Univerza *v Ljubljani*
Fakulteta *za strojništvo*

Boštjan Mavrič, Mihael Boštjan Končar,
Matjaž Zadnik, Božidar Šarler

Zbirka numeričnih implementacij osnovnih rešitvenih postopkov računalniške dinamike tekočin v jeziku Python

A Collection of Numerical Implementations of Basic Solution Procedures of Computational Fluid Dynamics in Python

Računalniška dinamika tekočin
Computational Fluid Dynamics

Magistrski študijski program
STROJNIŠTVO - Razvojno raziskovalni program

Master's study programme
MECHANICAL ENGINEERING - Research and Development program

Ljubljana, 13. 5. 2021

Predgovor / Foreword

Predmet Računalniška dinamika tekočin novega magistrskega študijskega programa Fakultete za strojništvo Univerze v Ljubljani, ki se začenja izvajati v študijskem letu 2021/2022, je zasnovan za doseganje dveh bistvenih ciljev:

1. Razumevanju rešitvenih postopkov za numerično reševanja enačb, ki popisujejo različne tekočine pri različnih pogojih.
2. Praktični uporabi programov ANSYS Fluent in OpenFOAM za reševanje množice tovrstnih inženirskih problemov.

Ta skripta predstavlja zbirko numeričnih implementacij osnovnih rešitvenih postopkov, povezanih z reševanjem vodilnih enačb dinamike tekočin. V skripti prikažemo, kako so obravnavani rešitveni postopki numerično implementirani. Namenjena je v pomoč domačim in mednarodnim študentom, zato je napisana v slovenščini in angleščini.

Koda v pričujoči knjižici ne more v popolnosti nadomestiti izvedljivih interaktivnih skript, ki so objavljene v prostem dostopu na portalu Github. Z njimi lahko študentje interaktivno preiskujejo fazni prostor rešitev ter raziskujejo povezavo med prostimi parametri metode in numerično rešitvijo. Dostop do njih je opisan na začetku vsakega poglavja. Vsako poglavje predstavlja blok dveh učnih ur pri katerih se študentom razloži podrobnosti implementacije zbranih metod, demonstrira njihovo delovanje ter preveri rezultate o njihovi stabilnosti in natančnosti, ki so obravnavani med predavanji.

The Computational Fluid Dynamics course of the new masters study program of the Faculty of Mechanical Engineering, University of Ljubljana, which will be implemented in the academic year 2021/2022, is designed to achieve two essential objectives:

1. Understanding solution procedures for numerically solving equations that describe different fluids under different conditions.
2. Practical use of ANSYS Fluent and OpenFOAM codes to solve a multitude of related engineering problems.

This script represents a collection of the numerical implementations of basic solution procedures associated with the solution of governing equations of fluid dynamics. We show in the script how the discussed solution procedures are numerically implemented.

It is intended to be used by domestic and international students, so it is written in Slovene and English.

The code in the present booklet cannot fully replace the executable interactive scripts, which are published in open-access on Github portal. By using them, the students can interactively explore the phase space of the solutions and investigate the relation between the free parameters of the numerical method and the numerical solution. Access to the scripts is explained at the beginning of each chapter. Each chapter corresponds to a block of two contact hours during which the details

of the implementation of numerical methods are explained to the students, the functioning of the implementations is demonstrated and the results on accuracy and stability, which were discussed during the lectures, are checked.

Ljubljana, 13. 5. 2021 prof. dr. Božidar Šarler

Kazalo

Poglavje 1

Uvod / Introduction

1.1 Namen / Purpose

Ta dokument je spremljajoči dokument skript, ki demonstrirajo metodo končnih volumnov v jeziku Python in so na voljo v več GitHub repozitorijih. Skripte so napisane za uporabo v okolju Jupyter (`https://jupyter.org`), ki je namenjeno spodbujanju razvoja interaktivnih rešitev na področju znanstvenega računanja in statistike. Ta dokument in skripte, ki jih spremlja, so napisane v dveh jezikih. Črno obarvan tekst je v slovenščini, sivo obarvan pa v angleščini.

This document is accompanying document to the scripts that demonstrate finite volume method in Python and are available through several GitHub repositories. The scripts are written for Jupyter environment (`https://jupyter.org`), which is developed to facilitate the development of interactive solutions in the field of scientific computing and statistics. This document and the scripts it is accompanying are written in two languages. Black text is written in Slovene and the gray text is written in English.

1.2 Namestitev okolja Jupyter /Jupyter environment installation

Do okolja Jupyter je mogoče dostopati na več načinov. Za krajšo, enkratno uporabo, ko lahko delamo z omejeno strojno opremo, je najprimernejši dostop do okolja, ki ga za demonstracijske namene vzdržuje projekt Jupyter. Do tega okolja lahko dostopamo na povezavi `https://jupyter.org/try`. Za zahtevnejše in dolgotrajneše delo je okolje Jupyter možno namestiti na lokalni računalnik prek paketnih upravljalcev `pip` ali `conda`. Natančna navodila za namestitev z obema upravljalnikoma najdemo na spletnih straneh projekta Jupyter `https://jupyter.org/install`.

Jupyter environment can be accessed in several ways. For simple, one-time use, when we can work with limited hardware, it is the most convenient to access the environment provided by Jupyter project for demonstration purposes. It can be accessed through the link `https://jupyter.org/try`. For more demanding and longer lasting work, it is possible to install the Jupyter environment to the local computer through package managers `pip` or `conda`. Detailed installation instructions with both managers are availibal on the project Jupyter web pages `https://jupyter.org/install`.

Za upravljalec `conda` je v repozitorijih priložena nastavitvena datoteka `FVM_demos_env.yml`, ki omogočajo ponovljivo nastavitev okolja v katerem se izvajajo skripte. Okolje uvozimo s klicem

`conda env create -f FVM_demos_env.yml`. Pravilno namestitev preverimo s klicem `conda env list`, ki ob pravilni namestitvi med ostalimi okolji izpiše tudi okolje `FVM_demos`. Aktiviramo ga s klicem `conda activate FVM_demos`. Okolje Jupyter nato poženemo s klicem `jupyter notebook`.

Attached in the repositories is the configuration file `FVM_demos_env.yml` for `conda` manager, which enables repeatable configuration of the environment in which the scripts can be run. The environment can be imported by calling `conda env create -f FVM_demos_env.yml`. Correctness of the configuration can be checked by calling `conda env list`, which will list environment `FVM_demos` as available if configuration was successful. The environment is activated by calling `conda activate FVM_demos`. The Jupyter environment can then be started by calling `jupyter notebook`.

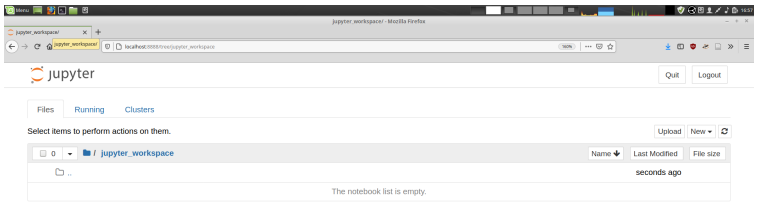
1.3 Kako uporabljati okolje Jupyter / How to use the Jupyter environment

Pred pričetkom uporabe je potrebno skripte najprej prenesti iz repozitorija. Naslov repozitorija s skriptami je podan v uvodu v vsako poglavje. To je mogoče prek uporabe brskalnika ali pa neposredno s programom `git`.

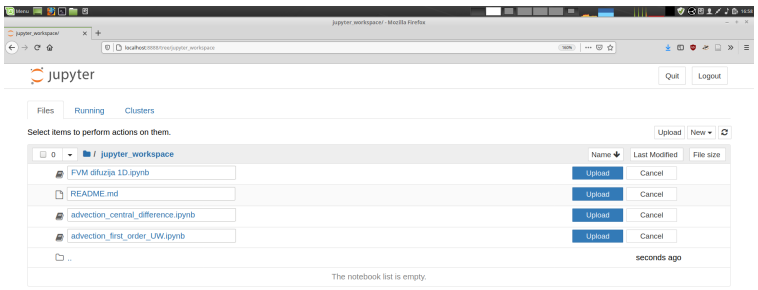
The scripts need to be downloaded from the repositories before they can be run. The address of the repository with scripts is given in the introduction of each chapter. The scripts can be downloaded either through web browser or directly using the program `git`.

Da lahko skripte izvajamo jih moramo najprej naložiti v okolje Jupyter. To naredimo tako, da na v začetnem oknu na sliki 1.1 izberemo možnost “Upload”, ter datoteke izberemo v brskalniku datotek, ki se odpre. Izbrane datoteke se bodo pojavile na seznamu v začetnem oknu na sliki 1.2, kjer s klikom na gumb “Upload” sprožimo nalaganje datoteke v okolje Jupyter. Ko je datoteka naložena, jo lahko odpremo s klikom na njeno ime v seznamu.

To run the scripts we must first load them in the Jupyter. We do this by selecting the option “Upload” in the start screen shown in figure 1.1 and choosing the files to be run in the file picker window that opens. The selected files will appear in a list in the start screen as shown in figure 1.2, where we upload them by clicking “Upload”. When the file is loaded we can open it by clicking on its name in the list.



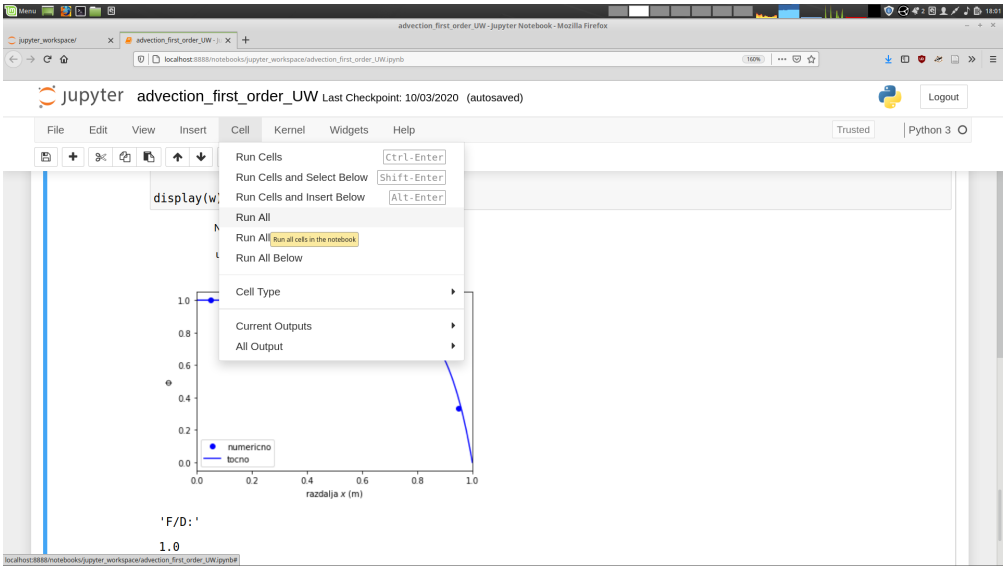
Slika 1.1: Začetno okno okolja Jupyter. / Start screen of Jupyter environment.



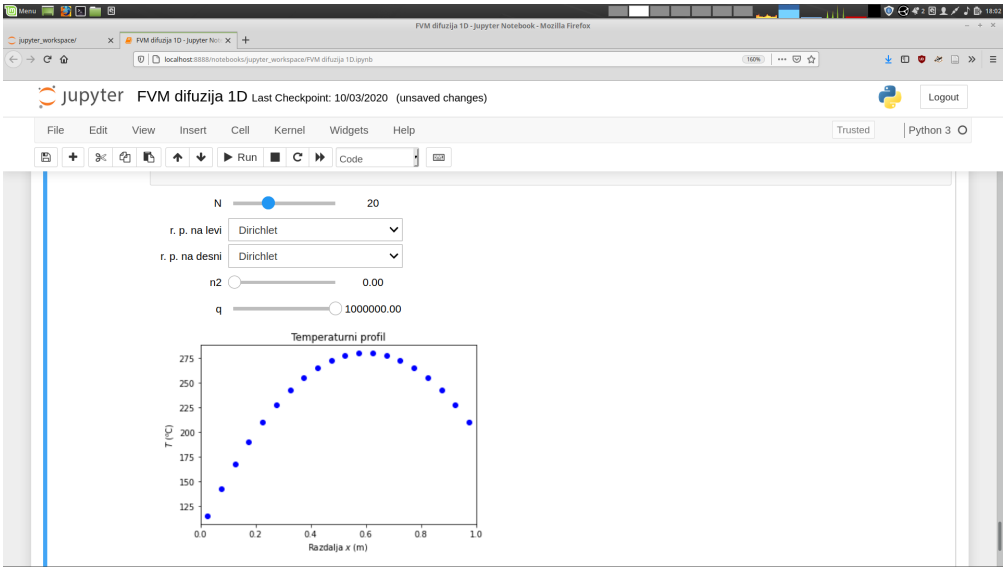
Slika 1.2: Vmesnik za nalaganje datotek v okolju Jupyter./ Interface for uploading the files to the Jupyter environment.

Vsebina datoteke se bo prikazala v novem zavihku brskalnika. Prikazano okno je sestavljeno iz glave z več orodnimi vrsticami ter telesa, kjer je prikazana vsebina skripte, razdeljena v celice, ki jih je mogoče izvajati posamično. Za namene demonstracije je najbolje v orodni vrstici izbrati možnost “Cell” - ”Run All” (slika 1.3a), kar sproži zaporedno izvajanje vseh celic v dokumentu. S pomikom na dno dokumenta se za zadnjo celico razkrijejo rezultati aplikacije (slika 1.3b).

File contents will be shown in a new tab in the web browser. The displayed interface consists of a header with several toolbars and the body, where the script content is displayed, sectioned int cells, that can be run individually. To demonstrate the implemented methods it is best to select from the toolbar “Cell” - ”Run All” (shown in 1.3a), which triggers sequential execution of all cells in the script. By moving to the bottom of the document, the results are shown after the last cell in the document (figure 1.3b).



(a) Izbirni meni za začetek izvajanja vseh celic. / Menu selection to run all cells.



(b) Primer rezultatov simulacije z drsniki za nadzor nad parametri simulacije. / An example of the results with sliders used to control the simulation output.

Slika 1.3: Izvajanje skript v okolju Jupyter. / Executing a script in Jupyter environment.

Poglavje 2

Ustaljeni problemi / Stationary problems

Skripte, ki implementirajo metode predstavljene v tem poglavju je možno prenesti iz repozitorija na https://github.com/bmavric/FVM_codes.

The scripts implementing the methods presented in this section can be downloaded from the repository at https://github.com/bmavric/FVM_codes.

Implementacije so v naslednjih datotekah / The implementations are in the following files:

- `Aproksimacija_interpolacija.ipynb` - Aproksimacija in interpolacija / Approximation and interpolation
- `advection_central_difference.ipynb` - Centralna diferenčna shema / Central difference scheme
- `advection_first_order_UW.ipynb` - Priveterna shema prvega reda / First order upwinding scheme
- `advection_hybrid_uw.ipynb` - Hibridna shema / Hybrid scheme

2.1 Implementacija interpolacijskih in aproksimacijskih metod / Implementation of interpolation and approximation methods

[ ]:
```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

np.warnings.filterwarnings('error', category=np.VisibleDeprecationWarning)
```

2.1.1 Motivacija /Motivation

V devetih točkah na premici imamo znane podatke o vrednosti veličine f_n, želimo pridobiti vrednosti na območju med točkami. V prvem primeru nas najbolj zanimajo vrednosti f med točkami, gradienti niso pomembni. Med tem ko nas v drugem primeru nas najbolj zanimajo gradienti spremenljivke f najprej po celotni domeni, nato posebej v okolici točke $\overline{\mathbf{p}}$, na koncu pa želimo karseda najboljši približek.

Točke z znanimi vrednostmi $\mathbf{p}_n = (1), (2), (3), (4), (5), (6), (7), (8), (9)$.

Poznane vrednosti veličine $f_n = 1, 2, 0.5, 5, 1, 0, -1, 3, -4$.

Data f_n are given on 9 points in a line. We wish to obtain the function values in the area between the points. First, we are interested in the methods that reproduce only the function values and then the methods that can also determine the gradients of the function throughout the domain and finally near the point $\overline{\mathbf{p}}$, trying to obtain the best possible approximation.

The values are given at points $\mathbf{p}_n = (1), (2), (3), (4), (5), (6), (7), (8), (9)$.

The known values are $f_n = 1, 2, 0.5, 5, 1, 0, -1, 3, -4$.

[ ]:
```
pn = np.asarray([1,2,3,4,5,6,7,8,9])
fn = np.asarray([[1,2,0.5,5,1,0,-1,3,-4]])
fn = np.array([np.sin(pn)])
```

2.1.2 Shepardova interpolacija / Shepard interpolation

Kadar nas zanimajo vrednosti spremenljivke f med točkami, ne pa tudi gradienti, uporabimo Shepardovo interpolacijo. Najpreprostejša oblika kolokacije je Shepardova shema, ki ne zahteva reševanja sistema enačb. V tem primeru ima aproksimirana funkcija $f(\mathbf{p})$ tako obliko:

$$f(\mathbf{p}) = \frac{\sum_{n=1}^{N} \psi_n(\mathbf{p}) f(\mathbf{p}_n)}{\sum_{n=1}^{N} \psi_n(\mathbf{p})},$$

kjer so ψ_n oblikovne funkcije oblike $\psi_n = |\mathbf{p} - \mathbf{p}_n|^{-\mu}$. Število teh funkcij mora biti enako številu točk $\mathbf{p}_n$. Za koeficient μ običajno vzamemo 2.

When we are only interested in values of variable f between points and gradients are not important, we can use Shepard interpolation. The simplest form of collocation is the Shepard scheme, which does not require solving equations. In this case the approximated function $f(\mathbf{p})$ has the following form:

$$f(\mathbf{p}) = \frac{\sum_{n=1}^{N} \psi_n(\mathbf{p}) f(\mathbf{p}_n)}{\sum_{n=1}^{N} \psi_n(\mathbf{p})},$$

where ψ_n trial functions are defined as $\psi_n = |\mathbf{p} - \mathbf{p}_n|^{-\mu}$. The number of this functions is equal to the number of points $\mathbf{p}_n$. Coefficient μ is 2 for most cases.

Implementacija Shepardove interpolacije v kodi / Implementation of Shepard interpolation in code

[ ]:
```
def Shepard(p,pn,fn):
    """Funkcija interpolira vrednost v točki p s Shepardovo interpolacijo na podlagi vrednosti fn v točkah pn /
    Function interpolates value in point p with Shepard interpolation using data fn from points pn."""
    stevec = 0 #inicializacija člena v števcu / init. of in numerator
    imenovalec = 0 #inicializacija člena v imenovalcu / init. of in denominator
    for i in range(len(pn)): #korakanje po točkah pn / marching through points pn
        mu = 2
        psi_n = ((p-pn[i])**2)**0.5 #definicija vrednosti oblikovne funkcije / definition of trial function value
        if psi_n == 0: #v izogib deljenju z nič postavimo pogoj / to avoid dividing by zero, we itroduce condition
            return fn[0,i] #izračun interpolirane vrednosti, če je p=pn / calc. of interpolated value, if p=pn
        else:
            stevec += psi_n**(-mu) * fn[0,i] #izračun števca / calc. of numerator
            imenovalec += psi_n**(-mu) #izračun imenovalca / calc. of denumerator
    return stevec / imenovalec #izračun interpolirane vrednosti / calc. of interpolated value
```

[ ]:
```
#izris rezultatov / plotting the results
x_points = np.linspace(0.9,9,1000)
y_Shepard = np.array([])
for i in range(len(x_points)):
    y_Shepard = np.append(y_Shepard,Shepard(x_points[i],pn, fn))

g1, = plt.plot(x_points,y_Shepard,"-g",label = 'Shepardova inteprolacija \nShepard interpolation')
g2, = plt.plot(pn,fn[0], "o",color='purple', label = "Referenčne točke p_n(x) / \nReference points p_n(x)")
title = plt.title('Shepardova interpolacija \nShepard interpolation')
legend = plt.legend(handles = [g2], bbox_to_anchor=(1.05, 1), loc='upper left')
```

2.1.3 Kolokacija s polinomskimi baznimi funkcijami /Collocation with polynomial base functions

Naslednja kolokacijska shema je kolokacija s polinomskimi funkcijami. Tu je aproksimirana funkcija oblike $f(\mathbf{p}) = \psi(\mathbf{p}) \cdot \mathbf{c}$, kjer so ψ_n funkcije polinomskega baznega vektorja.

Če je število funkcij ψ_n enako številu točk $\mathbf{p}_n$, potem lahko koeficiente $\mathbf{c}$ določimo z enačbo:

$\sum_{n=1}^{N} \psi_n(\mathbf{p}_n) \cdot c_n = \mathbf{f}_n$ Next collocation scheme is collocation with polynomial functions. The approximated function is defined as $f(\mathbf{p}) = \psi(\mathbf{p}) \cdot \mathbf{c}$, where ψ_n are functions from the polynomial base vector.

If the number of this functions matches the number of points $\mathbf{p}_n$, then the vector of coefficients $\mathbf{c}$ can be determined through equation:

$\sum_{n=1}^{N} \psi_n(\mathbf{p}_n) \cdot c_n = \mathbf{f}_n$

Implementacija interpolacije s polinomskimi baznimi funkcijami v kodi / Implementation of interpolation with polynomial basis functions in code

[ ]:
```
def interpolation(pn, fn):
    """Funkcija poda interpolirano funkcijo med točkami pn s kolokacijo s␣
 ↪polinomskimi baznimi funkcijami na podlagi vrednosti fi v točkah pn /
    Function gives interpolated function between points pn with collocation␣
 ↪with polynomial base functions using data fn from points pn."""
    apr_fun = lambda x: np.asarray([1,x,x**2,x**3,x**4,x**5,x**6,x**7,x**8])
    fn = fn[0] #poznana vrednost v točki z indeksom 0 / known value in pont␣
 ↪index 0
    i = 1
    A = apr_fun(pn[0]) #incializacija matrike koef. s prvo vrstico / init. of␣
 ↪matrix of coeff. with first row
    b = fn #incializacija vektorja konstant s prvo vrstico / init. of vector of␣
 ↪constants. with first row
    while i < len(pn): #gradnja sistema enačb / building system of equations
        A = np.vstack((A,apr_fun(pn[i])))
        i += 1
    c = np.linalg.solve(A,b) #reševanje sistema - dobimo vektor koeficientov c /
 ↪ solving of system - we get vector of coeff. c
    #definicija interpolirane funkcije / definition of interoplated function
    def fun(o):
        return c[0] + c[1]*o + \
        c[2]*o**2 + c[3]*o**3 + \
            c[4]*o**4 + c[5]*o**5 + \
               c[6]*o**6 + c[7]*o**7 + \
                    c[8]*o**8
    return fun
```

[ ]:
```
#izris rezultatov / plotting the results
x_points = np.linspace(0.9,9,1000)
f1 = interpolation(pn, fn)
fig = plt.figure(5)
g1, = plt.plot(x_points,f1(x_points),"-g",label = 'Interpolacija \n9 oblikovnih␣
 ↪funkicij')
g2, = plt.plot(pn,fn[0], "o",color='purple', label = "Referenčne točke p_n(x) /
 ↪\nReference points p_n(x)")
```

```
title = plt.title('Interpolacija z 9 oblikovnimi funkcijami /\n Interpolation␣
↪with 9 trial functions')
legend = plt.legend(handles = [g2], bbox_to_anchor=(1.05, 1), loc='upper left')
```

2.1.4 Aproksimacija s klasično metodo najmanjših kvadratov (MNK) / Classic least squares method (LSM) approximation

Kadar nas zanimajo gradienti spremenjivke f po domeni uporabimo aproksimacijo z metodo najmanjših kvadratov. Pri tej metodi je izbira števila funkcij v baznem vektorju ψ poljubna. V splošnem velja, da z več funkcijami dobimo boljšo natančnost rešitve. Te funkcije so lahko polinomi ali katerekoli druge oblikovne funkcije. V tem prikazu smo zaradi enostavnosti izbrali polinomske oblikovne funkcije. Metoda najmanjših kvadratov je definirana z enačbo:

$\sum_{n=1}^{N} \underline{\psi}(\mathbf{p}_n)\mathbf{c} = \sum_{n=1}^{N} \psi(\mathbf{p}_n) f_n;$

$\psi_{ij}(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n)$

When we are interested in gradients of variable f throughout the domain, the interpolation is not an adequate approach. In this case the least squares approximation is used. The number of trial functions used in the method is arbitrary. In general, use of more trial functions leads to more precise solution. Trial functions can be polynomials or any other trial functions. In the demonstration, we have chosen polynomial trial functions for simplicity. The least squares method is defined by the equation:

$\sum_{n=1}^{N} \underline{\psi}(\mathbf{p}_n)\mathbf{c} = \sum_{n=1}^{N} \psi(\mathbf{p}_n) f_n;$

$\psi_{ij}(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n)$

Implementacija klasične MNK v kodi / Implementation of classic LSM in code

[ ]:
```
def LSM(p,fi,order):
    """Pripravi aproksimacijsko funkcijo fun iz "order" oblikovnih funkcij␣
↪("order" <= 14).
    Metoda: klasičen MNK
    Argumenti: matrika točk: p, matrika vrednosti: fi in število oblikovnih␣
↪funkcij: order /
    Builds aproximation function fun from "order" trial function ("order" <= 14)
    Method: classic LSM
    Arguments: matrix of points: p, matrix of values: fi and number of trial␣
↪functions: order """
    #bazni vektor z oblikovnimi funkcijami / base vector with trial functions
    apr_fun = lambda x: np.
↪asarray([1,x,x**2,x**3,x**4,x**5,x**6,x**7,x**8,x**9,x**10,x**11,x**12,x**13,x**14][:
↪order+1])
    i = 0
    A = np.zeros([order+1, order+1]) #inic. matrike koeff / init. - matrix of␣
↪coeff
    b = np.zeros(order+1) #inic. vektorja konstant / init. - vector of constants
    while i < len(p): #gradnja sistema enačb / building system of equations
```

```
        Fi_p = np.outer(apr_fun(p[i]), apr_fun(p[i]))

        Fi_f = apr_fun(p[i]) * fi[0,i]

        A += Fi_p
        b += Fi_f
        i += 1
    c = np.linalg.solve(A,b) #reševanje sistema - dobimo vektor koeficientov c /
↪ solving of system - we get vector of coeff. c
    Cselect= np.append(c,np.zeros(15-len(c))) #filter c do reda 14 / filter c␣
↪to order 14
    #definicija aproksimirane funkcije / definition of aproximated function
    def fun(o):
        return Cselect[0] + Cselect[1]*o + \
        Cselect[2]*o**2 + Cselect[3]*o**3 + \
            Cselect[4]*o**4 + Cselect[5]*o**5 + \
                Cselect[6]*o**6 + Cselect[7]*o**7 + \
                    Cselect[8]*o**8 + Cselect[9]*o**9 + \
                        Cselect[10]*o**10+Cselect[11]*o**11+ \
                            Cselect[12]*o**12 + Cselect[13]*o**13 +\
                                Cselect[14]*o**14
    return fun
```

[ ]:
```
#izris rezultatov / plotting the results

def plotting_LSM(fun, fn, pn, order1, order2):
    %matplotlib inline
    x_points = np.linspace(0.5,9,1000)
    f1 = LSM(pn, fn, order1)
    f2 = LSM(pn, fn, order2)
    fig = plt.figure(2)
    g1, = plt.plot(x_points,f1(x_points),"-g",label = f'{order1} trial␣
↪functions')
    g2, = plt.plot(x_points,f2(x_points),"-b",label = f'{order2} trial␣
↪functions')
    g3, = plt.plot(pn,fn[0], "o",color='purple', label = "referenčne točke p_n /
↪\nreference points p_n")

    plt.legend(handles = [g1,g2,g3],
           title='Aproksimacija z MNK /\nAproximation with LSM',
           bbox_to_anchor=(1.05, 1), loc='upper left')

interact(plotting_LSM,
         fun = fixed(LSM),
         pn = fixed(pn),
         fn = fixed(fn),
         order1 = widgets.IntSlider(
```

```
            min=0,
            max=14,
            value=0,
            description="Število oblikovnih funkcij (prva krivulja) / Number␣
↪of trial functions (first curve)"),
        order2 = widgets.IntSlider(
            min=0,
            max=14,
            value=1,
            description="Število oblikovnih funkcij (druga krivulja) / Number␣
↪of trial functions (second curve)")
       )
```

2.1.5 Aproksimacija z metodo uteženih najmanjših kvadratov / Weighted least squares approximation

Kadar narava pojava veleva, da ima vrednost spremenljivke f v določeni točki $\overline{\mathbf{p}}$ še posebno velik vpliv na celotno fiziko, uporabimo metodo uteženih najmanjših kvadratov. Metoda je definirana z enačbo:

$\sum_{n=1}^{N} \theta(d_n)\underline{\psi}(\mathbf{p}_n)\mathbf{c}(\mathbf{p}) = \sum_{n=1}^{N} \theta(d_n)\psi(\mathbf{p}_n)f_n;$

$\psi_i j(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n).$

Vpliv vrednosti f v $\overline{\mathbf{p}}$ je ovrednoten z utežjo θ, ki je v našem primeru definirana z enačbo:

$\theta = e^{\frac{-d^2}{4}}$

in je odvisna zgolj od oddaljenosti $d = |\mathbf{p} - \mathbf{p}|$.

When value of the variable f in certain point $\overline{\mathbf{p}}$ has strong impact on the physics of the phenomena, we use weighted least squares approximation. The method is defined by equation:

$\sum_{n=1}^{N} \theta(d_n)\underline{\psi}(\mathbf{p}_n)\mathbf{c}(\mathbf{p}) = \sum_{n=1}^{N} \theta(d_n)\psi(\mathbf{p}_n)f_n;$

$\psi_i j(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n).$

The impact of the value of f in point $\overline{\mathbf{p}}$ is evaluated by weight function θ, that is defined by equation:

$\theta = e^{\frac{-d^2}{4}}$

and only depends on distance $d = |\mathbf{p} - \mathbf{p}|$.

Implementacija utežene MNK v kodi / Implementation of weighted LSM in code

[ ]:
```
def w_LSM(p,fi, pW, order):
    """Pripravi aproksimacijsko funkcijo fun iz "order" oblikovnih funkcij␣
↪("order" <= 14).
    Metoda: utežena MNK
    Argumenti: vektor točk: p, matrika vrednosti: fi, položaj utežne točke: pW
    in število oblikovnih funkcij: order /
```

```
    Builds approximation function fun from "order" trial function ("order" <=
↪14)
    Method: weighted LSM
    Arguments: matrix of points: p, matrix of values: fi, position of weight
↪point: pW
    and number of trial functions: order """
    #bazni vektor z oblikovnimi funkcijami / base vector with trial functions
    apr_fun = lambda x: np.
↪asarray([1,x,x**2,x**3,x**4,x**5,x**6,x**7,x**8,x**9,x**10][:order+1])
    i = 0
    A = np.zeros((order+1, order+1)) #inic. matrike koeff / init. - matrix of
↪coeff
    b = np.zeros(order+1) #inic. vektorja konstant / init. - vector of constants
    while i < len(p): #gradnja sistema enačb / building the system of equations
        d = np.absolute(p[i]-pW)
        W = np.exp(-d**2/4) # Utežna funkcija / Weight function

        Fi_p =np.outer(apr_fun(p[i]), apr_fun(p[i]))

        Fi_f = apr_fun(p[i]) * fi[0,i]
        A += Fi_p * W
        b += Fi_f * W
        i += 1
    c = np.linalg.solve(A,b) # reševanje sistema - dobimo vektor koeficientov c
↪/ solving of system - we get vector of coeff. c
    Cselect= np.append(c,np.zeros(11-len(c))) # dopolni c do reda 11 / fill-in
↪c up to order 11
    # definicija aproksimirane funkcije / definition of the approximated
↪function
    def fun(o):
        return Cselect[0] + Cselect[1]*o + \
        Cselect[2]*o**2 + Cselect[3]*o**3 + \
            Cselect[4]*o**4 + Cselect[5]*o**5 + \
                Cselect[6]*o**6 + Cselect[7]*o**7 + \
                    Cselect[8]*o**8 + Cselect[9]*o**9 + \
                        Cselect[10]*o**10

    return fun
```

[ ]:
```
# izris rezultatov / plotting the results

def plotting_w_LSM(fun, fn, pn, weight_curve_1,weight_curve_2, order1, order2):
    %matplotlib inline
    w1 = weight_curve_1
    w2 = weight_curve_2
    x_points = np.linspace(0.5, 9, 1000)
```

```
    f1 = w_LSM(p=pn, fi=fn, pW=w1, order=order1)
    f2 = w_LSM(p=pn, fi=fn, pW=w2, order=order2)
    fig = plt.figure(2)
    g1, = plt.plot(x_points,f1(x_points),"-g",label = f'{order1} trial␣
↪functions\nweight on point p = {w1}')
    g2, = plt.plot(x_points,f2(x_points),"-b",label = f'{order2} trial␣
↪functions\nweight on point p = {w2}')
    g3, = plt.plot(pn,fn[0], "o",color='purple', label = "referenčne točke p_n /
↪\nreference points p_n")

    plt.legend(handles = [g1,g2,g3],
           title='Aproksimacija z MNK /\n Aproximation with LSM',
           bbox_to_anchor=(1.05, 1), loc='upper left')
    return

interact(plotting_w_LSM,\
         fun = fixed(w_LSM),\
         pn = fixed(pn),\
         fn = fixed(fn),\
         order1 = widgets.IntSlider(\
             min=0,\
             max=14,\
             value=2,\
             description="Število oblikovnih funkcij (prva krivulja) / Number␣
↪of trial functions (first curve)"),\
         order2 = widgets.IntSlider(\
             min=0,\
             max=14,\
             value=3,\
             description="Število oblikovnih funkcij (druga krivulja) / Number␣
↪of trial functions (second curve)"),\
         weight_curve_1 = pn,\
         weight_curve_2 = pn\
        )
```

2.1.6 Aproksimacija z metodo premičnih najmanjših kvadratov / Moving least squares approximation

Kadar potrebujemo zelo natančno aproksimacijo spremenljivke f po domeni uporabimo premično metodo najmanjših kvadratov. V tem primeru rešujemo enačbo

$\sum_{n=1}^{N} \theta(d_n)\underline{\psi}(\mathbf{p}_n)\mathbf{c}(\overline{\mathbf{p}}) = \sum_{n=1}^{N} \theta(d_n)\psi(\mathbf{p}_n)f_n;$

$\psi_i j(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n)$

za vsako točko, kjer računamo vrednost f. Pri tem je utež θ definirana z razdaljo med točko z znano vrednostjo in točko v kateri rešujemo aproksimacijski problem $d = |\mathbf{p} - \mathbf{p}_n|$ preko utežne funkcije $\theta = e^{\frac{-d^2}{4}}$ (za ta prikaz).

If we want the most precise approximation, moving least squares method is used. In this case we are solving equation

$\sum_{n=1}^{N} \theta(d_n)\underline{\psi}(\mathbf{p}_n)\mathbf{c}(\overline{\mathbf{p}}) = \sum_{n=1}^{N} \theta(d_n)\psi(\mathbf{p}_n)f_n;$

$\psi_i j(\mathbf{p}_n) = \psi_i(\mathbf{p}_n)\psi_j(\mathbf{p}_n)$

for each point, where the value of f is being approximated. Weight θ depends on the distance $d = |\mathbf{p} - \mathbf{p}_n|$ between the point with known value of f and the point, where approximation is performed. In this demonstration weight function $\theta = e^{\frac{-d^2}{4}}$ is used.

Implementacija premične MPK v kodi / Implementation of moving LSM in code

[ ]:
```
def movingLSM(pn,fi, p,order):
    """Vrne aproksimirano vrednost v točki p, na podlagi znanih vrednosti fn v točkah pn in "order" oblikovnih funkcij ("order" <= 14).
    Metoda: premična MNK
    Argumenti: matrika točk: pn, matrika vrednopsti: fi, položaj  točke: p, kjer vrednost iščemo
    in število oblikovnih funkcij: order /
    Returns aproximated value in point p, based on known values fn in points pn and "order" trial functions ("order" <= 14).
    Method: moving LSM
    Arguments: matrix of points: pn, matrix of values: fi, position of weight point: p, where we aproximate value
    and number of trial functions: order
    """
    #bazni vektor z oblikovnimi funkcijami / base vector with trial functions
    apr_fun = lambda x: np.asarray([1,x,x**2,x**3,x**4,x**5,x**6,x**7,x**8,x**9,x**10][:order+1])
    i = 0
    A = np.zeros((order+1, order+1)) #inic. matrike koeff / init. - matrix of coeff
    b = np.zeros(order+1) #inic. vektorja konstant / init. - vector of constants
    while i < len(pn): #gradnja sistema enačb / buildng system of equations
        d = pn[i]-p
        W = np.exp(-d**2/4) #Funkcija uteži
        Fi_pn = np.outer(apr_fun(pn[i]), apr_fun(pn[i]))

        Fi_f = apr_fun(pn[i]) * fi[0,i]
        A += Fi_pn * W
        b += Fi_f * W
        i += 1
    c = np.linalg.solve(A,b)#reševanje sistema - dobimo vektor koeficientov c / solving of system - we get vector of coeff. c
    Cselect= np.append(c,np.zeros(11-len(c))) #filter c do reda 14 / filter c to order 14
    #izračun aproksimirane vrednosti v p / calculation of aproximated value in p
```

```
    aprox_value = Cselect[0] + Cselect[1]*p + \
        Cselect[2]*p**2 + Cselect[3]*p**3 + \
            Cselect[4]*p**4 + Cselect[5]*p**5 + \
                Cselect[6]*p**6 + Cselect[7]*p**7 + \
                    Cselect[8]*p**8 + Cselect[9]*p**9 + \
                        Cselect[10]*p**10
    return aprox_value
```

[ ]:
```
#izris rezultatov / plotting the results

def plotting_m_LSM(fun, fn, pn, order1, order2):
    x_points = np.linspace(0.5,9,100)
    y1=[]
    y2=[]
    for i in x_points:
        y1.append(movingLSM(pn,fn, i, order1))
        y2.append(movingLSM(pn,fn, i, order2))
    y1 = np.array(y1)
    y2 = np.array(y2)
    fig = plt.figure(3)
    g1, = plt.plot(x_points,y1,"-g",label = f'{order1} trial functions')
    g2, = plt.plot(x_points,y2,"-b",label = f'{order2} trial functions')
    g3, = plt.plot(pn,fn[0], "o",color='purple', label = "referenčne točke p_n /
 ↪\nreference points p_n")

    plt.legend(handles = [g1,g2,g3],
           title='Aproksimacija z premično MNK /\nAproximation with moving LSM',
           bbox_to_anchor=(1.05, 1), loc='upper left')


interact(plotting_m_LSM,
         fun = fixed(movingLSM),
         pn = fixed(pn),
         fn = fixed(fn),
         order1 = widgets.IntSlider(
             min=0,
             max=14,
             value=0,
             description="Število oblikovnih funkcij (prva krivulja) / Number␣
 ↪of trial functions (first curve)"),
         order2 = widgets.IntSlider(
             min=0,
             max=14,
             value=1,
             description="Število oblikovnih funkcij (druga krivulja) / Number␣
 ↪of trial functions (second curve)")
       )
```

2.1.7 Prikaz aproksimacije z MNK v 2D / Demonstration of approximation with LSM in 2D

Aproksimiramo s šestimi oblikovni funkcijami. / We approximate with 6 trial functions.

$\psi = (1, x, y, x^2, xy, y^2)$

[ ]:
```
def apr_fun2D(p):
    """Aproksimacija kvadratne funkcije v 2D, argument 2D vektor polozaja␣
↪p[x,y]"""
    return np.array([[1, p[0], p[1], p[0]**2, p[0]*p[1], p[1]**2]])
```

2.1.8 Seznam točk z znanimi vrednostmi f / List of points with known values of f

[ ]:
```
p = np.asarray([[1,1],[1,-1],[-1,1],[-1,-1],[0,0],[1,0],[-1,0],[0,1],[0,-1]])
fi1 = np.asarray([[1,-1,0,0,1,0,-1,-1,+1]])
fi2 = np.asarray([[+1,-0.5,1,1,-1,0,0,0,0]])
```

2.1.9 Implementacija klasične MNK v 2D / Implementation of classic LSM in 2D

[ ]:
```
def LSM_2D(apr_fun,p,fi):
    """Pripravi aproksimacijsko funkcijo fun iz 5 oblikovnih funkcij.
    Metoda: klasičen MNK 2D
    Argumenti: matrika točk: p, matrika vrednosti: fi /
    Builds aproximation function fun from 5 trial function.
    Method: classic LSM 2D
    Arguments: matrix of points: p, matrix of values: fi """
    i = 0
    A = np.zeros((len(apr_fun(p[0])[0]),len(apr_fun(p[0])[0]))) #inic. matrike␣
↪koeff / init. - matrix of coeff
    b = np.zeros(len(apr_fun(p[0])[0])) #inic. vektorja konstant / init. -␣
↪vector of constants
    while i < len(p[:,0]):#gradnja sistema enačb / buildng system of equations
        Fi_p = apr_fun(p[i]) * apr_fun(p[i]).T
        Fi_f = apr_fun(p[i]) * fi[0,i]
        A += Fi_p
        b += Fi_f[0]
        i += 1
    c = np.linalg.solve(A,b) #reševanje sistema - dobimo vektor koeficientov c /
↪ solving of system - we get vector of coeff. c
    return c
```

Izris rezultatov na 3D grafu. / Ploting the results on 3D plot.

[ ]:
```
c = LSM_2D(apr_fun2D, p, fi1)
f = lambda x, y: c[0] + c[1]*x + c[2]*y + c[3]*x**2 + c[4]*x*y + c[5]*y**2
x_points = np.linspace(-1,1,num=100)
y_points = np.linspace(-1,1,num=100)
X,Y = np.meshgrid(x_points, y_points)
Z = f(X,Y)
x_ref = p[:,0]
y_ref = p[:,1]
f_ref = fi1[0]
```

[ ]:
```
%matplotlib notebook
fig = plt.figure(1,figsize=(6,6))
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='red')
ax.set_title('Aproksimirana funkcija z referenčnimi točkami');
#Referencne točke / reference points
ax.scatter3D(x_ref, y_ref, f_ref,color="black");
ax.set_xlabel('px')
ax.set_ylabel('py')
ax.set_zlabel('fi');
```

Izris rezultatov na presečne ravnine / Ploting the results on section planes.

[ ]:
```
%matplotlib inline
fi = fi1[0]
fig2 = plt.figure(2)
g1, = plt.plot(x_points,f(x_points,0),"-b",label = "y=0")
g2, = plt.plot(x_points,f(x_points,1),"-b",label = "y=1")
g3, = plt.plot(x_points,f(x_points,-1),"-.b",label = "y=-1")
g4, = plt.plot(p[:,0],fi, "ob",label = "referenčne x točke")
g5, = plt.plot(y_points,f(0,y_points),"-g",label = "x=0")
g6, = plt.plot(y_points,f(1,y_points),"--g",label = "x=1")
g7, = plt.plot(y_points,f(-1,y_points),"-.",label = "x=-1")
g8, = plt.plot(p[:,1],fi,"og", label = "referenčne y točke")
plt.legend(handles=[g1,g2,g3,g4,g5,g6,g7,g8], title='Aproksimacijske krivulje',␣
 ↪bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xlabel('px ali py')
plt.ylabel('fi');
```

2.1.10 Implementacija utežene MNK v 2D / Implementation of weighted LSM in 2D

[ ]:
```
def w_LSM_2D(apr_fun,p,fi, pW):
    """Pripravi aproksimacijsko funkcijo fun iz 5 oblikovnih funkcij.
    Metoda: utežena MNK 2D
    Argumenti: vektor točk: p, matrika vrednosti: fi, položaj utežne točke: pW /
    Builds aproximation function fun from 5 trial function.
```

```
    Method: weighted LSM 2D
    Arguments: matrix of points: p, matrix of values: fi, position of weight␣
↪point: pW """
    i = 0
    A = np.zeros((len(apr_fun(p[0])[0]),len(apr_fun(p[0])[0]))) #inic. matrike␣
↪koeff / init. - matrix of coeff
    b = np.zeros(len(apr_fun(p[0])[0])) #inic. vektorja konstant / init. -␣
↪vector of constants
    while i < len(p[:,0]): #gradnja sistema enačb / buildng system of equations
        d = ((p[i,0]-pW[0])**2+(p[i,1]-pW[1])**2)**0.5
        W = np.exp(-d**2/4) #Funkcija uteži / weight function
        Fi_p = apr_fun(p[i]) * apr_fun(p[i]).T
        Fi_f = apr_fun(p[i]) * fi[0,i]
        A += Fi_p * W
        b += Fi_f[0] * W
        i += 1
    c = np.linalg.solve(A,b) #reševanje sistema - dobimo vektor koeficientov c /
↪ solving of system - we get vector of coeff. c
    return c
```

Utež postavimo na točko $(1, -1)$. / The weight is set at point (1,-1)

[ ]:
```
pW = np.asarray([1,-1])
```

Izris rezultatov na 3D grafu / Ploting the results on 3D plot

[ ]:
```
c = w_LSM_2D(apr_fun2D, p, fi1, pW)
f = lambda x, y: c[0] + c[1]*x + c[2]*y + c[3]*x**2 + c[4]*x*y + c[5]*y**2

x_points = np.linspace(-1,1,num=100)
y_points = np.linspace(-1,1,num=100)
X,Y = np.meshgrid(x_points, y_points)
Z = f(X,Y)

x_ref = p[:,0]
y_ref = p[:,1]
f_ref = fi1[0]
```

[ ]:
```
%matplotlib notebook
fig = plt.figure(figsize=(6,6))
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='red')
ax.set_title('Aproksimirana funkcija z referenčnimi točkami');
#Referencne točke / reference points
ax.scatter3D(x_ref, y_ref, f_ref,color="black");
ax.scatter3D([pW[0]],[pW[1]],[-2], color="green", linewidth=10);
```

```
ax.set_xlabel('px')
ax.set_ylabel('py')
ax.set_zlabel('fi');
```

Izris rezultatov na presečne ravnine / Plotting the results on section planes

[ ]:
```
%matplotlib inline
fi = fi1[0]
fig2 = plt.figure()
g1, = plt.plot(x_points,f(x_points,0),"-b",label = "y=0")
g2, = plt.plot(x_points,f(x_points,1),"-b",label = "y=1")
g3, = plt.plot(x_points,f(x_points,-1),"-.b",label = "y=-1")
g10, = plt.plot(pW[0],[0],"o", label = "Utežna točka X\ny=0", markersize=10,␣
↪color = 'pink')
g4, = plt.plot(p[:,0],fi, "ob",label = "referenčne x točke")
g5, = plt.plot(y_points,f(0,y_points),"-g",label = "x=0")
g6, = plt.plot(y_points,f(1,y_points),"--g",label = "x=1")
g7, = plt.plot(y_points,f(-1,y_points),"-.",label = "x=-1")
g9, = plt.plot(pW[1],[0],"o", label = "Utežna točka Y\n(x=0)", markersize=10,␣
↪color = 'purple')
g8, = plt.plot(p[:,1],fi,"og", label = "referenčne y točke")
plt.xlabel('px ali py')
plt.ylabel('fi')
l=plt.legend(handles=[g1,g2,g3,g4,g10,g5,g6,g7,g8,g9], title='Aproksimacijske␣
↪krivulje',
             bbox_to_anchor=(1.05, 1), loc='upper left')
```

2.1.11 Implementacija premične MNK v 2D / Implementation of moving LSM in 2D

[ ]:
```
def movingLSM_2D(apr_fun,pn,fi, p):
    """Vrne aproksimirano vrednost v točki p, na podlagi znanih vrednosti fn v␣
↪točkah pn in 5 oblikovnih funkcij.
    Metoda: premična MNK 2D
    Argumenti: matrika točk: pn, matrika vrednopsti: fi, položaj  točke: p,␣
↪kjer vrednost iščemo /
    Returns aproximated value in point p, based on known values fn in points pn␣
↪and 5 trial functions.
    Method: moving LSM 2D
    Arguments: matrix of points: pn, matrix of values: fi, position of weight␣
↪point: p, where we aproximate value"""
    i = 0
    A = np.zeros((len(apr_fun(pn[0])[0]),len(apr_fun(pn[0])[0]))) #inic.␣
↪matrike koeff / init. - matrix of coeff
    b = np.zeros(len(apr_fun(pn[0])[0])) #inic. vektorja konstant / init. -␣
↪vector of constants
```

```
    while i < len(pn[:,0]): #gradnja sistema enačb / buildng system of equations
        d = ((pn[i,0]-p[0])**2+(pn[i,1]-p[1])**2)**0.5
        W = np.exp(-d**2/4) #Funkcija uteži / weight function
        Fi_pn = apr_fun(pn[i]) * apr_fun(pn[i]).T
        Fi_f = apr_fun(pn[i]) * fi[0,i]
        A += Fi_pn * W
        b += Fi_f[0] * W
        i += 1
    c = np.linalg.solve(A,b) #reševanje sistema - dobimo vektor koeficientov c /
↪ solving of system - we get vector of coeff. c
    #izračun aproksimirane vrednosti v p / cal. of aproximated walue in p
    aprox_value = c[0] + c[1]*p[0] + c[2]*p[1] + c[3]*p[0]**2 + c[4]*p[0]*p[1]␣
↪+ c[5]*p[1]**2
    return aprox_value
```

Izris rezultatov na 3D grafu / Ploting the results on 3D plot

[ ]:
```
#Risanje grafa
min_border = -1
max_border = 1
num_points = 100
x_points = np.linspace(min_border,max_border,num=num_points)

X,Y = np.meshgrid(x_points, x_points)

Z = np.ones(np.shape(X))

for i in range(np.shape(X)[0]):
     for k in range(np.shape(X)[0]):
         Z[i,k] =  movingLSM_2D(apr_fun=apr_fun2D, pn = p, fi = fi1, p = np.
↪asarray([X[i,k],Y[i,k]]))

x_ref = p[:,0]
y_ref = p[:,1]
f_ref = fi1[0]
```

[ ]:
```
%matplotlib notebook
fig = plt.figure(50,figsize=(6,6))
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='red')
ax.set_title('Aproksimirana funkcija z referenčnimi točkami');
#Referencne točke
ax.scatter3D(x_ref, y_ref, f_ref,color="black");
#ax.scatter3D([pW[0]],[pW[1]],[-2], color="green", linewidth=10);

ax.set_xlabel('px')
ax.set_ylabel('py')
```

```
ax.set_zlabel('fi');
```

2.2 Metoda končnih volumnov za konvekcijsko difuzijske probleme / The Finite Volume Method for Convection-Diffusion problems

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (104 - 125).

2.2.1 Definicija naloge / Problem definition

Ohranjena količina ϕ se v enodimenzionalni domeni prenaša z robnimi pogoji $\phi_0 = 0$ pri $x = 1$ in $\phi_L = 0$ pri $x = L$. Reši stacionarno konvekcijsko-difuzijsko enačbo za različne vrednosti hitrosti u:

$$\frac{\partial}{\partial x}(\rho u \phi) = \frac{\partial}{\partial x}(\Gamma \frac{\partial}{\partial x}\phi)$$

Podatki:

- Dolžina domene: $L = 1m$
- Gostota: $\rho = 1\frac{kg}{m^3}$
- Prevodnost: $\Gamma = 0.1\frac{kg}{ms}$

Rešitev dobljeno z MKV primerjaj z analitično rešitvijo:

$$\frac{\phi - \phi_0}{\phi_L - \phi_0} = \frac{\exp(\rho u x/\gamma) - 1}{\exp(\rho u L/\gamma) - 1}$$

A conserved property ϕ is transported through the one-dimensional domain with boundary conditions $\phi_0 = 0$ at $x = 1$ and $\phi_L = 0$ at $x = L$. Solve the stationary convection-diffusion equation for different values of velocity u:

$$\frac{\partial}{\partial x}(\rho u \phi) = \frac{\partial}{\partial x}(\Gamma \frac{\partial}{\partial x}\phi)$$

.

The data:

- Domain length: $L = 1m$
- Density: $\rho = 1\frac{kg}{m^3}$
- Conductivity: $\Gamma = 0.1\frac{kg}{ms}$

Compare the solution obtained by FVM with the analytical solution:

$$\frac{\phi - \phi_0}{\phi_L - \phi_0} = \frac{\exp(\rho u x/\Gamma) - 1}{\exp(\rho u L/\Gamma) - 1}$$

.

[ ]:
```
# -*- coding: utf-8 -*-

#advection_central_difference, a demonstration of FVM for transport equation in␣
 ↪1D
#Copyright (C) 2020  Boštjan Mavrič, Matjaž Zadnik
#
#This program is free software: you can redistribute it and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.
#
#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.
#
#You should have received a copy of the GNU General Public License
#along with this program.  If not, see <http://www.gnu.org/licenses/>.

from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual



def TDMAsolver(a, b, c, d):
        '''
        TDMA solver, a b c d can be NumPy array type or Python list type.
        refer to http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
        and to http://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-
_TDMA_(Thomas_algorithm)
        '''
        nf = len(d) # number of equations
        ac, bc, cc, dc = map(np.array, (a, b, c, d)) # copy arrays
        for it in range(1, nf):
            mc = ac[it-1]/bc[it-1]
            bc[it] = bc[it] - mc*cc[it-1]
            dc[it] = dc[it] - mc*dc[it-1]

        xc = bc
        xc[-1] = dc[-1]/bc[-1]

        for il in range(nf-2, -1, -1):
            xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]
```

```
        return xc

def CD_advection(N, u):
    # Transport v 1D z advekcijo in difuzijo / 1D advection-diffusion transport

    # Robni pogoji / boundary conditions
    fiA=1 # pri / at x=0
    fiB=0 # pri / at x=L

    # Snovni parametri / material parameters
    L=1 # [m] dolžina odseka / domain length
    rho=1 # [kg/m^3] gostota transportirane količine / density
    Gama=0.1 # [kg/ms] posplošen koeficient prevodnosti / conductivity

    # Diskretizacija / discretization details
    dx=L/N # [m] krajevni korak / spatial step size


    # mrežne točke od 2 do N-1 / mesh points from 2 to N-1
    F=rho*u
    D=Gama/dx
    Pe=F/D

    aE=D-F/2
    aW=D+F/2
    aP=aE+aW
    Su=0

    # Matrika sistema brez upoštevanja robnih pogojev / System matrix without␣
 ↪the boundary conditions
    d=aP*np.ones(N) # diagonala / diagonal entries
    s=-aW*np.ones(N-1) # spodnja obdiagonala / lower diagonal
    z=-aE*np.ones(N-1) # zgornja obdiagonala / upper diagonal
    b=Su*np.ones(N) # vektor vrednosti izvorov / vector of source terms


    # levi rob (točka 1) / left boundary (point 1)
    aE=D-F/2
    aW=0
    Sp=-(2*D+F)
    Su=(2*D+F)*fiA
    aP=aE+aW-Sp

    d[0]=aP
    b[0]=Su

    # Desni rob (točka N) / right boundary (point N)
```

```
    aE=0
    aW=D+F/2
    Sp=-(2*D-F)
    Su=(2*D-F)*fiB
    aP=aE+aW-Sp

    d[N-1]=aP
    b[N-1]=Su

    # Rešimo sistem enačb / Solve the system of equations
    fi=TDMAsolver(s, d, z, b)

    # krajevna mreža / Spatial mesh
    x_tab=np.array([dx*i for i in range(N)])
    x_tab=x_tab+dx/2

    def analiticFi(x):
        'Točna rešitev'
        return fiA+(fiB-fiA)*(np.exp(rho*u*x/Gama)-1)/(np.exp(rho*u*L/Gama)-1)

    # Graf phi profila / Plot the profile of phi
    y_values=analiticFi(x_tab)
    eps_rel=np.linalg.norm(y_values-fi)/np.linalg.norm(y_values)

    title_string="Central difference scheme \n Pe=%.2f    Relative $L_2$ error=␣
↪%.2e" % ( Pe, eps_rel)
    plt.plot(x_tab,fi,'bo',label='FVM')
    x_tab1=np.linspace(0,N*dx,1000)
    plt.plot(x_tab1,analiticFi(x_tab1),'b-',label='analytical')
    plt.xlim(0,N*dx)
    plt.xlabel('position $x$ (m)')
    plt.ylabel('$\Phi$ ')
    plt.legend(loc='best')
    plt.title(title_string)
    plt.show()
```

[ ]:
```
#Pripravi UI / generate UI
N_slider = widgets.IntSlider(min=5, max=50, step=5, value=5)
u_slider = widgets.FloatSlider(min=-10, max=10, step=0.5, value=2.5)

w=widgets.interactive(CD_advection,
                      N=N_slider,
                      u=u_slider
                     )

display(w)
```

```
#CD_advection(5, 2.5)
```

2.3 Stabilizacija metode končnih volumnov za konvekcijsko difuzijske probleme / Stabilization of the finite volume method for convection-diffusion problems

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (104 - 125).

2.3.1 Definicija naloge / Task definition

Reši isti transportni problem kot v 'advection_central_difference.ipynb', toda tokrat z uporabo priveterne metode prvega reda.

Solve the same transport problem as in 'advection_central_difference.ipynb', but this time with the first order upwinding formula.

2.3.2 Priveterna shema prvega reda / First-order upwinding

Priveterna shema prvega reda spremeni izračun konvekcijskih tokov glede na smer advekcijske hitrosti u. S tem dobimo spremenjene formule za koeficienta a_W in a_E.

First order upwinding changes the values of convection fluxes depending on the direction of advection velocity u. This gives the following modified formulas for coefficients a_W and a_E.

$$a_W = D_W + \max(F_W, 0)$$

$$a_E = D_E + \max(-F_E, 0)$$

[ ]:
```
# -*- coding: utf-8 -*-

#advection_first_order_UW, a demonstration of stabilized FVM for transport␣
 ↪equation in 1D
#Copyright (C) 2020  Boštjan Mavrič, Matjaž Zadnik
#
#This program is free software: you can redistribute it and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.
#
#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.
```

```
#
#You should have received a copy of the GNU General Public License
#along with this program.  If not, see <http://www.gnu.org/licenses/>.

from __future__ import division
import matplotlib.pyplot as plt
import numpy as np

import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual


def upwind_advection(N, u):
    # Transport v 1D z advekcijo in difuzijo / 1D transport with advection and␣
 ↪diffusion

    # Robni pogoji / Boundary conditions
    fiA=1 # pri/at x=0
    fiB=0 # pri/at x=L

    # Snovni parametri / Material parameters
    L=1 # [m] dolžina odseka / domain length
    rho=1 # [kg/m^3] gostota transportirane količine / density
    #=20 # [m/s] hitrost tekočine / velocity
    Gama=0.1 # [kg/ms] posplošen koeficient prevodnosti / conductivity

    # Diskretizacija / discretization
    #N=50 # število mrežnih točk / number of discretization points
    dx=L/N # [m] krajevni korak / spatial step

    F=rho*u
    D=Gama/dx
    Pe=F/D
    # mrežne točke od 2 do N-1 / mesh points from 2 to N-1
    # Ločimo dva primera glede na predznak u. / We consider two cases,␣
 ↪depending on the sign of u.
    if u>0:
        aE=D
        aW=D+F
        aP=aW+aE

        # Matrika sistema brez upoštevanja robnih pogojev / System matrix␣
 ↪without accounting for the boundary conditions
        d=aP*np.ones(N) # diagonala / diagonal entries
        s=-aW*np.ones(N-1) # spodnja obdiagonala / upper diagonal
        z=-aE*np.ones(N-1) # zgornja obdiagonala / lower diagonal
```

```
        # vektor vrednosti izvorov / Source terms vector
        b=np.zeros(N)

        # levi rob (točka 1) / left boundary (point 1)
        aE=D
        aW=0
        Sp=-(2*D+F)
        Su=(2*D+F)*fiA
        aP=aE+aW-Sp

        d[0]=aP
        b[0]=Su

        # Desni rob (točka N) / right boundary (point N)
        aE=0
        aW=D+F
        Sp=-2*D
        Su=2*D*fiB
        aP=aE+aW-Sp

        d[N-1]=aP
        b[N-1]=Su

    else:
        aE=D-F
        aW=D
        aP=aW+aE

        # Matrika sistema brez upoštevanja robnih pogojev / System matrix␣
↪without accounting for the boundary conditions
        d=aP*np.ones(N) # diagonala / Diagonal entries
        s=-aW*np.ones(N-1) # spodnja obdiagonala / lower diagonal
        z=-aE*np.ones(N-1) # zgornja obdiagonala / upper diagonal

        # vektor vrednosti izvorov / Source term vector
        b=np.zeros(N)

        # levi rob (točka 1) / left boundary (point 1)
        aE=D-F
        aW=0
        Sp=-2*D
        Su=2*D*fiA
        aP=aE+aW-Sp

        d[0]=aP
        b[0]=Su
```

```
        # Desni rob (točka N) / Right boundary (point N)
        aE=0
        aW=D+F
        Sp=-(2*D-F)
        Su=(2*D-F)*fiB
        aP=aE+aW-Sp

        d[N-1]=aP
        b[N-1]=Su

    #### rešimo sistem enačb A x=b z algoritmom, ki upošteva tridiagonalno␣
↪zgradbo matrike A ###
    ### / Solve the system of equations AT=b using an algorithm that is␣
↪specialized for tridiagonal matrices ###
    def TDMAsolver(a, b, c, d):
        '''
        TDMA solver, a b c d can be NumPy array type or Python list type.
        refer to http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
        and to http://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-
_TDMA_(Thomas_algorithm)
        '''
        nf = len(d) # number of equations
        ac, bc, cc, dc = map(np.array, (a, b, c, d)) # copy arrays
        for it in range(1, nf):
            mc = ac[it-1]/bc[it-1]
            bc[it] = bc[it] - mc*cc[it-1]
            dc[it] = dc[it] - mc*dc[it-1]

        xc = bc
        xc[-1] = dc[-1]/bc[-1]

        for il in range(nf-2, -1, -1):
            xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]

        return xc

    fi=TDMAsolver(s, d, z, b)

    # Analitična rešitev / Analytical solution
    def analiticFi(x):
        'Točna rešitev'
        return fiA+(fiB-fiA)*(np.exp(rho*u*x/Gama)-1)/(np.exp(rho*u*L/Gama)-1)


    # krajevna mreža / Spatial grid
    x_tab=np.array([dx*i for i in range(N)])
    x_tab=x_tab+dx/2
```

```
    # Graf phi profila / Plot the phi profile
    y_values=analiticFi(x_tab)
    eps_rel=np.linalg.norm(y_values-fi)/np.linalg.norm(y_values)

    title_string=" First order upwind scheme \n Pe=%.2f    Relative $L_2$␣
↪error= %.2e" % ( Pe, eps_rel)
    plt.plot(x_tab,fi,'bo',label='numerical')
    x_tab1=np.linspace(0,N*dx,1000)
    plt.plot(x_tab1,analiticFi(x_tab1),'b-',label='analytical')
    plt.xlim(0,N*dx)
    plt.xlabel('Position $x$ (m)')
    plt.ylabel('$\Phi$ ')
    plt.legend(loc='best')
    plt.title(title_string)
    plt.show()
```

[ ]:
```
N_slider = widgets.IntSlider(min=5, max=50, step=5, value=5)
u_slider = widgets.FloatSlider(min=-10, max=10, step=0.5, value=2.5)
w=widgets.interactive(upwind_advection,
                      N=N_slider,
                      u=u_slider
                     )
display(w)
```

2.4 Stabilizacija metode končnih volumnov za konvekcijsko difuzijske probleme / Stabilization of the finite volume method for convection-diffusion problems

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (104 - 125).

2.4.1 Definicija naloge / Task definition

Reši isti transportni problem kot v 'advection_central_difference.ipynb', toda tokrat z uporabo hibridne diferenčne metode.

Solve the same transport problem as in 'advection_central_difference.ipynb', but this time with hybrid differencing method.

2.4.2 Hibridna diferenčna metoda / Hybrid differencing method

Hibridna diferenčna shema rešuje konvekcijsko-difuzijske enačbe s kombinacijo centralne diferenčne sheme pri nizkih Pecletovih številih in priveterne sheme prvega reda za visoka Pecletova števila. Za ta primer so uteži a_W in a_E:

Hybrid difference scheme solves the convection-diffusion equation by combining central difference scheme at low Peclet numbers and first order upwinding scheme for high Peclet numbers. In this case the coefficients a_W and a_E are:

$$a_W = \max\left[F_W, (D_W + F_W/2), 0\right]$$

$$a_E = \max\left[-F_E, (D_E - F_E/2), 0\right]$$

[ ]:
```
# -*- coding: utf-8 -*-

#advection_first_order_UW, a demonstration of stabilized FVM for transport equation in 1D
#Copyright (C) 2020  Boštjan Mavrič, Matjaž Zadnik
#
#This program is free software: you can redistribute it and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.
#
#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.
#
#You should have received a copy of the GNU General Public License
#along with this program.  If not, see <http://www.gnu.org/licenses/>.

from __future__ import division
import matplotlib.pyplot as plt
import numpy as np

import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual


def upwind_advection(N, u):
    # Transport v 1D z advekcijo in difuzijo / 1D advection-diffusion transport

    # Robni pogoji / Boundary conditions
    fiA=1 # pri / at x=0
    fiB=0 # pri / at x=L

    # Snovni parametri / Material parameters
    L=1 # [m] dolžina odseka / Domain length
    rho=1 # [kg/m^3] gostota transportirane količine / density
    #=20 # [m/s] hitrost tekočine / velocity
    Gama=0.1 # [kg/ms] posplošen koeficient prevodnosti / conductivity
```

```
    # Diskretizacija / Discretization
    dx=L/N # [m] krajevni korak / spatial step

    F=rho*u
    D=Gama/dx

    aW=np.max([F,(D + F/2), 0e0])
    aE=np.max([-F,(D - F/2), 0e0])

    # Pripravi matriko sistema linearnih enačb / Prepare the matrix of linear␣
↪system
    Pe=F/D
    if (abs(Pe)<2e0):

        aP=aW+aE

        # Matrika sistema brez upoštevanja robnih pogojev / System matrix␣
↪without the boundary conditions
        d=aP*np.ones(N) # diagonala / Diagonal elements
        s=-aW*np.ones(N-1) # spodnja obdiagonala / lower diagonal
        z=-aE*np.ones(N-1) # zgornja obdiagonala / upper diagonal

        # vektor vrednosti izvorov / source term vector
        b=np.zeros(N)

        # levi rob (točka 1) / left boundary (point 1)
        d[0]=(F/2+3*D)
        b[0]=(2*D+F)*fiA


        # Desni rob (točka N) / right boundary (point N)
        d[N-1]=3*D-F/2
        b[N-1]=(2*D+F)*fiB

    elif (Pe >= 2e0):
        aP=aW+aE
        # Matrika sistema brez upoštevanja robnih pogojev / System matrix␣
↪without boudnary conditions
        d=aP*np.ones(N) # diagonala / diagonal
        s=-aW*np.ones(N-1) # spodnja obdiagonala / lower diagonal
        z=-aE*np.ones(N-1) # zgornja obdiagonala / upper diagonal

        # vektor vrednosti izvorov / source term vector
        b=np.zeros(N)

        # levi rob (točka 1) / left boundary (point 1)
```

```
        d[0]=(F+2*D)
        b[0]=(2*D+F)*fiA

        # Desni rob (točka N) / right boundary (point N)
        d[N-1]=2*D + F
        b[N-1]=(2*D)*fiB

    elif (Pe <= -2e0):
        aP=aW+aE
        # Matrika sistema brez upoštevanja robnih pogojev / System matrix␣
↪without boundary conditions
        d=aP*np.ones(N) # diagonala / diagonal
        s=-aW*np.ones(N-1) # spodnja obdiagonala / lower diagonal
        z=-aE*np.ones(N-1) # zgornja obdiagonala / upper diagonal

        # vektor vrednosti izvorov / source term vector
        b=np.zeros(N)

        # levi rob (točka 1) / left boundary (point 1)
        d[0]=(-F+2*D)
        b[0]=(2*D)*fiA

        # Desni rob (točka N) / right boundary (point N)
        d[N-1]=-2*D + F
        b[N-1]=(2*D - F)*fiB

    #### rešimo sistem enačb AT=b z algoritmom, ki upošteva tridiagonalno␣
↪zgradbo matrike A ###
    #### Solve the system of equations AT=b, with an algorithm for tridiagonal␣
↪matrices ###
    def TDMAsolver(a, b, c, d):
        '''
        TDMA solver, a b c d can be NumPy array type or Python list type.
        refer to http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
        and to http://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-
_TDMA_(Thomas_algorithm)
        '''
        nf = len(d) # number of equations
        ac, bc, cc, dc = map(np.array, (a, b, c, d)) # copy arrays
        for it in range(1, nf):
            mc = ac[it-1]/bc[it-1]
            bc[it] = bc[it] - mc*cc[it-1]
            dc[it] = dc[it] - mc*dc[it-1]

        xc = bc
        xc[-1] = dc[-1]/bc[-1]
```

```
        for il in range(nf-2, -1, -1):
            xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]

        return xc

    fi=TDMAsolver(s, d, z, b)

    # Analitična rešitev / Analytic solution
    def analiticFi(x):
        'Točna rešitev'
        return fiA+(fiB-fiA)*(np.exp(rho*u*x/Gama)-1)/(np.exp(rho*u*L/Gama)-1)


    # krajevna mreža / Spatial mesh
    x_tab=np.array([dx*i for i in range(N)])
    x_tab=x_tab+dx/2

    # Graf rešitve / Produce a plot of solution
    y_values=analiticFi(x_tab)
    eps_rel=np.linalg.norm(y_values-fi)/np.linalg.norm(y_values)

    title_string=" Hibridna shema \n Pe=%.2f    Relativna $L_2$ napaka= %.2e" %␣
↪( Pe, eps_rel)

    plt.plot(x_tab,fi,'bo',label='numerično')
    x_tab1=np.linspace(0,N*dx,1000)
    plt.plot(x_tab1,analiticFi(x_tab1),'b-',label='analitično')
    plt.xlim(0,N*dx)
    plt.xlabel('razdalja $x$ (m)')
    plt.ylabel('$\Phi$ ')
    plt.legend(loc='best')
    #plt.title('1D profil kolicine $\Phi$')
    plt.title(title_string)
    plt.show()
```

[ ]:
```
#Pripravi GUI za manipulacijo parameterov / Prepare parameter manipulation GUI
N_slider = widgets.IntSlider(min=5, max=50, step=5, value=5)
u_slider = widgets.FloatSlider(min=-10, max=10, step=0.5, value=2.5)
w=widgets.interactive(upwind_advection,
                      N=N_slider,
                      u=u_slider
                     )
display(w)
```

Poglavje 3

Časovno odvisni problemi / Time-dependent problems

Skripte, ki implementirajo metode predstavljene v tem poglavju je možno prenesti iz repozitorija na https://github.com/bmavric/transient_FVM.

The scripts implementing the methods presented in this section can be downloaded from the repository at https://github.com/bmavric/transient_FVM.

Implementacije so v naslednjih datotekah: / The implementations are in the following files:

- `Eksplicitna_dif_1D/Eksplicitna_Difuzija_1D.ipynb` - Eksplicitna Eulerjeva metoda za reševanje difuzije / Explicit Euler method for diffusion
- `Implicitna_dif_1D/Implicitna_Difuzija_1D.ipynb` - Implicitna Eulerjeva metoda za reševanje difuzije / Implicit Euler method for diffusion
- `Crank-Nicholson_dif_1D/Crank-Nicolson_Difuzija_1D.ipynb` - Crank-Nicholsonova metoda za reševanje difuzije / Crank-Nicholson method for diffusion

3.1 Implementacija eksplicitne metode na 1D difuzijskem problemu / Implementation of explicit method for 1D diffusion problem

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (245-256).

3.1.1 Definicija naloge / Problem definition

Tanka plošča ima na začetku temperaturo $200°C$ po celotnem volumnu. Ob času $t = 0\,s$ temperatura vzhodnega robu plošče pade na $0°C$. Vse ostale površine plošče so izolirane. Z uporabo eksplicitne metode s primernim časovnim korakom izračunaj časovni potek temperaturne razporeditve po plošči in jo primerjaj z analitično rešitvijo pri časih: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

Preračunaj numerično rešitev, če uporabimo kritični časovni korak: $\Delta t = \rho c \frac{(\Delta x)^2}{2k} (= 8\,s)$ za čas $t = 40\,s$ in primerjaj rezultat z analitično rešitvijo.

Podatki:

- Debelina plošče: $L = 2cm$
- Toplotna prevodnost plošče: $k = 10 \frac{W}{mK}$
- Volumska specifična toplotna kapaciteta: $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Prikaži še numerično rešitev za vse tri vrste robnih pogojev (Dirichlet, Neumann, Robin) na vzhodnem oziroma zahodnem robu plošče. Pri tem upoštevaj konstantno temperaturo okoliške tekočine $T_\infty = 20°C$ in koeficient toplotne prestopnosti $h = 15 \frac{W}{m^2 K}$.

A thin plate is initially at a uniform temperature of $200°C$. At a certain time $t = 0\,s$ the temperature of the east side of the plate is suddenly reduced to $0°C$. The other surface is insulated. Use the explicit finite volume method in conjunction with a suitable time step size to calculate the transient temperature distribution of the slab and compare it with the analytical solution at time: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

Recalculate the numerical solution using a time step size equal to the limit $\Delta t = \rho c \frac{(\Delta x)^2}{2k} (= 8\,s)$ for $t = 40\,s$ and compare the results with the analytical solution.

The data:

- Plate thickness: $L = 2cm$
- Thermal conductivity: $k = 10 \frac{W}{mK}$
- Volume specific heat capacity $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Show numerical solution for all 3 types of boundary conditions (Dirichlet, Neumann, Robin) on east or west boundary. Assume constant temperature of surrounding fluid $T_\infty = 20°C$ and coefficient of heat transfer $h = 15 \frac{W}{m^2 K}$.

3.1.2 Osnove / Background

Vodilna enačba za 1D prevod toplote v časovno neustaljenih pogojih: / Governing equation for unsteady 1D heat conduction:

$$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}(k \frac{\partial T}{\partial x}) + S$$

Eksplicitna diskretizacija: / Explicit discretisation:

$$a_P T_P = a_w T_W^0 + a_E T_E^0 + [a_P^0 - (a_W + a_E - S_p)] T_P^0 + S_u$$

Kjer je: / Where is:

$$a_P = a_P^0$$

$$a_P^0 = \rho c \frac{\Delta x}{\Delta t}$$

$$a_W = \frac{k_w}{\delta x_{WP}}$$

$$a_E = \frac{k_e}{\delta x_{PE}}$$

3.1.3 Robni pogoji / Boundary conditions:

Dirichletov robni pogoj / Dirichlet boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0$
- $S_u = \frac{2k_w}{\delta x_{WP}}(T_B - T_P^0)$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0$
- $S_u = \frac{2k_e}{\delta x_{PE}}(T_B - T_P^0)$

Neumannov robni pogoj / Neumann boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0$
- $S_u = q_w$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0$
- $S_u = q_e$

Robinov robni pogoj / Robin boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$

- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0$
- $S_p = -\frac{h}{k_w}$
- $S_u = \frac{h}{k_w} T_\infty$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0$
- $S_p = -\frac{h}{k_e}$
- $S_u = \frac{h}{k_e} T_\infty$

3.1.4 Implementacija metode v kodi / Implementation of method in code:

[ ]:
```
#za lažje delo z matrikami uvozimo knjižnjico numpy / for easier work with␣
↪matrices numpy library is imported
import numpy as np
```

[ ]:
```
def explicit_diffusion1D(n, t, dt='Auto', L = 0.02, k = 10, Tinit = 200,␣
↪boundaryW = 'Dirichlet', boundaryE = 'Dirichlet',  \
                         TBW = 0, TBE = 0, roc = 10*10**6, h = 15, Tinf = 20, q␣
↪= 0):
    """Funkcija omogoča reševanje 1D difuzije toplote v časovno neustaljenem␣
↪stanju, brez izvornega člena, z eksplicitno metodo.

    Dovoljeni robni pogoji (vrstni red ni pomemben): / Allowed boundary␣
↪conditions (order is not important):
    Dirichlet-Dirichlet, Dirichlet-Neumann, Dirichlet-Robin, Robin-Neumann

    Vhodni podatki / input data:
    št. delitev / number of division -> n [/]
    čas simulacije / total time of simulation -> t [s]
    časovni korak / time step -> dt [s], privzeta vrednost / default value dt =␣
↪rho * cp * dx^2/2k
    Dolžina domene / domain legth -> L[m]
    Toplotna prevodnost / thermal conductivity -> k [W/m.K]
    Začetna temperatura / initial temperature -> Tinit [°C]
    Tipa robnih pogojev / types of boundary condition boundaryW in/and boundaryE
    Temperatura zahodnega robu (RP) / West boundary temp. (BC) -> TBW [°C]
    Temperatura vzhodnega robu (RP) / East boundary temp. (BC) -> TBE [°C]
```

```
   Volumsko specifična termičnaa masa / Volume specific thermal mas -> roc [J/
↪m^3.K]
   Toplotna prestopnost (RP) / coefficient of heat transfer (BC) -> h [W/m^2.K]
   Temperatura okoliške tekočine (RP) / Sorrounding fluid temp. (BC) -> Tinf␣
↪[°C]
   Toplotni tok na robu (RP) / Heat flux (boundary cindition) (BC) -> q[W/m^2]

   Rezultat funkcije je numpy numerično polje (vrstica - dt, stolpec - dx). /
   Result of function is numpy array (row - dt, column -dx).

   """

    #velikost prostorske diskretizacije (dx) / spatial discretisation size (dx)
   dx = L / n

   #inicializacija / initialisation
   Temperatures = np.asarray([]) #inicializacija matrike rezultatov (temp. v␣
↪časovnem koraku) / init. of results matrix (temp. in time step)
   Temp_int = Tinit * np.ones((1,n)) #vrstični vektor začetnih temperatur␣
↪(t=0)/ initial temperature row vector (t=0)
   Temperatures = Temp_int #dodajanje Tinit v matriko rez. / adding Tinit to␣
↪results matrix
   tc = 0. #inic. časa simulacije / init. of time of simulation

   #določanje privzetega časovnega koraka (največji dovoljen) / init. of␣
↪default time step (max allowed)
   if dt == 'Auto':
       dt = roc * x**2/(2*k)

   #zanka za "časovno korakanje" naprej do max. časa sim. t / loop for forward␣
↪"time marching" up to max. simulation time t
   while tc < t:
       tc += dt #izračun pretečenega časa v trenutnem časovnem koraku /␣
↪calculation of elapsed time in current time step

       #inicializacija vektorja novih in starih temp. / init. of vector of new␣
↪and old temp.
       Temp_vector_new = np.zeros(n)
       Temp_vector_old = Temperatures[-1,:]
       #inicializacija temp. v sosednjih točkah / init. of temp. in␣
↪neighbouring nodes
       TW0 = 0.
       TP0 = 0.
       TE0 = 0.
```

```
        #vpeljava robnih pogojev zahodno / introduction of west boundary␣
↪conditions

        if boundaryW == 'Dirichlet':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[0]
            TE0 = Temp_vector_old[1]
            #vrednosti koeficientov za zahodno točko / values of coefficients␣
↪for west node
            aP0 = roc * dx/dt
            aP = aP0
            aW = 0
            aE = k/dx
            SP = 0.
            Su = 2*k/dx * (TBW - TP0)
            #Računanje temperature v novem dt (zahodni rob) / Calculating␣
↪temperatures in new dt (west boundary)
            Temp_vector_new[0] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        if boundaryW == 'Neumann':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[0]
            TE0 = Temp_vector_old[1]
            #vrednosti koeficientov za zahodno točko / values of coefficients␣
↪for west node
            aP0 = roc * dx/dt
            aP = aP0
            aW = 0
            aE = k/dx
            SP = 0.
            Su = q
            #Računanje temperature v novem dt (zahodni rob) / Calculating␣
↪temperatures in new dt (west boundary)
            Temp_vector_new[0] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        if boundaryW == 'Robin':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[0]
            TE0 = Temp_vector_old[1]
            #vrednosti koeficientov za zahodno točko / values of coefficients␣
↪for west node
```

```
            aP0 = roc * dx/dt
            aP = aP0
            aW = 0
            aE = k/dx
            SP = -h/k
            Su = h/k * Tinf
            #Računanje temperature v novem dt (zahodni rob) / Calculating␣
↪temperatures in new dt (west boundary)
            Temp_vector_new[0] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        #vpeljava robnih pogojev vzhodno / introduction of east boundary␣
↪conditions

        if boundaryE == 'Dirichlet':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[-1]
            TW0 = Temp_vector_old[-2]
            #vrednosti koeficientov za vzhodno točko / values of coefficients␣
↪for east node
            aP0 = roc * dx/dt
            aP = aP0
            aW = k/dx
            aE = 0
            SP = 0.
            Su = 2*k/dx * (TBE - TP0)
            #Računanje temperature v novem dt (vzhodni rob) / Calculating␣
↪temperatures in new dt (west boundary)
            Temp_vector_new[-1] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        if boundaryE == 'Neumann':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[-1]
            TW0 = Temp_vector_old[-2]
            #vrednosti koeficientov za vzhodno točko / values of coefficients␣
↪for east node
            aP0 = roc * dx/dt
            aP = aP0
            aW = k/dx
            aE = 0
            SP = 0.
            Su = q
```

```
            #Računanje temperature v novem dt (zahodni rob) / Calculating␣
↪temperatures in new dt (west boundary)
            Temp_vector_new[-1] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        if boundaryE == 'Robin':
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TP0 = Temp_vector_old[-1]
            TW0 = Temp_vector_old[-2]
            #vrednosti koeficientov za vzhodno točko / values of coefficients␣
↪for east node
            aP0 = roc * dx/dt
            aP = aP0
            aW = k/dx
            aE = 0
            SP = -h/k
            Su = h/k * Tinf
            #Računanje temperature v novem dt (vzhodni rob) / Calculating␣
↪temperatures in new dt (east boundary)
            Temp_vector_new[-1] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0 + Su ) / aP

        #izračun temp. v notranjih točkah / calculation of temp. in inner nodes
        for i in range(1,n-1):
            #definicija starih temp. sosednjih točk / definition of old temp.␣
↪in neighbouring nodes
            TW0 = Temp_vector_old[i-1]
            TP0 = Temp_vector_old[i]
            TE0 = Temp_vector_old[i+1]
            #vrednosti koeficientov za notranje točke / values of coefficients␣
↪for inner nodes
            aP0 = roc * dx/dt
            aP = aP0
            aW = k/dx
            aE = k/dx
            SP = 0.
            #Računanje temperature v novem dt / Calculating temperatures in new␣
↪dt
            Temp_vector_new[i] = ( aW*TW0 + aE*TE0 + (aP0 - (aW + aE -SP)) *␣
↪TP0) / aP

        #zapis rezultatov v matriko rezultatov / writing results in results␣
↪matrix
        Temperatures = np.vstack((Temperatures, Temp_vector_new))
    print('Calculation completed.')
```

```
    return Temperatures
```

3.1.5 Uporaba funkcije / Use of function

Vnos snovnih lastnosti / Definition of material properties

[ ]:
```
L = 0.02 #[m]
k = 10 #[W/m.K]
roc = 10*10**6 #[J/m^3.K]
h = 15 #[W/m^2.K]
q = 0 #[W/m^2]
```

Definicija vhodnih parametrov, robnih pogojev, izračun in prikaz rezultata / Definition of input parameters, boundary conditions, calculation and display of results.

[ ]:
```
#uvoz potrebnih knjižnic / import of required libraries
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.widgets import Slider
%matplotlib notebook
```

[ ]:
```
#inicializacija vhodnih podatkov izračuna / initialisation of calculation input␣
 ↪parameters
n_slider=t_slider=dt_slider=w_bound=e_bound=T_init
TBW_slider=TBE_slider=q_slider=TB_inf=h_slider=0

#inicializacija widget-a za interaktivno izbiro št. delitev / initialisation␣
 ↪for interactive num. of division selection
n_slider = widgets.IntSlider(min=3, max = 100, step=1, value=5,␣
 ↪description='Num. of div. [/]')

dt_slider = widgets.FloatSlider(min=0.01, max = roc * ((L/(n_slider.value))**2)/
 ↪(2*k), step=0.01, value=2,
                                    description='Time step [s]')

def sync_dt_with_n(change):
    dt_max=roc * ((L/(n_slider.value))**2)/(2*k)
    dt_slider.max=2*dt_max
    dt_slider.step=dt_max/10
    if(dt_slider.value>2*dt_max):
        dt_slider.value=2*dt_max

n_slider.observe(sync_dt_with_n, names='value')
```

```
t_slider =  widgets.IntSlider(min=5, max = 300, step=1, value=20,␣
 ↪description='Max time [s]')

#inicializacija gumbov / init. of buttons
button1 = widgets.Button(
        description='Confirm',
        disabled = False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
        tooltip='Description',
        icon='check')

button2 = widgets.Button(
            description='Confirm',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
            icon='check')

button3 = widgets.Button(
            description='Run calculation',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
            icon='check')



T_init = widgets.IntSlider(min=-100, max=300, step=10,␣
 ↪value=200,description='Temp. init [°C]')

w_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Neumann',
                           description='West BC')
e_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Dirichlet',
                           description='East BC')

display(n_slider,  dt_slider, t_slider, T_init, w_bound, e_bound, button2)

def on_button_clicked(button):
    '''Aktivnost ob kliku na gumb / activity when button is clicked'''
    #klicanje globalnih spremenljivk / calling global variables
    global n_slider, t_slider, dt_slider, w_bound, e_bound, T_init, TBW_slider,␣
 ↪TBE_slider, q_slider, TB_inf, h_slider

    #definicija akcij za posamezen gumb / definition of actions for individual␣
 ↪button
```

```
    if button == button2:
        #branje novih vrednosti parametrov / reading new values
        TBW_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. west')
        TBE_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. east')
        q_slider = widgets.IntSlider(min=0, max=300, step=10, value=0,␣
↪description='q [W/m^2]')
        TB_inf = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=20,description='Temp. fluid surr.')
        h_slider = widgets.IntSlider(min=1, max=100, step=1, value =␣
↪15,description='h coeff.')

        if w_bound.value == 'Dirichlet':
            display(TBW_slider)
        if e_bound.value == 'Dirichlet':
            display(TBE_slider)
        if e_bound.value == 'Neumann' or w_bound.value == 'Neumann':
            display(q_slider)
        if e_bound.value == 'Robin' or w_bound.value == 'Robin':
            display(TB_inf,h_slider)
        display(button3)
        return

    if button == button3:
        #zagon izračun / calculation run
        print("Calculation started")
        Temperatures = explicit_diffusion1D(\
                    n=n_slider.value,\
                    t = t_slider.value,\
                    dt = dt_slider.value,\
                    L = L,\
                    k = k,\
                    Tinit = T_init.value,\
                    boundaryW = w_bound.value,\
                    boundaryE = e_bound.value, \
                    TBW = TBW_slider.value,\
                    TBE = TBE_slider.value,\
                    roc = roc,\
                    h = h_slider.value,\
                    Tinf = TB_inf.value,\
                    q = q_slider.value)

        #izris rezultatov izračuna / plotting the results
        %matplotlib notebook
```

```
        fig, ax = plt.subplots()
        ax.set_xlabel('x [m]')
        ax.set_ylabel('T [°C]')
        plt.ylim(min(Temperatures.flatten())-2,max(Temperatures.flatten())+2)
        plt.grid(b=True,which='both')

        #CReate slider axes
        ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
        ax_time.spines['top'].set_visible(True)
        ax_time.spines['right'].set_visible(True)

        # Create sliders
        time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)

        # Plot default data
        x = np.linspace(0, L, n_slider.value)
        default_y = Temperatures[0,:]
        line, = ax.plot(x,default_y,'o-')

        # Update values
        def update(val):
            i = time.val//dt_slider.value
            line.set_data(x,Temperatures[int(i),:])
            fig.canvas.draw_idle()
        time.on_changed(update)

#klicanje funkcije aktivnosti / calling function "when clicked"
button1.on_click(on_button_clicked)
button2.on_click(on_button_clicked)
button3.on_click(on_button_clicked)
```

3.1.6 Primerjava numerične rešitve z analitično / Comparison with analytical solution

Izračun temperaturnega polja za primerjavo / Calculation of temperature profile for comparison

Vnos števila delitev, časa simulacije in velikosti časovnega koraka / Input - number of division, simulation time and time step size

[ ]:
```
display(n_slider,  dt_slider, t_slider)
```

Ker analitična reštev velja za primer opisan v definiciji naloge v izračunu privzamemo naslednje vrednosti: /

Analytical solution is only valid for case described in definition, therefore we apply next values:

[ ]:
```
Temperatures_A = explicit_diffusion1D(\
                     n=n_slider.value,\
                     t=t_slider.value,\
                     dt=dt_slider.value,\
                     L = L,\
                     k = k,\
                     Tinit = 200,\
                     boundaryW = 'Neumann',\
                     boundaryE = 'Dirichlet', \
                     TBW = 0,\
                     TBE = 0,\
                     roc = roc,\
                     h = 0,\
                     Tinf = 0,\
                     q = 0)
```

Zapis analitične enačbe v kodi. / Implementation of analytical equation in code.

$\frac{4}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{2n-1}exp(-\alpha\lambda_n^2 t)cos(\lambda_n x)$, kjer je / where is $\lambda_n = \frac{(2n-1)\pi}{2L}$ in / and $\alpha = \frac{k}{\rho c}$

Opomba: upoštevamo prvih 100 členov zaporedja. / Note: first 100 terms of sequence are incorporated.

[ ]:
```
def analytical(i,x,L=L,dt=dt_slider.value):
    t = dt*i
    exp = 0
    a = k / roc
    for e in range(1,100):
        lam = (2*e-1)*np.pi/2/L
        exp += ((-1)**(e+1))/(2*e-1) * np.exp(-a*lam**2 * t) * np.cos(lam*x)
    return 4*200/np.pi * exp
```

Izris rešitve / Plotting the solution. Opomba: Analitična rešitev pri času $t = 0\,s$ ni dobro definirana. / Note: Analytical solution at time $t = 0\,s$ is not well defined.

[ ]:
```
fig, ax = plt.subplots()
ax.set_xlabel('x [m]')
ax.set_ylabel('T [°C]')
plt.ylim(min(Temperatures_A.flatten())-2,max(Temperatures_A.flatten())+2)
plt.grid(b=True,which='both')
#Create slider axes
ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
ax_time.spines['top'].set_visible(True)
ax_time.spines['right'].set_visible(True)

# Create sliders
```

```
time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
 ↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)

#Plot default data
x = np.linspace(0, L, n_slider.value)
default_y = Temperatures_A[0,:]
line, = ax.plot(x,default_y,'o-')
line1, = ax.plot(x,default_y,'r--')

#Update function
def fun1(i):
    x = np.linspace(0,L,100)
    Temp_vector = analytical(i,x)
    return x, Temp_vector

# Update values
def update(val):
    i = time.val//dt_slider.value
    line.set_data(x,Temperatures_A[int(i),:])
    analytical_data = fun1(i)
    line1.set_data(analytical_data[0],analytical_data[1])
    fig.canvas.draw_idle()
time.on_changed(update)
```

[ ]:
```
#Izpis tabele za verifikacijo izračuna s knjigo
print(explicit_diffusion1D(n=5,\
                      t=20,\
                      dt=2,\
                      L = L,\
                      k = k,\
                      Tinit = 200,\
                      boundaryW = 'Neumann',\
                      boundaryE = 'Dirichlet', \
                      TBW = 0,\
                      TBE = 0,\
                      roc = roc,\
                      h = 0,\
                      Tinf = 0,\
                      q = 0))
```

3.2 Implementacija implicitne metode na 1D difuzijskem problemu / Implementation of implicit method for 1D diffusion problem

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (245-256).

3.2.1 Definicija naloge / Problem definition

Tanka plošča ima na začetku temperaturo $200°C$ po celotnem volumnu. V določenem trenutku $t = 0\,s$ temperatura vzhodnega roba plošče pade na $0°C$. Vse ostale površine plošče so izolirane. Z uporabo implicitne metode s primernim časovnim korakom izračunaj časovni potek temperaturne razporeditve po plošči in jo primerjaj z analitično rešitvijo pri časih: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

Podatki:

- Debelina plošče: $L = 2cm$
- Toplotna prevodnost plošče: $k = 10\frac{W}{mK}$
- Volumska specifična toplotna kapaciteta: $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Prikaži še numerično rešitev za vse 3 vrste robnih pogojev (Dirichlet, Neumann, Robin) na vzhodnem oziroma zahodnem robu plošče. Pri tem upoštevaj konstantno temperaturo okoliške tekočine $T_\infty = 20°C$ in koeficient toplotne prestopnosti $h = 15\frac{W}{m^2 K}$.

A thin plate is initially at a uniform temperature of $200°C$. At a certain time $t = 0\,s$ the temperature of the east side of the plate is suddenly reduced to $0°C$. The other surface is insulated. Use the implicit finite volume method in conjunction with a suitable time step size to calculate the transient temperature distribution of the slab and compare it with the analytical solution at time: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

The data:

- Plate thickness: $L = 2cm$
- Thermal conductivity: $k = 10\frac{W}{mK}$
- Volume specific heat capacity $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Show the numerical solution for all 3 types of boundary conditions (Dirichlet, Neumann, Robin) on east or west boundary. Assume constant temperature of surrounding fluid $T_\infty = 20°C$ and coefficient of heat transfer $h = 15\frac{W}{m^2 K}$.

3.2.2 Osnove / Background

Vodilna enačba (1D prevod toplote v časovno neustaljenih pogojih): / Governing equation (unsteady 1D heat conduction):

$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}(k \frac{\partial T}{\partial x}) + S$

Implicitna diskretizaicija: / Implicit discretisation:

$a_P T_P = a_w T_W + a_E T_E + a_P^0 T_P^0 + S_u$

Kjer je: / Where is:

$a_P = a_P^0 + a_W + a_E - S_p$

$a_P^0 = \rho c \frac{\Delta x}{\Delta t}$

$a_W = \frac{k_w}{\delta x_{WP}}$

$a_E = \frac{k_e}{\delta x_{PE}}$

Izvorni člen: / Source term:

$b = S_u + \frac{1}{2} S_P T_p$

3.2.3 Robni pogoji: / Boundary conditions:

Dirichletov robni pogoj / Dirichlet boundary condition:

- Zahod /West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = \frac{2k_w}{\delta x_{WP}} T_B$
- $S_p = -\frac{2k_w}{\delta x_{WP}}$
- Vzhod /East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = \frac{2k_e}{\delta x_{PE}} T_B$
- $S_p = -\frac{2k_e}{\delta x_{PE}}$

Neumannov robni pogoj / Neumann boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = q_w$
- $S_p = 0$

- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = q_e$
- $S_p = 0$

Robinov robni pogoj / Robin boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_p = -\frac{h}{k_w}$
- $S_u = \frac{h}{k_w} T_\infty$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - Sp$
- $S_p = -\frac{h}{k_e}$
- $S_u = \frac{h}{k_e} T_\infty$

3.2.4 Implementacija metode v kodi / Implemetation of method in code:

[ ]:
```
#za lažje delo z matrikami uvozimo knjižnjico numpy / for easier work with␣
↪matrices numpy library is imported
import numpy as np
```

Ker implicitna metoda zahteva reševanje sistema enačb, najprej pripravimo poljuben solver tridiagonalnega sistema linearnih načb. Prikazana je uporaba Thomasovega algoritma. / Because implicit method requires solving system of linear equations, we first prepare any tridiagonal linear system equation solver. The solver presented below is based on Thomas algorithm.

[ ]:
```
def Thomas_solver(a, b, c, z):
    '''
```

```
    Thomas_solver solves tridiagonal system of linear eqation.

    a = underdiagonal (vector), b = diagonal (vector), c = upperdiagonal␣
↪(vector), d = vector of constants

    a b c d can be NumPy array type or Python list type.
    For more info refer to http://en.wikipedia.org/wiki/
↪Tridiagonal_matrix_algorithm
    '''

    m = len(b)
    U = [0 for k in range(0, m)]
    L = [0 for k in range(1, m)]
    U[0] = b[0]

    for k in range(0, m - 1):
        L[k] = a[k] / U[k]
        U[k + 1] = b[k + 1] - L[k] * c[k]

    y = [el for el in z]
    for k in range(1, m):
        y[k] = z[k] - L[k - 1] * y[k - 1]

    x = [el for el in y]
    x[m - 1] = x[m - 1] / U[m - 1]
    for k in range(m - 2, -1, -1):
        x[k] = (y[k] - c[k] * x[k + 1]) / U[k]
    return x
```

Implementacija metode / Method implementation

[ ]:
```
def implicit_diffusion1D(n, t, dt, L = 0.02, k = 10, Tinit = 200, boundaryW =␣
↪'Dirichlet', boundaryE = 'Dirichlet',  \
                         TBW = 0, TBE = 0, roc = 10*10**6, h = 15, Tinf = 20, q␣
↪= 0):
    """Funkcija omogoča reševanje 1D difuzije toplote v časovno neustaljenem␣
↪stanju, brez izvornega člena, z implicitno metodo.

    Dovoljeni robni pogoji (vrstni red ni pomemben): / Allowed boundary␣
↪conditions (order is not important):
    Dirichlet-Dirichlet, Dirichlet-Neumann, Dirichlet-Robin, Robin-Neumann

    Vhodni podatki / input data:
    št. delitev / number of division -> n [/]
    čas simulacije / total time of simulation -> t [s]
    časovni korak / time step -> dt [s]
    Dolžina domene / domain legth -> L[m]
```

```
   Toplotna prevodnost / thermal conductivity -> k [W/m.K]
   Začetna temperatura / initial temperature -> Tinit [°C]
   Tipa robnih pogojev / types of boundary condition boundaryW in/and boundaryE
   Temperatura zahodnega robu (RP) / West boundary temp. (BC) -> TBW [°C]
   Temperatura vzhodnega robu (RP) / East boundary temp. (BC) -> TBE [°C]
   Volumsko specifična termičnaa masa / Volume specific thermal mas -> roc [J/
↪m^3.K]
   Toplotna prestopnost (RP) / coefficient of heat transfer (BC) -> h [W/m^2.K]
   Temperatura okoliške tekočine (RP) / Sorrounding fluid temp. (BC) -> Tinf␣
↪[°C]
   Toplotni tok na robu (RP) / Heat flux (boundary cindition) (BC) -> q[W/m^2]

   Rezultat funkcije je numpy numerično polje (vrstica - dt, stolpec - dx). /
   Result of function is numpy array (row - dt, column -dx).

   """

   #velikost prostorske diskretizacije (dx) / spatial discretisation size (dx)
   dx = L / n

   #inicializacija / initialisation
   Temperatures = np.asarray([]) #inicializacija matrike rezultatov (temp. v␣
↪časovnem koraku) / init. of results matrix (temp. in time step)
   Temp_int = Tinit * np.ones((1,n)) #vrstični vektor začetnih temperatur␣
↪(t=0)/ initial temperature row vector (t=0)
   Temperatures = Temp_int #dodajanje Tinit v matriko rez. / adding Tinit to␣
↪results matrix
   tc = 0. #inic. časa simulacije / init. of time of simulation


   #zanka za "časovno korakanje" naprej do max. časa sim. t / loop for forward␣
↪"time marching" up to max. simulation time t
   while tc < t:
       tc += dt #izračun pretečenega časa v trenutnem časovnem koraku /␣
↪calculation of elapsed time in current time step

       #inicializacija vektorja novih in starih temp. / init. of vector of new␣
↪and old temp.
       Temp_vector_new = np.zeros(n)
       Temp_vector_old = Temperatures[-1,:]

       #inicializacija koeficientov pod diagonalo, na diagonali, nad diagonalo␣
↪v sosednjih točkah /
       #init. of coefficients under diagonal, on diagonal and upper diagonal
       under_diag = np.zeros(n-1)
       diag = np.zeros(n)
```

```
        upper_diag = np.zeros(n-1)
        constants = np.zeros(n)

        #vpeljava robnih pogojev zahodno / introduction of west boundary␣
↪conditions

        if boundaryW == 'Dirichlet':
            #vrednosti koeficientov za zahodno točko / values of coefficients␣
↪for west node
            aP0 = roc * dx/dt
            aW = 0
            aE = k/dx
            SP = -2*k/dx
            Su = 2*k/dx * TBW
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -aE
            constants[0] = aP0*Temp_vector_old[0]+Su


        if boundaryW == 'Neumann':
            aP0 = roc * dx/dt
            aW = 0
            aE = k/dx
            SP = 0
            Su = q
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -aE
            constants[0] = aP0*Temp_vector_old[0]+Su


        if boundaryW == 'Robin':
            aP0 = roc * dx/dt
            aW = 0
            aE = k/dx
            SP = -h/k
            Su = h/k * Tinf
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -aE
```

```
            constants[0] = aP0*Temp_vector_old[0]+Su


        #vpeljava robnih pogojev vzhodno / introduction of east boundary␣
↪conditions

        if boundaryE == 'Dirichlet':
            ##definicija starih temp. v točki / definition of old temp. in node
            TP0 = Temp_vector_old[0]
            #vrednosti koeficientov za vzhodno točko / values of coefficients␣
↪for east node
            aP0 = roc * dx/dt
            aW = k/dx
            aE = 0
            SP = -2*k/dx
            Su = 2*k/dx * TBE
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -aW
            constants[-1] = aP0*Temp_vector_old[-1]+Su

        if boundaryE == 'Neumann':
            aP0 = roc * dx/dt
            aW = k/dx
            aE = 0
            SP = 0
            Su = q
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -aW
            constants[-1] = aP0*Temp_vector_old[-1]+Su

        if boundaryE == 'Robin':
            aP0 = roc * dx/dt
            aW = k/dx
            aE = 0
            SP = -h/k
            Su = h/k * Tinf
            aP = aP0 + aW + aE - SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -aW
```

```
            constants[-1] = aP0*Temp_vector_old[-1]+Su

        #Priprava koeficientov in konstant - notranje točke / Calculating␣
↪coefficients and constants - inner nodes
        for i in range(1,n-1):
            #vrednosti koeficientov za notranje točke / values of coefficients␣
↪for inner nodes
            aP0 = roc * dx/dt
            aW = k/dx
            aE = k/dx
            SP = 0
            Su = 0
            aP = aP0 + aW + aE - SP
            #Računanje temperature v novem dt / Calculating temperatures in new␣
↪dt
            diag[i] = aP
            under_diag[i-1] = -aW
            upper_diag[i] = -aE
            constants[i] = aP0*Temp_vector_old[i]+Su

        #Reševanje 3-diagonalnega sistema LE / Solving 3-diagonal system of LE
        Temp_vector_new = Thomas_solver(a=under_diag, b=diag, c=upper_diag,␣
↪z=constants)

        #zapis rezultatov v matriko rezultatov / writing results in results␣
↪matrix
        Temperatures = np.vstack((Temperatures, Temp_vector_new))
    print('Calculation completed.')
    return Temperatures
```

3.2.5 Uporaba funkcije / Use of function

Definicija snovnih lastnosti / Definition of material properties

[ ]:
```
L = 0.02 #[m]
k = 10 #[W/m.K]
roc = 10*10**6 #[J/m^3.K]
h = 15 #[W/m^2.K]
q = 0 #[W/m^2]
```

Definicija vhodnih parametrov, robnih pogojev, izračun in prikaz rezultata / Definition of input parameters, boundary conditions, calculation and display of results.

[ ]:
```
#uvoz potrebnih knjižnic / import of needed libraries
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
```

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.widgets import Slider
%matplotlib notebook
```

[ ]:
```
#inicializacija vhodnih podatkov izračuna / initialisation of calculation input␣
 ↪parameters
n_slider=t_slider=dt_slider=w_bound=e_bound=T_init
TBW_slider=TBE_slider=q_slider=TB_inf=h_slider=0

#inicializacija widget-a za interaktivno izbiro št. delitev / initialisation␣
 ↪for interactive num. of division selection
n_slider = widgets.IntSlider(min=3, max = 100, step=1, value=5,␣
 ↪description='Num. of div. [/]')

dt_slider = widgets.FloatSlider(min=0.01, max = 5, step=0.01, value=2,
                                    description='Time step [s]')

def sync_dt_with_n(change):
    return


n_slider.observe(sync_dt_with_n, names='value')

t_slider =  widgets.IntSlider(min=5, max = 300, step=1, value=20,␣
 ↪description='Max time [s]')

#inicializacija gumbov / init. of buttons
button1 = widgets.Button(
        description='Confirm',
        disabled = False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
        tooltip='Description',
        icon='check')

button2 = widgets.Button(
            description='Confirm',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
            icon='check')

button3 = widgets.Button(
            description='Run calculation',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
```

```
            icon='check')



T_init = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=200,description='Temp. init [°C]')

w_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Neumann',
                           description='West BC')
e_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Dirichlet',
                           description='East BC')

display(n_slider,  dt_slider, t_slider, T_init, w_bound, e_bound, button2)

def on_button_clicked(button):
    '''Aktivnost ob kliku na gumb / activity when button is clicked'''
    #klicanje globalnih spremenljivk / calling global variables
    global n_slider, t_slider, dt_slider, w_bound, e_bound, T_init, TBW_slider,␣
↪TBE_slider, q_slider, TB_inf, h_slider

    #definicija akcij za posamezen gumb / definition of actions for individual␣
↪button

    if button == button2:
        #branje novih vrednosti parametrov / reading new values
        TBW_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. west')
        TBE_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. east')
        q_slider = widgets.IntSlider(min=0, max=300, step=10, value=0,␣
↪description='q [W/m^2]')
        TB_inf = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=20,description='Temp. fluid surr.')
        h_slider = widgets.IntSlider(min=1, max=100, step=1, value =␣
↪15,description='h coeff.')

        if w_bound.value == 'Dirichlet':
            display(TBW_slider)
        if e_bound.value == 'Dirichlet':
            display(TBE_slider)
        if e_bound.value == 'Neumann' or w_bound.value == 'Neumann':
            display(q_slider)
        if e_bound.value == 'Robin' or w_bound.value == 'Robin':
            display(TB_inf,h_slider)
```

```
        display(button3)
        return

    if button == button3:
        #zagon izračun / calculation run
        print("Calculation started")
        Temperatures = implicit_diffusion1D(\
                    n=n_slider.value,\
                    t = t_slider.value,\
                    dt = dt_slider.value,\
                    L = L,\
                    k = k,\
                    Tinit = T_init.value,\
                    boundaryW = w_bound.value,\
                    boundaryE = e_bound.value, \
                    TBW = TBW_slider.value,\
                    TBE = TBE_slider.value,\
                    roc = roc,\
                    h = h_slider.value,\
                    Tinf = TB_inf.value,\
                    q = q_slider.value)

        #izris rezultatov izračuna / plotting the results
        %matplotlib notebook

        fig, ax = plt.subplots()
        ax.set_xlabel('x [m]')
        ax.set_ylabel('T [°C]')
        plt.ylim(min(Temperatures.flatten())-2,max(Temperatures.flatten())+2)
        plt.grid(b=True,which='both')

        #CReate slider axes
        ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
        ax_time.spines['top'].set_visible(True)
        ax_time.spines['right'].set_visible(True)

        # Create sliders
        time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)

        # Plot default data
        x = np.linspace(0, L, n_slider.value)
        default_y = Temperatures[0,:]
        line, = ax.plot(x,default_y,'o-')

        # Update values
```

```
        def update(val):
            i = time.val//dt_slider.value
            line.set_data(x,Temperatures[int(i),:])
            fig.canvas.draw_idle()
        time.on_changed(update)

#klicanje funkcije aktivnosti / calling function "when clicked"
button1.on_click(on_button_clicked)
button2.on_click(on_button_clicked)
button3.on_click(on_button_clicked)
```

3.2.6 Primerjava numerične rešitve z analitično / Comparison with analytical solution

Izračun temperaturnega polja za primerjavo / Calculation of temperature profile for comparison

Vnos števila delitev, časa simulacije in velikosti časovnega koraka / Input - number of division, simulation time and time step size

[ ]:
```
display(n_slider,  dt_slider, t_slider)
```

Ker analitična reštev velja za primer opisan v definiciji naloge v izračunu privzamemo naslednje vrednosti: /

Analytical solution is only valid for case described in definition, therefore we apply next values:

[ ]:
```
Temperatures_A = implicit_diffusion1D(\
                    n=n_slider.value,\
                    t=200,\
                    dt=dt_slider.value,\
                    L = L,\
                    k = k,\
                    Tinit = 200,\
                    boundaryW = 'Neumann',\
                    boundaryE = 'Dirichlet', \
                    TBW = 0,\
                    TBE = 0,\
                    roc = roc,\
                    h = 0,\
                    Tinf = 0,\
                    q = 0)
```

Zapis analitične enačbe v kodi. / Implementation of analytical equation in code.

$\frac{4}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{2n-1}exp(-\alpha\lambda_n^2 t)cos(\lambda_n x)$, kjer je / where is $\lambda_n = \frac{(2n-1)\pi}{2L}$ in / and $\alpha = \frac{k}{\rho c}$

Opomba: upoštevamo prvih 100 členov zaporedja. Note: first 100 terms of sequence are incorporated

[ ]:
```
def analytical(i,x,L=L,dt=dt_slider.value):
    t = dt*i
    exp = 0
    a = k / roc
    for e in range(1,100):
        lam = (2*e-1)*np.pi/2/L
        exp += ((-1)**(e+1))/(2*e-1) * np.exp(-a*lam**2 * t) * np.cos(lam*x)
    return 4*200/np.pi * exp
```

Izris rešitve / Plotting the solution. Opomba: Analitična rešitev pri času $t = 0\,s$ ni dobro definirana. / Note: Analytical solution at time $t = 0\,s$ is not well defined.

[ ]:
```
fig, ax = plt.subplots()
ax.set_xlabel('x [m]')
ax.set_ylabel('T [°C]')
plt.ylim(min(Temperatures_A.flatten())-2,max(Temperatures_A.flatten())+2)
plt.grid(b=True,which='both')
#Create slider axes
ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
ax_time.spines['top'].set_visible(True)
ax_time.spines['right'].set_visible(True)

# Create sliders
time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
 ↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)

#Plot default data
x = np.linspace(0, L, n_slider.value)
default_y = Temperatures_A[0,:]
line, = ax.plot(x,default_y,'o-')
line1, = ax.plot(x,default_y,'r--')

#Update function
def fun1(i):
    x = np.linspace(0,L,100)
    Temp_vector = analytical(i,x)
    return x, Temp_vector

# Update values
def update(val):
    i = time.val//dt_slider.value
    line.set_data(x,Temperatures_A[int(i),:])
    analytical_data = fun1(i)
    line1.set_data(analytical_data[0],analytical_data[1])
    fig.canvas.draw_idle()
time.on_changed(update)
```

[ ]:
```
#Izpis tabele za verifikacijo izračuna s knjigo
izpis=implicit_diffusion1D(n=5,\
                      t=120,\
                      dt=2,\
                      L = L,\
                      k = k,\
                      Tinit = 200,\
                      boundaryW = 'Neumann',\
                      boundaryE = 'Dirichlet', \
                      TBW = 0,\
                      TBE = 0,\
                      roc = roc,\
                      h = 0,\
                      Tinf = 0,\
                      q = 0)
print('time step = 40s\n',izpis[40//2,:])
print('time step = 80s\n',izpis[80//2,:])
print('time step = 120s\n',izpis[120//2,:])
```

3.3 Implementacija Crank-Nicolsonove metode na 1D difuzijskem problemu / Implementation of Crank-Nicolson method on 1D diffusion problem

Povzeto po primeru: / Adapted from the case: H.K. Versteeg, M.L. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Second Edition, Prentice Hall, Harlow, 2007 (245-256).

3.3.1 Definicija naloge / Problem definition

Tanka plošča ima na začetku temperaturo $200°C$ po celotnem volumnu. V določenem trenutku $t = 0\,s$ temperatura vzhodnega robu plošče pade na $0°C$. Vse ostale površine plošče so izolirane. Z uporabo Crank-Nicolsonove metode s primernim časovnim korakom izračunaj časovni potek temperaturne razporeditve po plošči in jo primerjaj z analitično rešitvijo pri časih: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

Preračunaj numerično rešitev, če uporabimo kritični časovni korak: $\Delta t = \rho c\frac{(\Delta x)^2}{k} (= 16\,s)$ za čas $t = 40\,s$ in primerjaj rezultat z analitično rešitvijo.

Podatki:

- Debelina plošče: $L = 2cm$
- Toplotna prevodnost plošče: $k = 10\frac{W}{mK}$
- Volumsko specifična termična masa $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Prikaži še numerično rešitev za vse 3 vrste robnih pogojev (Dirichlet, Neumann, Robin) na vzhodnem oziroma zahodnem robu plošče. Pri tem upoštevaj konstantno temperaturo okoliške tekočine $T_\infty = 20°C$ in koeficient toplotne prestopnosti $h = 15\frac{W}{m^2 K}$.

A thin plate is initially at a uniform temperature of $200°C$. At a certain time $t = 0\,s$ the temperature of the east side of the plate is suddenly reduced to $0°C$. The other surface is insulated. Use the Crank-Nicolson finite volume method in conjunction with a suitable time step size to calculate the transient temperature distribution of the slab and compare it with the analytical solution at time: $t_1 = 40\,s, t_2 = 80\,s, t_3 = 120\,s$

Recalculate the numerical solution using a time step size equal to the limit $\Delta t = \rho c \frac{(\Delta x)^2}{k} (= 16\,s)$ for $t = 40\,s$ and compare the results with the analytical solution.

The data:

- Plate thickness: $L = 2cm$
- Thermal conductivity: $k = 10 \frac{W}{mK}$
- Volume specific thermal mass $\rho c = 10 \cdot 10^6 \frac{J}{m^3 K}$

Show numerical solution for all 3 types of boundary conditions (Dirichlet, Neumann, Robin) on east or west boundary. Assume constant temperature of surrounding fluid $T_\infty = 20°C$ and coefficient of heat transfer $h = 15 \frac{W}{m^2 K}$.

3.3.2 Osnove / Basis

Vodilna enačba za 1D prevod toplote v časovno neustaljenih pogojih: / Govering equation for unsteady 1D heat conduction:

$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}(k \frac{\partial T}{\partial x}) + S$

Crank-Nicolsonova diskretizaicija: / Crank-Nicolson discretisation:

$a_P T_P = a_W [\frac{T_W + T_W^0}{2}] + a_E [\frac{T_E + T_E^0}{2}] + [a_P^0 - \frac{a_E}{2} - \frac{a_W}{2}] T_P^0 + S_u + \frac{1}{2} S_P T_P^0$

Kjer je: / Where is:

$a_P = \frac{1}{2}(a_W + a_E) + a_P^0 - \frac{1}{2} S_p$

$a_P^0 = \rho c \frac{\Delta x}{\Delta t}$

$a_W = \frac{k_w}{\delta x_{WP}}$

$a_E = \frac{k_e}{\delta x_{PE}}$

Izvorni člen: / Source term:

$b = S_u + \frac{1}{2} S_P T_p + \frac{1}{2} S_P T_p^0$

3.3.3 Robni pogoji: / Boundary conditions:

Dirichletov robni pogoj: / Dirichlet boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$

- $S_u = \frac{2k_w}{\delta x_{WP}} T_B$
- $S_p = -\frac{2k_w}{\delta x_{WP}}$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = \frac{2k_e}{\delta x_{PE}} T_B$
- $S_p = -\frac{2k_e}{\delta x_{PE}}$

Neumannov robni pogoj / Neumann boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = q_w$
- $S_p = 0$
- Vzhod /East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - S_p$
- $S_u = q_e$
- $S_p = 0$

Robinov robni pogoj / Robin boundary condition:

- Zahod / West:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = 0$
- $a_E = \frac{k_e}{\delta x_{PE}}$
- $a_P = a_P^0 + a_w + a_E - S_p$

- $S_p = -\frac{h}{k_w}$
- $S_u = \frac{h}{k_w} T_\infty$
- Vzhod / East:
- $a_P^0 = \rho c \frac{\Delta x}{\Delta t}$
- $a_W = \frac{k_w}{\delta x_{WP}}$
- $a_E = 0$
- $a_P = a_P^0 + a_w + a_E - Sp$
- $S_p = -\frac{h}{k_e}$
- $S_u = \frac{h}{k_e} T_\infty$

Implementacija metode v kodi / Implemetation of method in code:

[ ]:
```
#za lažje delo z matrikami uvozimo knjižnjico numpy / for easier work with␣
↪matrices numpy library is imported
import numpy as np
```

Ker Crank-Nicolsonova metoda zahteva reševanje sistema enačb najprej pripravimo poljuben solver trodiagonalnega sistema linearnih načb. Prikazana je uporaba Thomasovega algoritma. / Because Crank-Nicolson method requires solving system of linear equations, we first prepare any tridiagonal linear system equation solver. The solver presented below is based on Thomas algorithm.

[ ]:
```
def Thomas_solver(a, b, c, z):
    '''
    Thomas_solver solves tridiagonal system of linear eqation.

    a = underdiagonal (vector), b = diagonal (vector), c = upperdiagonal␣
↪(vector), d = vector of constants

    a b c d can be NumPy array type or Python list type.
    For more info refer to http://en.wikipedia.org/wiki/
↪Tridiagonal_matrix_algorithm
    '''

    m = len(b)
    U = [0 for k in range(0, m)]
    L = [0 for k in range(1, m)]
    U[0] = b[0]

    for k in range(0, m - 1):
        L[k] = a[k] / U[k]
        U[k + 1] = b[k + 1] - L[k] * c[k]

    y = [el for el in z]
    for k in range(1, m):
```

```
        y[k] = z[k] - L[k - 1] * y[k - 1]

    x = [el for el in y]
    x[m - 1] = x[m - 1] / U[m - 1]
    for k in range(m - 2, -1, -1):
        x[k] = (y[k] - c[k] * x[k + 1]) / U[k]
    return x
```

Implementacija metode / Method implementation

[ ]:
```
def CrankNicolson_diffusion1D(n, t, dt, L = 0.02, k = 10, Tinit = 200,␣
↪boundaryW = 'Dirichlet', boundaryE = 'Dirichlet',  \
                        TBW = 0, TBE = 0, roc = 10*10**6, h = 15, Tinf = 20, q␣
↪= 0):
    """Funkcija omogoča reševanje 1D difuzije toplote v časovno neustaljenem␣
↪stanju, brez izvornega člena, z Crank-Nicholsonovo metodo.

    Dovoljeni robni pogoji (vrstni red ni pomemben): / Allowed boundary␣
↪conditions (order is not important):
    Dirichlet-Dirichlet, Dirichlet-Neumann, Dirichlet-Robin, Robin-Neumann

    Vhodni podatki / input data:
    št. delitev / number of division -> n [/]
    čas simulacije / total time of simulation -> t [s]
    časovni korak / time step -> dt [s], privzeta vrednost / default value dt =␣
↪rho * cp * dx^2/k
    Dolžina domene / domain legth -> L[m]
    Toplotna prevodnost / thermal conductivity -> k [W/m.K]
    Začetna temperatura / initial temperature -> Tinit [°C]
    Tipa robnih pogojev / types of boundary condition boundaryW in/and boundaryE
    Temperatura zahodnega robu (RP) / West boundary temp. (BC) -> TBW [°C]
    Temperatura vzhodnega robu (RP) / East boundary temp. (BC) -> TBE [°C]
    Volumsko specifična termičnaa masa / Volume specific thermal mas -> roc [J/
↪m^3.K]
    Toplotna prestopnost (RP) / coefficient of heat transfer (BC) -> h [W/m^2.K]
    Temperatura okoliške tekočine (RP) / Sorrounding fluid temp. (BC) -> Tinf␣
↪[°C]
    Toplotni tok na robu (RP) / Heat flux (boundary cindition) (BC) -> q[W/m^2]

    Rezultat funkcije je numpy numerično polje (vrstica - dt, stolpec - dx). /
    Result of function is numpy array (row - dt, column -dx).

    """

     #velikost prostorske diskretizacije (dx) / spatial discretisation size (dx)
    dx = L / n
```

```
   #določanje privzetega časovnega koraka (največji dovoljen) / init. of default time step (max allowed)
   if dt == 'Auto':
       dt = roc * x**2/(k)


   #inicializacija / initialisation
   Temperatures = np.asarray([]) #inicializacija matrike rezultatov (temp. v časovnem koraku) / init. of results matrix (temp. in time step)
   Temp_int = Tinit * np.ones((1,n)) #vrstični vektor začetnih temperatur (t=0)/ initital temperature row vector (t=0)
   Temperatures = Temp_int #dodajanje Tinit v matriko rez. / adding Tinit to results matrix
   tc = 0. #inic. časa simulacije / init. of time of simulation


   #zanka za "časovno korakanje" naprej do max. časa sim. t / loop for forward "time marching" up to max. simulation time t
   while tc < t:
       tc += dt #izračun pretečenega časa v trenutnem časovnem koraku / calculation of elapsed time in current time step

       #inicializacija vektorja novih in starih temp. / init. of vector of new and old temp.
       Temp_vector_new = np.zeros(n)
       Temp_vector_old = Temperatures[-1,:]

       #inicializacija koeficientov pod diagonalo, na diagonali, nad diagonalo v sosednjih točkah /
       #init. of coefficients under diagonal, on diagonal and upper diagonal
       under_diag = np.zeros(n-1)
       diag = np.zeros(n)
       upper_diag = np.zeros(n-1)
       constants = np.zeros(n)

       #vpeljava robnih pogojev zahodno / introduction of west boundary conditions

       if boundaryW == 'Dirichlet':
           #vrednosti koeficientov za zahodno točko / values of coefficients for west node
           aP0 = roc * dx/dt
           aW = 0
           aE = k/dx
           SP = -2*k/dx
           Su = 2*k/dx * TBW
```

```
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -0.5*aE
            constants[0] =(aP0-0.5*aE+0.5*SP)*Temp_vector_old[0]+ 0.
↪5*aE*Temp_vector_old[1]+Su


        if boundaryW == 'Neumann':
            aP0 = roc * dx/dt
            aW = 0
            aE = k/dx
            SP = 0
            Su = q
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -0.5*aE
            constants[0] = (aP0-0.5*aE+0.5*SP)*Temp_vector_old[0]+ 0.
↪5*aE*Temp_vector_old[1]+Su


        if boundaryW == 'Robin':
            aP0 = roc * dx/dt
            aW = 0
            aE = k/dx
            SP = -h/k
            Su = h/k * Tinf
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na zahodnem robu / Calculating␣
↪coefficients and constants on west boundary
            diag[0] = aP
            upper_diag[0] = -0.5*aE
            constants[0] = (aP0-0.5*aE+0.5*SP)*Temp_vector_old[0]+ 0.
↪5*aE*Temp_vector_old[1]+Su


        #vpeljava robnih pogojev vzhodno / introduction of east boundary␣
↪conditions

        if boundaryE == 'Dirichlet':
            #vrednosti koeficientov za vzhodno točko / values of coefficients␣
↪for east node
            aP0 = roc * dx/dt
```

```
            aW = k/dx
            aE = 0
            SP = -2*k/dx
            Su = 2*k/dx * TBE
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -0.5*aW
            constants[-1] = (aP0-0.5*aW+0.5*SP)*Temp_vector_old[-1]+ 0.
↪5*aW*Temp_vector_old[-2]+Su

        if boundaryE == 'Neumann':
            aP0 = roc * dx/dt
            aW = k/dx
            aE = 0
            SP = 0
            Su = q
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -0.5*aW
            constants[-1] = (aP0-0.5*aW+0.5*SP)*Temp_vector_old[-1]+ 0.
↪5*aW*Temp_vector_old[-2]+Su

        if boundaryE == 'Robin':
            aP0 = roc * dx/dt
            aW = k/dx
            aE = 0
            SP = -h/k
            Su = h/k * Tinf
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Priprava koeficientov in konstant na vzhodnem robu / Calculating␣
↪coefficients and constants on east boundary
            diag[-1] = aP
            under_diag[-1] = -0.5*aW
            constants[-1] = (aP0-0.5*aW+0.5*SP)*Temp_vector_old[-1]+ 0.
↪5*aW*Temp_vector_old[-2]+Su

        #Priprava koeficientov in konstant - notranje točke / Calculating␣
↪coefficients and constants - inner nodes
        for i in range(1,n-1):
            #vrednosti koeficientov za notranje točke / values of coefficients␣
↪for inner nodes
            aP0 = roc * dx/dt
```

```
            aW = k/dx
            aE = k/dx
            SP = 0
            Su = 0
            aP = 0.5*(aW + aE) + aP0 -0.5*SP
            #Računanje temperature v novem dt / Calculating temperatures in new␣
↪dt
            diag[i] = aP
            under_diag[i-1] = -0.5 * aW
            upper_diag[i] = -0.5 * aE
            constants[i] = (aP0-0.5*(aW+aE)+0.5*SP)*Temp_vector_old[i]+ 0.
↪5*aW*Temp_vector_old[i-1] +0.5*aE*Temp_vector_old[i+1] +Su

        #Reševanje 3-diagonalnega sistema LE / Solving 3-diagonal system of LE
        Temp_vector_new = Thomas_solver(a=under_diag, b=diag, c=upper_diag,␣
↪z=constants)

        Matrix=np.zeros((n,n))
        Matrix[0,0]=diag[0]
        Matrix[0,1]=upper_diag[0]
        Matrix[-1,-1]=diag[0]
        Matrix[-1,-2]=under_diag[-1]
        for i in range(1,n-1):
            Matrix[i,i]=diag[i]
            Matrix[i,i-1]=under_diag[i-1]
            Matrix[i,i+1]=upper_diag[i]
        #print(Matrix)
        #print(constants)
        #zapis rezultatov v matriko rezultatov / writing results in results␣
↪matrix
        Temperatures = np.vstack((Temperatures, Temp_vector_new))
    print('Calculation completed.')
    return Temperatures
```

Uporaba funkcije / Use of function

Definicija snovnih lastnosti / Definition of material properties

[ ]:
```
L = 0.02 #[m]
k = 10 #[W/m.K]
roc = 10*10**6 #[J/m^3.K]
h = 15 #[W/m^2.K]
q = 0 #[W/m^2]
```

Definicija vhodnih parametrov, robnih pogojev, izračun in prikaz rezultata /

Definition of input parameters, boundary conditions, calculation and display of results.

[ ]:
```
#uvoz potrebnih knjižnic / import of needed libraries
import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.widgets import Slider
%matplotlib notebook
```

[ ]:
```
#inicializacija vhodnih podatkov izračuna / initialisation of calculation input␣
 ↪parameters
n_slider=t_slider=dt_slider=w_bound=e_bound=T_init
TBW_slider=TBE_slider=q_slider=TB_inf=h_slider=0

#inicializacija widget-a za interaktivno izbiro št. delitev / initialisation␣
 ↪for interactive num. of division selection
n_slider = widgets.IntSlider(min=3, max = 100, step=1, value=5,␣
 ↪description='Num. of div. [/]')

dt_slider = widgets.FloatSlider(min=0.01, max = 5, step=0.01, value=2,
                                    description='Time step [s]')

def sync_dt_with_n(change):
    dt_max=roc*((L/(n_slider.value))**2)/k
    dt_slider.max=2*dt_max
    dt_slider.step=dt_max/10
    if(dt_slider.value>2*dt_max):
        dt_slider.value=2*dt_max
    return

n_slider.observe(sync_dt_with_n, names='value')

t_slider =  widgets.IntSlider(min=5, max = 300, step=1, value=20,␣
 ↪description='Max time [s]')

#inicializacija gumbov / init. of buttons
button1 = widgets.Button(
        description='Confirm',
        disabled = False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
        tooltip='Description',
        icon='check')

button2 = widgets.Button(
            description='Confirm',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
```

```
            icon='check')

button3 = widgets.Button(
            description='Run calculation',
            disabled=False,
            button_style='', # 'success', 'info', 'warning', 'danger' or ''
            tooltip='Description',
            icon='check')



T_init = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=200,description='Temp. init [°C]')

w_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Neumann',
                           description='West BC')
e_bound = widgets.Dropdown(options=['Dirichlet','Neumann', 'Robin'],
                           value='Dirichlet',
                           description='East BC')

display(n_slider,  dt_slider, t_slider, T_init, w_bound, e_bound, button2)

def on_button_clicked(button):
    '''Aktivnost ob kliku na gumb / activity when button is clicked'''
    #klicanje globalnih spremenljivk / calling global variables
    global n_slider, t_slider, dt_slider, w_bound, e_bound, T_init, TBW_slider,␣
↪TBE_slider, q_slider, TB_inf, h_slider

    #definicija akcij za posamezen gumb / definition of actions for individual␣
↪button

    if button == button2:
        #branje novih vrednosti parametrov / reading new values
        TBW_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. west')
        TBE_slider = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=0,description='Temp. east')
        q_slider = widgets.IntSlider(min=0, max=300, step=10, value=0,␣
↪description='q [W/m^2]')
        TB_inf = widgets.IntSlider(min=-100, max=300, step=10,␣
↪value=20,description='Temp. fluid surr.')
        h_slider = widgets.IntSlider(min=1, max=100, step=1, value =␣
↪15,description='h coeff.')

        if w_bound.value == 'Dirichlet':
```

```
            display(TBW_slider)
        if e_bound.value == 'Dirichlet':
            display(TBE_slider)
        if e_bound.value == 'Neumann' or w_bound.value == 'Neumann':
            display(q_slider)
        if e_bound.value == 'Robin' or w_bound.value == 'Robin':
            display(TB_inf,h_slider)
        display(button3)
        return

    if button == button3:
        #zagon izračun / calculation run
        print("Calculation started")
        Temperatures = CrankNicolson_diffusion1D(\
                    n=n_slider.value,\
                    t = t_slider.value,\
                    dt = dt_slider.value,\
                    L = L,\
                    k = k,\
                    Tinit = T_init.value,\
                    boundaryW = w_bound.value,\
                    boundaryE = e_bound.value, \
                    TBW = TBW_slider.value,\
                    TBE = TBE_slider.value,\
                    roc = roc,\
                    h = h_slider.value,\
                    Tinf = TB_inf.value,\
                    q = q_slider.value)

        #izris rezultatov izračuna / plotting the results
        %matplotlib notebook

        fig, ax = plt.subplots()
        ax.set_xlabel('x [m]')
        ax.set_ylabel('T [°C]')
        plt.ylim(min(Temperatures.flatten())-2,max(Temperatures.flatten())+2)
        plt.grid(b=True,which='both')

        #Create slider axes
        ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
        ax_time.spines['top'].set_visible(True)
        ax_time.spines['right'].set_visible(True)

        # Create sliders
        time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)
```

```
        # Plot default data
        x = np.linspace(0, L, n_slider.value)
        default_y = Temperatures[0,:]
        line, = ax.plot(x,default_y,'o-')

        # Update values
        def update(val):
            i = time.val//dt_slider.value
            line.set_data(x,Temperatures[int(i),:])
            fig.canvas.draw_idle()
        time.on_changed(update)

#klicanje funkcije aktivnosti / calling function "when clicked"
button1.on_click(on_button_clicked)
button2.on_click(on_button_clicked)
button3.on_click(on_button_clicked)
```

Primerjava numerične rešitve z analitično / Comparison with analytical solution

Izračun temperaturnega polja za primerjavo / Calculation of temperature profile fro comparison

Vnos števila delitev, časa simulacije in velikosti časovnega koraka / Input - number of division, simulation time and time step size

[ ]:
```
display(n_slider,  dt_slider, t_slider)
```

Ker analitična reštev velja za primer opisan v definiciji naloge v izračunu privzamemo naslednje vrednosti: /

Analytical solution is only valid for the case described in the definition, therefore we apply the next values:

[ ]:
```
Temperatures_A = CrankNicolson_diffusion1D(\
                      n = n_slider.value,\
                      t = t_slider.value,\
                      dt = dt_slider.value,\
                      L = L,\
                      k = k,\
                      Tinit = 200,\
                      boundaryW = 'Neumann',\
                      boundaryE = 'Dirichlet', \
                      TBW = 0,\
                      TBE = 0,\
                      roc = roc,\
                      h = 0,\
                      Tinf = 0,\
                      q = 0)
```

Zapis analitične enačbe v kodi. / Implementation of analytical equation in code.

$\frac{4}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{2n-1}exp(-\alpha\lambda_n^2 t)cos(\lambda_n x)$, kjer je / where is $\lambda_n = \frac{(2n-1)\pi}{2L}$ in / and $\alpha = \frac{k}{\rho c}$

Opomba: upoštevamo prvih 100 členov zaporedja. Note: first 100 terms of sequence are incorporated

[ ]:
```
def analytical(i,x,L=L,dt=dt_slider.value):
    t = dt*i
    exp = 0
    a = k / roc
    for e in range(1,100):
        lam = (2*e-1)*np.pi/2/L
        exp += ((-1)**(e+1))/(2*e-1) * np.exp(-a*lam**2 * t) * np.cos(lam*x)
    return 4*200/np.pi * exp
```

Izris rešitve / Plotting the solution.

Opomba: Analitična rešitev pri času $t = 0\,s$ ni dobro definirana. / Note: Analytical solution at time $t = 0\,s$ is not well defined.

[ ]:
```
fig, ax = plt.subplots()
ax.set_xlabel('x [m]')
ax.set_ylabel('T [°C]')
plt.ylim(min(Temperatures_A.flatten())-2,max(Temperatures_A.flatten())+2)
plt.grid(b=True,which='both')
#Create slider axes
ax_time = fig.add_axes([0.3, 0.9, 0.4, 0.05])
ax_time.spines['top'].set_visible(True)
ax_time.spines['right'].set_visible(True)

# Create sliders
time = Slider(ax=ax_time, label='Time ', valmin = 0, valmax=dt_slider.
 ↪value*t_slider.value//dt_slider.value,
              valfmt=' %1.1f s',valstep=dt_slider.value)

#Plot default data
x = np.linspace(0, L, n_slider.value)
default_y = Temperatures_A[0,:]
line, = ax.plot(x,default_y,'o-')
line1, = ax.plot(x,default_y,'r--')

#Update function
def fun1(i):
    x = np.linspace(0,L,100)
    Temp_vector = analytical(i,x)
    return x, Temp_vector

# Update values
def update(val):
    i = time.val//dt_slider.value
```

```
    line.set_data(x,Temperatures_A[int(i),:])
    analytical_data = fun1(i)
    line1.set_data(analytical_data[0],analytical_data[1])
    fig.canvas.draw_idle()
time.on_changed(update)
```

Poglavje 4

Reševanje Navier-Stokesovih enačb / Solution of Navier-Stokes equations

Skripte, ki implementirajo metode predstavljene v tem poglavju je možno prenesti iz repozitorija na https://github.com/bmavric/SIMPLE_codes.

The scripts implementing the methods presented in this section can be downloaded from the repository at https://github.com/bmavric/SIMPLE_codes.

Implementacije so v naslednjih datotekah / The implementations are in the following files:

- `SIMPLE_notebook.ipynb` - Algoritem SIMPLE za reševanje problema gnane kotanje / SIMPLE algorithm to solve the lid-driven cavity problem

4.1 Implementacija algoritma SIMPLE za primer gnane kotanje / Implementation of SIMPLE algorithm for lid-driven cavity problem

4.1.1 Motivacija / Motivation

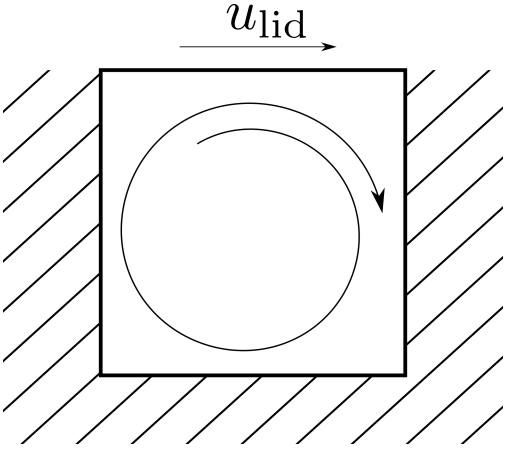
Rešiti želimo problem gnane kotanje, ki je enostaven primer za primerjavo in verifikacijo metod za reševanje Navier-Stokesovih enačb. Geometrija problema je ilustrirana na spodnji sliki.

We want to solve the lid-driven cavity problems, which is a simple case used to compare and verify metods for solution of Navier-Stokes equations. The problem geometry is illustrated in the figure below.

Problem je definiran z sledečimi brezdimenzijskimi enačbami za komponenti hitrosti (u, v) ter tlak p.

The problem is defined with the following dimensionless equations for velocity components (u, v) and pressure p.

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\nabla^2 u$$



$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\nabla^2 v$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

4.1.2 Tlačni popravek / Pressure correction

Ker problem gnane kotanje ni sklopljen z okolico prek tlaka, je potrebno sistem tlačnih enačb regularizirati. V algoritmu SIMPLE dobimo enačbo za tlak prek pogoja, da ima popravljeno hitrostno polje divergenco enako nič. V zveznih spremenljivkah je to enačba, ki povezuje Laplaceov operator na tlačnem polju p z divergenco poskusne hitrosti (u^*, v^*).

Because the lid-driven cavity problem is not coupled to the environment through pressure, the system of equations for p needs to be regularized. In the SIMPLE algorithm, the pressure equation is obtained through the condition that the crrected velocity field is divergence-free. In continuous variables this gives an equation that connects Laplace operator on pressure with divergence of guess velocity (u^*, v^*).

$$\nabla^2 p = \frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y}$$

.

Na robu imamo Neumannove robne pogoje za tlak, ki preprečujejo tok tekočine skozi steno.

At the boundary we have Neumann pressure boundary conditions, which prevent the fluid to penetrate the walls.

$$\frac{\partial p}{\partial n} = g$$

Ta sistem enačb je degeneriran, saj tlačno polje p ni enolično določeno. Če imamo polje p, ki reši parcialno diferencialno enačbo in robne pogoje, potem polje $p' = p+C$, kjer je C poljubna konstanta, tudi rešitev, saj v enačbi nastopajo prvi in drugi odvodi p, ki jih dodana konstanta C ne spremeni. Ob numeričnem reševanju sistema enačb lahko tako dobimo rešitev, kjer je vrednost konstante C

tako visoka, da zaradi končne natančnosti računalnika ni mogoče dovolj točno izračunati odvoda p, ki je potreben za izračun popravljenega hitrostnega polja.

This system of equations is degenerated, since the pressure field p is not uniquely defined. If we have a field p, which solves the PDE and the boundary conditions, then the field $p = p' + C$, where C is an arbitrary constant, is also a solution since the equations are given in terms of first and second derivatives, which are not affected by additional constant C. When numerically solving the system of discretized pressure equations, we can obtain solution with such a high value of the constant C, that we cannot accurately calculate the corrected velocity field because of limited accuracy of computer architecture.

Da to težavo odpravimo, zahtevamo še, da je povprečna vrednost tlačnega polja v kotaniji enaka 0, da pa ohranimo sistem rešljiv je potrebno uvesti Lagrangeov multiplikator α. S tem je sistem regularen saj s tem pogojem enolično določimo vrednost konstante C, vrednost α pa določimo prek diskretiziranega sistema enačb in ima vrednost velikosti diskretizacijske napake. Nov, regulariziran sistem enačb je tako:

To remedy this problem, we additionaly require that the average value of pressure in the cavity equals 0. To keep the system solvable, we interoduce lagrange multiplicator α. This regularizes the system, since it uniquely determines the value of constant C. The value of α is determined when solving the system of equations and should be of the same magnitude as the discretization error. New, regularized system of equations is:

$$\nabla^2 p = \frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y} - \alpha$$

$$\int_V p dV = 0$$

V diksretiziranih enačbah se uvedena regularizacija odraža v dodatni vrstici in stolpcu napolnjenima z vrednostjo 1, ki sta dodana sistemu enačb dobljenih iz SIMPLE algoritma P. Nov sistem linearnih enačb je v bločni obliki:

In discretized equations the introduced regularization is reflected in an additional row and column filled with values 1, that are added to the matrix system P obtained through SIMPLE algorithm. In block form the system of equations is given in block form as:

$$\begin{bmatrix} & & & 1 \\ & P & & 1 \\ & & & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} p \\ \\ \alpha \end{bmatrix} = \begin{bmatrix} \frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y} \\ \\ 0 \end{bmatrix}$$

.

[ ]:
```
#SIMPLE_demo, a demonstration of fluid flow solver using SIMPLE algorithm
#Copyright (C) 2020  Boštjan Mavrič
#
#This program is free software: you can redistribute it and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.
```

```
#
#This program is distributed in the hope that it will be useful,
#but WITHOUT ANY WARRANTY; without even the implied warranty of
#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#GNU General Public License for more details.
#
#You should have received a copy of the GNU General Public License
#along with this program.  If not, see <http://www.gnu.org/licenses/>.

!export MKL_DEBUG_CPU_TYPE=5 # Remove Intel cheats on AMD hardware

from __future__ import division
import matplotlib.pyplot as plt
import math as mt
import numpy as np
import scipy.sparse as sp
import scipy.sparse.linalg as sp_linalg

from IPython.display import display, clear_output

import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual
```

4.1.3 Sistem shranjevanja vrednosti polj / System of field value storage

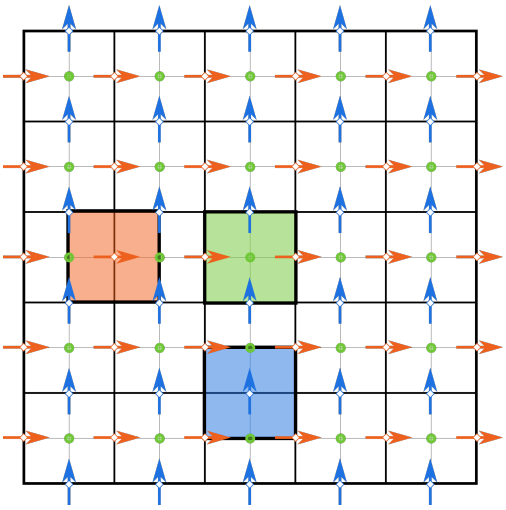
Problem se rešuje na zamaknjeni mreži končnih volumnov predstavljenih na spodnji sliki. Vrednosti tlaka so podane v zelenih točkah, vrednosti u v točkah z rdečo puščico ter vrednosti v v točkah z modro puščico. Z istimi barvami so označeni tudi ustrezni kontrolni volumni, ki jih uporabimo za generiranje diskretiziranih vodilnih enačb.

The problem is being solved on staggered finite volume grid shown on the figure below. The pressure values are given in green points, the values of u in points with blue arrow and the values of u in the points with blue arrow. The same colors are used also to denote corresponding control volumes that are used to generate discretized governing equations.

Vrednosti polj za takšno mrežo lahko neposredno shranimo v matriko. Če je vzdolž ene osi N volumnov, potem za shranjevanja tlačnih vrednosti potrebujemo matriko velikosti $N \times N$. Iz skice je razvidno da za vrednosti u potrebujemo dodatni stolpec, za vrednosti v pa dodatno vrstico. Vrednosti u tako lahko shranimo v matriko velikosti $(N+1) \times N$, vrednosti v pa v $N \times (N+1)$. Tako shranjene vrednosti so primerno urejene za izračun desnih strani enačb.

Field values can be stored directly in a matrix for such a grid. If we position N control volumes along one axis, then we need matrix of size $N \times N$ to store pressure values. From the illustration we can see that we nneds an extra column to store values of u and an extra row to store values v. This means that the values of u can be stored in a matrix of size $(N+1) \times N$, and values of v in a matrix of size $N \times (N+1)$. Such values are ordered in a convenient manner to calculate elements of the right-hand-side vectors.

Za reševanje sistemov enačb, ki so podane z matrikami, pa moramo spremenljivke p, u in v urediti



v dolge vektorje. To naredimo tako da vrstice iz matrik z vrednostmi polj v vektor postavimo eno za drugo v dolge vektorje. Te preslikave podatkov in indeksov med obe načinoma shranjevanja so implementirane v spodnjih funkcijah.

To solve systems of equations, which are given in matrix form, we need to store values of p, u and v in long vectors. We do this by placing the rows of the matrices with field values sequentially one by one in the long vectors. The mappings between data and indices of the two storage schemes are implemented in the functions below.

[ ]:
```
# Preslikava med indeksi premaknjene mreže kontrolnih volumnov na dolge␣
 ↪vektorje za reševanje sistema linearnih enačb.
# / Mapping of the indices from staggered grid to the "long" vectors used to␣
 ↪solve systems of linear equations.
def glob_ind_u(i, j, N):
    return (i) + j*(N+1)

def glob_ind_v(i, j, N):
    return (i) + j*N

def glob_ind_p(i, j, N):
    return (i) + j*N

# Interpolacija hitrostnih vrednosti na položaje tlačnih točk.
# / Interpolate velocity values to the positions of pressure variables.
def interpolate_to_pressure_pos(us, vs, vec, N):
    for i in range(0, N):
        for j in range(0, N):
            vec[0, i, j]=(us[i, j]+ us[i+1, j])/2
            vec[1, i, j]=(vs[i, j]+ vs[i, j+1])/2

# Preslikaj  vrednosti iz dolgih vektorjev na matrike, ki ustrezajo zamaknjeni␣
 ↪mreži montrolnih volumov. /
```

```
# / Map values from "long" vectors to the arrays that correspond to staggered␣
 ↪grid.
def reshuffle_us(u_sol, us, N):
    for i in range(0, N+1):
        for j in range(0, N):
            us[i, j]=u_sol[glob_ind_u(i, j, N)]

def reshuffle_vs(v_sol, vs, N):
    for i in range(0, N):
        for j in range(0, N+1):
            vs[i, j]=v_sol[glob_ind_v(i, j, N)]

def reshuffle_p(p_sol, ps, N):
    for i in range(0, N):
        for j in range(0, N):
            ps[i, j]=p_sol[glob_ind_p(i, j, N)]
```

4.1.4 Implementacija algoritma SIMPLE / Implementation of SIMPLE algorithm

[ ]:
```
# Funkcija, ki reši sistem enačb.
# / Main function used to solve the system of equations.
def SIMPLE_demo(N, FD_kind, Re, alpha_p, alpha_vel):
    Nmax_iters=200

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18,6))

    p_corr_norm=np.zeros(Nmax_iters)
    v_corr_norm=np.zeros(Nmax_iters)
    plt.ion()
    plt.show()

    ###
    # Shranjevanje spremenljivk.
    # / Storage for unknowns.

    # Shramba za u.
    # / Storage of u.
    # u ima eno vrstico spremenljivk več vzdolž x kot vzdolž y.
    # / For u, there is one more row of variables along x axis than along y.
    Nall_u=(N)*(N+1)
    us=np.zeros([N+1, N])
    us_old=np.zeros([N+1, N])
    ds_u=np.zeros([N+1, N])
    rhs_u=np.zeros(Nall_u)
    u_sol=np.zeros(Nall_u)
```

```
    # Shramaba za v
    # / Storage of v.
    # v ima eno eno vrstico spremenljivk vzdolž y osi kot vzdolž x.
    # / There is one more row of v variables along y axis than along x.
    Nall_v=(N+1)*(N)
    vs=np.zeros([N, N+1])
    vs_old=np.zeros([N, N+1])
    ds_v=np.zeros([N, N+1])
    rhs_v=np.zeros(Nall_v)
    v_sol=np.zeros(Nall_v)

    # Shramba za p.
    # / Storage of p.
    Nall_p=(N)*(N)
    ps=np.zeros([N, N])
    vel_at_p=np.zeros([2, N, N])
    vel_norm_at_p=np.zeros([N, N])
    ps_prime=np.zeros([N, N])

    # Za stabilizacijo tlačnega popravka potrebujemo eno enačbo več kot je␣
↪tlačnih vrednosti.
    # / To stabilize pressure correction equation we need on more unknown for␣
↪regularization of the system.
    rhs_p=np.zeros(Nall_p+1)
    p_sol=np.zeros(Nall_p+1)
    p_old=np.zeros(Nall_p+1)

    # Shranjevanje matrik.
    # / Storage for matrices.
    # Redke matrike sestavimo v obliki "seznama seznamov". Preden jih uporabimo␣
↪v reševalniku jih spremenimo v format CSR.
    # / We use sparse matrices in "List-of-lists" for construction. Before␣
↪using them in the solver, they are converted to CSR format.
    sparse_u=sp.lil_matrix((Nall_u, Nall_u))
    sparse_v=sp.lil_matrix((Nall_v, Nall_v))
    sparse_p=sp.lil_matrix((Nall_p+1, Nall_p+1))

    # Generirajmo zamaknjeno mrežo.
    # / Generation of staggered grid.
    h=1./(N)

    # Točke z vrednostjo u.
    # / Points with known value of u.
    points_u=np.zeros([2, Nall_u])
    for i in range(0, N+1):
        for j in range(0, N):
            Ip=glob_ind_u(i, j, N)
```

```
            points_u[0, Ip]=h*i
            points_u[1, Ip]=h/2 + h*j

    # Točke z vrednostjo v.
    # / Points with known value of v.
    points_v=np.zeros([2, Nall_v])
    for i in range(0, N):
        for j in range(0, N+1):
            Ip=glob_ind_v(i, j, N)
            points_v[0, Ip]=h/2 + h*i
            points_v[1, Ip]=h*j

    # Točke z vrednostjo tlaka.
    # / Points with known value of p.
    points_p=np.zeros([2, Nall_p])
    points_p_contour=np.zeros(N)
    for i in range(0, N):
        points_p_contour[i]=h/2 + h*i
        for j in range(0, N):
            Ip=glob_ind_p(i, j, N)
            points_p[0, Ip]=h/2 + h*i
            points_p[1, Ip]=h/2 + h*j


    rho=1e0
    eta=1e0/Re
    v_lid=1e0

    # Izraze poenostavimo za konstantne vrednosti gostote in transportnih␣
↪koeficientov.
    # / The expressions are simplified by assuming constant density and␣
↪constant transport coefficients.
    Gamma=eta

    # Difuzijski koeficienti iz V&M učbenika.
    # / Diffusion coefficents from V&M book.
    Dw=Gamma/h
    De=Gamma/h
    Ds=Gamma/h
    Dn=Gamma/h

    A=h

    # Začetek iteracijske zanke.
    # / Start of the the iteration loop.
    for itr in range(0, Nmax_iters):
        Pe_u=0.
```

```
        ###################################
        # Pripravi sistem enačb za poskusno hitrost (u, v).
        # / Set up the system of equations for guess velocity (u, v).
        ###################################

        ###################################
        # u-komponenta hitrosti, notranjosti.
        # / u-component of velocity, interior.
        for i in range(1, N):
            for j in range(1, N-1):
                # Določi potrebne indekse točk.
                # / Determine indices of points under consideration.
                Ip=glob_ind_u(i, j, N)
                Ie=glob_ind_u(i+1, j, N)
                Iw=glob_ind_u(i-1, j, N)
                In=glob_ind_u(i, j+1, N)
                Is=glob_ind_u(i, j-1, N)

                # Določi advektivne tokove.
                # / Determine advective fluxes.
                Fw=rho*(us[i, j] + us[i-1, j])/2
                Fe=rho*(us[i+1, j] + us[i, j])/2

                Fs=rho*(vs[i, j] + vs[i-1, j])/2
                Fn=rho*(vs[i, j+1] + vs[i-1, j+1])/2

                # Določi koeficiente v linearnem sistemu enačb.
                # / Determine the coefficents used in linear systems of␣
↪equations.
                if(FD_kind=="central"):
                    a_w=Dw+Fw/2
                    a_e=De-Fe/2
                    a_s=Ds+Fs/2
                    a_n=Dn-Fn/2
                elif(FD_kind=="upwind"):
                    a_w=Dw+max([Fw, 0.])
                    a_e=De+max([-Fe, 0.])
                    a_s=Ds+max([Fs, 0.])
                    a_n=Dn+max([-Fn, 0.])
                elif(FD_kind=="hybrid"):
                    a_w=max([Fw, (Dw+Fw/2), 0])
                    a_e=max([-Fe, (De-Fe/2), 0])
                    a_s=max([Fs, (Ds+Fs/2), 0])
                    a_n=max([-Fn, (Dn-Fn/2), 0])

                a_p=a_w+a_e+a_s+a_n +(Fe-Fw)+(Fn-Fs)
```

```
            # Nadzorujmo Pecletovo število.
            # / Monitor the Peclet number.
            Pe_u=max([Pe_u, max([Fw, Fe, Fs, Fn])/Dw])

            # Nastavimo matrične elemente.
            # / Populate the matrix elements.
            sparse_u[Ip, In]=-a_n
            sparse_u[Ip, Iw]=-a_w
            sparse_u[Ip, Ip]=a_p
            sparse_u[Ip, Ie]=-a_e
            sparse_u[Ip, Is]=-a_s

            # Shranimo vrednosti za tlani popravek.
            # / Store values for pressure correction.
            ds_u[i, j]=h/a_p

            # Nastavimo vrednost desne strani za u komponento.
            # / Set up RHS for u-component.
            rhs_u[Ip]=A*(ps[i-1, j] - ps[i, j])

    # u-kompnenta hitrosti v točkah v stiku s severno/južno steno.
    # / u-component of velocity in cells in contact with north/south walls.
    for i in range(1, N):
        for j in [0, N-1]:

            Ip=glob_ind_u(i, j, N)
            Ie=glob_ind_u(i+1, j, N)
            Iw=glob_ind_u(i-1, j, N)
            In=glob_ind_u(i, j+1, N)
            Is=glob_ind_u(i, j-1, N)

            Fw=rho*(us[i, j] + us[i-1, j])/2
            Fe=rho*(us[i+1, j] + us[i, j])/2

            Fs=rho*(vs[i, j] + vs[i-1, j])/2
            Fn=rho*(vs[i, j+1] + vs[i-1, j+1])/2

            if(FD_kind=="central"):
                a_w=Dw+Fw/2
                a_e=De-Fe/2
                a_s=Ds+Fs/2
                a_n=Dn-Fn/2
            elif(FD_kind=="upwind"):
                a_w=Dw+max([Fw, 0.])
                a_e=De+max([-Fe, 0.])
                a_s=Ds+max([Fs, 0.])
```

```
                    a_n=Dn+max([-Fn, 0.])
                elif(FD_kind=="hybrid"):
                    a_w=max([Fw, (Dw+Fw/2), 0])
                    a_e=max([-Fe, (De-Fe/2), 0])
                    a_s=max([Fs, (Ds+Fs/2), 0])
                    a_n=max([-Fn, (Dn-Fn/2), 0])

                sparse_u[Ip, Iw]=-a_w
                sparse_u[Ip, Ie]=-a_e

                # Stena pri y=0.
                # / Wall at y=0.
                if(j==0):
                    a_s=a_s+Ds
                    a_p=a_w+a_e+a_s+a_n +(Fe-Fw)+(Fn-Fs)
                    a_p=a_p
                    sparse_u[Ip, Ip]=a_p
                    sparse_u[Ip, In]=-a_n

                    # Nastavimo vrednost v vektorju desnih strani.
                    # / Set up RHS for u-component.
                    rhs_u[Ip]=A*(ps[i-1, j] - ps[i, j])

                # Stena pri y=1, kjer je gibajoč pokrov.
                # / Wall at y=1, the position of moving lid.
                elif(j==N-1):
                    a_n=a_n+Dn
                    a_p=a_w+a_e+a_s+a_n + (Fe-Fw)+(Fn-Fs)
                    a_p=a_p
                    sparse_u[Ip, Ip]=a_p
                    sparse_u[Ip, Is]=-a_s

                    # Nastavimo desne strani za u-komponento. Dodatni člen␣
↪povzroči gibanje tekočine.
                    # / Set up RHS for u-component, the last term provides the␣
↪driving.
                    rhs_u[Ip]=A*(ps[i-1, j] - ps[i, j]) + a_n*v_lid

                ds_u[i, j]=h/a_p

        # u kompnenta hitrosti v celicah na stiku z vzhodno/zahodno steno.
        # / u-component of velocity in cells in contact with east/west walls.
        for i in [0, N]:
            for j in range(0, N):

                Ip=glob_ind_u(i, j, N)
```

```
            # Dirichletovi robni pogoji.
            # / Dirichlet boundary conditions.
            sparse_u[Ip, Ip]=1

            ds_u[i, j]=0

            # Nastavimo vrednosti v vektorju desnih strani za u-komponento.
            # / Set up RHS for u-component.
            rhs_u[Ip]=0

    # Ponovimo postopek še za v-komponento.
    # / The same procedure for v-component.
    ####################################
    # v-komponenta hitrosti , notranjost.
    # v-component of velocity, interior.
    for i in range(1, N-1):
        for j in range(1, N):

            Ip=glob_ind_v(i, j, N)
            Ie=glob_ind_v(i+1, j, N)
            Iw=glob_ind_v(i-1, j, N)
            In=glob_ind_v(i, j+1, N)
            Is=glob_ind_v(i, j-1, N)

            Fw=rho*(us[i, j] + us[i, j-1])/2
            Fe=rho*(us[i+1, j] + us[i+1, j-1])/2

            Fs=rho*(vs[i, j-1] + vs[i, j])/2
            Fn=rho*(vs[i, j] + vs[i, j+1])/2

            if(FD_kind=="central"):
                a_w=Dw+Fw/2
                a_e=De-Fe/2
                a_s=Ds+Fs/2
                a_n=Dn-Fn/2
            elif(FD_kind=="upwind"):
                a_w=Dw+max([Fw, 0.])
                a_e=De+max([-Fe, 0.])
                a_s=Ds+max([Fs, 0.])
                a_n=Dn+max([-Fn, 0.])
            elif(FD_kind=="hybrid"):
                a_w=max([Fw, (Dw+Fw/2), 0.])
                a_e=max([-Fe, (De-Fe/2), 0.])
                a_s=max([Fs, (Ds+Fs/2), 0.])
                a_n=max([-Fn, (Dn-Fn/2), 0.])

            a_p=a_w+a_e+a_s+a_n +(Fe-Fw) +(Fn-Fs)
```

```
            sparse_v[Ip, In]=-a_n
            sparse_v[Ip, Iw]=-a_w
            sparse_v[Ip, Ip]=a_p
            sparse_v[Ip, Ie]=-a_e
            sparse_v[Ip, Is]=-a_s

            ds_v[i, j]=h/a_p

            # Nastavimo vrednost v vektorju desnih strani za v-komponento.
            # / Set up RHS for v-component.
            rhs_v[Ip]=A*(ps[i, j-1] - ps[i, j])

    # v-komponenta hitrosti v celicah v stiku z vzhodno/zahodno steno.
    # / v-component of velocity in cells in contact with east/west walls.
    for i in [0, N-1]:
        for j in range(1, N):

            Ip=glob_ind_v(i, j, N)
            Ie=glob_ind_v(i+1, j, N)
            Iw=glob_ind_v(i-1, j, N)
            In=glob_ind_v(i, j+1, N)
            Is=glob_ind_v(i, j-1, N)

            Fw=rho*(us[i, j] + us[i-1, j])/2
            Fe=rho*(us[i+1, j] + us[i, j])/2

            Fs=rho*(vs[i, j] + vs[i-1, j])/2
            Fn=rho*(vs[i, j+1] + vs[i-1, j+1])/2

            if(FD_kind=="central"):
                a_w=Dw+Fw/2
                a_e=De-Fe/2
                a_s=Ds+Fs/2
                a_n=Dn-Fn/2
            elif(FD_kind=="upwind"):
                a_w=Dw+max([Fw, 0.])
                a_e=De+max([-Fe, 0.])
                a_s=Ds+max([Fs, 0.])
                a_n=Dn+max([-Fn, 0.])
            elif(FD_kind=="hybrid"):
                a_w=max([Fw, (Dw+Fw/2), 0])
                a_e=max([-Fe, (De-Fe/2), 0])
                a_s=max([Fs, (Ds+Fs/2), 0])
                a_n=max([-Fn, (Dn-Fn/2), 0])

            sparse_v[Ip, In]=-a_n
```

```
            sparse_v[Ip, Is]=-a_s

            if(i==0):
                a_w=a_w + Dw
                a_p=a_w+a_e+a_s+a_n +(Fe-Fw)+(Fn-Fs)
                a_p=a_p
                sparse_v[Ip, Ip]=a_p
                sparse_v[Ip, Ie]=-a_e
            elif(i==N-1):
                a_e=a_e + De
                a_p=a_w+a_e+a_s+a_n +(Fe-Fw)+(Fn-Fs)
                a_p=a_p
                sparse_v[Ip, Ip]=a_p
                sparse_v[Ip, Iw]=-a_w

            ds_v[i, j]=h/a_p

            # Nastavi vrednosti v vektorju desnih strani za v-komponento.
            # / Set up RHS for v-component.
            rhs_v[Ip]=A*(ps[i, j-1] - ps[i, j])

    # v-komponenta hitrosti v celicah v stiku s severno/južno steno.
    # / v-component of velocity in cells in contact with north/south walls.
    for i in range(0, N):
        for j in [0, N]:

            Ip=glob_ind_v(i, j, N)

            sparse_v[Ip, Ip]=1

            ds_v[i, j]=0

            # Nastavi vrednosti v vektorju desnih strani za u-komponento.
            # / Set up RHS for u-component.
            rhs_v[Ip]=0

    # Rešimo sistem enačb za u.
    # / Solve the system of equations for u.
    u_sol=sp_linalg.spsolve(sparse_u.tocsr(), rhs_u)
    reshuffle_us(u_sol, us, N)

    # Rešimo sistem enačb za v.
    # / Solve the system of equations for v.
    v_sol=sp_linalg.spsolve(sparse_v.tocsr(), rhs_v)
    reshuffle_vs(v_sol, vs, N)

    # Sestavimo tlačno matriko.
```

```
        # / Assemble the pressure matrix.
        for i in range(0, N):
            for j in range(0, N):
                Ip=glob_ind_p(i, j, N)
                Ie=glob_ind_p(i+1, j, N)
                Iw=glob_ind_p(i-1, j, N)
                In=glob_ind_p(i, j+1, N)
                Is=glob_ind_p(i, j-1, N)

                a_e=ds_u[i+1, j]/h
                a_w=ds_u[i, j]/h
                a_n=ds_v[i, j+1]/h
                a_s=ds_v[i, j]/h

                # Obravnavamo robne pogoje.
                # / Treat the boundary conditions.
                if(i==0) :
                    a_w=2*a_w
                else:
                    sparse_p[Ip, Iw]=-a_w

                if(i==N-1):
                    a_e=2*a_e
                else:
                    sparse_p[Ip, Ie]=-a_e

                if(j==0):
                    a_s=2*a_s
                else:
                    sparse_p[Ip, Is]=-a_s

                if(j==N-1):
                    a_n=2*a_n
                else:
                    sparse_p[Ip, In]=-a_n

                a_p=a_e+a_w+a_n+a_s
                sparse_p[Ip, Ip]=a_p

                # Izračunajmo vrednosti v vektorju desnih strani.
                # / Determine values in RHS vector for pressure.
                rhs_p[Ip]=h*(us[i,j] - us[i+1,j] + vs[i,j] - vs[i,j+1])

        # Dodamo regularizacijsko enačbo.
        # / Add the regularization equation sum(p)=0.
        rhs_p[Nall_p]=0
        sparse_p[0:Nall_p, Nall_p]=1e0
```

```
        sparse_p[Nall_p, 0:Nall_p]=1e0

        # Rešimo sistem enačb.
        # / Solve the system of equations.
        p_old=p_sol.copy()
        p_sol=sp_linalg.spsolve(sparse_p.tocsr(), rhs_p)
        p_vals=p_sol[0:Nall_p]

        reshuffle_p(p_sol, ps_prime, N)
        p_corr_norm[itr]=np.linalg.norm(ps_prime)/(np.linalg.norm(ps))

        # Posodobimo vrednosti tlaka.
        # / Update pressure.
        ps=ps+alpha_p*ps_prime

        # Popravimo vrednosti u.
        # / Correct u values.
        for i in range(1, N):
            for j in range(0, N):
                us[i, j]=us[i, j] + ds_u[i, j]*(ps_prime[i-1, j]-ps_prime[i, j])

        v_corr_norm[itr]=np.linalg.norm(us-us_old)/(np.linalg.norm(us_old))

        # Popravimo vrednosti v.
        # / Correct v values.
        for i in range(0, N):
            for j in range(1, N):
                vs[i, j]=vs[i, j] + ds_v[i, j]*(ps_prime[i, j-1]-ps_prime[i, j])

        # Podrelaksacija popravkov hitrosti.
        # / Under-relax velocity corrections.
        us=alpha_vel*us + (1.-alpha_vel)*us_old
        us_old=us.copy()

        vs=alpha_vel*vs + (1.-alpha_vel)*vs_old
        vs_old=vs.copy()

        # Priprava podatkov za izris.
        # / Extract data for plotting.
        interpolate_to_pressure_pos(us, vs, vel_at_p, N)
        for i in range(0, N):
            for j in range(0, N):
                vel_norm_at_p[i, j]=np.linalg.norm(vel_at_p[:, i, j])

        pmin=np.amin(ps)
        pmax=np.amax(ps)
```

```
        ax1.cla()
        ax2.cla()
        ax3.cla()

        ax1.set_title("Streamlines and pressure field")

        ax1.contourf(points_p_contour, points_p_contour, ps.transpose(),␣
↪levels=np.linspace(pmin, pmax, 30), cmap='YlGnBu_r')
        ax1.streamplot(points_p[0, 0:N], points_p[1, 0:Nall_p:N], vel_at_p[0, :
↪, :].transpose(), vel_at_p[1, :, :].transpose(), color=vel_norm_at_p.
↪transpose(), density=1, linewidth=3., cmap='hot')

        ax1.set_aspect('equal')
        ax1.set_xlim(0, 1)
        ax1.set_ylim(0, 1)

        ax2.set_title("Relative magnitude of corrections")
        ax2.set_yscale('log')
        press_line=ax2.plot(p_corr_norm[1:itr], label="Pressure")
        vel_line=ax2.plot(v_corr_norm[1:itr], label="Velocity")
        ax2.legend()

        ax3.plot(points_p_contour, vs[:, int(N/2)+1], label="v @ y=0.5")
        ax3.plot(points_p_contour, us[int(N/2)+1, :], label="u @ x=0.5")
        ax3.set_title("Velocity profiles")
        ax3.legend()
        #plt.draw()
        clear_output(wait=True)
        display(fig)
        #fig.show()
        plt.pause(.0001)


    # Izriši končno rešitev.
    # / Plot final solution.
    plt.ioff()
    plt.clf()
    plt.title("Final velocity")
    plt.axes().set_aspect('equal')
    plt.streamplot(points_p[0, 0:N], points_p[1, 0:Nall_p:N], vel_at_p[0, :, :].
↪transpose(), vel_at_p[1, :, :].transpose(), color=vel_norm_at_p.transpose(),␣
↪density=3)
    plt.show()

    pmin=np.amin(ps)
    pmax=np.amax(ps)
```

```
    plt.clf()
    plt.title("Final pressure")
    plt.axes().set_aspect('equal')
    plt.contourf(points_p_contour, points_p_contour, ps.transpose(), levels=np.
↪linspace(pmin, pmax, 20))
    plt.show()
```

[ ]:
```
def wrapper_for_visualization(N, alpha_p, alpha_v, Re, uw_type):
    SIMPLE_demo(N, uw_type, Re, alpha_p, alpha_v)

N_slider = widgets.IntSlider(min=10, max=100, step=10, value=20)
Re_slider = widgets.FloatSlider(min=1, max=1000, step=50, value=100)
alpha_p_slider = widgets.FloatSlider(min=0.01, max=1., step=0.1, value=0.9)
alpha_v_slider = widgets.FloatSlider(min=0.01, max=1., step=0.1, value=0.9)

uw_inter = widgets.Dropdown(options=['central','upwind', 'hybrid'],
                            value='central',
                            description='Upwinding scheme')

w=widgets.interactive(wrapper_for_visualization,
                      N=N_slider,
                      alpha_p=alpha_p_slider,
                      alpha_v=alpha_v_slider,
                      Re=Re_slider,
                      uw_type=uw_inter)


display(w)
```

Literatura

[1] F. Moukalled, L. Mangani, M. Darwish, The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab, Springer Verlag, Cham, 2016.

[2] J.H. Ferziger, M. Perić, R.L. Street, Computational Methods for Fluid Dynamics, 4th Edition, Springer Nature, Cham, 2020.

[3] H. K. Versteeg, W. Malalasekera, An Introduction to Computational Fluid Dynamics, The Finite Volume Method, 2nd Edition, Pearson, Harlow, 2007.

[4] G.R. Liu, Mesh Free Methods: Moving Beyond Finite Element Method, CRC Press, Boca Raton, 2nd Edition, 2009