upwind

January 27, 2022

[1]:

```python
import math
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt


def velfct(uval, x, t):
    # *************** TODO ***************
    # implement the correct velocity,
    # below are a dummy values
    # the functions math.log(5.0)
    # and pow(3,2) might come handy
    # ************************************
    a = 6 * np.log(5)
    v = (a * x) / ((t + 2) ** 2)
    return v


def icfct(x):
    # *************** TODO ***************
    # implement the correct initial data,
    # below are dummy values
    # the function abs() might come handy
    # ************************************
    if ((x - 1 / 8) > 1e-22) and ((7 / 8 - x) > 1e-22):
        uic = (-8 / 3) * np.abs(x - 1 / 2) + 1
    else:
        uic = 0
    return uic


def visfct(ax, xx, tt, U):
    # this function serves to visualise the solution
    XX, TT = np.meshgrid(xx, tt)
    ax.plot_surface(XX, TT, U, cmap="viridis", edgecolor="none")
    ax.set_xlabel("space")
    ax.set_ylabel("time")
```

```
    plt.show()
    return


def CL1D_upwind(N, maxvel, vis_flag):
    # first order finite difference upwinding scheme
    # for transport problems of the form
    #   d_t u(x,t) + velfct(u(x,t),x,t) d_x u(x,t) = 0 on (0,3) \times (0,1),
    # with initial condition
    #   u(0,x) = icfct(x)
    # and constant values at influx boundaries
    #
    # input data:
    #   N      : the spatial domain is discretised with
    #            6*N+1 equidistributed mesh points
    #   maxvel : maximal value of the velocity,
    #            needed for numerical stability
    # vis_flag : # switch on(1) / off(0) for visualisation
    #
    # B Stinner 2020

    # visualisation settings
    if vis_flag == 1:
        fig = plt.figure()
        ax = plt.axes(projection="3d")

    # get grid and initialise
    n = 6 * N + 1  # number of spatial mesh points
    h = 3.0 / (n - 1)  # spacial step size
    delta = 0.96 * h / maxvel  # time step size
    lam = delta / h
    xx = np.linspace(0, 3, n)  # spatial mesh
    no_steps = int(1 // delta)  # number of time steps
    tt = delta * np.arange(no_steps + 1)  # time mesh
    U = np.zeros((no_steps + 1, n))  # array to store the discrete solution,
    # one row for each time point
    for i in range(0, n):  # initialise first column with initial data,
        U[0][i] = icfct(xx[i])
    b = np.zeros(n)  # help vector for storing velocity values

    # initial visualisation if uncommented, could be useful for testing initial
↪data
    # if vis_flag == 1:
    #     visfct (ax,xx,tt,U)

    # main time loop
    # the discrete solution of the previous step is stored in U[m],
```

```python
    # we compute the discrete solution at time tt[m+1]
    # and store it in U[m+1]

    m = 0  # time step counter
    while m < no_steps:
        time = tt[m + 1]

        # compute the velocities in the mesh points
        for j in range(0, n):
            b[j] = velfct(U[m][j], xx[j], tt[m])
        # in general, vector versions are faster
        # but the velocity function has to be coded as appropriate

        # upwind scheme
        # left boundary
        j = 0
        if b[j] > 0:  # influx, keep value
            U[m + 1][j] = U[m][j]
        else:  # otherwise take forward difference for d_x u
            U[m + 1][j] = U[m][j] - lam * b[j] * (U[m][j + 1] - U[m][j])
        # interior mesh points
        for j in range(1, n - 1):
            if b[j] > 0:  # take backward difference for d_x u
                U[m + 1][j] = U[m][j] - lam * b[j] * (U[m][j] - U[m][j - 1])
            else:  # otherwise take forward difference for d_x u
                U[m + 1][j] = U[m][j] - lam * b[j] * (U[m][j + 1] - U[m][j])
        # right boundary
        j = n - 1
        if b[j] < 0:  # influx, keep value
            U[m + 1][j] = U[m][j]
        else:  # otherwise take backward difference for d_x u
            U[m + 1][j] = U[m][j] - lam * b[j] * (U[m][j] - U[m][j - 1])

        # prepare for next time step
        # print('After step ')
        # print(m+1)
        m = m + 1
    # end while loop

    if vis_flag == 1:
        visfct(ax, xx, tt, U)
    return U


# end function CL1D_upwind

vis_flag = 0
```

```python
maxvel = 6
# **************** TODO ****************
# after implementing the relative error computation (below)
# increase N from 60 until that relative error is small enough;
# it might be sensible to set the vis_flag to zero
# **************************************
N = 135
solU = CL1D_upwind(N, maxvel, vis_flag)
unum = solU[-1][5 * N]
print(unum)
print(f"Relative Error: {1-unum}")
# **************** TODO ****************
# store the exact solution u(5/2,1) in uexact
# and compute the relative error
# **************************************
# uexact = ...
# relerr = np.abs(.../...)
# print(relerr)
```

```
0.9302579918780685
Relative Error: 0.06974200812193154
```

[ ]:

```python
d
```

[ ]:

```python

```