed up convergence of golden section method. brenth(f, a, b,
args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985e-16, maxiter=100,
full_output=False, disp=False) Find root of f in [a,b] A variation on the
classic Brent routine to find a zero of the function f between the arguments
a and b that uses hyperbolic extrapolation instead of inverse quadratic
extrapolation. There was a paper back in the 1980's ... f(a) and f(b) can
not have the same signs. Generally on a par with the brent routine, but not
as heavily tested. It is a safe version of the secant method that uses
hyperbolic extrapolation. The version here is by Chuck Harris. f : Python
function returning a number. a : Number, one end of the bracketing interval.
b : Number, the other end of the bracketing interval. xtol : Number, the
routine converges when a root is known to lie within xtol of the value
return. Should be >= 0. The routine modifies this to take into account the
relative precision of doubles. maxiter : Number, if convergence is not
achieved in maxiter iterations, and error is raised. Must be >= 0. args :
tuple containing extra arguments for the function f. f is called by
apply(f,(x)+args). If full_output is False, the root is returned. If
full_output is True, the return value is (x, r), where x is the root, and r
is a RootResults object containing information about the convergence. In
particular, r.converged is True if the the routine converged. See also:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder brentq(f, a, b, args=(), xtol=9.9999999999999998e-13,
rtol=4.4408920985e-16, maxiter=100, full_output=False, disp=False) Find root
of f in [a,b] The classic Brent routine to find a zero of the function f
between the arguments a and b. f(a) and f(b) can not have the same signs.
Generally the best of the routines here. It is a safe version of the secant
method that uses inverse quadratic extrapolation. The version here is a
slight modification that uses a different formula in the extrapolation step.
A description may be found in Numerical Recipes, but the code here is
probably easier to understand. f : Python function returning a number. a :
Number, one end of the bracketing interval. b : Number, the other end of the
bracketing interval. xtol : Number, the routine converges when a root is
known to lie within xtol of the value return. Should be >= 0. The routine
modifies this to take into account the relative precision of doubles.
maxiter : Number, if convergence is not achieved in maxiter iterations, and
error is raised. Must be >= 0. args : tuple containing extra arguments for
the function f. f is called by apply(f,(x)+args). If full_output is False,
the root is returned. If full_output is True, the return value is (x, r),
where x is the root, and r is a RootResults object containing information
about the convergence. In particular, r.converged is True if the the routine
converged. See also: fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg --
multivariate local optimizers leastsq -- nonlinear least squares minimizer
fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained multivariate optimizers
anneal, brute -- global optimizers fminbound, brent, golden, bracket --
local scalar minimizers fsolve -- n-dimenstional root-finding brentq,
brenth, ridder, bisect, newton -- one-dimensional root-finding fixed_point -
- scalar fixed-point finder broyden1(F, xin, iter=10,
alpha=0.10000000000000001, verbose=False) Broyden's first method. Updates
Jacobian and computes inv(J) by a matrix inversion at every iteration. It's
very slow. The best norm |F(x)|=0.005 achieved in ~45 iterations.
broyden2(F, xin, iter=10, alpha=0.40000000000000002, verbose=False)
Broyden's second method. Updates inverse Jacobian by an optimal formula.
There is NxN matrix multiplication in every iteration. The best norm
|F(x)|=0.003 achieved in ~20 iterations. Recommended. broyden3(F, xin,
iter=10, alpha=0.40000000000000002, verbose=False) Broyden's second method.
```

```
Updates inverse Jacobian by an optimal formula. The NxN matrix
multiplication is avoided. The best norm |F(x)|=0.003 achieved in ~20
iterations. Recommended. broyden_generalized(F, xin, iter=10,
alpha=0.10000000000000001, M=5, verbose=False) Generalized Broyden's method.
Computes an approximation to the inverse Jacobian from the last M
interations. Avoids NxN matrix multiplication, it only has MxM matrix
multiplication and inversion. M=0 .... linear mixing M=1 .... Anderson
mixing with 2 iterations M=2 .... Anderson mixing with 3 iterations etc.
optimal is M=5 brute(func, ranges, args=(), Ns=20, full_output=0, finish=)
Minimize a function over a given range by brute force. :Parameters: func --
Function to be optimized ranges : tuple Tuple where each element is a tuple
of parameters or a slice object to be handed to numpy.mgrid args -- Extra
arguments to function. Ns : number Default number of samples if not given
full_output : number Nonzero to return evaluation grid. :Returns: (x0, fval,
{grid, Jout}) x0 : ndarray Value of arguments giving minimum over the grird
fval : number Function value at minimum grid : tuple tuple with same length
as x0 representing the evaluation grid Jout : ndarray -- Function values
over grid: Jout = func(*grid) :SeeAlso: fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq -- nonlinear
least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained
multivariate optimizers anneal, brute -- global optimizers fminbound, brent,
golden, bracket -- local scalar minimizers fsolve -- n-dimenstional root-
finding brentq, brenth, ridder, bisect, newton -- one-dimensional root-
finding fixed_point -- scalar fixed-point finder Notes ------------------
Find the minimum of a function evaluated on a grid given by the tuple
ranges. check_grad(func, grad, x0, *args) fixed_point(func, x0, args=(),
xtol=1e-10, maxiter=500) Given a function of one variable and a starting
point, find a fixed-point of the function: i.e. where func(x)=x. See also:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None,
maxfun=None, full_output=0, disp=1, retall=0, callback=None) Minimize a
function using the downhill simplex algorithm. :Parameters: func : the
Python function or method to be minimized. x0 : ndarray - the initial guess.
args : extra arguments for func. callback : an optional user-supplied
function to call after each iteration. It is called as callback(xk), where
xk is the current parameter vector. :Returns: (xopt, {fopt, iter, funcalls,
warnflag}) xopt : ndarray minimizer of function fopt : number value of
function at minimum: fopt = func(xopt) iter : number number of iterations
funcalls : number number of function calls warnflag : number Integer warning
flag: 1 : 'Maximum number of function evaluations.' 2 : 'Maximum number of
iterations.' allvecs : Python list a list of solutions at each iteration
:OtherParameters: xtol : number acceptable relative error in xopt for
convergence. ftol : number acceptable relative error in func(xopt) for
convergence. maxiter : number the maximum number of iterations to perform.
maxfun : number the maximum number of function evaluations. full_output :
number non-zero if fval and warnflag outputs are desired. disp : number non-
zero to print convergence messages. retall : number non-zero to return list
of solutions at each iteration :SeeAlso: fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq -- nonlinear
least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained
multivariate optimizers anneal, brute -- global optimizers fminbound, brent,
golden, bracket -- local scalar minimizers fsolve -- n-dimenstional root-
finding brentq, brenth, ridder, bisect, newton -- one-dimensional root-
finding fixed_point -- scalar fixed-point finder Notes ----------- Uses a
Nelder-Mead simplex algorithm to find the minimum of function of one or more
variables. fmin_bfgs(f, x0, fprime=None, args=(), gtol=1.0000000000000001e-
05, norm=inf, epsilon=1.49011611938e-08, maxiter=None, full_output=0,
disp=1, retall=0, callback=None) Minimize a function using the BFGS
algorithm. :Parameters: f : the Python function or method to be minimized.
x0 : ndarray the initial guess for the minimizer. fprime : a function to
compute the gradient of f. args : extra arguments to f and fprime. gtol :
number gradient norm must be less than gtol before succesful termination
norm : number order of norm (Inf is max, -Inf is min) epsilon : number if
fprime is approximated use this value for the step size (can be scalar or
vector) callback : an optional user-supplied function to call after each
iteration. It is called as callback(xk), where xk is the current parameter
vector. :Returns: (xopt, {fopt, gopt, Hopt, func_calls, grad_calls,
warnflag}, ) xopt : ndarray the minimizer of f. fopt : number the value of
f(xopt). gopt : ndarray the value of f'(xopt). (Should be near 0) Bopt :
ndarray the value of 1/f''(xopt). (inverse hessian matrix) func_calls :
number the number of function_calls. grad_calls : number the number of
gradient calls. warnflag : integer 1 : 'Maximum number of iterations
exceeded.' 2 : 'Gradient and/or function calls not changing' allvecs : a
list of all iterates (only returned if retall==1) :OtherParameters: maxiter
: number the maximum number of iterations. full_output : number if non-zero
then return fopt, func_calls, grad_calls, and warnflag in addition to xopt.
disp : number print convergence message if non-zero. retall : number return
a list of results at each iteration if non-zero :SeeAlso: fmin, fmin_powell,
fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq --
nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla --
constrained multivariate optimizers anneal, brute -- global optimizers
fminbound, brent, golden, bracket -- local scalar minimizers fsolve -- n-
dimenstional root-finding brentq, brenth, ridder, bisect, newton -- one-
dimensional root-finding fixed_point -- scalar fixed-point finder Notes ----
------------------------------ Optimize the function, f, whose gradient is
```

```
given by fprime using the quasi-Newton method of Broyden, Fletcher,
Goldfarb, and Shanno (BFGS) See Wright, and Nocedal 'Numerical
Optimization', 1999, pg. 198. fmin_cg(f, x0, fprime=None, args=(),
gtol=1.0000000000000001e-05, norm=inf, epsilon=1.49011611938e-08,
maxiter=None, full_output=0, disp=1, retall=0, callback=None) Minimize a
function with nonlinear conjugate gradient algorithm. :Parameters: f -- the
Python function or method to be minimized. x0 : ndarray -- the initial guess
for the minimizer. fprime -- a function to compute the gradient of f. args -
- extra arguments to f and fprime. gtol : number stop when norm of gradient
is less than gtol norm : number order of vector norm to use epsilon :number
if fprime is approximated use this value for the step size (can be scalar or
vector) callback -- an optional user-supplied function to call after each
iteration. It is called as callback(xk), where xk is the current parameter
vector. :Returns: (xopt, {fopt, func_calls, grad_calls, warnflag},
{allvecs}) xopt : ndarray the minimizer of f. fopt :number the value of
f(xopt). func_calls : number the number of function_calls. grad_calls :
number the number of gradient calls. warnflag :number an integer warning
flag: 1 : 'Maximum number of iterations exceeded.' 2 : 'Gradient and/or
function calls not changing' allvecs : ndarray if retall then this vector of
the iterates is returned :OtherParameters: maxiter :number the maximum
number of iterations. full_output : number if non-zero then return fopt,
func_calls, grad_calls, and warnflag in addition to xopt. disp : number
print convergence message if non-zero. retall : number return a list of
results at each iteration if True :SeeAlso: fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq -- nonlinear
least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained
multivariate optimizers anneal, brute -- global optimizers fminbound, brent,
golden, bracket -- local scalar minimizers fsolve -- n-dimenstional root-
finding brentq, brenth, ridder, bisect, newton -- one-dimensional root-
finding fixed_point -- scalar fixed-point finder Notes --------------------
------------------------ Optimize the function, f, whose gradient is given
by fprime using the nonlinear conjugate gradient algorithm of Polak and
Ribiere See Wright, and Nocedal 'Numerical Optimization', 1999, pg. 120-122.
fmin_cobyla(func, x0, cons, args=(), consargs=None, rhobeg=1.0,
rhoend=0.0001, iprint=1, maxfun=1000) Minimize a function using the
Constrained Optimization BY Linear Approximation (COBYLA) method Arguments:
func -- function to minimize. Called as func(x, *args) x0 -- initial guess
to minimum cons -- a sequence of functions that all must be >=0 (a single
function if only 1 constraint) args -- extra arguments to pass to function
consargs -- extra arguments to pass to constraints (default of None means
use same extra arguments as those passed to func). Use () for no extra
arguments. rhobeg -- reasonable initial changes to the variables rhoend --
final accuracy in the optimization (not precisely guaranteed) iprint --
controls the frequency of output: 0 (no output),1,2,3 maxfun -- maximum
number of function evaluations. Returns: x -- the minimum See also: fmin,
fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers
leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla -- constrained multivariate optimizers anneal, brute -- global
optimizers fminbound, brent, golden, bracket -- local scalar minimizers
fsolve -- n-dimenstional root-finding brentq, brenth, ridder, bisect, newton
-- one-dimensional root-finding fixed_point -- scalar fixed-point finder
fmin_l_bfgs_b(func, x0, fprime=None, args=(), approx_grad=0, bounds=None,
m=10, factr=10000000.0, pgtol=1.0000000000000001e-05, epsilon=1e-08,
iprint=-1, maxfun=15000) Minimize a function func using the L-BFGS-B
algorithm. Arguments: func -- function to minimize. Called as func(x, *args)
x0 -- initial guess to minimum fprime -- gradient of func. If None, then
func returns the function value and the gradient ( f, g = func(x, *args) ),
unless approx_grad is True then func returns only f. Called as fprime(x,
*args) args -- arguments to pass to function approx_grad -- if true,
approximate the gradient numerically and func returns only function value.
bounds -- a list of (min, max) pairs for each element in x, defining the
bounds on that parameter. Use None for one of min or max when there is no
bound in that direction m -- the maximum number of variable metric
corrections used to define the limited memory matrix. (the limited memory
BFGS method does not store the full hessian but uses this many terms in an
approximation to it). factr -- The iteration stops when (f^k -
f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr*epsmch where epsmch is the machine
precision, which is automatically generated by the code. Typical values for
factr: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely
high accuracy. pgtol -- The iteration will stop when max{|proj g_i | i = 1,
..., n} <= pgtol where pg_i is the ith component of the projected gradient.
epsilon -- step size used when approx_grad is true, for numerically
calculating the gradient iprint -- controls the frequency of output. <0
means no output. maxfun -- maximum number of function evaluations. Returns:
x, f, d = fmin_lbfgs_b(func, x0, ...) x -- position of the minimum f --
value of func at the minimum d -- dictionary of information from routine
d['warnflag'] is 0 if converged, 1 if too many function evaluations, 2 if
stopped for another reason, given in d['task'] d['grad'] is the gradient at
the minimum (should be 0 ish) d['funcalls'] is the number of function calls
made. License of L-BFGS-B (Fortran code) ==================================
The version included here (in fortran code) is 2.1 (released in 1997). It
was written by Ciyou Zhu, Richard Byrd, and Jorge Nocedal . It carries the
following condition for use: This software is freely available, but we
expect that all publications describing work using this software , or all
commercial products using it, quote at least one of the references given
below. References * R. H. Byrd, P. Lu and J. Nocedal. A Limited Memory
Algorithm for Bound Constrained Optimization, (1995), SIAM Journal on
Scientific and Statistical Computing , 16, 5, pp. 1190-1208. * C. Zhu, R. H.
Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for
large scale bound constrained optimization (1997), ACM Transactions on
```

```
Mathematical Software, Vol 23, Num. 4, pp. 550 - 560. See also:
scikits.openopt, which offers a unified syntax to call this and other
solvers fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate
local optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder fmin_ncg(f, x0, fprime, fhess_p=None, fhess=None, args=(),
avextol=1.0000000000000001e-05, epsilon=1.49011611938e-08, maxiter=None,
full_output=0, disp=1, retall=0, callback=None) Minimize the function f
using the Newton-CG method. :Parameters: f -- the Python function or method
to be minimized. x0 : ndarray -- the initial guess for the minimizer. fprime
-- a function to compute the gradient of f: fprime(x, *args) fhess_p -- a
function to compute the Hessian of f times an arbitrary vector: fhess_p (x,
p, *args) fhess -- a function to compute the Hessian matrix of f. args --
extra arguments for f, fprime, fhess_p, and fhess (the same set of extra
arguments is supplied to all of these functions). epsilon : number if fhess
is approximated use this value for the step size (can be scalar or vector)
callback -- an optional user-supplied function to call after each iteration.
It is called as callback(xk), where xk is the current parameter vector.
:Returns: (xopt, {fopt, fcalls, gcalls, hcalls, warnflag},{allvecs}) xopt :
ndarray the minimizer of f fopt : number the value of the function at xopt:
fopt = f(xopt) fcalls : number the number of function calls gcalls : number
the number of gradient calls hcalls : number the number of hessian calls.
warnflag : number algorithm warnings: 1 : 'Maximum number of iterations
exceeded.' allvecs : Python list a list of all tried iterates
:OtherParameters: avextol : number Convergence is assumed when the average
relative error in the minimizer falls below this amount. maxiter : number
Maximum number of iterations to allow. full_output : number If non-zero
return the optional outputs. disp : number If non-zero print convergence
message. retall : bool return a list of results at each iteration if True
:SeeAlso: fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate
local optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder Notes --------------------------------------------- Only one of
fhess_p or fhess need be given. If fhess is provided, then fhess_p will be
ignored. If neither fhess nor fhess_p is provided, then the hessian product
will be approximated using finite differences on fprime. fhess_p must
compute the hessian times an arbitrary vector. If it is not given, finite-
differences on fprime are used to compute it. See Wright, and Nocedal
'Numerical Optimization', 1999, pg. 140. fmin_powell(func, x0, args=(),
xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None, full_output=0, disp=1,
retall=0, callback=None, direc=None) Minimize a function using modified
Powell's method. :Parameters: func -- the Python function or method to be
minimized. x0 : ndarray the initial guess. args -- extra arguments for func
callback -- an optional user-supplied function to call after each iteration.
It is called as callback(xk), where xk is the current parameter vector direc
-- initial direction set :Returns: (xopt, {fopt, xi, direc, iter, funcalls,
warnflag}, {allvecs}) xopt : ndarray minimizer of function fopt : number
value of function at minimum: fopt = func(xopt) direc -- current direction
set iter : number number of iterations funcalls : number number of function
calls warnflag : number Integer warning flag: 1 : 'Maximum number of
function evaluations.' 2 : 'Maximum number of iterations.' allvecs : Python
list a list of solutions at each iteration :OtherParameters: xtol : number
line-search error tolerance. ftol : number acceptable relative error in
func(xopt) for convergence. maxiter : number the maximum number of
iterations to perform. maxfun : number the maximum number of function
evaluations. full_output : number non-zero if fval and warnflag outputs are
desired. disp : number non-zero to print convergence messages. retall :
number non-zero to return a list of the solution at each iteration :SeeAlso:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder Notes ----------------------- Uses a modification of Powell's
method to find the minimum of a function of N variables fmin_tnc(func, x0,
fprime=None, args=(), approx_grad=0, bounds=None, epsilon=1e-08, scale=None,
offset=None, messages=15, maxCGit=-1, maxfun=None, eta=-1, stepmx=0,
accuracy=0, fmin=0, ftol=-1, xtol=-1, pgtol=-1, rescale=-1) Minimize a
function with variables subject to bounds, using gradient information.
:Parameters: func : callable func(x, *args) Function to minimize. Should
return f and g, where f is the value of the function and g its gradient (a
list of floats). If the function returns None, the minimization is aborted.
x0 : list of floats Initial estimate of minimum. fprime : callable fprime(x,
*args) Gradient of func. If None, then func must return the function value
and the gradient (f,g = func(x, *args)). args : tuple Arguments to pass to
function. approx_grad : bool If true, approximate the gradient numerically.
bounds : list (min, max) pairs for each element in x, defining the bounds on
that parameter. Use None or +/-inf for one of min or max when there is no
bound in that direction. scale : list of floats Scaling factors to apply to
each variable. If None, the factors are up-low for interval bounded
variables and 1+|x] fo the others. Defaults to None offset : float Value to
substract from each variable. If None, the offsets are (up+low)/2 for
interval bounded variables and x for the others. messages : Bit mask used to
```

```
select messages display during minimization values defined in the MSGS dict.
Defaults to MGS_ALL. maxCGit : int Maximum number of hessian*vector
evaluations per main iteration. If maxCGit == 0, the direction chosen is -
gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -
1. maxfun : int Maximum number of function evaluation. if None, maxfun is
set to max(100, 10*len(x0)). Defaults to None. eta : float Severity of the
line search. if < 0 or > 1, set to 0.25. Defaults to -1. stepmx : float
Maximum step for the line search. May be increased during call. If too
small, it will be set to 10.0. Defaults to 0. accuracy : float Relative
precision for finite difference calculations. If <= machine_precision, set
to sqrt(machine_precision). Defaults to 0. fmin : float Minimum function
value estimate. Defaults to 0. ftol : float Precision goal for the value of
f in the stoping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -
1. xtol : float Precision goal for the value of x in the stopping criterion
(after applying x scaling factors). If xtol < 0.0, xtol is set to
sqrt(machine_precision). Defaults to -1. pgtol : float Precision goal for
the value of the projected gradient in the stopping criterion (after
applying x scaling factors). If pgtol < 0.0, pgtol is set to 1e-2 *
sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.
rescale : float Scaling factor (in log10) used to trigger f value rescaling.
If 0, rescale at each iteration. If a large value, never rescale. If < 0,
rescale is set to 1.3. :Returns: x : list of floats The solution. nfeval :
int The number of function evaluations. rc : Return code as defined in the
RCSTRINGS dict. :SeeAlso: - scikits.openopt, which offers a unified syntax
to call this and other solvers - fmin, fmin_powell, fmin_cg, fmin_bfgs,
fmin_ncg : multivariate local optimizers - leastsq : nonlinear least squares
minimizer - fmin_l_bfgs_b, fmin_tnc, fmin_cobyla : constrained multivariate
optimizers - anneal, brute : global optimizers - fminbound, brent, golden,
bracket : local scalar minimizers - fsolve : n-dimenstional root-finding -
brentq, brenth, ridder, bisect, newton : one-dimensional root-finding -
fixed_point : scalar fixed-point finder fminbound(func, x1, x2, args=(),
xtol=1.0000000000000001e-05, maxfun=500, full_output=0, disp=1) Bounded
minimization for scalar functions. :Parameters: func -- the function to be
minimized (must accept scalar input and return scalar output). x1, x2 :
ndarray the optimization bounds. args -- extra arguments to pass to
function. xtol : number the convergence tolerance. maxfun : number maximum
function evaluations. full_output : number Non-zero to return optional
outputs. disp : number Non-zero to print messages. 0 : no message printing.
1 : non-convergence notification messages only. 2 : print a message on
convergence too. 3 : print iteration results. :Returns: (xopt, {fval, ierr,
numfunc}) xopt : ndarray The minimizer of the function over the interval.
fval : number The function value at the minimum point. ierr : number An
error flag (0 if converged, 1 if maximum number of function calls reached).
numfunc : number The number of function calls. :SeeAlso: fmin, fmin_powell,
fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers leastsq --
nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla --
constrained multivariate optimizers anneal, brute -- global optimizers
fminbound, brent, golden, bracket -- local scalar minimizers fsolve -- n-
dimenstional root-finding brentq, brenth, ridder, bisect, newton -- one-
dimensional root-finding fixed_point -- scalar fixed-point finder Notes ----
-------------------------------------------------- Finds a local minimizer
of the scalar function func in the interval x1 < xopt < x2 using Brent's
method. (See brent for auto-bracketing). fsolve(func, x0, args=(),
fprime=None, full_output=0, col_deriv=0, xtol=1.49012e-08, maxfev=0,
band=None, epsfcn=0.0, factor=100, diag=None, warning=True) Find the roots
of a function. Description: Return the roots of the (non-linear) equations
defined by func(x)=0 given a starting estimate. Inputs: func -- A Python
function or method which takes at least one (possibly vector) argument. x0 -
- The starting estimate for the roots of func(x)=0. args -- Any extra
arguments to func are placed in this tuple. fprime -- A function or method
to compute the Jacobian of func with derivatives across the rows. If this is
None, the Jacobian will be estimated. full_output -- non-zero to return the
optional outputs. col_deriv -- non-zero to specify that the Jacobian
function computes derivatives down the columns (faster, because there is no
transpose operation). warning -- True to print a warning message when the
call is unsuccessful; False to suppress the warning message. Outputs: (x,
{infodict, ier, mesg}) x -- the solution (or the result of the last
iteration for an unsuccessful call. infodict -- a dictionary of optional
outputs with the keys: 'nfev' : the number of function calls 'njev' : the
number of jacobian calls 'fvec' : the function evaluated at the output
'fjac' : the orthogonal matrix, q, produced by the QR factorization of the
final approximate Jacobian matrix, stored column wise. 'r' : upper
triangular matrix produced by QR factorization of same matrix. 'qtf' : the
vector (transpose(q) * fvec). ier -- an integer flag. If it is equal to 1
the solution was found. If it is not equal to 1, the solution was not found
and the following message gives more information. mesg -- a string message
giving information about the cause of failure. Extended Inputs: xtol -- The
calculation will terminate if the relative error between two consecutive
iterates is at most xtol. maxfev -- The maximum number of calls to the
function. If zero, then 100*(N+1) is the maximum where N is the number of
elements in x0. band -- If set to a two-sequence containing the number of
sub- and superdiagonals within the band of the Jacobi matrix, the Jacobi
matrix is considered banded (only for fprime=None). epsfcn -- A suitable
step length for the forward-difference approximation of the Jacobian (for
fprime=None). If epsfcn is less than the machine precision, it is assumed
that the relative errors in the functions are of the order of the machine
precision. factor -- A parameter determining the initial step bound (factor
* || diag * x||). Should be in interval (0.1,100). diag -- A sequency of N
positive entries that serve as a scale factors for the variables. Remarks:
"fsolve" is a wrapper around MINPACK's hybrd and hybrj algorithms. See also:
```

```
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimenstional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder golden(func, args=(), brack=None, tol=1.49011611938e-08,
full_output=0) Given a function of one-variable and a possible bracketing
interval, return the minimum of the function isolated to a fractional
precision of tol. :Parameters: func - objective function args - additional
arguments (if present) brack : a triple (a,b,c) where (a< func(a),func(c).
If bracket is two numbers (a, c) then they are assumed to be a starting
interval for a downhill bracket search (see bracket); it doesn't always mean
that obtained solution will satisfy a<=x<=c tol : number x tolerance stop
criterion full_output : number 0 for false 1 for true :SeeAlso: fmin,
fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local optimizers
leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla -- constrained multivariate optimizers anneal, brute -- global
optimizers fminbound, brent, golden, bracket -- local scalar minimizers
fsolve -- n-dimenstional root-finding brentq, brenth, ridder, bisect, newton
-- one-dimensional root-finding fixed_point -- scalar fixed-point finder
Notes -------------------------------------- Uses analog of bisection
method to decrease the bracketed interval. leastsq(func, x0, args=(),
Dfun=None, full_output=0, col_deriv=0, ftol=1.49012e-08, xtol=1.49012e-08,
gtol=0.0, maxfev=0, epsfcn=0.0, factor=100, diag=None, warning=True)
Minimize the sum of squares of a set of equations. Description: Return the
point which minimizes the sum of squares of M (non-linear) equations in N
unknowns given a starting estimate, x0, using a modification of the
Levenberg-Marquardt algorithm. x = arg min(sum(func(y)**2,axis=0)) y Inputs:
func -- A Python function or method which takes at least one (possibly
length N vector) argument and returns M floating point numbers. x0 -- The
starting estimate for the minimization. args -- Any extra arguments to func
are placed in this tuple. Dfun -- A function or method to compute the
Jacobian of func with derivatives across the rows. If this is None, the
Jacobian will be estimated. full_output -- non-zero to return all optional
outputs. col_deriv -- non-zero to specify that the Jacobian function
computes derivatives down the columns (faster, because there is no transpose
operation). warning -- True to print a warning message when the call is
unsuccessful; False to suppress the warning message. Outputs: (x, {cov_x,
infodict, mesg}, ier) x -- the solution (or the result of the last iteration
for an unsuccessful call. cov_x -- uses the fjac and ipvt optional outputs
to construct an estimate of the covariance matrix of the solution. None if a
singular matrix encountered (indicates infinite covariance in some
direction). infodict -- a dictionary of optional outputs with the keys:
'nfev' : the number of function calls 'njev' : the number of jacobian calls
'fvec' : the function evaluated at the output 'fjac' : A permutation of the
R matrix of a QR factorization of the final approximate Jacobian matrix,
stored column wise. Together with ipvt, the covariance of the estimate can
be approximated. 'ipvt' : an integer array of length N which defines a
permutation matrix, p, such that fjac*p = q*r, where r is upper triangular
with diagonal elements of nonincreasing magnitude. Column j of p is column
ipvt(j) of the identity matrix. 'qtf' : the vector (transpose(q) * fvec).
mesg -- a string message giving information about the cause of failure. ier
-- an integer flag. If it is equal to 1 the solution was found. If it is not
equal to 1, the solution was not found and the following message gives more
information. Extended Inputs: ftol -- Relative error desired in the sum of
squares. xtol -- Relative error desired in the approximate solution. gtol --
Orthogonality desired between the function vector and the columns of the
Jacobian. maxfev -- The maximum number of calls to the function. If zero,
then 100*(N+1) is the maximum where N is the number of elements in x0.
epsfcn -- A suitable step length for the forward-difference approximation of
the Jacobian (for Dfun=None). If epsfcn is less than the machine precision,
it is assumed that the relative errors in the functions are of the order of
the machine precision. factor -- A parameter determining the initial step
bound (factor * || diag * x||). Should be in interval (0.1,100). diag -- A
sequency of N positive entries that serve as a scale factors for the
variables. Remarks: "leastsq" is a wrapper around MINPACK's lmdif and lmder
algorithms. See also: fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg --
multivariate local optimizers leastsq -- nonlinear least squares minimizer
fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained multivariate optimizers
anneal, brute -- global optimizers fminbound, brent, golden, bracket --
local scalar minimizers fsolve -- n-dimenstional root-finding brentq,
brenth, ridder, bisect, newton -- one-dimensional root-finding fixed_point -
- scalar fixed-point finder line_search(f, myfprime, xk, pk, gfk, old_fval,
old_old_fval, args=(), c1=0.0001, c2=0.90000000000000002, amax=50) Find
alpha that satisfies strong Wolfe conditions. :Parameters: f : objective
function myfprime : objective function gradient (can be None) xk : ndarray -
- start point pk : ndarray -- search direction gfk : ndarray -- gradient
value for x=xk args : additional arguments for user functions c1 : number --
parameter for Armijo condition rule c2 : number - parameter for curvature
condition rule :Returns: alpha0 : number -- required alpha (x_new = x0 +
alpha * pk) fc : number of function evaluations gc : number of gradient
evaluations Notes -------------------------------- Uses the line search
algorithm to enforce strong Wolfe conditions Wright and Nocedal, 'Numerical
Optimization', 1999, pg. 59-60 For the zoom phase it uses an algorithm by
newton(func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50) Given a
function of a single variable and a starting point, find a nearby zero using
Newton-Raphson. fprime is the derivative of the function. If not given, the
Secant method is used. See also: fmin, fmin_powell, fmin_cg, fmin_bfgs,
fmin_ncg -- multivariate local optimizers leastsq -- nonlinear least squares
```

```
minimizer fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained multivariate
optimizers anneal, brute -- global optimizers fminbound, brent, golden,
bracket -- local scalar minimizers fsolve -- n-dimensional root-finding
brentq, brenth, ridder, bisect, newton -- one-dimensional root-finding
fixed_point -- scalar fixed-point finder ridder(f, a, b, args=(),
xtol=9.9999999999999998e-13, rtol=4.4408920985e-16, maxiter=100,
full_output=False, disp=False) Find root of f in [a,b] Ridder routine to
find a zero of the function f between the arguments a and b. f(a) and f(b)
can not have the same signs. Faster than bisection, but not generaly as fast
as the brent rountines. A description may be found in a recent edition of
Numerical Recipes. The routine here is modified a bit to be more careful of
tolerance. f : Python function returning a number. a : Number, one end of
the bracketing interval. b : Number, the other end of the bracketing
interval. xtol : Number, the routine converges when a root is known to lie
within xtol of the value return. Should be >= 0. The routine modifies this
to take into account the relative precision of doubles. maxiter : Number, if
convergence is not achieved in maxiter iterations, and error is raised. Must
be >= 0. args : tuple containing extra arguments for the function f. f is
called by apply(f,(x)+args). If full_output is False, the root is returned.
If full_output is True, the return value is (x, r), where x is the root, and
r is a RootResults object containing information about the convergence. In
particular, r.converged is True if the the routine converged. See also:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimensional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder rosen(x) rosen_der(x) rosen_hess(x) rosen_hess_prod(x, p) DATA
__all__ = ['anderson', 'anderson2', 'anneal', 'approx_fprime', 'bisect...
```

```
help(scipy.optimize.bisect)
```

```
Help on function bisect in module scipy.optimize.zeros: bisect(f, a, b,
args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985e-16, maxiter=100,
full_output=False, disp=False) Find root of f in [a,b] Basic bisection
routine to find a zero of the function f between the arguments a and b. f(a)
and f(b) can not have the same signs. Slow but sure. f : Python function
returning a number. a : Number, one end of the bracketing interval. b :
Number, the other end of the bracketing interval. xtol : Number, the routine
converges when a root is known to lie within xtol of the value return.
Should be >= 0. The routine modifies this to take into account the relative
precision of doubles. maxiter : Number, if convergence is not achieved in
maxiter iterations, and error is raised. Must be >= 0. args : tuple
containing extra arguments for the function f. f is called by
apply(f,(x)+args). If full_output is False, the root is returned. If
full_output is True, the return value is (x, r), where x is the root, and r
is a RootResults object containing information about the convergence. In
particular, r.converged is True if the the routine converged. See also:
fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg -- multivariate local
optimizers leastsq -- nonlinear least squares minimizer fmin_l_bfgs_b,
fmin_tnc, fmin_cobyla -- constrained multivariate optimizers anneal, brute -
- global optimizers fminbound, brent, golden, bracket -- local scalar
minimizers fsolve -- n-dimensional root-finding brentq, brenth, ridder,
bisect, newton -- one-dimensional root-finding fixed_point -- scalar fixed-
point finder
```

```
f = cos(x) - x
scipy.optimize.bisect??(f._fast_float_(x), 0, 1)
```

```
0.73908513321566716
```

```
bisect_method(f, 0, 1, 1e-15)
```

```
(0.73908513321516089, [(0, 1), (0.5, 1), (0.5, 0.75), (0.625, 0.75),
(0.6875, 0.75), (0.71875, 0.75), (0.734375, 0.75), (0.734375,
0.7421875), (0.73828125, 0.7421875), (0.73828125, 0.740234375),
(0.73828125, 0.7392578125), (0.73876953125, 0.7392578125),
(0.739013671875, 0.7392578125), (0.739013671875, 0.7391357421875),
(0.73907470703125, 0.7391357421875), (0.73907470703125,
0.739105224609375), (0.73907470703125, 0.7390899658203125),
(0.73908233642578125, 0.7390899658203125), (0.73908233642578125,
0.73908615112304688), (0.73908424377441406, 0.73908615112304688),
(0.73908424377441406, 0.73908519744873047), (0.73908472061157227,
0.73908519744873047), (0.73908495903015137, 0.73908519744873047),
(0.73908507823944092, 0.73908519744873047), (0.73908507823944092,
0.73908513784408569), (0.73908510804176331, 0.73908513784408569),
(0.7390851229429245, 0.73908513784408569), (0.7390851303935051,
0.73908513784408569), (0.7390851303935051, 0.73908513411879539),
(0.73908513225615025, 0.73908513411879539), (0.73908513318747282,
0.73908513411879539), (0.73908513318747282, 0.73908513365313411),
(0.73908513318747282, 0.73908513342030346), (0.73908513318747282,
0.73908513330388814), (0.73908513318747282, 0.73908513324568048),
(0.73908513318747282, 0.73908513321657665), (0.73908513320202474,
    0.73908513321657665), (0.73908513320930069, 0.73908513321657665),
    (0.73908513321293867, 0.73908513321657665), (0.73908513321475766,
    0.73908513321657665), (0.73908513321475766, 0.73908513321566716),
    (0.73908513321475766, 0.73908513321521241), (0.73908513321498504,
    0.73908513321521241), (0.73908513321509872, 0.73908513321521241),
    (0.73908513321515557, 0.73908513321521241), (0.73908513321515557,
    0.73908513321518399), (0.73908513321515557, 0.73908513321516978),
    (0.73908513321515557, 0.73908513321516267), (0.73908513321515912,
    0.73908513321516267)])
```

```
f = cos(x) - x
g = f._fast_float_(x)
timeit('scipy.optimize.bisect(g, 0.0r, 1.0r)')
```

```
    625 loops, best of 3: 14.9 Âµs per loop
```

```
f = cos(x) - x
g = f._fast_float_(x)
timeit('bisect_method(g, 0.0r, 1.0r, 0.00001r)')
```

```
    625 loops, best of 3: 520 Âµs per loop
```

Scipy's bisect is a lot faster than ours, of course. It's optimized code written in C. Let's look.

1. Extract the scipy spkg:

   ```
   tar jxvf scipy-20071020-0.6.p3.spkg
   ```

2. Track down the bisect code. This just takes guessing and knowing how to get around source code. I found the bisect code in

   ```
     scipy-20071020-0.6.p3/src/scipy/optimize/Zeros/bisect.c
   ```

   Here it is:

   ```
   /* Written by Charles Harris charles.harris@sdl.usu.edu */

   #include "zeros.h"

   double
   bisect(callback_type f, double xa, double xb, double xtol, double rtol, int iter, default_parameters *params)
   {
       int i;
       double dm,xm,fm,fa,fb,tol;

       tol = xtol + rtol*(fabs(xa) + fabs(xb));

       fa = (*f)(xa,params);
       fb = (*f)(xb,params);
       params->funcalls = 2;
       if (fa*fb > 0) {ERROR(params,SIGNERR,0.0);}
       if (fa == 0) return xa;
       if (fb == 0) return xb;
       dm = xb - xa;
       params->iterations = 0;
       for(i=0; iiterations++;
           dm *= .5;
           xm = xa + dm;
           fm = (*f)(xm,params);
           params->funcalls++;
           if (fm*fa >= 0) {
               xa = xm;
           }
           if (fm == 0 || fabs(dm) < tol)
               return xm;
       }
       ERROR(params,CONVERR,xa);
   }
   ```

```
%cython
cdef extern from "math.h":
   double abs(double)

def bisect_cython(f, double a, double b, double eps):
   cdef double two = 2, fa, fb, fc, c, dm, fabs
```

```
    cdef int iterations = 0
    fa = f(a); fb = f(b)
    while 1:
        iterations += 1
        c = (a+b)/two
        fc = f(c)
        fabs = -fc if fc < 0 else fc
        if fabs < eps: return c, iterations
        if fa*fc < 0:
            a, b = a, c
            fb = fc
        elif fc*fb < 0:
            a, b = c, b
            fa = fc
        else:
            raise ValueError, "f must have a sign change in the
interval (%s,%s)"%(a,b)
```

__Users_wa..._code_sage670_spyx.c   __Users_wa...age670_spyx.pyx.html

```
f = cos(x) - x
g = f._fast_float_(x)
print bisect_cython(g, 0.0r, 1.0r, 0.000001r)
print scipy.optimize.bisect(g, 0.0r, 1.0r, maxiter=20)
```

```
(0.73908519744873047, 20)
0.739084243774
```

```
print "scipy"
timeit('scipy.optimize.bisect(g, 0.0r, 1.0r, maxiter=20)')
print "sage/python"
timeit('bisect_method(g, 0.0r, 1.0r, 0.000001r)')
print "sage/cython"
timeit('bisect_cython(g, 0.0r, 1.0r, 0.000001r)')
```

```
scipy
625 loops, best of 3: 11.6 Âµs per loop
sage/python
625 loops, best of 3: 654 Âµs per loop
sage/cython
625 loops, best of 3: 4.83 Âµs per loop
```

```
# Wow, the Cython code above is faster than scipy...  This is
probably because
# the algorithm is slightly different... ?
```

Newton's Method

```
%hide
%html

Often $f$ is differentiable and its values can help us give an even
more efficient root
finding algorithm called <b>Newton's Method</b>.

<br><br>
The equation of the tangent line to the graph of $f(x)$ at the
point $(c,f(c))$
is
$$
  y = f(c) + f'(c)(x-c).
$$
This is because the above line goes through the point $(c,f(c))$
and is a line of slow $f'(c)$.

<br><br><b>Newton's method: </b>
```

```
Approximate $f(x)$ by the line above, find a root of that line,
then replace $c$
by that root.  Iterate.  In particular, by easy algebra, the root
of the linear function
is
$$c - \frac{f(c)}{f'(c)}.$$

<br><br> As an algorithm:
<ol>
<li> Choose a starting value $c$.
<li> Let $c = c - f(c)/f'(c)$.
<li> If $|f(c)| < \epsilon$ output $c$ and terminate.
 Otherwise, go to 2.
<ol>
```

Often f is differentiable and its values can help us give an even more efficient root finding algorithm called **Newton's Method**.

The equation of the tangent line to the graph of $f(x)$ at the point $(c, f(c))$ is

$$y = f(c) + f'(c)(x - c).$$

This is because the above line goes through the point $(c, f(c))$ and is a line of slow $f'(c)$.

Newton's method: Approximate $f(x)$ by the line above, find a root of that line, then replace $c$ by that root. Iterate. In particular, by easy algebra, the root of the linear function is

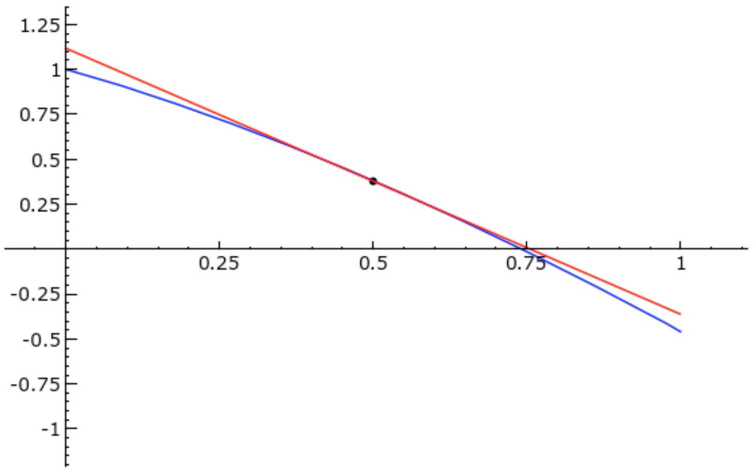
$$c - \frac{f(c)}{f'(c)}.$$

As an algorithm:

1. Choose a starting value $c$.
2. Let $c = c - f(c)/f'(c)$
3. If $|f(c)| <$ output $c$ and terminate. Otherwise, go to 2.

```
x = var('x')
@interact
def _(f = cos(x) - x, c = float(1/2), a=float(0), b=float(1)):
   show(plot(f,a,b) + point((c,f(c)),rgbcolor='black',pointsize=20)
+ \
          plot(f(c) + f.derivative()(c)*(x-c), a,b,color='red'),
a,b)
```

f	cos(x) - x
c	0.5
a	0.0
b	1.0

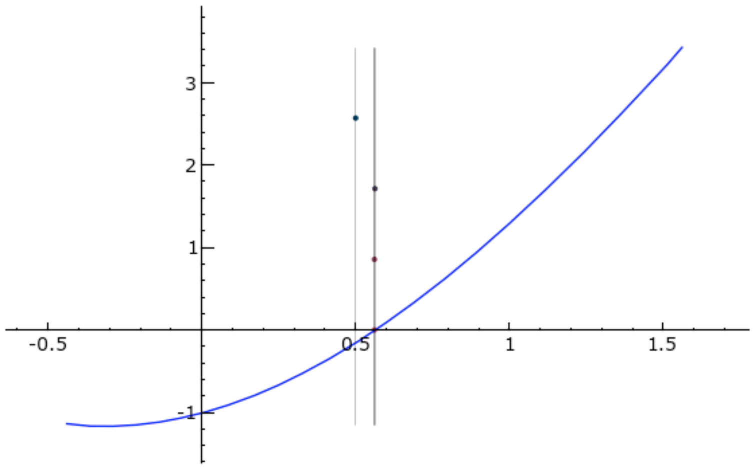


```
def newton_method(f, c, eps, maxiter=100):
   x = f.variables()[0]
   fprime = f.derivative(x)
   try:
       g = f._fast_float_(x)
       gprime = fprime._fast_float_(x)
   except AttributeError:
       g = f; gprime = fprime
   iterates = [c]
   for i in xrange(maxiter):
      fc = g(c)
      if abs(fc) < eps: return c, iterates
      c = c - fc/gprime(c)
      iterates.append(c)
   return c, iterates



html("<h1>Double Precision Root Finding Using Newton's
Method</h1>")

@interact
def _(f = x^2 - 2, c = float(0.5), eps=(-3,(-16..-1)),
interval=float(0.5)):
    eps = 10^(eps)
    print "eps = %s"%float(eps)
    time z, iterates = newton_method(f, c, eps)
    print "root =", z
    print "f(c) = %r"%f(z)
    n = len(iterates)
    print "iterations =", n
    html(iterates)
    P = plot(f, z-interval, z+interval, rgbcolor='blue')
    h = P.ymax(); j = P.ymin()
    L = sum(point((w,(n-1-float(i))/n*h),
rgbcolor=(float(i)/n,0.2,0.3), pointsize=10) + \
            line([(w,h),(w,j)],rgbcolor='black',thickness=0.2) for
i,w in enumerate(iterates))
    show(P + L, xmin=z-interval, xmax=z+interval)
```

Double Precision Root Finding Using Newton's Method

f	x^2 - 2
c	0.5
eps	
interval	0.5

```
eps = 1e-07
Time: CPU 0.10 s, Wall: 1.13 s
root = 0.560987729237
f(c) = 3.0186408928045694e-12
iterations = 4
[0.5, 0.56285808829213679, 0.5609893442488848, 0.56098772923711848]
```



```
float(sqrt(2))
```

```
1.4142135623730951
```

```
help(scipy.optimize.newton)
```

```
Help on function newton in module scipy.optimize.minpack: newton(func, x0,
fprime=None, args=(), tol=1.48e-08, maxiter=50) Given a function of a single
variable and a starting point, find a nearby zero using Newton-Raphson.
fprime is the derivative of the function. If not given, the Secant method is
used. See also: fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg --
multivariate local optimizers leastsq -- nonlinear least squares minimizer
fmin_l_bfgs_b, fmin_tnc, fmin_cobyla -- constrained multivariate optimizers
anneal, brute -- global optimizers fminbound, brent, golden, bracket --
local scalar minimizers fsolve -- n-dimensional root-finding brentq,
brenth, ridder, bisect, newton -- one-dimensional root-finding fixed_point -
- scalar fixed-point finder
```

```
x = var('x')
f = x^2 - 2
g = f._fast_float_(x)
gprime = f.derivative()._fast_float_(x)
scipy.optimize.newton??(g, 1, gprime)
```

```
1.4142135623730951
```

```
# Interestingly newton in scipy is written in PURE PYTHON, so
should be easy to beat using Cython.
scipy.optimize.newton??
```

```
%cython

def newton_cython(f, double c, fprime, double eps, int
maxiter=100):
   cdef double fc
   cdef int i
   for i from 0 <= i < maxiter:
      fc = f(c)
      absfc = -fc if fc < 0 else fc
      if absfc < eps:
          return c
      c = c - fc/fprime(c)
   return c
```

__Users_wa..._code_sage690_spyx.c __Users_wa...age690_spyx.pyx.html

```
x = var('x')
f = x^2 - cos(x) + sin(x^2)
g = f._fast_float_(x)
gprime = f.derivative()._fast_float_(x)

timeit('scipy.optimize.newton(g, 1.0r, gprime)')
timeit('newton_cython(g, 1.0r, gprime, 1e-13)')
```

```
625 loops, best of 3: 16.8 Âµs per loop
625 loops, best of 3: 17.7 Âµs per loop
```

```
newton_cython(g,1.0r,gprime,1e-13)
```

Easy wrappers

```
var('x')
f = cos(x)-x
```

```
f.find_root(0,1)
```

```
0.7390851332151559
```

Real Root Isolation

```
R.<x> = PolynomialRing(QQ)
```

```
f = (x-1)^3*(x-2/3)^2*(x+1939)
```

```
help(f.real_root_intervals)
```

```
f.real_root_intervals()
```