ThesisBook

Mélanie

3/15/23

Table of contents

Preface

This is a Quarto book. tests

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

2 Motivation

Let A be non singular square matrix of dimension $n \geq 1$ and let $b \in \mathbb{R}^n$. We consider the linear system $Ay = b$, where $y \in \mathbb{R}^n$. The system has for unique solution $y^* = A^{-1}b$. This is a fundamental problem to solve in numerical analysis, and there are numerous numerical methods to solve this, whether they are direct methods or iterative methods. In this thesis, we consider an iterative method. We consider the initial value problem

$$y'(t) = Ay(t) - b, \;\; y(0) = y_0$$

where $y_0 \in \mathbb{R}^n$ and $t \in \mathbb{R}$. Multiplying the equation by e^{-At}, where e^{-At} is the usual matrix exponential, and rearranging the terms yields

$$e^{-At}y'(t) - Ae^{-At}y(t) = e^{-At}b$$

We recognise on the left hand side the derivative of the product $e^{-At}y(t)$, and thus, by the fundamental theorem of calculus,

$$\left[e^{-Au}y(u)\right]_0^t = \int_0^t -e^{-Au}b \, du.$$

Multiplying by $A^{-1}A$ inside the integral in the LHS, we get

$$e^{-At}y(t) - y_0 = A^{-1}\left[e^{-Au}\right]_0^t b = A^{-1}e^{-At}b - A^{-1}b.$$

Multiplying each side by e^{At} and rearranging the terms we get an expression for $y(t)$,

$$y(t) = e^{At}(y_0 - A^{-1}b) + A^{-1}b.$$

Note that each of those step can be taken backward , which means that the solution we have is unique. We have thus proved

Theorem 2.1. *Let A be a non singular, square matrix of dimension $n \geq 1$, $b \in \mathbb{R}^n$ a vector, and consider the initial value problem*

$$y'(t) = Ay(t) - b,\ y(0) = y_0 \tag{2.1}$$

where $t \to y(t)$ is a function from $\mathbb{R}$ to $\mathbb{R}^n$. Then the problem has a unique solution in the form of

$$y(t) = e^{At}(y_0 - y^*) + y^*,$$

where $y^* = A^{-1}b$, and e^{At} is defined using the usual matrix exponential.

Let $\lambda_1, \lambda_2, \dots, \lambda_n$ be the (not necessarly distinct) eigenvalues of A, write $\lambda_i = a_i + iy_i$, where $a_i, b_i \in \mathbb{R}$ are respectively the real part and the imaginary parts of the i^{th} eigenvalue. The following holds

Theorem 2.2. *$y(t) \to y^*$ as $t \to +\infty$ for any initial value y_0 if and only if, for all $i = 1, \dots, n$, $a_i < 0$, that is, all the eigenvalues of A have a strictly negative real part.*

Proof. (In the diagonalisable case)

We assume that A is diagonalisable. Write $A = P\Delta P^{-1}$ where Δ is diagonal.

$$\Delta = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}$$

Then $e^{At} = Pe^{\Delta t}P^{-1}$, where

$$e^{\Delta t} = \begin{pmatrix} e^{\lambda_1 t} & & & \\ & e^{\lambda_2 t} & & \\ & & \ddots & \\ & & & e^{\lambda_n t} \end{pmatrix}$$

Let $z(t) = P^{-1}(y(t) - y^*)$, where $y(t)$ is the unique solution to Equation 2.1 for some arbitrary initial value y_0.

Since P is non singular, $y(t) \to y^*$ if and only if $z(t) \to 0$. We have

$$z(t) = P^{-1}e^{At}(y_0 - y^*)$$

We note that $P^{-1}e^{At} = e^{\Delta t}P^{-1}$, thus

$$z(t) = e^{\Delta t}P^{-1}(y_0 - y^*).$$

Looking at the i^{th} element $z(t)_i$, we have

$$|z(t)_i| = e^{a_i t}\left(P^{-1}(y_0 - y^*)\right)_i$$

where $a_i = \Re[\lambda_i]$. Clearly, if $a_i < 0$, $z(t)_i \to 0$ as $t \to +\infty$. If this holds for any $i = 1, \dots, n$, then $z(t) \to 0$ as $t \to +\infty$. This proves ($\Leftarrow$).

This is also a necessary condition. Indeed, since y_0 is arbitrary, we can chose it so that $P^{-1}(y_0 - y^*) = (1, \dots, 1)^T$. Then $z(t) = (e^{\lambda_1 t}, e^{\lambda_2 t}, \dots, e^{\lambda_n t})^T$ which converges to 0 only if all the eigenvalues have a strictly negative real part.

□

Remark. A general proof is available on (Bellman 1953, chap. 1)

We now go back to the original problem of solving the linear system $Ay = b$. If all the eigenvalues of A have a strictly negative real part, then, any numerical solver for the initial value problem $y'(t) = Ay(t) - b$ with $y(0) = y_0$ where t is some pseudo-time variable also becomes an iterative solver for the linear system $Ay = b$, as $y(t) \to y^*$.

Remark. If all the eigenvalues of A have a strictly positive real part, then we can simply solve $y' = (-A)y - (-b) = -Ay + b$ instead.

3 A test problem - Convection diffusion equation

As a test case for the solver, we consider the steady state convection-diffusion equation.

$$u_x = bu_{xx} + 1$$

where b is some physical parameter and $u(x)$ is defined on the interval $[0, 1]$. The boundary condition are given by $u(0) = u(1) = 0$. This equation has an analytical solution that is given by

$$u(x) = x - \frac{e^{-(1-x)/b} - e^{-1/b}}{1 - e^{-1/b}}.$$

We are however interested in solving this numerically, with a finite difference approach. We partition the interval $[0, 1]$ into equidistant points $x_i, i = 0, \dots n$. We note the distance between each points as Δx, and we have $u(x_0) = u(0) = 0$ and $u(x_n) = u(1) = 0$. We use the notation $u^i = u(x_i)$. We approximate, for $i \geq 1$ the derivative

$$u_x^i = \frac{u^i - u^{i-1}}{\Delta x}$$

and the second order derivative is approximated by

$$u_{xx}^i = \frac{u^{i+1} - 2u^i + u^{i-1}}{\Delta x^2}$$

Note that the first derivative is approximated backward in time. For $i = 1, \dots, n-1$, we thus have the approximation

$$u_x^i = \frac{u^i - u^{i-1}}{\Delta x} = b\frac{u^{i+1} - 2u^i + u^{i-1}}{\Delta x^2} + 1$$

This can be given in matrix format by letting $u = (u^1, \dots, u^{n-1})^T$

$$Au = Bu + e$$

where $e = (1, 1, \dots, 1)^T$,

$$A = \frac{1}{\Delta x} \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \end{bmatrix}$$

and

$$B = \frac{b}{\Delta x^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix}.$$

With $M = A - B$, we have to solve the linear system

$$Mu = e$$

where M is a square matrix of dimension $(n-1) \times (n-1)$ and e is the one vector of dimension $n - 1$.

Remark. (To make better but I think it works) It is apparent that M is diagonally dominant. Since all elements of the diagonal are positive, then so are the eigenvalues real part. Assuming M is non singular, we have that $-M$ is stable.

To solve this linear system, we use the method highlighted before. We thus introduce a pseudo time variable t and we consider the ODE.

$$u'(t) = e - Mu(t)$$

Assuming M is non singular, we can use Theorem 2.2 to guarantee that the ODE will converge to a steady state independently of the initial value we chose. In the next chapter, we will apply the Runge-Kutta scheme we saw earlier to the problem and will see how parameters changes results.

4 Runge Kutta solver applied to the test problem

We now have to solve the ODE $u' = e - Mu$ where M depends on the problem parameters b and $\Delta x = 1/(n+1)$, where n is the chosen number of subdivisions of $[0,1]$. Since we are only interested on the asymptotic behavior of u, we only need to care about the stability of the numerical solver we wish to use. We consider the following RK scheme with two stages.

blablbal

This solver has two parameters Δt and α. The objective is for the solver to converge to a steady state solution as fast as possible. Set $u_0 = u(0) = e$ as an initial value. We define the relative residual after k steps as

$$r_k = ||Mu_k - e||/||e||.$$

where $||.||$ is the 2-norm. since $||e|| = \sqrt{n-1}$,

$$r_k = \frac{||Mu_k - e||}{\sqrt{n-1}}$$

If the solver we chose is stable, then $||r_k|| \to 0$ as $k \to \infty$. We define now the convergence at step n to be the ratio of residual at step k and $k-1$. That is

$$c_k = \frac{||r_k||}{||r_{k-1}||} = \frac{||Mu_k - e||}{||Mu_{k-1} - e||}$$

where $||.||$ is the 2-norm.

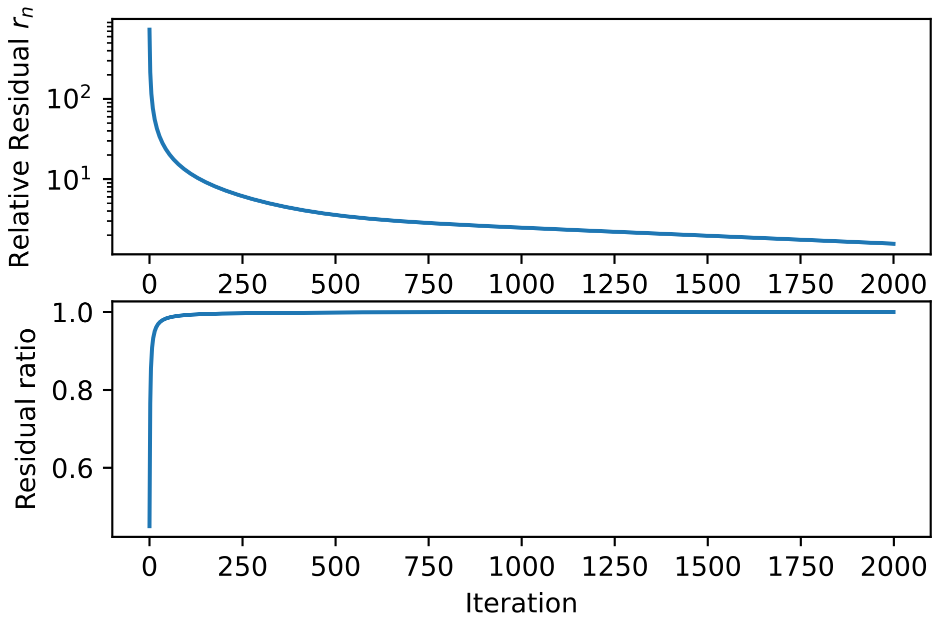


Figure 4.1: Evolution of the residual norm over iteration, with problem parameters $n = 100$ and $b = 0.5$, and RK parameters $\Delta t = 0.00009$ and $\alpha = 0.3$.

5 Testing ground for bachelor thesis

```python
import numpy as np
import matplotlib.pyplot as plt

class testProblem:
## Define it as
    def __init__(self,b,n) -> None:
        self.n = n
        self.b = b
        self.deltaX = 1 / (n+1)
        self.M = self.buildM(b,n,self.deltaX)
        self.e = self.buildE(n, self.deltaX)


    def buildM(self,b,n,deltaX):
        """
        we go from u0 to u(n+1).
        """
        deltaX = 1 / (n+1)
        A = deltaX *(np.eye(n) -1 * np.eye(n,k = -1))
        B = b* (-2*np.eye(n) + np.eye(n, k = -1) + np.eye(n,k=1))
        return A-B

    def buildE(self,n,deltaX):
        return deltaX**2 *np.ones(n)

    def f(self,y):
        return self.e - self.M@y

    def oneStepSmoother(self,y,t,deltaT,alpha):
        """
        Perform one pseudo time step deltaT of the solver for the diff eq
        y' = e - My = f(y). .
        """
        k1 = self.f(y)
        k2 = self.f(y + alpha*deltaT*k1)
        yNext = y + deltaT*k2
        return yNext

    def findOptimalParameters(self):
        #This is where the reinforcement learning algorithm
        #take place in
        return 0 , 0


    def mainSolver(self,n_iter = 10):
        """ Main solver for the problem, calculate the approximated solution
        after n_iter pseudo time steps. """
        resNormList = np.zeros(n_iter+1)
        t = 0
        #Initial guess y = e
        y = np.ones(e)
        resNormList[0] = np.linalg.norm(self.M@y-self.e)
        ##Finding the optimal params
        alpha, deltaT = self.findOptimalParameters()
        ##Will need to be removed, just for debugging
        alpha = 0.5
        deltaT = 0.00006
        #For now, we use our best guess
        for i in range(n_iter):
            y = self.oneStepSmoother(y,t,deltaT,alpha)
            t += deltaT
            resNorm = np.linalg.norm(self.M@y - self.e)
            resNormList[i+1] = resNorm
        return y , resNormList

    def mainSolver2(self,alpha, deltaT, n_iter = 10):
        """ Like the main solver, except we give
        the parameters explicitely """
        resNormList = np.zeros(n_iter+1)
        t = 0
        #Initial guess y = e
        y = np.ones(n)
        resNormList[0] = np.linalg.norm(self.M@y-self.e)
        #For now, we use our best guess
        for i in range(n_iter):
            y = self.oneStepSmoother(y,t,deltaT,alpha)
            t += deltaT
            resNorm = np.linalg.norm(self.M@y - self.e)
            resNormList[i+1] = resNorm
        return y , resNormList
```

We now have everything we need to get going, let's plot the residual norm over iteration as a first test

```
#Create the object
b = 0.5
n = 100

alpha = 0.138138138138813813
deltaT = 3.5143143143143143
convDiffProb = testProblem(b,n)
y, resNormList = convDiffProb.mainSolver2(0.093093,5.6003,20)

x = np.linspace(0,1,n+2) #Create space
yTh = np.zeros(n+2)
yTh[1:n+1] = np.linalg.solve(convDiffProb.M,convDiffProb.e)

yApprox = np.zeros(n+2)
yApprox[1:n+1] = y
fig, (ax1,ax2) = plt.subplots(1,2)

ax1.plot(resNormList)
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Residual norm")
ax1.set_yscale('log')

ax2.plot(x,yTh,label = 'Discretised solution')
ax2.plot(x,yApprox,label = "iterative solution")
ax2.legend()

fig.show()
```

```
/tmp/ipykernel_5673/2529022355.py:27: UserWarning:

Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI
```

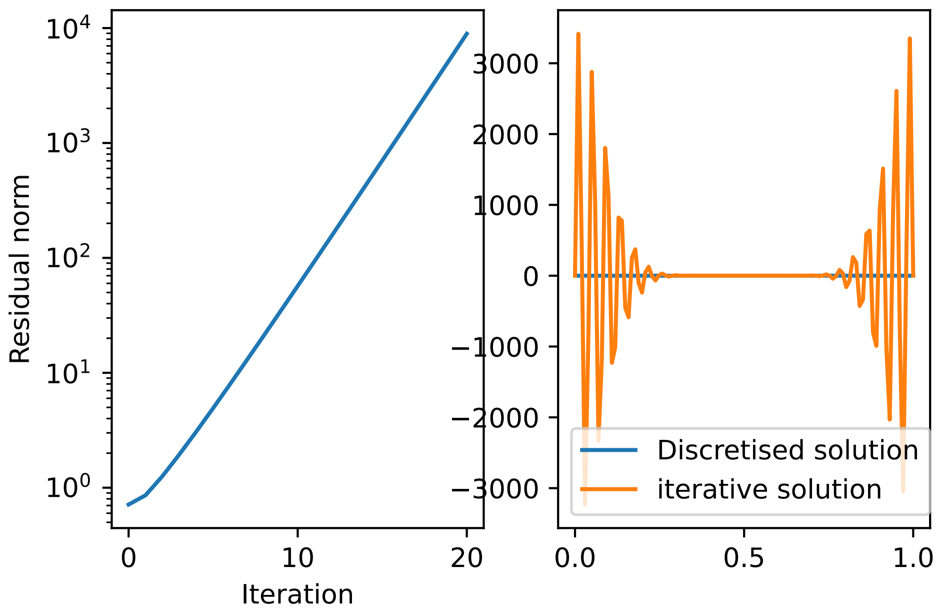


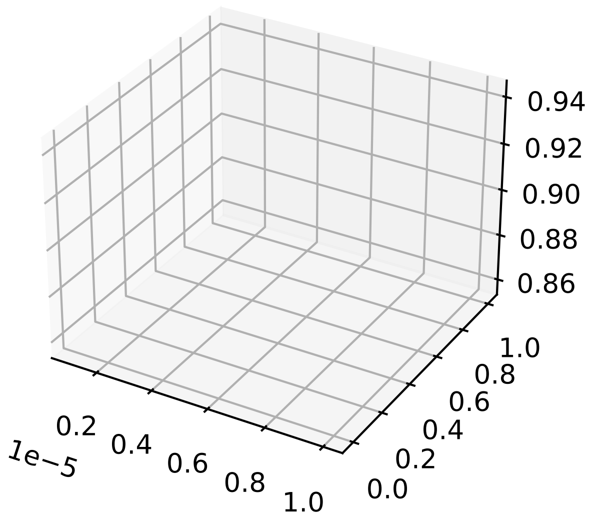
Figure 5.1: Evolution of the residual norm over a number of iteration.

```
from matplotlib import cm
from matplotlib.ticker import LinearLocator

def resRatio(resNormList):
    return resNormList[-1] / resNormList[-2]



l = 100
deltaTgrid = np.linspace(0.000001,0.00001,l)
alphaGrid = np.linspace(0,1,l)

deltaTgrid, alphaGrid = np.meshgrid(deltaTgrid,alphaGrid)

resRatioGrid2 = np.zeros((l,l))

for i in range(l):
    for j in range(l):
        #print('alpha', alphaGrid[j,0])
        #print('deltaT', deltaTgrid[0,i])
        y , resNormList = convDiffProb.mainSolver2(alphaGrid[j,i],deltaTgrid[j,i],10)
```

```
            ratio = resRatio(resNormList)
            #print('ratio', ratio)
            resRatioGrid2[j,i] = resRatio(resNormList)

fig, ax = plt.subplots(subplot_kw={"projection": "3d"})


clippedRatio = np.clip(resRatioGrid2,0.8,0.9)
surf = ax.contour(deltaTgrid,alphaGrid,clippedRatio,levels = [0.8,0.85,0.9])

transformedContour = np.log(1/(1+np.exp(-clippedRatio+1)))



print(np.nanmin(resRatioGrid2))
print(np.argmin(resRatioGrid2))
```

```
0.9999898995194059
99
```

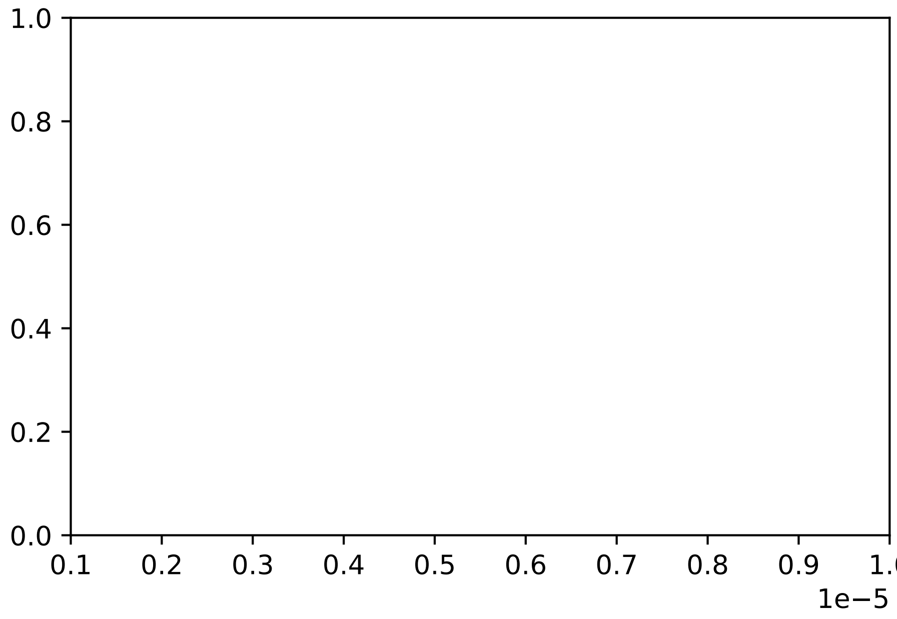
```
/tmp/ipykernel_5673/1941845570.py:30: UserWarning:

No contour levels were found within the data range.
```



Contour plot

```
fig, ax = plt.subplots()
cp = ax.contour(deltaTgrid,alphaGrid,resRatioGrid2,levels = [0.83,0.86,0.88,0.9,1], cmap=c
ax.clabel(cp)
#ax.view_init(elev = 90,azim = 150)
plt.show()
```

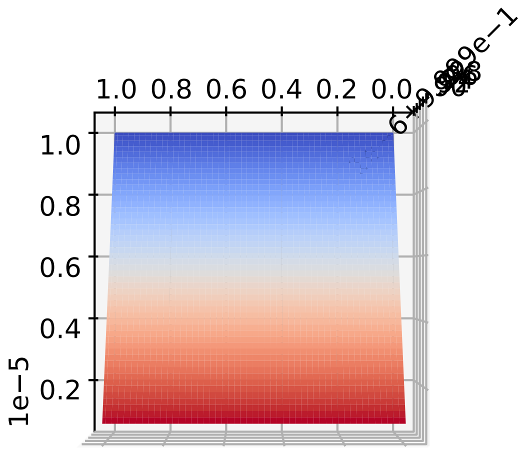
```
/tmp/ipykernel_5673/383841379.py:2: UserWarning:

No contour levels were found within the data range.

/tmp/ipykernel_5673/383841379.py:2: UserWarning:

The following kwargs were not used by contour: 'linewidth'
```



Surface plot

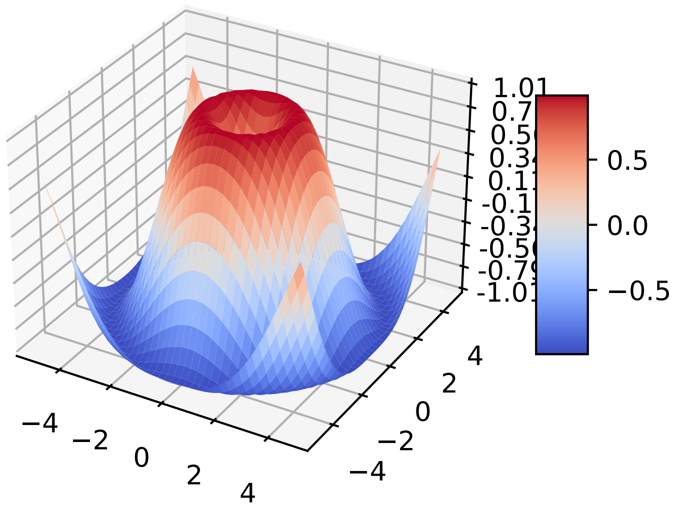
```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(deltaTgrid,alphaGrid,np.clip(resRatioGrid2,0.5,1), cmap=cm.coolwarm, linew
ax.view_init(elev = 90,azim = 180)
```

```
plt.show()
```



```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```



```
import sys
print(sys.executable)
```

```
/home/melanie/anaconda3/bin/python
```

```
import numpy as np
import matplotlib.pyplot as plt
```

Necessary functions go here.

```
def RK2(f,y,t,deltaT,alpha,**args):
    """Second order family of Rk2
    c = [0,alpha], bT = [1-1/(2alpha), 1/(2alpha)] , a2,1 = alpha """
    k1 = f(t,y,**args)
    k2 = f(t + alpha*deltaT, y + alpha*deltaT*k1,**args)
    yNext = y + deltaT*(k1*(1-1/(2*alpha)) + k2 * 1/(2*alpha))
    return yNext

def buildM(b,n):
    """
    we go from u0 to u(n+1).
    """
    deltaX = 1 / (n+1)
    A = 1/deltaX *(np.eye(n) -1 * np.eye(n,k = -1))
    B = b/deltaX**2 * (-2*np.eye(n) + np.eye(n, k = -1) + np.eye(n,k=1))
    return A-B

def buildE(n):
    return np.ones(n)

def f(t,y,M,e):
    return e - M@y


def mainSolver(deltaT, alpha,b,f = f,n_iter = 10,n_points=100):
    t = 0
    e = buildE(n_points)
    M = buildM(b,n_points)
    #First guess
    y = np.copy(e)
    resNorm = np.linalg.norm(M@y -e)

    for i in range(n_iter):
        y = RK2(f,y,t,deltaT,alpha,M = M,e = e)
        t += deltaT
        lastResNorm , resNorm = resNorm ,  np.linalg.norm(M@y - e)
    return resNorm / lastResNorm
```

```
mainSolver(0.0001,0.5,0.5)
```

```
0.9678775609609744
```

To facilitate everything, we discretise the space with 100 interior points only, and with parameter $b = 0.5$.

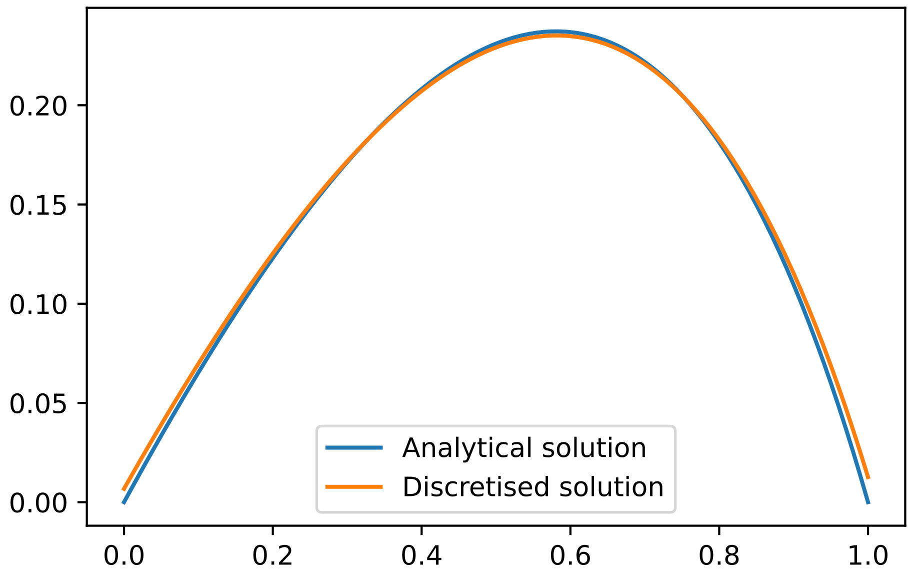
This is how the solution looks like with the discretisation

```
b = 0.5
n = 100

M = buildM(b,n)
```

```
e = buildE(n)

x = np.linspace(0,1,n+2)
x2 = np.linspace(0,1,n)

analyticSol = x - (np.exp(-(1-x)/b)-np.exp(-1/b))/(1-np.exp(-1/b))
u = np.linalg.solve(M,e)

plt.plot(x,analyticSol,label = 'Analytical solution')
plt.plot(x2,u,label = 'Discretised solution')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f69d6f22700>
```

Discretised solution vs analytical solution

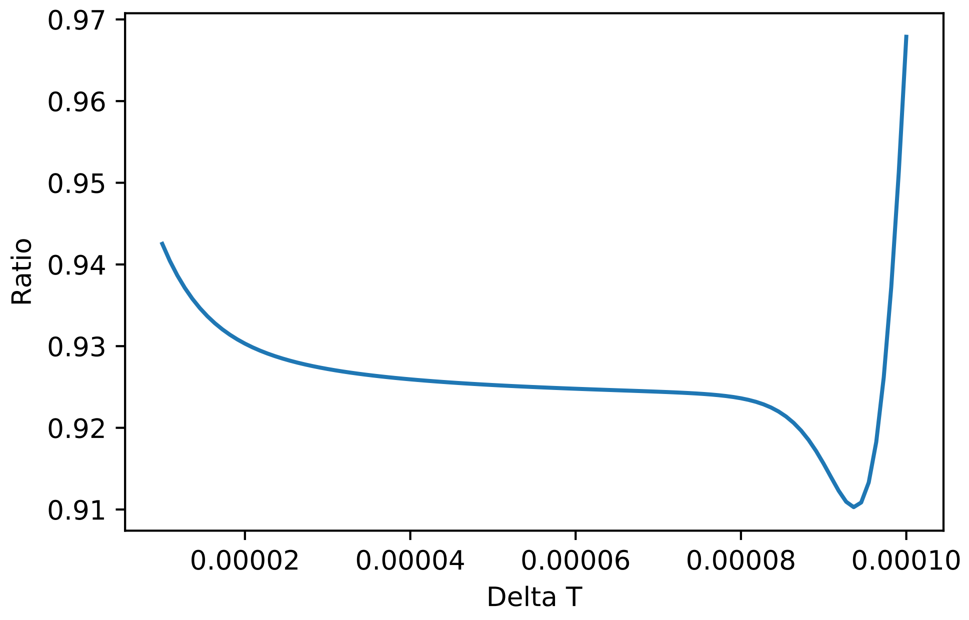


How would changing the parameters affect the residuals ratio after 10 iterations?

```
deltaTGrid = np.linspace(0.00001,0.0001,100)

ratio = np.zeros(100)
i = 0
for deltaT in deltaTGrid:
    ratio[i] = mainSolver(deltaT,0.6,0.5)
    i+=1

plt.plot(deltaTGrid,ratio)
plt.xlabel('Delta T')
plt.ylabel('Ratio')
```

```
Text(0, 0.5, 'Ratio')
```

Impact of the choice of time step with the residual ratios.

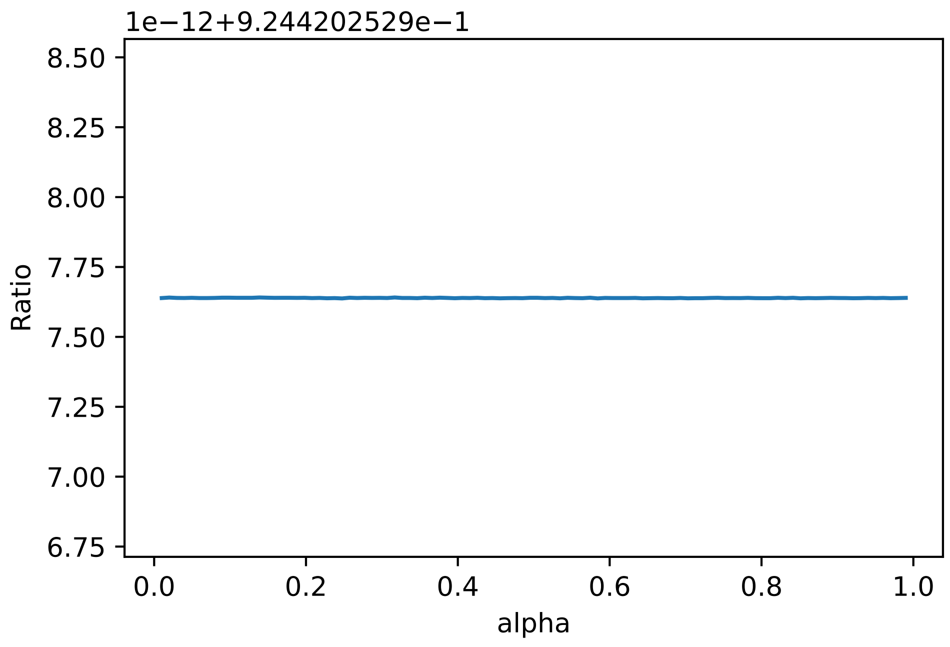


How would changing the RK parameter change the residual ratio after 10 iterations? Here we take the optimal delta T we found earlier.

```
    #fig-cap: Changing alpha does not do much...
    alphaGrid = np.linspace(0.01,0.99,100)

    ratio = np.zeros(100)
    i = 0
    for alpha in alphaGrid:
        ratio[i] = mainSolver(0.00007,alpha,0.5)
        i+=1
```

```
plt.plot(alphaGrid,ratio)
plt.xlabel('alpha')
plt.ylabel('Ratio')
```

```
Text(0, 0.5, 'Ratio')
```



Pendulum test

```
def f(t,y):
    g = 9.81
    l = 1
    f1 = y[1]
    f2 = -g/l* np.sin(y[0])
    return np.array([f1,f2])



#Pendulum
deltaT = 0.01
t_min = 0
```

```
n = 1000
t = t_min
tArray = np.zeros(n+1)
tArray[0] = t
y = np.array([np.pi/2,0])
yArray = np.zeros((n+1,2))
yArray[0] = y
for i in range(n):
    y = RK2(f,y,t,deltaT,0.9)
    t+=deltaT
    tArray[i+1] = t
    yArray[i+1] = y


plt.plot(tArray,yArray[:,0])
```

6 Summary

In summary, this book has no content whatsoever.

References

Bellman, Richard Ernest. 1953. *Stability Theory of Differential Equations /.* New York : McGraw-Hill,.