Projectile motion two dimensions[1]

Objectives

In this laboratory, you will write simulate the motion of a point-like, massive object in two dimensions under the influence of gravity and — in part 2 — a drag force. After completing this activity, you should be able to:

- *Simulate the motion of an object under the influences of forces.*
- *Display and format output using the Python print() function.*
- *Validate numerical simulation code by (i) comparing with known analytic results and/or (ii) looking for convergence of the numerical results.*

Simulating motion under the influence of forces

Consider an object located at position $\vec{r}_0$ moving with velocity $\vec{v}$. Suppose that we wish to find its new position $\vec{r}_1$ after a short time interval Δt. If the time interval Δt is sufficiently small, we can approximate the instantaneous velocity by the average velocity, *i.e.* $\vec{v} \approx \vec{v}_{\text{avg.}}$. We can then employ the relationship between average velocity and displacement:

$$\vec{v}_{\text{avg.}} = \frac{\Delta\vec{r}}{\Delta t} = \frac{\vec{r}_1 - \vec{r}_0}{\Delta t}.$$

Rearrangement of this relationship and replacing $\vec{v}_{\text{avg.}}$ with $\vec{v}$ yields the **position-update equation**:

$$\boxed{\vec{r}_1 = \vec{r}_0 + \vec{v}\Delta t.} \tag{1}$$

In words, Eq. (1) says that if we know an object's starting position and velocity, we can find its position a time Δt later.

Suppose that we now want to find the next position along the object's path, $\vec{r}_2$. We can simply employ the position-update equation again, this time using $\vec{r}_1$ as the starting position: $\vec{r}_2 = \vec{r}_1 + \vec{v}\Delta t$.

By repeating this process, we can ***iteratively calculate*** the path of the object, provided that we know the object's velocity at each point along the path. *This series of repetitive calculations — especially when the number of iterations required is large — is best performed on a computer.*

What if, however, the object's velocity is changing due to the application of a force? Then we must first update the velocity using Newton's Second Law before we update the position.

[1] Adapted from *VPython Introductory Computational Physics* by Ruth Chabay and Bruce Sherwood.

The main ideas involved are the following:

- ***Forces cause an object to accelerate.*** When an object experiences a net force, its velocity changes according to Newton's Second Law:

$$\vec{F}_{\text{net}} = m\vec{a} = m\frac{d\vec{v}}{dt}.$$

- An object's **momentum** $\vec{p}$ is related to its velocity by

$$\boxed{\vec{p} = m\vec{v},} \tag{2}$$

so **Newton's Second Law** can also be expressed as

$$\vec{F}_{\text{net}} = \frac{d\vec{p}}{dt}.$$

- For small time intervals (Δt), we can make the approximation

$$\vec{F}_{\text{net}} \approx \frac{\Delta\vec{p}}{\Delta t},$$

and then rearrange the 2nd Law to obtain the **momentum-update equation**

$$\boxed{\vec{p}_f = \vec{p}_i + \vec{F}_{\text{net}}\Delta t.} \tag{3}$$

Therefore, to predict an object's path over a long time interval given a known force or forces, we can use the algorithm on the following page.

It is important to remember that *Eqs. (1) and (3) are approximations*, and become more accurate as the time interval Δt becomes smaller, such that the average velocity becomes closer to the instantaneous velocity. By choosing a small time interval in your computer program, not only will you make the positions and momenta calculated from the update equations more exact, but you will also find the object's location at each step along its path, which you can use to animate the motion.

The above discussion assumes that the net force is constant throughout the object's motion. Often, however the the force on an object changes with its position (*e.g.*, the spring force) or velocity (*e.g.*, the drag force). *Changing forces can be accounted for by updating the net force at each step using the new position and velocity before applying the momentum-update equation, Eq. (3).* Therefore, to predict an object's path over a long time interval given a known set of forces, we can use the algorithm at the top of the following page.

Simulation algorithm for an *object moving under the influence of forces*

Setup

1. Create variables for the simulation parameters such as *g* and *m*.
2. Specify the initial position, $\vec{r}_0$, and momentum of the object, $\vec{p}_0$.
3. Specify the starting time of the simulation, t_0.
4. Create a sphere (*or other shape*) at the initial position.
5. Specify a time step Δt for your simulation; use a value that is small enough that the object doesn't move very far during one update.

Simulation

6. Construct a loop that steps through the simulation time in increments of Δt. In this loop:
   - Calculate $\vec{F}_{\text{net}}$ acting on the system for the current values of $\vec{r}$ and $\vec{v}$.
   - Update the object's momentum: $\vec{p}_f = \vec{p}_i + \vec{F}_{\text{net}}\Delta t$
   - Update the object's velocity: $\vec{v} = \vec{p}/m$
   - Update the object's position: $\vec{r}_f = \vec{r}_i + \vec{v}\Delta t$
   - (*If needed*) update the net force: $\vec{F}_{\text{net}} = \vec{F}_{\text{net}}(\vec{r}_f, \vec{v})$
   - Increase the simulation time by Δt.

Part 1: Motion under the influence of a constant net force

In this first activity you will simulate the motion of a projectile launched across level ground with an initial speed v_0 and initial angle θ_0 (see the figure) under only the influence of gravity — which is constant near the surface of the Earth. You will modify the provided code to print the starting values, simulation parameters, and range and time-of-flight. You will also validate your code by comparing the simulation range and time-of-flight with those from the analytic expressions derived from the kinematics equations in 2D.

1. Getting started

- Log onto GlowScript and create a new program in your QCIPU folder named "ProjectileMotion" or something similar.
- In a separate tab, go to the page https://www.glowscript.org/#/user/ruthsvandewater/folder/QCIPU2021/program/ProjectileMotion/edit
- Copy the linked code into your GlowScript editor, but **do *not* run it**. Spend *5 or 10 minutes* going through the code with your partner(s) making sure that you understand what it does before proceeding to the next step. An explanation of some of the VPython commands that are used in the code is provided at the top of the next page.
- Click **Run this program**. *Does the path of the projectile look physically reasonable?*

Some useful VPython Commands

- v = **vec**(vx, vy, vz) creates a vector $\vec{v}$ with components v_x, v_y, and v_zThe
- The components of v are accessed with the methods **v.x**, **v.y**, and **v.z**
- **mag**(v) gives the magnitude of a vector $\vec{v}$
- **hat**(v) creates a unit vector along the direction of the vector $\vec{v}$
- **ball = sphere(pos=vector(x,y,z), radius=R)** creates a sphere of radius R at the location $\vec{r} = (x, y, z)$
- The position of ball is accessed with the method **ball.pos**

- **Play around:** *What parameters in the code determine the initial speed and launch angle?* Vary these parameters and confirm that the code does what you expect.
- **Stop to think:** *Why is the path of the projectile shown as disconnected dots and instead of a curve? What parameter in the code determines the spacing between the dots?* Vary this parameter and again confirm that the code does what you expect.

2. Displaying results

- At the end of the program, add code to print out the horizontal distance traveled by the projectile (i.e., the *range* or x_f) and the time at which the projectile hits the ground (i.e., the *time-of-flight* or t_f).

Check in with an instructor before continuing.

3. Checking results

One important way to check that a numerical simulation code is working correctly is to compare it to known analytic expressions, if such closed-form solutions are available. As you saw in introductory mechanics, the motion of a projectile in two dimensions under only the influence of gravity is described by a set of kinematics equations. (For a refresher, see, *e.g.*, https://openstax.org/books/university-physics-volume-1/pages/4-3-projectile-motion.) Therefore, in this case, you can indeed validate your simulation by comparing with theoretical expectations.

- In your code (*not on pencil and paper!*), calculate the range and time of flight that that are predicted by the 2D kinematics equations *for your starting conditions*. Print these results. *If all of your code is working correctly, the simulation and predicted values should at this point be roughly similar.*
- In your code, write a **function** to calculate the percentage difference between two values,

$$\%\,\text{difference} = \frac{x_1 - x_2}{(x_1 + x_2)/2} \times 100\,\%\,,$$

and place it near the top of your code.

- Use this function to calculate the percentage differences between the simulation results and predicted values for the horizontal range and the time of flight. Print these percentage differences.
- You are now ready to ***quantitatively*** check your numerical simulation code. Using your simulation code, fill out **Table 1** for a projectile with m=10 kg. Set the initial speed and launch angle to $v_0 = 20\ \mathrm{m/s}$ and $\theta_i = 45^\circ$

Δt (s)	% difference in x_f	% difference in t_f
0.1		
0.05		
0.01		
0.005		
0.001		

Table 1. Using your simulation code, fill out the table for a projectile with m=10 kg given an initial speed and launch angle of $v_0 = 20\ \mathrm{m/s}$ and $\theta_i = 45^\circ$.

- *How small does the step size **dt** in your code have to be to obtain agreement with the analytical results to within 1%? What about to within 0.1% or 0.01%?*

You have now seen that there is an important *tradeoff* between the accuracy of your simulation and the time that it takes to perform the computation. In practice, one typically has a target precision that they wish to achieve. One then chooses simulation parameters that are sufficient to obtain the needed the precision goal, but do not unnecessarily waste computer cycles.

Check in with an instructor upon completion of Part 1.

Part 2: Motion including a resistive force

In the second activity you will simulate the motion of a projectile launched across level ground with an initial speed v_0 and initial angle θ_0 under the influence of gravity *and* a drag force that depends upon the object's current velocity. As before, you will modify the provided code to print the starting values, simulation parameters, and range and time-of-flight. Additionally, you will have to add the drag force to the appropriate locations in the code and update the net force in the simulation loop accordingly.

With the drag force present, there are no longer closed-form analytic expressions for the object's trajectory with which to compare. Therefore, you will instead validate your code by reducing the step size Δt in your simulation until the calculated range and time-of-flight no longer change within a certain amount. This incremental approach to the correct answer as the simulation step size decreases is called ***convergence***. Because, in real life, time is continuous (*i.e.*, there is no "step size"), the object's true trajectory cannot depend upon the value of Δt that you use in your computer simulation. Therefore, for simulations that proceed in discrete steps, you must *always* check that your final result is stable to the desired number of significant digits when you further decrease the step size. Checking for convergence is a *standard validation procedure* in all numerical simulations of this kind.

4. Adding a drag force

The drag force is a resistive force that opposes the motion of an object. Therefore, the direction of the drag force is ***opposite** the direction of the velocity vector*. The drag force on a reasonably-sized object (like a soccer ball) in air around sea-level is approximately proportional to the speed of the object squared. In other words,

$$\overrightarrow{F}_{\text{drag}} = -c_d v^2 \hat{v},$$

where c_d **is a measurable parameter** that depends upon the object's properties (shape, size, *etc.*)

- In your QCIPU folder, create a new program named "ProjectileWithDrag" or something similar. Copy your code from Part 1 into this new code. (*Close the tab with your code from Part 1 so the you do not accidentally edit it instead of the new program.*)
- In this new program, remove the calculation of the analytic results for the range and time of flight, as well as the percentage differences. These do not apply in the presence of a drag force. Also **reset the step size dt to 0.1 seconds.**
- In suitable places in your program, add code to calculate the drag force according the the equation above. Choose c_d=0.3 for now. *What are the units of c_d?*
- **Sanity check:** Change the starting velocity to 100 m/s and run your code. *Does the behavior of the projectile look reasonable?* If not, fix your code so that it does.

Check in with an instructor before continuing.

5. Code validation

- Add to your loop a counter that counts the number of iterations that the computer does during the simulation. Print out this number of iterations.
- Using your simulation, fill out **Table 2** on the following page for a projectile with m=10 kg and c_d=0.3. Set the initial speed and launch angle to $v_0 = 100$ m/s and $\theta_i = 45°$

Δt (s)	x_f (m)	t_f (s)	# of iterations
0.1			
0.1			
0.05			
0.01			
0.005			
0.001			
0.0005			

Table 2. Using your simulation code, fill out the table for a projectile with m=10 kg and a drag coefficient $c_{\text{drag}} = 0.3m^{-1}$ given an initial speed and launch angle of $v_0 = 100$ m/s and $\theta_i = 45°$.

- *How small does dt have to be for the simulation to converge to 2 significant digits? What about to 3 or 4 significant digits?*

Check in with an instructor upon completion of Part 2.

Part 3: An improved simulation algorithm

As you just saw in the previous section, even the simulation of a simple system such as a projectile in 2D with drag can require a very small step size (**dt**, in this case) and thousands of iterations to converge to 3- or 4-digit accuracy. As the step size is reduced and the number of iterations increases, the time it takes to run the simulation of course increases too. You can therefore imagine that, *at some point, it will simply become impractical to improve your simulation with the "brute-force" approach of decreasing the step size*.

Fortunately, however, there is another way to improve numerical computer simulations besides running longer: **you can *improve the algorithm***. Mathematicians since at least the 17th century — possibly earlier — have spent countless hours developing and improving algorithms to solve differential equations (of which Newton's Second Law is one) numerically, with computer scientists jumping in on the action more recently. The algorithm that you employed in Parts 1 and 2 of this laboratory was devised by Leonhard Euler[2] (1707–1783) and is known as **Euler's method**. In Part 3, you will program and test an improved version of

[2] https://en.wikipedia.org/wiki/Leonhard_Euler

Euler's method developed by Lewis Fry Richardson[3] (1881-1953) and known as the **Euler-Richardson method**.[4]

To improve upon Euler's method, it is useful to revisit the position and momentum-update equations,

$$\vec{r}_f = \vec{r}_i + \vec{v}\Delta t \quad \text{and} \quad \vec{p}_f = \vec{p}_i + \vec{F}_{\text{net}}\Delta t \,.$$

In these equations, $\vec{r}_i$ and $\vec{p}_i$ are the position and momentum at the *beginning* of a time interval (time t_i), while $\vec{r}_f$ and $\vec{p}_f$ are the position and momentum at the *end* of the same time interval (time $t_f = t_i + \Delta t$). The velocity and net force, however, do not have any "initial" or "final" subscripts, *so what $\vec{v}$ and $\vec{F}_{\text{net}}$ should we use to update the position and momenta?*

In parts 1 and 2 – although you may not have explicitly thought about it – you used the velocity and net force at the *beginning* of the time interval, $\vec{v}(t_i)$ and $\vec{F}_{\text{net}}(t_i, \vec{v}_i)$, in the position- and momentum-update equations. A **better approximation** for the average velocity and average net force in the interval between t_i and $t_f = t_i + \Delta t$, however, would probably be obtained by **using the velocity and and net force at the *midpoint* of the time interval, $\vec{v}_{\text{mid}}$ and $\vec{F}_{\text{net}}(t_{\text{mid}}, \vec{v}_{\text{mid}})$.** This is the essence of the Euler-Richardson method. The exact algorithm is provided at the top of the next page.

6. Improving the simulation

- In your QCIPU folder, create a new program named "ProjectileImproved" or something similar. Copy your code from Part 2 into this new code. (*Close the tab with your code from Part 2 so the you do not accidentally edit it instead of the new program.*)
- **Reset the step size dt to 0.1 seconds.** Modify the loop in your code that calculates the trajectory of the ball to employ the improved algorithm at the top of the following page.
- **Sanity check:** *If your new code is working correctly, you will likely not notice a difference in the motion of the ball when you run it. Your calculated range and time-of-flight should also be in the same ballpark as before.*
- Once you believe that the improved algorithm is working, fill out **Table 3** on the following page using the same parameters as for Table 2. (This will allow you to directly compare the two.)
- Is the Euler-Richardson algorithm from Part 3 more accurate than the Euler algorithm from Parts 1 and 2? How can you tell?

[3] https://en.wikipedia.org/wiki/Lewis_Fry_Richardson

[4] L. F. Ricardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. Philosophical Transactions of the Royal Society of London, 210:307–357, 1911

Improved algorithm for an object moving under the influence of forces

(Steps 1–5 are the same as at the top of p. 3.)

6. Construct a loop that steps through the simulation time in increments of Δt. In this loop:
   - Calculate $\vec{F}_{\text{net}}$ acting on the system for the current values of $\vec{r}$ and $\vec{v}$.
   - Use the velocity and net force at the beginning of the interval to estimate the position and velocity at the middle of the time interval:
     $$\vec{x}_{\text{mid}} = \vec{x}_i + \vec{v}_i(\Delta t/2) \text{ and } \vec{p}_{\text{mid}} = \vec{p}_i + (\vec{F}_{\text{net}})_i(\Delta t/2)$$
   - Use the position and velocity at the middle of the interval to estimate the net force at the middle of the interval: $(\vec{F}_{\text{net}})_{\text{mid}} = \vec{F}_{\text{net}}(\vec{r}_{\text{mid}}, \vec{v}_{\text{mid}})$
   - Use the velocity and net force at the middle of the interval to estimate the position and velocity at the end of the time interval:
     $$\vec{x}_{\text{f}} = \vec{x}_{\text{mid}} + \vec{v}_{\text{mid}}(\Delta t/2) \text{ and } \vec{p}_f = \vec{p}_{\text{mid}} + (\vec{F}_{\text{net}})_{\text{mid}}(\Delta t/2)$$
   - Use the position and velocity at the end of the interval to estimate the net force at the end of the interval: $(\vec{F}_{\text{net}})_f = \vec{F}_{\text{net}}(\vec{r}_f, \vec{v}_f)$
   - Increase the simulation time by Δt.

Δt (s)	x_f (m)	t_f (s)	# of iterations
0.1			
0.1			
0.05			
0.01			
0.005			
0.001			
0.0005			

Table 3. Using your improved simulation code, fill out the table for a projectile with m=10 kg and a drag coefficient $c_{\text{drag}} = 0.3m^{-1}$ given an initial speed and launch angle of $v_0 = 100$ m/s and $\theta_i = 45°$.

Check in with an instructor upon completion of Part 3.