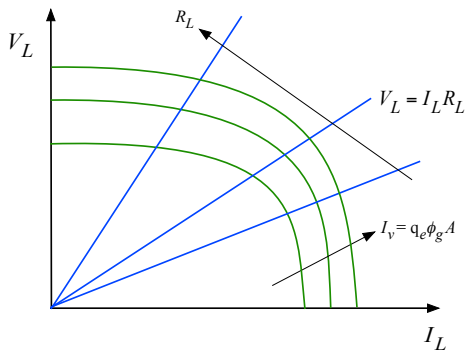
**ME249 Project 3** **Due date: Wednesday November 26, 2025**

You may team up with a partner for this project. Do not share information or results with other groups.

Code Files **from Project 3** to be used for Part 1:

CodeP3.1F25.ipynb
CodeP3.2F25.ipynb



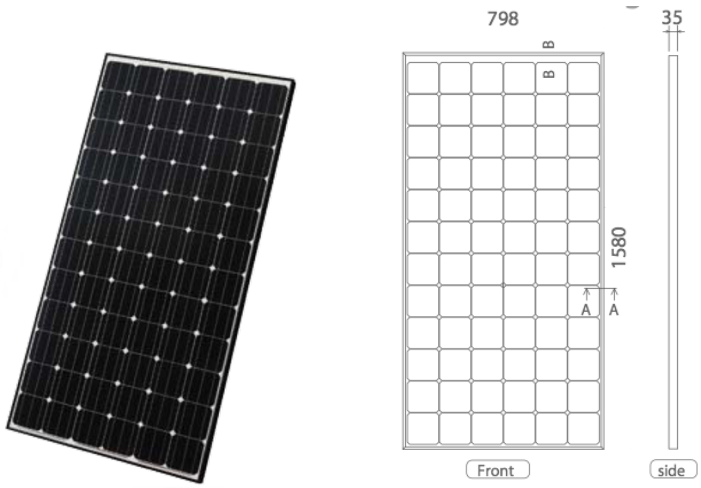




**Figure 3.1.1.** Solar PV panel design (units in mm).

## Part 1 – Exploration of Custom Loss Functions.

### Task 3.1.1

Task 1.1 of this project considers performance data for a solar panel of the type shown above. The panel contains 72 solar cells connected in series, each with an area of 173 cm$^2$. Performance test data for this type of unit is provided as a dataset that includes the following performance parameters:

Specified operating parameters:

Outside air temperature, $T_{air}$ (°C)

Incident direct normal solar radiation intensity, $I_D$ (W/m$^2$)

Load resistance, $R_L$ (Ohms)

Performance (output) parameters:
Panel output voltage to load $V_L$ (V)
Panel power output $\dot{W}$ (W)

Data set CodeP3.1F25.ipynb (with input data [ $T_{air}$, $I_D$, $R_L$ ]) provided for this project is a collection of data at solar radiation intensities up to the peak incident levels of solar radiation usually possible on the surface of the collector panel (maximum of about 1300 W/m$^2$). The goal of this initial task is to assemble a machine-learning-based model of the performance of the panel based mainly on the data at flux levels below 1300 W/m$^2$.

The CodeP3.2F25.ipynb provided with this project is similar to the CodeP2.4F25 file provided for Project 2 earlier. Consider the CodeP3.2F25.ipynb program to be a starting-point for this initial code setup. In this task, complete the following modifications to the skeleton code CodeP3.2F25.ipynb and the indicated training steps:

(a) For the original data set CodeP3.1F25.ipynb, determine the median value for each parameter and normalize the data by dividing each parameter value by its median value.

(b) Take the normalized original data created in part (a) and separate it randomly into two data sets: a training set with 2/3rds of the data and a second validation set with 1/3rd of the data.

(c) Substitute the normalized training set data into the skeleton code and convert it to a neural network model that can be trained using the training data set. For this first model, use a `keras.sequential` network with having these specs:

- specify a `RandomUniform` initializer (see skeleton code)
- an inlet layer having 6 neurons with `activation=K.elu, input_shape=[3]`
- 3 hidden layers with 6, 12 and 6 neurons
- an outlet layer with 2 neurons with no activation function
- set `activation=K.elu` for all the neurons except the outlet layer, and use the `RMSprop` optimizer, as configured in the skeleton program.

Using the `model.fit` routine with backpropagation as configured in the skeleton program is recommended.

(d) Train the neural network model constructed in part (c) using the training data. Try to get the mean absolute error below 0.025 if possible. You can adjust the initialization and/or the learning parameter a bit to try to improve convergence.

(e) Compare the trained model predictions to the training data set, report the mean absolute error for the fit, and create a log-log plot of predicted power output vs. data value power output for each set of data point operating conditions ( to be included in your report).

(f) Repeat the steps of part (e), comparing the model predictions this time to the normalized validation data. Report the mean absolution error and include the log-log plot specified in (e) for these data in the summary report.

(g) Taking the air temperature to be fixed at 20 °C, use the trained model created in this task to create predictions of the solar power output for 4 Ohms < $R_L$ < 8 Ohms and 500 < $I_D$ < 1200 W/m$^2$, and create a surface plot of the power delivered ( $\dot{W}$ in Watts) to the load as a function of these two variables.

## Task 3.1.2 Construction of a neural network model with a custom loss function

To explore the implementation of a custom loss function, in this section you will implement one in the neural network model constructed in task 1.1.

### Summary of Process to implement a custom loss function

1. **Define the Custom Loss Function**:
   - The function takes `y_true` and `y_pred` as input tensors.
   - It performs the necessary calculations to return a single scalar value representing the loss.
2. **Compile the Model with the Custom Loss**:
   - Use the `model.compile()` method to specify the custom loss function.
3. **Train the Model**:
   - The custom loss function will be used automatically during training, backpropagating the error through the network to update weights.

### Example: Implementing a Custom Mean Squared Logarithmic Error (MSLE) Loss Function

**Steps:**

1. **Define the Custom Loss Function**

```
import tensorflow as tf
```

```
from tensorflow.keras import backend as K

# Custom MSLE loss function
def custom_msle(y_true, y_pred):
    first_log = K.log(y_true + 1)
    second_log = K.log(y_pred + 1)
    return K.mean(K.square(first_log - second_log))
```

This function calculates the logarithm of the true and predicted values (with a +1 to prevent log(0) errors), then computes the squared difference and returns the mean.

**2. Compile the Model with the Custom Loss Function**

When defining your Keras model, you can specify `custom_msle` as the loss function when compiling the model.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a simple neural network model
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),  # example input shape
    Dense(32, activation='relu'),
    Dense(1)  # Output layer for regression
])

# Compile the model with the custom loss function
model.compile(optimizer='adam', loss=custom_msle, metrics=['mae'])

# model.summary() # Optionally view model architecture
```

**Starting Point for Task 3.1.2**

Here you will set up a neural network training code by executing the following instructions:

Use a copy of the code developed in Task3.1.1 as the starting point for this Task.

Make a list of the design features for the neural network model you assembled for Task 3.1.1 of Project 3. Present this list in a first table for your report for Project 4 to document the starting point for your code changes. Execute steps (a) through (c) below:

(a) Make the following modifications to your starting point model:
`model.compile(loss='mse',optimizer=rms)`

This changes the loss function to mean squared error (the basline Keras mse).

Second, in the cell that calls the fit routine, deactivate the early stopping mechanism by changing this code:

```
historyData = model.fit(xarray,yarray,epochs=100,callbacks=[es])
```

to this:

```
historyData = model.fit(xarray,yarray,epochs=6000)
```

Note this also sets the training to 6000 epochs.

(b) Run this modified version of your code for 6000 epochs. Use the results of this training to fill in the first row (baseline results) in the table below. Do a hand calculation to fill in the cell in the far-right column (computing the rms error). Also, to compare the trained model predictions to the training data set, create a log-log plot of predicted power output vs. data value power output (for results to 6000 epochs) for each set of data point operating conditions in the training set.

| **loss** | **Loss at end of 6000 epochs** | **best mse of 6000 epochs** | **rms error = sqrt(best mse)** |
|---|---|---|---|
| mse loss baseline | | | |
| | | | |
| **Custom m4e loss** | **Loss at end of 6000 epochs** | **best m4e of 6000 epochs** | **r4s error = sqrt(sqrt(best m4e))** |
| | | | |

(c) Next, in a copy of your modified code used for the mse loss baseline, replace the compile statement in the modified Task 3.1.1 code with the loss function definition and new compile statement below:

```
from keras import backend as K
def mean_double_squared_error(y_true, y_pred):
     return K.mean(K.square(y_pred - y_true)*K.square(y_pred - y_true)))

model.compile(optimizer = "rmsprop", loss = mean_double_squared_error)
```

Here, the objective is to define a different custom loss function that more heavily weights outlier data error. This error function (m4e) is proportional to the 4$^{th}$ power of the difference between the predicted and data values. Note that when you use a custom loss function, you need to put its name without quotes, as you pass the function object, not a string.

Run this version of your code with the custom `mean_double_squared_error` loss function for 6000 epochs and use the results of this training to fill in the appropriate row in the table above. Also, to compare the trained model predictions to the training data set, create a log-log plot of predicted power output vs. data value power output (for results to 6000 epochs) for each set of data point operating conditions in the training set. In your report, provide an assessment of whether this loss function appears to offer any training advantages over a conventional mse loss function.

**Part 2 – Construction of a PINN solution for 2D Heat Transfer**

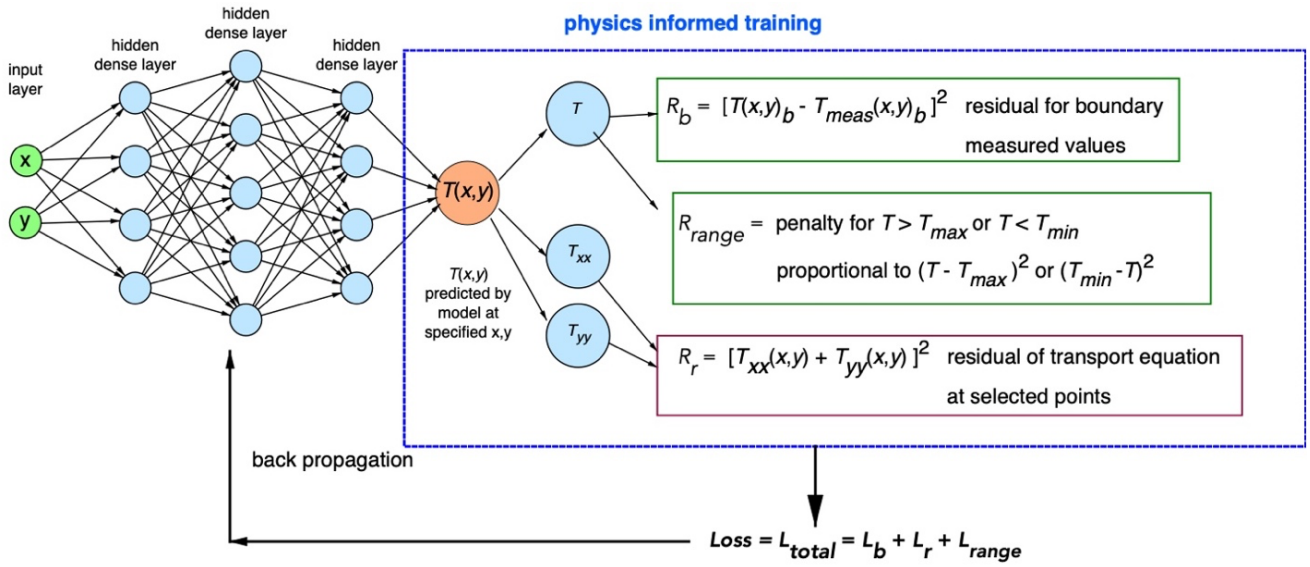



**Figure 3.2.1**

Code Files for Part 2:
CodeP3.3F25.ipynb
CodeP3.4F25.ipynb

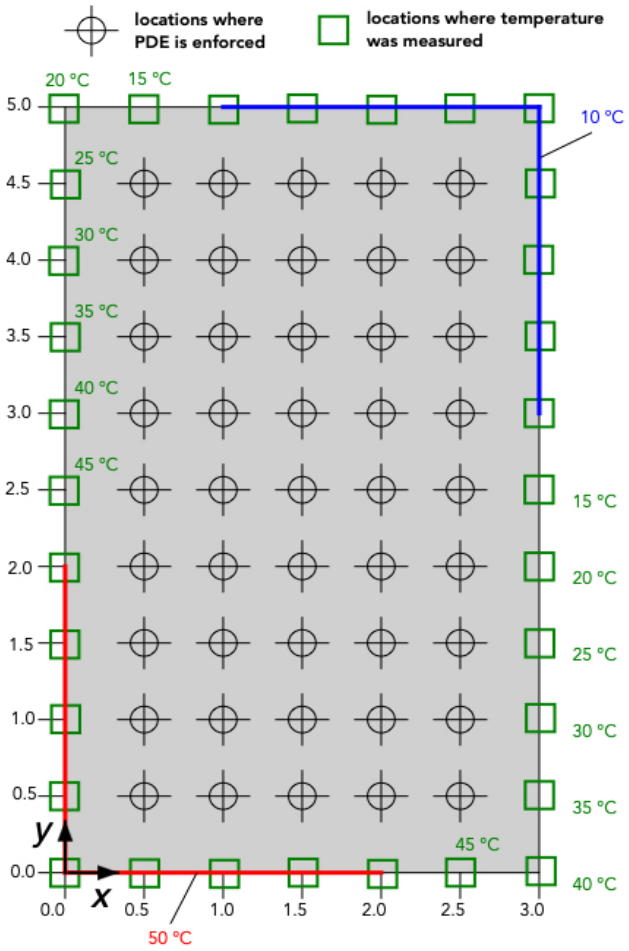
In Part 2 of this project, the objective is to construct a Physics Inspired Neural Network (PINN) solution for a two-dimensional heat conduction process. The PINN for this case is depicted schematically in Figure 3.2.1. The aim is to predict the temperature field for the domain of material shown in Figure 3.2.2. The temperatures measured along the boundary of the domain are indicated in Figure 3.2.2. Notice that the lower left corner portion of the boundary is the hottest and the upper right corner portion is the coldest, so heat can be expected to flow primarily from the hot lower left region to the upper right region.

There are no measurements of interior temperatures, but the 2D conduction heat transfer equation requires that the Laplacian of the temperature must equal zero in the interior. So, as indicated in Figure 3.2.1, here you will define a loss function that is closest to zero when the neural network model predicts temperatures that match the measured perimeter values at those locations, and at the interior locations indicate din Fig. 3.2.2, it predicts that the Laplacian is close to zero, and the temperature is not above the maximum perimeter temperature (50 °C) and is not below the minimum perimeter temperature. Note that these limitations on the range of temperature are a consequence of the second law of thermodynamics.

There are no measurements of interior temperatures, but the 2D conduction heat transfer equation requires that the Laplacian of the temperature must equal zero in the interior. So, as indicated in Figure 3.2.1, here you will define a loss function that is closest to zero when the neural network model predicts temperatures that match the measured perimeter values at those locations, and at the interior locations indicated in Fig. 3.2.2, it predicts that the Laplacian is close to zero, and the temperature is not above the maximum perimeter temperature (50 °C) and is not below the minimum perimeter temperature. Note that these limitations on the range of temperature are a consequence of the second law of thermodynamics.

The structure of the neural network to be used here will be similar to the network designs considered in Project 2, but here an additional feature will be added. Rather than use a standard keras loss function, you will incorporate a custom loss function that embodies the summation of the three loss components indicated in Figure 3.2.1.

**Figure 3.2.2.** Domain of the material body with indicated measured temperatures at the boundary. The coordinates $x$ and $y$ are in centimeters.




**Task 3.2.1**

The python notebook code file CodeP3.3F25.ipynb contains one cell in which a custom loss function is defined. Comments in the file indicate how this code calculates each loss component and how the components are combined to get the total loss used in backpropagation training. Begin by going over this code to familiarize yourself with the following steps:

- Computation of the mean squared error for the boundary temperature values (done by randomly sampling a point on the boundary and computing the contribution for that point. This is done enough times so that, on the average, each is sampled each epoch.

- The Laplace residual is computed at randomly chosen points in the interior. This calculation uses derivative information available from tensorflow as it executes the backpropagation process.

- Also, at each interior point where the Laplace residual is computed and at each boundary point where the temperature residue is computed, a penalty is added to the loss function proportional to how far beyond the limit the temperature is at that point.

- When their calculation is completed, the contributions of the three loss components are combined to compute the total loss, with an adjustable parameter kappa multiplying the mean squared Laplace residual, and a constant weight of 20 multiplying the temperature out-of-range penalty. For each epoch in the neural network training, this custom loss function is called and the total loss value, computed as described above, is returned as the loss value used to evaluate how well the trained network satisfies the solution requirements.

To assemble the PINN solution code:

(i) In the code in file CodeP3.3F25.ipynb, the array `pointsBoundaryChk` specifies the [x, y] coordinates of the boundary points where the difference between the predicted temperature and boundary value is to be matched. This array is only partially listed in this file. You must add to it to list all the points in the table on the next page.
(ii) Likewise, in the code in file CodeP3.3F25.ipynb, the `points` array specifies the [x, y] coordinates of the interior points where the Laplace residual is to be computed. This array is only partially listed in this file. You must add to it to list all the points in the corresponding table on the next page.
(iii) Once steps (i) and (ii) are completed, cut and paste the entire code from the revised file CodeP3.3F25.ipynb into the code file CodeP3.4F25.ipynb at the location just below the comments statement: #CUSTOM LOSS FUNCTION HERE. (Note this is a bit before the compile statement.)

Once the custom loss function is installed, try running the first four cells of the code. It should run for 100 epochs if everything has been correctly modified.

**Task 3.2.2**
Run the first three cells of the modified code CodeP3.4F25.ipynb developed in Task 4.2.1, and then run cell four for an increasing number of epochs to try to reduce the loss to less than 0.005. Once this level of loss has been achieved, run the last two cells of the code to generate a surface plot and a heat map of the temperature field. In Fig. 2.3 is a surface plot of the temperature field for the heat transport equation (Laplacian = 0) solved using a finite difference method and the Gauss-Siedel solution scheme with the same temperature boundary conditions specified in Figure 2.1, and with a mesh size matching the spacing of the interior nodes in the PINN model.

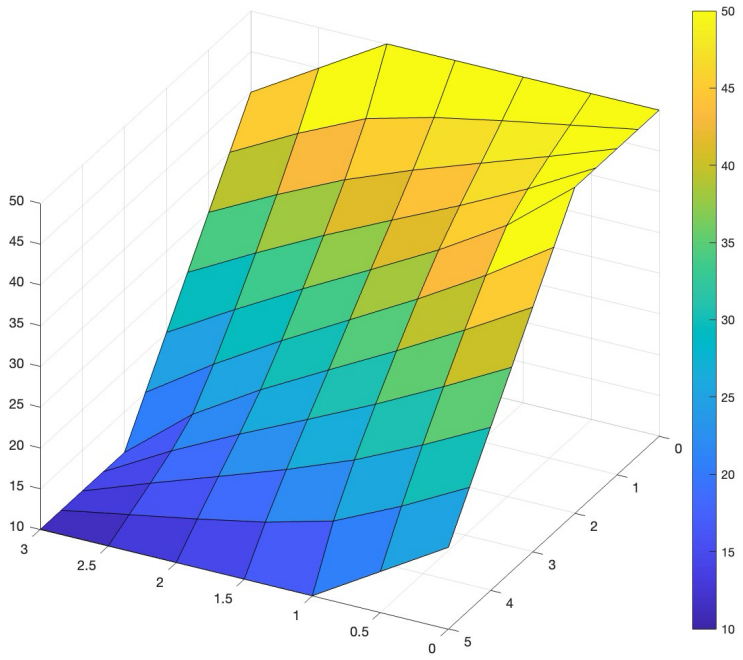
Coordinates of boundary points for temperature residual computation

```
pointsBoundaryChk = tf.constant([
        [ 3.0, 0.0 ],
        [ 3.0, 0.5 ],
        [ 3.0, 1.0 ],
        [ 3.0, 1.5 ],
        [ 3.0, 2.0 ],
        [ 3.0, 2.5 ],
        [ 3.0, 3.0 ],
        [ 3.0, 3.5 ],
        [ 3.0, 4.0 ],
        [ 3.0, 4.5 ],
        [ 3.0, 5.0 ],
        [ 0.0, 0.0 ],
        [ 0.0, 0.5 ],
        [ 0.0, 1.0 ],
        [ 0.0, 1.5 ],
        [ 0.0, 2.0 ],
        [ 0.0, 2.5 ],
        [ 0.0, 3.0 ],
        [ 0.0, 3.5 ],
        [ 0.0, 4.0 ],
        [ 0.0, 4.5 ],
        [ 0.0, 5.0 ],
        [ 0.5, 0.0 ],
        [ 1.0, 0.0 ],
        [ 1.5, 0.0 ],
        [ 2.0, 0.0 ],
        [ 2.5, 0.0 ],
        [ 0.5, 5.0 ],
        [ 1.0, 5.0 ],
        [ 1.5, 5.0 ],
        [ 2.0, 5.0 ],
        [ 2.5, 5.0 ]
    ], dtype=tf.float32)
```

Coordinates of interior points for Laplace residual computation

```
points = tf.constant([
            [0.5, 0.5],
            [0.5, 1.0],
            [0.5, 1.5],
            [0.5, 2.],
            [0.5, 2.5],
            [0.5, 3.],
            [0.5, 3.5],
            [0.5, 4.],
            [0.5, 4.5],
            [1.0, 0.5],
            [1.0, 1.0],
            [1.0, 1.5],
            [1.0, 2.],
            [1.0, 2.5],
            [1.0, 3.],
            [1.0, 3.5],
            [1.0, 4.],
            [1.0, 4.5],
            [1.5, 0.5],
            [1.5, 1.0],
            [1.5, 1.5],
            [1.5, 2.],
            [1.5, 2.5],
            [1.5, 3.],
            [1.5, 3.5],
            [1.5, 4.],
            [1.5, 4.5],
            [2., 0.5],
            [2., 1.0],
            [2., 1.5],
            [2., 2.],
            [2., 2.5],
            [2., 3.],
            [2., 3.5],
            [2., 4.],
            [2., 4.5],
            [2.5, 0.5],
            [2.5, 1.0],
            [2.5, 1.5],
            [2.5, 2.],
            [2.5, 2.5],
            [2.5, 3.],
            [2.5, 3.5],
            [2.5, 4.],
            [2.5, 4.5]
        ], dtype=tf.float32)
```

If you have achieved a sufficiently low loss for your trained model, the plot of the PINN solution produced by the code in the last cell of the CodeP3.4F25.ipynb should be comparable to the Gauss-Seidel solution shown in Fig. 2.3. In you project report show the plot in Fig. 2.3 together with the plot achieved with your program to demonstrate how well they agree.




Consider the solution you obtain for Task 3.2.2 to be the baseline solution for the PINN model approach when considering the specified explorations defined in Tasks 3.2.3 and 3.2.4 below.

**Figure 3.2.3.** Gauss-Siedel finite-difference solution for problem defined in Figs. 3.2.1 and 3.2.2.

**Task 3.2.3**

Make a copy of the program used to produce the baseline results in Task 3.2.2. Edit the code in the copy to change kappa from 1.3 to 5.0 (change this in two places). Note that this pushes the training to more tightly match the Laplace equation in the interior. Run the code as described in Task 3.2.2 for the baseline case (aim to get a loss less than 0.01 or as close as you can get to that), and compare your plotted surface to Figure 2.3. Document your results, including surface figures, in your report. Based on your results would you say that this change in kappa improved the training and/or solution accuracy? Briefly explain your answer, describing any changes in training behavior.

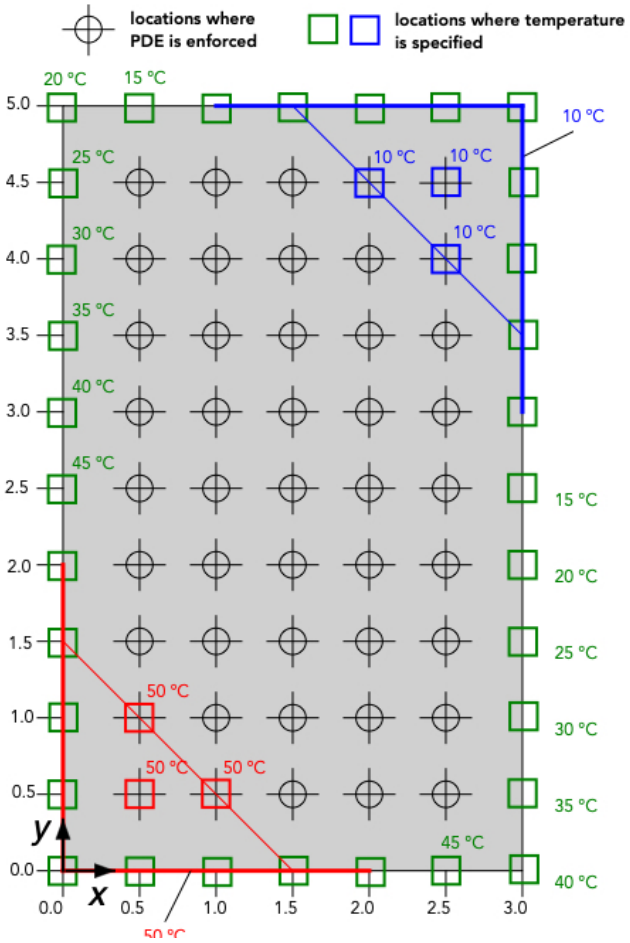
**Task 3.2.4**

Make another copy of the program used to produce the baseline results in Task 3.2.2. Edit the code in the copy modify the network design to be:

```
model = Sequential([
        Dense(8, input_dim=2, activation='elu'),
        Dense(16, activation='elu'),
        Dense(1)  # Output temperature T
    ])
```

Note that this will increase the complexity of the model, allowing it to model more nuances in the behavior of the solution. Run the code as described in Task 3.2.2 for the baseline case (aim to get a loss less than 0.01 or as close as you can get to that) and compare your plotted surface to Figure 3.3. Based on your results would you say that this change in the network improved the training and/or solution accuracy? Briefly explain your answer, describing any changes in training behavior.

**Figure 3.2.4.** Domain of the material body with indicated measured temperatures at the boundary. The coordinates $x$ and $y$ are in centimeters.




**Task 3.2.5**

Consider the slightly different steady conduction heat transfer circumstances for the hexagonal domain depicted in Figure 3.2.4. Note that the PINN code used to predict the temperature field for the rectangular domain considered in Task 3.2.3 can be easily adapted to the circumstances in Figure 3.2.4. To do so, start with a copy of the code used for Task 3.2.3 and make the following changes:

(i) In the loss function, remove the three blue and three red interior point locations in Fig. 3.2.4 from the array of $(x,y)$ points at which the residue of the governing equation is computed as contributions to the loss function.

(ii) In the loss function, add the three blue and three red interior point locations in Fig. 3.2.4 to the set of array points at which boundary condition residues are computed as contributions to the loss function.

(iii) In the CodeP3.4F25 program, add the six additional coordinates and the corresponding temperatures to the `xdata` and `ydata` arrays.

Once these changes are made, run the first three cells of the modified code CodeP3.4F25.ipynb, and then run cell four for an increasing number of epochs to try to reduce the loss to less than 0.005. Include the resulting temperature surface in your report and assess ow similar that temperature distribution is to the temperature field obtained in Task 3.2.3.

**Overall Project 3 Tasks to be divided between coworkers:**

(1) Program modifications for Part 1
(2) Training process and computations for comparisons
(3) Plotting and interpretations of results for Part 1
(4) Program modifications for neural network modeling in Part 2
(5) Plotting and analysis of the results for Part 2
(6) Write-up of the results and conclusions

**Deliverables:**

Written final report should include:

(1) Written summary of how the work was divided between coworkers.

(2) Assessment of the results and comparisons for the two different neural network designs considered in Part 1.

(3) Plots and tables requested in Parts 1 and 2. (should not be in the Appendix, be sure to label axes with units)

(4) An assessment of viability of the first neural network design considered for system control in Part 2.

(5) Your assessments and conclusions should be clearly written with quantitative information to justify them.

(6) A copy of your programs should be attached to the report as an appendix.

**Grade will be based on:**

(1) thoroughness of documentation of your analysis, especially the logic behind design choices for neural network
(2) accuracy and clarity of interpretation
(3) thoroughness and the documentation of the reasons for your assessments of results.

Summary report due: **Wednesday November 26, 2025**