Solution of the heat equation with the FDM

$$u_t = c^2\nabla^2 u : 0 < x < a,\ 0 < y < b$$

Let's import the required packages first of all.

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

Before we get on with the problem at hand, let's make our life easier by defining function for frequent tasks.

First of all let's create the toplevel function to wrap everything. It all the data creates a grid, does the computation and creates and saves the visualization. The fuction takes:

- boundary conditions (a 4×2, array-like where each row has the boundary type (`dirichlet` or `neumann` [these are keywords]), and corresponding value (temperature for dirichlet and temperature derivative for neumann) in the top, right, bottom, left sequence),
- type of analysis to be performed (`steady` or `transient` [these are keywords]),
- the system dimensions (a $1{\times}2$ array-like that has the length of the 2-D system being considered for analysis in the order x, y),
- resolution (a 1×2 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y),
- the value of c^2 as appears in the original differential equation at the top of this documentation (optional, not needed for steady-state analysis, default is 0.005),
- the time for which the computation needs to be done for transient analysis (optional, not needed for steady-state analysis, defaults to 60),
- the time resolution i.e. the number of sub-intervas to compute per unit of time (optional, not needed for steady state analysis, defaults to 2),
- accuracy to which to compute the data (optional, not needed for transient-state analysis, defaults to 0.1),
- the name of the file to save the generated visual as (optional, defaults to `analysis.gif`),
- interval through which each frame in the visual lasts (optional, not needed for steady state analysis, defaults to 200 ms).

```
def analyze(bound, type, sys_scale, res, c=0.005, t=60, tres=2, accuracy=0.1, name="analysis
    res+=[tres]
    u=np.zeros((sys_scale[1]*res[1]+1,sys_scale[0]*res[0]+1,t*res[2]+1))
    calculate(bound, type, u, c, res, accuracy)
    visualize(type, u, name , inv)
```

Now, let's create a function that will create visualization of the data. The function takes:

- type of analysis to be performed (`steady` or `transient` [these are keywords]),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`, the time index is not relevant for steady-state analysis, in which case the t=0 data is plotted),
- the name of the file to save the generated visual as,
- interval through which each frame in the visual lasts.

```
def visualize(type, u, name, inv):
    if (type is steady):
        f=plot(0, u)
        f.savefig(name)
    else:
        animate(plot, u, u.shape[2], name, inv)
```

The function to plot thermal map contours. The function takes:

- the time index value in the 3D dataset,
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`).

```
def plot(t, u):
    plt.clf()
    plt.contourf(u[:,:,t], cmap=plt.cm.jet)
    plt.colorbar()
    return plt
```

The function to animate our plots. The function takes:

- the function responsible for creating each frame,
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- number of frames to add in the animation,
- the name of the file to save the generated visual as,
- interval through which each frame in the visual lasts.

```
def animate(plot, u, f, name, inv):
    anim=ani.FuncAnimation(plt.figure(), plot, frames=f, fargs=(u,), interval=inv)
    anim.save(name)
```

The toplevel function that wraps all computation operations. The function takes:

- boundary conditions (a 4×2, array-like where each row has the boundary type (`dirichlet` or `neumann` [these are keywords]), and corresponding value (temperature for dirichlet and temperature derivative for neumann) in the top, right, bottom, left sequence),
- type of analysis to be performed (`steady` or `transient` [these are keywords]),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`, the time index is not relevant for steady-state analysis, in which case the time dimension is used to compare successive accuracy improvement),
- the value of c^2 as appears in the original differential equation at the top of this documentation,
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t),
- accuracy to which to compute the data.

```
def calculate(bound, type, u, c, res, acc):
    if(type is steady):
        std(bound, u, acc)
    if(type is transient):
        trans(bound, u, c, res)
```

The function to manage the steady state computation. The function takes:

- boundary conditions (a 4×2, array-like where each row has the boundary type (`dirichlet` or `neumann` [these are keywords]), and corresponding value (temperature for dirichlet and temperature derivative for neumann) in the top, right, bottom, left sequence),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`, the time index is used to compare successive accuracy improvement),
- accuracy to which to compute the data.

```
def std(bound, u, acc):
    k=1
    while(k):
        grid_fill(bound, steady, 0, u)
        if(np.all(abs(u[:,:,1]-u[:,:,0])<=acc)):
            k=0
        u[:,:,0]=u[:,:,1]
```

The function to manage transient state computation. The function takes:

- boundary conditions (a 4×2, array-like where each row has the boundary type (`dirichlet` or `neumann` [these are keywords]), and corresponding

value (temperature for dirichlet and temperature derivative for neumann) in the top, right, bottom, left sequence),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the value of c^2 as appears in the original differential equation at the top of this documentation,
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t).

```
def trans(bound, u, c_2, res):
    for t in range(u.shape[2]-1):
        grid_fill(bound, transient, t, u, c_2, res)
```

A function to loop through our dataset manage calculations. The function takes:

- boundary conditions (a 4×2, array-like where each row has the boundary type (`dirichlet` or `neumann` [these are keywords]), and corresponding value (temperature for dirichlet and temperature derivative for neumann) in the top, right, bottom, left sequence),
- type of analysis to be performed (`steady` or `transient` [these are keywords]),
- the value of time index at which to fill the dataset with computed values,
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the value of c^2 as appears in the original differential equation at the top of this documentation, (optional defaults to 0.005),
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t, optional, defaults to `[1,1,1]`).

```
def grid_fill(bound, type, t, u, c_2=0.005, res=[1,1,1]):
        for y in range(0, u.shape[0]):
            for x in range(0, u.shape[1]):
                ev=edge(u, x,y)
                if(ev!=-1):
                    bound[ev][0](bound[ev][1], type, u, x, y, t, c_2, res)
                else:
                    type(u, x, y, t, c_2, res, xp=u[y][x+1][t], xm=u[y][x-1][t], yp=u[y+1][
```

The function to determine whether a grid point is a boundary point. The function takes:

- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate,
- the y co-ordinate.

```
def edge(u, x, y):
    if (y== u.shape[0]-1 and x != 0):
        return 0
    elif (x== u.shape[1]-1 and y!=u.shape[0]-1):
        return 1
    elif (y==0 and x!=u.shape[1]-1):
        return 2
    elif (x==0 and y != 0):
        return 3
    else:
        return -1
```

The function to determine whether a grid point is a corner point. The function takes:

- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate,
- the y co-ordinate.

```
def corner(u, x, y):
    if (y== u.shape[0]-1 and x == 0):
        return 3
    elif (x== u.shape[1]-1 and y==u.shape[0]-1):
        return 0
    elif (y==0 and x==u.shape[1]-1):
        return 1
    elif (x==0 and y == 0):
        return 2
    else:
        return -1
```

Now let's define our boundary conditions.

For Dirichlet's boundary condition we have the data values given at boundaries at all times. The function takes:

- the value of the temperature at the boundary,
- type of analysis to be performed (`steady` or `transient` [these are keywords], not needed except for syntactical consistency),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate of the grid point in consideration,
- the y co-ordinate of the grid point in consideration,

- the t co-ordinate of the grid point in consideration(not needed except for syntactical consistency),
- the value of c^2 as appears in the original differential equation at the top of this documentation, (not needed except for syntactical consistency),
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t, not needed except for syntactical consistency).

```
def dirichlet(value, type, u, x, y, t, c_2, res):
    u[y, x,:]=value
```

For Neumann's boundary condition we have the value of first derivative at the boundaries. The function takes:

- the value of the temperature at the boundary,
- type of analysis to be performed (`steady` or `transient` [these are keywords]),
- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate of the grid point in consideration,
- the y co-ordinate of the grid point in consideration,
- the t co-ordinate of the grid point in consideration,
- the value of c^2 as appears in the original differential equation at the top of this documentation,
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t).

```
def neumann(value, type, u, x, y, t, c_2, res):
    ev=edge(u, x,y)
    cv=corner(u, x,y)
    if (ev==0 and cv==-1):
        type(u, x, y, t, c_2, res, xp=u[y][x+1][t], xm=u[y][x-1][t], ym=u[y-1][x][t], yp=u[y
    elif(ev==0 and cv==0):
        type(u, x, y, t, c_2, res, yp=u[y-1][x][t]+2*res[0]**-1*value, xp=u[y][x-1][t]+2*res
    elif (ev==1 and cv==-1):
        type(u, x, y, t, c_2, res, xp=u[y][x-1][t]+2*res[1]**-1*value, xm=u[y][x-1][t], yp=u
    elif (ev==1 and cv==1):
        type(u, x, y, t, c_2, res, xp=u[y][x-1][t]+2*res[1]**-1*value, ym=u[y+1][x][t]-2*res
    elif (ev==2 and cv==-1):
        type(u, x, y, t, c_2, res, ym=u[y+1][x][t]-2*res[0]**-1*value, xp=u[y][x+1][t], xm=u
    elif (ev==2 and cv==2):
        type(u, x, y, t, c_2, res, ym=u[y+1][x][t]-2*res[0]**-1*value, xm=u[y][x+1][t]-2*res
    elif (ev==3 and cv==-1):
        type(u, x, y, t, c_2, res, xm=u[y][x+1][t]-2*res[1]**-1*value, xp=u[y][x+1][t], yp=u
    elif (ev==3 and cv==3):
        type(u, x, y, t, c_2, res, xm=u[y][x+1][t]-2*res[1]**-1*value, yp=u[y-1][x][t]+2*res
```

Now let's create functions to represent the equations.

First the function to represent the steady state equation.

$$u(x, y, t) = \frac{1}{4}(u(x+\Delta x, y, t) + u(x-\Delta x, y, t) + u(x, y+\Delta y, t) + u(x, y-\Delta y, t))$$

The function takes:

- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate of the grid point in consideration,
- the y co-ordinate of the grid point in consideration,
- the t co-ordinate of the grid point in consideration,
- the value of c^2 as appears in the original differential equation at the top of this documentation,
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t),
- the value of the $u(x+\Delta x, y, t)$,
- the value of the $u(x-\Delta x, y, t)$,
- the value of the $u(x, y+\Delta y, t)$,
- the value of the $u(x, y-\Delta y, t)$.

```
def steady(u, x, y, t, c_2, res, xp, xm, yp, ym):
        u[y][x][t+1]=(1/4)*(yp+ym+xp+xm)
```

The function to represent the transient state equation:

$$u(x, y, t+\Delta t) = u(x, y, t) + \frac{\Delta t . c^2}{(\Delta x \Delta y)^2}((\Delta y)^2(u(x+\Delta x, y, t) + u(x-\Delta x, y, t) - 2u(x, y, t)) + (\Delta x)^2(u(x, y+\Delta y, t) + u(x, y-\Delta y, t) - 2u(x, y, t)))$$

The function takes:

- the dataset (a 3D array-like in the indexing order `arr[y][x][t]`),
- the x co-ordinate of the grid point in consideration,
- the y co-ordinate of the grid point in consideration,
- the t co-ordinate of the grid point in consideration,
- the value of c^2 as appears in the original differential equation at the top of this documentation,
- resolution (a 1×3 array-like where each number is the number of computational sub-intervals to add for unit physical length of the system along the corresponding dimension in the sequence x, y, t),
- the value of the $u(x+\Delta x, y, t)$,

- the value of the $u(x - \Delta x, y, t)$,
- the value of the $u(x, y + \Delta y, t)$,
- the value of the $u(x, y - \Delta y, t)$.

```
def transient(u, x, y, t, c_2, res, xp, xm, yp, ym):
        u[y][x][t+1]=u[y][x][t]+(res[2]**-1*c_2*(res[1]*res[0])**2)*(res[0]**-2*(xp+xm-2*u[y
```

Now we come to our main code.

We set data to represent the system and and other analysis parameters. And call the `analyze()` function with all the requisite arguments (see the definition of `analyze()` for more information).

```
system=[5,3]
res=[10,10]
time=60
time_res=4
c_2=0.005
acc=.01
bound=[[dirichlet, 500], [dirichlet, 700], [dirichlet, 800], [dirichlet, 600]]
analysis=transient
name="dirichlet_transient.gif"
inv=50
analyze(bound, analysis, system, res, t=time, tres=time_res, accuracy=acc, c=c_2, name=name,
```

Next up some results.

- system : $0 < x < 5, \ 0 < y < 3$

  boundary :

  `- top : isothermal, 500`

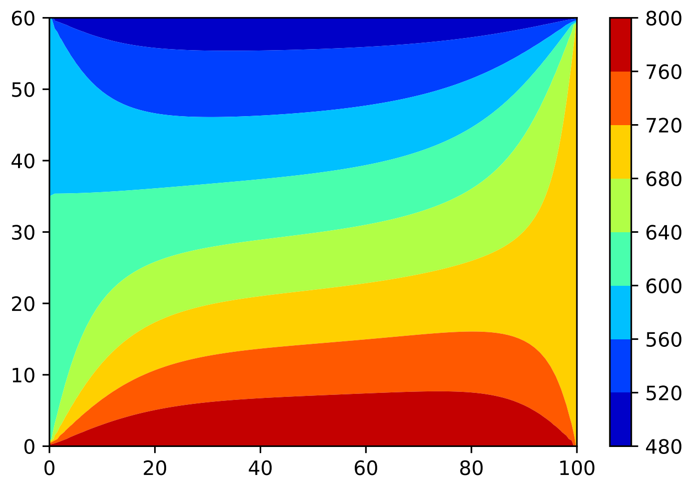
  `- right : isothermal, 700`

  `- bottom : isothermal, 800`

  `- left : isothermal, 600`

  accuracy : 0.01

  Output



- system : $0 < x < 5,\ 0 < y < 3$

  boundary :

  `- top : adiabatic, 10`

  `- right : isothermal, 400`

  `- bottom : adiabatic, 10`

  `- left : isothermal, 800`

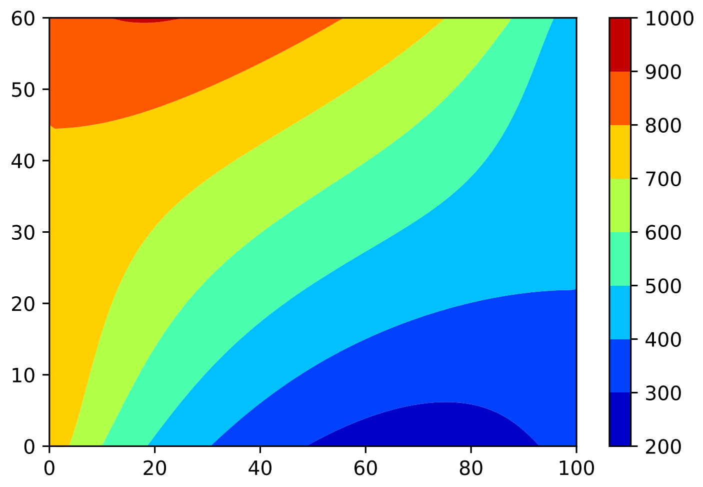
  resolution :

  `- x : 20`

  `- y : 20`

  accuracy : 0.01

  Output



- system : $0 < x < 5,\ 0 < y < 3, 0 < t < 60$

  boundary :

  `- top : isothermal, 500`

  `- right : isothermal, 700`

  `- bottom : isothermal, 800`

  `- left : isothermal, 600`

  resolution :

  `- x : 20`

  `- y : 20`

  `- t : 4`

  c^2 : 0.005

  Output

- system : $0 < x < 5,\ 0 < y < 3, 0 < t < 60$

  boundary :

  `- top : adiabatic, 10`

  `- right : isothermal, 400`

`- bottom : adiabatic, 10`

`- left : isothermal, 800`

resolution :

`- x : 20`

`- y : 20`

`- t : 4`

c^2 : 0.005

Output