# Appendix A. Solving differential equations numerically

Differential equations arise often in physics, as well as in other fields (e.g. economics, biology, etc.). Differential equations are characterized by the fact that they relate a function and its derivatives. Ordinary differential equations (ODEs) contain a function of only 1 independent variable. Partial differential equations (PDEs) contain functions of multiple independent variables and typically relate the function to its partial derivatives. PDEs are generally more complex to solve.

## A.1 Ordinary Differential Equations

ODEs are characterized by the fact that they have only 1 independent variable. For example

$$\frac{d^2y}{dx^2} + a(x)\frac{dy}{dx} = b(x) \tag{A.1}$$

is called a second order ODE. The order of the ODE is given by its highest derivative (in this case, we have a second order derivative). Generally, the goal is to solve for the function $y(x)$.

Any order ODE can be reduced to a coupled-system of first order ODEs, by introducing a variable substitution. The above equation can be re-written by introducing a new function $z(x)$:

$$\begin{aligned} z(x) &\equiv \frac{dy}{dx} \\ \frac{dz}{dx} &= b(x) - a(x)z(x) \end{aligned} \tag{A.2}$$

More generally, the problems can be reduced to solving a set of coupled first order equations of the form:

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, ...) \tag{A.3}$$

## A.2 Initial value and boundary value problems

ODE problems can be divided into two categories, based on the constraints that we have on the functions. Initial value problems (IVP), are characterized by a set of constraints on the function at a certain value of $x$. For example, the function and all of its derivatives are known at $x = 0$, and we wish to find their values as a function of $x$.

Boundary value problems (BVP), are characterized by a set of constraints given at more than one value of $x$, and one typically wants to know the value of the function and its derivatives between the "boundaries". BVPs are generally more difficult to solve.

In this appendix, we will only look at some basic methods for solving differential equations, as it is quite a complex field. One of the main issues in these algorithms (in a real world application), is to control numerical stability and minimize round-off errors. We will only concern ourselves with some of the more basic algorithms for solving these equations.

## A.3 Initial value problems

In this section, we look at how to solve the general problem of a set of first order coupled differential equations:

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, ...) \tag{A.4}$$

where the functions $f_i$ are known, and the unknown functions, $y_i$, are known at a certain value $x = x_0$:

$$\begin{aligned} y_0(x_0) &= c_0 \\ y_1(x_0) &= c_1 \end{aligned} \tag{A.5}$$

### The Euler Method

The simplest way to solve IVP problems is to use the Euler method. Although this is not a recommended method, as it is easily unstable, it is very useful in understanding the basic idea behind more complex algorithms. The basic idea is given by re-writing a derivative

$$\frac{dy}{dx} = f(x, y) \tag{A.6}$$

as

$$\frac{\Delta y}{\Delta x} = f(x, y) \tag{A.7}$$

At a small distance $h = \Delta x$ from $x_0$, we thus have:

$$\begin{aligned} k_1 &= \Delta x f(x_0, y(x_0)) = h f(x_0, y(x_0)) \\ y(x_0 + h) &= y(x_0) + k_1 \end{aligned} \tag{A.8}$$

which yields an estimate of the functions at $x_0 + h$, which can be used iteratively to get the function at any value of $x$. It can be shown (by comparing with a power series expansion) that the error in the estimate is of order $h^2$.

## The Runge-Kutta method

The Runge-Kutta (RK) method builds upon the Euler method to yield a better estimate of $y(x)$ after the step. The second order RK method uses an estimate of the derivative of the function at the middle of the interval to correct the result from a simple Euler step.

$$\begin{aligned} k_1 &= hf(x_0, y_0) \\ k_2 &= hf(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1) \\ y(x_0 + h) &= y(x_0) + k_2 \end{aligned} \tag{A.9}$$

It can be shown that the error in $y$ is of order $h^3$ for this method. In general the $n$th order RK method has an error of order $h^{n+1}$.

By far the most common Runge-Kutta method is the 4th order method (RK4), given by:

$$\begin{aligned} \vec{k}_1 &= h\vec{f}\,(x_0, \vec{y}(x_0)) \\ \vec{k}_2 &= h\vec{f}\,(x_0 + \frac{1}{2}h, \vec{y}(x_0) + \frac{1}{2}\vec{k}_1) \\ \vec{k}_3 &= h\vec{f}\,(x_0 + \frac{1}{2}h, \vec{y}(x_0) + \frac{1}{2}\vec{k}_2) \\ \vec{k}_4 &= h\vec{f}\,(x_0 + h, \vec{y}(x_0) + \vec{k}_3) \\ \vec{y}(x_0 + h) &= \vec{y}(x_0) + \frac{1}{6}\vec{k}_1 + \frac{1}{3}\vec{k}_2 + \frac{1}{3}\vec{k}_3 + \frac{1}{6}\vec{k}_4 \end{aligned} \tag{A.10}$$

where we have generalized the $k_i$ to vectors, for a system of ODEs (each equation above is true by component).

**EXAMPLE A-1**

Solving for the motion of a simple harmonic oscillator with a friction term

The friction term is proportional to velocity (and in the direction opposite of motion):

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0$$

We can re-write this as:

$$\begin{aligned}\frac{dy_0}{dx} &= y_1 \\ \frac{dy_1}{dx} &= -\frac{1}{m}(ky_0 + cy_1)\end{aligned}$$

Here $y_0$ is really $x(t)$, the position of the mass as a function of time, and $y_1$ is really $\frac{dx}{dt}$, the speed as a function of $t$. Let's say that we know the position and velocity at $t = 0$, we can then use the RK4 method to solve for the position as a function of $t$.

## A.4 Python code to solve ODEs with IVPs

The following code is written in Python 3 and can be run in Windows, Linux, or Mac OS. The internet provides plenty of resources for finding out how to run Python on your own computer. In addition to python, you will need to install the python package called "matplotlib", which can be found on the internet or is also a part of the "scipy" package.

The code implements the RK4 stepper in a simple way (that is, the step-size is held constant). In order to use the code, you must first write the ODEs in the form of $n$ first order coupled ODEs:

$$\begin{aligned}\frac{dy_0}{dx} &= f_0(y_0, y_1, \dots, y_n, x) \\ \frac{dy_1}{dx} &= f_1(y_0, y_1, \dots, y_n, x) \\ &\dots \\ \frac{dy_n}{dx} &= f_n(y_0, y_1, \dots, y_n, x)\end{aligned}$$

The functions are then the following:

- `step()` - No need to modify, implements 1 RK4 step
- `solve()` - Use this to solve the ODE, you must pass it the following arguments:
  - `stepsize` - the size of the step to use for stepping the solution
  - `x0` - the initial value of the independent variable
  - `nSteps` - the number of steps to perform, so that the solution is found between `x0` and `x0` $+$ `nSteps` $\times$ `stepsize`
  - `derivs` - the derivatives (these are specified later)
  - `initialConditions` - this is a Python list (an array) with the initial values of all the $y_i$ at x0. The dimension of this array must be the same as there are unknown functions to solve. This is filled later in the code.
- `plot()` - this plots the results. It creates one plot, then the user must close the screen and the program will then show the next plot (going through all the $y_i$).
- `derivs()` - this function must be modified by the user to actually hold the $f_i$. It takes as arguments `yin`, which is a Python list (array) of the values of all the $y_i$, as well as the value of $x$. The user needs to modify the variable `dy`, which is a list containing all the $f_i$. In the example, `dy` is filled with the derivatives from the simple harmonic oscillator above.

In principle the code can be used without understanding too many of the details. The user needs to modify `dy` inside the `derivs()` function, set the `initialConditions` list, and decide on the stepping parameters when calling `solve()`.

```
#! /usr/bin/python

#from ROOT import TGraph, TCanvas
import matplotlib.pyplot as plt

#RK4 stepper
def step(stepsize, yin, x, derivs):
    dy=derivs(x,yin)
    k1=[]
    y2=[]
    for i in range(0,len(yin)):
        k1.append(stepsize*dy[i])
        y2.append(yin[i]+0.5*k1[i])

    dy=derivs(x+0.5*stepsize,y2)
    k2=[]
    y3=[]
    for i in range(0,len(yin)):
        k2.append(stepsize*dy[i])
        y3.append(yin[i]+0.5*k2[i])

    dy=derivs(x+0.5*stepsize,y3)
    k3=[]
    y4=[]
    for i in range(0,len(yin)):
        k3.append(stepsize*dy[i])
        y4.append(yin[i]+k3[i])

    dy=derivs(x+stepsize,y4)
    k4=[]
    yout=[]
    for i in range(0,len(yin)):
        k4.append(stepsize*dy[i])
        yout.append(yin[i]+k1[i]/6.0+k2[i]/3.0+k3[i]/3.0+k4[i]/6.0)

    return yout

def solve(stepsize,x0,nSteps,derivs,initialConditions):
    y=[]
    y.append([x0,initialConditions])
    print("Initial ", y)
    for i in range (0,nSteps):
      x=x0+i*stepsize
      #the new value is the one at the END of the step, so label it x+stepsize
```

```
      y.append([x+stepsize,step(stepsize,y[len(y)-1][1:][0],x,derivs)])
    return y

def plot(yx):
    ny=len(yx[1])
    for i in range(0,ny):
        y=[]
        x=[]
        for j in range (0,len(yx)):
            x.append(yx[j][0])
            y.append(yx[j][1][i])
        plt.plot(x,y)
        plt.show()

#problem-specific derivatives
def derivs(x,yin):
    m=1
    k=0.5
    c=2
    print("derivatives at ", x, " ", yin)
    dy=[yin[1],-1.0/m*(k*yin[0]+c*yin[1])]
    return dy

########################################
########################################
#initial conditions for each function
initialConditions=[1,0]
yx=solve(0.1,0.0,100,derivs,initialConditions)

#Do the drawing
plot(yx)
```

## A.5 Boundary Value Problems

Boundary value problems are much more difficult to solve than initial value problems, so we will only briefly cover them in this appendix. Boundary value problems (often “two point boundary value problems”) are characterized by the fact that we do not know the values of all of the dependent variables (y, the functions and derivatives) at one point. Rather, we know some of the values at one point and some of the values at the other point. This is the situation, for example, when solving for an electric field in space, given fixed electric potentials at some electrodes and a charge distribution.

There are two basic methods for solving these: “shooting” methods and “relaxation” methods. In the shooting methods, one guesses values for the unknown dependent variables at the starting point, then integrates them as if they were an IVP, and then corrects the initial guesses until the boundary

conditions at the other end are met. Some variants involve “shooting” from both sides until they match at some middle point. The tricky part of the algorithm is in calculating how to make the adjustments based on the errors achieved at the point that was aimed at.

Relaxation methods make initial guesses of the function at all points between the boundaries and then adjust those guesses until they are self-consistent.

## Relaxation Method

The relaxation method simplifies the differential equations into many “finite difference equations”. If there are $N$ points between the boundaries, spaced by a distance $d$, then at some point $k$ we approximately have:

$$\begin{aligned} x_{k+1} &= x_k + d \\ \frac{dy}{dx} &\sim \frac{y_{k+1} - y_k}{d} \end{aligned} \tag{A.11}$$

So a differential equation

$$\frac{dy}{dx} = f(x, y) \tag{A.12}$$

can be approximated, at each point, $k$, by:

$$(y_{k+1} - y_k) - df\left(x_k + \tfrac{1}{2}d, \tfrac{1}{2}(y_{k+1} + y_k)\right) = 0 \tag{A.13}$$

The $y_k$ and $f$ could also be vectors in the case of a higher order ODE.

We can also represent the second derivative with the finite difference as:

$$\frac{d^2y}{dx^2} \sim \frac{y_{k+1} - 2y_k + y_{k-1}}{d^2} \tag{A.14}$$

We will do a simple example here of solving Poisson’s equation in one dimension.

### EXAMPLE A-2

Poisson's equation

Poisson's equation describes the electric potential, given a charge distribution (Laplace's equation is a special case when the charge distribution is 0). Poisson's equation is:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon}$$

where $V$ is the electric potential, $\rho$ is the charge density, and $\epsilon$ is the permittivity of the material.

Let's say that we want to solve the equation in one dimension, in vacuum, between $x = 0$ and $x = L = 1$ m, where we have a charge density given by:

$$\rho(x) = 0.5 \times 10^{-8} + 0.1 \times 10^{-8} x$$

and we have the boundary conditions $V(x = 0\,\mathrm{m}) = 100$ V and $V(x = 1\,\mathrm{m}) = 200$ V.

We start by choosing a grid size, say $h = 0.01$ m. We then divide our interval up into $L/h + 1 = 101$ grid points, and attempt to solve for $V_k$ $(k = 0 \ldots 100)$, at each point on the grid. We will use the fact that (from our above approximation of the second derivative):

$$V_k = \frac{h^2 \frac{\rho}{\epsilon} + V_{k+1} + V_{k-1}}{2}$$

We will fix the values of $V$ at $k = 0$ and $k = 100$ to be equal to 100 V and 200 V, respectively, and (arbitrarily) set all the other $V_k = 0$.

We will iteratively calculate the $V_k$ using the above equation until they stop changing. This is implemented in the following code, which is in C++ and needs to be compiled against CERN's ROOT libraries.

```
#include <iostream>
#include <TGraph.h>
#include <TCanvas.h>
#include <math.h>
#include <vector>
using namespace std;
int main(int argc, char* argv[])
{
 //simple Poisson equation solver in 1D
 double Lx=1.0;
 double epsilon=8.85e-12;//permittivity of free space
 double V0=100;//voltage at x=0
 double VL=200;//voltage at x=L
 double h=0.01;//grid size
 int N = int(Lx/h)+1;//number of grid points, k=0..N-1 spanning [0,L]
 vector<double>V(N,0.);//potential, 0 everywhere
 //Boundary conditions:
 V[0]=V0;
 V[N-1]=VL;
 double diff=10;
 double maxdiff=1e-5;//stop when V changes by less than this
 double newV=0;
 double rhoeps=0;//rho/epsilon as a function of x
 double lastdiff=0;
 int MaxCount=50000;//to avoid infinite loop!
 int count=0;
 while(diff>maxdiff && count<MaxCount){
   diff=0;
   for(int i=1;i<N-1;i++){//don't include boundaries in the loop
     rhoeps=(0.5e-8+0.1e-8*(h*i))/epsilon;// x = i*h, so this gives
rho(x)/epsilon
     newV=0.5*(h*h*rhoeps+V[i+1]+V[i-1]);
     lastdiff=fabs(newV-V[i]);
     if(lastdiff >diff) diff=lastdiff;
     V[i]=newV;
   }
   count++;
 }
 if(count>=MaxCount){
   cout<<"Warning, did not converge, had to bail after "<<count<<"
iterations"<<endl;
 }
 cout<<"achieved precision of "<<diff<<endl;
 //Make a plot of the potential:
```

```
 TGraph g;
 g.SetMarkerStyle(8);
 for(int i=0;i<V.size();i++){
   g.SetPoint(i,i*h,V[i]);
 }
 TCanvas c1;
 g.Draw("ap");
 c1.Print("potential.gif");
 return 0;
}
```

In this simple program, we first start by creating a vector, $V$, to hold our answer at each point in $x$, separated by $h$. We fix the boundary value at the edges of $V$, and set all other values of $V = 0$. We then loop through the values of $V$ and update them according to our equation. We keep track of the difference between the previous value and the updated value. Once that difference is small enough, we claim to have converged (as the difference, we take that of the $V$ which changed the most, tracked by `lastdiff` and `diff`).