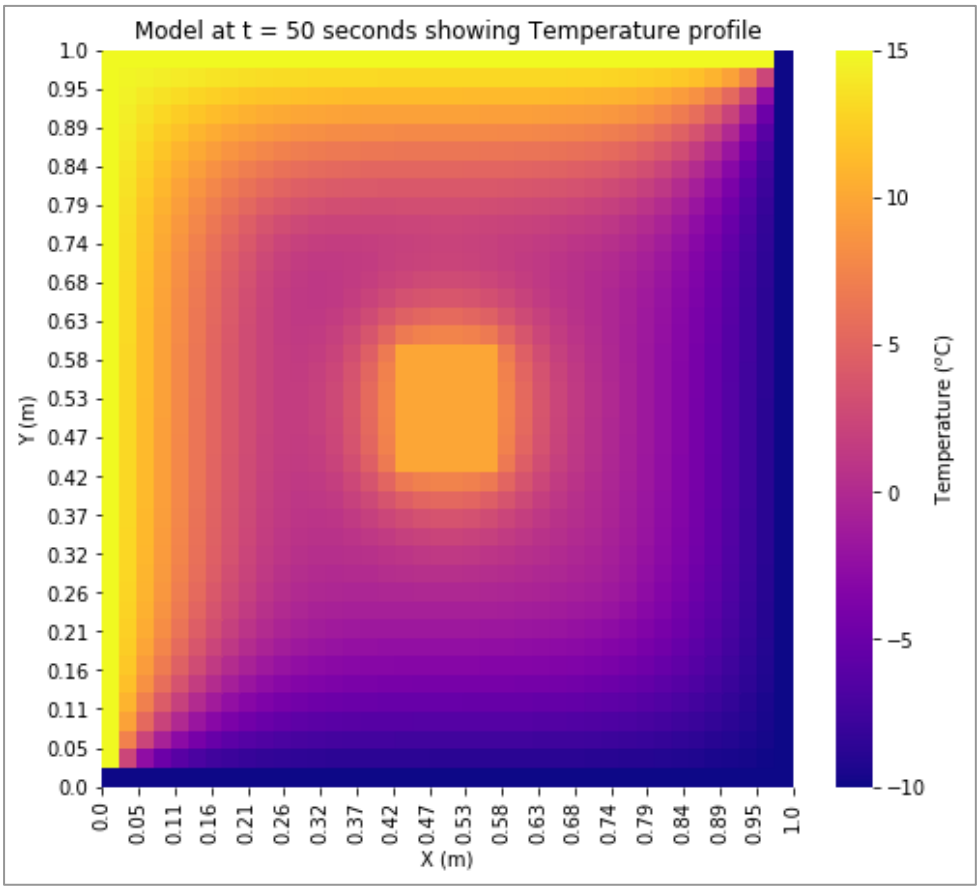

2D Heat Conduction Model using Python

Erin Baker

Summary:

For the last two months in my spare time I have been working on creating a model in python that simulates heat conduction in a 2D plane of material using python. My goal for this was to refresh my knowledge of my degree whilst on placement, as well as develop my coding and modelling skills. I have future plans to develop this model further in the future to include other modes of heat transfer as well as aspects of mass transfer.

For a basis of the model I looked at a model uploaded to Mathworks file exchange by a student Amr Mohamed[1]. While I could have simply copied the code into python and done the same calculation, instead I wanted to make sure I understood the numerical methods and calculations so I could develop the model further.

The model I wrote allows for multiple materials to be used in calculation, calculates a steady state profile, can calculate time to steady state, includes additional check on error, among other additions.

Overview of Model:

Structure of discretized model in python:

The sheet of material which I am modelling has been discretized into finite areas. With future addition of mass transfer, and possibly calculating each physical property on each iteration, I created the discretized area in python in the form of a NumPy array of dictionaries.

This means that I do not need numerous arrays storing each property associated with each discretized area. Instead there is a single array where any property in any position, in any calculated time, can easily be pulled out. I named this array 'topology' and below is an example of how this structure works.

```
Dictionary found in array cell:

{'T':25,'kT': 0,'cp':0,'rho':0,'alpha':0,'BC':False}

Cell reference notation:

topology[3,1,1] returns {'T':25,'kT': 0,'cp':0,'rho':0,'alpha':0,'BC':False}

topology[3,1,1]['T'] returns 25     (indexing individual property/value)
```

[1] 2D heat equation using FDM – Amr Mousa Mohamed
https://uk.mathworks.com/matlabcentral/fileexchange/55058-2d-heat-equation-using-finite-difference-method-with-steady-state-solution

Steady State Calculation:

Whilst the reference material was useful for getting set-up, it does not explain where the equations used comes from, nor how they are derived. I was able to find plenty of online tutorials for Eulers method and 2nd and 4th order Runge-Kutta methods so I was happy with those equations given. However, because the steady state equation simply appears in their code without explanation of where it came from- I decided to derive it myself. A full derivation can be found in the appendix in hand-written form.

Steady State Equation:

$$T_{i,j}^{k+1} = \frac{\Delta x^2 \cdot \left(T_{i+1,j}^{k} + T_{i-1,j}^{k}\right) + \Delta y^2 \cdot (T_{i,j+1}^{k} + T_{j-1}^{k})}{2 \cdot (\Delta x^2 + \Delta y^2)}$$

Where *k* represents the current iteration of calculation. The values of *i* and *j* are the index positions of the discretized array. This equation is put in a loop to calculate the temperature profile until a user-defined tolerance is met.

The time to reach steady state in the reference MATLAB model is calculated as: *total iterations x time interval (dt)*. I was not happy with this approach because it wrongly assumes that each interval of the steady state equation is equivalent to a step size of *dt*.

Instead, I calculated time to reach steady state by placing a tolerance control around the calculations for the dynamic state, such that when the greatest and minimum temperatures approach a user defined minimum difference to the steady state value- then the calculation stops. Then if this tolerance is met, the model multiplies the total number of dynamic iterations (each represent dt) with the time interval.

Dynamic State Calculation:

For calculating the temperature at any given moment in time (up until steady state is reached), either an Euler's approximation or 2nd order Runge-Kutta equation can be used. The equations for these have come from the reference MATLAB model[2]. However, as previously mentioned, I did not want to simply copy and paste the maths, so I some time learning about these methods online. Aside from maths tutorials on Euler's method and Runge-Kutta I found the following two videos rather helpful.

1. MATLAB Help – Finite Difference Method by Dr. Carlos Montalvo: youtu.be/EGFfQt0-zls
2. Heat Transfer L10 p1 - Solutions to 2D Heat Equation by Ron Hugo: youtu.be/lY2QVs7aFMY

The first is a tutorial on using the finite difference method (FDM) to model the heating of a metal rod. I followed this tutorial and made the example in python. The second video is part of a series of videos which I watched which helped provide a background understanding of modelling these systems.

When I was implementing Euler and RK methods into my code, I discovered a variety of ways in which the calculation can go wrong caused by a mistake- including ways the MATLAB code would not pick up. The MATLAB code only detects a non-converging calculation when the model is converging where $T_{i,j}^{k} \neq T_{i,j}^{s.s}$. However, I realised this does not detect an ever-increasing temperature caused by a miscalculation. So, I added an additional check to stop the calculation loop if any temperature goes above the highest temperature from the initial conditions.

Earlier when I showed the dictionary in each cell, there is a key called 'BC'. This entry I added in the dict to store a boolean value which identifies the cell as being a boundary condition or not. When the dynamic calculations are carried out, I have written them to ignore cells marked as being a boundary condition- even if they are not at the literal boundary of the model. This allows the model to simulate much more interesting shapes and patterns.

[2] 2D heat equation using FDM – Amr Mousa Mohamed
https://uk.mathworks.com/matlabcentral/fileexchange/55058-2d-heat-equation-using-finite-difference-method-with-steady-state-solution

Model examples and results:

Heat dissipation from centre:

In this example the only material used is aluminium, the boundary conditions are set to 0 °C, and in the centre, there is an area of high temperature with a boundary condition set to 'False'. The heat dissipates through the metal sheet as time progresses.

Model Parameters:

Length (m)	Width (m)	ΔT (s)	Total Time (s)	Steady State Tolerance (-)	Dynamic State Tolerance (-)
1	1	0.6	50	0.01	0.01

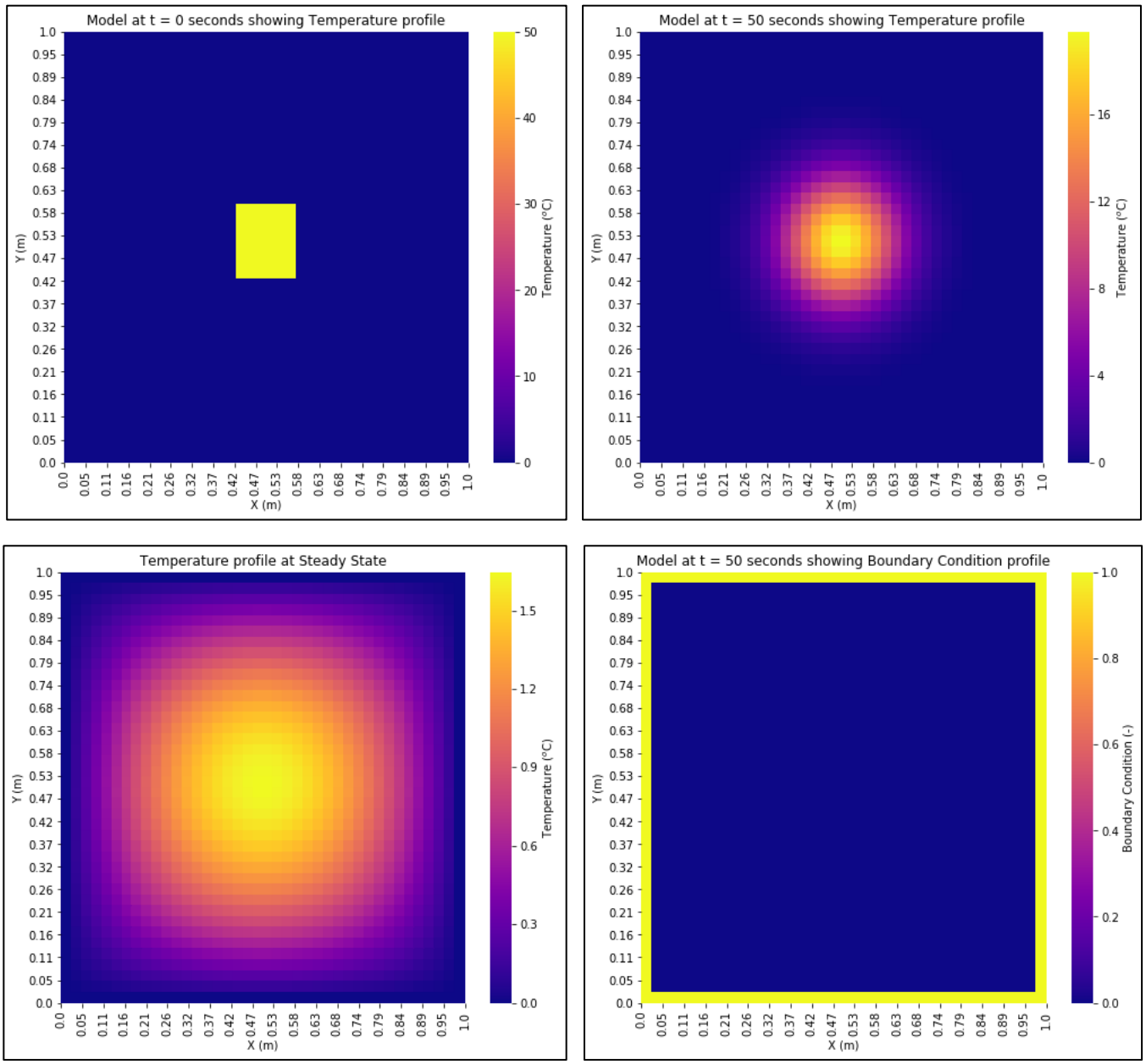

Figure 1) Temperature profile at: initial conditions (top left), and 50 seconds (top right) using Euler's method. Steady state temperature profile (bottom left). Boundary condition profile (bottom right).

Multiple materials and centre fixed temperature:

In this example I am using aluminium, copper, and silver in the model in concentric areas. In the centre of the model is an area with fixed temperature which is represented with a “Boundary Condition = True” in these cells. There are varying temperatures along the edge of the model.

Model Parameters:

Length (m)	Width (m)	ΔT (s)	Total Time (s)	Steady State Tolerance (-)	Dynamic State Tolerance (-)
1	1	0.6	50	0.01	0.01

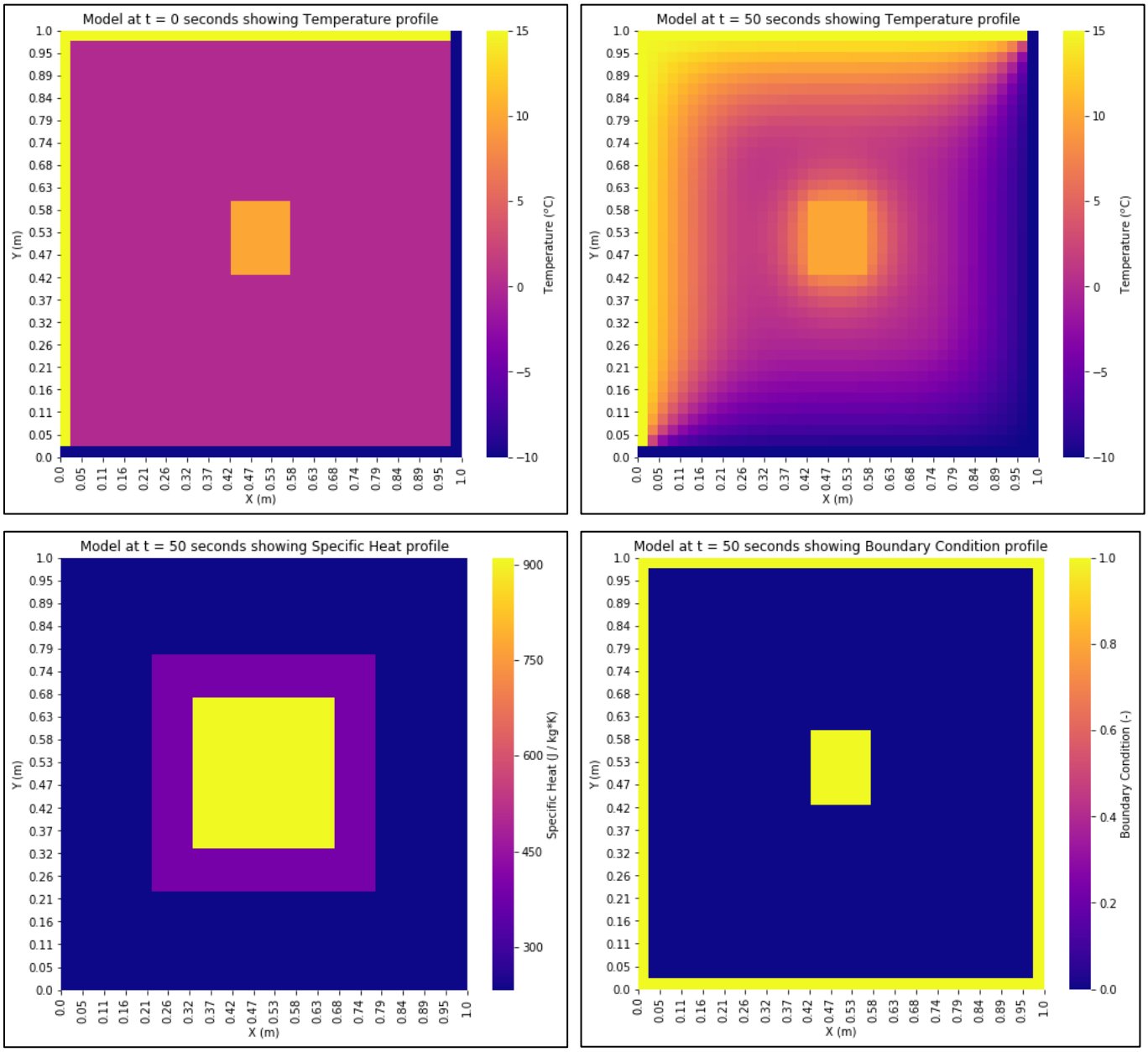

Figure 2) Temperature profile at: initial conditions (top left), 50 seconds (top right) using 2nd order Runge-Kutta method. Specific heat profile showing different materials (bottom left). Boundary condition profile (bottom right).

Calculating time to reach steady state temperatures:

Heat dissipation example:

Using the heat dissipation example shown earlier, I calculated a time to reach steady state by varying the tolerance on the dynamic calculation, I also increased the total time allowed to 1000 seconds. Every other model parameter was kept the same as shown previously (importantly dt = 0.06 and ss_tol = 0.01). The results calculated using Euler's method are shown below.

total_time (s)	dyn_tol (°C)	Error to SS (°C)	t_solve (s)
1000	1	1	456
1000	0.1	0.1	663
1000	0.01	0.009	693

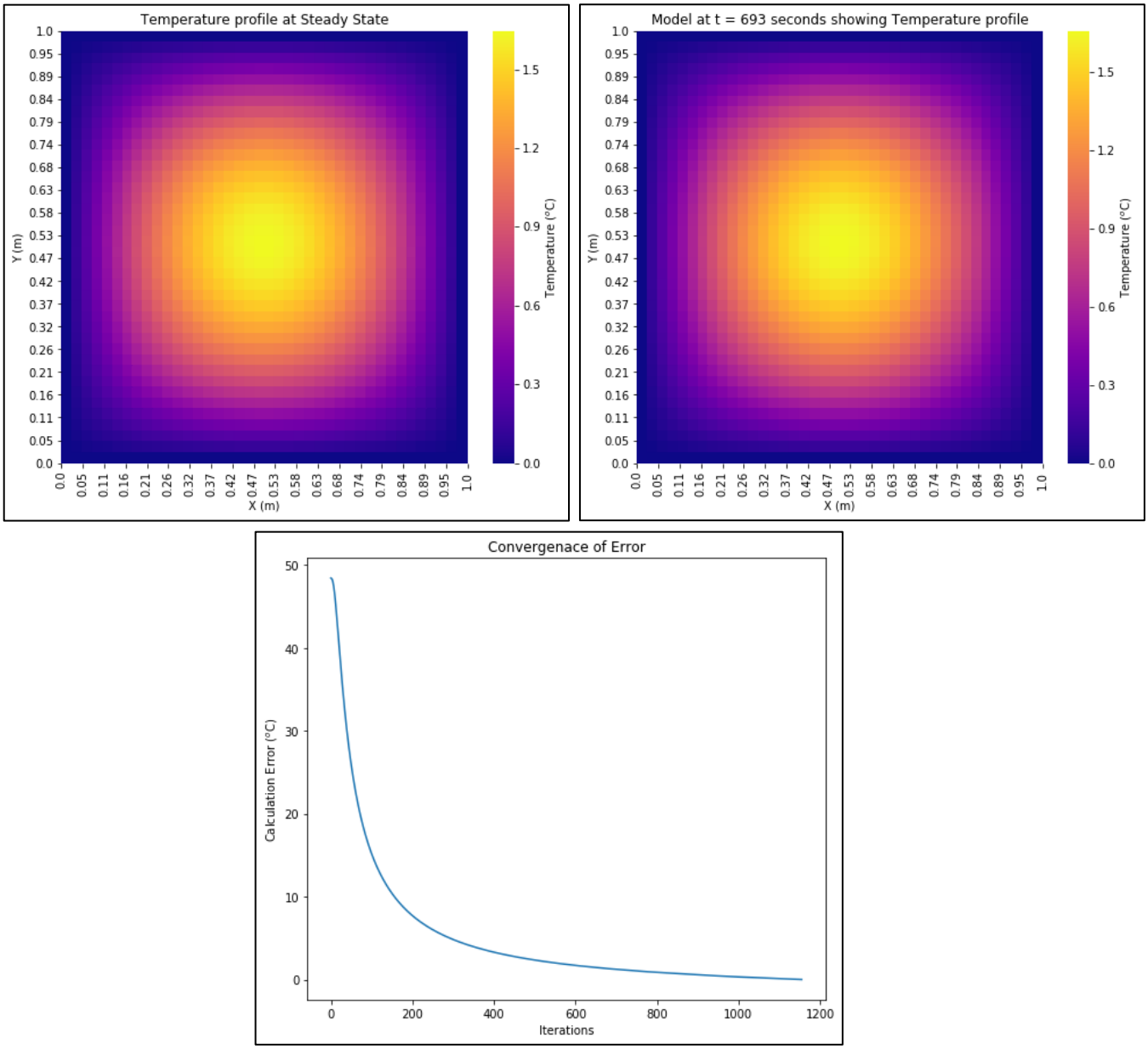

Figure 3) Temperatures at steady state (top left) compared with dynamic calculation stopped at steady state (top right) with a greatest difference of 0.01°C between profiles. Plot of difference between dynamic state and steady state calculation against dynamic iterations (bottom).

Multiple materials and fixed centre example:

Using the other previous example to calculate time to steady state. The model using Euler's converges at 0.25 °C above steady state. The calculation is stopped by an IF statement that checks for non-steady state convergence. If the IF statement is not in-place then this error begins to increase.

total_time (s)	dyn_tol (°C)	Error to SS (°C)	t_solve (s)
1000	1	0.999	266
1000	0.5	0.496	304
1000	0.01	0.248	336

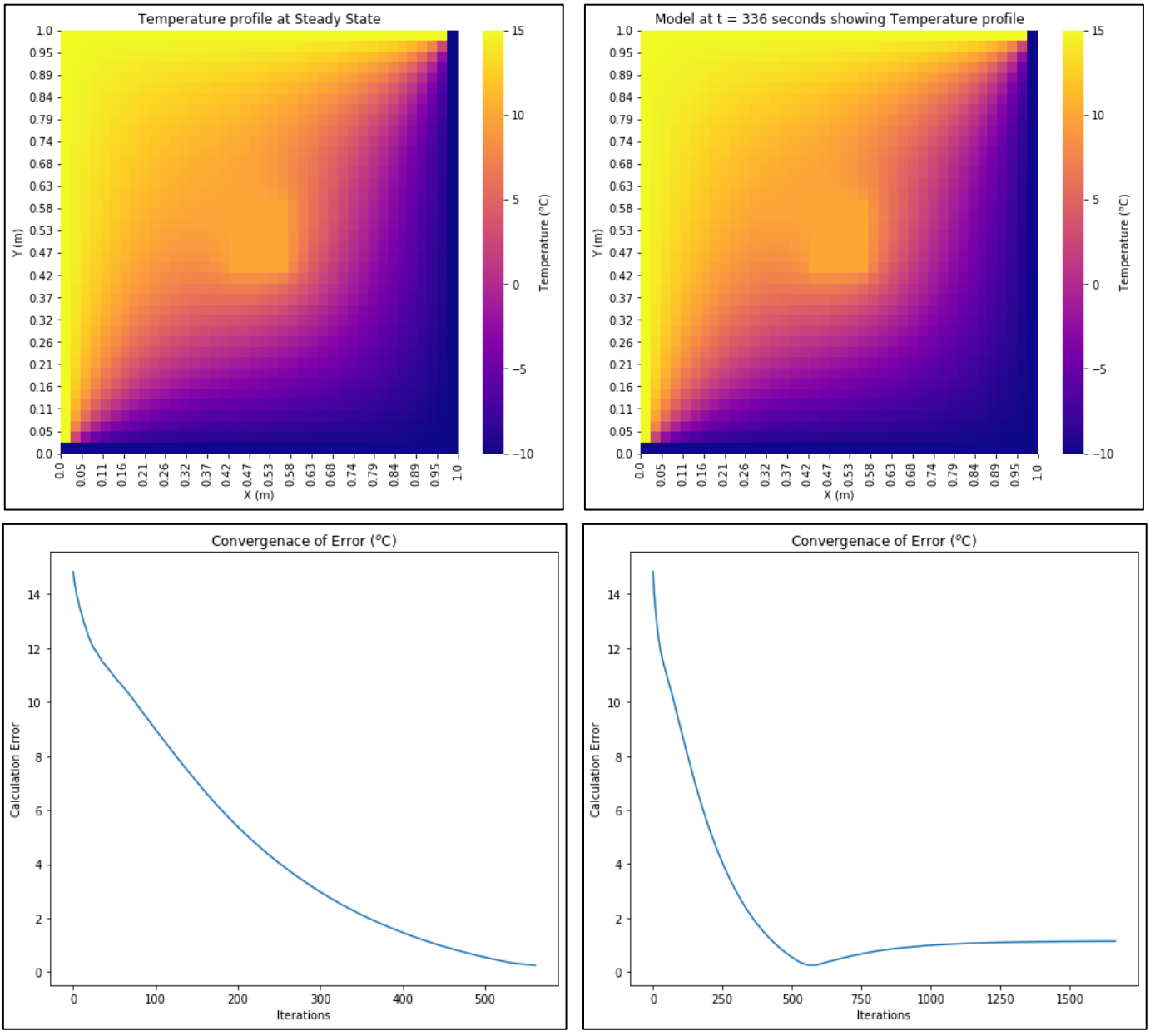

Figure 4) Temperatures at steady state (top left) compared with dynamic calculation stopped at convergence (top right) with a greatest difference of 0.248°C between profiles. Plot of difference between dynamic state and steady state calculation against dynamic iterations (bottom left). Increase of error if calculation allowed to continue beyond non-SS convergence (bottom right).

Future Changes:

In the future I hope to develop this model further with additional features and capabilities. These include:

- A user interface to easily create model scenarios and run calculations from
- Additional forms of heat transfer than conduction, and to allow for sources of heat in addition to fixed temperatures.
- Once I have added the above two capabilities, I hope to add mass transfer calculations to the model as well.
- Allow properties of each individual cell to be calculated upon each iteration.

My overall goal is to create modelling tool that can model simple heat and mass transfer examples that can be used as an educational tool.

For these I will likely have to switch from written out calculation procedures to using built in SciPy ODE and PDE solvers such as *scipy.integrate.solve_ivp*. However, for this project, I wanted to get the experience of writing my own solvers to develop my experience with using them and to understand how they work with more simple examples before using a more complex solver such as *solve_ivp*.

As for the user interface, I want the user to be able to 'paint' initial values onto a grid when creating a model. They can also save/load models into the program as well. There will be a panel where they can modify calculation properties such as time, tolerances, and perhaps which method is used. Then when they run the model the results are shown in a new window/ tab.

Appendix: Steady State Derivation

The derivation for the steady state equation is shown on the following pages

8/03/20 **Calculating steady-state temperature:**

- Use the FDM + Eulers equation to develop a s.s equation.
- Implement into the script and run comparison with the matlab reference.

Steady-state in example matlab code.

$$T_{i,j} = \frac{1}{4}\cdot\left(T_{i+1,j} + T_{i,j+1} + T_{i-1,j} + T_{i,j-1}\right)$$

They are simply taking the average of the surrounding points until the difference from one iteration to the next becomes less than a pre-defined tolerance.

Original eqn.

$$T^{k+1}_{i,j} = T^{k}_{i,j} + \Delta t \cdot \alpha \cdot \left[\left(\frac{T^{k}_{i-1,j} - 2\cdot T^{k}_{i,j} + T^{k}_{i+1,j}}{\Delta y^2}\right) + \left(\frac{T^{k}_{i,j-1} - 2\cdot T^{k}_{i,j} + T^{k}_{i,j+1}}{\Delta x^2}\right)\right]$$

re-arrange for $\frac{\partial T}{\partial t}\Big|_t$ becomes...

$$\frac{T^{k+1}_{i,j} - T^{k}_{i,j}}{\Delta t} = \alpha \cdot \left[\left(\frac{T^{k}_{i-1,j} - 2\cdot T^{k}_{i,j} + T^{k}_{i+1,j}}{\Delta y^2}\right) + \left(\frac{T^{k}_{i,j-1} - 2\cdot T^{k}_{i,j} + T^{k}_{i,j+1}}{\Delta x^2}\right)\right]$$

① s.s $\frac{\partial T}{\partial t} = 0$ →

$$② = \alpha \cdot \left[\left(\frac{T^{k}_{i-1,j} - 2\cdot T^{k}_{i,j} + T^{k}_{i+1,j}}{\Delta y^2}\right) + \left(\frac{T^{k}_{i,j-1} - 2\cdot T^{k}_{i,j} + T^{k}_{i,j+1}}{\Delta x^2}\right)\right]$$

Re-arrange for $T_{i,j}$

$$0 = \alpha \cdot \left(\frac{T^k_{i-1,j} - 2\cdot T^k_{i,j} + T^k_{i+1,j}}{\Delta y^2} \right) + \alpha \cdot \left(\frac{T^k_{i,j-1} - 2\cdot T^k_{i,j} + T^k_{i,j+1}}{\Delta x^2} \right)$$

multiply by $\Delta y^2 \cdot \Delta x^2$

$$0 = \alpha \cdot \Delta x^2 \left(T^k_{i-1,j} - 2\cdot T^k_{i,j} + T^k_{i+1,j} \right) + \alpha \cdot \Delta y^2 \cdot \left(T^k_{i,j-1} - 2\cdot T^k_{i,j} + T^k_{i,j+1} \right)$$

Expand the brackets (removing k notation)

$$0 = \alpha \cdot \Delta x^2 \cdot T_{i-1,j} - 2\cdot\alpha\cdot\Delta x^2 \cdot T_{i,j} + \alpha \cdot \Delta x^2 \cdot T_{i+1,j} + \alpha \cdot \Delta y^2 \cdot T_{i,j-1} - 2\cdot\alpha\cdot\Delta y^2 \cdot T_{i,j} + \alpha \cdot \Delta y^2 \cdot T_{i,j+1}$$

Collect like terms of $T_{i,j}$ and re-arrange.

$$T_{i,j} \cdot 2\cdot\alpha\cdot(\Delta x^2 + \Delta y^2) = \alpha \cdot \Delta x^2 \cdot T_{i-1,j} + \alpha \cdot \Delta x^2 \cdot T_{i+1,j} + \alpha \cdot \Delta y^2 \cdot T_{i,j-1} + \alpha \cdot \Delta y^2 \cdot T_{i,j+1}$$

$$T_{i,j} = \frac{\alpha \cdot \Delta x^2 \cdot T_{i-1,j} + \alpha \cdot \Delta x^2 \cdot T_{i+1,j} + \alpha \cdot \Delta y^2 \cdot T_{i,j-1} + \alpha \cdot \Delta y^2 \cdot T_{i,j+1}}{2\cdot\alpha\cdot(\Delta x^2 + \Delta y^2)}$$

- Cancel out α from eqn. $\alpha = k_T / c_p \cdot \rho$, this may appear in-correct however if you allow $t \to \infty$ the final s-s conditions will be reached regardless of the physical conditions it occupies.

$$T_{i,j} = \frac{\Delta x^2 \cdot T_{i-1,j} + \Delta x^2 \cdot T_{i+1,j} + \Delta y^2 \cdot T_{i,j-1} + \Delta y^2 \cdot T_{i,j+1}}{2\cdot(\Delta x^2 + \Delta y^2)}$$

$$T_{i,j} = \frac{\Delta x^2 \cdot (T_{i-1,j} + T_{i+1,j}) + \Delta y^2 \cdot (T_{i,j-1} + T_{i,j+1})}{2\cdot(\Delta x^2 + \Delta y^2)}$$

Use this equation to solve for s-s