Journal of Applied Engineering Mathematics

Volume 11, December 2024

NUMERICAL APPROXIMATION FOR HEAT FLOW BETWEEN ADJACENT REGIONS

Ethan Cleaver

Mechanical Engineering Department
Brigham Young University
Provo, Utah 84602
epcleaver12@comcast.net

ABSTRACT

The heat equation is applied to adjacent regions with an interior boundary condition which creates an insulator effect between the regions. Regions are treated separately so that the interior boundary is transformed into two exterior, or regular, boundaries. Results are found and compared using numerical approximations for different values of heat flow and insulation between regions. It is shown that larger intervals of approximation are adequate for modeling systems with slower heat flow or greater insulation. The largest interval that does not diverge should be used to prevent excessive time for computation.

NOMENCLATURE

α = Thermal conductivity
u = temperature distribution
u_{ss} = steady state temperature distribution
U_t = transient temperature distribution

INTRODUCTION

The heat equation (1) is regularly applied during analysis of heat flow through a single region of consistent thermal conductivity. In typical use, this region can be idealized based on the number of dimensions present. These problems typically fall under either a rectangular coordinate system or some type of polar coordinate system. This region has associated boundary conditions and initial conditions which both influence the flow of heat through the region. The setup for such a problem is seen in figure 1 for a 2D rectangular case.

$$1)\ \nabla^2 u = \frac{1}{\alpha}\frac{\partial u}{\partial t}$$

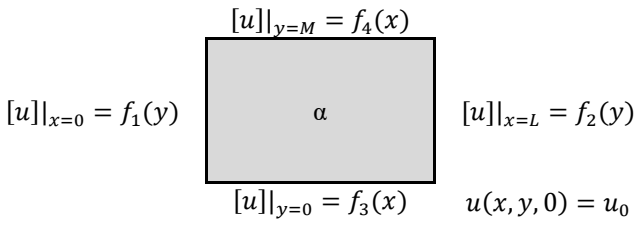

Figure 1. A typical setup for a standard heat equation problem in two dimensions. Boundary conditions can be of Dirichlet, Neuman, or Robin type.

$$2)\ \alpha(x) = \alpha_1 + H(x-1)(\alpha_2 - \alpha_1) + k\delta(x-1)$$

However, in some cases it may be desirable to know how heat flows through multiple regions in contact with each other, such as in figure 2. These regions may have similar or different coefficients of thermal conductivity. Additionally, it is generally appropriate to include an insulating condition in the form of an interior boundary condition between two regions of interest. Such a condition would create a discontinuity in temperature between the two regions at the boundary between them. This could be modelled by using a position-dependent function for thermal conductivity (2) by using heaviside and dirac-delta functions, however, the heat equation (1) can only be applied for constant values of α. In order to solve problems such as these by using the heat equation, numerical approximation methods must be utilized. The accuracy of these methods is largely dependent on any values that determine the rate at which models change with respect to time.

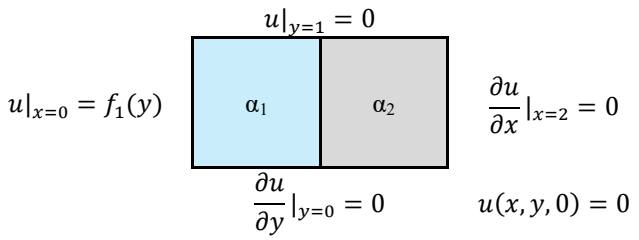

Figure 2. Two regions in contact with each other containing an interior boundary condition at x=1 (the interface between regions). Initial condition and exterior boundary conditions are chosen arbitrarily.

MODELING

Using numerical approximation methods allows the two regions to be treated individually, transforming the interior boundary condition into two equivalent exterior boundary conditions. For region 1, this would appear as in figure 3. Region 2 can be defined similarly. The constant 'k' can be given physical significance based on its value. A value of k=0 would be equivalent to perfect insulation between the regions, at which point numerical approaches would be unnecessary. Larger values of 'k' would correspond to 'weaker' insulation and higher rates of heat transfer between regions 1 and 2. While evaluating a numerical approach, the new boundary conditions can be treated as regular Neuman conditions, however, they must be re-evaluated at every interval of time. Similarly, the initial conditions for each successive time interval must also be re-evaluated at every interval. As a result of this, the initial conditions shown in figures 2 and 3 are only applicable to the very first approximation interval.

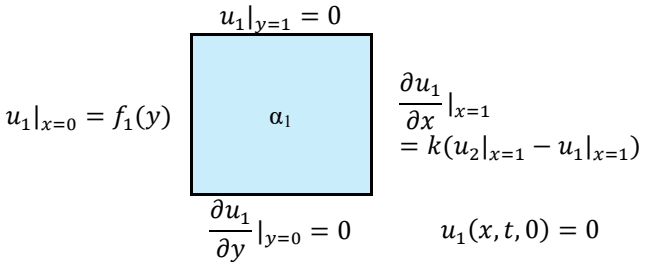

Figure 3. Region 1 with interior boundary condition transformed into an equivalent exterior boundary condition. Region 2 can be similarly constructed.

SOLVING

From this point, regions 1 and 2 can be solved as independent heat equations (1). For each region, this can be divided into a steady-state solution, and a transient solution. The full solutions of the heat equations for each region will be given by adding the steady state and transient solutions (see equations 6 and 13). The objective of this paper is not to show extensive derivations that can be easily found elsewhere, so discussion of derivations will be limited. Derivations will also only be shown for region 1. Equations are contained in appendix A.

Steady State Solution

For the steady state solutions, $\frac{\partial u}{\partial t}$ is set equal to zero, and the principle of superposition is applied so that non-homogeneous boundaries can be handled one at a time. For each case, all boundaries except for one are set to be homogeneous. Steady state solutions for region 1 and region 2 are given by equations 7-9, 12 and 14-15, 19, respectively.

Transient Solution

For transient solutions, all boundaries are set to be homogeneous as these conditions were handled by the steady state solution. The full solution $u(x,y,t)$ must be equal to the initial condition at t=0, therefore the initial condition for the transient solution is defined by equation (3). U_{t1} can then be solved using two separations of variables given by equations (4) and (5). These separations lead to three eigenvalue problems that can be solved to yield a solution for U_{t1} in terms of a double summation containing a constant that is dependent on both variables of summation. This constant is solved for by setting $t = 0$ and performing a double integration. Transient solutions for region 1 and region 2 are given by equations 10-12 and 16-19, respectively.

$$3)\ U_{t1}|_{t=0} = u_1|_{t=0} - u_{ss1}$$

$$4)\ U_{t1}(x,y,t) = \phi(x,y)T(t)$$

$$5)\ \phi(x,y) = X(x)Y(y)$$

RESULTS AND ANALYSIS

Full solutions for the heat flow through regions 1 and 2 are given by equations 6-12 and 13-19 in appendix A, respectively. As this is a numerical approximation, however, some variables should be interpreted differently than they typically would be. Here, 't' refers to the interval of time between successive approximations. Similarly, $u_1|_{t=0}$ actually represents u_1 as given by the previous approximation interval. The term: $(u_2|_{x=1} - u_1|_{x=1})$ also represents the boundary values given by the previous approximation interval.

In practice, numerical approaches use computer software to generate data as computations are typically quite complex and numerous. This paper uses a code written in python, which can be found in appendix B.

Two approximations were calculated and graphed to show the importance of appropriate choice of interval. Both of these used intervals of 0.05s, 0.025s, and 0.01s and took data at 3 different times. The rightmost boundary was set as $f_1(y)=50$ for simplicity. As can be seen in figure 4, the first approximation used values of k=0.5, α_1=0.4, and α_2=2. These values were chosen to simulate relatively slow rates of heat transfer. Although each graph contains 3 lines of data, only one can be

seen. The reason for this is that all approximation intervals are equally good at approximating slow rates of heat transfer.

In contrast, figure 5 shows the second approximation that used values of k=1, α_1=2, and α_2=5. These values were chosen to simulate much faster rates of heat transfer. Here, it can be seen that approximation intervals of 0.025s and 0.01s both eventually converge to essentially the same values. However, it is also observed that interval of 0.025s takes slightly longer to converge, although the difference is not of much consequence. In this approximation, it is important to note that using an interval of 0.05s leads to a diverging result. This is directly caused by the boundary between regions. As time progresses, the rate of heat transfer through this boundary increases and then decreases before reaching a constant (steady-state) value. By using a larger approximation interval, this heat transfer is treated as a constant rate for each interval. Here, the interval of 0.05s leads to diverging results because it sets this rate to be either too high or too low for too long, leading to far more or less heat transfer than actually occurs.

It may reasonably be assumed that two regions in contact with each other will eventually have the same temperature at the shared surface, however this is not seen here. The reason for this result arises from the homogenous boundary at y=1. By holding the temperature here at 0, much of the heat introduced to the system at x=0 is lost through this boundary instead of being transferred to the second region. As a result of this, there is always a heat difference between the two regions at the shared boundary.

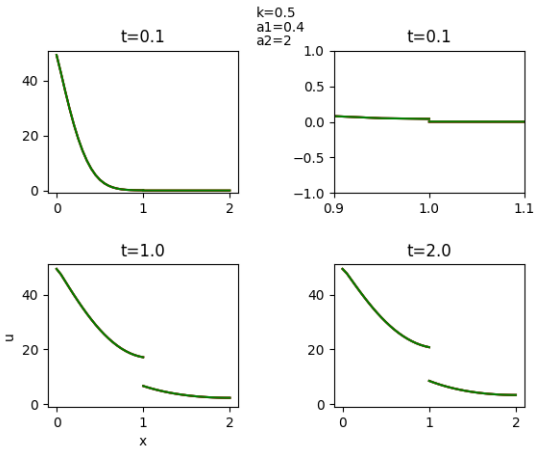

Figure 4. Curves shown for data at y=0 for the first approximation. Graph at the top-right shows an enlarged section of the top-left graph. Approximation intervals are graphed in different colors; however, all are essentially the same as interval 0.01s, which is graphed in green on top of them.

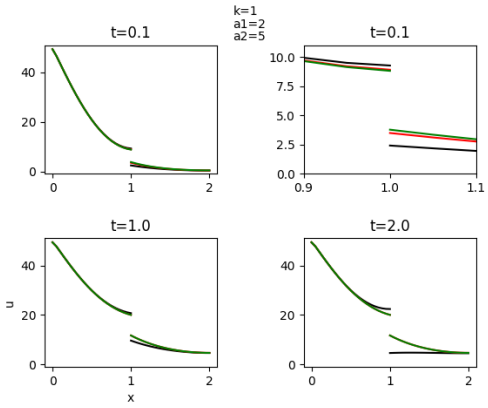

Figure 5. Curves shown for data at y=0 for the second approximation. Graph at the top-right shows an enlarged section of the top-left graph. Intervals 0.05s, 0.025s, and 0.01s are graphed in black, red, and green, respectively.

CONCLUSIONS

As seen earlier, it can be useful to utilize numerical methods in approximating the flow of heat between adjacent regions or materials. Particularly, it has been noted that for slow rates of heat transfer and for interior boundaries having greater values of insulation, larger intervals of approximation are adequate. For higher rates of heat transfer or for interior boundaries with low insulation to heat, however, larger intervals of approximation may cause simulations to diverge and yield useless data. In general, when performing numerical approximations for heat flow between two regions, the largest interval of approximation that does not diverge should be used. The reason for this is that computations take far longer to perform when using progressively smaller intervals. If heat transfer rates are exceptionally high or if there is very little insulation between regions, other methods of solution should be considered.

APPENDIX A – FULL SOLUTIONS

6) $u_1 = u_{ss1} + U_{t1}$

7) $u_{ss1} = \sum_{l=1}^{\infty} \left[a_l \cos(\mu_l y) \cosh\left(\mu_l (x-1)\right) + b_l \cos(\mu_l y) \sinh(\mu_l x) \right]$

8) $a_l = \dfrac{2}{\cosh(-\mu_l)} \int_0^1 f_1(y) \cos(\mu_l y) dy$

9) $b_l = \dfrac{2k}{\mu_l \cosh(\mu_l)} \int_0^1 \left(u_2|_{x=1} - u_1|_{x=1} \right) \cos(\mu_l y) dy$

10) $U_{t1} = \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} A_{nm} \cos(\lambda_n y) \sin(\nu_m x) e^{-\alpha_1 (\lambda_n^2 + \nu_m^2) t}$

11) $A_{nm} = 4 \int_0^1 \int_0^1 (u_1|_{t=0} - u_{ss1}) \sin(\nu_m x) \cos(\lambda_n y) dy dx$

12) $\mu_l = \pi \left(l - \frac{1}{2} \right),\ \lambda_n = \pi \left(n - \frac{1}{2} \right),\ \nu_m = \pi \left(m - \frac{1}{2} \right)$ applicable to equations 6 to 11

13) $u_2 = u_{ss2} + U_{t2}$

14) $u_{ss2} = \sum_{l=1}^{\infty} c_l \cos(\mu_l y) \cosh(\mu_l (x-2))$

15) $c_l = \dfrac{4k}{\mu_l \sinh(-\mu_l)} \int_0^1 (u_2|_{x=1} - u_1|_{x=1}) \cos(\mu_l y) dy$

16) $U_{t2} = \sum_{n=1}^{\infty} [B_{n0} \cos(\lambda_n y) e^{-\alpha_2 (\lambda_n^2) t}] + \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} [B_{nm} \cos(\nu_m (x-1)) \cos(\lambda_n y) e^{-\alpha_2 (\lambda_n^2 + \nu_m^2) t}]$

17) $B_{n0} = 2 \int_1^2 \int_0^1 (u_2|_{t=0} - u_{ss2}) \cos(\lambda_n y) dy dx$

18) $B_{nm} = 4 \int_1^2 \int_0^1 (u_2|_{t=0} - u_{ss2}) \cos(\lambda_n y) \cos\left(\nu_m (x-1)\right) dy dx$

19) $\mu_l = \pi \left(l - \frac{1}{2} \right),\ \lambda_n = \pi \left(n - \frac{1}{2} \right),\ \nu_m = m\pi$ applicable to equations 13 to 18

APPENDIX B – PYTHON CODE

```
import math
import pytest
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

def volume_list_approx(x_list, y_list, z_list, x_f=lambda x: 1,
                       y_f=lambda y: 1):
    # used for approximating double integrals with 2D arrays as a basis
    total = 0
    for j in range(1, len(y_list)):
        for i in range(1, len(x_list[0])):
            area = (x_list[j][i] - x_list[j][i-1]) * (y_list[j][i] - y_list[j-1][i])
            height = (
                (z_list[j][i]*x_f(x_list[j][i])*y_f(y_list[j][i])) +
                (z_list[j][i-1]*x_f(x_list[j][i-1])*y_f(y_list[j][i-1])) +
                (z_list[j-1][i]*x_f(x_list[j-1][i])*y_f(y_list[j-1][i])) +
                (z_list[j-1][i-1]*x_f(x_list[j-1][i-1])*y_f(y_list[j-1][i-1]))
                ) / 4
            total += area * height
    return total

def area_list_approx(x_l, y_l, z_list, x_f=lambda x: 1, y_f=lambda y: 1,
                     type='x'):
    # used for approximating single integrals with 1D lists as a basis
    total = 0
    if type == 'x':
        for i in range(1, len(x_l)):
            base = x_l[i] - x_l[i-1]
            height = (z_list[i]*x_f(x_l[i]) + z_list[i-1]*x_f(x_l[i-1])) / 2
            total += base*height
    elif type == 'y':
        for j in range(1, len(y_l)):
            base = y_l[j] - y_l[j-1]
            height = (z_list[j]*y_f(y_l[j]) + z_list[j-1]*y_f(y_l[j-1])) / 2
            total += base*height
    return float(total)

def approx_integral(start, stop, function, step=0.01):
    # used for approximating single integrals
    x = start + step
    y0 = function(start)
    area = 0.0
    while x <= stop:
        y1 = function(x)
        area += (y0 + y1)/2*step
        y0 = y1
        x += step
    return area

def single_summation(x, y, u_list, constant_list, x_f=lambda x,u: 1,
                     y_f=lambda y,u: 1):
    # for single summations that contain separate functions of x and y
    # that both depend on u_l
    # for a given set of x and y
    # constant_list is a_l, b_l, and B_n0
    # (should account for e^(stuff) as necessary)
    total = 0
    for l in range(len(u_list)):
        total += x_f(x, u_list[l]) * y_f(y, u_list[l]) * constant_list[l]
    return total

def double_summation(x, y, nu_list, lam_list, constant_array,
                     x_f=lambda x,nu: 1, y_f=lambda y,lam: 1):
    # for double summations that contain separate functions of
    # x (depending on nu) and y (depending on lam)
    # for a given set of x and y
    # constant_array should account for A_nm, B_nm, and e^(stuff) as necessary
    total = 0
    for n in range(len(lam_list)):
        for m in range(len(nu_list)):
            total += x_f(x, nu_list[m]) * y_f(y, lam_list[n]) * constant_array[n][m]
    return total

def update(zz1, zz2, t):
    y11 = lambda y: 2 / math.cosh(-u) * math.cos(u*y) * y_bound(y)
    y12 = lambda y: 2*k / (u*math.cosh(u)) * math.cos(u*y)
    y13 = lambda y, u: math.cos(u*y)
    x13a = lambda x, u: math.cosh(u*(x-1))
    x13b = lambda x, u: math.sinh(u*x)
    y14 = lambda y: math.cos(lam*y)
    x14 = lambda x: math.sin(nu*x)
    y15 = lambda y, lam: math.cos(lam*y)
    x15 = lambda x, nu: math.sin(nu*x)

    y21 = lambda y: 4*k * math.cos(u*y) / (u * math.sinh(-u))
    y22 = lambda y, u: math.cos(u*y)
    x22 = lambda x, u: math.cosh(u*(x-2))
    y23 = lambda y: math.cos(lam*y)
    x23a = lambda x: math.cos(nu*(x-1))
    y24 = lambda y, lam: math.cos(lam*y)
    x24 = lambda x, nu: math.cos(nu*(x-1))

    a_l = []
    b_l = []
    y_l = []
    z_list = []
    for j in range(len(yy)):
        y_l.append(yy[j][-1])
        z_list.append(zz2[j][0] - zz1[j][-1])
    for u in u_list:
        a_l.append(approx_integral(0, 1, y11))
        b_l.append(area_list_approx(1, y_l, z_list, y_f=y12, type='y'))
    u_ssa = []
    for j in range(len(zz1)):
        u_ssa.append([])
        for i in range(len(zz1[j])):
            x = xx1[j][i]
            y = yy[j][i]
            u_ssa[j].append(single_summation(x, y, u_list, a_l, x13a, y13)
                + single_summation(x, y, u_list, b_l, x13b, y13))
    u_ssa = np.array(u_ssa)
    A_nm = []
    for n in range(len(lam_list)):
        lam = lam_list[n]
        A_nm.append([])
        for m in range(len(nu_list)):
            nu = nu_list[m]
            A_nm[n].append(volume_list_approx(xx1, yy, zz1 - u_ssa, x14, y14)
                * (4) * math.e**(-alpha1*((lam**2) + (nu**2))*t))
    Ua = []
    for j in range(len(zz1)):
        Ua.append([])
        for i in range(len(zz1[j])):
            x = xx1[j][i]
            y = yy[j][i]
            Ua[j].append(double_summation(x, y, nu_list, lam_list, A_nm,
                         x15, y15))
    Ua = np.array(Ua)

    c_l = []
    for u in u_list:
        c_l.append(area_list_approx(1, y_l, z_list, y_f=y21, type='y'))
    u_ssb = []
```



```
    for j in range(len(zz2)):
        u_ssb.append([])
        for i in range(len(zz2[j])):
            x = xx2[j][i]
            y = yy[j][i]
            u_ssb[j].append(single_summation(x, y, u_list, c_l, x22, y22))
    u_ssb = np.array(u_ssb)
    B_n0 = []
    B_nm = []
    for n in range(len(lam_list)):
        lam = lam_list[n]
        B_n0.append(volume_list_approx(xx2, yy, zz2 - u_ssb, y_f=y23)
                    * 2 * math.e**(-alpha2*(lam**2)*t))
        B_nm.append([])
        for m in range(len(nu_list2)):
            nu = nu_list2[m]
            B_nm[n].append(volume_list_approx(xx2, yy, zz2 - u_ssb, x23a, y23)
                    * 4 * math.e**(-alpha2*((lam**2) + (nu**2))*t))
    Ub0 = []
    Ub = []
    for j in range(len(zz2)):
        Ub.append([])
        Ub0.append([])
        for i in range(len(zz2[j])):
            x = xx2[j][i]
            y = yy[j][i]
            Ub[j].append(double_summation(x, y, nu_list2, lam_list, B_nm,
                                x24, y24))
            Ub0[j].append(single_summation(1, y, lam_list, B_n0, y_f=y24))
    Ub0 = np.array(Ub0)
    Ub = np.array(Ub)

    return u_ssa + Ua, u_ssb + Ub0 + Ub

class gridSave:
    # used purely for saving data in between frames
    def __init__(self, grid):
        self.data = grid


def animateFunc(frame):
    # this gets called at every frame of the animation
    ax1.cla()
    ax2.cla()
    ax3.cla()
    ax4.cla()
    if frame == 0:
        #code redacted here because of similarity to other code sections
        #mostly just formatting for graphs
    # for ax1 with change in time as 0.05s
    t = 0.05
    zz01_1.data, zz01_2.data = update(zz01_1.data, zz01_2.data, t)
    ax1.plot_surface(xx1, yy, zz01_1.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    ax1.plot_surface(xx2, yy, zz01_2.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    # for ax2 with change in time as 0.025s
    t = 0.025
    for i in range(2):
        zz001_1.data, zz001_2.data = update(zz001_1.data, zz001_2.data, t)
    ax2.plot_surface(xx1, yy, zz001_1.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    ax2.plot_surface(xx2, yy, zz001_2.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    # for ax3 with change in time as 0.01s
    t = 0.01
    for i in range(5):
        zz0001_1.data, zz0001_2.data = update(zz0001_1.data, zz0001_2.data, t)
    ax3.plot_surface(xx1, yy, zz0001_1.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    ax3.plot_surface(xx2, yy, zz0001_2.data, cmap=cm.coolwarm,
                     vmin=-1, vmax=51)
    # for plotting ax4 based on data from the other plots at y=0
    ax4.plot(xx1[0], zz01_1.data[0], color='black')
    ax4.plot(xx2[0], zz01_2.data[0], color='black')
    ax4.plot(xx1[0], zz001_1.data[0], color='red')
    ax4.plot(xx2[0], zz001_2.data[0], color='red')
    ax4.plot(xx1[0], zz0001_1.data[0], color='green')
    ax4.plot(xx2[0], zz0001_2.data[0], color='green')
    ax4.set_ylim(-1, 51)
        #other sections of code redacted as they are purely formatting for graphs



fps = 20
time = 2
step = 0.05
k = 0.5
alpha1 = 0.4
alpha2 = 2
y_bound = lambda y: 50
terms = 20

eigen_u_l = lambda l: math.pi*(l-0.5)
eigen_lam_n = lambda n: math.pi*(n-0.5)
eigen_nu_m1 = lambda m: math.pi*(m-0.5)
eigen_nu_m2 = lambda m: math.pi*m
u_list = [eigen_u_l(l) for l in range(1, terms+1)]
lam_list = [eigen_lam_n(n) for n in range(1, terms+1)]
nu_list = [eigen_nu_m1(m) for m in range(1, terms+1)]
nu_list2 = [eigen_nu_m2(m) for m in range(1, terms+1)]

xx1 = np.arange(0, 1 + step/2, step)
yy = np.arange(0, 1 + step/2, step)
xx1, yy = np.meshgrid(xx1, yy)
xx2 = xx1 + 1
zz01_1 = gridSave(yy * 0)
zz01_2 = gridSave(zz01_1.data + 0)
zz001_1 = gridSave(yy * 0)
zz001_2 = gridSave(zz001_1.data + 0)
zz0001_1 = gridSave(yy * 0)
zz0001_2 = gridSave(zz0001_1.data + 0)

Fig = plt.figure(figsize=(16, 12))
ax1 = Fig.add_subplot(2, 2, 1, projection = '3d')
ax2 = Fig.add_subplot(2, 2, 2, projection = '3d')
ax3 = Fig.add_subplot(2, 2, 3, projection = '3d')
ax4 = Fig.add_subplot(2, 2, 4)
Fig2, ((ax5, ax6), (ax7, ax8)) = plt.subplots(2, 2)
Fig.subplots_adjust(wspace = 0.5, hspace = 0.5)
Fig2.subplots_adjust(wspace = 0.5, hspace = 0.5)
vid = FuncAnimation(Fig, animateFunc, frames=fps*time+1,
                    interval=1000/fps)
vid.save(f'Project{k}.gif')
Fig2.savefig(f'k{k}.png')
```

