Inverse heat transfer software

* This document is a draft of the diploma thesis describing the nuts and bolts of the software solution

# Architecture

* **Computing engine solving the heat transfer equations**
  + One for classical problems, one for inverse problems
  + Helper module for interpolation
    - Turns out to be much quicker than the classical numpy interpolation, mostly because of the specific way we need the interpolated values – we need them mostly in ascending order, when we are moving through the time series of heat-fluxes and temperatures.
* **GUI with integrated graph, menu options and simulation controllers**
  + **Qt5 Designer**
    - .ui file will be created, which can then be transformed into a python file, or be just imported and parsed as is.
  + **Graphs for both temperature and heat flux displaying**
    - They are created as individual modules (classes), to increase the possible customization of both plots.
  + **User input generators**
    - It is very modular, code does not have to be changed (open for extension, closed for modification) – infrastructure in place
      * Adding a new variable is a matter of seconds
      * We can easily specify the name of input element to target it, and also the variable name the value will be then assigned to – so we can use it further downstream and send it wherever we want (to the computation engine in this case)
    - All the units are well documented
    - It offers an easy possibility of declaring default values, so that user does not have to input everything from scratch before every simulation.
    - There is an input validation, so when the field is expecting a whole number, inputting a decimal number or some letters will cause the validation to fail, and the wrong value will be replaced with a default value and user will be notified
    - Little challenge was not to confuse users nor computation engine with units (on small objects it is better to measure distance in centimetres, however the computation engine is expecting everything to be in SI units – metres)
      * It was handled by a “multiplicate\_to\_SI” coefficient, which is a part of each user input row, and is responsible for transforming values visible by user (centimetres) to SI values (metres). In this case the coefficient has a value of 0.01., as a length in centimetres must be multiplied by this number to yield a length in metres.
    - **Possible improvements**:
      * Transferring the information to a JSON file, and just load it on \_\_init\_\_ - this way the user input would be completely separated from the code (as we would not be touching the .py code file, but rather a JSON data file, which adds a little comfort in the way we are not afraid of mistakes there “because it is not the code”, and can be more user friendly)
    - **Simulation step**
      * Defines a time value for one simulation step – on how big time intervals will we cut the whole timeframe of the measurements.
      * Assumption: The larger this step, the quicker the simulation will be. However, the precision goes down with its increase.
      * Can be arbitrary decimal value (float)
      * Is expressed in seconds
    - **Object length**
      * Defines the length of the 1D object we are creating the simulation for
      * Can be arbitrary decimal value (float)
      * Is expressed in centimetres
    - **Position of interest**
      * Defines the distance from the beginning of the element in place in which the temperatures were measured
      * Can be arbitrary decimal value (float)
      * Is expressed in centimetres
      * Possible improvement: always check if this value is not higher than the Object length, as it would make no sense – and notify user about this
    - **Number of elements**
      * Defines how granular will the simulation grid be (how many nodes will be evenly placed on the whole Object length)
      * **Assumption**: The higher the number of elements, the more precise the calculation should be in theory. However, this will increase the simulation time, and the simulation error is not always going down paradoxically.
      * It can be arbitrary positive whole number, and has no units
    - **Plotting period**
      * Defines how frequently to update the plot, in the sense of simulation seconds
      * **Assumption**: Higher values will cause the simulation to be quicker, because the plotting does not have to occur so often, so less time will be spent on this
      * Can be arbitrary positive whole number
      * Is expressed in seconds
    - **Theta**
      * Determines explicitness (0) and implicitness (1) of the algorithm approach.
      * LINK SOME LINK HERE :) – for a resource where this is described
      * Explicitness means we are more focusing on matching already calculated values rather than on matching the future values from the measurement
      * Optimal values is 0.5
        + By 0.5, it produces error of 0,464
        + It crashes by 0.4
        + 0.6 = 0,463
        + 0.99 = 0,459
        + With the increase we can observe slight decrease of the error margin
        + VALUES UNDER 0,5 ARE CAUSING ERRORS

Probably there is a bug that is preventing us from doing that somehow

INVESTIGATE TO BE ABLE TO COMPARE THE APPROACHES

* + - * Can be arbitrary decimal value (float) between 0 and 1 and has no units
    - **Robin Alpha**
      * Defines the convection heat transfer coefficient on the end of the object
      * Can be arbitrary decimal (float) number
      * Is expressed in Watt/Kelvin
    - **Window span**
      * Only for inverse simulation
      * Defines how far into the future to look when matching the temperatures during inverse simulation.
      * **Assumption**: Increasing the value will cause the simulation to take longer but has no big positive impact on the precision. On the contrary, we were getting the best results when the window span was only 1.
        + One disadvantage of having the window span as the smallest value of 1 is that it makes a lot of sharp spikes in heat fluxes – as it is only interested in one window at a time, and so reacts sharply on any change.
        + The window span of 2 has shown to be a nice compromise, as the precision is not so much smaller (error margin of 83 vs 76 by window span of 1) – and the heat fluxes were much smoother in time, with no drastic movements
      * It can be arbitrary positive whole number, and has no units
    - **Tolerance**
      * Only for inverse simulation
      * Defines how accurate should the temperature-matching be
      * **Assumption**: Higher values will cause the simulation to take longer time
      * Can be arbitrary decimal value (float) and has no units
  + **GUI features**:
    - Information panel for the user on the top
    - Highlighting the button that was clicked to give a visible feedback
    - Locking the inputs when simulation is running, not to confuse user
    - Hovering over input variable shows its description
    - Possibility of saving the data and/or plots from the simulation
* **Multithreaded infrastructure for the GUI**
  + Workers, that enable sending signals between threads (on progress (time), finish (error margin) etc.)
  + COMPARE IT WITH THE PREVIOUS TKINTER APPROACH, LIST THE ADVANTAGES ETC.
  + They are enabled by PyQt5 library, which supports this multithreaded behaviour very well, in contrast to Tkinter, which did not have this capability (and had to be hacked around)
* **Material service**
  + Is responsible for supplying material data to the GUI. From these data the material choice menu is created in GUI.
* **CSV file with data from measurements**
  + It contains comma-separated rows with measurement data regarding:
    - Time from the beginning of measurement
    - Temperature measured inside the body
    - Heat flux applied to the body
    - Ambient Temperature in the room
* **CSV file with material properties**
  + Wikipedia scraper for updated information
  + DESCRIBE THE MECHANISM OF FETCHING THESE DATA AND POINT OUT THE POSSIBILITIES
    - Can choose arbitrary materials, I chose metals
    - Fetching rho, cp and lambda
* **Performance tester**
  + DESCRIBE THE PROCESS OF PERFORMANCE PROFILING

GUI choice

* DESCRIBE THE DIFFERENCE BETWEEN PYQT5 AND TKINTER
* PyQt5 should be replaced in the end by PySide2, because of its better licensing conditions (software using PyQt5 must contribute some money to PyQt5 if being monetized)
* Tkinter is a basic library for building python GUIs, as it already comes packaged in a standard library, therefore these is no need for installation
* It is very easy to use, but is not very suitable for bigger applications
* PyQt5 offers multiple benefits over Tkinter
  + Performance:
    - The new GUI frameworks allow for the easy possibility of multithreading, which is making the application quicker and more responsive. There is no need to switch attention between calculating and listening mode as was necessary in case of Tkinter version.
  + Functionality
    - It has a richer library of available widgets and behaviours that can be easily implemented.
  + UI development
    - PyQt5 offers an app called Qt Designer, which is itself a GUI for creating GUIs. This UI is then completely separated from the business logic, and therefore almost anybody without any programming skills can create it. As long as the names of the widgets remain the same, it is possible to change layout of the UI freely, without having to worry about breaking the code.
    - <https://doc.qt.io/qt-5/qtdesigner-manual.html>
  + Usage outside python
    - Qt as a platform for creating GUIs is not used only in Python, but in a variety of widely used languages and platforms (<https://en.wikipedia.org/wiki/Qt_(software)>)
    - Both the knowledge of it, and the possibility of transforming the app into a different language, if necessary, can prove to be very useful.

Computation engine description

* What equations are used, how they look like in python
* OPTIMIZATIONS DURING DEVELOPMENT – examples how they affected the performance
  + The cheapest option however is to use better hardware
  + Deep copying is very slow – better to [:]

Basic flow of the software

* How the components cooperate together

Dependences and libraries

* Requirements.txt file created for the purpose of quick installation of all dependencies, as well as making sure some dependency will not change unexpectedly to break the program
* Scipy had to be downgraded from 1.3.3 to 1.2.1 to overcome problems with converting to .exe (<https://github.com/scipy/scipy/issues/11062>)

Possible features and improvements

* Storing user preferences
* Storing history of all user simulations
* Making the calculation and graph plotting separate, so calculating thread is not slowed down by plotting
  + It would require some architectural changes

# PARAMETERS TESTING

In order to have better understanding of all parameters that can vary in the simulation, it can be a good idea to find out how changing these parameters influences the final result. Namely, how do the simulation time and simulation error change when we play with the parameters.

parameters = {

"rho": 7850,

"cp": 520,

"lmbd": 50,

"dt": 1,

"object\_length": 0.01,

"place\_of\_interest": 0.0045,

"number\_of\_elements": 100,

"callback\_period": 500,

"robin\_alpha": 13.5,

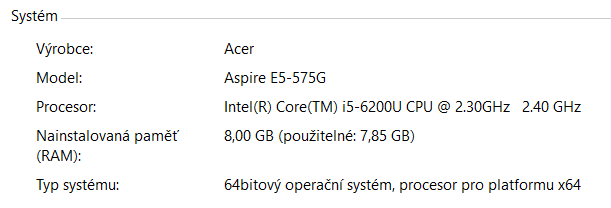
"theta": 0.5,

"window\_span": 2,

"tolerance": 1e-05

} … the default parameters

Hardware used for testing:

Our goal was to find out which variable parameters will cause the results to be the best (the simulation being the quickest or the most precise).

There is a slight issue with just trying the simulation once with all the possible parameters, and that is the variable speed of CPU, which makes spotting small differences almost impossible. Therefore, it is beneficial to perform all the simulations multiple times and average out all the values afterwards. The more simulations we will do, the more precise should the average be, but it will also take longer time.

Tests were carried out on a laptop, and also on a Virtual Private Server. Running them on a server could have the advantage of less other running processes. These running processes could increase the volatility of the results and make it less valuable.

## Results of the parameter testing:

* **Classic problems**
  + **Number of elements**



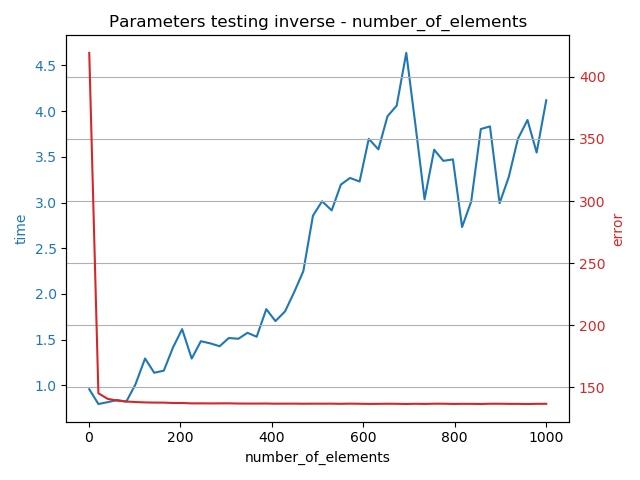
* + - We see that with increasing number of elements in our grid the simulation time is increasing in approximately linear fashion. The error margin is decreasing sharply in the beginning and seems to decrease in a slow manner even afterwards.
    - These effects are caused by the fact that the more elements are in the grid, the more calculations need to be done. More elements also better reflect the real conditions, when the number of elements is much higher.
    - It would be nice to observe the interval of the sharp error decrease (1-20 elements) to identify what is the least number of elements that yield a good result. Also the other interval (20-1000) is worth exploring, as it can show whether the error is really decreasing even in higher number of elements.
    - The optimal number of elements seems to lie around **20-40 elements**, where the error margin has already decreased sharply, and the simulation time has not risen so much yet.
    - **Discussion**: The length of the whole object must be also taken into account to generalize the recommendation of using the certain number of elements. Longer objects will need to be divided into more elements than shorter objects. Probably it could be worth to give the recommendation using not element number, but the size of one element (object 1 cm long divided into 20 elements = **0.5 mm per one element**). However, this value can be highly depending on the material that is used in the experiment – completely different results could have been yielded when we would use a wooden object instead of a steel one.
  + **Dt**



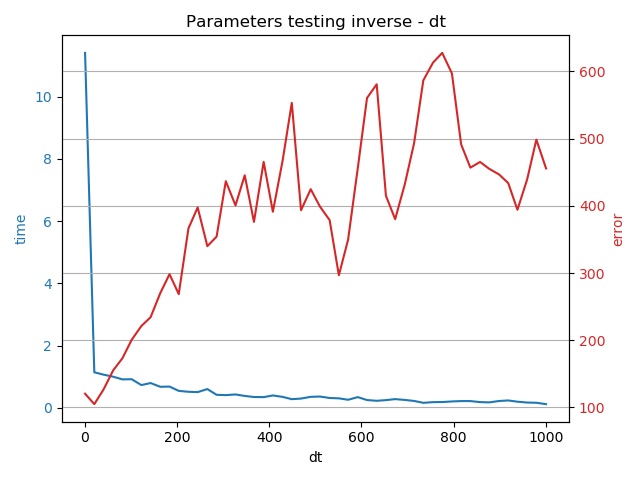
* + - It is apparent that with the increasing dt (timestep) the error is also increasing. Simulation time, on the other hand, is going down.
    - Reason for this behaviour is that with the higher values of dt there is less steps in the simulation to be calculated, therefore it takes less time for the simulation to finish.
    - With the higher dt we are also taking bigger steps at a time, which means we are neglecting what happened between those bigger intervals. This fact is causing the error margin to increase, because less information is taken into account, and this uncertainty is responsible for the error.
    - Sharp rises and falls in error margin between 600 and 1000 can be worth exploring – the probable explanation is that some of these timesteps are missing the rises or falls of the experiment heat flux, so they are less accurate than others.
    - Optimal number of elements seems to be around **40-60**, which is high enough for the simulation time to be low, and also low enough for the error not to be so high.
    - **Discussion**: The recommended time step in seconds is very specific to this experiment. To illustrate the point, we simply cannot choose a time step of 50 seconds when the experiment took only 30 seconds in total. Therefore, a better recommendation can be to calculate the number of steps in the whole experiment. In this case it would be 4500 seconds and 50 seconds for one step = **90 steps in total**. However, every experiment is different, and it can happen that there is a lot of sudden spikes in the heat-flux, which would not be taken into account if the time step was bigger.
  + **Theta**



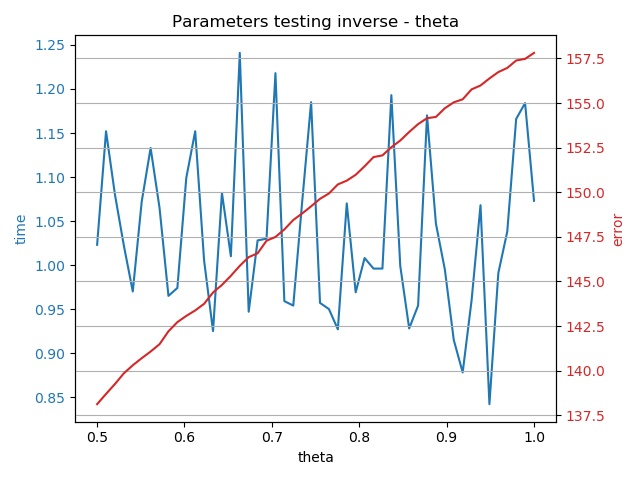
* + - The increase in theta is followed by a very small decrease in error. The time does not look like being dependant on theta at all.
* **Inverse problems**
  + **Number of elements**



* + **Dt**



* + **Theta**



* + **Window span**
  + **Tolerance**