BRYN MAWR COLLEGE

Department of Physics

Computational Module 3 – Iterative Methods

Prerequisite modules: Module 2 – Numerical Errors and Computational Speed
Estimated completion time: 3-6 hours
Learning objectives: Become familiar with iteration and a simple example of an iterative method for solving differential equations

Image credit: http://www.funnyphotos.net.au/ski-jump/

This module considers various versions of a familiar physical example – projectile motion – and how it might be studied ***numerically***; i.e., by computer. Such motion is described formally by differential equations, for which there are standard and sophisticated numerical solution techniques that will be explored in a later module. As a gentler introduction to numerical methods, in this chapter we present a simple approach to numerically solving ***ordinary differential equations***. This method will not give the most accurate results in general, but the basic idea underlies more advanced methods, and it will get you "warmed up" for the techniques you will see in later modules.

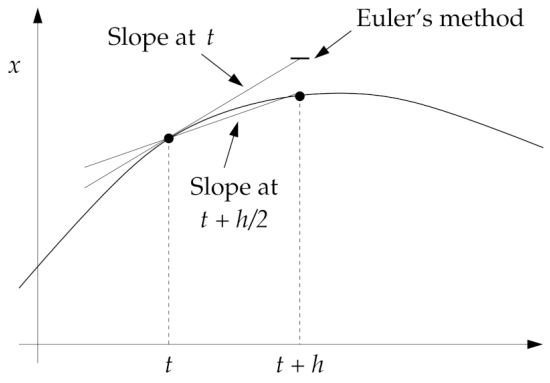

Figure 1: Euler's method for predicting $x(t)$

3.1 One-dimensional Motion Without Drag

Let's start by modeling simple one-dimensional (1-D) motion with constant acceleration, a, and without resistance. Such motion is described by two variables that are functions of time t: position $x(t)$ and velocity $v(t)$. Since velocity is rate of change of position with time, $v = dx/dt \simeq \Delta x/\Delta t$, we can write

$$v(t) \simeq \frac{x(t + \Delta t) - x(t)}{\Delta t}.$$

Rearranging this equation, we find the approximate expression (which becomes exact as $\Delta t \to \infty$)

$$x(t + \Delta t) \simeq x(t) + v(t)\Delta t. \tag{1}$$

This equation, depicted graphically in Figure (1), represents the idea behind ***Euler's method***: an approximate predicted value for x at a later time $t + h$ ($\Delta t = h$ here) can be obtained from the slope of the $x(t)$ function (i.e., v) at an earlier time t. (The line labeled "Slope at $t + h/2$" represents the key to a more accurate approximation which will be presented in a later Module.)

We also know that acceleration and velocity are related by $a \simeq \dfrac{\Delta v}{\Delta t} = \dfrac{v(t + \Delta t) - v(t)}{\Delta t}$, so we can get an approximate prediction for v similar to that for x:

$$v(t + \Delta t) \simeq v(t) + a\,\Delta t. \tag{2}$$

Breakpoint 1: The same basic approach used to project x and v forward in time could be applied to project *any* (well-behaved) function $f(x)$, where x need not represent position. Write an expression for $f(x + h)$ analogous to Eqs. (4) and (3), where h is an infinitesimal displacement.

It turns out that the accuracy of the Euler method can be significantly improved by making a small change: instead of updating the position first and the velocity second, the velocity is updated first,

and the position then is updated using the *new* velocity. That is, the equations used are:

$$v(t + \Delta t) \simeq v(t) + a\,\Delta t. \tag{3}$$

$$x(t + \Delta t) \simeq x(t) + v(t + \Delta t)\,\Delta t \tag{4}$$

These modified equations embody the ***Euler-Cromer method*** (which sometimes goes by other names). Note that this approach amounts to using the slope at the later time, rather than the one at the earlier time, to extrapolate the position at the later time. The Euler-Cromer method does a much better job of conserving energy than does the Euler method, which often causes the energy to increase steadily.

We can use Eqs. (3) and (4) to evolve the position over time, starting from some initial position and velocity. We put in the current position and velocity on the right-hand sides of the equations, compute the new values of those variables on the left-hand sides, then use those new values in the right-hand sides to get the next set of new values on the left, and so on. This approach to solving differential equations exemplifies the process of ***iteration*** – the output of one step of a process serves as the input for the next step. It also is an example of a ***finite-difference*** method of solving differential equations, since it involves small but finite increments of a key parameter (t here).

Before we present the code to compute the motion, we introduce a tool that bridges the gap between the mathematical description of a problem and the computer code that implements the solution. That tool is ***pseudocode*** – a programming language-independent description of how the code will be structured. While writing pseudocode before actually coding is not necessary for simple computational problems, it becomes increasingly useful as the complexity of the problem and the associated code grow. The problem we're considering here may not require pseudocode, but it's probably helpful to first encounter it in such a simple example.

There is no standard syntax for pseudocode, and its form can vary depending on the programming language one is modeling with it, but here's an example of the sort of pseudocode one might write to describe what we need to do to solve the problem of one-dimensional motion without drag. (Recall that lines beginning with a hash, `#`, are comments in Python.)

```
# Set up constants and initial values
initial & final times
time step
initial position
initial velocity
acceleration

while (time is less than end-time):
    update velocity
    update position
```

```
    store updated velocity
    store updated position

    increment time
    store updated time
```

The Python code below shows how to do this motion computation (using variable `y` instead of `x`) with a typical set of initial parameter values and the case of zero acceleration.

```
# Set up constant and initial values (SI units)
t = 0      # initial time
tf = 10    # final time
dt = 0.1   # time step
y = 0      # initial position, at origin
v = 50     # initial velocity, in positive-y direction
a = 0      # constant acceleration

# Create & initialize lists for variable values
time = [t]
ypos = [y]
yvel = [v]

while t <= tf - dt:
    # Update variables using Eqs. (1) & (2)
    v = v + a * dt
    y = y + v * dt

    # Add updated variables to the lists
    yvel.append(v)
    ypos.append(y)

    # Increment time
    t = t + dt
    time.append(t)
```

The top group of lines simply defines the values of constants and the initial values of variables. The middle set of lines creates and initializes lists that will be used to store the sequence of times at which the position and velocity are computed, as well as the corresponding values of those variables. The lists at first contain just the initial values of the variables. (It's possible, and sometimes desirable, to create a list without providing any initial values to it. The syntax for that is `mylist = []`.) Note that the various parameters and variables are given names that represent the corresponding quantities. This isn't necessary, but it makes the code a lot more readable than it would be if the names were unrelated to the quantities they represent (e.g., `var1`, `gfxxq`, `Dumbledore`, etc.).

The bottom section of code uses a `while` loop to update variable values. A `while` statement checks whether the indicated condition is met; if so, the loop is executed; if not, the program jumps to the first non-indented line following the `while` block. (We use a `while` loop rather than a `for` loop

here because we don't know ahead of time how many cycles through the loop we need to make, but we do know the stopping condition – the final time.) Note that the `while` condition here is written as `t <= tf - dt` (`dt` is playing the role of Δt in the earlier equations). This condition is *not* checking whether the time `t` is less than or equal to the final time `tf`, but whether it's less than or equal to the *earlier* time, `tf - dt`. That's because the `while` loop is going to calculate the variable values at the *next* time step. If the condition were `t <= tf`, then when `t` became equal to `tf` and the program entered the loop, the loop would then calculate the variable values at `tf + dt`, which would be beyond the desired range.

In the `while` loop, the new values of `y`, `v`, and `t` are added to the storage lists using the `.append()` ***method***. (The term "method" comes from object-oriented programming, which will be discussed in a later Module.) The lists can be made as long as necessary in this way.

Note that in this Euler-Cromer method, the code implements Eqs. (3) and (4) as exact, not approximate expressions.

There are several ways to run a program like this. In a Python IDE like Anaconda or Canopy, you would type or copy-and-paste the code into a `.py` file in the IDE editor, and then run it using the editor's run controls (e.g., the double green arrow button in Anaconda's Spyder IDE). To run it from the Python command line prompt, type the file name. (The directory containing the file must be in your Python "path" variable. If it's not, you will need to type the entire directory path of the file to identify it to Python; e.g. `python C:\Users\Anya\Documents\file.py`.) To run it in an IPython notebook, you would type or copy the code, or sections of it, into executable cells in the notebook, and then run the cells in order, or else run the entire notebook. There is no single protocol for how many lines of code should be placed in one notebook cell, but all the lines of a loop and all the lines of a user-defined function must be in one cell. A reasonable approach might be for contiguous code lines of similar character (e.g., those initializing parameters, or those defining lists) to be put into a single cell.

3.2 Two-dimensional Projectile Motion Without Drag

Extending the analysis of 1-D motion to two dimensions is not very difficult – we simply have to add position, velocity, and acceleration variables for the new coordinate, y. Again assuming constant acceleration, the expanded set of (approximate) kinematic equations then is

$$v_x(t + \Delta t) = v_x(t) + a_x\,\Delta t, \tag{5}$$

$$v_y(t + \Delta t) = v_y(t) + a_y\,\Delta t, \tag{6}$$

$$x(t + \Delta t) = x(t) + v_x(t + \Delta t)\Delta t, \tag{7}$$

$$y(t + \Delta t) = y(t) + v_y(t + \Delta t)\Delta t. \tag{8}$$

The extension of the earlier 1-D code to the 2-D case is straightforward and is left as an exercise.

3.3 Two-dimensional Motion with Drag

Although air drag is often considered negligible in kinematics problems, it actually plays an important role in the motion of many objects. In this section you will code in Python to solve for the position, velocity, and acceleration of a 2-D projectile motion with drag.

At very low speeds for small particles, air drag is approximately proportional to the particle's speed – this is ***linear drag***. For larger (everyday) objects traveling at higher speeds, air drag is approximately proportional to the speed squared – this case is referred to as ***quadratic drag***.

For the case of a linear drag force $\mathbf{f}_{lin}$, the corresponding acceleration of an object of mass m would have the form

$$\mathbf{a}_{lin} = \frac{\mathbf{f}_{lin}}{m} = -\frac{b\mathbf{v}}{m}, \tag{9}$$

where b is a constant that depends on the object's size and shape, and on the material through which it travels. In terms of components, which would be used in a computer program, we have

$$a_{lin,x} = -\frac{b}{m}v_x \;\; ; \quad a_{lin,y} = -\frac{b}{m}v_y. \tag{10}$$

The force $\mathbf{f}_{quad}$ due to quadratic drag, and the corresponding acceleration $\mathbf{a}_{quad}$ of an object of mass m, take the forms

$$\mathbf{a}_{quad} = \frac{1}{m}\,\mathbf{f}_{quad} = -\frac{1}{2m}\,C_D\rho A v\,\mathbf{v}, \tag{11}$$

where C_D is the coefficient of drag, which depends on the shape of the object; ρ is the density of the fluid through which the object moves (air in this case); A is the cross-sectional area of the object (perpendicular to the direction of travel); and $\mathbf{v}$ is the object's instantaneous velocity (v is its instantaneous speed).

Without air drag, an object in 2-D projectile motion has a horizontal acceleration of zero and a vertical acceleration of $-g$, so adding in (quadratic) air drag we have:

$$a_x = a_{quad,x} = -\frac{1}{2m}C_D\rho A v\,v_x, \tag{12}$$

$$a_y = a_{quad,y} - g = -\frac{1}{2m}C_D\rho A v\,v_y - g. \tag{13}$$

Recap

- The Euler-Cromer method — which "predicts" the value of a function at time $t + h$ from its value at t using a linear approximation — can be used to analyze motion in one or more dimensions. (The analysis will be accurate only if the time step h is chosen small enough.)
- Pseudocode is useful for planning the construction of complex or extensive code.

Exercises

Exercise #1

(a) Transfer the code presented in the module into an IPython notebook and run it, to model vertical motion without gravity. Predict, and then plot the y vs t data. (Remember that you will have to import the `plot` function from `matplotlib.pyplot`.) (b) Change the acceleration so that it's appropriate for vertical motion near the surface of the Earth with gravity, and predict what the plot should look like. Run the code and check your prediction. At the end of your IPython notebook discuss your results. [Tip: if you encapsulate the code in a function, you easily can feed it different values for the constants.]

Exercise #2

Suppose we want our code to model vertical motion near Earth, but not just close to its surface where g is constant. Modify the code above to work in this situation where the gravitational force on a mass m at a distance r from Earth's center is $F = GmM_E/r^2$. How does the peak height compare to the case of constant g if the initial velocity is 200 m/s? What if it's 2200 m/s? (This is 20% of escape velocity.) Is it legitimate that we usually ignore the height-dependence of g?

Exercise #3

The aim of this Exercise is to implement the equations in the section on 2-D motion without drag. (a) Write out pseudocode for this extended program. (b) Now write the code itself, and encapsulate it in a Python function to which you can pass the initial velocity components v_{xo} and v_{yo}, as well as the gravitational acceleration g, as arguments.

For the following parts of the Exercise, let $v_{xo} = 20$ m/s and $v_{yo} = 30$ m/s, and use the standard value for g. (c) Make plots of x vs. t and y vs. t. Do they look as you'd expect? (d) Plot the v_x vs. t and v_y vs. t graphs as well. Are they as expected? (e) Make a plot of y versus x, and eyeball (or compute in your code) both the height of the peak and the *range* of the motion (the horizontal distance from the launch point to the point in the descent at which the projectile returns to the launch height). Determine the same values analytically and check if they match the numerical results.

Exercise #4

Modify your Python function of Exercise #3 for the case of motion in the presence of quadratic drag. Design your function to take C_D, ρ, m, and A as additional arguments. (a) Model the motion of a ball of mass $m = 3.0$ kg and radius 0.05 m, and use the same initial velocity components and g value that you used in the previous Exercise: the aim is to compare the motions with and without drag. (Reasonable values of the constants, in MKS units, are $C_D = 0.5$, and $\rho = 1.3$ for air.) Plot graphs of x and y versus t, and y versus x. Briefly discuss how these compare with the corresponding plots for the case of no drag. Is the trajectory with drag symmetric? (b) Repeat the comparison (with and without drag) with $v_{xo} = 60$ m/s and $v_{yo} = 90$ m/s. What do you conclude from comparing the results of parts (a) and (b)? [Hint: you can easily treat the case of no drag with your current function.]

Mastery Exercise #1

Modify the kinematic code for 1-D motion to model a situation with changing mass; e.g., the case of a rocket rising vertically. Newton's second law in the form $F = dp/dt$ implies that $m\,dv/dt = V\,|dm/dt| - g$, where v is the rocket's speed, m is its mass, and V is the relative speed between the rocket and its exhaust. This latter quantity normally is incorporated in the related *specific impulse*, $\tau_s = V/g$ (with units of time). (a) Use your new program to compute the height as a function of time of a Saturn V rocket (used to launch the Apollo missions) during the firing of its first-stage engines (these rockets consisted of three stages in total). The Saturn V had a total mass at liftoff of $2,810,000$ kg. The first stage used $2,080,000$ kg of fuel (note that this is most of the mass!), with a specific impulse of 263 sec, and burned it all in 160 seconds. (b) Plot the height of the rocket as a function of time for the 160-second period, assuming that it flies straight up. (c) Also, determine the rocket's speed at the 160-sec mark. By the end of first-stage burn, the actual Saturn V rose to a height of 67 km while traveling 93 km downrange, and achieved a speed of roughly 2300 m/s. Your mileage may vary (as the saying goes) due to factors we've ignored and possibly to inaccurate numbers.

Mastery Exercise #2

Write code to compute the journey of a spacecraft (with a mass of 10^5 kg) propelled by a "solar sail" (with area $= 1$ km^2) traveling from Earth (1.5×10^8 km from the Sun) to Pluto, at 5.9×10^9 km from the Sun. (Assume the Sun, Earth, and Pluto lie along a line, and that the spacecraft leaves Earth orbit with no initial speed.) Use the following facts: (i) The *power density* (or *energy flux*) contained in solar radiation at a distance r from the Sun is $P = L/4\pi r^2$, where $L = 3.85 \times 10^{26}$ W is the Sun's *luminosity*; (ii) The pressure on a perfectly reflecting surface (the sail) due to electromagnetic radiation of power P is $p = 2P/c$, where $c = 3.0 \times 10^8$ m/s is the speed of light. Find the speed of the spacecraft as it passes Pluto, and the time for the journey. Express the speed in km/s and as a fraction of c, and express the time in days (to a precision of 0.01 days). What's your reaction to your speed result? [Hints: (i) remember to account for gravity, (ii) be careful with units, and (iii) don't use too small of a time step – this journey will require many days, so you want to choose an appropriate time step to avoid too long of a computation.]

Mastery Exercise #3

Write code to model a point particle moving in a square 2-D box with walls of length $L = 5$ and its lower-left corner based at the origin. Ignore gravity and drag, and assume that the particle reflects perfectly elastically (i.e., without change of kinetic energy) from the walls. (What does that imply for the relationships between its positions and velocities immediately before and after striking a wall?) Start the particle at the center of the box, with velocity components $v_x = 2.5$ and $v_y = 1.1$, and store its position at each time interval so that you can plot a "time-lapse" trace of its motion (using circle markers). Let the program run for 4 time units, but use incremental time steps much smaller than 1 unit. Write pseudocode for this problem before you start to code it. [Hint: you'll need to use `if` statements to deal with the walls.]

Breakpoint Answers

Breakpoint 1: $f(x+h) \simeq f(x) + h\,\dfrac{df(x)}{dx}$.

Scientist Profile

Jean Bartik was born (as Betty Jean Jennings) in Gentry County, Missouri, in 1924. She received a B.S. in mathematics from Northwest Missouri State Teachers College in 1945, a GED (master's) in English from the University of Pennsylvania in 1967, and an honorary D. Sc. from Northwest Missouri State University in 2002. Upon her graduation in 1945, she was hired to compute ballistic firing tables for the U.S. Army. (These tables were used by gunners to set their guns to hit their targets.) She was one of hundreds of human "computers," usually women, who calculated these tables by hand using mechanical calculators. Also in 1945, ENIAC – the first electronic computer – was completed at the University of Pennsylvania, and used for the computation of firing tables. Bartik was one of six human computers chosen to work on the new machine. She and the team learned to operate ENIAC and became its and, arguably, the world's first programmers. In 1947, Bartik became part of a group that converted ENIAC into a stored-program computer, which improved its efficiency and usefulness. Bartik also contributed to the early BINAC and UNIVAC 1 computers. [Information and image from the Computer History Museum, http://www.computerhistory.org/fellowawards/hall/bios/Jean,Bartik/. For a sense of the large number of factors involved in computing firing tables, see e.g., http://www.globalsecurity.org/military/library/policy/army/fm/6-40/Ch7.htm . For interesting information on the early mechanical and electronic calculating machines used to compute firing tables, and on the women who used them, see the Educational Materials link at http://topsecretrosies.com/ .]