

ThesisBook

Mélanie

3/15/23

Table of contents

Preface	3
1 Introduction	4
2 Motivation	5
3 Testing ground for bachelor thesis	8
4 Summary	25
References	26

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

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

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

$$y'(t) = Ay(t) - b, \quad y(0) = y_0 \quad (2.1)$$

where $t \rightarrow y(t)$ 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 necessarily 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. $y(t) \rightarrow y^*$ as $t \rightarrow +\infty$ 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, $y(t) \rightarrow y^*$ if and only if $z(t) \rightarrow 0$. 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

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

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

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 Testing ground for bachelor thesis

```
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 explicitly """
    y = np.array([0.00719735, 0.01434065, 0.02142834, 0.02845879, 0.03543034,
0.04234128, 0.04918987, 0.05597432, 0.06269281, 0.06934346,
0.07592437, 0.08243356, 0.08886904, 0.09522877, 0.10151064,
0.10771254, 0.11383227, 0.11986762, 0.1258163 , 0.13167601,
0.13744437, 0.14311899, 0.1486974 , 0.15417709, 0.15955552,
0.16483009, 0.16999815, 0.175057 , 0.1800039 , 0.18483606,
0.18955064, 0.19414475, 0.19861545, 0.20295975, 0.20717461,

```

```

0.21125694, 0.21520361, 0.21901143, 0.22267715, 0.22619749,
0.22956911, 0.23278861, 0.23585255, 0.23875743, 0.24149971,
0.24407579, 0.246482 , 0.24871466, 0.25076999, 0.25264419,
0.25433339, 0.25583367, 0.25714106, 0.25825152, 0.25916098,
0.25986529, 0.26036025, 0.26064162, 0.26070509, 0.26054628,
0.26016077, 0.25954408, 0.25869166, 0.25759892, 0.25626119,
0.25467375, 0.25283181, 0.25073051, 0.24836496, 0.24573017,
0.2428211 , 0.23963265, 0.23615963, 0.23239681, 0.22833888,
0.22398045, 0.21931606, 0.2143402 , 0.20904726, 0.20343156,
0.19748735, 0.1912088 , 0.18458999, 0.17762494, 0.17030756,
0.16263169, 0.15459109, 0.14617941, 0.13739022, 0.12821701,
0.11865315, 0.10869192, 0.09832652, 0.08755003, 0.07635541,
0.06473555, 0.05268319, 0.04019099, 0.02725147, 0.01385704])
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 = 0.13813813813813813
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_7622/2529022355.py:27: UserWarning:

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

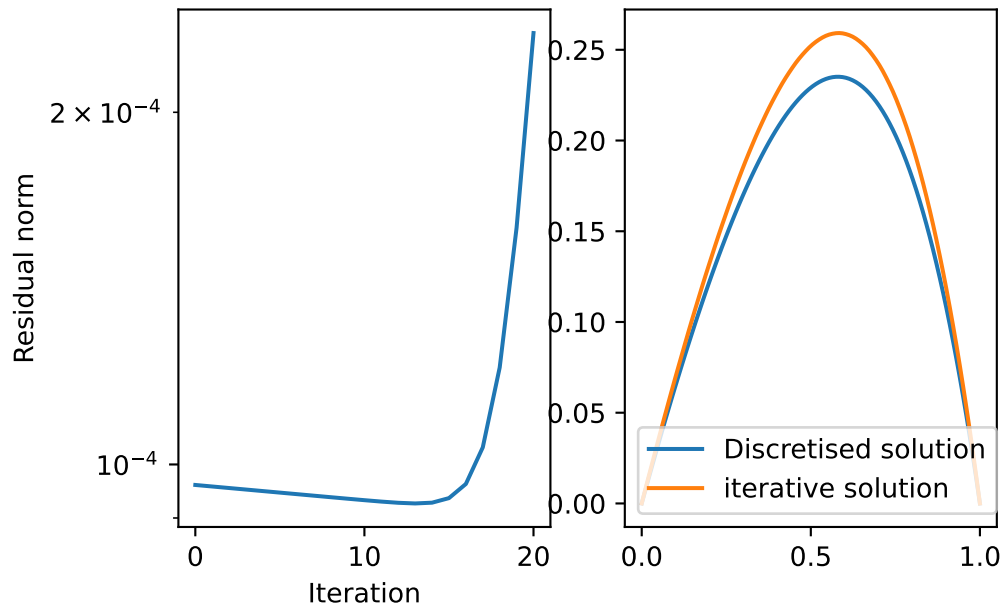


Figure 3.1: Evolution of the residual norm over a number of iteration.

```

from matplotlib import cm
from matplotlib.ticker import LinearLocator

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

l = 100
deltaTgrid = np.linspace(0.9,10,l)
alphaGrid = np.linspace(0,1,l)

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

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

for i in range(l):
    print(i)
    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.argmax(resRatioGrid2))

```

0
1

2
3
4
5
6
7
8
9
10

11
12
13
14
15
16
17
18
19

20
21
22
23
24
25
26
27
28
29

30
31
32
33
34
35
36
37
38
39

40
41
42
43
44
45
46
47
48
49

50
51
52
53
54
55
56
57
58
59

60
61
62
63
64
65
66
67
68

69
70
71
72
73
74
75
76
77
78

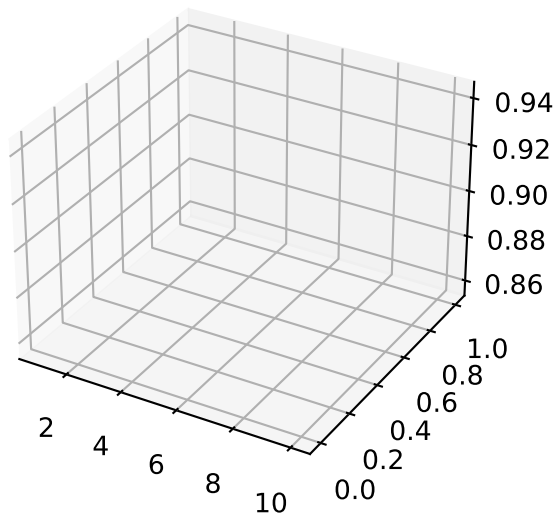
79
80
81
82
83
84
85
86
87
88

89
90
91
92
93
94
95
96
97
98

99
0.9971231634706321
952

/tmp/ipykernel_7622/930702461.py:31: UserWarning:

No contour levels were found within the data range.

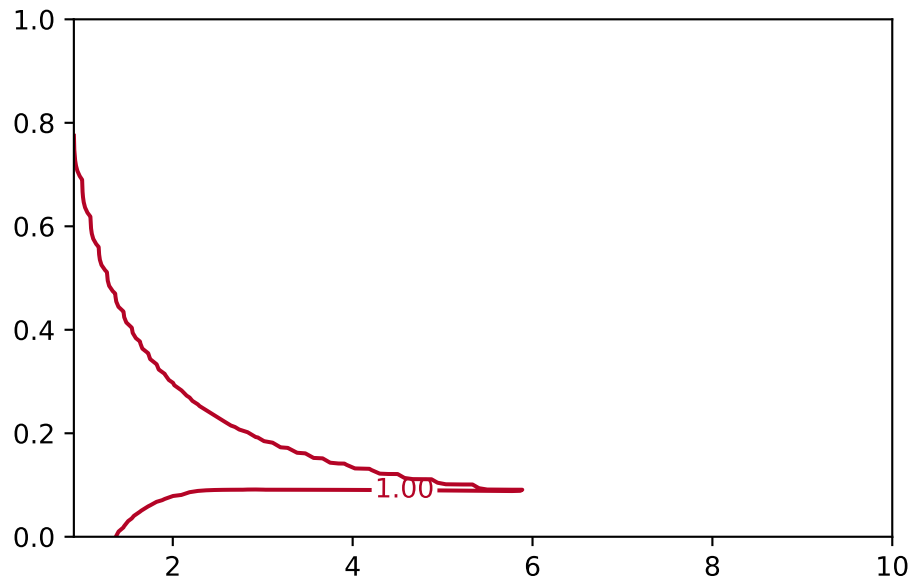


Contour plot

```
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()
```

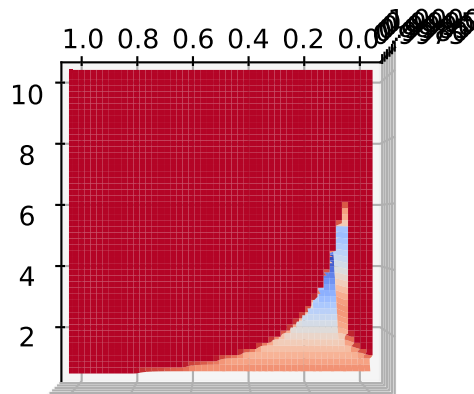
/tmp/ipykernel_7622/383841379.py:2: UserWarning:

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, linewidth=0)
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)

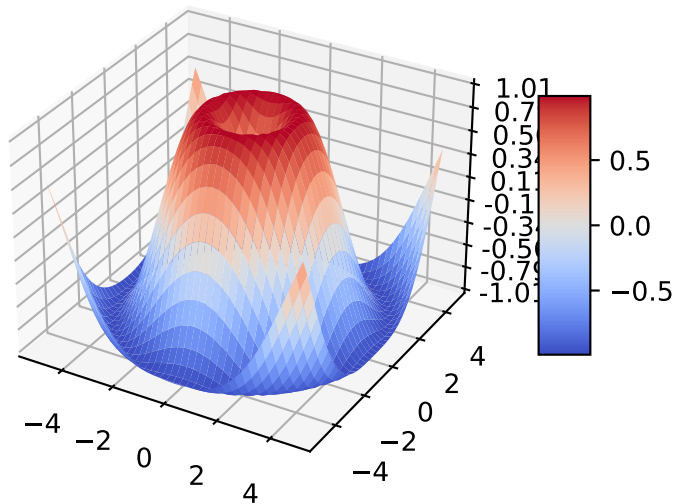
# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                      linewidth=0, antialiased=False)

# 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)

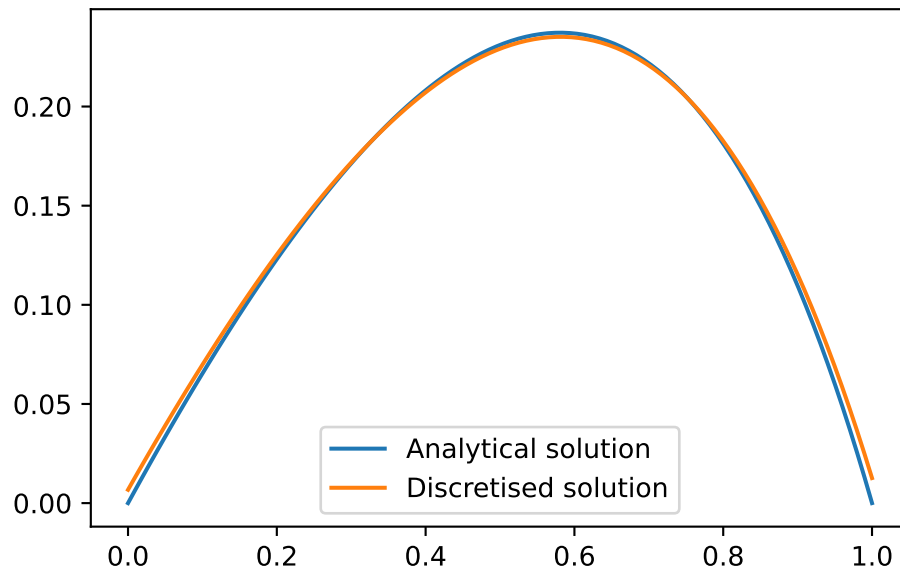
analyticSol = x - (np.exp(-(1-x)/b)-np.exp(-1/b))/(1-np.exp(-1/b))
u = np.linalg.solve(M,e)

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

```

<matplotlib.legend.Legend at 0x7fd1f15c41c0>

Discretised solution vs analytical solution



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

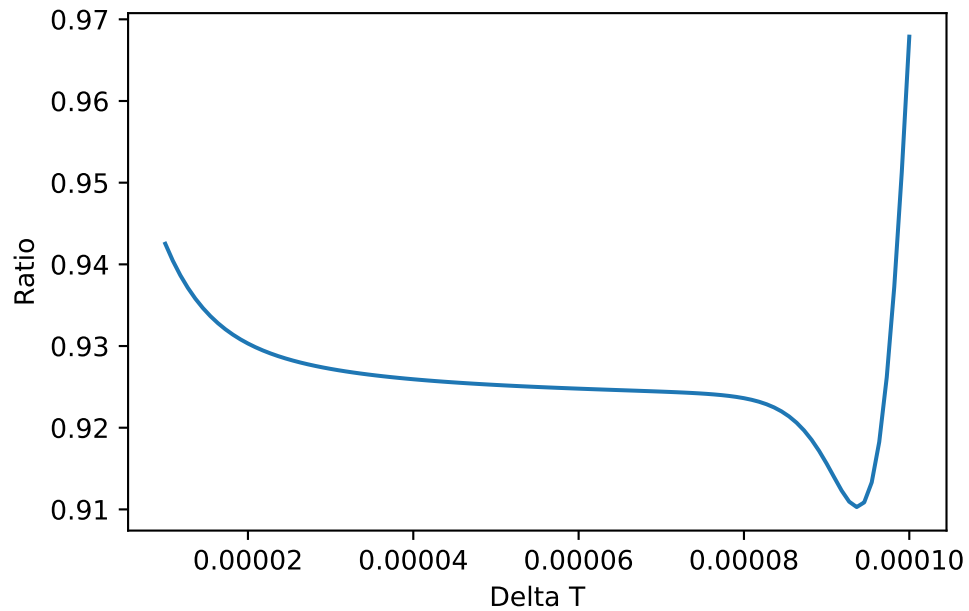
```
deltaTGrid = np.linspace(0.00001,0.0001,100)

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.



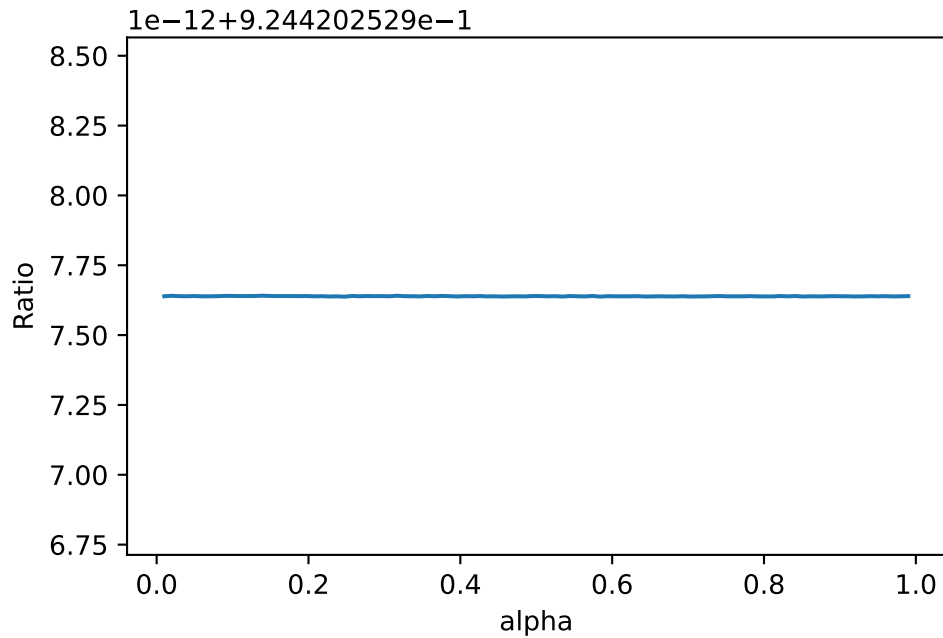
How would changing the RK parameter change the residual ratio after 10 iterations? Here we take the optimal delta T we found earlier.

```
#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])
```


4 Summary

In summary, this book has no content whatsoever.

References

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