Table of Contents

Lecture 9: Newton's method in > 1D

We've seen a lot about Newton iteration in one dimension. We've also seen a bit about how things can fail.

But life is not one-dimensional. Real applications often require solving equations in 2, 3, or even infinite dimensions. We can use Newton's method for all of these cases. Everything becomes more challenging as the dimension becomes higher. But we've already seen the kind of problems that can arise. Going further only requires more levels of abstraction.

This lecture will very much illustrate the following general principle:

```
There are two kinds of problems in mathematics and science: (1) linear, an
d (2) nonlinear.
We know how to solve linear problems using the tools of linear algebra; ve
ctors, matrices, etc.
We know how to solve nonlinear problems by turning them into a collection
 of linear problems.
```

An example 2D problem

Now suppose we want to solve the pair of coupled equations for two variables:

$$f(x, y) = 0, \quad g(x, y) = 0$$

For example:

$$f(x, y) = x^2 + y^2 - 1, \quad g(x, y) = y - x^2$$

This example might seem contrived. But it's a reasonable model for figuring out when a comet might cross a planetary orbit. Or:

Are we all going to die like the dinosaurs?

A planetary orbit like earth is very close to a **circle**. The orbit of a long-period comet is very close to a **parabola**. Therefore:

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

In [2]:

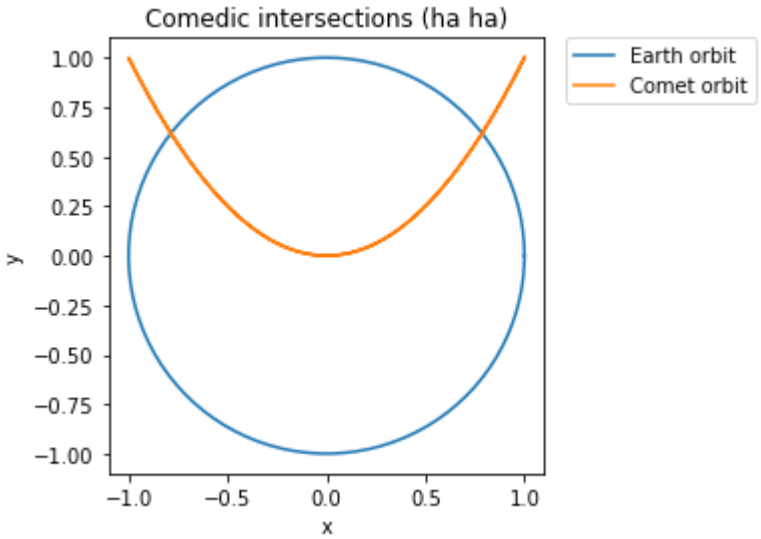
```
t = np.linspace(0,2*np.pi,200)
x, y = np.cos(t), np.sin(t)
```

In [3]:

```
fig, ax = plt.subplots()
ax.plot(x,y,label='Earth orbit')
ax.plot(x,x*x,label='Comet orbit')
ax.set(aspect=1,xlabel='x',ylabel='y',title='Comedic intersections (ha ha)')
ax.legend(bbox_to_anchor=(1.05, 1),loc=2,borderaxespad=0.)
# bbox_to_anchor says where to anchor a corner of the legend
# loc chooses which corner of the legend to anchor
# borderaxespad sets the pad between the outline of the legend, and the anchored
 edges
# plt.savefig('conic_intersection.pdf',bbox_inches='tight')
```

Out[3]:

```
<matplotlib.legend.Legend at 0x7f91e35525f8>
```



Making figures

Remember, when you make plots for quizzes/assignments, you need to make them look nice. Legends are really important for this, and there's an excellent matplotlib tutorial here (https://matplotlib.org/users/legend_guide.html).

The commands I've used above should work well for putting legends outside the axes (but still in the figure).

See if you can guess where they intersect...

In [4]:

```
phi = (np.sqrt(5)-1)/2

x, y = np.sqrt(phi), phi

print(x**2 + y**2 - 1,' ',  y - x**2)
```

```
0.0   1.1102230246251565e-16
```

Taylor series in 2D

The way forward is to just do what we already know about Newton's method for each function and variable.

$$f(x+\Delta x, y+\Delta y) \approx f(x,y) + \Delta x \frac{\partial f(x,y)}{\partial x} + \Delta y \frac{\partial f(x,y)}{\partial y} + \mathcal{O}(\Delta x^2, \Delta y^2, \Delta x \Delta y)$$
$$g(x+\Delta x, y+\Delta y) \approx g(x,y) + \Delta x \frac{\partial g(x,y)}{\partial x} + \Delta y \frac{\partial g(x,y)}{\partial y} + \mathcal{O}(\Delta x^2, \Delta y^2, \Delta x \Delta y)$$

We would just repeat the pattern if there were more functions and variables.

In one dimension, we eventually did something like

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

But to take a couple steps back, this is the same things as solving

$$f'(x)\,\Delta x = -f(x)$$

We just need to divide by $f'(x)$. Two dimensions works more like the latter case than the former. Putting things into matrix notation we find we need to solve:

$$\begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

Rather than a single $f'(x)$. now we have a *matrix* of partial derivatives. But the principle is very similar.

The Jacobian

It's useful to give this matrix a name. It's called the ***Jacobian matrix***. At every point x, y there we have the matrix of partial derivatives

$$\mathbf{J}(x,y) \equiv \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{bmatrix}$$

In index notation:

$$\mathbf{J}_{i,j}(\boldsymbol{x}) \equiv \frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$$

This works for any number of dimensions; even infinite if you are careful. Therefore, if:

$$\boldsymbol{x} \in \mathbb{R}^d, \quad \boldsymbol{F} : \mathbb{R}^d \to \mathbb{R}^d$$

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{x} - \mathbf{J}^{-1}(\boldsymbol{x}) \, . \, \boldsymbol{f}(\boldsymbol{x})$$

With the abstract notation, Newton iteration looks the same in all dimensions

$$\boldsymbol{x}_{n+1} = \boldsymbol{F}(\boldsymbol{x}_n)$$

Matrix inverses are hard

Notice that I've defined the *matrix inverse* $\mathbf{J}^{-1}$. You'll likely recall from your linear algebra course that matrix inverses can be difficult and time consuming. In one dimension, Newton's method only required iterating $F(x)$. In multiple dimension, even computing $\boldsymbol{F}(\boldsymbol{x})$ itself can be tricky.

For this reason, it's better to do things the slightly different way:

1. Solve the linear equation for $\boldsymbol{\Delta x}$:
$$\mathbf{J}(\boldsymbol{x}_n) \, . \, \boldsymbol{\Delta x} = -\boldsymbol{f}(\boldsymbol{x}_n)$$
2. Then set:
$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \boldsymbol{\Delta x}.$$

Mathematically, these are identical actions. But they are very different conceptually when you actually need to make it work.

```
The first method *inverts* a matrix; without reference to a right-hand sid
e vector.
The other method *solves* a matrix equation for a specific right-hand side
 vector.
```

These two pictures differ greatly in their computational cost and simplicity.

Example

$$\begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

The first approach

Here is the matrix and right-hand side.

In [5]:

```
A = np.array([[1,1],[1,-2]])
b = np.array([6,3])
```

First form the inverse of the matrix.

In [6]:

```
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[ 0.66666667  0.33333333]
 [ 0.33333333 -0.33333333]]
```

Now apply it to a specific right-hand side.

In [7]:

```
print(A_inv.dot(b))
```

```
[5. 1.]
```

This might be worth it if we wanted to apply the inverse to a lot of right-hand side vectors. In 2D, one more vector might be worth it. In more dimensions, the following is much more what we would want to do.

The second approach

This just gives the answer for the *given* right-hand side.

In [8]:

```
print(np.linalg.solve(A,b))
```

```
[5. 1.]
```

In fact, the way the inverse is actually built is to do this for multiple right-hand sides. Ie,

In [9]:

```
print(np.linalg.solve(A,np.array([1,0])))
print(np.linalg.solve(A,np.array([0,1])))
```

```
[0.66666667 0.33333333]
[ 0.33333333 -0.33333333]
```

This is 2 times the effort as just going the solve. In N dimensions, it's N times more effort.

5 minute break

Make sure you understand all the previous parts.

Ask if you aren't sure.

If you have your laptop, there's a quiz question for you to do. Just one! But make sure you answer it.

And here are some questions for you:

- Try out the Newton stepping method
- What do you do if you don't have a formula for the derivative? Can you estimate the derivative numerically? (This is called the secant method (https://en.wikipedia.org/wiki/Secant_method))
- How would this work in multiple dimensions (https://en.wikipedia.org/wiki/Broyden%27s_method)?
- What is the order of the secant method? (You might be able to guess it)
- Try the Newton stepper with an example function

2D, or not 2D ?

In Lecture 6, we looked at a 1D example in the complex plane.

$$h(z) = z^5 - 1$$

The roots in this case are

$$z^* \;=\; \exp \frac{2\,\pi\,i\,n}{5}, \quad \text{where} \quad n = 0, 1, 2, 3, 4$$

In that case, we iterated the following map

$$z_{n+1} \;=\; F(z_n), \quad \text{where} \quad F(z) \;=\; \frac{4z^5 + 1}{5z^4}$$

Complex to 2D: $\mathbb{C}$ to $\mathbb{R}^2$

This is actually a 2D problem in disguise. The complex plane is a restricted version of the full 2D plane. We could solve this problem as a real 2D problem instead. That is,

$$f(x, y) = \text{Real}\{\, h(x + iy)\,\}, \quad g(x, y) = \text{Imag}\{\, h(x + iy)\,\}$$

This implies

$$f(x, y) = x^5 - 10x^3y^2 + 5xy^4 - 1, \quad g(x, y) = 5x^4y - 10x^2y^3 + y^5$$

Now the solutions are the real and imaginary parts of the complex exponnetial solution:

$$x^*\,,\; y^* \;=\; \cos \frac{2\pi n}{5},\; \sin \frac{2\pi n}{5}, \quad \text{for} \quad n = 0, 1, 2, 3, 4$$

But if we make code that can solve a problem that could reduce to the complex plane, then we make code that can solve problems that can't.

First define a function that takes in a 2D vector, and gives back a 2D vector:

In [10]:

```
def f(x):

    x0, x1 = x[0], x[1]

    f0 = x0**5 - 10*(x0**3)*(x1**2) + 5*(x0)*(x1**4) - 1
    f1 = 5*(x0**4)*(x1) - 10*(x0**2)*(x1**3) + x1**5

    return np.array([f0,f1])
```

Now define a function that takes in a 2D vector, and gives back a 2D Jacobian matrix:

In [11]:

```
def J(x):

    x0, x1 = x[0], x[1]

    J00, J01 = 5*x0**4 - 30*(x0**2)*(x1**2) + 5*x1**4, -20*(x0**3)*(x1) + 20*(x0
)*(x1**3)
    J10, J11 = 20*(x0**3)*(x1) - 20*(x0)*(x1**3), 5*x0**4 - 30*(x0**2)*(x1**2) +
 5*x1**4

    return np.array([[J00,J01],[J10,J11]])
```

In [12]:

```
x0, x1 = 2, 1

x = np.array([x0,x1])

z = x0 + 1j*x1

def h(z):
    return z**5 - 1

def Dh(z):
    return 5*z**4
```

Two different ways to compute the same thing:

In [13]:

```
print(f(x))
print('')
print(h(z))
```

```
[-39  41]

(-39+41j)
```

In [14]:

```
print(J(x))
print('')
print(Dh(z))
```

```
[[ -35 -120]
 [ 120  -35]]

(-35+120j)
```

Complex numbers are matrices

The fact that J is a matrix of the form

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

means we can encode it into complex numbers of the form $a + i\,b$. Matrices of this form have the same algebra as complex numbers. That is, if

$$(a + ib)(c + id) \;=\; (e + if)$$

then

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix} \cdot \begin{bmatrix} c & d \\ -d & c \end{bmatrix} \;=\; \begin{bmatrix} e & f \\ -f & e \end{bmatrix}.$$

Try it for yourself.

When can you reduce to complex?

If we have two functions of two variables, $f(x, y)$, $g(x, y)$. We can determine if they are the real and imaginary parts of a single complex-valued function, $h(x + iy)$ by checking the ***Cauchy-Riemann conditons***

$$\frac{\partial f}{\partial x} \;=\; \frac{\partial g}{\partial y}, \quad \frac{\partial f}{\partial y} \;=\; -\frac{\partial g}{\partial x}.$$

If this is the case, we can map the Jacobian matrix into the above form. But this doesn't have to be the case for 2D matrices in general.

Assuming we know how to compute $\boldsymbol{f}(\boldsymbol{x})$ and $\mathbf{J}(\boldsymbol{x})$, let's code Newton's method in any dimension

In [15]:

```
def Newton_step(f, J, xold):

    A, b = J(xold), -f(xold)

    Delta_x = np.linalg.solve(A,b)

    xnew = xold + Delta_x

    return xnew, Delta_x
```

2D Newton's method

Measuring errors in multiple dimensions

Unlike in 1D, there are multiple ways of measuring the size of a vector. This is really important for understanding convergence in multiple dimensions, such as for iterative procedures like Newton's method.

There are three main ways to measure the size of an error vector $x = (x_0, x_1, \ldots, x_n)$:

- The maximum norm ℓ^∞.
$$\|x\|_{\ell^\infty} = \max_i |x_i|$$
This is just the largest component of the vector:
- The Euclidean norm ℓ^2.
$$\|x\|_{\ell^2} = \sqrt{\sum_{i=0}^{n} x_i^2}$$
This corresponds to the geometric length of the vector in Euclidean space.
- The "taxicab" norm ℓ^1.
$$\|x\|_{\ell^1} = \sum_{i=0}^{n} |x_i|$$
This one is the sum of all the component lengths, and is a distance that you'd need to drive on a Cartesian grid to get to your destination.

In [16]:

```
def max_norm(x):
    return np.max(np.abs(x))

def l2_norm(x):
    return np.sqrt(x.dot(x))

def l1_norm(x):
    return np.sum(np.abs(x))
```

An important property of these norms is that they satisfy

```
l1_norm(x) >= l2_norm(x) >= max_norm(x)
```

In [17]:

```
x   = np.array([-3,5]) # initial guess

tol, imax = 1e-19, 20
eps, i    = np.inf, 0

norm = l1_norm

while ( eps > tol ) and (i < imax):
    x, dx = Newton_step(f,J,x)
    eps  = norm(dx)
    i   += 1

    print('i, |dx|, |f(x)|: {:>2d}, {:>6.1e}, {:>6.1e}'.format(i, norm(dx), norm
(f(x))))

print('x                  =',x)
print('x.x                =', x.dot(x))
print('arctan(y/x)/(2 \pi) = ',np.arctan(x[1]/x[0])/(2*np.pi))
```

```
i, |dx|, |f(x)|:  1, 1.6e+00, 2.9e+03
i, |dx|, |f(x)|:  2, 1.3e+00, 9.6e+02
i, |dx|, |f(x)|:  3, 1.0e+00, 3.2e+02
i, |dx|, |f(x)|:  4, 8.2e-01, 1.0e+02
i, |dx|, |f(x)|:  5, 6.6e-01, 3.4e+01
i, |dx|, |f(x)|:  6, 5.3e-01, 1.1e+01
i, |dx|, |f(x)|:  7, 4.3e-01, 4.0e+00
i, |dx|, |f(x)|:  8, 3.4e-01, 1.5e+00
i, |dx|, |f(x)|:  9, 3.6e-01, 6.0e-01
i, |dx|, |f(x)|: 10, 2.9e-01, 4.2e-01
i, |dx|, |f(x)|: 11, 7.3e-02, 4.3e-02
i, |dx|, |f(x)|: 12, 7.4e-03, 4.3e-04
i, |dx|, |f(x)|: 13, 9.4e-05, 6.9e-08
i, |dx|, |f(x)|: 14, 1.2e-08, 1.2e-15
i, |dx|, |f(x)|: 15, 2.3e-16, 3.3e-16
i, |dx|, |f(x)|: 16, 6.7e-17, 3.3e-16
i, |dx|, |f(x)|: 17, 6.7e-17, 3.3e-16
i, |dx|, |f(x)|: 18, 6.7e-17, 3.3e-16
i, |dx|, |f(x)|: 19, 6.7e-17, 3.3e-16
i, |dx|, |f(x)|: 20, 6.7e-17, 3.3e-16
x                  = [-0.80901699  0.58778525]
x.x                = 1.0
arctan(y/x)/(2 \pi) =  -0.1
```

Conic intersection or: Are we all going to die like the dinosaurs?!

Now we can refine the function and the Jacobian matrix to solve the **circle and parabola** problem at the beginning. Even though it's a simple problem, it is not one that reduces to 1D complex variables. Check using the Cauchy Riemann equations.

Really. Check!

In [18]:

```
def f(x):

    x0, x1 = x[0], x[1]

    f0 = x0**2 + x1**2 - 1
    f1 = x1 - x0**2

    return np.array([f0,f1])
```

In [19]:

```
def J(x):

    x0, x1 = x[0], x[1]

    J00, J01 =  2*x0, 2*x1
    J10, J11 = -2*x0, 1

    return np.array([[J00,J01],[J10,J11]])
```

In [20]:

```
x   = np.array([3,5]) # initial guess

tol, imax = 1e-12, 20
eps, i    = np.inf, 0

while ( eps > tol ) and (i < imax):
    x, dx = Newton_step(f,J,x)
    eps  = l1_norm(dx)
    i   += 1

    print('i, |dx|, |f(x)|: {:>2d}, {:>6.1e}, {:>6.1e}'.format(i, norm(dx), norm
(f(x))))

print('')
print(' x            =',x)
print('|x - x_exact| =  ', max_norm(x-np.array([np.sqrt(phi),phi])) )
```

```
i, |dx|, |f(x)|:  1, 3.7e+00, 9.4e+00
i, |dx|, |f(x)|:  2, 1.9e+00, 2.3e+00
i, |dx|, |f(x)|:  3, 7.9e-01, 4.4e-01
i, |dx|, |f(x)|:  4, 1.9e-01, 3.1e-02
i, |dx|, |f(x)|:  5, 1.3e-02, 1.9e-04
i, |dx|, |f(x)|:  6, 6.6e-05, 7.6e-09
i, |dx|, |f(x)|:  7, 2.4e-09, 1.1e-16
i, |dx|, |f(x)|:  8, 8.1e-17, 1.1e-16

 x            = [0.78615138 0.61803399]
|x - x_exact| =   1.1102230246251565e-16
```