

Physics 305, Fall 2009
Assignment 5
Due Monday, September 28, 10:00 pm

Imagine that you have poured yourself a cup of coffee. Initially it is at temperature T_0 . The room is at temperature T_r ($T_r < T_0$), and the coffee gives up heat to its surroundings. The coffee is initially at 100.0 C and the room is a comfortable 24.00 C. Being easily distracted, you sit and talk for 15.00 minutes. How cold is the coffee?

We will treat the coffee as having a single temperature throughout the cup. A simple model for the cooling, sometimes called “Newton’s law of cooling”, is that the rate at which the coffee cools is proportional to the temperature difference between the coffee and its environment, so we have

$$\frac{dT}{dt} = -K(T - T_r), \quad (1)$$

where T is the temperature of the coffee and K is a constant which characterizes the rate at which heat is transported to the air. As a physics aside, we note that the parameter K depends on many details of the system—the geometry of the cup, the air currents in the room, the convective properties of air, etc.—and can therefore be very difficult to calculate! We will use $K = 0.1 \text{ min}^{-1}$.

You should be able to solve this differential equation analytically, to find the correct answer to compare to your numerical solution.

1) Write a program to solve the differential equation using the backward Euler method. Your program should print the temperature of the coffee every minute, up to 15 minutes, and print the error relative to the analytic solution.

Allow the user to enter the size of the time step and experiment with the time step so that you achieve an accuracy after 15 minutes of better than 0.001 degree. You should demonstrate that accuracy by showing that doubling the time step changes the answer by an amount smaller than 0.001 degree. Report that time step and the number of total steps required. Also, use either a log-log plot or an appropriate table to demonstrate that your error is scaling in the expected way.

You will want to use double precision for this code.

As usual, you should separate the details of the equation (1) from the differential equation algorithm. In other words, your code should use a separate function to evaluate the derivative. It is important that the derivative is in general a function of two variables, time and temperature, even though in this case we have made it time independent. Your function should include both variables so that it is ready to be generalized.

It is also good practice to use a function to actually do the evolution from time $t = t_1$ to $t = t_2$, using a given time step and a given starting temperature, returning the value of the temperature at that time. This keeps your code more modular.

You'll want to be careful that your final step doesn't actually take you beyond the end time of fifteen minutes. This is a common problem and will create an error of order your time step even if you are using a more accurate method! One way to fix this is to have your time step divide into the total time an integral number of times and then be really sure that you are taking that number of steps. A better way is to adjust the size of your final step so that you end exactly at the desired time.

Be careful with your units! The computer doesn't know that one variable is supposed to be in seconds and the next in minutes; all it stores in the numerical values. A good general strategy is to pick a set of units (i.e. all times will be measured in seconds, all lengths in meters, etc.) that are appropriate to the problem and then be sure that all dimensionful variables conform to this convention. Note that this can require one to convert physical constants; for example, if an astronomer chooses the measure space and time in parsecs and years, the speed of light is *not* 3×10^8 !

2) Repeat the previous exercise using the second-order Runge-Kutta method. This should be as simple as a minor modification in the evolution of the ODE. Again, report the time step and number of total steps needed to achieve 0.001 deg accuracy. Again, use either a log-log plot or an appropriate table to demonstrate that your error is scaling in the expected way.

3) Modify your Runge-Kutta code to answer the question of when the coffee has cooled to 60 degrees. You do this by testing in each step whether the step has crossed the 60 degree mark and then figuring out when inside the step it did so! Note that it is not enough to just take the time at the end of the step; that would make an error of order Δt . See the notes for how to do better.

What time step is needed in order to achieve a time accuracy of 0.01 second? Demonstrate that accuracy by investigating the difference in the answer between two different time steps.

4) Now take either of your codes from part 1 or 2, but preferably the Runge-Kutta code, and modify it to draw a graph using PhilsPlot. In other words, the graph window should appear when you run the program and then fill in the values as the ODE is solved.

Instructions on PhilsPlot are in this chapter and in the documentation packet. Note that your graph should have axis labels and numbering.

You may find it useful to put the code for the intermediate printing and plotting in a separate function, so that it can be called in multiple places within your ODE solving loop. That keeps the Philsplot routines from cluttering up your ODE solver.

You are free to make the lines either by connecting up the line segments or by simply having a reasonably dense set of points. However, one thing you want to avoid is having many thousands of line segments or points. It is ok to use a finer time step, but if you do, you should only plot a fraction of them. One way to do this is to keep a target time of the next time you want to print, e.g.,

```
double next_print_time, print_interval;
....
if (time>next_print_time) {
    // Plot this step by whatever mechanism
    next_print_time += print_interval;
}
```

You probably only need about 100 line segments to make the graph look good.

Once you have that code working, experiment with PhilsPlot. For example, try making the plot reveal itself over a few seconds by putting a `flush_plot()` and suitable `delay_plot()` after every plot command. This will get you warmed up for the animations.

Note: if your code is in multiple files and you want to use a Makefile or shell script to automate your compilation, please submit the script or Makefile along with your programs. This will make it easier for the TA to compile your code!