پروژه درس ریاضیات مهندسی پیشرفته

نام استاد:

دکتر ابوالقاسمی

نام دانشجو:

دارا رحمت سمیعی

دی ۱۴۰۱

فهرست مطالب

فصل ۱

مقدمه و تعریف مسئله

هدف از پروژه درس ریاضیات مهندسی پیشرفته در ترم پاییز ۱۴۰۱ شمسی حل معادله دیفرانسیل پاره‌ای غیرخطی ۱.۱ می‌باشد.

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(u\frac{\partial u}{\partial x}) \tag{۱.۱}$$

$$BC: \begin{cases} u(0,t) = 1 \\ u(1,t) = 2 \end{cases}$$

$$IC: u(x,0) = 0$$

مسئله ۱.۱ را به سه روش زیر حل خواهیم کرد.

- حل با روش Implicit
- حل با روش P.S.O.R[۱]
- حل با روش Method of Lines

[۱] Point Successive Over-Relaxation Method

فصل ۲

حل با روش Implicit

در این بخش روش حل عددی معادله‌ی ۱.۱ با روش حل ضمنی می‌پردازیم. ابتدا معادله را گسسته‌سازی کرده و سپس به تشریح برنامه کامپیوتری خواهیم پراخت و در نهایت نتایج به دست‌آمده را مصورسازی و تاثیر پارامترهای مختلف مثل گام زمانی و اندازه گام مکانی را بررسی خواهیم کرد.

۱.۲ گسسته‌سازی معادله

روش گسسته سازی پیشرو [۱] نسبت به زمان و میانی [۲] نسبه به مکان خواهد بود.

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(u\frac{\partial u}{\partial x}) = \frac{\partial u}{\partial x}.\frac{\partial u}{\partial x} + u.\frac{\partial^2 u}{\partial x^2} \tag{۱.۲}$$

$$N_i = \frac{\partial u}{\partial x} = \frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x}$$

$$M_i = u_i$$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = N_i.\frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x} + M_i.\frac{u_{i+1}^{n+1} - 2.u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} \tag{۲.۲}$$

معادله‌ی ۲.۲ را در Δt ضرب کرده و مرتب می‌کنیم.

$$\lambda_1 = \frac{N_i.\Delta t}{2\Delta x}; \qquad \lambda_2 = \frac{M_i.\Delta t}{\Delta x^2}$$

$$-u_i^n = (\lambda_1 + \lambda_2).u_{i+1}^{n+1} + (-1 - 2\lambda_2).u_i^{n+1} + (\lambda_2 - \lambda_1).u_{i-1}^{n+1} \tag{۳.۲}$$

چنیش ماتریسی به‌صورت زیر در خواهد آمد:

$$\alpha = (\lambda_1 + \lambda_2); \qquad \beta = (-1 - 2\lambda_2); \qquad \gamma = (\lambda_2 - \lambda_1)$$

---

[۱] forward
[۲] Central

$$\begin{bmatrix} \beta & \alpha & 0 & \dots & 0 \\ \gamma & \beta & \alpha & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \gamma & \beta & \alpha \\ 0 & \dots & 0 & \gamma & \beta \end{bmatrix}^k \times \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{N-2}^{n+1} \\ u_{N-1}^{n+1} \end{bmatrix}^{k+1} = \begin{bmatrix} -u_1^n - \alpha.u_0^n \\ -u_2^n \\ \vdots \\ -u_{N-2}^n \\ -u_{N-1}^n - \gamma.u_N^n \end{bmatrix}$$

با معکوس کردن ماتریس سه قطری به دست آمده و حل دستگاه ماتریس مقادیر u در زمان $n+1$ به دست می‌آید

$$A \times U^{n+1} = U^n \longrightarrow U^{n+1} = U^n \times A^{-1}$$

۲.۲ برنامه‌ی کامپیوتری حل

برای نوشتن برنامه حل عددی معادله‌ی ۱.۱ از زبان برنامه‌نویسی python استفاده می‌کنیم. در بخش اول کد کتابخانه‌های لازم را import می‌کنیم. از پکیج numpy برای انجام عملیات‌های ریاضی و از پکیج matplotlib برای رسم نمودارها و کانتورهای نتایج استفاده می‌کنیم.

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

مقادیر متغییرهای مسئله را تعریف می‌کنیم

```
U_0 = 1.0          #lower boundary value
U_last = 2.0       #upper boundary value

L = 1.0            # Lenght
target_tol = 0.01 #target telorance
T = 500            #number of timesteps

delta_x = 0.005   # spacing step
delta_t = 0.001   #timestep
num_x = int(L/delta_x)
```

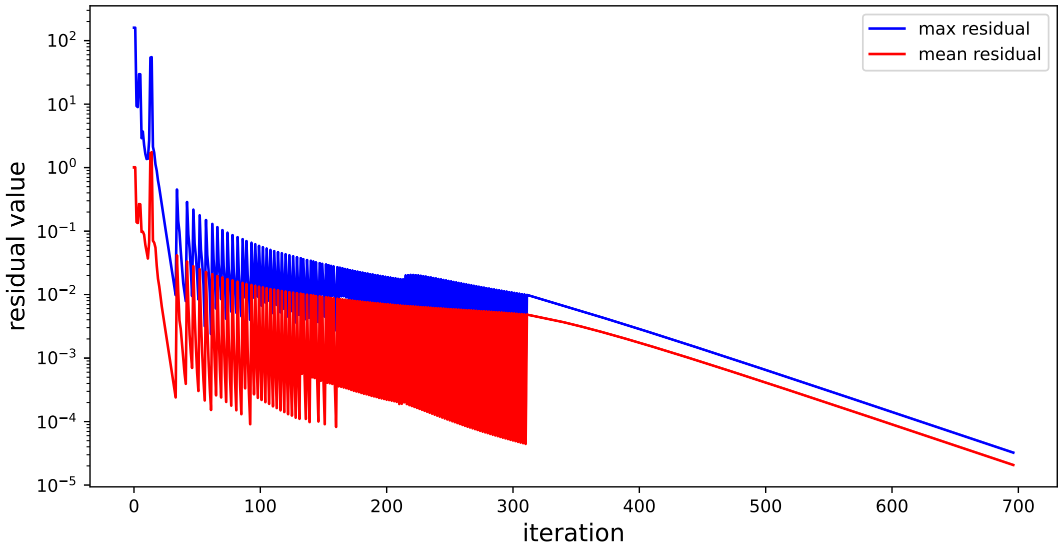
لیست‌هایی برای ذخیره‌ی مقادیر نهایی و مقادیر میانی هنگام حل ایجاد می‌کنیم:

```
U_all = []
residual_max = []
residual_mean = []

U = np.zeros(num_x)    #array for U
U[0] = U_0         #setting lower value of U
U[-1] = U_last       #setting upper value of U

U_all.append(U)
```

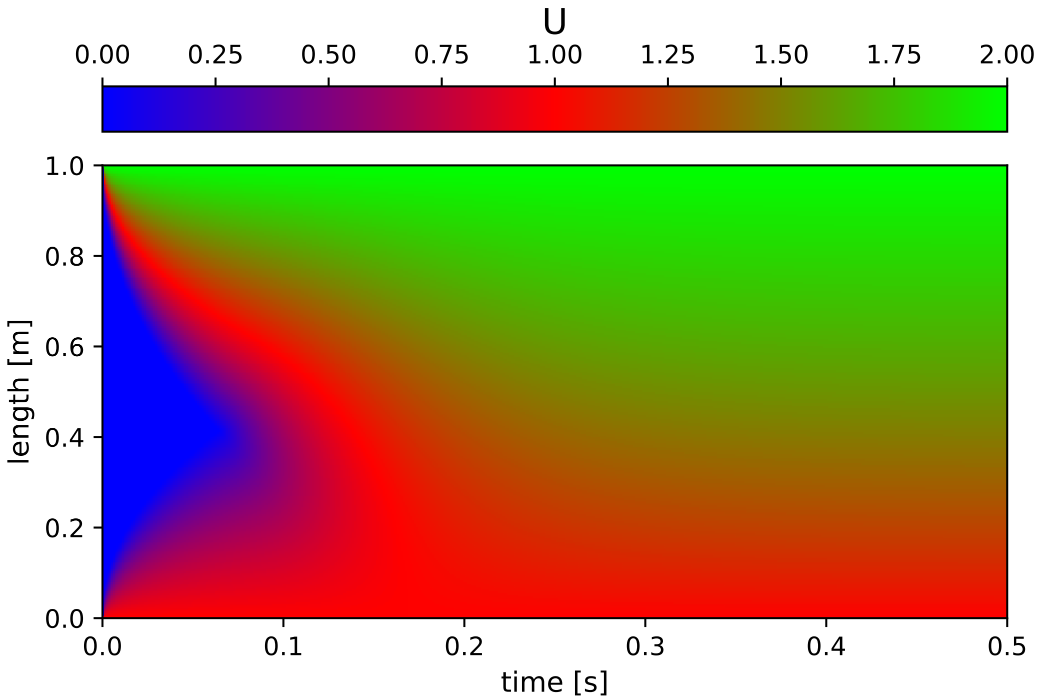
حال مراحل اصلی حل را تشریح می‌کنیم:

```
for n in range(T):     #entering the Timing loop
  Uk = np.copy(U)   #Uk in U_{n+1} in (k+1) iteration

  N = np.zeros(num_x) #array holder for N
  M = np.zeros(num_x) #array holder for M

  U[0] = U_0       #updating boundaris of U in older timestep
  U[-1] = U_last

  Uk[0] = U_0      #updating boundaris of U in older timestep
  Uk[-1] = U_last

  tol = 100      #setting a high tolerance

  while tol > target_tol:   #enting while loop for solving U in n+1
                                    timestep

    for i in range(1,num_x -1):   #updating N values from Uk
      N[i] = (Uk[i+1] - Uk[i-1]) / (2 * delta_x)

    M = np.copy(Uk)


    lambda_1 = N * delta_t /(2*delta_x)     #computing lambda1 and lambda2
    lambda_2 = M * delta_t /(delta_x**2)

    A = np.zeros((num_x,num_x))   #matrix holder

    for i in range(num_x):      #filling the A matix
      for j in range(num_x):
        if i == j:
          A[i,j] = -1 -2*lambda_2[i]
        if i+1 == j:
          A[i,j] = lambda_2[i] + lambda_1[i]
        if i-1 == j:
          A[i,j] = lambda_2[i] - lambda_1[i]

    C = A[1:-1,1:-1]   #fixing A matrix
    B = -1* U[1:-1]    #creating known array

    #modifying the lower and upper valus of kown U
    B[0] = B[0] - gamma[0] * U[0]
    B[-1] = B[-1] - alpha[-1] * U[-1]


    Uk_new = np.linalg.solve(C,B)    #sloving the matrix system

    tol = np.max(np.abs(Uk_new - Uk[1:-1]))   #computing telorance
    residual_max.append(tol)
    residual_mean.append(np.mean(np.abs(Uk_new - Uk[1:-1])))

    Uk[1:-1] = Uk_new    #updating the Uk


  U = Uk               #updaitng the U value and going to next timestep
  U_all.append(U)
```

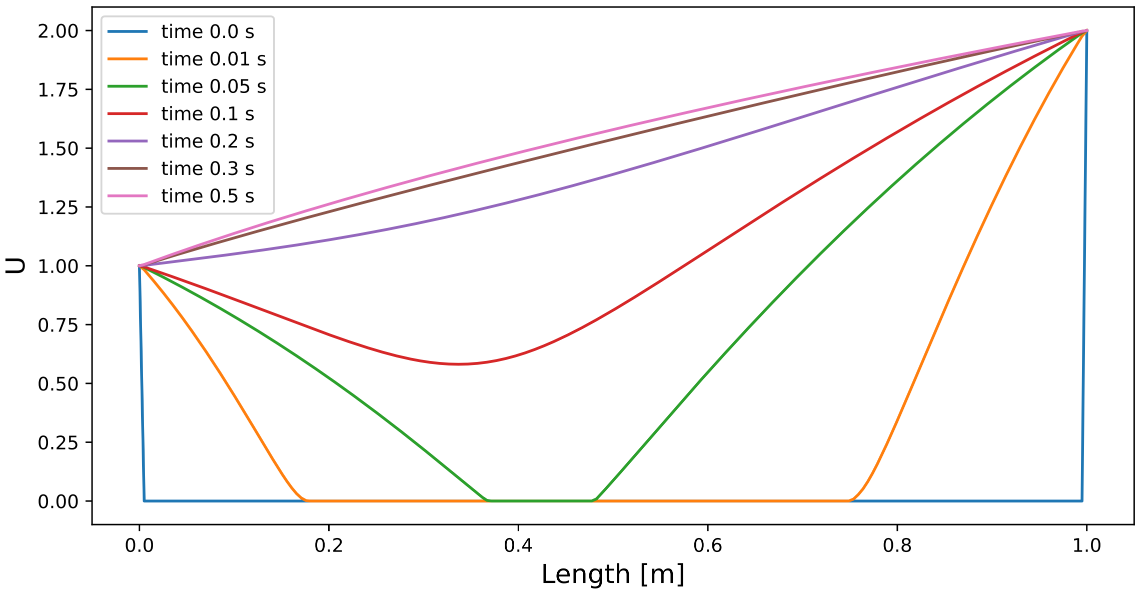
۳.۲ نتایج



شکل ۱.۲: نمودار تغییرات باقی‌مانده فرایند حل عددی



شکل ۲.۲: کانتور تغییرات U نسبته به زمان و مکان



شکل ۳.۲: نمودار تغییرات U نسبت به مکان در زمان‌های مختلف

در شکل ۳.۲ مشاهده می‌شود که روند حل بسیار نوسان دارد. پس گذشت حدود ۳۰۰ گام زمانی و پس از ثانیه 0.3 ثانیه نتایج حل تقریبا پایا شده و با افرایش گام زمانی تغییرات بسیار کم می‌باشد.

شکل ۳.۲ رنگ آبی نشانه مقدار صفر، رنگ قرمز مقدار ۱ و رنگ سبز نشانه مقدار ۲ می‌باشد. مشاهده می‌شود که در زمان‌های پایین و نزدیک به مقار اولیه رنگ آبی بسیار است اما با گذشت زمان مقادیر U تقریبا معادله‌ای خطی از مقادیر ۱ تا ۲ در طول بازه حل در می‌آیند. این موضوع در شکل ۳.۲ بهتر دیده می‌شود.

فصل ۳

حل با روش P.S.O.R

۱.۳ گسسته‌سازی مسئله

[۱] نسبت به زمان و میانی [۲] نسبه به مکان خواهد بود.

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(u\frac{\partial u}{\partial x}) = \frac{\partial u}{\partial x}.\frac{\partial u}{\partial x} + u.\frac{\partial^2 u}{\partial x^2} \tag{۱.۳}$$

$$N_i = \frac{\partial u}{\partial x} = \frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x}$$

$$M_i = u_i$$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = N_i.\frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x} + M_i.\frac{u_{i+1}^{n+1} - 2.u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} \tag{۲.۳}$$

معادله‌ی ۲.۳ را در Δt ضرب کرده و مرتب می‌کنیم.

$$\alpha = \frac{N_i.\Delta t}{2\Delta x}; \qquad \beta = \frac{M_i.\Delta t}{\Delta x^2}$$

$$u_i^{n+1} = \frac{1}{1+2\beta}[(\beta+\alpha)u_{i+1}^{n+1} + (\beta-\alpha)u_{i-1}^{n+1} + u_i^n] \tag{۳.۳}$$

به معادله‌ی ۳.۳ عبارت $u_i^{n+1,k} - u_i^{n+1,k}$ را اضافه می‌کنیم. k شمارنده‌ی حل می‌باشد.

$$u_i^{n+1,k+1} = u_i^{n+1,k} + \frac{1}{1+2\beta}[(\alpha+1)u_{i+1}^{n+1,k} + (\alpha+1)u_{i-1}^{n+1,k} + u_i^n(1+2\beta)u_i^{n+1,k}] \tag{۴.۳}$$

برای سرعت بخشیدن به فرایند حل ω را در عبارت اصلاح ضرب می‌کنید و ساده سازی می‌کنیم.:

$$u_i^{n+1,k+1} = (1-\omega)u_i^{n+1,k} + \frac{\omega}{1+2\beta}[(\beta+\alpha)u_{i+1}^{n+1,k} + (\beta-\alpha)u_{i-1}^{n+1,k} + u_i^n] \tag{۵.۳}$$

---

[۱] forward
[۲] Central

۲.۳ برنامه‌ی کامپیوتری حل

برای نوشتن برنامه حل عددی معادله‌ی ۱.۱ از زبان برنامه‌نویسی python استفاده می‌کنیم. در بخش اول کد کتابخانه‌های لازم را import می‌کنیم. از پکیج numpy برای انجام عملیات‌های ریاضی و از پکیج matplotlib برای رسم نمودارها و کانتورهای نتایج استفاده می‌کنیم.

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

مقادیر متغییرهای مسئله را تعریف می‌کنیم. به دلیل آنکه مقادیری که در بخش حل $Implicit$ برای Δx و Δt استفاده کردیم در حل $P.S.O.R$ منجر به واگرا شدن حل می‌شد از مقادیری که $\frac{\Delta t}{\Delta x}$ کوچکتری تشکیل می‌دهد استفاده کردیم.

```
U_0 = 1.0           #lower boundary value
U_last = 2.0        #upper boundary value

L = 1.0             # Lenght
target_tol = 0.01   #target telorance
T = 500             #number of timesteps

delta_x = 0.02      #spacing step
delta_t = 0.0001    #timestep
num_x = int(L/delta_x)
```

لیست‌هایی را برای ذخیره نتایج به‌دست آمده هنگام حل تعریف می‌کنیم.

```
U_all = []              #list for stoing results
residual_max = []       #list for storing residuals
U = np.zeros(num_x)     #creating an matirx for U

U[0] = U_0              #setting Lower boundary
U[-1] = U_last          #setting Upper boundary
U_all.append(U)
```

```
for t in range(T):  #entering the time stepping loop
  Un = U.copy()

  tol = 100
  #entering while loop for computing new U
  while tol > target_tol:
    Uold_k = Un.copy()

    #looping over x for computing new Uk
    for i in range(1,num_x-1):
      N = (Uold_k[i+1] - Uold_k[i-1])/(2*delta_x)
      M = Uold_k[i]

      alpha = (N * delta_t) / (2 * delta_x)
      beta  = (M * delta_t) / (delta_x ** 2)

      Un[i] = (1 - w)*Un[i] +..
              (w/(1 + (2*beta)))*((alpha +beta)*Un[i+1] +..
              (beta - alpha)*Un[i-1] + U[i])

    #Computing new tolerance
    tol = np.max(np.abs(Uold_k - Un))

    residual_max.append(tol)

  U_all.append(Un)
U = Un
```

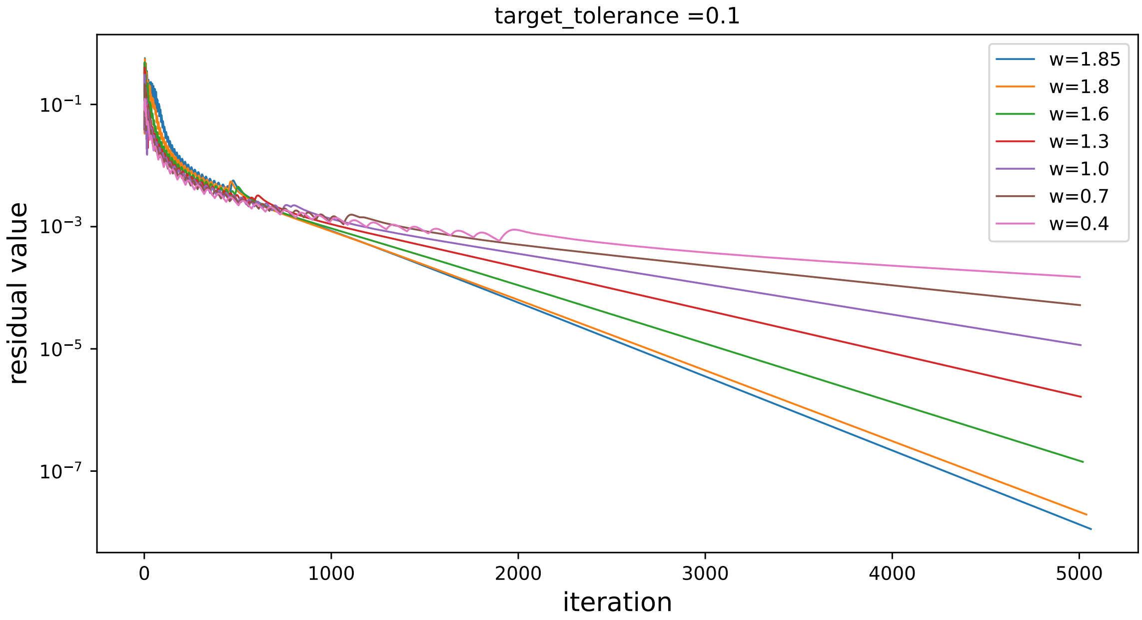
۳.۳ نتایج

تاثیر فاکتور آسایش [۳] را در چند ۷ مقدار مختلف و همچنین تاثیر دقت هدف را در ۳ مقدار بررسی کردیم. شاهد آن هستیم که در مجموع تعداد iteration های روش $P.S.O.R$ بسیار بیشتر از روش $Implicit$ است.

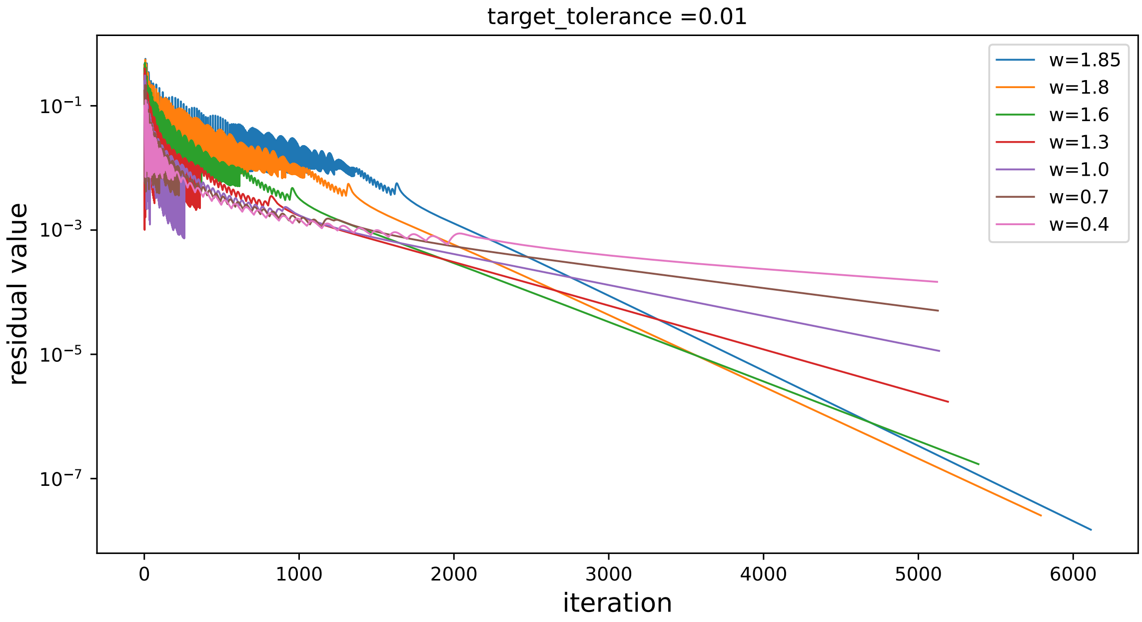
در دقت‌های کم تاثیر ω محدود است و تغییرات آن تاثیر بسزایی بر تعداد حلقه‌های حل ندارد. اما با افزایش دقد(کاهش تلورانس) این تاثیر افزایش می‌یابد. چنانکه شاهد هستیم در دقت ۰/۰۰۱ تعداد حلقه‌های حل ωهای کمتر از ۱/۰ نزدیک به ۷۰۰۰ می‌باشد ولی در ω های برابر ۶.۱ و ۸.۱ به ترتیب منجر به حل در نزدیک به ۹۰۰۰ و ۱۴۰۰۰ حلقه حل شدند.

در نهاتی نتایج به‌دست آمده‌ی این روش نیز با توجه به نمودار زمان و مکان ۳.۳ و ۳.۳ نیز مشابه روش $Implicit$ می‌باشد.

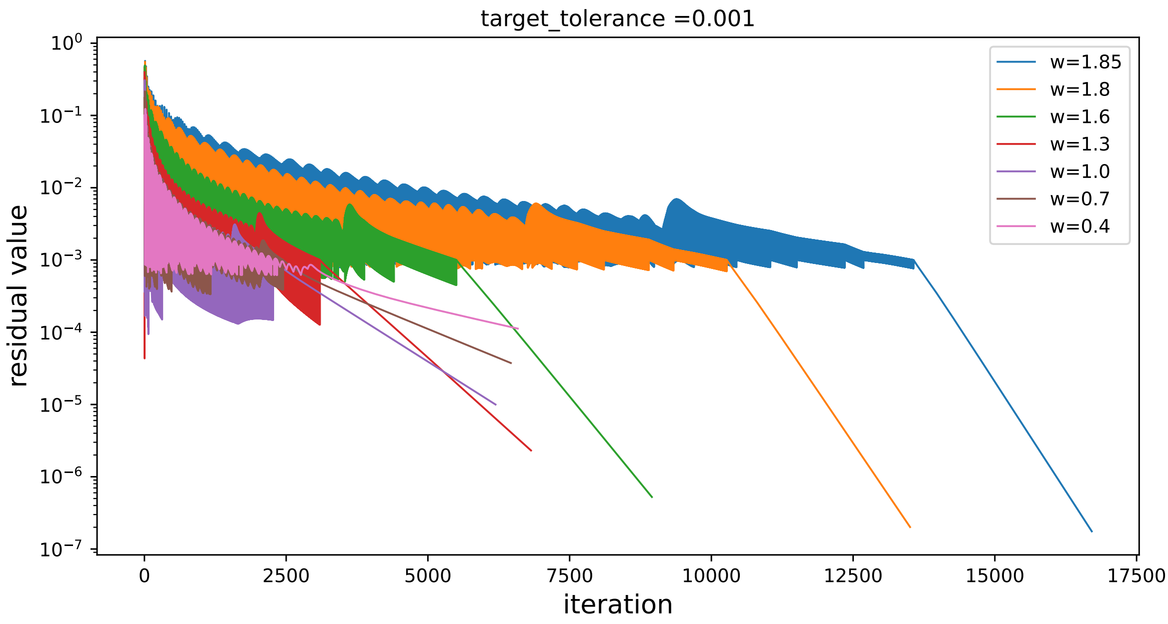
[۳] □□□□□□ □□□□□□□□□□



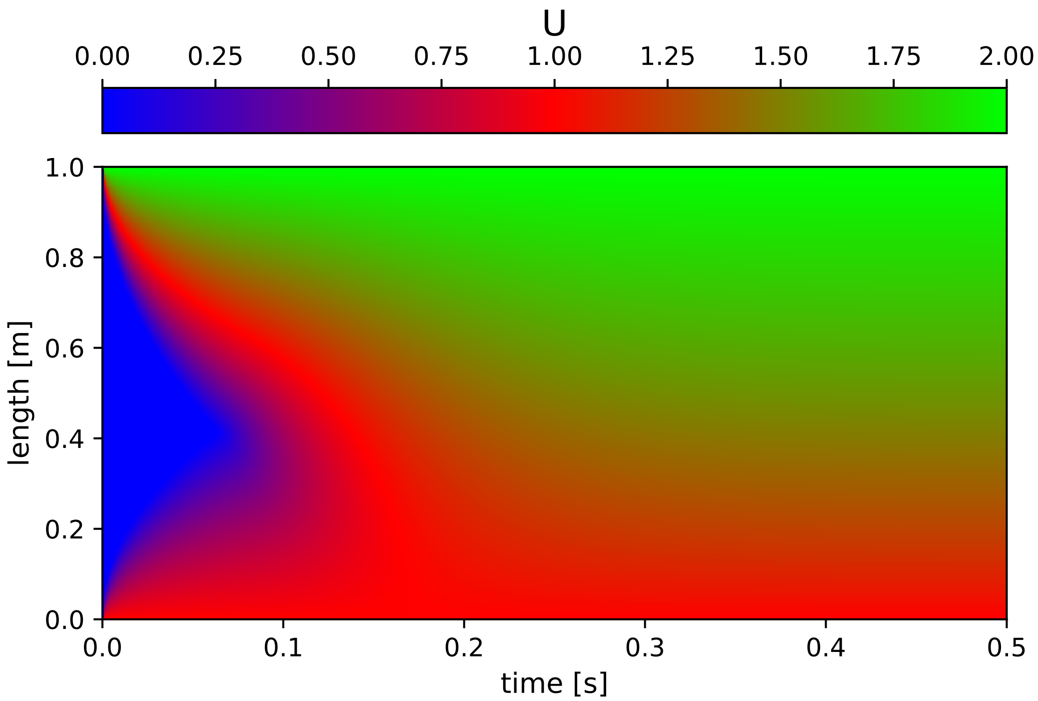
شکل ۱.۳: نمودار تغییرات مقدار باقی‌مانده‌ی حل در ضریب آسایش‌های مختلف



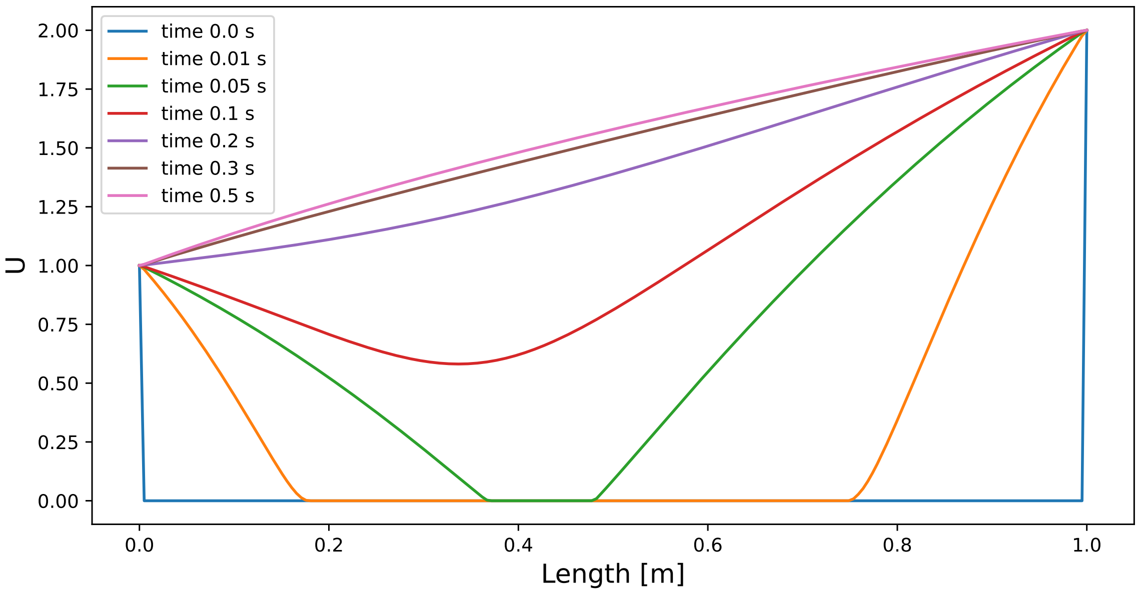
شکل ۲.۳: نمودار تغییرات مقدار باقی‌مانده‌ی حل در ضریب آسایش‌های مختلف



شکل ۳.۳: نمودار تغییرات مقدار باقی‌مانده‌ی حل در ضریب آسایش‌های مختلف



شکل ۴.۳: کانتور تغییرات U نسبته به زمان و مکان



شکل ۵.۳: نمودار تغییرات U نسبت به مکان در زمان‌های مختلف

فصل ۴

حل با روش Method of Line

در این روش هدف کلی تبدیل معادلات دیفرانسیل پاره‌ای به معادلات دیفرانسیل معمولی است. بدین صورت که معادله فقط نسبت به یک متغیر مستقل گسسته سازی می‌شود و دستگاهی از معادلات دیفرانسیل معمولی تشکیل می‌شود. سپس با استفاده از الگوریتم‌های حل دستگاه‌های معادلات دیفرانسیل معمولی جواب معادله به‌دست می‌آید.

۱.۴ گسسته‌سازی مسئله

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(u\frac{\partial u}{\partial x}) = \frac{\partial u}{\partial x}.\frac{\partial u}{\partial x} + u.\frac{\partial^2 u}{\partial x^2} \tag{۱.۴}$$

معادله‌ی ۱.۴ را نسبت به x گسسته‌سازی می‌کنیم.

$$\frac{\partial u}{\partial t} = (\frac{u_{i+1} - u_{i-1}}{2\Delta x})^2 + u_i\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \tag{۲.۴}$$

۱.۱.۴ مثال ساده

برای درک بهتر یک مثال ساده را کامل گسسته می‌کنیم و دستگاه معادلات آن را تشکیل می‌دهیم. با توجه به محدود مسئله $x = 0.2$ در نظر می‌گیریم. دستگاه معادلات زیر حالصل می‌شود:

$$\begin{cases} \frac{\partial u_1}{\partial t} = (\frac{u_2 - u_0}{2\Delta x})^2 + u_i\frac{u_2 - 2u_1 + u_0}{\Delta x^2} \\ \frac{\partial u_2}{\partial t} = (\frac{u_3 - u_1}{2\Delta x})^2 + u_i\frac{u_3 - 2u_2 + u_1}{\Delta x^2} \\ \frac{\partial u_3}{\partial t} = (\frac{u_4 - u_2}{2\Delta x})^2 + u_i\frac{u_4 - 2u_3 + u_2}{\Delta x^2} \\ \frac{\partial u_4}{\partial t} = (\frac{u_5 - u_3}{2\Delta x})^2 + u_i\frac{u_5 - 2u_4 + u_3}{\Delta x^2} \end{cases} \tag{۳.۴}$$

که از شرایط مرزی هم داریم:

$$\begin{cases} u_0 = 1 \\ u_5 = 2 \end{cases} \tag{۴.۴}$$

دستگاه معادلات ۳.۴ با الگوریتم‌های همچون $ode23$ و یا $ode45$ قابل حل است.

۲.۴ برنامه کامپیوتری حل

به دلیل وجود توابع از پیش تعریف شده در نرم‌افزار و زبان برنامه نویسی Matlab از این برنامه برای نوشتن کد حل معادله به روش Method of Lines استفاده می‌کنیم.

ابتدا تابعی که دستگاه معادلات دیفرانسیل را تشکیل می‌دهد را می‌نویسیم:

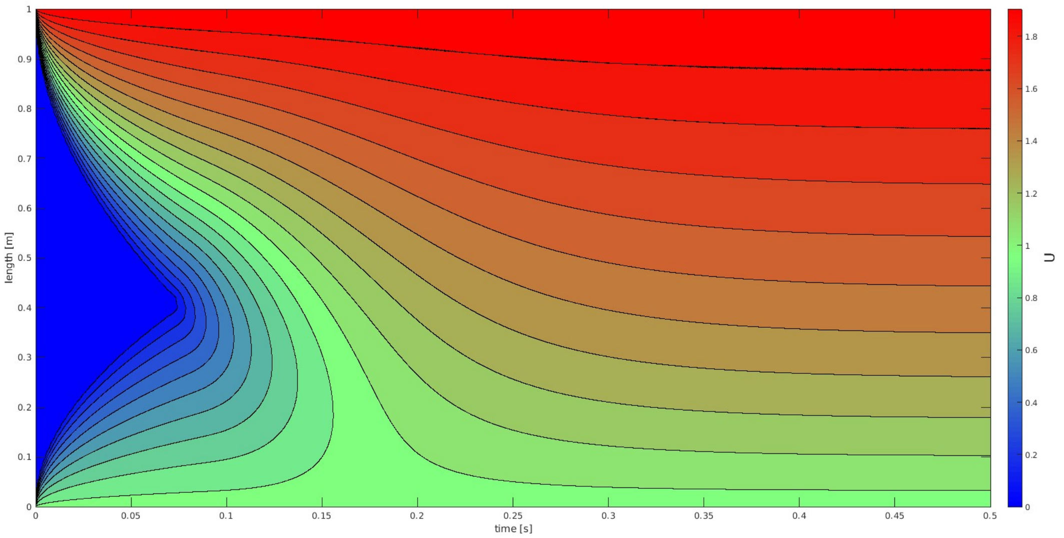
```
function dUdt = MOL(t, U, L, dx)
    U(1) = 1;        % lower boundary value
    U(end) = 2;      % upper boundary value

    npoints = (L /dx)+1;          % computing number of points
    dUdt = zeros(npoints,1);      %matrix for sorint derivatives

    dUdt(1) = 0;

    % loopig over the x
    for i=2:npoints-1
        dUdt(i) =((U(i+1)-U(i-1))/(2*dx))^2 + U(i)*((U(i+1) -
           2*U(i) + U(i-1))/(dx^2));
    end
    dUdt(end) = 0;
end
```

حال به تعریف شرایط اولیه، تعداد نقاط حل و تنظیمات $ode45$ می‌پردازیم.

```
clear; close all; clc;

L = 1;       % length
dx = 0.02;   % x sizing

npoints = (L/dx) + 1;

Uint = zeros(npoints,1);    %initial values

U(1) = 1;        % lower boundary value
U(end) = 2;      % upper boundary value

% ode45 function
[t,U] = ode45(@(t,U) MOL(t,U,L,dx),[0 0.5],Uint);
```

۳.۴ نتایج

نتایج نهایی به‌دست آمده مشابه نتایج دو الگوریتم بررسی شده پیشین است. اما با مقایسه با دو الگوریتم دیگر کد کمتری نیاز دارد و حساسیت کمتری نسبت به اندازه Δx دارد و تقریبا در تمامی مقادیر مختلف نتیجه همگرا می‌شود.



شکل ۱.۴: کانتور تغییرات U نسبته به زمان و مکان

فصل ۵

مراجع

- G. van Rossum, Python tutorial, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995
- MATLAB. (2022a). Natick, Massachusetts: The MathWorks Inc.
- Lin, Ching-long, Merryn H. Tawhai, Geoffrey Mclennan, and Eric A. Hoffman. ”Computational fluid dynamics.” IEEE Engineering in Medicine and Biology Magazine 28, no. 3 (2009): 25-33.
- Versteeg, Henk Kaarle, and Weeratunge Malalasekera. An introduction to computational fluid dynamics: the finite volume method. Pearson education, 2007.

فصل ۶

پیوست

۱.۶ برنامه مصورسازی نمودار باقی‌مانده‌ها ر پایتون

```
plt.figure(figsize=(10,5))

plt.plot(residual_max,label='max residual',c='b')
plt.plot(residual_mean,label="mean residual",c='r')

plt.yscale('log')
plt.legend()
plt.xlabel('iteration',fontsize=14)
plt.ylabel('residual value',fontsize=14)

plt.savefig("residual.pdf")
```

۲.۶ برنامه مصورسازی کانتور U نسبت به زمان و مکان در پایتون

```
UU = np.stack(U_all)

plt.imshow(UU.T,cmap='brg',origin='lower',extent=[0,T*delta_t,0,1],aspect=1
        /4)

cbar = plt.colorbar(location="top")
cbar.ax.tick_params(labelsize=10)
cbar.set_label("U",fontsize=14)

plt.ylabel("length [m]",fontsize=11)
plt.xlabel("time [s]",fontsize=11)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

plt.savefig("conotur.pdf")
```

۳.۶ برنامه مصورسازی نمودار U نسبت به مکان در زمان‌های مختلف در پایتون

```
X = np.linspace(0,1,num_x)

plt.figure(figsize=(10,5))

plt.plot(X,UU[0],label='time 0.0 s')
plt.plot(X,UU[10],label='time 0.01 s')
plt.plot(X,UU[50],label='time 0.05 s')
plt.plot(X,UU[100],label='time 0.1 s')
plt.plot(X,UU[200],label='time 0.2 s')
plt.plot(X,UU[300],label='time 0.3 s')
plt.plot(X,UU[-1],label='time 0.5 s')

plt.legend(loc='upper left')
plt.xlabel("Length [m]",fontsize=14)
plt.ylabel("U",fontsize=14)

plt.savefig("times.pdf")
```

۴.۶ برنامه مصورسازی کانتور U نسبت به زمان و مکان در matlab

```
x = linspace(0,L,npoints);
[X,T] = meshgrid(x,t);

r = linspace(1,0,100);
b = linspace(0,1,100);
g= [linspace(0,1,50), linspace(1,0,50)];

mymap = [b;g;r];
colormap(mymap')

contourf(T,X,U,20)

xlabel("time [s]");
ylabel("length [m]");
a=colorbar;
ylabel(a,'U','FontSize',16)
```