Integration of Differential Equations

Gabriel S. Cabrera

August 24, 2018

Contents

1 Introduction **1**

2 Theory **2**

2.1 Linear 1st Order ODEs 3

2.1.1 Analytical Solution 3

2.2 Linear Homogenous 2nd Order ODEs 3

2.2.1 Analytical Solution for Constant Coefficients 3

3 Simple Numerical Integration **4**

3.1 In Python 5

4 Why Use Integration to Solve Differential Equations? **6**

4.1 Programmatical Numerical Integration 7

4.1.1 Example 1 7

4.1.2 Example 2 8

5 Numerical Methods for 2nd Order ODEs **9**

5.1 Euler-Cromer 9

5.2 Runge-Kutta 4 10

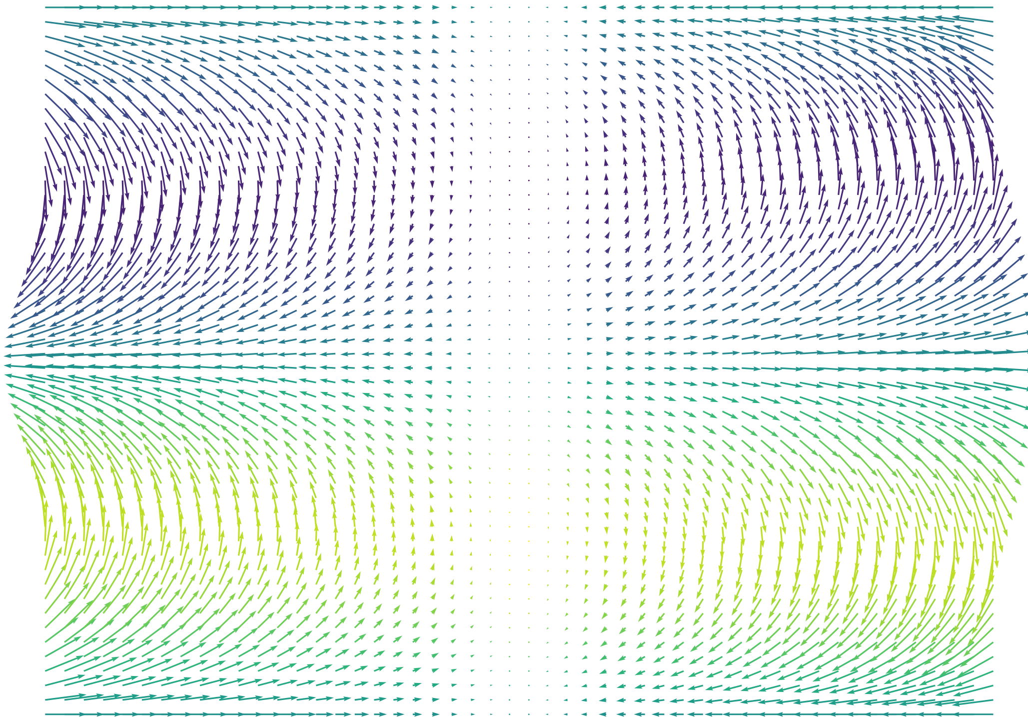
5.3 Leapfrog 12

1 Introduction

Our universe is host to an enormous variety of physical phenomena, and although they differ in just about every way imaginable it is often possible to describe many of them using common principles.

To model many such phenomena, we must create systems composed of differential equations; these can be *ordinary* differential equations or *partial* differential equations depending on the

given situation. For the sake of simplicity, we often use the abbreviations *ODE* and *PDE*, respectively.



From forces and energies, to flows, waves, and currents, we can use ODEs and PDEs to model many types of natural phenomena – in fact, any physical process with a quantity that determines its own rate of change can be modeled as a differential equation. For example, the movement of an object through a fluid is described with a differential equation – this is partially because its speed relative to the surrounding fluid determines the amount of drag it experiences. Another situation where a differential equation would be useful is when modeling an N-body system of charged particles in motion, since it would allow us to calculate the sum of the forces on each particle based on their relative distances.

All-in-all, differential equations are fundamentally engrained in nearly every branch of physics, as well as other fields such as mathematics, statistics, and many others.

2 Theory

Let us begin this section by going through the theory behind ODEs, since these are simpler than PDEs and are often analytically solvable; they are characterized by the fact that they contain only one independent variable – we will then briefly introduce PDEs. As an undergraduate, one is most likely to encounter a few select subcategories of ODEs, and a limited number of PDEs.

2.1 Linear 1st Order ODEs

This is the simplest form of an ODE, where we have a quantity and its first derivative with respect to the system's independent variable. This can be generalized as follows:

$$\frac{d}{dx}f(x) + P(x)f(x) = Q(x) \tag{1}$$

Where $f(x), P(x)$, and $Q(x)$ are arbitrary functions of the independent variable x.

2.1.1 Analytical Solution

Equations of this form are easy to work with, as they all have the following analytical solution[1]

$$f(x) = e^{-\int P(x)dx} \int e^{\int P(x)dx} Q(x)dx \tag{2}$$

Since these are easily solved, it can be more efficient to avoid numerical integration altogether when exposed to problems of this form, and instead calculate the analytical solution directly.

2.2 Linear Homogenous 2nd Order ODEs

Now, we will move on to a more tricky kind of ODE, one which contains an independent variable and its second derivative – its first derivative may also be included, depending on the purpose of the model. These can be generalized with the following:

$$\frac{d^2}{dx^2}f(x) + P(x)\frac{d}{dx}f(x) + Q(x)f(x) = 0 \tag{3}$$

Where $f(x), P(x)$, and $Q(x)$ are arbitrary functions of the independent variable x.

Second order linear homogenous ODEs with constant coefficients[2] can be solved analytically by finding their *characteristic polynomials*. Otherwise, it is perfectly reasonable to rely on numerical methods, which we will introduce later in this chapter.

2.2.1 Analytical Solution for Constant Coefficients

We are given an arbitrary linear homogenous second order ODE of the following form:

$$a\frac{d^2}{dx^2}f(x) + b\frac{d}{dx}f(x) + cf(x) = 0 \tag{4}$$

Where $f(x)$ is an arbitrary function, and $a, b, c \in \mathbb{R}$. Its characteristic polynomial is then of the following form:

$$ar^2 + br + c = 0 \tag{5}$$

[1] Assuming, of course, that the resulting integral has a solution!

[2] This implies that $P(x)$ and $Q(x)$ are constants, rather than functions.

Using our coefficients a, b, and c, we can determine the form of our solution $f(x)$. There are three distinct possibilities:

1. If $b^2 - 4ac > 0$, we have a solution with two real-valued roots r_1 and r_2, where $r_1 \neq r_2$. This gives us a solution of the following form:

$$f(x) = Ae^{r_1 x} + Be^{r_2 x} \tag{6}$$

   Where A and B are determined by the initial conditions of our system.

2. If $b^2 - 4ac = 0$, we have a solution with a single root r, which is used twice in place of r_1 and r_2. Our solution in this case takes the form shown below:

$$f(x) = Ae^{rx} + Bxe^{rx} \tag{7}$$

   As previously, A and B are constants for particular boundary conditions.

3. If $b^2 - 4ac < 0$, we have a solution with a pair of complex-conjugate roots r and $\bar{r}$ such that $r = \lambda + \mu i$ and $\bar{r} = \lambda - \mu i$. Now, we have a solution in another form:

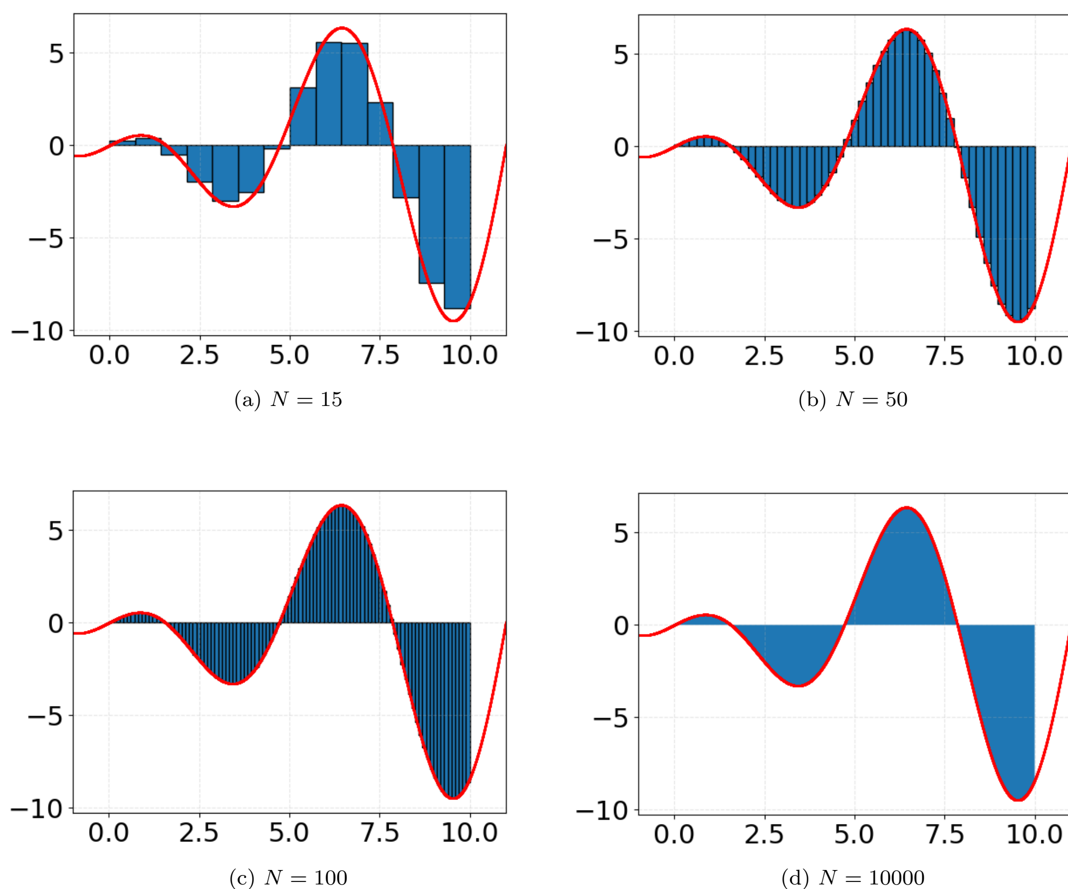
$$f(x) = Ae^{\lambda x} \cos(\mu x) + Be^{\lambda x} \sin(\mu x) \tag{8}$$

   Once again, recall that A and B are found when plugging in for a set of initial conditions.

These three sets of solutions originate from the quadratic equation, where we have a square root $\sqrt{b^2 - 4ac}$; as a result, these three situations arise naturally due to the nature of the solution to the characteristic polynomial, as it is a quadratic function.

3 Simple Numerical Integration

Before we delve into the methods used to integrate differential equations, let us revisit the concepts behind the integration of simple one-dimensional functions. When a student is first exposed to the concept of integration, it is commonplace to use Riemann sums as a way of visualizing this process since the integral of a one-dimensional function is, in essence, the area under the function's curve. Riemann sums are an intuitive way of visualizing such ideas, and can be used to approximate integrals; we see how this works in Figure 1 – note that this is an example of a *midpoint* Riemann sum, since the center of each bar intercepts the plotted function, rather than its left or right corner.



(a) $N = 15$

(b) $N = 50$

(c) $N = 100$

(d) $N = 10000$

Figure 1: Riemann sums of varying accuracy, for $f(x) = x\cos(x)$ from 0 to 10

Given a function $f(x)$, we can use a generalized midpoint Riemann sum to find the area under its curve $F(x)$:

$$F(x) \approx \sum_{i=0}^{N} f\left(a + \frac{N}{2}\right) \Delta x \tag{9}$$

Where N is the number of bars in the Riemann sum, a and b are the boundaries of our integration, and $\Delta x = \frac{b-a}{N}$ is the width of each bar.

3.1 In Python

As we saw in (9), a Riemann sum is simply the process of summing over the areas of many small rectangles. Such a process is straightforward to implement in Python, as we see in the script below:

```python
def f(x):
    '''
        Insert Function Here
    '''
    return x

def integrate(f, a, b, N):
    tot = 0
    dx = (b-a)/N
    for n in range(N):
        tot += f(a + n*dx)*dx
    return tot
```

4 Why Use Integration to Solve Differential Equations?

It is often rather difficult (or sometimes impossible) to solve differential equations analytically, so we must instead rely on numerical methods to find an acceptable approximation to our solution. So why use integration? Let us build an understanding by looking at a generalized 1$^{\text{st}}$-order differential equation:

$$f(x(t), t) = \frac{d}{dt}x(t) \tag{10}$$

We are given a function $f(x(t), t)$ that is always equal to the time-derivative $\frac{d}{dt}x(t)$; turning this equation into a discrete step-wise process means we can work with our differentials more directly. Recall that the derivative is defined as follows:

$$\frac{d}{dt}x(t) = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{11}$$

In the case where we are working with discrete values, we may rewrite (10) by using the above definition (while ignoring the limit):

$$f(x(t), t) = \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{12}$$

We can create a discrete algorithm by solving for $x(t + \Delta t)$:

$$x(t + \Delta t) = x(t) + f(x, t)\Delta t \tag{13}$$

Does this seem familiar? It may, because in essence it is the approximated representation of the following integral:

$$x(t + \Delta t) = x(t) + \int_{t}^{t+\Delta t} f(x(t), t)dt \tag{14}$$

In conclusion: given an initial condition x_0, a time step Δt and a final time T, we can perform a numerical integration on a step-by-step basis with a simple algorithm, which will give us the solution to our differential equation:

Algorithm 1 A simple numerical integration algorithm

1: t = 0
2: **while** ($t \leq T$) **do**
3: $x_{i+1} = x_i + f(x_i, t)\Delta t$
4: $t_{i+1} = t_i + \Delta t$
5: **end while**

4.1 Programmatical Numerical Integration

We can use `python` to solve an equation of the form shown in (10), it is simply a matter of:

1. Creating a `python` function for $f(x(t), t)$
2. Selecting an initial condition $x(0)$
3. Finding the right balance between precision and speed, then choosing a Δt and final value T
4. Implementing Algorithm 1

Following the above steps will leave us with a script of the following form:

```
def f(x, t):
    ...

x = ...
t = ...
dt = ...
T = ...

for i in range(int(T//dt)):
    t += dt
    x += f(x, t)*dt
```

Note that the above script is incomplete, and requires that a potential user fill in the blanks for all spots marked "`...`"; these are representative of the first three points in the aforementioned list[3].

4.1.1 Example 1

So how exactly can we use the above script to our advantage? Let's use the principles outlined above to solve the following differential equation:

$$\frac{d}{dt}x(t) = \cos(t) \tag{15}$$

Given the initial condition $x(0) = 0$, the time-step $\Delta t = 0.01$ and final time $T = 10$, we can use the following script:

[3] Those being the initial condition, time-step, and final time.

```python
import math

def f(t):
    return math.cos(t)

x = 0
t = 0
dt = 0.01
T = 10

for i in range(int(T//dt)):
    t += dt
    x += f(t)*dt

print(x)
```

This results in the following output:

```
-0.5448212197270629
```

We can compare this result to the analytical solution for (15):

$$x(t) = \sin(t) + C \tag{16}$$

Given that $x(0) = 0$, this gives us our particular solution:

$$x(t) = \sin(t) \tag{17}$$

We then find that $x(10) \approx -0.544$, which shows that our method was successful!

4.1.2 Example 2

Now that we've shown that our algorithm works as intended, let's attempt to solve a more complex differential equation:

$$\frac{d}{dt}x(t) = \sin(x + t)\cos(x - t)e^{2t} \tag{18}$$

Finding an analytical solution to the above is no simple task; it is preferable to rely on numerical methods. Given that $x(0) = 1$, $\Delta t = 0.001$, and $T = 5$, we have that:

```python
import math

def f(x, t):
    return math.sin(x + t)*math.cos(x - t)*math.exp(2*t)

x = 1
t = 0
dt = 0.001
T = 5

for i in range(int(T//dt)):
    t += dt
    x += f(x, t)*dt

print(x)
```

This results in the following output:

```
-431.77886890444825
```

Here we see the true beauty of numerical methods – a traditionally complex problem can be simplified significantly when given sufficient computing power!

5 Numerical Methods for 2nd Order ODEs

Let us now imagine that we are interested in approximating the solution to a linear, nonhomogenous second-order ODE – we can accomplish such a task by making use of integration algorithms.

All ODEs in this section will be of the following form:

$$\frac{d^2}{dt^2}f(x) + P(x)\frac{d}{dx}f(x) + Q(x)f(x) = R(x) \tag{19}$$

Where x is the only independent variable in our equation[4], and $f(x)$, $P(x)$, $Q(x)$, $R(x)$ are arbitrary functions of x.

For this section we will simplify our notation further by rewriting the above in a more physics-friendly manner:

$$a(t) + P(t)v(t) + Q(t)x(t) = R(t) \tag{20}$$

In this section, we will introduce three algorithms that can be used to numerically integrate over equations of the above form, each with its own strengths and weaknesses. This section will be mostly theoretical, such that in-depth examples will be brought up in a subsequent section.

5.1 Euler-Cromer

One of the simplest methods available to us is Euler-Cromer integration; if you are looking for a fast and simple algorithm, Euler-Cromer is well suited to the task. On the other hand, it has a lack of precision when compared to other algorithms, and is not a good fit for systems where the conservation of energy is of vital importance.

- **Speed**: `Excellent`
- **Simplicity**: `Excellent`
- **Precision**: `Bad`
- **Energy**: `Not Conserved`

To use this algorithm, we must start with a set of initial conditions x_0 and v_0. We also need to choose a time-step[5] Δt, and a total runtime T[6].

[4] Differential equations of multiple independent variables are PDEs, as introduced in **??**, and so involve more complexity.

[5] A smaller time-step will lead to more accurate results, but will increase a program's runtime.

[6] We can assume that our initial time is $t = 0$.

Algorithm 2 The Euler-Cromer integration algorithm for a linear nonhomogenous second-order ODE

1: t = 0
2: **while** $(t \leq T)$ **do**
3: $a_{i+1} = R(t_i) - P(t_i)v_i - Q(t_i)x_i$
4: $v_{i+1} = v_i + a_{i+1}\Delta t$
5: $x_{i+1} = x_i + v_{i+1}\Delta t$
6: $t_{i+1} = t + \Delta t$
7: $i = i + 1$
8: **end while**

In `python`, we can set up an integration loop as a function:

```
1  def integrator(x0, v0, T, dt):
2
3      def P(t):
4          ...
5
6      def Q(t):
7          ...
8
9      def R(t):
10         ...
11
12     N = int(T//dt)
13
14     t = dt
15     x = x0
16     v = v0
17
18     for i in range(N):
19         a = R(t) - P(t)*v - Q(t)*x
20         v += a*dt
21         x += v*dt
22         t += dt
23
24     return x
```

To use the above function, be sure to set up functions[7] for `P(t)`, `Q(t)`, and `R(t)`.

5.2 Runge-Kutta 4

In situations where we are in need of more precise results on a step-wise basis[8], we can use the fourth order Runge-Kutta algorithm[9]. As with the Euler-Cromer algorithm, RK4 is not energy-conserving – in addition to this, it doesn't run as quickly as Euler-Cromer due to its complexity.

RK4 is very useful in situations such as when solving partial differential equations, or finding the solutions for more complex systems that require extra precision.

- **Speed**: `Bad`
- **Simplicity**: `Bad`

[7] These can be replaced with constants if necessary.
[8] In other words, if we wish to increase accuracy without reducing Δt.
[9] We will use the abbreviation RK4.

- **Precision**: `Excellent`
- **Energy**: `Not Conserved`

Algorithm 3 The fourth order Runge-Kutta integration algorithm for a linear nonhomogenous second-order ODE

1: t = 0
2: **while** $(t \leq T)$ **do**
3: $x_\alpha = x_i$
4: $v_\alpha = v_i$
5: $a_\alpha = R(t_i) - Q(t_i)x_\alpha - P(t_i)v_\alpha$
6:
7: $x_\beta = x_\alpha + v_\alpha \frac{\Delta t}{2}$
8: $v_\beta = v_\alpha + a_\alpha \frac{\Delta t}{2}$
9: $a_\beta = R(t_i + \frac{\Delta t}{2}) - Q(t_i + \frac{\Delta t}{2})x_\beta - P(t_i + \frac{\Delta t}{2})v_\beta$
10:
11: $x_\gamma = x_\alpha + v_\beta \frac{\Delta t}{2}$
12: $v_\gamma = v_\alpha + a_\beta \frac{\Delta t}{2}$
13: $a_\gamma = R(t_i + \frac{\Delta t}{2}) - Q(t_i + \frac{\Delta t}{2})x_\gamma - P(t_i + \frac{\Delta t}{2})v_\gamma$
14:
15: $x_\delta = x_\alpha + v_\gamma \Delta t$
16: $v_\delta = v_\alpha + a_\gamma \Delta t$
17: $a_\delta = R(t_i + \Delta t) - Q(t_i + \Delta t)x_\delta - P(t_i + \Delta t)v_\delta$
18:
19: $a_{\text{avg}} = \frac{1}{6}\left(a_\alpha + 2a_\beta + 2a_\gamma + a_\delta\right)$
20: $v_{\text{avg}} = \frac{1}{6}\left(v_\alpha + 2v_\beta + 2v_\gamma + v_\delta\right)$
21:
22: $v_{i+1} = v_\alpha + a_{\text{avg}}\Delta t$
23: $x_{i+1} = x_\alpha + v_{\text{avg}}\Delta t$
24:
25: $t_{i+1} = t + \Delta t$
26: $i = i + 1$
27: **end while**

There are other ways to describe 4th-order Runge-Kutta integration, but the above shows each step of the process in depth rather than combining each step to maximize legibility. If one prefers to perform each step inline, that is also acceptable.

RK4 can be implemented in Python in a variety of ways; the skeleton code below is one of many methods that can be used, following the exact step-by-step method shown in the above algorithm:

```
1  def integrator(x0, v0, T, dt):
2
3      def P(t):
4          ...
5
6      def Q(t):
7          ...
8
9      def R(t):
```

```
10          ...
11
12      N = int(T//dt)
13
14      t = dt
15      x_1 = x0
16      v_1 = v0
17      dt_2 = dt/2
18
19      for i in range(N):
20          a_1 = R(t) - Q(t)*x_1 - P(t)*v_1
21
22          t += dt_2
23          x_2 = x_1 + v_1*dt_2
24          v_2 = v_1 + a_1*dt_2
25          a_2 = R(t) - Q(t)*x_2 - P(t)*v_2
26
27          x_3 = x_1 + v_2*dt_2
28          v_3 = v_1 + a_2*dt_2
29          a_3 = R(t) - Q(t)*x_3 - P(t)*v_3
30
31          t += dt_2
32          x_4 = x_1 + v_3*dt
33          v_4 = v_1 + a_3*dt
34          a_4 = R(t) - Q(t)*x_4 - P(t)*v_4
35
36          a_avg = (a_1 + 2*a_2 + 2*a_3 + a_4)/6
37          v_avg = (v_1 + 2*v_2 + 2*v_3 + v_4)/6
38
39          v_1 = v_1 + a_avg*dt
40          x_1 = x_1 + v_avg*dt
41
42      return x_1
```

Once again, the above script is incomplete and requires the user to create functions for `P(t)`, `Q(t)`, and `R(t)`.

5.3 Leapfrog

So far we've explored two integration algorithms – unfortunately, both of them are unable to model the conservation of energy; to remedy this, we can use **Leapfrog** integration[10]. Relative to Euler-Cromer and RK4, this algorithm is a middle-ground of sorts with regards to its complexity, efficiency, and precision – it is its ability to conserve energy that makes it invaluable, such as in the simulation of planetary orbits or the motion of an oscillating spring.

- **Speed**: `Moderate`
- **Simplicity**: `Moderate`
- **Precision**: `Moderate`
- **Energy**: `Conserved`

[10] Leapfrog integration is actually a type of Verlet integration, more information available here: https://en.wikipedia.org/wiki/Verlet_integration

Algorithm 4 The Leapfrog integration algorithm for a linear nonhomogenous second-order ODE

1: t = 0
2: **while** ($t \leq T$) **do**
3: $a_{i+1} = R(t_i) - P(t_i)v_i - Q(t_i)x_i$
4: $x_{i+1} = x_i + v_i\Delta t + \frac{1}{2}a_i\Delta t^2$
5: $v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t$
6: $t_{i+1} = t + \Delta t$
7: $i = i + 1$
8: **end while**

To model this in `python`, we can use the script below:

```
def integrator(x0, v0, T, dt):

    def P(t):
        ...

    def Q(t):
        ...

    def R(t):
        ...

    N = int(T//dt)

    t = dt
    x = x0
    v = v0
    a_i = R(t) - P(t)*v - Q(t)*x

    for i in range(N):
        t += dt
        x += v*dt + 0.5*a_i*dt**2
        a_iplus1 = R(t) - P(t)*v - Q(t)*x
        v += 0.5*(a_i + a_iplus1)*dt
        a_i = a_iplus1

    return x
```

As in all our previous examples, be sure to create functions for `P(t)`, `Q(t)`, `R(t)`.