A First Numerical Problem

Many problems that we encounter in many parts of physics involve ordinary differential equations (ODEs). Problems like projectile motion, harmonic motion, mechanics and much more require working and solving differential equations. In this chapter, we begin to delve into a problem that involves a first-order differential equation and use it to introduce some computational techniques that'll be utilized heavily in the subsequent chapters.

1.1 Radioactive Decay

We know that many nuclei are unstable and will decay into more stable forms of examples. One of the most prominent examples is Uranium-235, which contains 143 neutrons and 92 protons. Uranium has a very small probability that it would decay into two smaller nuclei. But determining when a single U-235 atom will decay into lighter elements is very difficult. We can only get a probability of U-235 decaying. An equivalent way of thinking about this is to see it is to take multiple of U-235 atoms and see how long it will take for half of the nuclei to decay. This is called the mean lifetime of a particle. For U-235, the mean lifetime is 1×10^9 years. If $N_u(t)$ is the number of uranium atoms present in a sample at time t , the behavior is given by the following differential equation:

$$\frac{dN_u}{dt} = -\frac{N_u}{\tau}$$

where τ is the time constant for the decay. Separating the integration and solving this differential equation will lead to

$$N_U = N_U(0)e^{-t/r}$$

where $N_U(0)$ is the number of nuclei present at t = 0 seconds.

1.2 A Numerical Approach

While we say in 1.1 that we can solve this without doing a numerical approach. this problem is a great way to see computational methods we'll see later. For the numerical method, we'll use a very useful concept that is the Taylor Expansion of N_U which is:

$$N_U(t) = N_U(0) + \frac{dN_U}{dt}\Delta t + \frac{1}{2}\frac{d^2N_U}{dt^2}\Delta t^2 + \dots$$

Where $N_U(0)$ is the value of our function at t = 0, $N_U(\Delta t)$ is its value at t = Δt seconds. . if we take Δt to be small, then it is usually a good approximation to simply ignore the terms of higher powers of Δt leaving us with:

$$N_U(\Delta t) \approx N_U(0) + \frac{dN_U}{dt}\Delta t$$

The same result can be obtained using the definition of a derivative. The derivative of N_U evaluated at a time t can be written as

$$\frac{dN_U}{dt} \equiv lim_{\Delta t \to 0} \frac{N_U(t+\Delta t) - N_U(t)}{\Delta t} \approx \frac{N_U(t+\Delta t) - N_U(t)}{\Delta t}$$

In this equation we have assumed that Δt is small but nonzero which we we can re arrange the above equation to

$$N_U(t+\Delta t) \approx N_U(t) + \frac{dN_U}{dt}\Delta t$$

Using the functional form of the derivate we stated earlier and inserted into the above equation we get,

$$N_U(t+\Delta t) \approx N_U(t) + \frac{N_U(t)}{\tau}\Delta t$$

Using this equation, we can calculate the value of the decay function by each step of Δt which we can choose. Therefore, we'll first take the function t=0, which we'll know, and using the above formula, we'll calculate $N_U(t+\Delta t)$ and using this value, we'll calculate the value of $N_U(t+2\Delta t)$ and so on forth until we get the entire range of values and can see the trend of the equation. This method is called the Euler's Method and is a useful general algorithm for solving ODEs. We do need to remember that these are just approximations and we need to make sure that the differences between the approximation and reality is negligible.

1.3 Design and Construction of a Working Program: Pseudocodes

In order to create a working program, we need to identify the goals of program and what it takes in and what it needs to output. While there might be many different small differences and nuances between programming languages, pseudocode is a 'language' where you define the general structure of the program in a way that is useful to anyone trying to write this programs. This is not an exact programming language but a general description of the essential parts of a program expressed in general English. The concept is to give enough idea and details in pseudocode so that any programmer fluent in any language can use it to write a program in their specialized programming language. The overall structure of the program consists of four basic tasks 1) declare the necessary variables 2) initialize all variables and parameters 3) do the calculation and 4) store the results.

Therefore, the pseudocode for the main program of the radioactive decay problem

```
- Some Comment text to describe the nature of the program
  -  Declare necessary variables and arrays
  - initialize variables
  - Do the actual calculation
  - store the results
```

While this is only the main program, the individual tasks such as the initialization of variables and the actual calculation will be done in functions and will be needed

The Pseudocode for function *initialize*

- Prompt for and assign $N_U(0)$, τ , and Δt
- Set initial value of time, t(0)
- Set number of time steps for calculation

The Pseudocode for function *calculate*

- For each time step i (beginning with i = 1), calculate N_U and t at step i+1:
  - $N_U(t_{i+1}) = \frac{N_U(t_i)}{\tau} dt$ (Use the Euler Method, (1.7))
  - $t_{i+1} = t_i + \Delta t$
  - repeat for n-1 time steps

1.4 Testing Your Program

Many people might think this is the most frustrating part of the coding process but testing the program is not trivial. We need to make sure that the output given is correct or not? Here are some general guidelines:

Does the output look reasonable? Before running the program and checking the output, you should have a rough idea of how the result should look. And when you implement the program, you need to check whether the results are consistent with your intuition and instincts. It can also improve the understanding of the subject and the problem. **Does your program agree with any exact results that are available?** Since we knew about the analytic solution for the radioactive decay. we are able to easily compare the numerical solution to the analytical solution. While we'll not always get analytic solutions or even examples we get to compare our results to, we should be able to keep some constraints and limits for the special values of parameters. This is necessary step to check the results but doesn't guarantee that the solutions are correct. **Always check that your program gives the same answer for different "step sizes"** Our decay program used the Euler's method's which had a time-step variable Δt , which will be the case for many other numerical methods having step- or grid-sized variables. The final answer however, should be independent of all of these variables. This is another necessary measure but also doesn't guarantee total program accuracy.

1.5 Numerical Considerations

Numerical Errors are one of the most important things to address whenever doing a numerical solutions to any problems. It is an inevitable problem to pop up in these methods, as there is only a finite amount of accuracy we can have while doing these methods. This is such an important matter such that there are entire books to devoted to studying this area of science, computer science and mathematics. In our example of radioactive decay, the errors were due to approximations we made during the derivation of the equation $N_U(t + \Delta t) \approx N_U(t) + \frac{N_U(t)}{\tau}\Delta t$. These are called round-off errors and are always present whenever numbers are represented by a finite number of digits. This is

not a problem now as modern programming language systems have a large number of significant digits. However, these methods are sensitive to the round-off errors.

Now that we know about the errors that can happen in numerical methods, we need to be answer two questions: *How do we know that the errors introduced by the step-variable is negligible?* and *How do we choose the step value for a calculation?* In our case, it becomes easy as there is an analytical solution to this problem. Comparing different step-sizes with the exact analytical solutions help us see the relation between the step-sizes and analytical solution. As we can see, as the step-sizes decrease, the values become closer to the analytical value. In the textbook, where step sizes of 0.05, 0,2 and 0.5 seconds, we can see that 0.05 s steps is where the differences between analytical and numerical solutions become very small.

While in this examples, we had a very simple way to check numerical error, it is not always the case. In these cases, we should always check the calculated results reach a fixed value or curve as the step size is made smaller. Another matter is what step size is suitable for a certain problem. Here again, there is no hard rules for us to follow. Ideally, you use a step size that is small to any important characteristic time scales in this equation. In this example, it would be better if the time scales were a small fraction of τ . You must use your knowledge and understanding of the problem and subject matter and problem in order to choose the appropriate scales and methods for any problem.

1.6 Programming Guidelines and Philosophy

We noted in earlier sections how that programming is being done individually but there are guidelines that are recommended guidelines that we can follow in order to make the code more legible and understandable to others when viewing our code. Some of them are:

1. **Program Structure.** Use subroutines and functions to organize the tasks and process of the program and make it more readable and understandable. The *main* program for the decay problem is just the outline of the program while anything else requiring more than few lines of code or are required to do repeatedly are done in subroutines and functions.
2. **Usage of Descriptive Names.** The names of functions, variables and other aspects of the code must adhere and be related to the problem at hand. Using descriptive naming schemes will make it easier to understand, as they act like mini-comments in the code itself.
3. **Use Comment Statements.** Use comment statements to explain program logic and describe variables. Using descriptive variable names should not need a large amount of comments to explain the function/subroutine.
4. **Sacrifice (almost) everything for clarity.** While it is tempting to write coding in a very 'efficient' or short manner using as less lines as possible in the misguided belief that it is going to make the code run faster which is not always the case. More importantly, coders needs to place much more importance in the *readability* and *clarity*. It is almost always better to write some more lines and add some variables in order to make the code more understandable. It's also pertinent to format the code in a proper ways like making the code look like an physics equation. This will not only help others but also yourself when you go back to review the code again in the future.

There might be cases where programmatic execution might matter on the lines of code and how variables are set up. This is only when the compute time and computer resources needed becomes substantial. Even then, the algorithmic execution and improvements matter a lot more than the actual terseness of the code. And modern programming language compilers often do a good job optimizing job without us even realizing it.

5. **Take time to make graphical output as clear as possible.** We need to be careful is the visualization of our data. What quantities to plot, which axis to plot them in, and in what manner. The axes should be labeled correctly (including units). and the parameter values should given directly in the graph.