Problem 1) In classes that you took earlier, you met problems that were difficult or impossible to solve analytically. Suggest at least three problems that are interesting for solving with a computer. Describe physics and **ALL** detailed equations needed to solve these problems. Please be as specific as needed. Later we may use these problems for midterm projects.

Certainly, there are many problems encountered during a conventional university physics education at both the undergraduate and graduate levels which are not analytically tractable. A few problems are discussed below from classical mechanics, electrodynamics, and quantum mechanics.

First, we discuss a simple example from classical mechanics. Early on in any physicist's academic journey, they encounter projectile motion but soon realize that one can only include air resistance or drag and still solve the equations of motion analytically in a few special cases. In particular, the relevant equation of motion for a projectile of mass m moving in a uniform gravitational field near the surface of the earth under drag is

$$m\ddot{\boldsymbol{r}}(t) = -g\hat{\boldsymbol{z}} + (bv + cv^2)\hat{\boldsymbol{v}}, \tag{1}$$

where we have oriented our z axis pointing away from the surface of the earth, b, c are drag coefficients, and $\hat{\boldsymbol{v}} = \dot{\boldsymbol{r}}/|\dot{\boldsymbol{r}}|$ is the velocity unit vector. Of course, if $c \neq 0$, we are immediately sunk. There is no mathematical machinery available to us to solve such a non-linear equation in closed form. There are several numerical schemes, however, which allow us to construct an approximate solution to this differential equation for some specified initial conditions. The most well-known approach to anyone having studied calculus is just the naïve Newton's method, which translates the derivatives into fractions with the time interval being very small. However, there are other more sophisticated methods, such as the Runge-Kutta class and finite element methods (of which Newton's method is a special case) that can yield more accurate results without sacrificing computational efficiency.

An electrodynamics problem of general interest is simulating wave propagation in media. The relevant set of equations govering these dynamics are Maxwell's:

$$\boldsymbol{\nabla} \cdot \boldsymbol{D} = \frac{\rho}{\epsilon} \tag{2}$$

$$\boldsymbol{\nabla} \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \tag{3}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{B} = 0 \tag{4}$$

$$\boldsymbol{\nabla} \times \boldsymbol{H} = \boldsymbol{J} + \frac{\partial \boldsymbol{D}}{\partial t}. \tag{5}$$

There are some simple, although not necessarily physical, scenarios which permit analytic solutions for the fields, but for more complicated geometries or media with dispersive properties, these equations become much more cumbersome or impossible to solve in closed form. For example, if we wish to simulate the two slit experiment, which illustrates interference, the geometry makes implementing boundary conditions more difficult on paper. Because these are sets of coupled partial differential equations, though, we can employ a finite element numerical scheme in order to evolve the system forward in time from an initial configuration of the fields at $t = 0$. One difficulty in simulating the two slit diffraction is that we are forced to truncate our spatial domain, and the artificial boundaries can induce some undesired artifacts due to the imposition of boundary conditions there, and hence, to properly simulate such systems, care needs to be taken to address these. For example, one can enlarge the simulated spatial domain such that waves propagating from the imposed boundaries do not interact with those within the region of interest. Another more clever and computationally efficient scheme is to devise artificial regions (or perfectly matched layers) which damp the waves exiting the physical region of interest such that they interfere minimally when they enter the physical region again.

Lastly, a problem from quantum mechanics that could be interesting to solve numerically is the atomic spectrum of elements with more than one electron. Because a multi-electron system is a three-body system or even more expansive than that, the Schödinger equation

$$\hat{H}\psi(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n) = \sum_i \left[-\frac{\hbar^2}{2m_i}\boldsymbol{\nabla}_i^2 + \frac{e^2}{8\pi\epsilon_0}\sum_{j\neq i}\frac{1}{|\boldsymbol{x}_i - \boldsymbol{x}_j|}\right]\psi(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n) = E\psi(\boldsymbol{x}_1,\dots,\boldsymbol{x}_n). \tag{6}$$

becomes nonlinear and impossible to solve using analytic mathematical methods, although it is critical for spectral analysis in many fields that such systems are understood.

Problem 2) Prepare your computer for the class with a language of your choice. Most likely it will be one of these: C/C++, Python, or MatLab. (You can use more than one if you want.) Briefly describe software that you are using

I plan to primarily use *Python3* on my personal Mac computer, which will be typeset in the Vim text editor within the terminal environment. Typically, although not for this course, I also utilize Jupyter notebooks for smaller simulations, data analysis, and graphical visualizations.

Problem 3)

Write a program to solve the quadratic equation $ax^2+bx+c = 0$ by using the quadratic formula to obtain roots:

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{7}$$

Your program should also be capable to handle complex roots. Note that you don't have to use complex variables, but you may. Choose you way to input a, b, and c.

```python
def get_roots(a,b,c):
    term1 = -b/2/a
    D = b**2 - 4*a*c
    if D > 0:
        term2 = D**0.5/2/a
        print(f'x_{{+}} = {term1+term2}, x_{{-}} = {term1-term2}')
    elif D == 0:
        print(f'x_{{+}} = x_{{-}} = {term1}')
    else:
        term2 = (-D)**0.5/2/a
        print(f'x_{{+}} = {complex(term1,term2)}, x_{{-}} = {complex(term1,-
   term2)}')

    return None


if __name__ == '__main__':

    c = input('Enter the (real) coefficients for a quadratic polynomial: ')
    c = [int(_) for _ in c.split()]
    get_roots(*c)
```

Figure 1: Python code used to solve the quadratic equation and print the roots. Note that if the script is run as the main program, the user is asked to input three real numbers, and this input is converted appropriately and subsequently inserted into the *get_roots* function.

The code used to solve the problem described above is shown below in Fig. 1. The primary function takes three real numbers as input and prints the two (possibly complex) roots of the function $f(x) = ax^2 + bx + c$. Note that there are three distinct categories of solutions, governed by the value of the discriminant $D = b^2 - 4ac$. If $D \geq 0$, then $\sqrt{D}$ is a purely real number and the solutions are real numbers. Of course, if $D = 0$, then we only have one root of multiplicity two. Alternatively, if $D < 0$, then $\sqrt{D} = i\sqrt{|D|}$ is purely imaginary, and the roots are complex conjugates of each other, where the real part is $b/(2a)$ and the imaginary parts are $\pm\sqrt{|D|}/(2a)$.

Problem 4)

Write a program that calculates a series of Fibonacci numbers and checks which ones are prime numbers. Explore what is the largest Fibonacci number you may get with your code.

There are two main functions used in the script shown in Fig. 2 implementing the solution of the prompt above: *is_prime* and *get_fibonacci*. The former simply checks whether an input number is prime by looping through candidate integer factors. Note that since any number less than two is non-prime by definition but not properly checked by the loop, it is hard coded at the beginning of the function that the function returns *False*. The latter function generates the fibonacci series, which is defined as the arithmetic sequence with

$a_0 = 0$, $a_1 = 1$, and $a_{n+1} = a_n + a_{n-1}$. The first optional parameter *iter_ max* defines the maximum number of extra numbers in the sequence which are generated and is by default 100. Additionally, the *check_ prime* parameter is of boolean type and governs whether the printing portion of the function checks whether the sequence entries are prime and is by default set to *True*. Note that the sequence generation can produce arbitrarily large numbers very quickly, limited only by the hardware on one's computer. However, because the prime number checking is very primitive, the while loop must search through roughly $\sqrt{N}$ numbers if the input is indeed not prime. It was checked that by roughly the 85th entry of the Fibonacci sequence, the speed of the prime functionality degrades. This could be improved in a few ways. First, there are some useful tricks to test factorization of small numbers. For example, if a number is not even, no even number should be a candidate, and if the sum of the digits is not divisible by 3, then the number is not divisible by 3 or any multiple thereof, and so on. Additionally, one could implement some parallelization to check candidates.

```python
import sys

def is_prime(n):
    if n < 2:
        return False

    flag = True
    test = 2
    while flag and test < n**0.5:
        if n % test == 0:
            flag = False
        test += 1

    return flag


def get_fibonacci(iter_max=100,check_prime=True):
    series = [0,1]
    i = 0
    while i < iter_max:
        series.append(series[-2]+series[-1])
        i += 1

    for i in range(len(series)):
        if check_prime:
            print(f'{i}:',series[i],{True: 'is prime', False: 'is not prime'}[
  is_prime(series[i])])
        else:
            print(f'{i}:',series[i])



if __name__ == '__main__':
    iter_max = 10
    if len(sys.argv) > 1:
        iter_max = int(sys.argv[1])
    get_fibonacci(iter_max=iter_max)
```

Figure 2: Python code used to produce the Fibonacci series and check whether a number in the sequence is prime.