Ordinary Differential Equations

—

Projectile Motion

Keegan W$^{m.}$ Karbach

February 25, 2020

Introduction

Differential equations are incredibly useful in the study of a vast number of physical systems, their study coming into existence with the invention of Calculus. Relating a physical quantity with how that quantity changes in another dimension (often time), these classes of equations are of paramount importance across many fields of study.

When modeling physical systems using differential equations, it is often useful to take an idealized case of the system in order to formulate an analytical solution to the model, creating a mathematical foundation for understanding. However, when extending the ideal case to a more realistic situation, it is often the case that the resulting differential equations have no closed-form solution; this particular difficulty has long been the study of mathematicians. In order to properly model these systems, we need approximations.

We can construct numerical approximations by examining the definition of the derivative:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

We can approximate this using Euler's method which involves a Taylor expansion of $f(x+h)$ resulting in

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}$$

for sufficiently small values of h.
In this project, we will examine this *forward derivative* as well as other methods derived in a similar way by examining parabolic motion. This case is often investigated in introductory physics courses due to its approachability from an analytical perspective – this approachability arises from the fact that many physical factors are not addressed (i.e. the projectile is modeled as a point, air resistance is neglected, wind speed is assumed to be zero).
We will build algorithms to estimate the system and compare these estimates to the analytical solution to ensure accuracy, then we will extend the numerical method beyond the basic analytical treatment to include those factors that are not generally included.

1 Problem 2.2

Write a program that computes $f'(x)$ using the right derivative formula and plots a graph of absolute error for the calculation of the listed functions.

For the right derivative formula:

$$f'(t) = \frac{f(t+\tau) - f(t)}{\tau} + \mathcal{O}(\tau)$$

We can create a function in python to calculate the derivative of a function `func` at a given point with a certain step size `tau`.

```
def fwd_dif(func, tau, point):
    return (func(point + tau) - func(point)) / tau
```

We can then pass a vector of step sizes to this function to calculate the absolute error from the analytical solution of the derivative.

(a) x^2 at $x = 1$

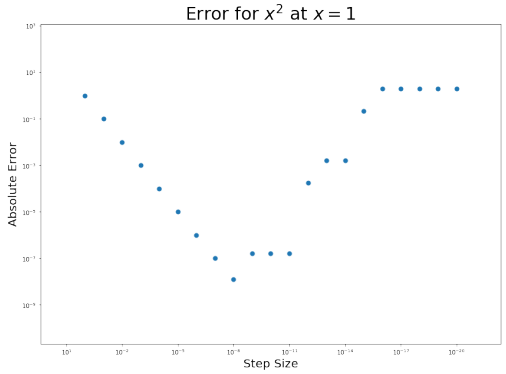

Figure 1: Here we can see something very similar as to what we see in Figure 1.13 in the text – error decreases linearly as the step size decreases. However, below 10^{-8}, the error begins to increase due to round-off error.

(b) x^5 at $x = 1$

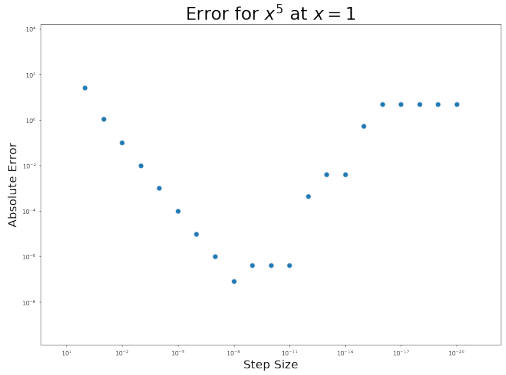

Figure 2: This exhibits much the same behavior, again with step values smaller than 10^{-8} experiencing round-off errors.

(c) $\sin x$ at $x = 0$

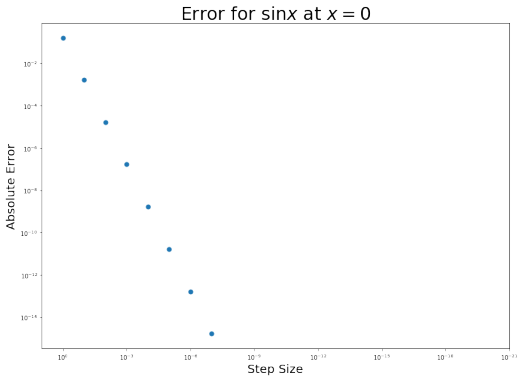

Figure 3: The sine function at $x = 0$ exhibits no round-off error past 10^{-7}, it drops to exactly zero.

(d) $\sin x$ at $x = \pi/4$

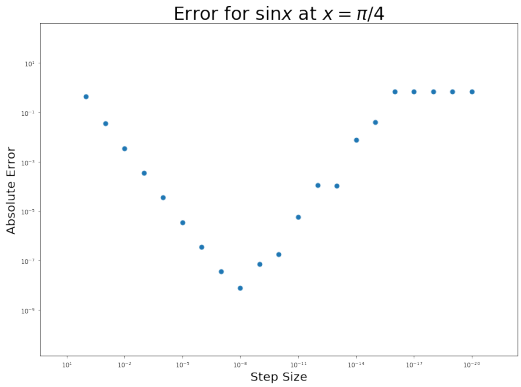

Figure 4: Here we again see round-off error beginning to become important at step sizes smaller than 10^{-8}.

(e) $\sin x$ at $x = \pi/2$

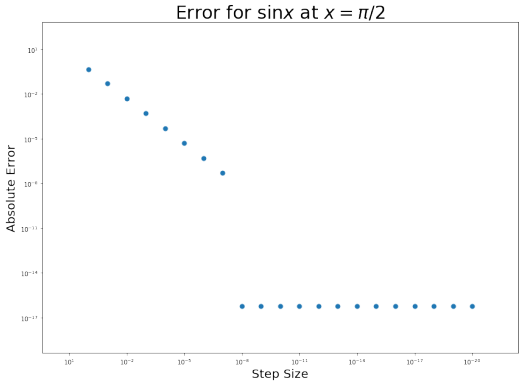

Figure 5: Here we again see a discontinuity at 10^{-7}, however at $x = \pi/2$, however the error doesn't drop to precisely zero but still remains constant.

2 Problem 2.3

The `balle` *program overestimates the range and time of flight. Fix this using 3-point interpolation to compute a corrected maximum range and time of flight with no air resistance.* $Y_0 = 0$, $v_0 = 50$, *and* $\theta = 45°$ *over a variety of time steps.*

First we need to check that our program runs correctly. Figure 6 uses the same initial parameters that are given in the book to produce Figure 2.3 and the output from my program is identical. We can therefore conclude that the program is working as intended.

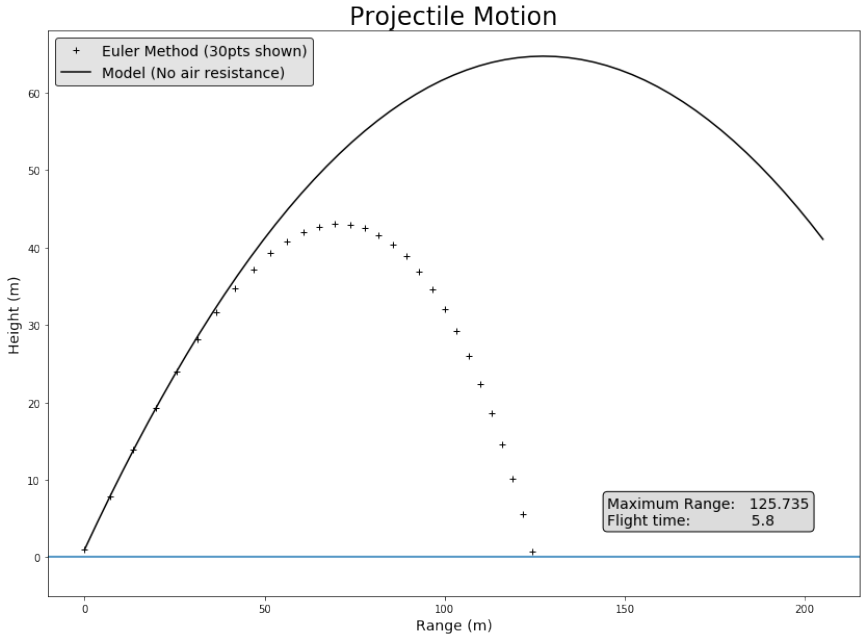

Figure 6: Output from `ball_path` for an initial height of 1 m, initial speed of 50 m/s, angle of $\theta = 45°$, and a time step of $\tau = 0.1$ s.

Now we can apply the `interpf` function on the final three points of the method (using the points immediately above and below $y = 0$) to find an improved range. Note: this will *still* be a bit of an approximation, we essentially make the finite mesh of the approximation much finer around the area of interest to greatly increase the precision of the estimate. This method of mesh refinement around $y = 0$ could be recursively done to an arbitrary level of accuracy. Figure 7 illustrates the technique as well as the improvements to range and time-of-flight. The relative errors are the percent deviation of the Euler method from the interpolated correction.

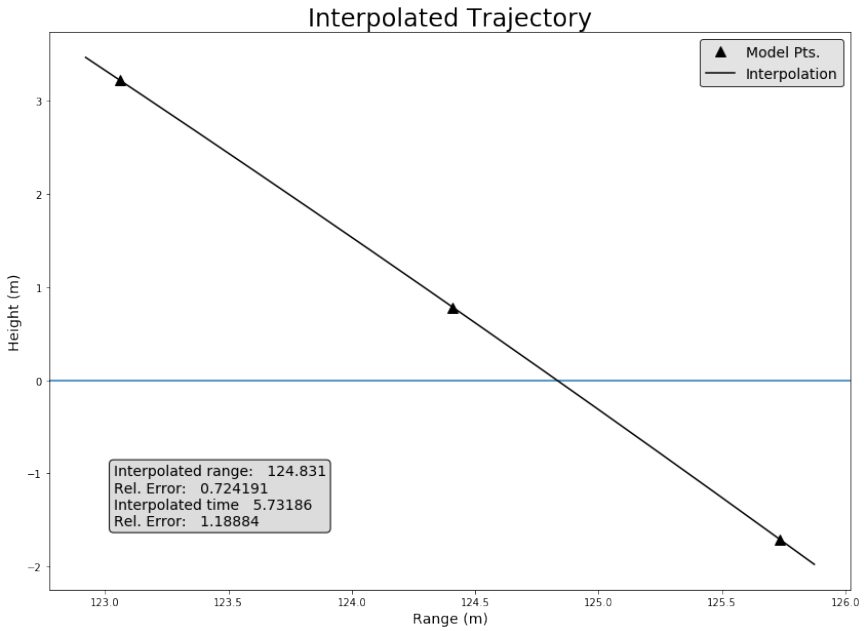

Figure 7

Now that we have a method for calculating the improvement in our method by using interpolation, we can run the code over an array of time step values. Figure 8 illustrates the findings; the error in both variables appears to decrease linearly on a log-log plot, save for the second data point in each variable. I'm unsure of the cause of this peculiarity.

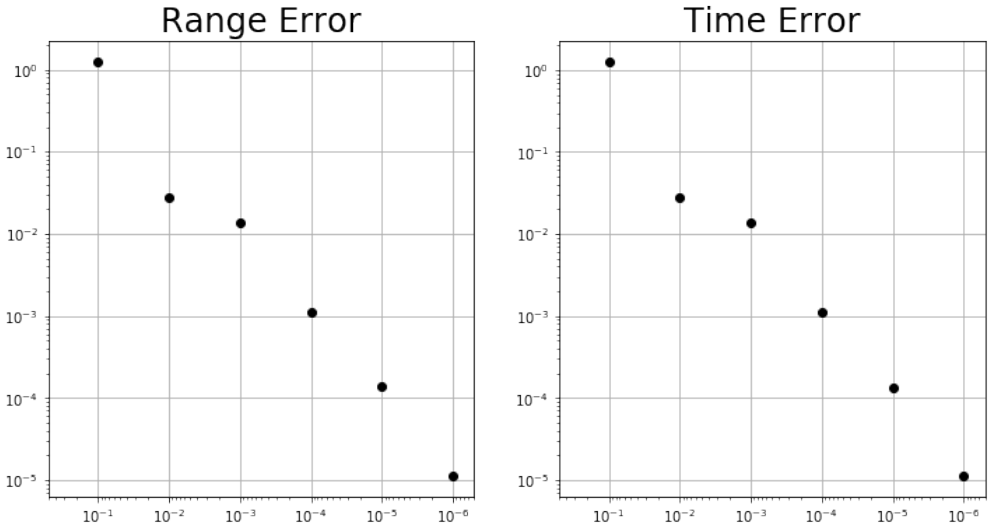

Figure 8: Improvements in estimation using 3-point interpolation versus base Euler method.

3 Problem 2.5

A batter hits a ball with an initial velocity of 50 m/s. Produce a plot of horizontal range as a function of angle for 30° < θ <60° and find the maximum range.

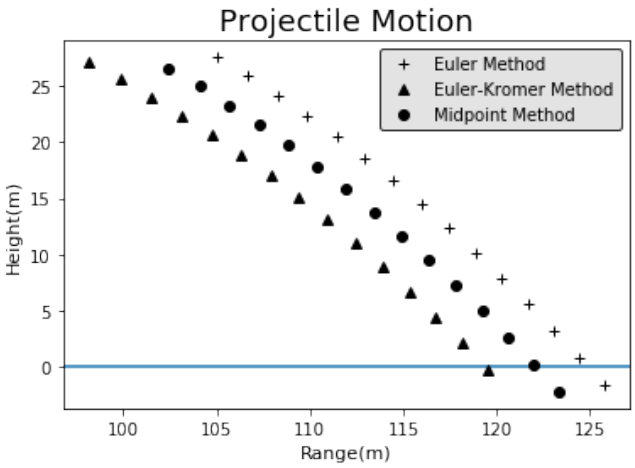

Figure 9

I modified the `balle` program to run the Euler-Cromer and midpoint methods and have compared the maximum range and corresponding angle-of-incidence for all three methods using the same initial conditions as shown in Figure 9. In Figure 10 we can see the expected features of the curves: the correct ordering of the methods and the asymmetry of the curves. The inset shows an enhancement of the area around the maximal values plotted in the figure.

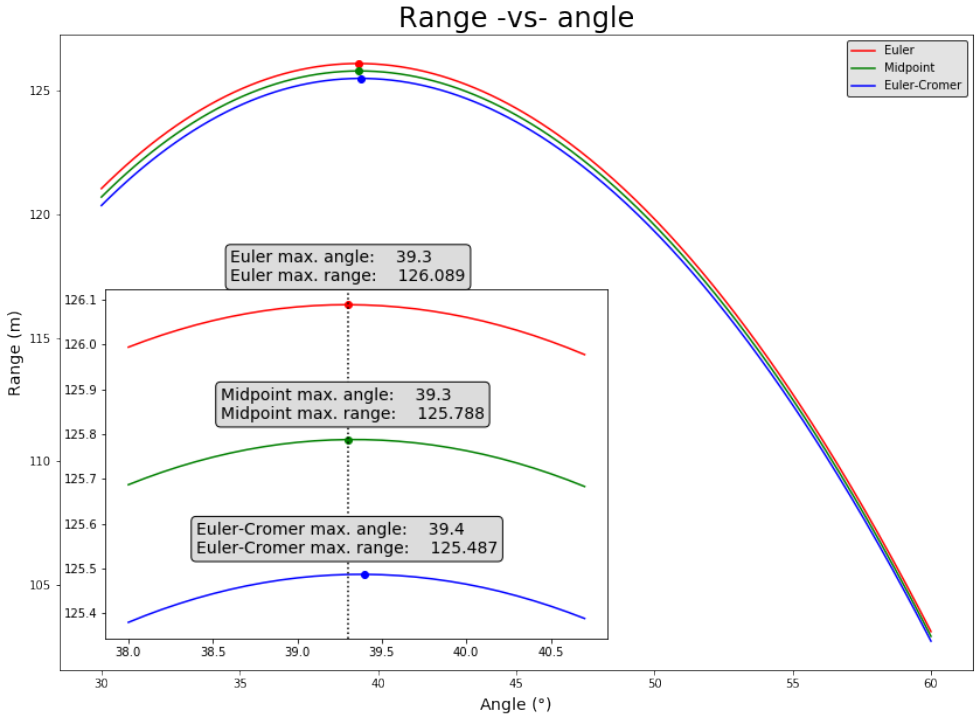

Figure 10: Maximum ranges and angles for three methods.

4 Problem 2.9

Modify `balle` *to account for wind. A batter hits a ball with* v_0 *=100 mph and an initial angle of* θ *=35°. Plot range -vs- wind velocity for wind speeds ranging [-150, 150].*

Incorporation of wind speed is a matter of changing the acceleration vector in the horizontal direction. Much the same as incorporating air resistance, this involves adding a quadratic velocity term scaled by the drag coefficient of the projectile – the implementation is as follows.

```
accel=drag*np.linalg.norm(v)*v
accel[1]=accel[1]-grav
accel[0]=accel[0]-drag*windlist[j]*np.abs(windlist[j])
```

This has the effect of acting on the modeled projectile iteratively at each time step, allowing the correct plotting of this model. Incidentally, this is one of the physical systems that doesn't have a closed-form analytical solution – when the headwind velocity is high enough, the function of the trajectory is no longer well-behaved, failing to pass the single-line test.

In order to test this implementation, Figure 11 shows projectile trajectories at a variety of wind velocities (negative wind velocity represents a head wind). The trajectories seem to follow the correct behavior, validating our computational model.

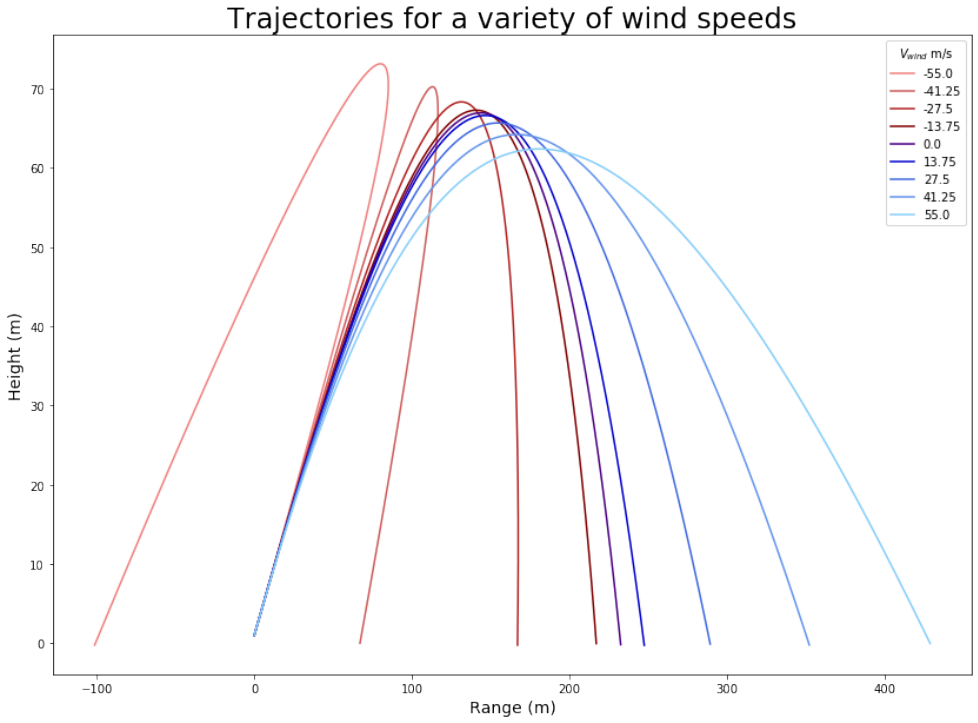

Figure 11: Verification of wind speed model.

Now in Figure 12, we plot the final range of the projectile versus the wind speed. I have plotted the axes to illustrate the behavior of the function around zero wind speed equal and to illustrate

the wind speed at which the projectile will land behind the original position. (Note: it is not recommended to play baseball in these conditions.)

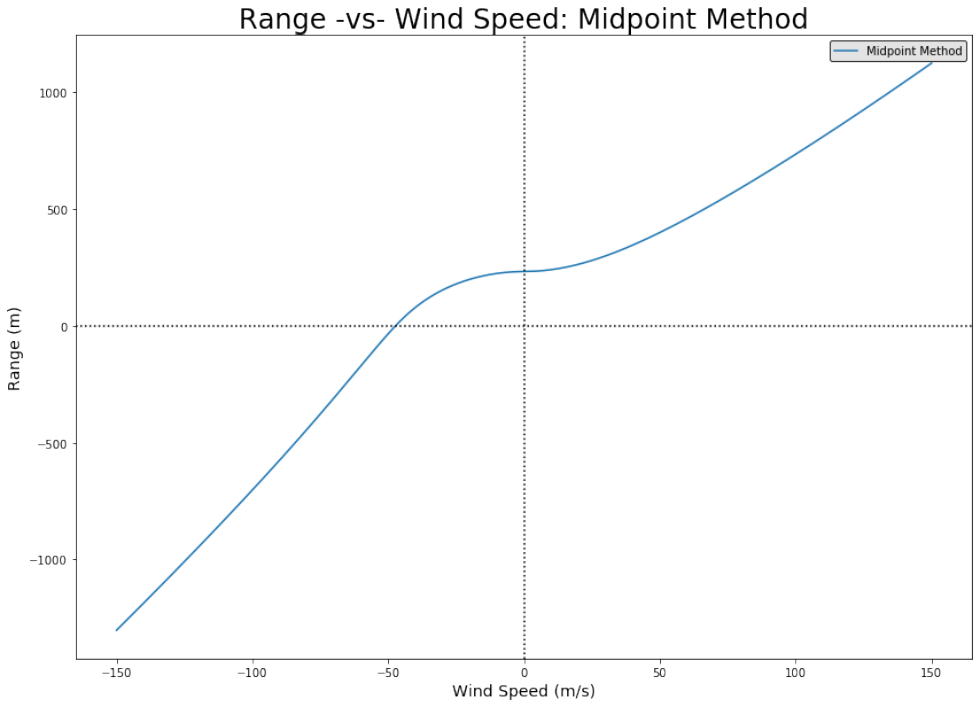

Figure 12: Range as a function of wind speed.

5 Problem 2.10

The drag coefficient for a baseball is really not a constant but rather varies with velocity. Using quadratic interpolation and the values in Table 1, re-create the plot of range versus angle and comment on the results, specifically whether the range is greater or less than the range found in Problem 2.5 when a variable geometric coefficient is introduced.

Variable geometric coefficients						
v (mph)	0	25	50	75	100	125
v (m/s)	0	11.176	22.352	33.528	44.704	55.88
C_d	0.5	0.5	0.5	0.4	0.28	0.23

Table 1

Our fist task in this scenario is to interpolate a (pseudo-)continuous function for the geometric coefficient values listed in Table 1. Let's first convert the velocities into a better unit. The problem in the book suggests to use quadratic interpolation (i.e. the `interpf()` function), interpolating using three points. The results of this are shown in the first plot in Figure 13; none of these seem acceptable over the entire range of values. However, the interpolation using the final three points shows promise for a limited range. In the second plot in Figure 13, a

linear interpolation and cubic interpolation are shown utilizing the 'SciPy' Python package [1]; both fit the values better than the quadratic interpolation over the entire range, but the cubic interpolation shows undesired deviation away from 0.5 in the lower half of the range.

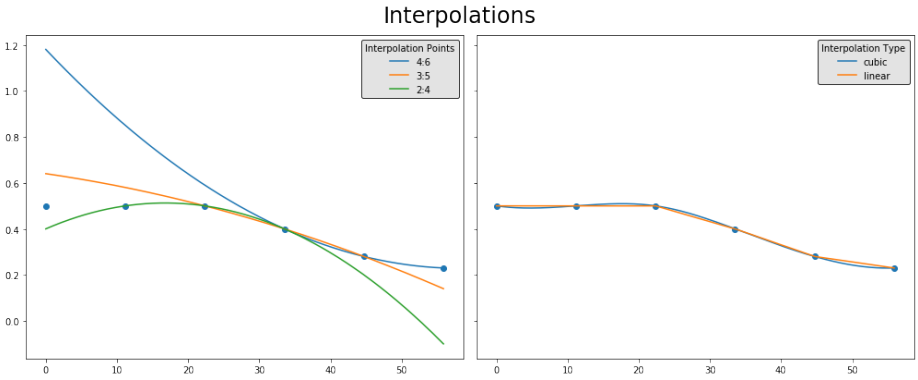

Figure 13: Exploration of interpolation

Using the observations gathered above, two interpolation models have been chosen based on fit to the unknown theoretical equation for C_d as a function of velocity. The piece-wise interpolation is constructed using a value of .5 until the value of the quadratic interpolation of the final three points of Table 1 drops below .5, at which time it is mapped to the interpolation. Next is the piece-wise linear interpolation created by SciPy; both of these will be utilized in the calculation of range versus angle-of-incidence and the results will be compared to what was computed in Problem 2.5.

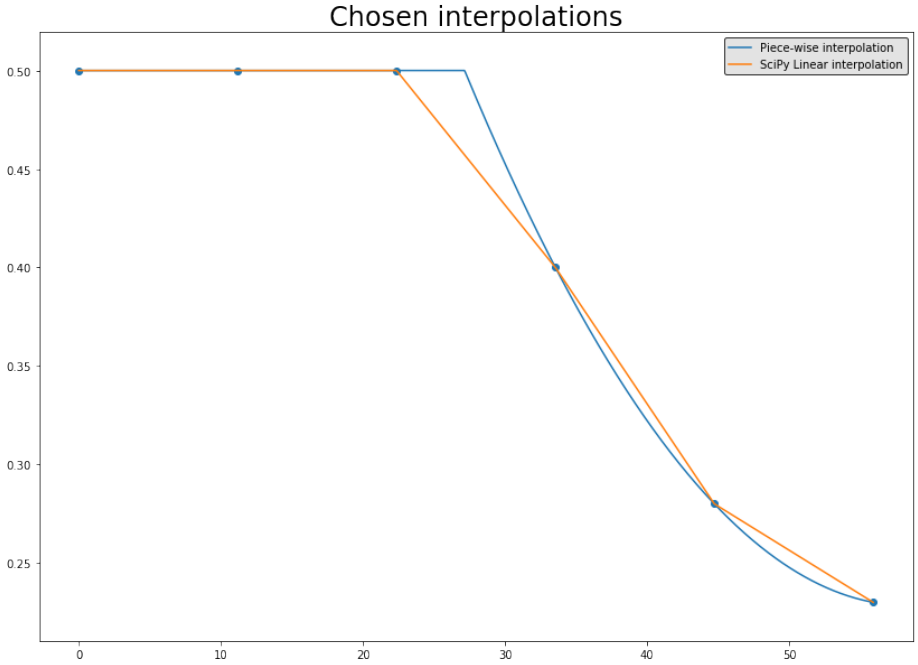

Figure 14: Chosen interpolations

Modifying the `balle` program to utilize a geometry coefficient that varies as a function of velocity was a matter of taking the interpolation vectors and choosing the correct coefficient based on the velocity of each time step. The modification to the code is as follows:

```
drag=-.5*Cd_pw[full_vvec<=np.linalg.norm(v)][-1]*rho*area/mass
```

Notice that C_d is being selected from its pseudo-continuous vector by comparing the magnitude of the instantaneous velocity to the position-matched velocity pseudo-continuous vector. The inequality has the effect of masking larger values of C_d as the projectile slows. The calculation of the drag term has now been moved into the iteration as it needs to be recalculated at each time step.

Figure 15 shows the plot of projectile range as a function of angle-of-incidence using a constant geometric coefficient (as in Problem 2.5) as well as the new calculations for both interpolations of the variable geometric coefficient values.

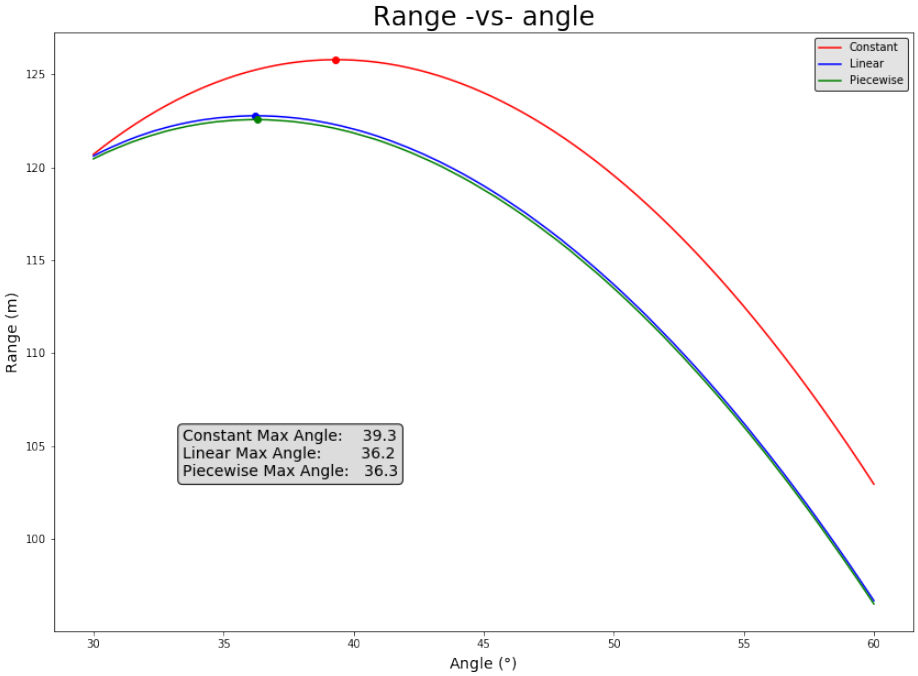

Figure 15: Range -vs- angle for variable geometric coefficient

The two curves representing variable drag coefficient are very close together; this is to be expected as the interpolations were both quite close. Figure 16 shows these two curves at a greater resolution with the corresponding maximum range angles. Both of these maximum values are less than what was calculated in the case with constant geometric coefficient due to the fact that the velocity regime that the projectile spends the most time in leads to a geometric coefficient greater than the constant given in Problem 2.5.

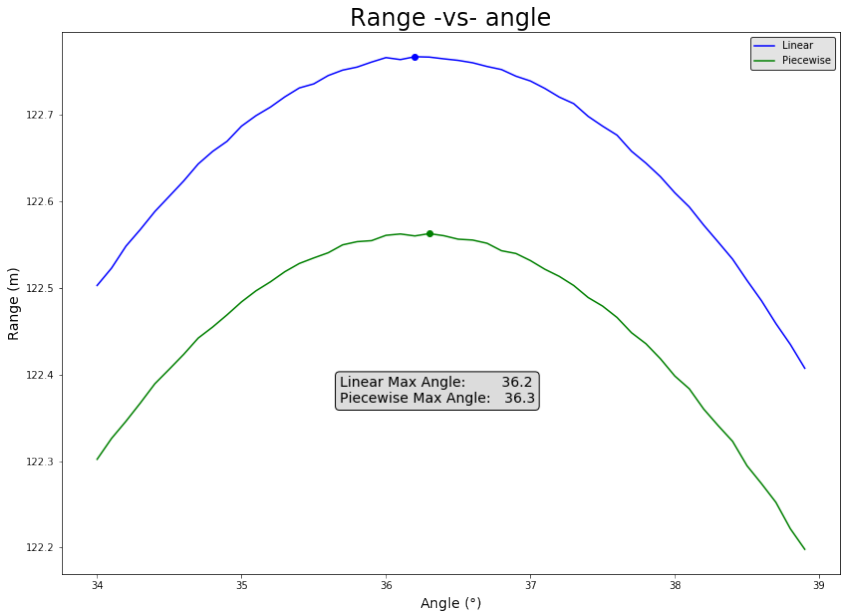

Figure 16: Variable geometric coefficient

To verify that the implementation was working as expected and for analysis of the results, values for the velocity and geometric coefficient were gathered at each time step. Histograms of these values are shown below in Figure 17. The vertical axes on these histograms are set to a logarithmic scale to better illustrate the shape of the various data – when using a linear scale, the largest bins dominate the plot. Hence, the modification of the trajectory due to variable drag coefficient is slight (this also means that if the value for C_d corresponding to the lowest velocity ranges was modified to be the same as the value given in the initial analysis, these modifications would result in curves that were much closer to the constant function).

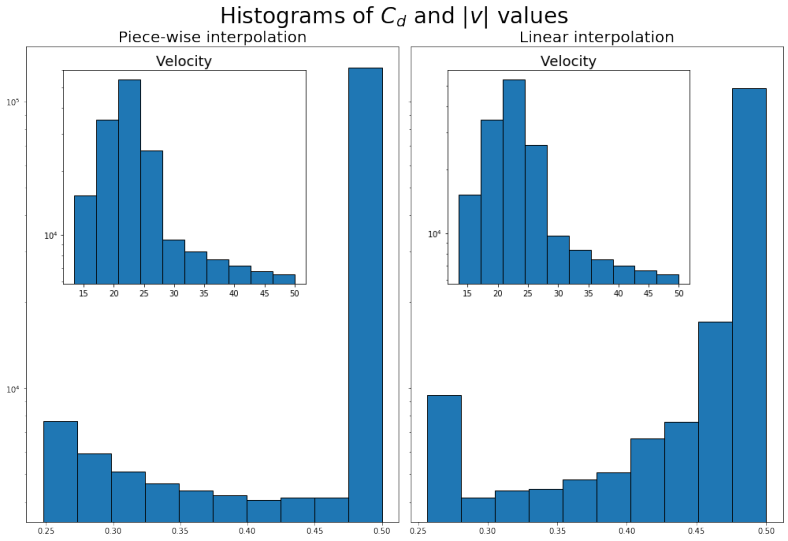

Figure 17: Histograms of C_d and $|v|$ values.

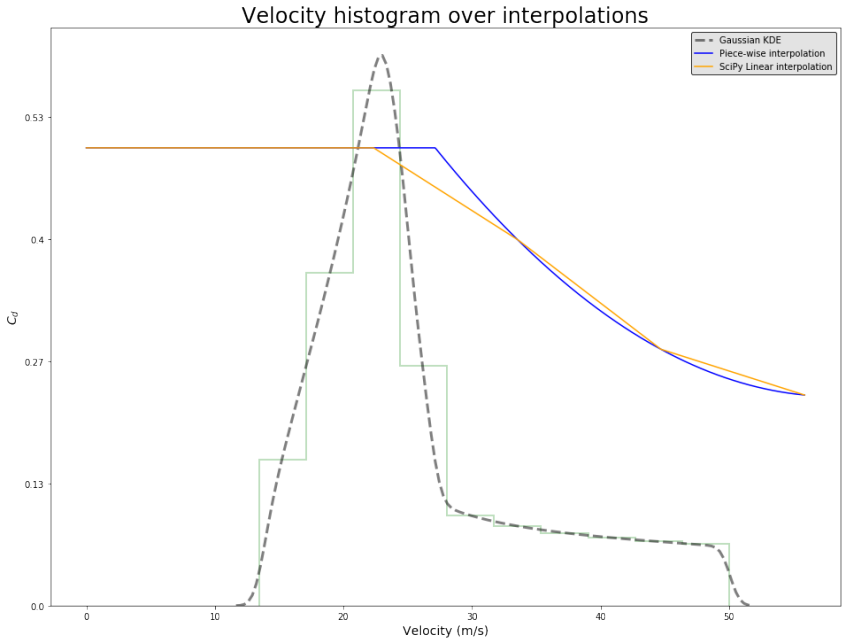

Figure 18: Velocity distributions over C_d interpolations

And finally, the distribution of velocities has been smoothed using a Gaussian KDE and superimposed over the interpolation functions to give a better intuitive understanding of how the geometric coefficient changes over the trajectory of the projectile.

Conclusion:

In this project, we have explored finite difference method implementations for ordinary differential equations. In this exploration, we began with a restricted theoretical model, neglecting the effects that movement through a fluid would have on a projectile, so that we could examine the precision of our numerical model. After computational validation, we examined the more realistic behavior of a projectile by adding fluidic drag, adding a non-zero velocity field to the fluid in which the projectile moved, and giving the projectile a drag coefficient that was dependent on the velocity of the projectile. This exploration has been illustrative of the power of numerical methods of differentiation, especially in systems that are either difficult or impossible to model analytically.

Theoretically, truncation and round-off error were explored as well as the process by which difference schemes are constructed (when in doubt, Taylor!). This is but an introduction to a very rich field of study. For instance, in more complex problems, a difference scheme of higher order can provide a more accurate model and error analysis can lead to optimization of methods, focusing on efficiency or accuracy. Error analysis will become even more important when models including partial differentials are introduced.

In terms of programming, the implementations were rather straightforward although more efficiency could be gained by vectorizing more of the data in lieu of nested iterations. This proved

to be more difficult than anticipated due to the conditional breaking included to stop calculation for heights less than zero. I'm sure that, given more time, algorithms much less computationally costly could have been constructed. Very little proved problematic once the primary method of differentiation was established; the three implementations are all slight variations on a theme. Personally, I very much enjoyed exploring plotting functions and the various methods of conveying information pertinent to the problem, both graphically and mathematically.

Appendices

A: Code

References

[1] Interpolation (scipy.interpolate) — scipy v1.4.1 reference guide. URL `https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html`.

[2] Alejando L Garcia. *Numerical methods for physics*. 2015. ISBN 9781514136683.