# TACC -- 3. Tutorial on Numerical Computation in Sage

## TACC -- 3. Tutorial on Numerical Computation in Sage

William Stein

University of Washington

This tutorial is mainly about using Sage for numerical computation (more like MATLAB).

Sage comes with several numerical libraries:

- Numpy: http://numpy.scipy.org/ and see especially NumPy for Matlab Users
- Scipy: http://www.scipy.org/
- CVXOPT: http://abel.ee.ucla.edu/cvxopt/
- GSL: http://www.gnu.org/software/gsl/
- R: http://www.r-project.org/

You can learn tons about using each of those systems from their web pages.
In this tutorial, we will instead focus on a few key nontrivial ideas needed to implement entirely new fast numerical code in Sage.

```
```

## 1-Dimension Root Finding in Sage

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

## Recall: 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(x=a)
```

```
-21.5359963633559
```

```
type(f(x=a))
```
```
    <type 'sage.symbolic.expression.Expression'>
```
```
timeit('float(f(x=a))')
```
```
    625 loops, best of 3: 141 µs per loop
```
```
g = fast_float(f, x)
```

```
g
```
```
    <sage.ext.interpreters.wrapper_rdf.Wrapper_rdf object at
    0x10c870a28>
```
```
g(a)
```
```
    -21.535996363355856
```
```
timeit('g(a)')
```
```
    625 loops, best of 3: 502 ns per loop
```
```
g.op_list()
```
```
    [('load_arg', 0), ('ipow', 3), 'neg', ('load_arg', 0), 'cos',
    ('ipow', 2), 'add', ('load_arg', 0), 'add', ('load_const', -5.0),
    'add', 'return']
```

It works with n variables as well.

```
f(x,y) = x^3 + y^2 - cos(x-y)^3
```

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

```
timeit('f(a,a)')
```
```
    625 loops, best of 3: 253 µs per loop
```
```
g = fast_float(f, x,y)
```

```
timeit('g(a,a)')
```
```
    625 loops, best of 3: 443 ns per loop
```

# The following example illustrates how you might teach (or implement) root finding using Sage:

## 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 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 = fast_float(f, f.variables()[0])
    except AttributeError:
        print "WARNING: Not using fast_float."
    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)
```

```
```

```
```

```
import scipy.optimize
```

```
scipy.optimize?
```

```
scipy.optimize.bisect?
```

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

```
0.73908513321566716
```

```
a=float(0); b=float(1)
timeit('scipy.optimize.bisect(f, a, b)')
```

```
125 loops, best of 3: 7.05 ms per loop
```

```
g = fast_float(f, x)
timeit('scipy.optimize.bisect(g, a, b)')
```

```
625 loops, best of 3: 16.8 µs per loop
```

The bisect we implemented above is slower:

```
bisect_method(f, a, b, 1e-8)[0]
```

```
0.73908513784408569
```

```
timeit('bisect_method(g, a, b, 1e-8)[0]')
```

```
625 loops, best of 3: 446 µs 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:

   ```
   wstein> ls spkg/standard/scipy*
   spkg/standard/scipy-0.7.p4.spkg
   spkg/standard/scipy_sandbox-20071020.p4.spkg
   wstein> tar jxvf spkg/standard/scipy-0.7.p4.spkg  # may have to get from http://sagemath.org/packages/standard/
   ```

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-0.7.p4/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;
   - 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);
    }
```

We can implement this directly in Sage using ***Cython*** (see http://cython.org -- in fact, browse that page for a while!).

```
%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)
```

**Problem:** Click on each of the .html link above to see the autogenerated code (double click on a line of Cython to see the corresponding C code).

```

```

```
f = cos(x) - x
g = fast_float(f, x)
print bisect_cython(g, 0.0, 1.0, 1e-16)
print scipy.optimize.bisect(g, 0.0, 1.0, maxiter=40)
```

```
(0.73908513321516067, 52)
0.739085133216
```

```

```

Note: Putting r after a number keeps the Sage preparser from turning it into a Sage arbitrary precision real number (instead of a standard Python float).

```
print "scipy"
timeit('scipy.optimize.bisect(g, 0.0r, 1.0r, maxiter=40)')
print "sage/python"
timeit('bisect_method(g, 0.0r, 1.0r, 1e-16r)')
print "sage/cython"
timeit('bisect_cython(g, 0.0r, 1.0r, 1e-16r)')
```

```
scipy
625 loops, best of 3: 17.5 µs per loop
sage/python
625 loops, best of 3: 883 µs per loop
sage/cython
625 loops, best of 3: 15.1 µs per loop
```

## 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 $|f(c)| < \epsilon$ 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)
```

```
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>")
var('x')
@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)
```

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

```
1.4142135623730951
```

```

```

```
scipy.optimize.newton?
```

```
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??
```

Write our own Newton method in Cython:

```
%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
```

Test them all

```
x = var('x')
f = (x^2 - cos(x) + sin(x^2))^3
g = f._fast_float_(x)
gprime = f.derivative()._fast_float_(x)

timeit('scipy.optimize.newton(g, 0.5r, gprime)')
timeit('newton_cython(g, 0.5r, gprime, 1e-13)')
```

```
625 loops, best of 3: 127 µs per loop
625 loops, best of 3: 37.9 µs per loop
```

```
newton_cython(g, 0.5r, gprime, 1e-13)
```

```
0.63810381041535857
```

```
plot(g, 0,1)
```

```

```

## Easy wrappers

Sage has code that uses Scipy behind the scenes. With ***your help*** it could have a lot more!

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

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

```
0.7390851332151559
```

**Problem**: Find a root of something else on another interval.

**Problem:** Type f.find_root?? to read the source code. The function just calls into another module. Type sage.numerical.optimize.find_root?? to read the source code of that, which does involve calling scipy.

## Bonus: Real Root Isolation

Sage is also a computer algebra system, so it also has subtle algorithms such as real root isolation.

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

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

```
f.real_root_intervals?
```

```
f.real_root_intervals()
```

```
[((-142243895477354596886616900600/733594097356134547791855501,
-142243895477354454756413816400/733594097356134547791855501), 1),
((245847389135172936379869572749/368771083702759445511619153400,
491694778270346145705171107347/737542167405518891023238306800), 2),
((209/256, 593/512), 3)]
```