Physics 305: Ch. 1

Solving ODEs I: Introduction to Differential Equations

We now begin our treatment of differential equations. In this first part we present a very brief, and somewhat idiosyncratic, introduction to obtaining a numerical solution to one sort of differential equation. See any basic text on Numerical Methods for more details.

In some simple physics problems, we are able to describe the state of the system by a function of the unknowns. For example, the vertical motion of a projectile under constant gravity can be written as $y = y_0 + v_0 t + gt^2/2$. The number of physically relevant "linear second-order homogeneous differential equations with constant coefficients" is very limited. You will find that nearly all the interesting equations are either trivial, or impossibly difficult to solve analytically. Traditional methods solve the trivial cases, but the difficult cases require numerical solutions.

We are often in the situation that we can easily describe the rate of change of a certain quantity in terms of the immediate state of the system, yet we cannot easily determine the state itself at all times. This is a "differential equation", as the differential of a function appears in the equation along with (perhaps) the function itself.

A simple example of a system that can be shown not to have a closed-form solution is that of the motions of three or more bodies due to their mutual gravitational attractions—the famous "three-body problem". It is quite simple to write down the accelerations, the second derivatives of the bodies' positions, in terms of those positions, but we cannot in any general way write down the positions themselves as a function of time. We can, however, examine the behavior of such a system in some detail by using numerical methods. We will tackle the three-body problem later in the semester.

Differential equations appear everywhere in physics. You are already quite familiar with at least one, $F = ma$, and will learn many others as you continue to study physics. In many ways, differential equations are the language of physics, as they express the rules that govern the system on a spatially and temporally local level. It

is important to learn how powerful computers can be in solving them!

1.1 Differential Equations

Consider a function $f(t)$ of a single independent variable t. A differential equation is an equality involving one or more derivatives of the function along with any combination of the function itself and the independent variable. This equality holds at *all* allowed values of the independent variable. Some examples include

$$\frac{df(t)}{dt} = t^2 \tag{1.1}$$

$$\frac{df(t)}{dt} = f(t)t^2 \tag{1.2}$$

$$\frac{df(t)}{dt} + f(t)^3 = f(t)t^2 \tag{1.3}$$

$$\frac{d^2f(t)}{dt^2} = f(t)t^2 \tag{1.4}$$

$$\frac{d^2f(t)}{dt^2} + t^3\frac{df(t)}{dt} = f(t)t^2 \tag{1.5}$$

The idea of solving the differential equation means that we are to find the function $f(t)$ that satisfies the equality.

Some nomenclature: A differential equation that describes a function of a single independent variable is called an ordinary differential equation (ODE). Those that describe functions of several variables are called partial differential equations (PDE). Both kinds are common in physics. The "order" of a differential equation is the number of the highest derivative in the equation. The first three examples above are called "first-order" differential equations because only the first derivative is involved. The last two examples are called "second-order" differential equations, because they contain second derivatives.

On the one hand, differential equations are deceptively complicated. Normally, a single equation signals that we seek the value of a single variable. Here, we instead must find an infinite number of values, namely the value of $f(t)$ at every value t. In a sense, the differential equation supplies with an infinite number of our normal equations (one for each value of t) that we must in turn solve simultaneously for our infinite variables.

However, differential equations often yield fairly simple solutions, essentially because the framework of differential and integral calculus is so powerful. Indeed, you will often find that a differential equation is simply an integral in disguise. For example, consider our first example

$$\frac{df(t)}{dt} = t^2 \tag{1.6}$$

You know the solution: $f(t) = t^3/3 + C$, where C is any numerical constant. However, you can also see that if we seek the function f whose derivative is some function $g(t)$, then the sought-after function is simply the integral of g, i.e.

$$f(t) = \int g(t')\, dt'. \tag{1.7}$$

The constant C above is just our familiar constant of integration. This link between differential equations and integral calculus is so ubiquitous that the terms "solving a differential equation" and "integrating a differential equation" have become synonymous.

The constant of integration that appears above is recurring theme. The solution to a differential equation is *never* unique, but instead always involves one or more constants of integration (one for each order of the differential equation). The value of these constants must be set by specifying some additional piece of information, such as the value of the function at some initial time. This extra data are called the "boundary conditions".

There are a great many techniques and tricks that can be used to solve differential equations or at least to restructure them into indefinite integrals. *This is not our goal here.* Instead, we will study how to solve differential equations (particularly ODEs) numerically. In contrast to all the manipulations that arise in solving an ODE analytically, numerical methods are intuitive: if we have been told how the derivative of the function depends on the value of the function and/or the independent variables, we simply step along adding up all of the derivatives to reconstruct the function itself!

1.2 A Physics Example

Consider the following simple physical system, which in fact does possess a closed-form solution. Many nuclei are unstable; in other words, after some time, they can be expected to undergo a reaction in which one or more particles are emitted or absorbed and the nucleus changes form. A typical example is the radioactive isotope ^{56}Ni. Here, the "Ni" means the nucleus has 28 protons and the "56" means that it has 56 nucleons, 28 protons and 28 neutrons. The mean lifetime of this configuration is about 8.8 days, after which an electron is absorbed into the nucleus (if any are available), converting one of the protons into a neutron and the nucleus into ^{56}Co (27 protons and 29 neutrons), and emitting several gamma ray photons as the nucleus relaxes into its ground state. Because this is a quantum mechanical system, all we can say is that the *mean* lifetime is $\tau_{Ni} = 8.8$ days. Some nuclei will decay sooner than this, and some later.

If we take a large number of these nuclei, say a gram of freshly made ^{56}Ni, then on average e^{-1} of the nuclei will have decayed into ^{56}Co after 8.8 days. The *rate* of decay is thus quite simple; it is just the fraction of ^{56}Ni nuclei present divided by the

mean lifetime:

$$\frac{df_{Ni}(t)}{dt} = -\frac{f_{Ni}(t)}{\tau_{Ni}} \tag{1.8}$$

where time is measured in days. We may not know how much of the sample is still ^{56}Ni, but we do know that the rate at which it is decaying is proportional to the amount present.

We can get a closed-form solution in this case quite directly. Rewrite the above as

$$\frac{df_{Ni}(t)}{f_{Ni}(t)} = -\frac{dt}{\tau_{Ni}}. \tag{1.9}$$

Now integrate both sides to obtain

$$\ln(f_{Ni}(t)) = -\frac{t}{\tau_{Ni}} + C, \tag{1.10}$$

where C is some constant of integration. Taking the exponential of both sides, we have

$$f_{Ni}(t) = Ce^{-t/\tau_{Ni}}. \tag{1.11}$$

Now we must determine the value of the constant C. We do know that at time $t = 0$ everything was ^{56}Ni; i.e. $f_{Ni}(0) = 1$. Plugging in $t = 0$, the value of C must be thus be 1. Thus, the solution is

$$f_{Ni}(t) = e^{-t/\tau_{Ni}}. \tag{1.12}$$

This is typical of differential equations. We need to know not only the equation, but also something of the solution at a particular point. These are known generally as *boundary conditions*; in this case the boundary condition is a particular case known as an *initial value* and the whole system as an *initial value problem*. Knowing the initial conditions and the way in which the system evolves, we can determine the state of the system for all time.

More often than not, we either can't solve or don't want to solve differential equations analytically Instead, we can use numerical methods to determine a solution.

1.3 It's just an integral

First let's make the connection with what we have done before. We begin with a first order differential equation, which can be written as

$$\frac{df(t)}{dt} = \mathcal{H}\left(t, f(t)\right) \tag{1.13}$$

($\mathcal{H}$ is some function that can depend on both $f(t)$ and t itself.) Besides the equation, we know the value of f at the starting time, $f(t_0)$.

Here is a formal solution:

$$f(t) = f(t_0) + \int_{t_0}^{t} \mathcal{H}(t', f(t'))\, dt' \tag{1.14}$$

Think about this carefully — remember that if you differentiate an integral with respect to the upper limit of integration you get the integrand evaluated at the upper limit. So, differentiate both sides of Eq. 1.14 and you get Eq. 1.13. Since the lower limit of the integral is t_0 you can see that $f(t)$ starts off properly at $t = t_0$.

So, to solve a first order differential equation, we just have to do an integral. And, if we can't do the integral analytically, we can just use the methods we learned for approximate numerical integration: left-hand rule, midpoint rule, etc.

Of course, if it were really just that simple we would not be starting a new chapter. The difficulty is that $\mathcal{H}(t', f(t'))$ depends of $f(t')$, and $f(t')$ is the thing you are trying to solve for. So you can't evaluate the integrand, and hence (apparently) can't do the integral until you have already solved the problem.

(As you know, you are welcome to try to guess an answer for $f(t)$, and plug it in to Eq. 1.13 to see if it works. Such guessing is one of many time-honored methods for finding exact solutions.)

1.4 Euler's method (the left-hand rule)

Ignoring the apparent problem of not knowing the integrand, let's begin doing the integral in Eq. 1.14 using the left-hand (or piecewise constant) method. Since in this example our variable is time, call the size of the interval (or bin) in this integration the "timestep". To do the first interval in the left-hand rule integration, we just take the function at the left, or $f(t_0)$ and multiply it by the timestep h. This works, because we know $f(t_0)$ — it's the initial condition.

This has approximately given us $f(t_0 + h)$:

$$f(t_0 + h) \approx f(t_0) + h\mathcal{H}(t_0, f(t_0)) \tag{1.15}$$

One way to look at this is that we found $f(t_0 + h)$ from a numerical integral with one interval. Another way to look at this is as the first order of a Taylor expansion of $f(t_0 + h)$ around t_0, and this is the starting point for the analysis of accuracy that we will do shortly. But the most important way of looking at this is the intuitive way —- we know f at the starting point, and we know how fast it is changing, $\left.\frac{df(t)}{dt}\right|_{t_0}$. Thus, we know (approximately) what $f(t)$ will be a short time later.

Now that we know $f(t_0 + h)$, we can evaluate its derivative $\mathcal{H}$ at time $t_0 + h$), and we can calculate the second interval in the left-hand rule integration

$$f(t_0 + 2h) \approx f(t_0 + h) + h\mathcal{H}(t_0 + h, f(t_0 + h)) \tag{1.16}$$

Now simply continue stepping along until you have reached the desired ending time. This is called "Euler's method" (sometimes the "backward Euler's method").

An outline (not algorithm) for this procedure looks like:

1. must have the initial values $f(t_0)$
2. determine the size of the timestep h (or Δt)
3. set f to the initial value $f = f(t_0)$ and t to the initial time, t_0.
4. find the value of the derivative $df/dt = \mathcal{H}(f(t_0), t_0)$
5. get the value of f at a time Δt in the future, $f = f + df/dt \times \Delta t$.
6. update the value of the time $t = t + \Delta t$
7. write out the current time and the current value of f
8. go back to step 3 and do it again as often as you like.

Being very sketchy, code to implement this looks something like:

```
// set up everything first, including initial values for y(t)
for( t=t_start; t<t_end+eps/2.0; t+= eps ){
    y_new = y + eps * H(y,t);
    // Here H(y,t) is a function which computes the derivative
    y = y_new;
}
```

To find out how accurate this procedure is, begin with the same analysis we used for the left-hand rule in numerical integration. Equation 1.15 for the first step in the process is the first part of a Taylor series for $f(t_0 + h)$:

$$f(t_0 + h) = f(t_0) + h \left.\frac{df}{dt}\right|_{t_0} + \frac{h^2}{2} \left.\frac{d^2 f}{dt^2}\right|_{t_0} + \dots \tag{1.17}$$

Just as in the left-hand rule for an ordinary integral, we have kept the term proportional to h, and dropped the terms proportional to h^2 and higher. So, just as before, we make an error of order h^2 in each interval, or timestep. As before, to get to the desired finishing time t_f, we need order h^{-1} timesteps ($N_{steps} = (t_f - t_0)/h$), so these errors will add up to a total error of order h.

The story will be similar in subsequent timesteps, except for one complication — instead of being able to evaluate $\left.\frac{df}{dt}\right|_{t_i}$ exactly, we can only do it approximately, since for all t_i other than t_0 we only know $f(t_i)$ approximately. Will this spoil things? No — we have made errors that added up to order h in evaluating t_i. This means that the derivative $\left.\frac{df}{dt}\right|_{t_i}$ will be wrong by order h. But, in each step of the Euler algorithm,

$$f(t_i + h) \approx f(t_i) + \mathbf{h}\mathcal{H}(t_i, f(t_0)) \tag{1.18}$$

this quantity gets multiplied by another factor of h, so it just amounts to an additional error of order h^2 in going from time t_i to time t_{i+1}. We no longer have a simple expression for the leading part of the error in terms of $\frac{d^2 f}{dt^2}$, but we know that the total accumulated error will still be order h. Same song and dance as before — if you make the step size one third as big (using three times as many steps), the error will go down by about a factor of three, etc.

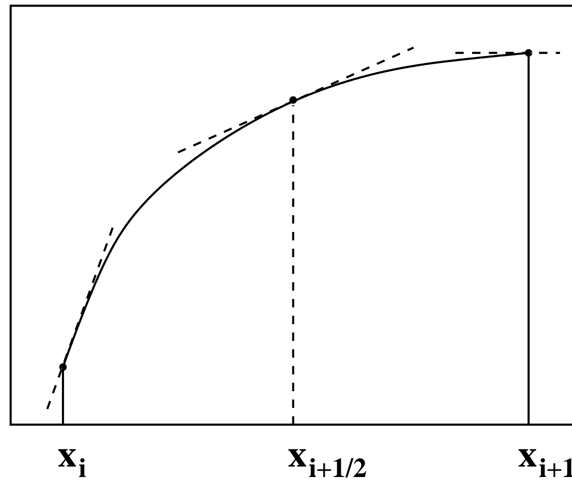

Figure 1.1: The second-order Runge-Kutta scheme uses the slope of the function at the midpoint to integrate from x_i to x_{i+1}.

1.5 Second order Runge-Kutta method (the midpoint rule)

In the last section we learned how to use the left-hand rule for numerical integration to approximately solve Eq. 1.14, realizing that we can compute $f(t')$ as we go, and thus evaluate the integrand. The next natural question is whether we can use the midpoint, trapezoidal, or maybe even Simpson's rule

Now, we quickly learned that the midpoint rule is a much better way to do a numerical integral than the left hand rule. The "second order Runge-Kutta method" is essentially using the midpoint rule to do the integral in Eq. 1.14. The problem is that to use the midpoint rule, we need the integrand at the midpoint of the first time step, and it looks like we don't know that until we have already solved the problem. To handle this, we use the Euler algorithm, which only needs values at the initial time (left side of the bin) to get an approximate $f(t)$ at the midpoint. Then, we use this to do a midpoint rule integration.

Again, the process for one step is just an left-hand rule (Euler method) half step, followed by a midpoint rule whole step:

$$\begin{aligned} f_{i+1/2} &= f_i + \frac{h}{2}\mathcal{H}(x_i, f_i) \\ f_{i+1} &= f_i + h\mathcal{H}(x_{i+1/2}, f_{i+1/2}) + O(h^3). \end{aligned} \tag{1.19}$$

Figure 1.1 illustrates this.

A code fragment for one step of the second order Runge-Kutta method would look something like:

```
// as always, I leave it to you to set up everything
```

```
for( t=t_start; t<t_end+h/2.0; t+= h ){
    y_half = y + 0.5*h * H(y,t); // Euler method computation
                                   // of y at the midpoint of the interval
    t_half = t+0.5*h;            // this is a little bit pedantic, but...

        // Now use the estimated values at the midpoint to do the real step:
    y_new = y + h * H(y_half,t_half);   // step forward by a full "h"
    y = y_new;
    // remember that the "for loop" we are in automatically updates t here
}
```

What about the errors here? First, the analysis we did for the midpoint rule integration (Taylor expand around the **midpoint** of the interval, keep one more term in the Taylor series than we kept in the Euler method discussion, discover that you make an error of order h^3 in each step, $\ldots$) is just the same here. But an additional error has been introduced because $f(t)$ at the midpoint of the interval wasn't computed very well. How much does this hurt us? Looking back at the discussion of the Euler method, we see that f at the midpoint is wrong at order h^2, and hence the derivative $H(f(t),t)$ at the midpoint will have an error of order h^2. But, this gets multiplied by another factor of h for the bin width, and so it is just another error of order h^3 **in each step**. And, as always, when integrating to a fixed final time, this accumulates to an error of order h^2.

Programming warning: In the code sketch for the second order Runga-Kutta method we wrote:

```
    y_new = y + h * H(y_half,t_half);   // step forward by a full "h"
    y = y_new;
```

Of course, this simpler fragment does exactly the same thing:

```
    y = y + h * H(y_half,t_half);   // step forward by a full "h"
```

We wrote the first form because we are being extra careful here. In programming these algorithms, it is very important to keep track of where in your code you are overwriting the old values of a variable with the new values. We will soon be doing problems where we have more than one variable (velocity and position for second order equations, or $x(t)$ and $y(t)$ for motion in two dimensions.) It is very easy to replace the old value of one of these variables by its new value before you are actually finished using the old value. We will discuss this more in later notes, but it is not too early to start giving fair **WARNINGS**. When you have more than one variable, think of the system as a unified whole — it is probably not time to overwrite one variable until you are ready to overwrite all of them. Other than that, there is no substitute for clear thinking and **TESTING**.

1.6 The predictor-corrector method (trapezoid rule)

Now you might be wondering about an analogue to the trapezoid rule method for doing a numerical integral. It works much like the second order Runge-Kutta method, except that we use the Euler method to get an approximate value at the end of the time step. This is called the "predictor". Then we use this, together with the value at the start of the interval (which we already know), to get an improved ("corrected") value by what is basically a trapezoid rule integration. Roughly:

```
// as always, I leave it to you to set up everything
for( t=t_start; t<t_end+eps/2.0; t+= eps ){
    y_pred = y + eps * H(y,t); // Euler method computation - full time step now
    t_new = t+eps;            // this is a little bit pedantic, but...
        // Now use the estimated value at the beginning and end to do the real step
        // "trapezoid rule" - use average of values at beginning and end
    y_new = y + eps * 0.5*( H(y,t) + H(y_pred,t_new) );
    y = y_new;
    // remember that the "for loop" we are in automatically updates t here
}
```

I leave it to you to convince yourself that this is also a second order method. That is, you expect errors in your final answer of order h^2.

1.7 The fourth order Runge-Kutta method

When we discussed numerical integration we observed that if we took a clever average: (2/3)*(midpoint rule) + (1/3)*(trapezoid rule), we would get a really slick integration algorithm called Simpson's rule. Can we do the same thing with numerical integration? That is, can we combine lower order estimates of the derivative at the middle and end of the time interval to get an algorithm where the largest errors in the second order Runge-Kutta and the predictor-corrector method cancel out? Yes, we can. But you have to be a little careful because, in addition to the errors we discussed in the assignments on numerical integration, we have additional errors coming from the fact that we are evaluating our derivatives using not quite right estimates of f at the midpoint and end of the interval. The resulting analogue of Simpson's rule is the "fourth order Runge-Kutta method", after C. Runge (1856-1927) and M.W. Kutta (1867-1944). You can read about it in any numerical analysis book, and it is an amazingly good way to accurately solve a **well behaved** first order differential equation. The method is[1].

$$k_1 = h\mathcal{H}(x_i, f_i)$$

$$
\begin{aligned}
k_2 &= h\mathcal{H}(x_{i+1/2}, f_i + \frac{k_1}{2}) \\
k_3 &= h\mathcal{H}(x_{i+1/2}, f_i + \frac{k_2}{2}) \\
k_4 &= h\mathcal{H}(x_{i+1}, f_i + k_3) \\
f_{i+1} &= f_i + \frac{k_1}{6} + \frac{2}{3}\left(\frac{k_2 + k_3}{2}\right) + \frac{k_4}{6} + O(h^5).
\end{aligned}
\tag{1.20}
$$

You can recognize the weights from Simpson's rule for one integration bin — 1/6 at the endpoints and 2/3 at the midpoint. The extra complication comes because we now need the integrand at the midpoint and endpoint of the timestep to better accuracy than we did for the second order methods.

We will stick to second order methods in this class, but someday you may need to know about these more sophisticated methods.

1.8 The last step problem

In the above discussion of accuracy, it was implicitly assumed that the problem was of the form "what is the value of $f(t)$ at some ending time t_f? Sometimes, the question is a little bit different —- "At what time does the function reach a particular value? For example, in the coffee cooling problem in this week's assignment, we first ask "What is the temperature of the coffee at a given time"? But a very practical question might be "At what time does the coffee reach a temperature of 50° C"? Your first attempt to answer this might involve writing a program to use the second order Runge-Kutta method to follow the temperature of the coffee. Then, after each time step check the temperature and when you get a value below 50°, print out t_i as the answer. But this is not good enough. What you really know is that the temperature passed through 50° sometime during the last time interval of your integration. This is an error of order h, or first order in the timestep. There is little point in using an integration algorithm which has error of order h^2 it you are going to throw it all away at the end by making a much larger error!

One way to handle this is to use the fact that we know the derivative of $f(t)$. So, if at time t_i the temperature is T_i, with T_i just a little bit less than 50°, you can estimate when the temperature passed through 50,

$$
t_{50} \approx t_i - (50 - T_i) / \left(\left. \frac{-\partial T(t)}{\partial t} \right|_{t_i} \right) \tag{1.21}
$$

The logic here is pretty simple: if at time 100.0 seconds a car traveling 20 meters per second is two meters past the finish line, at what time did the car cross the finish line? Clearly a good approximation is 100.0 seconds minus (2 meters)/(20 meters/sec) = 99.8 seconds. You should be able to convince yourself that the error in this estimate

of the finishing time is order h^2. (Of course, if you use the RK4 algorithm and want to keep fourth order accuracy, you will have to work even harder.)

Experience indicates that many beginning numerical analysts will get this wrong the first time they try to program it. You need to think carefully about signs. Also, you need to think carefully about whatever kind of `for` or `while` loop you use to program it. When you exit the loop, is the time variable set to the beginning of the last step, or the end of the last step? Should your estimated ending time be larger or smaller that the time you exited the loop? Signs matter! For example, in Eq. 1.21 we tried to keep what is going on clear by inserting minus signs so that each of the factors is positive. When you are debugging your program, print all these things out and **check for sanity**.

Bibliography

[1] Here quoted from "Numerical Recipes . . . ", W.H. Press *et al.*, Cambridge University Press, 1988. This is a wonderful desk reference for working scientists.