Testing ground for bachelor thesis

Mélanie Fournier

1 Explicit RK2 and stability function

```
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 """
    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_6136/2529022355.py:27: UserWarning:

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

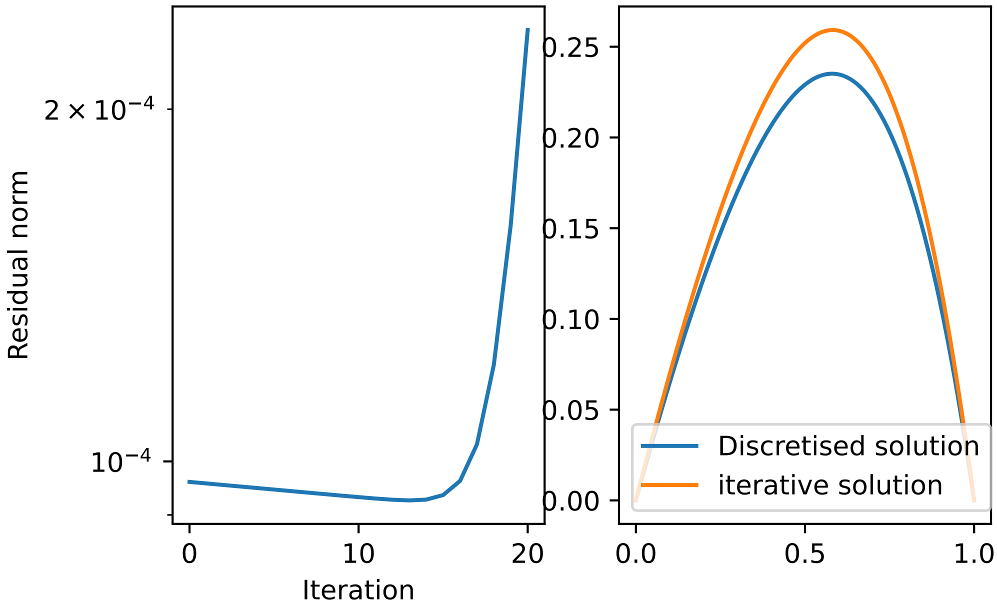


Figure 1: Evolution of the residual norm over a number of iteration.

```
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)
```

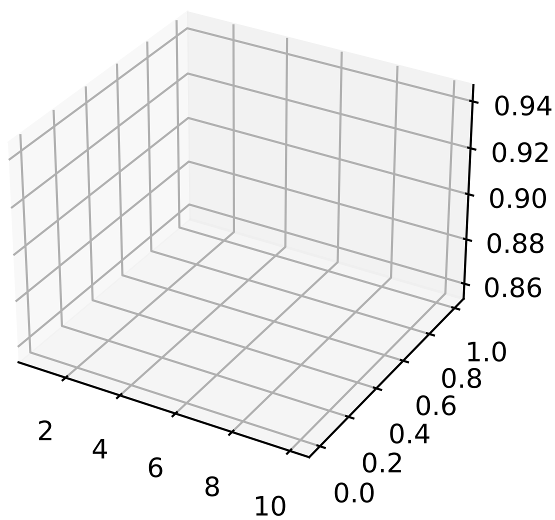
```python
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})


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
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
```

```
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

```
0.9971231634706321
952
```

```
/tmp/ipykernel_6136/930702461.py:31: UserWarning:

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

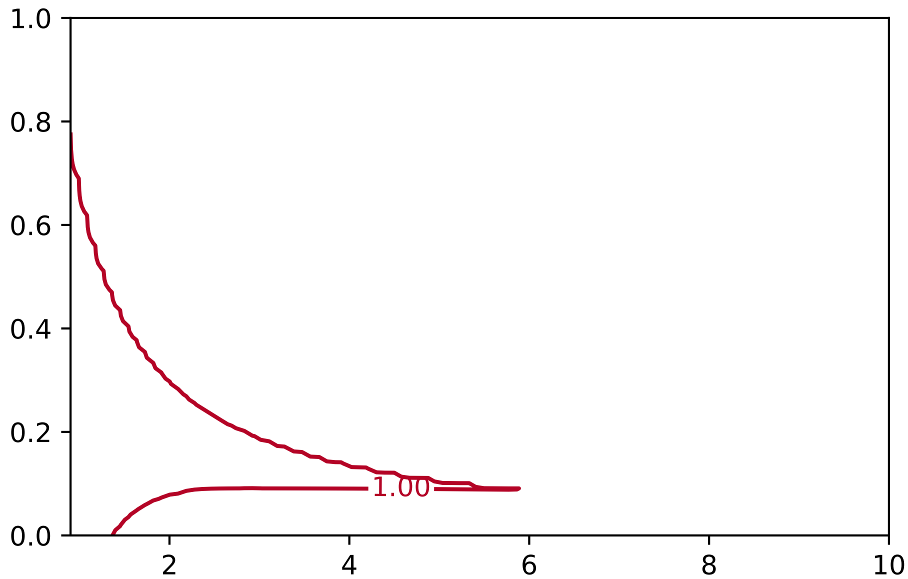


Contour plot

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

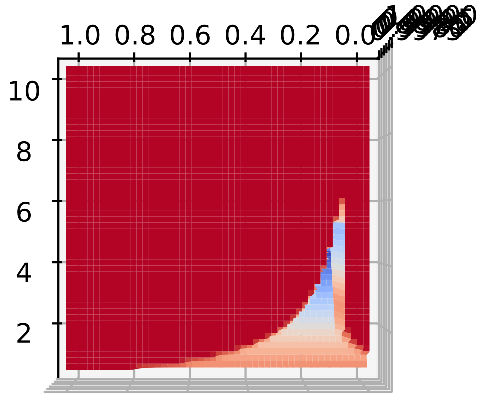
```
plt.show()
```

```
/tmp/ipykernel_6136/383841379.py:2: UserWarning:

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

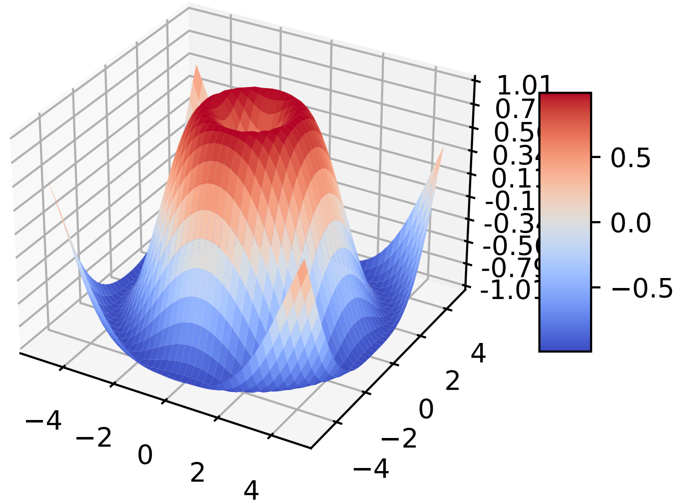


Surface plot

```
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)
```

```python
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
```

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

```
0.9678775609609744
```

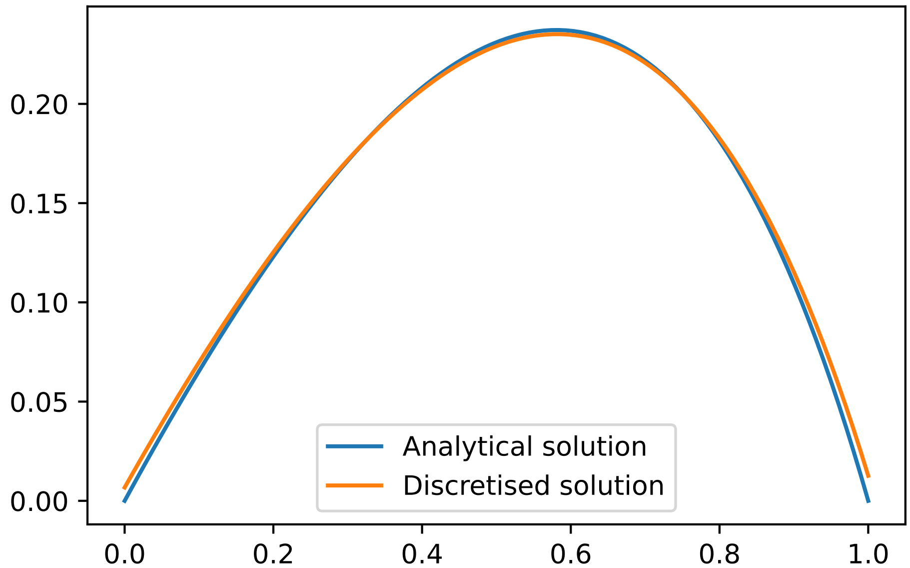
To facilitate everything, we discretise the space with 100 interior points only, and with parameter $b = 0.5$.

This is how the solution looks like with the discretisation

```python
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 0x7fc24c5de7c0>
```

Discretised solution vs analytical solution

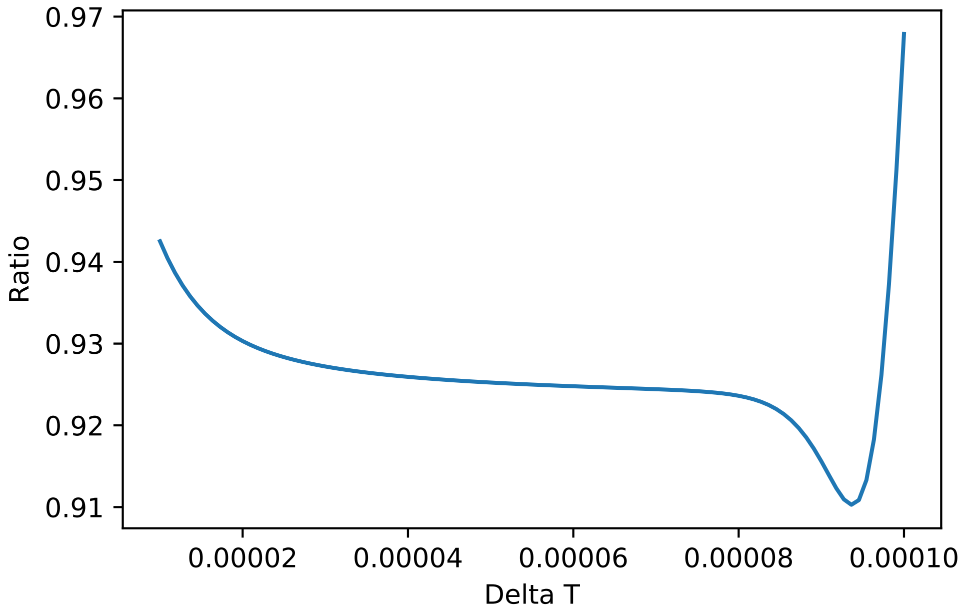


How would changing the parameters affect the residuals ratio after 10 iterations?

```
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')
```

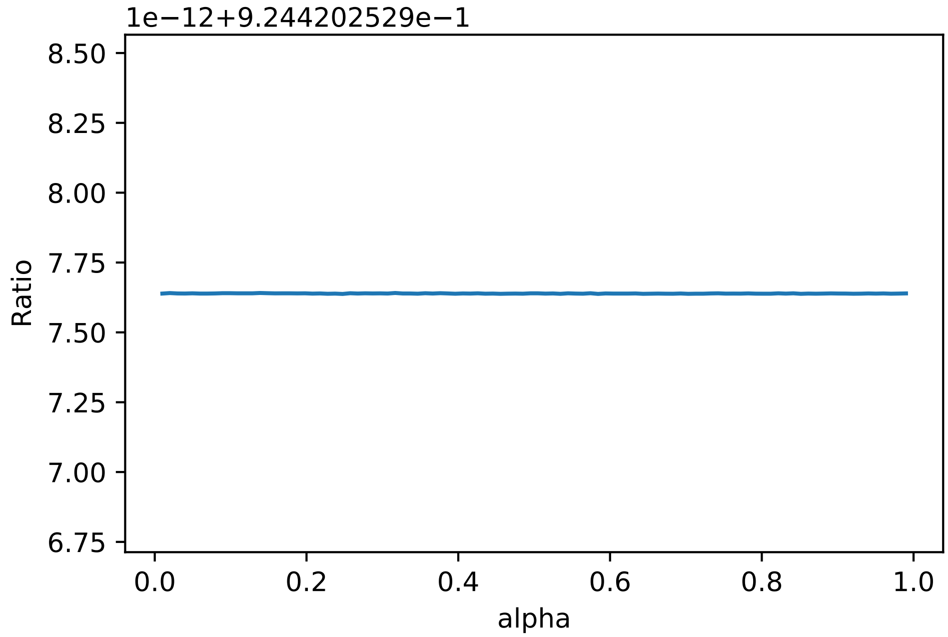
Impact of the choice of time step with the residual ratios.



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])
```