HW3_EBM Solution

October 26, 2022

1 Homework 3 - Solving an EBM

Till Wagner (26 October 2022)

In this assignment you will solve the classic 1-D diffusive Energy Balance Model in 2 ways: First, with a standard spatial grid and the inbuilt odeint and solve_ivp solvers. Then using a simple Forward Euler method (which blows up when dt is too big). Finally, with a staggered spatial grid and Implicit Euler time stepping. We can compare all that to analytical solutions.

[1]:
```
import numpy as np
from scipy.integrate import odeint
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

To start out with we use central difference spatial integration on a standard grid and time stepping with ODEINT. ODEINT uses the LSODA solver which uses linear multistep methods (which utilize information from multiple time steps). The specific integration method for a given problem is chosen depending on whether the ODE appears stiff or not stiff. If *not stiff* it uses the Adams method which is EXPLICIT, if *stiff* it uses the Backward Differentiation Formula (BDF) which is IMPLICIT.

We then compare that to the newer solve_ivp solver, for which we can specify the integration method. Here we consider RK2(3) and BDF.

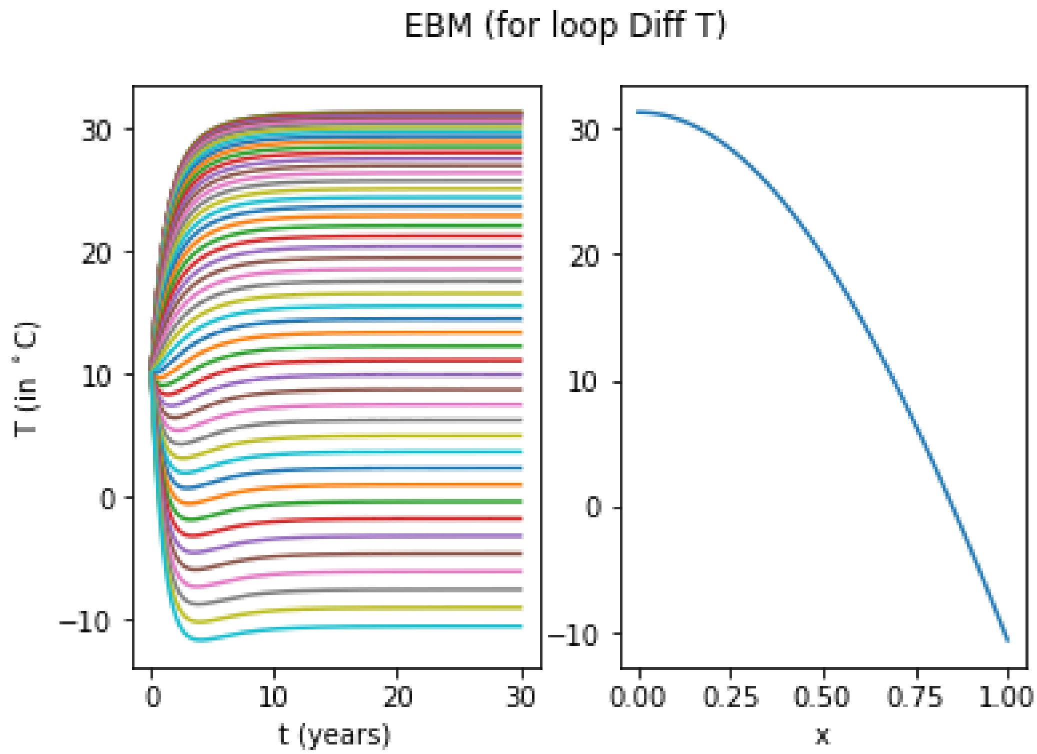
We run the model at a spatial resolution n = 50, and over 30 years. Everything except the Forward Euler solution converges for large time steps.

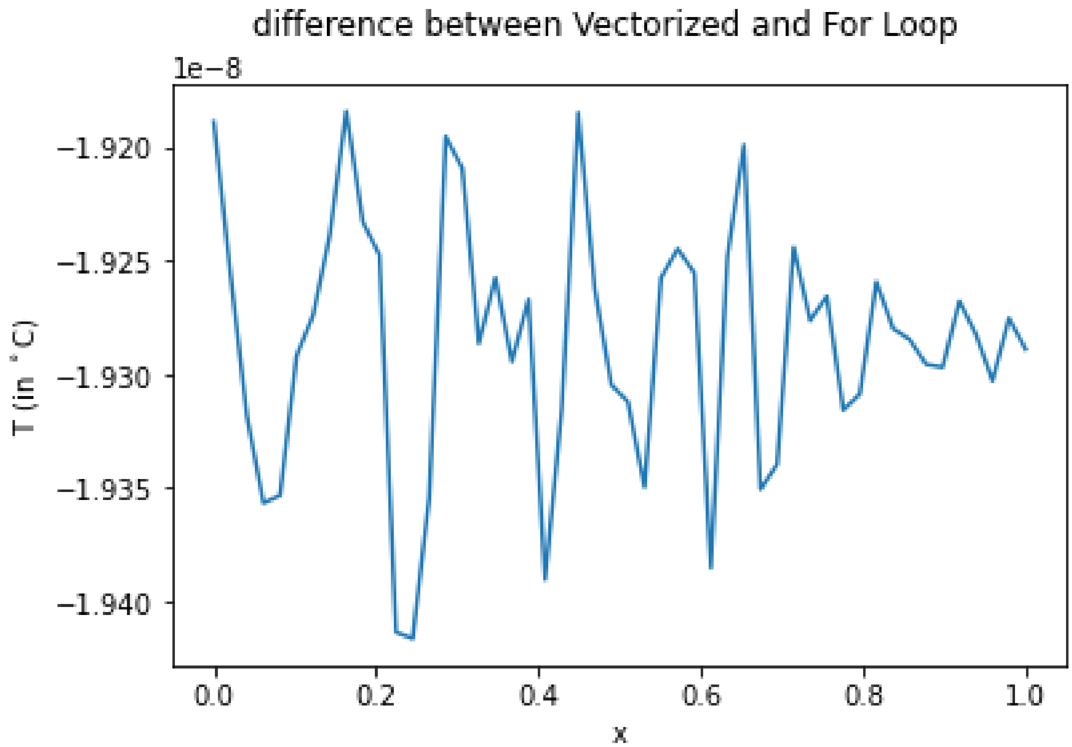
[2]:
```
# Model parameters -------------------------
Q = 340;          #solar constant/4 = 1360/4
A = 203.3;        #const. longwave radiation out
B = 2.09;         #temp. dependent longwave out
a0 = 0.681;       #1. Legendre Poly. albedo cfft
a2 = -0.202;      #2. Legendre Poly. albedo cfft
S2 = -0.477;      #solar forcing value in NCC81
cw = 6.3;         #heat capacity of 50m ocean mixed layer (W/m^2*yr)
D = 0.649;        #heat diffusion
P2 = lambda x: 1/2*(3*x**2-1); #second order Legendre polynomial

# -----------------------------------------------------------------------
```

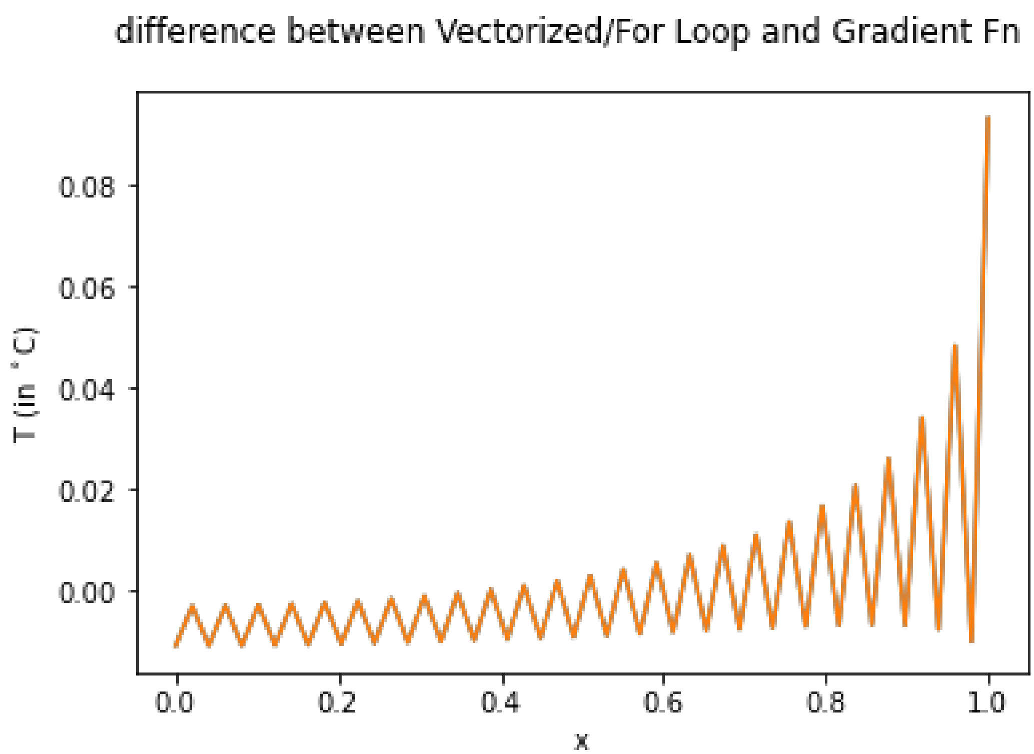
```
n = 50 # grid resolution (number of points between equator and pole)
x = np.linspace(0,1,n)
dx = 1.0/(n-1)

# time stepping
T0 = 10*np.ones(x.shape) # initial condition (constant temp. 10C everywhere)
dur = 30
nt  = dur*12
time = np.linspace(0,dur,nt) # time span in years
dt = time[1]-time[0]

#constant forcing using the Legendre Polynomials as in North et al (1981)
C = Q*(1+S2*P2(x))*(a0 + a2*P2(x)) - A

# ODE with spatial finite differencing------------------------------------
Tdot = np.zeros(x.shape)

## solve c_wdT/dt = D(1-x^2)d^2T/dx^2 - 2xDdT/dx + C - BT
## use central difference.
## Here are 3 ways to do it:
###########################################

# with a basic for loop:
def odefunc(T,t):

    Tdot[0] = D*2*(T[1]-T[0])/dx**2   #boundary condition equator
    Tdot[-1] = -D*2*x[-1]*(T[-1]-T[-2])/dx   #boundary condition pole
    for i in range(1,n-1):
        Tdot[i]=(D/dx**2)*(1-x[i]**2)*(T[i+1]-2*T[i]+T[i-1])-(D*x[i]/
↪dx)*(T[i+1]-T[i-1])

    f  = (Tdot+C-B*T)/cw
    return f

# with a vectorized diff operator
def diffop(T):

    Tkp1 = np.append(T[1:],T[-1])
    Tkm1 = np.append(T[1],T[0:-1])
    Tdot = D*((1-x**2)/(dx**2)*(Tkp1-2*T+Tkm1)-x*(Tkp1-Tkm1)/dx)
    Tdot[-1] = 2*Tdot[-1]

    return Tdot

def odefunc2(T,t):

    f  = (diffop(T)+C-B*T)/cw
```

```
    return f

# using the numpy gradient function
# (doesn't use ghost point which gives some improved accurace)
def odefunc3(T,t):

    dTdx = np.gradient(T,x) #approximate the gradient
    dTdx[0] = 0. #impart B.C. @ equator
    dTdx[-1]=0. #trivial "B.C." @ pole

    Tdot =  D*np.gradient((1-x**2) * dTdx, x) # calculate diffusivity
    return (Tdot+C-B*T)/cw

############################################

sol_for  = odeint(odefunc,T0,time) # solve with for loop Diff T
sol_vec  = odeint(odefunc2,T0,time) # solve with vectorized Diff T
sol_grad = odeint(odefunc3,T0,time) # solve with gradient fn

sol_for_final = sol_for[-1,:] #converged Temp profiles
sol_vec_final = sol_vec[-1,:]
sol_grad_final= sol_grad[-1,:]

# plot everything
fig = plt.figure(1)
fig.suptitle('EBM (for loop Diff T)')
plt.subplot(121)
plt.plot(time,sol_for)
plt.xlabel('t (years)')
plt.ylabel('T (in $^\circ$C)')
plt.subplot(122)
plt.plot(x,sol_for_final)
plt.xlabel('x')
plt.show()

fig = plt.figure(2)
fig.suptitle('difference between Vectorized and For Loop')
plt.plot(x,sol_for_final-sol_vec_final)
plt.xlabel('x')
plt.ylabel('T (in $^\circ$C)')
plt.show()

fig = plt.figure(3)
fig.suptitle('difference between Vectorized/For Loop and Gradient Fn')
plt.plot(x,sol_for_final-sol_grad_final)
plt.plot(x,sol_vec_final-sol_grad_final)
plt.xlabel('x')
```

```
plt.ylabel('T (in $^\circ$C)')
plt.show()
```

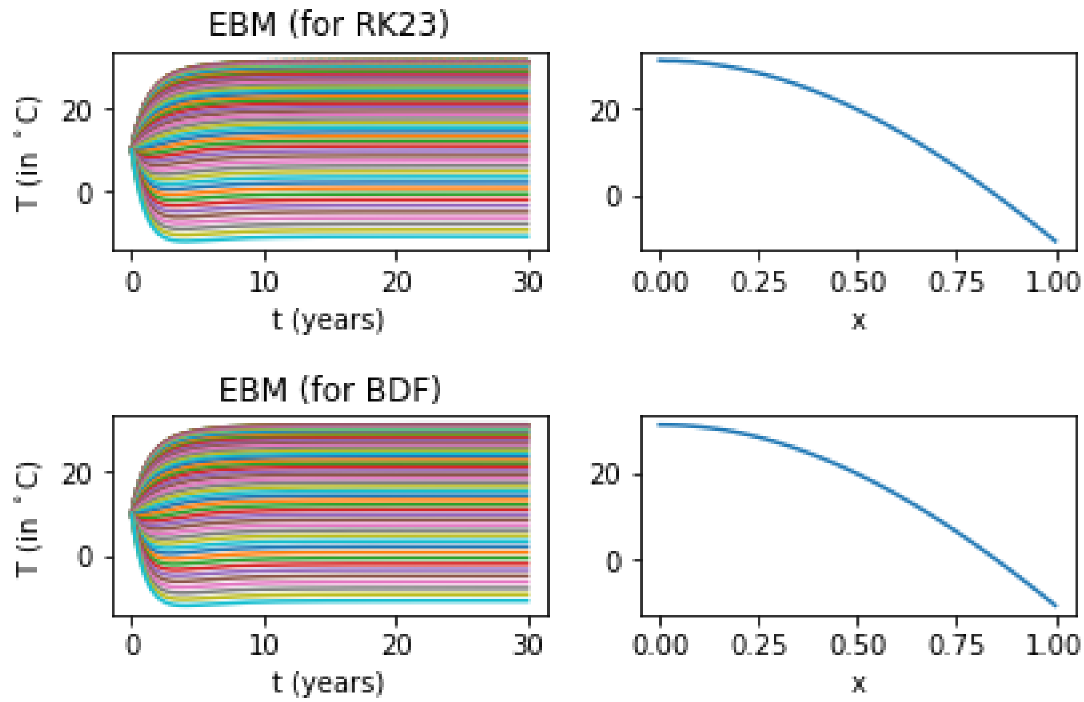



difference between Vectorized and For Loop
1e−8
−1.920
−1.925
−1.930
−1.935
−1.940
T (in ◦C)
0.0
0.2
0.4
0.6
0.8
1.0
x

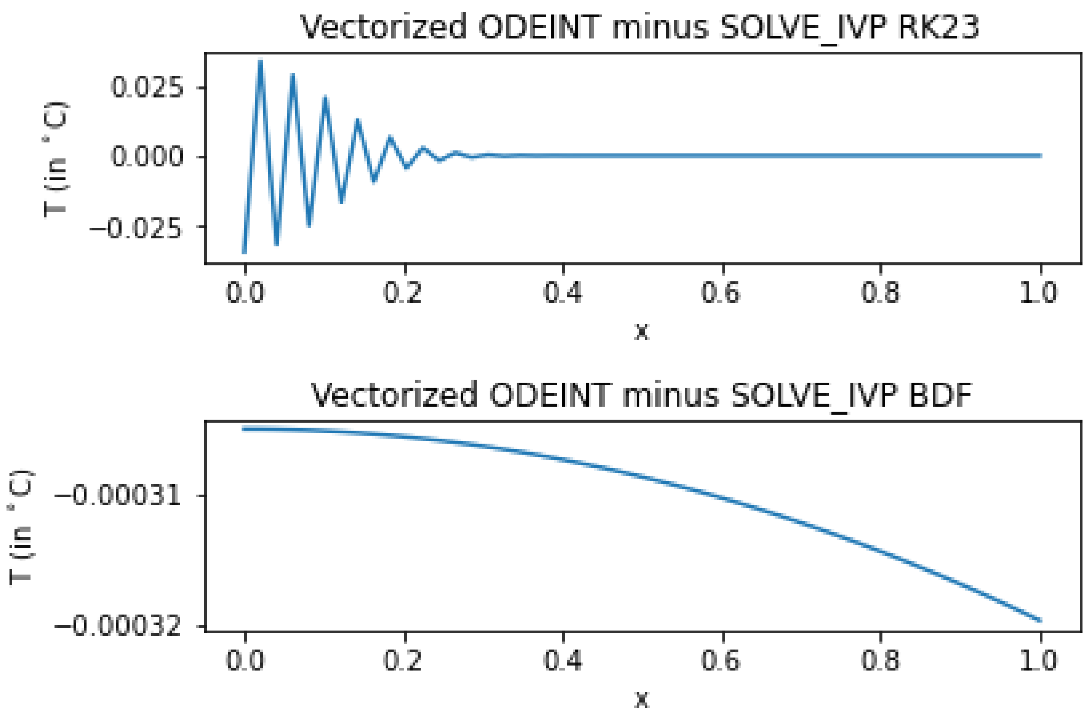

difference between Vectorized/For Loop and Gradient Fn
0.08
0.06
0.04
0.02
0.00
T (in ◦C)
0.0
0.2
0.4
0.6
0.8
1.0
x

Next use solve_ivp so that we can specify the method - and compare EXPLICIT and IMPLICIT solutions

[3]:
```
# with a vectorized diff operator
def ivp(t,T):
    return (diffop(T)+C-B*T)/cw

sol_RK23 = solve_ivp(ivp, [0,dur], T0, method='RK23', t_eval = time) # solve␣
 ↪with Runge-Kutta 2(3)
sol_BDF  = solve_ivp(ivp, [0,dur], T0, method='BDF', t_eval = time) # solve␣
 ↪with Backw. Diff. Form.

sol_RK23 = np.transpose(sol_RK23.y) #need to transpose since IVP has rows/
 ↪columns flipped
sol_BDF = np.transpose(sol_BDF.y)
sol_RK23_final= sol_RK23[-1,:] #get converged Temp profile
sol_BDF_final = sol_BDF[-1,:]

#plot output
fig = plt.figure(1)
plt.subplot(221)
plt.title('EBM (for RK23)')
plt.plot(time,sol_RK23)
plt.xlabel('t (years)')
plt.ylabel('T (in $^\circ$C)')
plt.subplot(222)
plt.plot(x,sol_RK23_final)
plt.xlabel('x')

plt.subplot(223)
plt.title('EBM (for BDF)')
plt.plot(time,sol_BDF)
plt.xlabel('t (years)')
plt.ylabel('T (in $^\circ$C)')
plt.subplot(224)
plt.plot(x,sol_BDF_final)
plt.xlabel('x')

fig.tight_layout(pad=1.5)
plt.show()
```



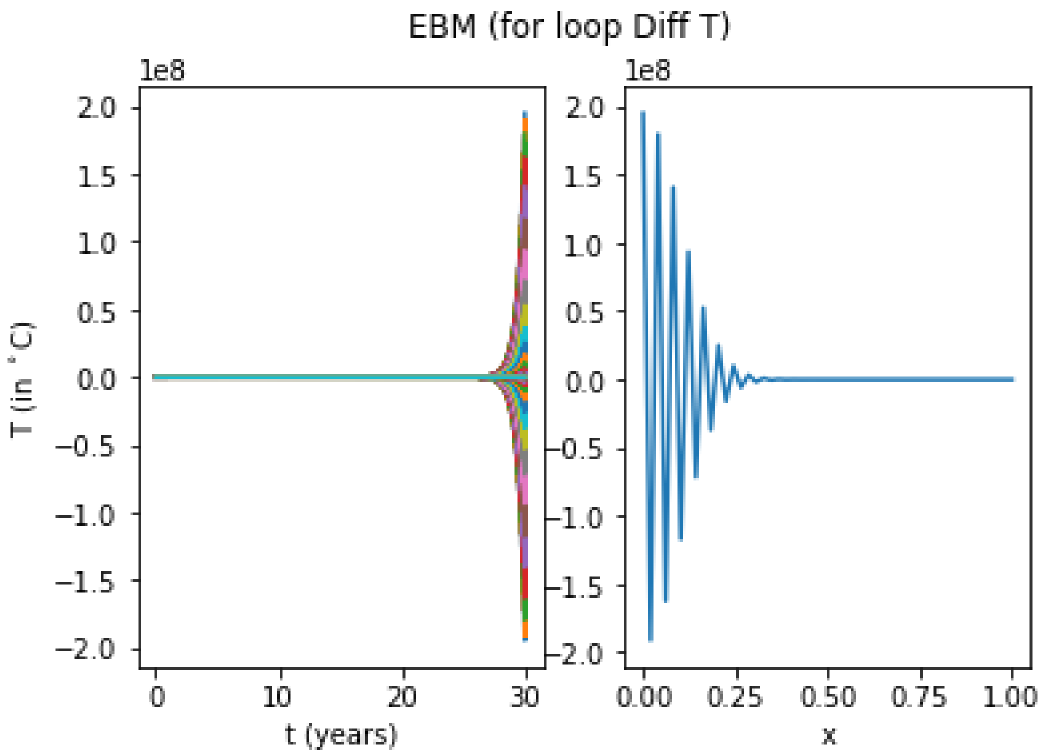
[4]:
```
# now plot difference between ODEINT and SOLVE_IVP output

fig = plt.figure(2)
plt.subplot(211)
plt.title('Vectorized ODEINT minus SOLVE_IVP RK23')
plt.plot(x,sol_vec_final-sol_RK23_final)
plt.xlabel('x')
plt.ylabel('T (in $^\circ$C)')

plt.subplot(212)
plt.title('Vectorized ODEINT minus SOLVE_IVP BDF')
plt.plot(x,sol_vec_final-sol_BDF_final)
plt.xlabel('x')
plt.ylabel('T (in $^\circ$C)')

fig.tight_layout(pad=1.2)
plt.show()
```



What happens if you use simple Forward Euler?

[5]:
```
############################################
# solve it with simple Forward Euler

# time stepping
dur = 30

############################################
# nt  = dur*490 #converges for this
nt = dur*489 #diverges for this
############################################

time = np.linspace(0,dur,nt) # time span in years
dt = time[1]-time[0]

T = T0;
sol_FE = np.zeros([nt,n])
for i in range(0,nt):
    dTdt  = (diffop(T)+C-B*T)/cw
    T = T + dTdt*dt
    sol_FE[i,:]=T

fig = plt.figure(1)
```

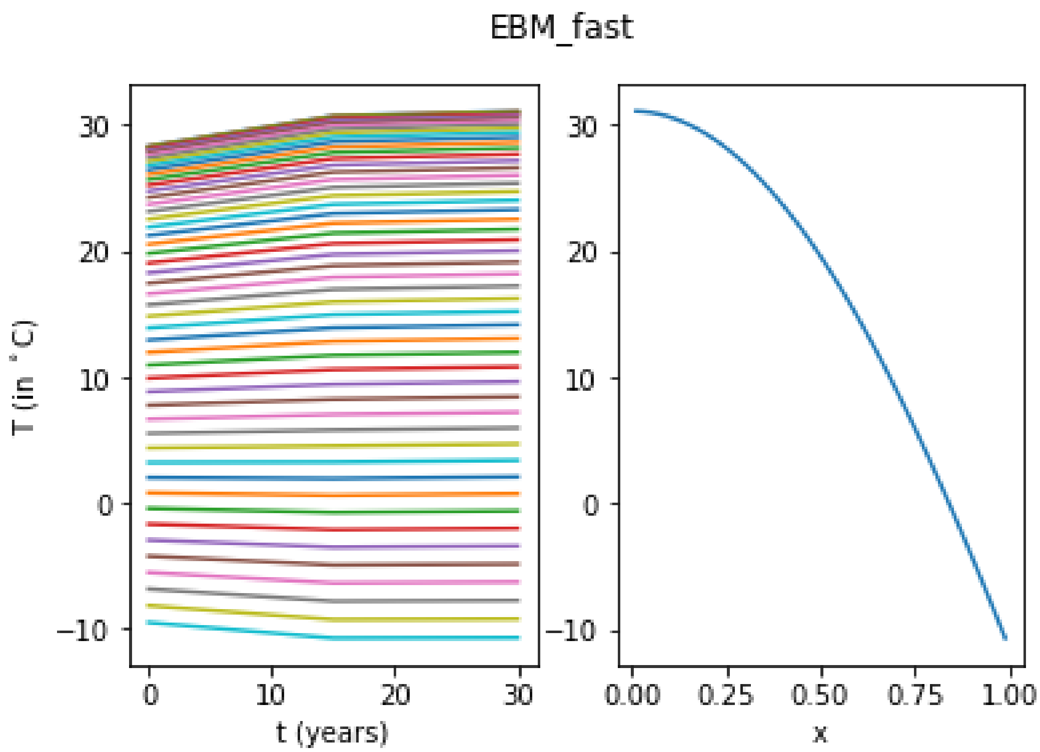
```
fig.suptitle('EBM (for loop Diff T)')
plt.subplot(121)
plt.plot(time,sol_FE)
plt.xlabel('t (years)')
plt.ylabel('T (in $^\circ$C)')
plt.subplot(122)
plt.plot(x,sol_FE[-1,:])
plt.xlabel('x')
plt.show()
```



Implicit Euler Solution with Staggered Grid

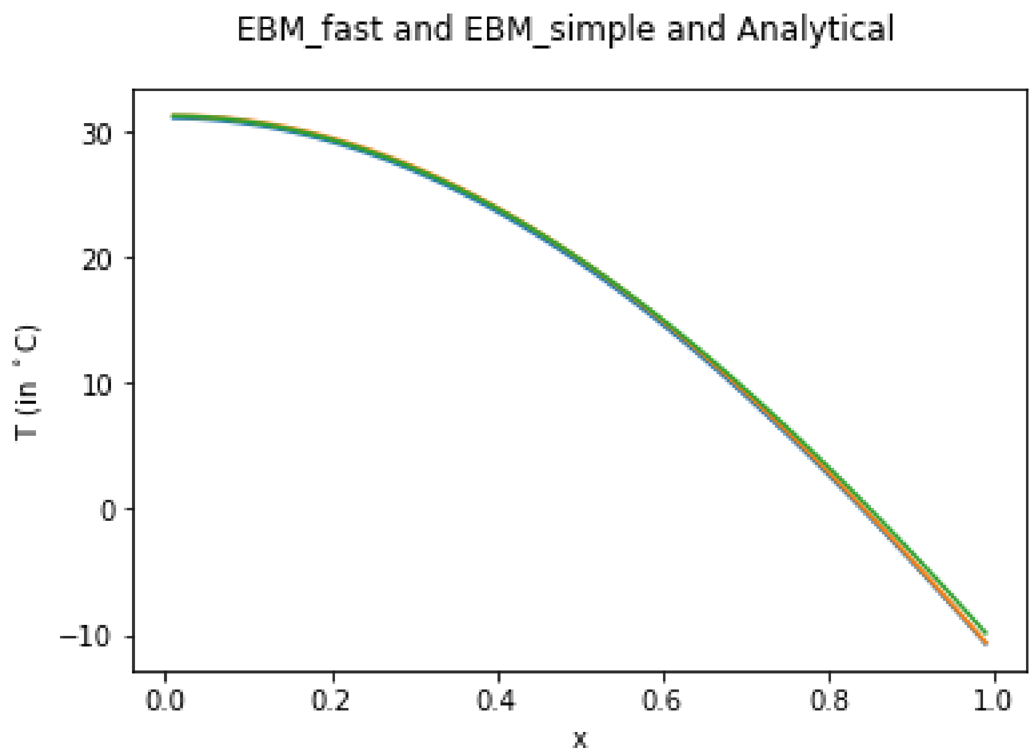
[6]:
```
# ------------------------------------------------------------------------
nt = .1
dur = 30
dt = 1/nt

# Spatial Grid ---------------------------------------------------------
dx = 1.0/n # grid box width
x = np.arange(dx/2,1+dx/2,dx) #native grid
xb = np.arange(dx,1,dx)
# Diffusion Operator (WE15, Appendix A) -----------------------------------
lam = D/dx**2*(1-xb**2)
```

```
L1=np.append(0, -lam)
L2=np.append(-lam, 0)
L3=-L1-L2
DiffOp = - np.diag(L3) - np.diag(L2[:n-1],1) - np.diag(L1[1:n],-1);

Tfast = 10*np.ones(x.shape) # initial condition (constant temp. 10C everywhere)
sol_Imp = np.zeros([int(dur*nt),n])
tImp = np.linspace(0,dur,int(dur*nt))

I = np.identity(n)
invMat = np.linalg.inv(I+dt/cw*(B*I-DiffOp))

# integration over time using implicit difference and
# over x using central difference (through diffop)

# Governing equation [cf. WE15, eq. (2)]:
# T(n+1) = T(n) + dt*(dT(n+1)/dt), with c_w*dT/dt=(C-B*T+diffop*T)
# -> T(n+1) = T(n) + dt/cw*[C-B*T(n+1)+diff_op*T(n+1)]
# -> T(n+1) = inv[1+dt/cw*(1+B-diff_op)]*(T(n)+dt/cw*C)

for i in range(0,len(tImp)):
    T = Tfast+dt/cw*C
    Tfast = np.dot(invMat,T)
    sol_Imp[i,:]=Tfast

sol_Imp_final = Tfast

fig = plt.figure(1)
fig.suptitle('EBM_fast')
plt.subplot(121)
plt.plot(tImp,sol_Imp)
plt.xlabel('t (years)')
plt.ylabel('T (in $^\circ$C)')
plt.subplot(122)
plt.plot(x,sol_Imp_final)
plt.xlabel('x')
plt.show()
```



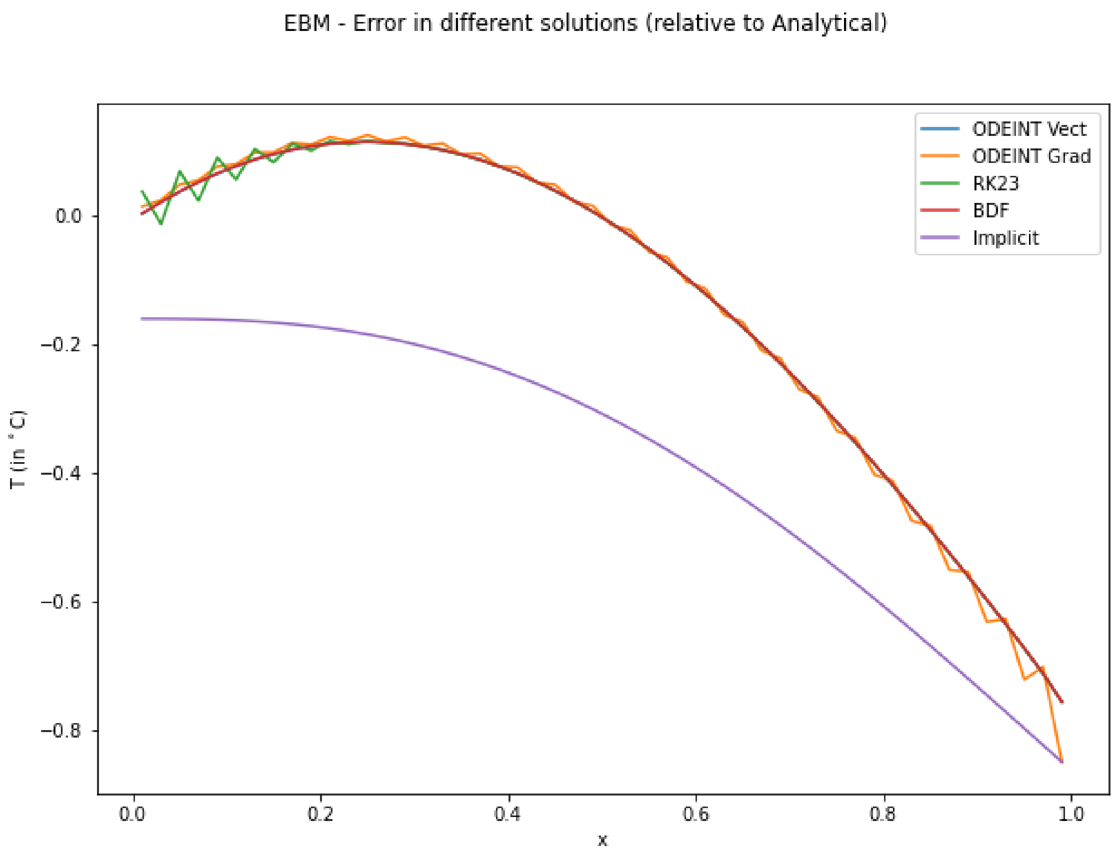
Compare to Analytical Solution of EBM (following North 1975)

[7]:
```
H0 = a0 + a2*S2/5;
H2 = a0*S2 + a2 + a2*S2*2/7;
H4 = 18/35*a2*S2;
T0 = (Q*H0-A)/B;
T2 = Q*H2/(6*D+B);
T4 = Q*H4/(20*D+B);
P2 = lambda X: 1/2*(3*X**2-1);
P4 = lambda X: 1/8*(3-30*X**2+35*X**4);
sol_A = lambda X: T0 + T2*P2(X) + T4*P4(X);

fig = plt.figure(1)
fig.suptitle('EBM_fast and EBM_simple and Analytical')
plt.plot(x,sol_Imp_final)
plt.plot(x,sol_vec_final)
plt.plot(x,sol_A(x))
plt.ylabel('T (in $^\circ$C)')
plt.xlabel('x')
plt.show()
```



Now compare the final error in the different numerical methods relative to the analytical solution

[8]:
```
solA = sol_A(x)

fig = plt.figure(1)
fig.set_size_inches(10,7)
fig.suptitle('EBM - Error in different solutions (relative to Analytical)')
plt.plot(x,sol_vec_final-solA,label ='ODEINT Vect')
plt.plot(x,sol_grad_final-solA,label='ODEINT Grad')
plt.plot(x,sol_RK23_final-solA,label='RK23')
plt.plot(x,sol_BDF_final-solA,label ='BDF')
plt.plot(x,sol_Imp_final-solA,label ='Implicit')
plt.ylabel('T (in $^\circ$C)')
plt.xlabel('x')
plt.legend(loc='upper right')
plt.show()
```


EBM - Error in different solutions (relative to Analytical)
ODEINT Vect
ODEINT Grad
RK23
BDF
Implicit
0.0
−0.2
−0.4
−0.6
−0.8
T (in °C)
0.0
0.2
0.4
0.6
0.8
1.0
x