Chapter 6

Lab 6: Differentiation and Projectile Motion

6.1 Introduction

In this lab we will apply numerical differentiation to elementary functions. We'll apply the Euler method to compute the trajectory of a projectile, and compare our results with the analytic solution. We'll also show that our numerical technique easily accommodates air resistance, a problem that has no analytic solution using elementary functions.

6.2 Preparation

Our code is going to get complicated enough that we will need to pay some attention to variable scope, as illustrated here:

```
i=1
j=2
k=3
def f(i,j):
    print(i,j,k)
f(i,j)
f(j,i)
```

Try to predict the output of this snippet before running it. The first three lines define integers i,j, and k. These have global scope, which means they can be accessed from anywhere. The function `f(i,j)` has parameters i and j which have a scope limited to the function f. Even though they have the same name as the global variables i and j, they are independent quantities. Within the function f the variable i is the first parameter, and j is the second parameter. Because they have the same name, the global variables i and j are *shadowed* by the local parameters i and j. The global variable k is not shadowed.

In this lab, we will also be passing a function as an argument to another function, as in this simple example:

```
def show(f):
    print(f(2))
    print(f(3))

def f(i):
    return i**2

def g(i):
    return i**3

show(f)
show(g)
```

Here the `show` function takes another function `f` as an argument. Within the show function, the function `f` is called using paranthesis just like any other function, as in `f(2)` We define two additional functions, `f` which returns the square of its argument, and `g` which returns the cube. When `show(f)` the output 4 and 9. When `show(g)` the output 8 and 27. Run the code as is, and also check what happens if you define `g(i)` to require a second argument as in `g(i,j)`.

6.3 Numerical Differentiation

In lecture we derived the right derivative (aka foward derivative) formula

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

for numerically determining the derivative of the function f. Remember we do not calculate the $\mathcal{O}(h)$ term, that indicates that the truncation error is of order h. We also derived the centered derivative formula:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

To evaluate a derivative using any of these formulas, we need to choose an appropriate value of h. If h is too large, the truncation error (the amount the estimated value differs from the actual value) will dominate. If h is too small, we will encounter problems with floating point precision.

△ **Jupyter Notebook Exercise 6.1:** Implement the right derivative formula as `right(f, x, h)` where f is the function to be evaluated, x is the location to evaluate the derivative, and h is the step size for the numerical integration. Check you code on several functions with known derivatives, like this:

```
def f(x):   # derivative 0
    return 2
def g(x):   # derivative 3
    return 3*x
def h(x):   # derivative 4x
    return 2*x**2

print(right(f,1,0.01))
print(right(g,1,0.01))
print(right(h,1,0.01))
```

△ **Jupyter Notebook Exercise 6.2:** Implement the center derivative function as `center` and test it in the same manner as in the previous exercise for `right`.

△ **Jupyter Notebook Exercise 6.3:** Compare the performance of `right` and `center` like this:

```
def f(x):  # derivative 6x**2
    return 2*x**3

print("right:", right(f,1,0.1),   "center:", center(f,1,0.1))
print("right:", right(f,1,0.01),  "center:", center(f,1,0.01))
print("right:", right(f,1,0.001), "center:", center(f,1,0.001))
```

Recall that the truncation error goes as h for the right derivative and as h^2 for the center derivative. Are these results consistent with that expecation?

From now on, we will use the center derivative function only due to its better performance. We can plot the derivative of a function like this:

```
def f(x):
    return 0.5*x**2

x = np.linspace(0,1,100)
y = center(f, x, 0.1)
plt.plot(x,ya,"-b")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Notice how the argument x passed to the function `right(f,x,h)` and then to `f(x)` is now a numpy array of 100 values from 0 to 1. The derivative is now evalued at 100 places with a single call.

△ **Jupyter Notebook Exercise 6.4:** Define $f(x) = x^3$. Use your `center` function to evaluate it's derivative $f'(x)$ in the x range $[-2, 2]$. Plot both $f(x)$ and $f'(x)$ in that range (in the same plot) using different colors for each. Add a legend and axis labels.

△ **Jupyter Notebook Exercise 6.5:** Define $f(x) = \sin(x)$ using the `np.sin` function. Use your `center` function to evaluate it's derivative $f'(x)$ in the x range $[0, 2\pi]$. Plot $f(x)$, $f'(x)$, and $\cos(x)$ in that range (all in the same plot) using different colors for each. Add a legend and axis labels.

6.4 Projectile Motion

In lecture, we derived the Euler Method for iteratively determining the trajectory of a particle:

$$\begin{aligned}\vec{v}_{n+1} &= \vec{v}_n + \tau\vec{a}_n \\ \vec{r}_{n+1} &= \vec{r}_n + \tau\vec{v}_n\end{aligned}$$

△ **Jupyter Notebook Exercise 6.6:** Implement a function

```
def euler(dt, x, y, vx, vy, ax, ay):
    # your code here
    return x, y, vx, vy
```

which calculates the next iteration of x,y,v_x, and v_y from the current values of x,y,v_x,v_y,a_x and a_y. Notice that this function returns several different variables at once using a comma separated list referred to as tuple in python. To retrieve the individual variables from the tuple, simply call the function like this:

```
x,y,vx,vy = euler(x,y,vx,vy,ax,ay)
```

One downside of this convenient approach is that you must get the order of the variables correct! Check you implementation against the following test values:

```
print(np.around(euler(0.134, 0.659, 0.282, 0.662, 0.643, 0.900, 0.451),2))
print(np.around(euler(0.924, 0.959, 0.575, 0.299, 0.710, 0.699, 0.471),2))
```

and ensure that you get the correct output:

```
[0.75 0.37 0.78 0.7 ]
[1.24 1.23 0.94 1.15]
```

We will use the Euler Method to simulate projectile motion. We'll take the initial velocity to be 20 m/s and take $g = 9.8$ m/s^2. Here's a snippet of code that sets these constants and determines the x and y coordinates of the initial velocity from an angle θ, which is set to 45°:

```
tau  = 2*np.pi
vi     = 20    # [m/s]
g      = 9.8   # [m/s^2]
theta  = tau/8
dt  = 0.01 # [s]
x   = 0    # [m]
y   = 0    # [m]
vx  = vi*np.cos(theta)
vy  = vi*np.sin(theta)
```

The trajectory of the particle can be determined using the following algorithm:

```
Create an empty array tjx # will contain $x$ positions of the trajectory
Create an empty array tjy # will contain $y$ positions of the trajectory
while $y \geq 0$:
    Append the $x$ position to tjx
    Append the $y$ position to tjy
    Compute the next values of $x, y, v_x$ and $v_y$ using the Euler Method
Plot tjy versus tjx
```

Notice that the algorithm stops just before the projectile reaches $y \leq 0$.

△ **Jupyter Notebook Exercise 6.7:** Use the Euler Method to plot the trajectory of a projectile with the initial conditions described above.

△ **Jupyter Notebook Exercise 6.8:** Derive an expression (paper and pencil) for the maximum range of the trajectory and evaluate the range for these initial conditions. Are the results consistent?

△ **Jupyter Notebook Exercise 6.9:** Extend your simulation to record v_x and v_y at each step along with the x and y positions. Take the mass of the projectile to be $m = 0.215$ kg and plot the kinetic energy, potential energy, and total energy as a function of time. To build an array containing the time of each step, for plotting quanties versus time, you can do:

```
t = np.arange(tjx.size)*dt
```

Include a legend. The x and y axes have changed to time and energy, so make certain to change the axes labels!

6.5 Projectile Motion with Drag

In this section we will consider the effect of air restistance on a baseball thrown a 22 m/s. We can model drag as a deceleration:

$$\vec{a} = -k|\vec{v}|\vec{v}$$

where $k = 0.00622$ m$^{-}1$ for typical baseballs.

△ **Jupyter Notebook Exercise 6.10:** Extend your simulation to include the effect of drag. Plot the trajectory without drag and the trajectory with drag in the same plot. Include a legend and (as always) label all axes.

△ **Jupyter Notebook Exercise 6.11:** Including the effect of air restistance, plot the kinetic energy, potential energy, and total energy (kinetic plus potential) of the projectile as a function of time. Is the total energy of the projectile conserved?