Question 1 Consider the heat equation, for $0 \le x \le 1$, $0 \le t \le 1$,

$$\begin{cases} u_t = 2u_{xx}, \\ u(x,0) = 2cosh(x), \\ u(0,t) = 2e^{2t}, \\ u(1,t) = (e^2+1)e^{2t-1}. \end{cases}$$

Part 1

Verify u(x, y) = x^2+ y^2 is the exact solution.

Plug the boundary conditions into u(x, y) to get:

$$u(x,1) = x^2 + (1)^2$$
$$u(x,2) = x^2 + (2)^2$$
$$u(3,y) = (3)^2 + y^2$$
$$u(5,y) = (5)^2 + y^2$$

Which all match the boundary conditions of our system.
Then, define Laplace of u with U_{xx} and U_{yy}, and plug it back into the first equation of our system to get:

$$u_{xx} = 2, u_{yy} = 2$$
$$u_{xx} + u_{yy} + \frac{u}{x^2+y^2} = 5$$
$$2 + 2 + \frac{x^2+y^2}{x^2+y^2} = 5$$
$$2+2+1 = 5$$

As long as x^2 + y^2 is not equal to 0 which is always true on and within our boundary. Thus, x^2+ y^2 is the exact solution to our system.

Coding Setup

First, I defined the exact solution to the system (u) and boundary functions (u_t, u_b, u_l, u_r for the top, bottom, left, and right, respectively). The code for these is as follows:

```python
def u(x, y):
   return x**2 + y**2


def u_l(y):
   return y**2 + 9


def u_r(y):
   return y**2 + 25


def u_b(x):
   return x**2 + 1


def u_t(x):
   return x**2 + 4
```

Additionally, I defined an indexing function, a way to get an (i, j) pair from the index, and a function to get the (x, y) coordinates from a given (i, j) pair. The code for these is as follows:

```python
def index(i, j, m, n):
   return i + j*m


def get_ij_from_index(ind, m, n):
   return ind % m, ind//m


def get_xy_from_ij(i, j, x_min, h_x, y_min, h_y):
   return x_min + i * h_x, y_min + j * h_y
```

Part 2

The code for the Finite Difference method is as follows:

```python
def finite_difference(x_min, x_max, y_min, y_max, hx, hy, top, bottom, left, right,
r, f):
   # find the number of divisions in x and y
   m = int((x_max - x_min)/hx + 1)
   n = int((y_max - y_min)/hy + 1)

   size = n * m
```

```
    # initialize our sparse matrix
    a = sp.lil_matrix((size, size))
    b = np.zeros((size, 1))
    for i in range(0, m):
        for j in range(0, n):

            ij = index(i, j, m, n)
            x, y = get_xy_from_ij(i, j, x_min, hx, y_min, hy)

            # left boundary
            if i == 0:
                a[ij, ij] = 1
                b[ij, 0] = left(y)

            # right boundary
            elif i == m-1:
                a[ij, ij] = 1
                b[ij, 0] = right(y)

            # bottom boundary
            elif j == 0:
                a[ij, ij] = 1
                b[ij, 0] = bottom(x)

            # top boundary
            elif j == n-1:
                a[ij, ij] = 1
                b[ij, 0] = top(x)

            # handles the interior points
            else:
                # find the indexes for the points above, below, left and right of
 our current point
                ij_up = index(i, j+1, m, n)
                ij_down = index(i, j-1, m, n)
                ij_left = index(i-1, j, m, n)
                ij_right = index(i+1, j, m, n)

                # calculate the coefficients of the nearby points
                a[ij, ij] = -2 * (1/hx**2 + 1/hy**2) + r(x, y)
                a[ij, ij_down] = 1/hy**2
                a[ij, ij_up] = 1/hy**2
                a[ij, ij_left] = 1/hx**2
                a[ij, ij_right] = 1/hx**2

                # get our right hand side
                b[ij, 0] = f(x, y)
    # change the matrix form and solve
    a = sp.csr_matrix(a)
    u_res = np.array([splinalg.spsolve(a, b)]).T

    return u_res
```

The basic flow of this method is:

- Determine factors of the area and the matrix (number of steps in x and y, the size of the matrix), and initialize the matrices
- For each point in the region:
    - If the point is on a boundary, set the value in the point's column and row to be 1 and determine the right hand side with the relevant boundary
    - Otherwise, determine the coefficients of the 5 points in a cross around our point and the right hand side of that point.
- Solve the matrix equation and return the solution.

Figures 1 through 8 show the average of the absolute error resulting from this method for various values of h and k.

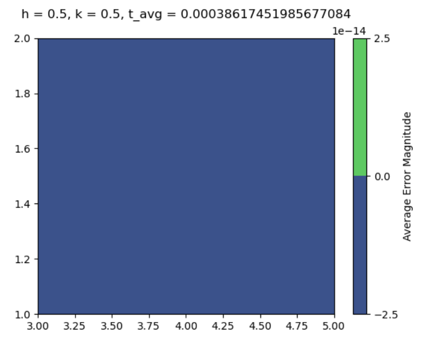


Figure 1 - h = 0.5

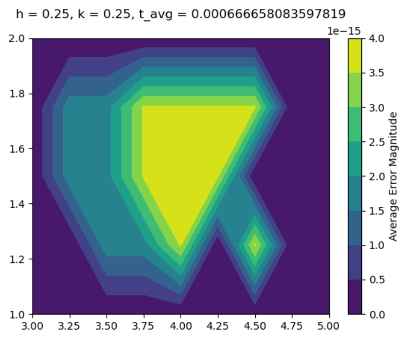


Figure 2 - h = 0.25

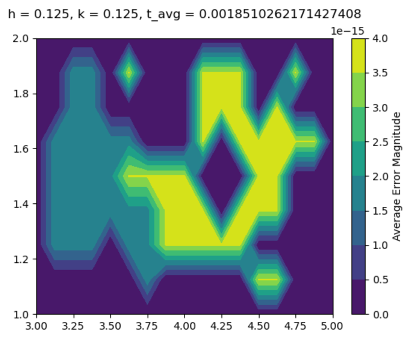


Figure 3 - h = 0.125

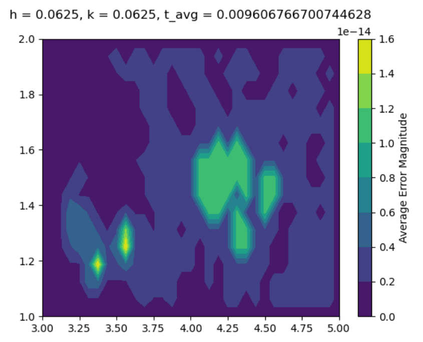


Figure 4 - h = 0.0625

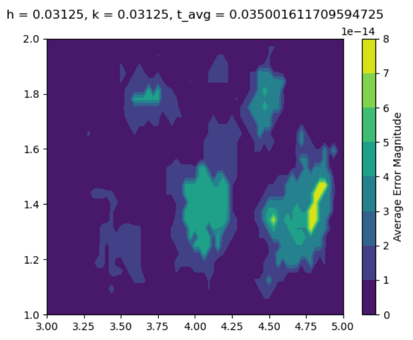


Figure 5 - h = 0.03125

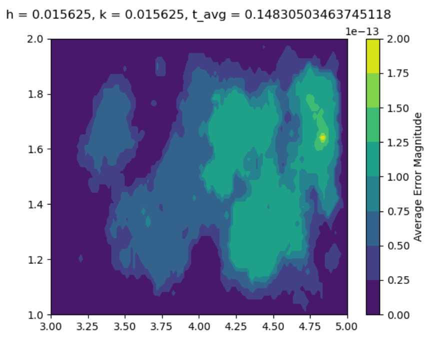
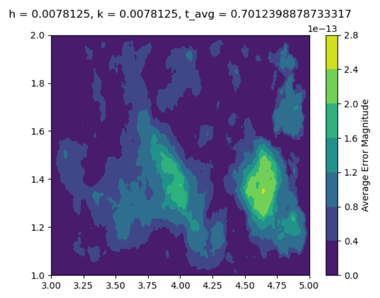


Figure 6 - h = 0.015625



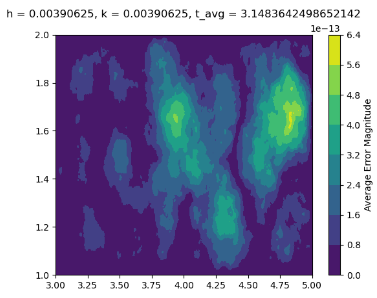


Figure 7 - h = 0.0078125

Figure 8 - h = 0.00390625

Plotting the mean of the absolute error for various values of h and k (Figure 9) reveals an odd trend. The error increases as h and k get smaller. The solution is almost certainly, if worked out by hand, an exact solution for the system, but the number of calculations causes the lack of precision due to machine precision to be magnified, causing the error to reach a few orders of magnitude higher than where it started.

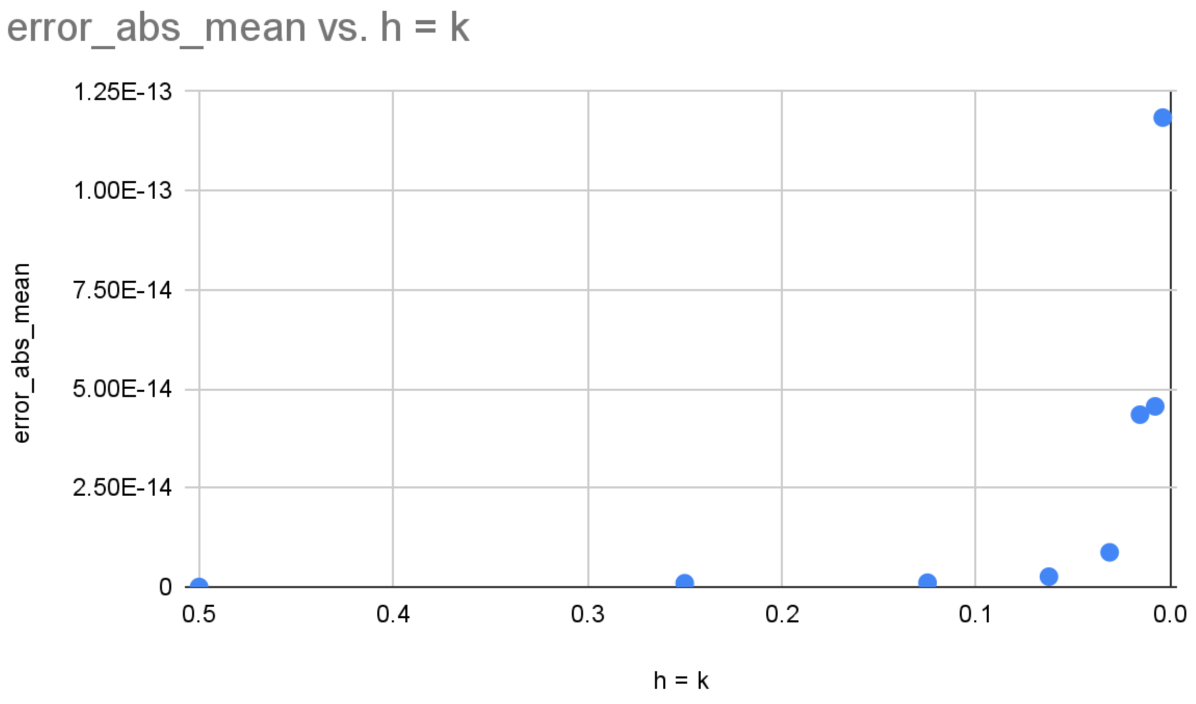


Figure 9 - Mean of the absolute error of Finite Difference against h

Finite Differences should be able to calculate an exact model of the system because the error resulting from the approximation of u_i'' as $(u_{i-1} - 2u_i + u_{i+1})/h^2$ is dependent on the fourth derivative, which is 0 for our functions. As such, there should be no error for this particular approach to solving the system, beyond machine precision.

The time cost increases drastically as h and k increase, since there are more and more slices over which to calculate. Figure 10 shows how the average time of 30 trials increases as h and k decrease.

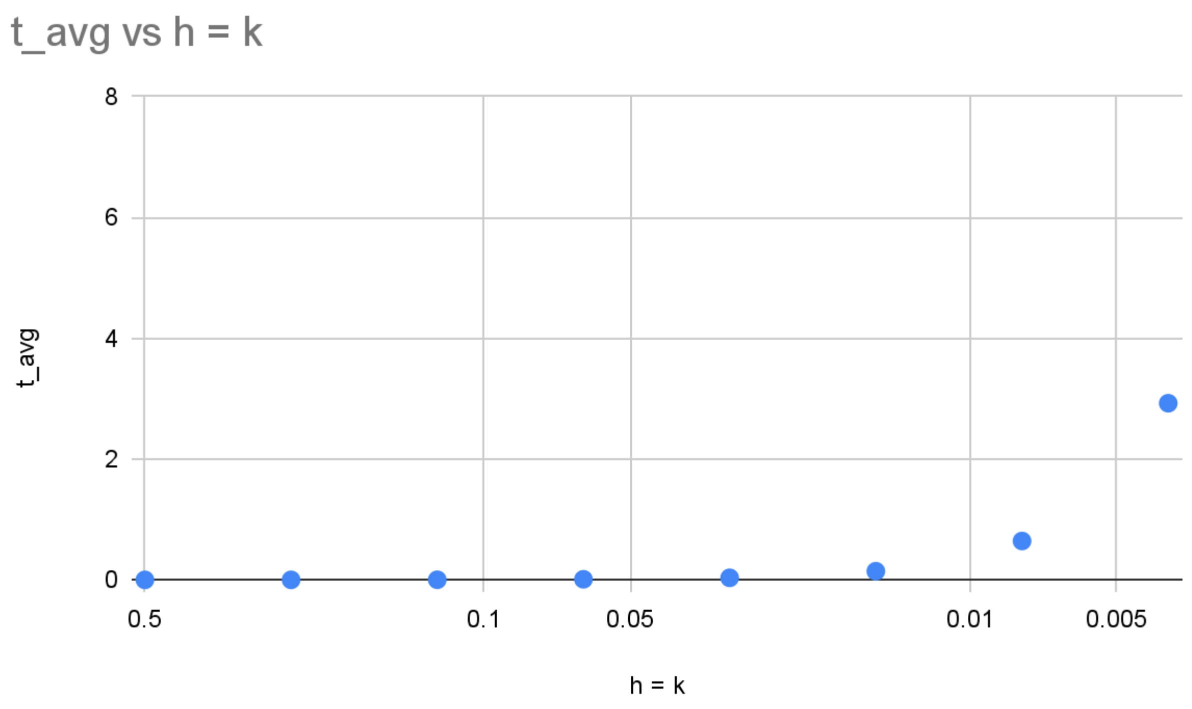


Figure 10 - Average time for Finite Difference method against values of h

Part 3

The code for the Finite Element method is as follows:

```
def finite_element(x_min, x_max, y_min, y_max, hx, hy, top, bottom, left, right, r,
f):
   # initialize size of bounds and matrices
   m = int((x_max - x_min) / hx + 1)
   n = int((y_max - y_min) / hy + 1)

   size = n * m

   # initialize a and b
   a = sp.lil_matrix((size, size))
   b = np.zeros((size, 1))
   for i in range(0, m):
       for j in range(0, n):

           # get current index and the coordinates
```

```python
        ij = index(i, j, m, n)
        x, y = get_xy_from_ij(i, j, x_min, hx, y_min, hy)

        # left boundaries
        if i == 0:
            a[ij, ij] = 1
            b[ij, 0] = left(y)

        # right boundaries
        elif i == m - 1:
            a[ij, ij] = 1
            b[ij, 0] = right(y)

        # bottom boundaries
        elif j == 0:
            a[ij, ij] = 1
            b[ij, 0] = bottom(x)

        # top boundaries
        elif j == n - 1:
            a[ij, ij] = 1
            b[ij, 0] = top(x)

        else:
            # handle interior points
            # calculate the 6 barycenters surrounding our point
            bcenters = [[x + 2 / 3 * hx, y + 1 / 3 * hy],
                         [x + 1 / 3 * hx, y + 2 / 3 * hy],
                         [x - 1 / 3 * hx, y + 1 / 3 * hy],
                         [x - 2 / 3 * hx, y - 1 / 3 * hy],
                         [x - 1 / 3 * hx, y - 2 / 3 * hy],
                         [x + 1 / 3 * hx, y - 1 / 3 * hy]]

            # calculate the value of the r function for each of the barycenters
            r_barys = np.array([r(*bar) for bar in bcenters])

            # find indices of the 6 nearby points
            ij_up = index(i, j + 1, m, n)
            ij_up_right = index(i + 1, j + 1, m, n)
            ij_down = index(i, j - 1, m, n)
            ij_down_left = index(i - 1, j - 1, m, n)
            ij_left = index(i - 1, j, m, n)
            ij_right = index(i + 1, j, m, n)

            # calculate the area of the triangle
            area_c = hx * hy / 2
```

```
                # give coefficients to the 7 points surrounding our specific point
                a[ij, ij] = 2*(hx**2 + hy**2)/(hx * hy) - r_barys.sum() * area_c/9
                a[ij, ij_down] = -hx/hy - (r_barys[4] + r_barys[5]) * area_c/9
                a[ij, ij_down_left] = -(r_barys[3] + r_barys[4]) * area_c/9
                a[ij, ij_up] = -hx/hy - (r_barys[1] + r_barys[2]) * area_c/9
                a[ij, ij_up_right] = -(r_barys[0] + r_barys[1]) * area_c/9
                a[ij, ij_left] = -hy/hx - (r_barys[2] + r_barys[3]) * area_c/9
                a[ij, ij_right] = -hy/hx - (r_barys[5] + r_barys[0]) * area_c/9

                # (sum of f(barycenter)) * (phi(barycenter) = 1/3) * (area of
triangle = hx*hy/2)
                f_sum = np.array([f(*bar) for bar in bcenters]).sum()
                b[ij, 0] = -f_sum * area_c / 3
    # solve the system
    a = sp.csr_matrix(a)
    u_res = np.array([splinalg.spsolve(a, b)]).T
    # return
    return u_res
```

The basic flow of this method is:

- Determine factors of the area and the matrix (number of steps in x and y, the size of the matrix), and initialize the matrices
- For each point in the region:
    - If the point is on a boundary, set the value in the point's column and row to be 1 and determine the right hand side with the relevant boundary
    - Otherwise:
        - Calculate the barycenters of the 6 triangles surrounding our point, the values of r(x, y) [in this case $1/(x^2 + y^2)$] for each barycenter.
        - Determine the coefficients of the 7 points in a cross around our point
        - Determine the right hand side of that point.
- Solve the matrix equation and return the solution.

Figures 11 through 18 show the average of the absolute error resulting from this method for various values of h and k.

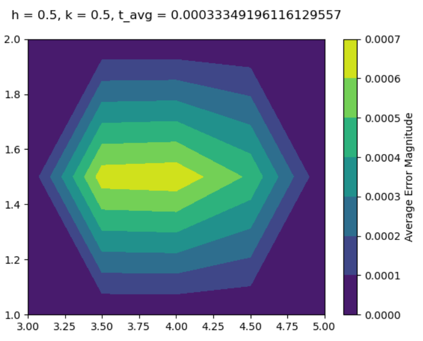


Figure 11 - h = 0.5

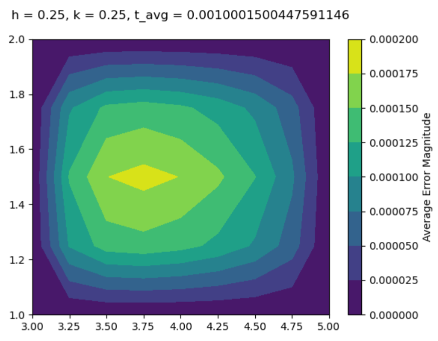


Figure 12 - h = 0.25

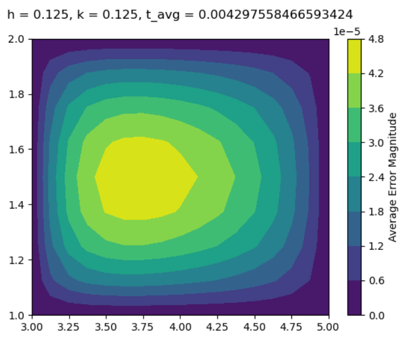


Figure 13 - h = 0.125

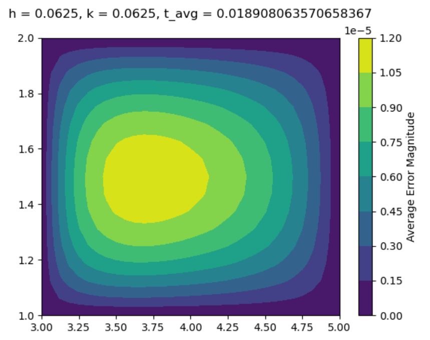


Figure 14 - h = 0.0625

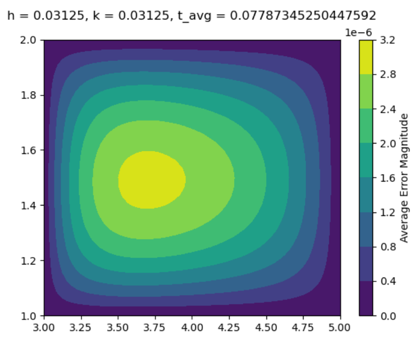


Figure 15 - h = 0.03125

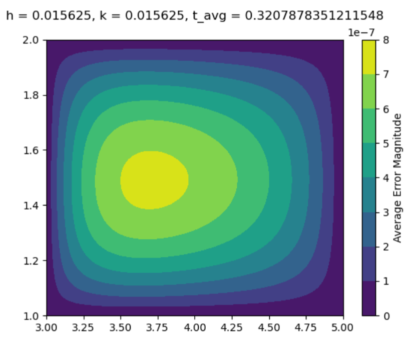


Figure 16 - h = 0.015625

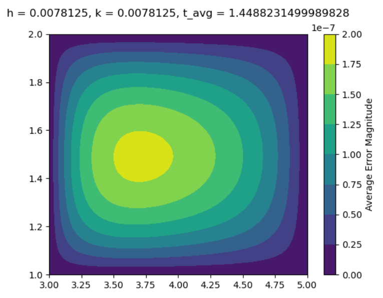


Figure 17 - h = 0.0078125

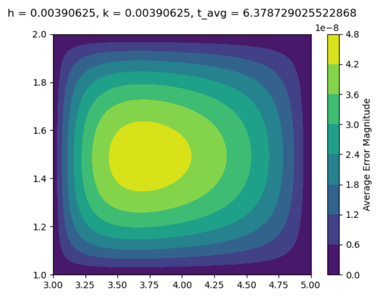


Figure 18 - h = 0.00390625

Unlike with the Finite Difference method, this problem has an intuitive curve as the number of steps increases. Figure 19 shows the mean of the absolute error against many values of h and k, and has a log scale for the y axis to exaggerate the differences and show a truer pattern of the trend of improvement.

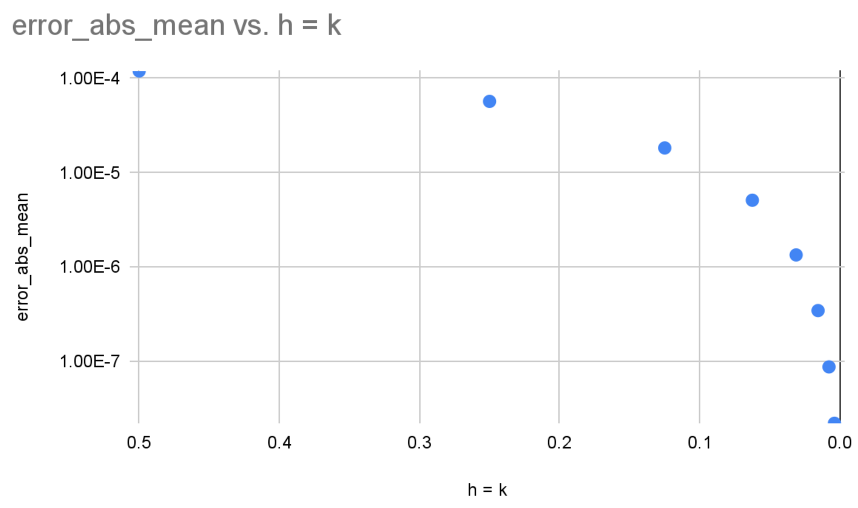


Figure 19 - Mean of the absolute error of Finite Difference against h

The time cost for Finite Elements also increases drastically as h and k increase, since there are more and more slices over which to calculate. Figure 20 shows how the average time of 30 trials increases as h and k decrease.

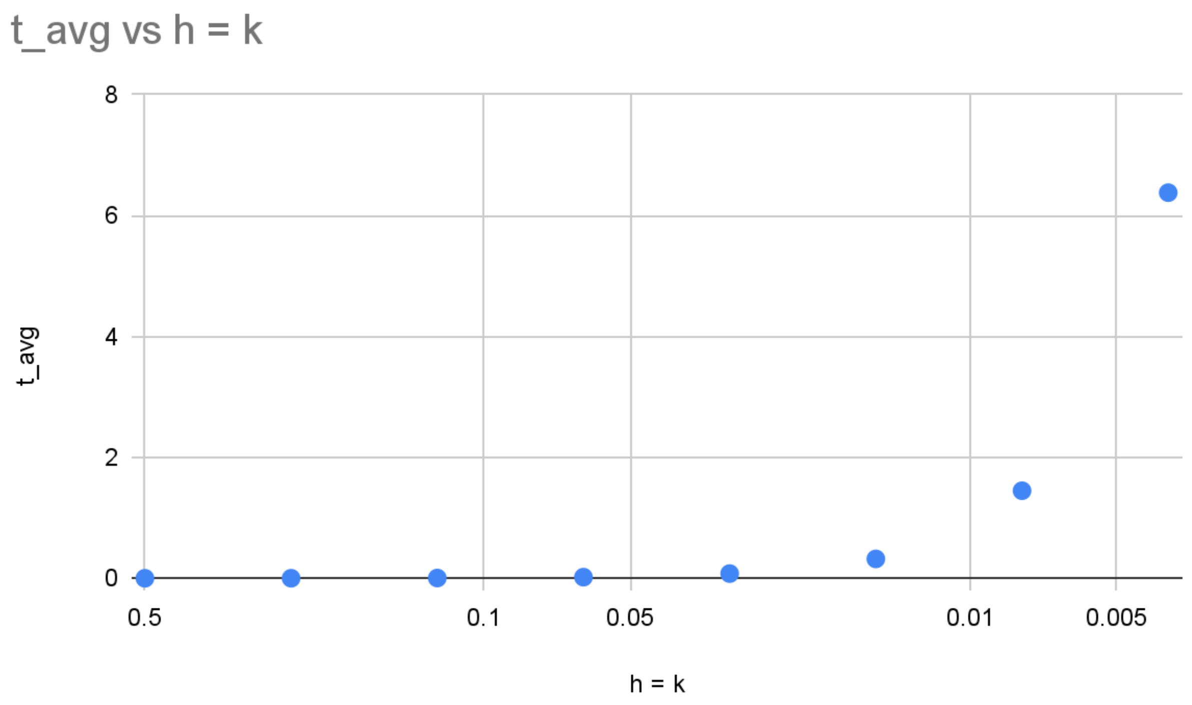


Figure 20 - Average time for Finite Difference method against values of h

Notably, the time it takes to solve later iterations of Finite Elements reaches much higher than Finite Differences, eventually reaching over double the time it takes for Finite Differences to solve. Finite Element introduces a lot more calculations, as calculating both r and f for each of our barycenters is required. However, this method also shows a promising trend of improvement for this problem. It likely is not worth the additional computational time though.

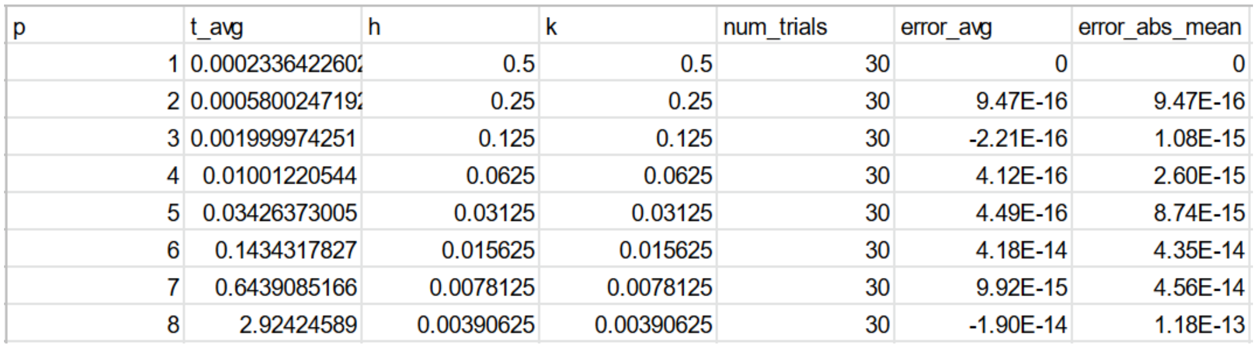
Part 4

For this particular system, the Finite Differences Method is more well-suited to solving the problem, in terms of accuracy, speed, and even space. Finite Elements requires storing 7 values per interior point, while Finite Difference only requires 5. Additionally, each interior point requires Finite Elements to calculate and store 12 values, 6 values of r at the barycenters and 6 values of f at the barycenters, though these are less expensive in the long run since they are discarded after each iteration.

There may be many systems which Finite Element is more suited towards, however. The time cost may be more expensive, but the accuracy will probably reflect that. This system, since it had a fourth derivative equal to 0, was easily solved by Finite Difference, but Finite Element was also promising and was continuing its path of improvement quickly as h and k lessened.

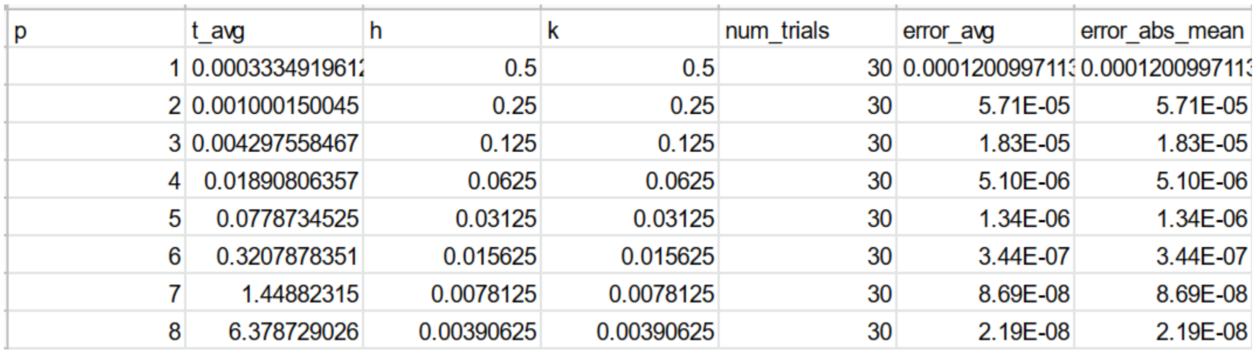
When speed is favored over accuracy, using the Finite Difference method would prove useful, many situations may call for Finite Element instead.

Details about the trial data (grouped by h and k values), can be found below in Figures 21 and 22.



p	t_avg	h	k	num_trials	error_avg	error_abs_mean
1	0.0002336422602	0.5	0.5	30	0	0
2	0.0005800247192	0.25	0.25	30	9.47E-16	9.47E-16
3	0.001999974251	0.125	0.125	30	-2.21E-16	1.08E-15
4	0.01001220544	0.0625	0.0625	30	4.12E-16	2.60E-15
5	0.03426373005	0.03125	0.03125	30	4.49E-16	8.74E-15
6	0.1434317827	0.015625	0.015625	30	4.18E-14	4.35E-14
7	0.6439085166	0.0078125	0.0078125	30	9.92E-15	4.56E-14
8	2.92424589	0.00390625	0.00390625	30	-1.90E-14	1.18E-13

Figure 21 - Trial Data from Finite Difference Method



p	t_avg	h	k	num_trials	error_avg	error_abs_mean
1	0.0003334919612	0.5	0.5	30	0.0001200997113	0.0001200997113
2	0.001000150045	0.25	0.25	30	5.71E-05	5.71E-05
3	0.004297558467	0.125	0.125	30	1.83E-05	1.83E-05
4	0.01890806357	0.0625	0.0625	30	5.10E-06	5.10E-06
5	0.0778734525	0.03125	0.03125	30	1.34E-06	1.34E-06
6	0.3207878351	0.015625	0.015625	30	3.44E-07	3.44E-07
7	1.44882315	0.0078125	0.0078125	30	8.69E-08	8.69E-08
8	6.378729026	0.00390625	0.00390625	30	2.19E-08	2.19E-08

Figure 22 - Trial Data from Finite Element Method