Lecture 4

Rootfinding: Newton's Method in higher dimensions, secant method,fractals, Matlab - fzero

T. Gambill

Department of Computer Science
University of Illinois at Urbana-Champaign

6/16/2013

Newton's Method in higher dimensions

Given $f : \mathbb{R}^m \to \mathbb{R}^m$ then we can consider f as a vector of m functions.

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}$$

where $f_i : \mathbb{R}^m \to \mathbb{R}$. We can write the Taylor Series for each f_i as follows.

$$f_i(\mathbf{x_{k+1}}) = f_i(\mathbf{x_k}) + [\nabla f_i(\mathbf{x_k})]^T * (\mathbf{x_{k+1}} - \mathbf{x_k}) + \ldots$$

Combining these in a columnar vector gives,

$$\begin{bmatrix} f_1(\mathbf{x_{k+1}}) \\ f_2(\mathbf{x_{k+1}}) \\ \vdots \\ f_m(\mathbf{x_{k+1}}) \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{x_k}) \\ f_2(\mathbf{x_k}) \\ \vdots \\ f_m(\mathbf{x_k}) \end{bmatrix} + \mathbf{J} * (\mathbf{x_{k+1}} - \mathbf{x_k}) + \ldots$$

Newton's Method in higher dimensions

$$\begin{bmatrix} f_1(\mathbf{x_{k+1}}) \\ f_2(\mathbf{x_{k+1}}) \\ \vdots \\ f_m(\mathbf{x_{k+1}}) \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{x_k}) \\ f_2(\mathbf{x_k}) \\ \vdots \\ f_m(\mathbf{x_k}) \end{bmatrix} + \mathbf{J} * (\mathbf{x_{k+1}} - \mathbf{x_k}) + \ldots$$

The matrix **J** is called the Jacobian,

$$\mathbf{J}_{ij} = \frac{\partial f_i(\mathbf{x}_k)}{\partial x_j}.$$

In the case with one dimension, to obtain Newton's method we ignored higher order terms and set $f(x_{k+1}) = 0$ and then solved for x_{k+1}. We do the same for higher dimensions to obtain the formula,

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J}^{-1} * \mathbf{f}(\mathbf{x_k})$$

Although, we will later see why we should (and can) avoid computing the inverse of the Jacobian and instead solve the system of equations,

$$\mathbf{J} * (\mathbf{x_{k+1}} - \mathbf{x_k}) = -\mathbf{f}(\mathbf{x_k})$$

Newton's Method Example 1

Using the formula,

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J}^{-1} * \mathbf{f}(\mathbf{x_k})$$

find a root for the system of equations defined by,

$$\begin{aligned} f_1(\mathbf{x}) &= x_1 + 2x_2 - 2 &= 0 \\ f_2(\mathbf{x}) &= x_1^2 + 4x_2^2 - 4 &= 0 \end{aligned}$$

(The solution of this system is $[0, 1]^T$.) The Jacobian is given by,

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}$$

If we choose $[1, 1]^T$ as a starting guess then we generate the following values for each iteration as shown on the next slide.

Newton's Method Example 1

```python
import numpy as np
from scipy import optimize
import numpy.linalg

def newtonJ(f,x,tol):
    k = 1
    y = np.array([10.,10.])
    print('    k              x_k[0]           x_k[1]    ')
    while numpy.linalg.norm(y) > tol:
        y = f(x)
        delta_x = numpy.linalg.solve(J(x),-y)
        delta_x = delta_x.reshape(2,)
        x = x + delta_x
        print('%5d %22.20f %22.20f' % (k,x[0],x[1]))
        k = k + 1

def J(x):
    y = np.array([[1.,2.],[2.*x[0],8.*x[1]]])
    return y

def f(x):
    y = np.array([[1.*x[0]+2.*x[1]],[x[0]**2+ 4.*x[1]**2]])-np.array([[2.],[4.]])
    return y


if __name__ == "__main__":
    newtonJ(f, np.array([1.,1.]), 1.e-8)
```

```
k              x_k[0]                 x_k[1]
1 -0.50000000000000000000 1.25000000000000000000
2 -0.08333333333333331483 1.04166666666666674068
3 -0.00320512820512797170 1.00160256410256409687
4 -0.00000512001310694994 1.00000256000655363131
5 -0.00000000001310730548 1.00000000000655364651
6 -0.00000000000000001246 1.00000000000000000000
```

Newton's Method Example 2

Using the formula,

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J}^{-1} * \mathbf{f}(\mathbf{x_k})$$

find a root for the system of equations defined by,

$$\begin{aligned} f_1(\mathbf{x}) &= x_1 + 2x_2 - 2 &= 0 \\ f_2(\mathbf{x}) &= -2x_1 + x_2 - 4 &= 0 \end{aligned}$$

(The solution of this system is $[-1.2, 1.6]^T$.) The Jacobian is given by,

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}$$

Why?
If we choose $[1, 1]^T$ as a starting guess then we generate the following values for each iteration shown on the next slide.

Newton's Method Example 2

```python
import numpy as np
from scipy import optimize
import numpy.linalg

def newtonJ(f,x,tol):
    k = 1
    y = np.array([10.,10.])
    print('     k                 x_k[0]                x_k[1]      ')
    while numpy.linalg.norm(y) > tol:
        y = f2(x)
        delta_x = numpy.linalg.solve(J(x),-y)
        x = x + delta_x
        print('%5d %22.20f %22.20f' % (k,x[0],x[1]))
        k = k + 1

def J(x):
    y = np.array([[1.,2.],[-2,1.]])
    return y

def f2(x):
    y = np.dot(np.array([[1.,2.],[-2., 1.]]),x)-np.array([[2.],[4.]])
    return y


if __name__ == "__main__":
    newtonJ(f2, np.array([[1.],[1.]]), 1.e-8)
```

```
k              x_k[0]              x_k[1]
1 -1.20000000000000017764 1.60000000000000008882
2 -1.20000000000000017764 1.60000000000000008882
```

Fractals: What?

Definition

Fractal A mathematical pattern (geometric object) that is reproducible at **any** level of magnification or reduction.

Definition

Fractal A term used by Benoit Mandelbrot to refer to geometric objects with fractional dimensions rather than integer dimensions. Also used "fractal" to refer to shapes that are self-similar: they look the same at any zoom level.

Fractals: Application

Scientifically used to describe highly irregular objects

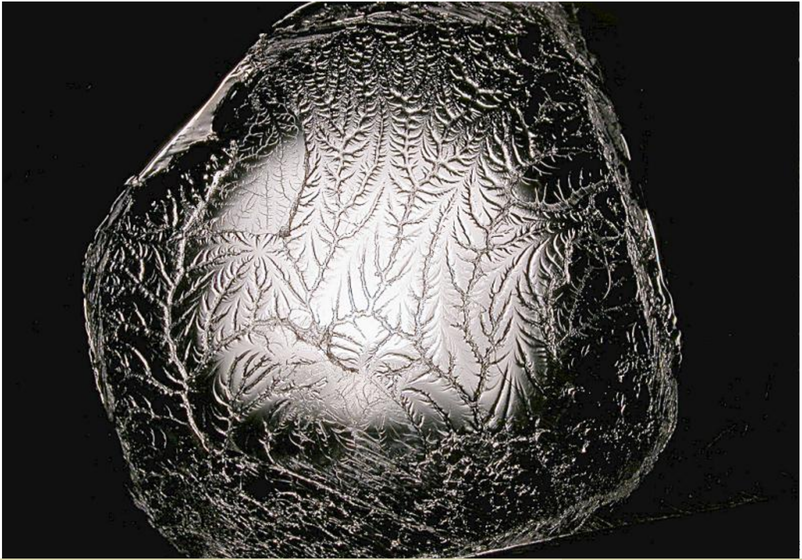
- fractal image compression
- Seismology
- Cosmology
- life sciences:
  - clouds and fluid turbulence
  - trees
  - coastlines

More interesting observations:

- New music/New art
- Video games/graphics
- Chaos theory
- the Butterfly effect: small changes produces large effects

Fractals: Air Pressure

Air channels between two glued pieces of acrylic



Fractals: high voltage dielectric breakdown

Lichtenberg: Branching discharges decrease to hairlike then to molecular

Fractals: Microwaving a CD

Heat vaporizes the aluminum leaving fractal metallic islands

Fractals: Romanesco Broccoli

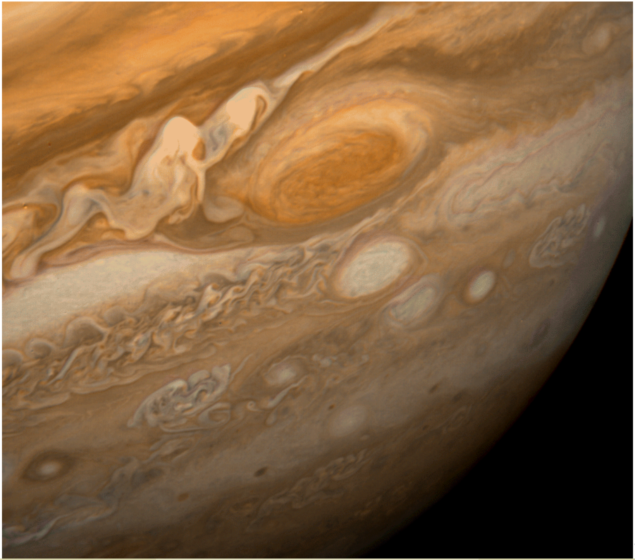
growth follows fractal pattern

Fractals: Trees

structure follows fractal pattern

Fractals: Jupiter

Atmosphere modeled with fractals



Fractals: Caves

Stalactite/Stalagmite formation

Fractals: Canyons

Erosion patter

Fractals: Clouds

visualization

Fractals: Ferns

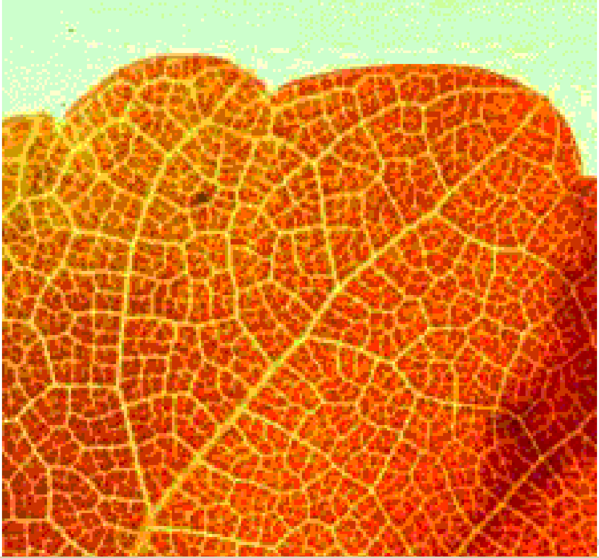
growth

Fractals: Big Trees

growth

Fractals: leaves

structure



Fractals: lightning

formation

Fractals: cauliflower

structure

Fractals: mountain

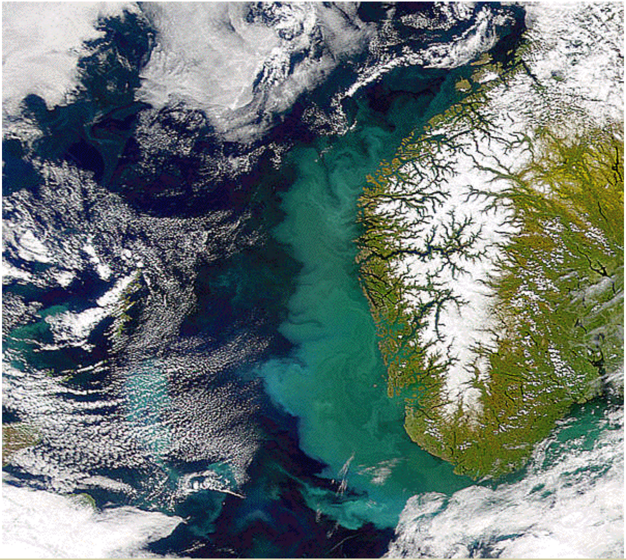
formation

Fractals: mountain

visualization

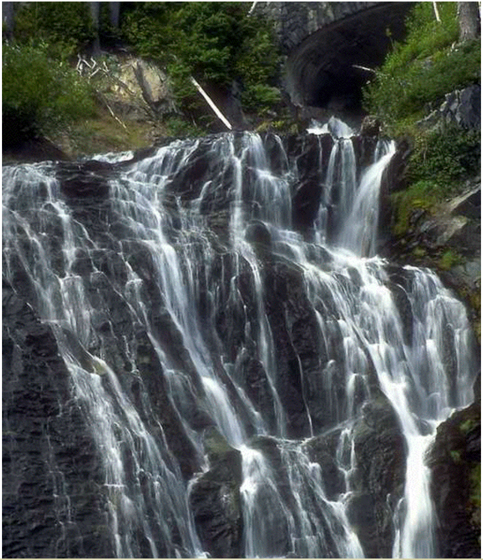
Fractals: Norwegian rivers

structure



Fractals: waterfalls

pattern



Fractals: coastlines

structure

Fractals: Math

Recall Complex Numbers: $z \in \mathbb{C}$ means

$$z = x + iy,$$

where $i = \sqrt{-1}$

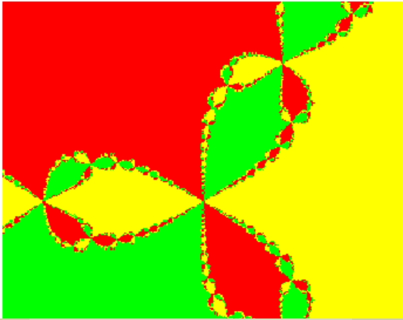
Things to notice:

- still think of the x-y plane, but now it's in $\mathbb{C}^1$ instead of $\mathbb{R}^2$
- $f(z) = z^2 + 1$ has two roots: $z_{1,2} = \pm i$
- $f(z) = z^3 + 1$ has three roots: $z_1 = -1$, $z_{2,3} = \frac{-1 \pm i\sqrt{3}}{2}$
- $f(z) = z^4 + 1$ has four roots: $z_{1,2} = \frac{\pm\sqrt{2} + i\sqrt{2}}{2}$, $z_{3,4} = \frac{\pm\sqrt{2} - i\sqrt{2}}{2}$

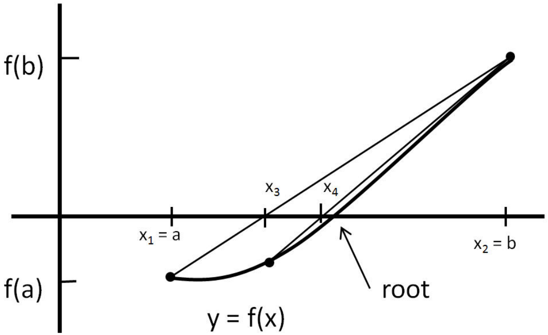
Fractals: Newton's Algorithm

The big idea:

- Take a complex function like $f(z) = z^3 + 1$
- Pick a bunch of initial guesses z_1 as the roots
- Run Newton's Method
- The initial guesses z_1 will each converge to one of $n = 3$ roots
- Color each guess in the plane depending on the root to which it converged



Secant Method



Given two guesses x_{k-1} and x_k, the next guess at the root is where the line through $f(x_{k-1})$ and $f(x_k)$ crosses the x axis.

Secant Method

Given

$$\begin{aligned} x_k &= \text{current guess at the root} \\ x_{k-1} &= \text{previous guess at the root} \end{aligned}$$

Approximate the first derivative with

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Substitute approximate $f'(x_k)$ into formula for Newton's method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

to get

$$x_{k+1} = x_k - f(x_k) \left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \right]$$

Secant Method

Two versions of this formula are (equivalent in exact math)

$$x_{k+1} = x_k - f(x_k)\left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right] \qquad (\star)$$

and

$$x_{k+1} = \frac{f(x_k)x_{k-1} - f(x_{k-1})x_k}{f(x_k) - f(x_{k-1})} \qquad (\star\star)$$

Equation ($\star$) is better since it is of the form $x_{k+1} = x_k - f(x_k)\Delta$. Even if Δ is inaccurate the change in the estimate of the root will be small at convergence because $f(x_k)$ will also be small.
Equation ($\star\star$) is susceptible to catastrophic cancellation:

- $f(x_k) \to f(x_{k-1})$ as convergence approaches, so cancellation error in denominator can be large.
- $|f(x)| \to 0$ as convergence approaches, so underflow is possible

Secant Algorithm

initialize: $x_1 = \ldots,\ x_2 = \ldots$

for $k = 2, 3 \ldots$

  $x_{k+1} = x_k - f(x_k)(x_k - x_{k-1})/(f(x_k) - f(x_{k-1}))$

  if converged, stop

end

```python
import numpy as np
from scipy import optimize

def secant(f,xprev, x,tol):
    k = 1
    print('    k                 x_k-1               x_k      f(x_k)')
    print('%5d %22.20f %22.20f %11.8g' % (k,xprev,x,f(x)))
    k = k + 1
    while np.abs( f(x) ) > tol:
        xnew = x - f(x)*(x - xprev)/(f(x)-f(xprev))
        print('%5d %22.20f %22.20f %11.8g' % (k,x,xnew,f(xnew)))
        k = k + 1
        xprev = x
        x   = xnew


def f(x):
    return x - x**(1./3.) - 2.


if __name__ == "__main__":
    secant(f, 4., 3., 1.e-25)
```

Secant Example

Solve:

$$x - x^{1/3} - 2 = 0$$

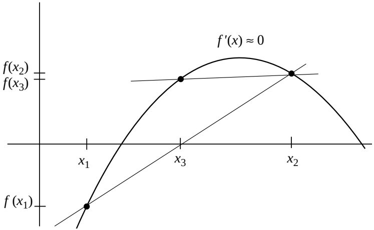
Python produces the root 3.521379706804568.

k	x_k-1	x_k	f(x_k)
1	4.000000000000000000	3.000000000000000000	-0.44224957
2	3.000000000000000000	3.517342617808598692	-0.003455471
3	3.517342617808598692	3.521416652513002085	3.1625043e-05
4	3.521416652513002085	3.521379704427526124	-2.0347151e-09
5	3.521379704427526124	3.521379706804566023	-1.3322676e-15
6	3.521379706804566023	3.521379706804567799	0

Conclusions:

- Converges almost as quickly as Newton's method ($r = \frac{1+\sqrt{5}}{2} \approx 1.62$).
- There is no need to compute $f'(x)$.
- The algorithm is simple.
- Two initial guesses are necessary
- Iterations are not guaranteed to stay inside an ordinal bracket.

Divergence of Secant Method



Since

$$x_{k+1} = x_k - f(x_k)\left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right]$$

the new guess, x_{k+1}, will be far from the old guess whenever $f(x_k) \approx f(x_{k-1})$ and $|f(x)|$ is not small.

Summary

- Plot $f(x)$ before searching for roots
- Bracketing finds coarse interval containing roots and singularities
- Bisection is robust, but converges slowly
- Newton's Method
  - Requires $f(x)$ and $f'(x)$.
  - Iterates are not confined to initial bracket.
  - Converges rapidly ($r = 2$).
  - Diverges if $f'(x) \approx 0$ is encountered.
- Secant Method
  - Uses $f(x)$ values to approximate $f'(x)$.
  - Iterates are not confined to initial bracket.
  - Converges almost as rapidly as Newton's method ($r \approx 1.62$).
  - Diverges if $f'(x) \approx 0$ is encountered.

fzero Function

fzero is a hybrid method that combines bisection, secant and reverse quadratic interpolation

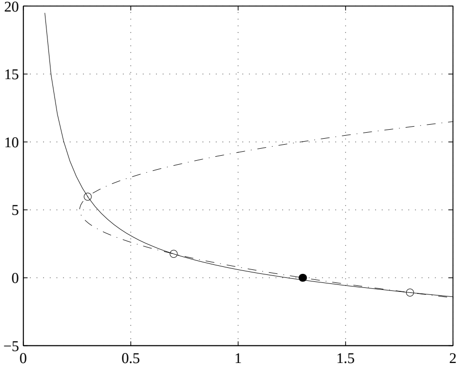
```
r = fzero('fun',x0)
r = fzero('fun',x0,options)
r = fzero('fun',x0,options,arg1,rg2,...)
```

x0 can be a scalar or a two element vector

- If x0 is a scalar, fzero tries to create its own bracket.
- If x0 is a two element vector, fzero uses the vector as a bracket.

Reverse Quadratic Interpolation

Find the point where the x axis intersects the sideways parabola passing through three pairs of $(x, f(x))$ values.



`fzero` Function

`fzero` chooses next root as

- Result of reverse quadratic interpolation (RQI) if that result is inside the current bracket.
- Result of secant step if RQI fails, and if the result of secant method is in inside the current bracket.
- Result of bisection step if both RQI and secant method fail to produce guesses inside the current bracket.

fzero Function

Optional parameters to control `fzero` are specified with the `optimset` function.

Tell `fzero` to display the results of each step:

```
>> options = optimset('Display','iter');
>> x = fzero('myFun',x0,options)
```

Tell `fzero` to use a relative tolerance of 5×10^{-9}:

```
>> options = optimset('TolX',5e-9);
>> x = fzero('myFun',x0,options)
```

Tell `fzero` to suppress all printed output, and use a relative tolerance of 5×10^{-4}:

```
>> options = optimset('Display','off','TolX',5e-4);
>> x = fzero('myFun',x0,options)
```

fzero Function

Allowable options (specified via `optimset`):

Option type	Value	Effect
'Display'	'iter'	Show results of each iteration
	'final'	Show root and original bracket
	'off'	Suppress all print out
'TolX'	tol	Iterate until $\lvert\Delta x\rvert < \max\left[\texttt{tol}, \texttt{tol} * \texttt{a}, \texttt{tol} * \texttt{b}\right]$ where $\Delta x = (b-a)/2$, and $[a, b]$ is the current bracket.

The default values of 'Display' and 'TolX' are equivalent to

```
options = optimset('Display','iter','TolX',eps)
```

fzero example

Take

$$f(x) = x^{10} - 1$$

```
>> f = @(x)x.^10 - 1;
>>  options = optimset('display','iter');
>> [x,fx]=fzero(f,0.5,options)
```

Instructor Notes

- Approximating $\frac{df(x)}{dx} \approx \frac{Im(f(x+ih))}{h}$ where $i = \sqrt{-1}$ and $h \in \mathbb{R}$ where $h \approx 0$