Ph 20.3 – Numerical Solution of Ordinary Differential Equations

Due: Week 5 -v20170314-

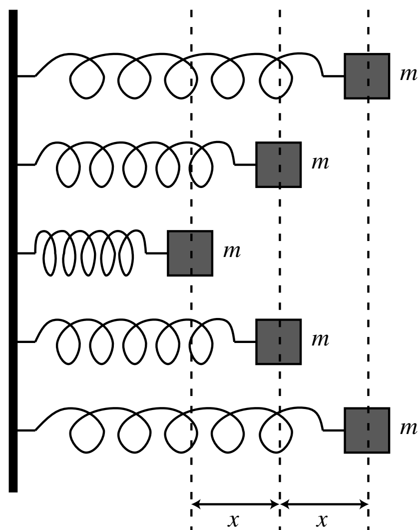
This Assignment

So far, your assignments have tried to familiarize you with the hardware and software in the Physics Computing Lab, and to introduce you to the basic *process* of computational physics. In particular you should now be familiar with the basic idea: taking some physical or mathematical system formulated in terms of equations, implementing it using Python code, and plotting the resulting output. The purpose of this assignment is to deepen your understanding of how to investigate a system once you've implemented it in code.

In the first assignment, you already saw that it was sometimes appropriate to think of a Lissajous figure as a function of time and sometimes more useful to think of it as tracing out a fixed curve in the plane. In this assignment we will explore one of the most important model systems in all of physics: the simple harmonic oscillator. In order to do this we'll implement several methods for the numerical solution of simple differential equations. If you have already studied differential equations in a course on calculus, you may want to review them for this (and future) assignments; if you haven't yet, don't worry: differential equations are just a fancy name for equations that contain derivatives. Just like in the last assignment, because we know the analytic behavior of the system we can evaluate the behavior of the three different methods we'll use. While some aspects of this behavior will be obvious just from plotting the various solutions directly (i.e., from plots of position and velocity as functions of time), we can gain a better understanding by looking at the energy as a function of time or by looking at the geometry of the *phase-space* curve of position versus velocity traced out as a function of time.

The basic coding task of this assignment is to implement equations (7), (8), and (12). These three equations only differ by what's written in their subscripts, but you'll see they all have very different behavior, so it's very important to make sure you understand how to translate them to code in order to implement them correctly. In the previous two assignments, we were able to avoid loops entirely because each point in the array could be acted on independently, and therefore "at the same time" using `numpy` routines. In this assignment this is no longer the case: e.g. the point x_{i+1} depends explicitly on the previous point x_i. So you will need to use loops. You are of course free to continue using `numpy` arrays rather than Python lists! Note that it is in general much faster for a computer to alter the values of an array of fixed size rather than to append new values to the end of an array (talk to your TA if you'd like to know more!). So if, like in this assignment, you know how long your list or array will end up being you may wish to create a dummy list/array of that size to start with using e.g. `[0]*n` or `np.zeros(n)` and then alter its values as you go.

After completing the first three assignments of Ph 20 you should feel very comfortable using Python to create, plot, and analyze numerical data. Starting with the next assignment we will step away from Python to introduce other useful scientific computing tools: makefiles, version control, and Mathematica. We will return briefly to Python in the last assignment of the term. The remaining two terms in the Ph 2x sequence will use Python exclusively: Ph 21 for data analysis, Ph 22 for numerical methods.



An ideal, horizontal spring with a mass m attached to its free end oscillates freely, displacing the mass a variable distance, x, with time.

The equilibrium position is that for which the spring is neither compressed nor extended.

The force on the mass is proportional to the displacement x of the mass, m, from its equilibrium position:

$F = ma = -kx.$

Figure 1: The implementation of a simple harmonic oscillator as a mass on a spring.

Ordinary Differential Equations in Mechanics

Newton's second law is *force* = *mass* × *acceleration*, or (for the motion of a system with one degree of freedom)

$$F = ma \left[= m\frac{dv}{dt} = m\frac{d^2x}{dt^2} \right], \tag{1}$$

where x, v, a, and t are respectively the position, velocity, accelation, and of course time. If the acceleration, a, or the velocity, v, are known explicitly as functions of time alone, we can obtain the position, x, very simply. [One would say that the problem is *reduced to quadratures*, which is a fancy way of saying that the solution can be written explicitly as an integral of known functions.] We get out our Simpson routine, and integrate. If the velocity is known explicitly,

$$x(t) = x_0 + \int_0^t v(t')\, dt'; \tag{2}$$

if the acceleration is known explicitly, first integrate

$$v(t) = v_0 + \int_0^t a(t')\, dt' \tag{3}$$

to get the velocity at a set of equally spaced times; then integrate again to get the position.

In most interesting problems, however, acceleration and velocity are not known beforehand as functions of time alone: rather, the expression for the force may involve the position and/or velocity themselves (we have already seen this in the grapefruit problem in assignment 1). Consequently, the straightforward integration of the acceleration or velocity is not a viable route. Enter numerical techniques for the integration of ordinary differential equations. We shall try out the simplest of these on a a very simple system: the *simple harmonic oscillator*, as exemplified by the case of a mass attached to a spring (Fig. 1).

Errors in numerical solutions

It is important to be aware of two types of errors that arise when we use computers to solve mathematical problems.

The first is *roundoff error*, which occurs because computers use *finite representations* of real numbers[1]. For instance, the real number $\sqrt{2}$ has a decimal representation with an infinite number of digits; however, a computer has to make do with (typically) 64 bits of information to store this number, corresponding to (roughly) 15 decimal digits. One example of roundoff error is computing $3+10^{-18}$: the computer sees only the first 15 or so digits of the number 3, so the answer to $3+10^{-18}$ will simply be 3. (There do exist arbitrary precision numerical routines that can store numbers to any number of digits required for a given accuracy; for example, section 20.6 of Numerical Recipes or some algorithms in Mathematica. However, these routines are typically complicated and are much much slower than finite representation arithmetic).

Another kind of computational error is *truncation error*, which is introduced when we replace procedures such as differentiation and integration, which are based on *limits*, with finite representations of these procedures. This all sounds very abstract, so here is an example. The derivative $f'(x)$ of a real function is defined by

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}, \tag{4}$$

and the limit (a symbolic operation) tells us what happens to the ratio on the right hand side of the equation when $h \to 0$. On a computer, we can do something like

$$\frac{df(x)}{dx} \simeq \frac{f(x+h)-f(x)}{h}, \tag{5}$$

and we choose h small, but still finite (so that you don't divide by zero!). The $\simeq$ symbol is there to point out that the equation is only approximate, because the exact definition of the derivative requires that you really take the limit as $h \to 0$. The error you make here is called *truncation error*, which is different than roundoff error. Truncation error depends not on properties of the computer memory, but on the value that you choose for h. The name 'truncation error' comes from the most usual source of such error, which is truncating an infinite Taylor series at a finite number of terms. We have already seen truncation error in action in the last assignment where we compared Simpson's and Trapezoidal methods of computing definite integrals.

Note that if h is too small, Eq. (5) will not work because of *roundoff* error (think about what happens if x is 3 and h is 10^{-18}). One of the hardest jobs in computational science is to choose parameters like h small enough that your solutions are acceptable (that is, truncation error is small enough), but large enough that your computations are still viable on the computer you have at hand (that is, roundoff error is small enough).

Integration of ordinary differential equations

Turning Eq. (5) around, we get a simple prescription to integrate numerically the first-order differential equation $df/dx = g(x)$:

$$f(x+h) \simeq f(x) + hg(x). \tag{6}$$

[1] We are here ignoring *symbolic* computation (e.g. *Mathematica*), and we are considering instead numerical computation in a language such as Python or C++

If you are given $f_0 \equiv f(x_0)$ at the initial $x = x_0$, you can apply (6) repeatedly to obtain the values $f_i \simeq f(x_0 + ih)$, for as many steps i as you wish.

Compare what we have just done with its analytic analogue: analytic integration (when it is possible) yields the symbolic representation of $f(x)$, given the initial value $f(x_0)$ and the symbolic representation of $g(x)$; numerical integration yields the *approximate* values $f_i \simeq f(x_i)$ at the discrete points x_i, given $f(x_0)$ and $g(x)$ [we are happy to work with a symbolic representation of $g(x)$, but we what we really need are only the values $g_i \equiv g(x_i)$].

Euler methods for the spring

Going back to the spring, a let us apply the method that we have just learned. For simplicity, we shall set $k/m = 1$ (where k is the spring constant and m is the mass); if you're bothered that this isn't general you can convince yourself that we can always do this by an appropriate choice of units. We want to obtain the approximate values $x_i \simeq x(t_i)$ and $v_i \simeq v(t_i)$ for $t_i = t_0 + ih$, with $i = 1, \ldots, N$ [careful with the notation, here x becomes a dependent variable, while t is the new independent variable]. We get to pick the initial conditions x_0 and v_0, and for simplicity we also set $t_0 = 0$. The straightforward application of Eq. (6) then yields the *explicit Euler method*,

$$x_{i+1} = x_i + hv_i, \quad v_{i+1} = v_i - hx_i. \tag{7}$$

Here the old position and velocity are used to compute, respectively, the new velocity and position. It should be clear that this choice is arbitrary. In fact, we could also have used an *implicit Euler method*,

$$x_{i+1} = x_i + hv_{i+1}, \quad v_{i+1} = v_i - hx_{i+1}, \tag{8}$$

where the *new* position and velocity are used to compute the new velocity and position. In this case, we need the *unknown* values x_{i+1} and v_{i+1} to update the *known* values x_i and v_i, so Eq. (8) encodes the linear system

$$\begin{pmatrix} 1 & -h \\ h & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{i+1} \\ v_{i+1} \end{pmatrix} = \begin{pmatrix} x_i \\ v_i \end{pmatrix}, \tag{9}$$

which we need to solve before we can get x_{i+1} and v_{i+1} as functions of x_i and v_i. The solution is easy to obtain in the case of the spring (and you will work it out in this week's assignment), but for many differential equations the discrete system (9) is not linear, which makes its solution difficult. This is why in practice explicit methods [generalizations of (7)] are used more often than implicit methods [generalizations of (8)].

Assignment (Part 1)

1. Write a program in Python to investigate numerically the motion of a mass on a spring, implementing the explicit Euler method. Plot x and v as functions of time for a few cycles of oscillation, choosing h small enough that the graph looks smooth, but not smaller. Choose reasonable initial conditions.

2. Work out the analytic solution to this problem for your initial conditions, and compare it to your numerical solution by plotting the *global errors* $x_{\text{analytic}}(t_i) - x_i$ and $v_{\text{analytic}}(t_i) - v_i$. Choose h so that the errors become apparent within a few cycles of oscillation.

3. Show that the *t*runcation error is proportional to h for reasonably small values of h. One way to do this is to plot the maximum value of $x_{\text{analytic}}(t_i) - x_i$ versus h for several runs

of different h integrating up to the same final time. You need only a few values of h, say $h = h_0$, $h_0/2$, $h_0/4$, $h_0/8$ and $h_0/16$.

4. Now compute the numerical evolution of the normalized total energy $E = x^2 + v^2$ (which should be conserved in this physical system), and plot E as a function of time. What is the long-range trend for E? How does it compare with the evolution of the global errors?

5. Solve the system (9) to obtain equations similar to (7) for the implicit Euler method (you need expressions for x_{i+1} and v_{i+1} in terms of x_i and v_i only). Implement the implicit Euler method numerically, and investigate how the global errors and the evolution of the energy change with respect to the explicit Euler method (to draw a fair comparison, use the same h for both methods).

6. When you are done with all this, read on.

The conservation of energy and motion in phase space

Fair enough: we had been cautioned that Eq. (6) was only approximate, and that it would introduce errors in our solutions. Still, it is troubling to find that the cherished principle of energy conservation does not hold for the *numerical* spring. A modern way to look at this kind of problems is the following. When we write the finite-difference equations (7) and (8), we do something very drastic: we take a differential equation that has many properties (including the fact that its analytic solutions conserve energy) and we replace it with a very different beast, a discrete *iterative map*. This is essentially a rule (call it F) to produce the values x_{i+1} and v_{i+1} from the values x_i and v_i, as in

$$(x_{i+1}, v_{i+1}) = F[(x_i, v_i)]; \tag{10}$$

or, equivalently, to produce the n'th values x_n and v_n from the n'th iteration of the map,

$$(x_n, v_n) = F^{(n)}[(x_0, v_0)] = \underbrace{F[F[\ldots F}_{n\,\text{times}}[(x_0, v_0)]\ldots]]. \tag{11}$$

Depending on how we generate F from the particular differential equation that we are trying to solve, the iterative map can have very different properties from the original analytic equation. For instance, in the case of the spring, the explicit and implicit Euler methods generate iterative maps that do not conserve energy (that is, they are *dissipative*).

There is a branch of numerical analysis (known as *backward error analysis*) that studies the behavior of numerical methods for differential equations by building the iterative maps, and then finding the analytic differential equations (typically modifications of the original ones) that the iterative maps are *actually* solving exactly. Because error has to creep in somewhere (we never said computers are perfect!), you will seldom be able to find a map that yields exact solutions for the very equation you want; but you can try to arrange things so that the modified equation that you are really solving is not so different, at least with regard to its crucial properties.

So what are the crucial properties that physicists are mostly concerned with? This is a hard question! We can give a general answer for a very broad class of physical systems of interest to physicists, those described by *Hamilton's equations*. You will meet the Hamiltonian formalism later in the course of your studies; suffice it to say now that the dynamics of these systems can be derived from a *variational principle*, which states that the physical solutions are found to be the functions that extremize certain physical quantities, such as the the time integral of the total energy. The simple harmonic oscillator is one such system.

It is especially instructive to study the behavior of Hamiltonian systems in *phase space*, whose dimensions are (roughly speaking) all the generalized[2] positions of the system and all the corresponding generalized velocities. For the one-dimensional spring, phase space is just the (x, v) plane. Any imaginable evolution of the spring appears as a trajectory $(x(t), v(t))$ in phase space, but the actual physical solutions are all circles, defined by $x^2 + v^2 = E$; that is, the physical trajectories trace out the curves of constant energy!

Symplecticity (that is, *conservation of volume in phase space*) is a characteristic property of Hamiltonian systems. What does it mean to conserve volume (or, for the spring, area) in phase space? Suppose we are given a group of springs with slightly different initial conditions, so that at the time t_0 they occupy the region S_0 in phase space almost uniformly; at the later time t_1, the springs will have all evolved to occupy a different region S_1 in phase space, but the area of S_1 will be the same as the original area of S_0. In the numerical arena, one can build *symplectic numerical integrators* that, indeed, conserve area in phase space. In doing so, they produce solutions that, at least qualitatively, look more physical than those produced by nonsymplectic integrators (for instance, *on the average* they conserve energy). A very simple symplectic integrator is a cross (of sorts) between the explicit and implicit Euler methods:

$$x_{i+1} = x_i + h v_i, \quad v_{i+1} = v_i - h x_{i+1}; \tag{12}$$

this is the *symplectic Euler method*. Backward error analysis would show that (12) solves exactly a modified analytic system that is (first of all) Hamiltonian, and (second) much closer to original system.

This is the reason why symplectic integrators are often used for Hamiltonian systems. They are not, however, the solution to all evils, because of what I call the 'conservation of pain' or that is sometimes also called the *blanket principle* (the blanket is always too short to cover both your feet and your head). Error will spare the symplectic structure of phase space, but it must reappear elsewhere: for instance, symplectic integrators might preserve better the amplitude of the spring oscillations, which appears in the expression for the energy; but the precise position of the mass within each oscillation (i.e., its *phase*) will accumulate a growing error with respect to the exact solution.

Assignment (Part 2)

1. Investigate the phase-space geometry of the trajectories produced by the explicit and implicit Euler methods. Set h so that deviations from closed circles (i.e. the analytic solutions, which you should also plot) are obvious.

2. Implement the symplectic Euler method in Python and investigate the phase-space geometry of the trajectories that it produces. Compare them with the trajectories obtained at the previous point, for the same h. This is your Ph20 Beautiful Plot™ of the week.

3. Study the evolution of total energy obtained with the symplectic Euler method. Set h so that deviations from the constant value of the analytic solution are obvious. What do they look like? How does this evolution relate to what you just saw in phase space?

4. ⋆ (optional) Examine the long-term evolution of the global error in the phase for trajectories produced by the symplectic Euler method, as compared to the exact solution. To do so, plot

[2] In the Hamiltonian formalism it is often useful to use position-like variables that do not correspond directly to measurements of distance.

the symplectic-Euler solution on top of the exact solution, and see if the oscillations develop a lag. Many oscillations may be needed for the error to grow appreciably; plot only the last few (of many) to estimate the lag visually.

If this assignment was interesting to you, consider taking Ph 22, which takes the knowledge obtained from this assignment as a starting point and builds up to full N-body simulations of gravitationally-interacting objects.