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 scipy interpolation (scipy.interpolate), 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 contains the information about the UI. It can then be transformed into a python file or be just imported and parsed as is.
    - During the development it is easier to use the .ui file, but before
  + **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.
    - It offers for custom descriptions of each parameter, so user can have better idea what that parameter means
    - 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)
  + **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)
  + SENDING INFO THROUGH QUEUE + DESCRIBE IT
  + DESCRIBE MULTITHREADING + SOURCES (difference from multiprocessing, advantages, disadvantages…)
  + Creating GUI applications is little bit different from other application’s development and has its own specific hurdles. One of the problems is the unresponsiveness (freezing) of the GUI when some background tasks is going on (like a heat transfer simulation in our case).
  + The GUI is naturally running only on one thread (synchronously), and when this thread is busy with simulation, it cannot respond to user actions (events) in the GUI itself. This is not only user non-friendly but causes the whole application to be impossible to be controlled (by buttons etc.) – the only way is to wait until the simulation finishes.
  + Another way to look at things is that the one GUI thread is running in an infinite loop, listening for events. When user clicks a button or does some other interaction with a GUI, an event is registered, and some action is being executed. However, until this event is satisfied (the action is complete), the GUI cannot process any other events (resulting from user action) – and the application looks unresponsive or even crashes.
  + If we want to “do more things at once” (run some calculations on the background, and still listen for the user input, that is possibly influencing these background processes), we need to incorporate second thread, whose only purpose would be to run background processes – so the main GUI thread’s only responsibility would be to listen for user-based events (mouse clicks), and transmitting the commands to the “background thread,” which is doing the heavy lifting.
  + Tkinter, our first choice as a GUI framework, is single threaded only, it has no capabilities whatsoever for multiple threads to operate at the same time. However, there is a possible solution how to overcome this problem (be it a little bit hacky). Tkinter has a method called after(timestep, function), which allows for a certain function to be called in regular timesteps from the main GUI thread.
    - We took advantage of this to call small pieces of simulation between handing the focus to the GUI to listen for the user input. It meant that the main (and only) GUI thread was constantly switching between simulation mode and listening mode (with the listening period lasting couple milliseconds).
    - This solution fulfilled the desired result (being able to both run calculations and listen for user input at the same time) but was far from being perfect. One of the disadvantages of this solution is that the calculation was interrupted by the listening periods, which caused the time for the whole calculation to be higher than without those listening “pauses”. Also, there is a need to cut simulation into a lot of smaller pieces, as the simulation cannot run constantly because of those interruptions. These pieces cannot be very big, because they themselves could cause the application to be unresponsive or “laggy,” when being in the middle of simulating this big chunk. These pieces also should not be very small, because of the pausing time, which would cause the application to spend more time in listening mode than in calculating mode, which would slow things down immensely.
  + After contemplating all the Tkinter disadvantages, we decided to replace it with PyQt5, which natively supports multithreaded behaviour.
    - PyQt5 provides an easy way how to handle multithreading with its own classes (QRunnable and QThreadPool). This way it is trivial to define a function that should run on the background and can be spawned and controlled by user interaction in the GUI. It also makes easy to retrieve information from the background thread, that can emit data which will be picked up by the main thread.
    - Data from the background thread can be emitted on different occasions.
      * By emitting positive values after each step in a calculation, we make sure that the timer in the GUI is being incremented (it is a sign of the calculation running).
      * After the calculation thread finishes, it emits two values. First is the message that it has finished, so we can reflect the state in GUI and save the resulting graphs, if wanted. Second message caries over the result of the calculation in the form of the error margin – this will be displayed in GUI afterwards, to be visible for the user.
    - However, it is not enough to have a one-directional connection from the calculation thread to the main GUI thread – we also need a communication channel in the opposite direction, to be able to control the calculation (to pause it, stop it etc.).
    - Specifically for the purpose of multithreaded communication, python standard library offers a module named queue (<https://docs.python.org/3/library/queue.html>), which we took advantage of. Queue is a data structure that allows for the communication between two software components – both components can insert information onto the queue (put() method), or retrieve the information from there (get() method).
    - We set up a shared queue between main GUI thread and calculation thread, so that the main thread can send commands via this queue, and calculation thread is listening for these commands. These commands are in the form of words like “pause”, “stop” or “continue” – to which the calculation thread is reacting accordingly.
    - After we had a working bidirectional connection between the main GUI thread and the calculation thread, the last problem had to be solved – how to best update the plot in GUI, so that the user can see the simulation in real time.
      * There were multiple possible ways of achieving this (make it either a responsibility of the main GUI thread, the calculation thread, or even create a brand-new thread just for the purpose of plotting the results). In the end, we decided to give this job to the calculation thread, so the main GUI thread can be as responsive as possible for the user and incorporating a new thread could be unnecessarily complicating the whole architecture.
      * It certainly has a drawback of slowing the simulation down, because calculation thread must also update the plot, but is arguably the most concise and least difficult to implement solution.
        + In the case of speed problems, the dedicated thread for graph plotting would come into play. However, the speed improvements are hard to predict, as there would be extra communication overhead of the calculation thread sending the calculated data frequently to the plotting thread, probably through the shared queue.
      * Plotting the results from the calculation thread is done by passing it a reference of a GUI plot, whose plot() method will be called with the data from the calculating thread – effectively updating the plot.
  + Apart from multithreaded solution, there is also a multiprocessed solution
    - Multithreaded solution is generally easier and quicker to setup, it has the benefits of shared memory, but in python only code from one thread can be executed at the same time – meaning the speed improvement does not have to be so huge as with multiprocessed solution (GIL problem - <https://realpython.com/python-gil/>).
      * One nice thing about multithreading is that there does not have to be only one other thread, but depending on the computer processor, there can be multiple of them. And even when there are no available threads at the moment, the function to execute will be queued, and as soon as one thread is available, it will start with the execution.
    - Multiprocessing generally takes longer time to implement and instantiate, has some additional memory overhead, and communication between processes is not very smooth. However, once it is running, it can really run in parallel with the main process, without any GIL issues
  + <https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)>
  + <https://en.wikipedia.org/wiki/Thread_(computing)>
  + <https://www.learnpyqt.com/courses/concurrent-execution/multithreading-pyqt-applications-qthreadpool/>- the code was taken from this source and then modified for our needs
  + <https://stackoverflow.com/questions/29158220/tkinter-understanding-mainloop> - description of Tkinter problems
* **Material service**
  + Is responsible for supplying material data to the GUI. From these data the material choice menu is created in GUI.
  + We created our own database of material properties, there seems to be no python module that would satisfy our need
    - There is a library <https://pypi.org/project/materials/> , but it only includes very limited choice of materials, which is not suitable for us
  + Our material database exists in the form of CSV file with material properties – metals\_properties.csv
  + It was generated by fetch\_elements.py script, which is taking material data from Wikipedia
    - Script is visiting the Wikipedia page of all metals and takes the properties we need for the simulations - rho, cp and lambda
    - Material data can be refreshed anytime just by running this script
    - It uses the fact, that Wikipedia has almost static and non-changing structure of the webpage, and so the information there can be easily identified through web scraping (<https://en.wikipedia.org/wiki/Web_scraping>)
* **CSV file with data from measurements – DATA.csv**
  + It contains comma-separated rows with measurement data regarding:
    - Time from the beginning of measurement [seconds]
    - Temperature measured inside the body [Celsius]
    - Heat flux applied to the body [Watt]
    - Ambient Temperature in the room [Celsius]
  + There is a standalone module – experiment\_data\_handler.py, that has the responsibility of processing the data in a .csv file, and transforming them into a python data structures (lists of values)
* **Performance tester**
  + <https://www.youtube.com/watch?v=8qEnExGLZfY>
  + <https://osf.io/upav8/>
  + <https://www.youtube.com/watch?v=YjHsOrOOSuI>
  + <https://hackernoon.com/the-rules-of-optimization-why-so-many-performance-efforts-fail-cf06aad89099>
    - - 3 rules of optimization:
    - 1. Don't
    - 2. Don't ... yet
    - - finish your code (having even some comparison of performance)
    - - have tests
    - - now
    - 3. Profile
    - - there is no way we can correctly judge where the most time is spent,
    - so better use profilers
  + One of our goals was to optimize the code for the computing engine as much as possible, so the calculations and therefore the whole simulation would be as quick as possible (without sacrificing any precision).
  + This process was a gradual one – we started with a code that was written intuitively, so that it is logically performing all the calculation step by step, and we can be confident (in a certain manner) that nothing has been forgotten or missed.
  + This initial implementation laid down the foundation – we knew how quick our engine is, and what precision it can achieve. All further improvements had one goal – increase the speed without worsening the precision (quality of the result).
    - Without this first and “non-optimized” implementation it would be harder to make further improvements, as we would have no comparison according the quality of the result.
  + After creating a reference implementation, we already had some idea which parts could be rewritten, omitted or transferred to make the code run faster. However, before making any changes, there is a need to note down what performance we have at the moment – making a profiling of the code.
    - This profiling has also the goal of uncovering which exact parts of the code are taking the most time to perform. The road of optimization should then try to eliminate these bottlenecks first, before focusing on other parts that are not so crucial.
  + Developing a performance profile of our computation engine showed, without a surprise, that by far the most time is spent in the function EvaluateNewStep(), which is responsible for the actual calculation of temperature distribution in the object.
    - The most crucial part of this function, that we simply cannot omit – the solving of a linear equation (A\*T=b) – was initially taking only around 15 % of the overall function. Meaning that whole 85 % of the function was spent by just preparing the variables for the linear equation.
    - Our goal from this moment was to ensure that most of the time will be spent by solving the system of linear equations – meaning getting rid of as much other stuff as possible.
    - Optimization would go on gradually, and the steps we were taking were as follows:
      * refactor the way the parameters A and b are being created
      * factor out some calculations that were the same for every step (moving them out of the function not to calculate them every time)
      * creating our own interpolation engine, that is optimized for our purpose
  + After the final step of the optimization, the percentage of linear equation solving in the EvaluateNewStep() function rose up to 80 % - meaning more than **5-fold improvement** in the speed of the whole function, and also the whole simulation.
    - One reminder for the point about the hardware optimizing being generally advised first – running the code on a more powerful computer was able to increase the speed **25-fold**, without any struggles with code optimization.
  + Numpy arrays are very expensive to modify
    - <https://towardsdatascience.com/hitchhiker-tips-on-effectively-using-python-numpy-arrays-66649b7bd5fb>
    - <https://www.jessicayung.com/numpy-views-vs-copies-avoiding-costly-mistakes/>
    - <https://docs.scipy.org/doc/numpy/user/quickstart.html>
  + The performance profiling was done on two computers with very different computational power, which uncovered one point about performance improvements – that generally the better and cheaper option how to make the code run faster is to use better hardware.
    - Reason being the high cost of additional development process (as the development time is quite expensive), and the risk of the optimization breaking something. Usually the optimization also means the code becomes less understandable, making it harder to maintain and enhance in the future, because it was written with only the speed in mind.
  + How to run the line profiling in python
    - <https://github.com/rkern/line_profiler>
    - <https://stackoverflow.com/questions/23885147/how-do-i-use-line-profiler-from-robert-kern>
    - Line profiler we used is not a part of a standard python library, therefore it needs to be installed by a package manager (pip)
      * pip install line\_profiler
    - Then we place a decorator (<https://www.geeksforgeeks.org/decorators-in-python/>) "@profile" above the function we want to test
    - Finally, we run our script.py with following command
      * kernprof -lv script.py
    - The results will show us how much time in total, and as a percentage, was spent on each line of the script – nicely identifying possible performance bottlenecks

# Programming concepts worth pointing out

* Creating a class for material, storing the material properties inside

class Material:

def \_\_init\_\_(self, rho, cp, lmbd):

self.rho = rho # Mass density

self.cp = cp # Specific heat capacity

self.lmbd = lmbd # Heat conductivity

self.Mat = Material(rho, cp, lmbd)

# 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)
  + “*PyQt5 is available under a GPL or commercial license, and PySide2 under a LGPL license.”*
  + *“if you plan to distribute your software commercially you will either need to purchase a commercial license from Riverbank for PyQt5 or use PySide2”*
* 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 (<https://docs.python.org/3.8/library/tkinter.html>)
* 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 [:]

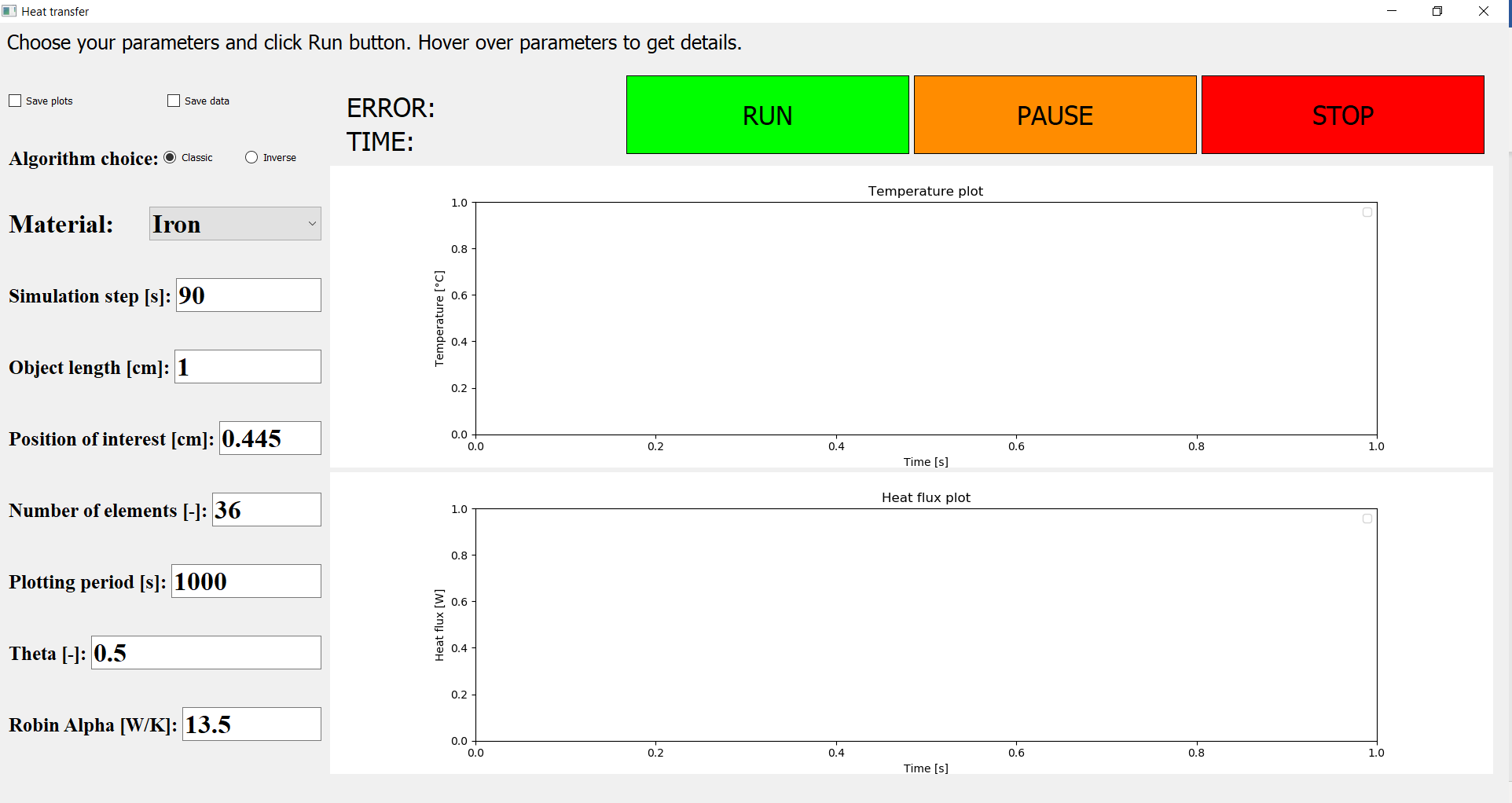
# Basic flow of the software

* How the components cooperate together

# Dependences and libraries

* Requirements.txt file was 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 – therefore it also includes the version of the libraries, that are known to be working well
  + All required libraries can be installed by running “pip install -r requirements.txt”
  + For this, PIP package manager for python must be available, but it comes packaged with all newer python versions - 3.4 or higher
    - <https://www.w3schools.com/python/python_pip.asp>
  + It can be beneficial to setup a virtual environment just for this project
    - <https://medium.com/@boscacci/why-and-how-to-make-a-requirements-txt-f329c685181e>
    - <https://pipenv.readthedocs.io/en/latest/>
  + 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>
* Matplotlib
  + <https://matplotlib.org/>
  + It is the most widely used python library for plotting graphs. It offers countless possibilities of visualising information into 2D graphs, with all imaginable features like multiple or logarithmic axes.
  + Its components also have very good support in python GUI frameworks like Tkinter or PyQt5, which makes Matplotlib the easiest choice for embedding graphs into GUI applications. This is probably the biggest difference and benefit against other data visualisation frameworks, like plotly (<https://plot.ly/python/>)
* Numpy
  + It stands for Numerical Python
  + <https://numpy.org/>
  + <https://www.freelancinggig.com/blog/2018/12/09/what-is-the-difference-between-numpy-and-scipy/>
  + “I*t is used for efficient operation on homogeneous data that are stored in arrays. In other words, it is used in the manipulation of numerical data. NumPy makes Python an alternative to Matlab”*
  + *“NumPy is written in C and it is faster than SciPy in all aspects of execution.”*
  + [*https://github.com/numpy/numpy*](https://github.com/numpy/numpy)
* Scipy
  + It stands for Scientific Python
  + <https://www.scipy.org/>
  + <https://www.freelancinggig.com/blog/2018/12/09/what-is-the-difference-between-numpy-and-scipy/>
  + “*It is a collection of tools for Python. These tools support operations like integration, differentiation, gradient optimization, and much more.”*
  + It is written in python, depends on numpy, and has an open github repository - <https://github.com/scipy/scipy>
* PyQt5
  + It was already described in GUI choice part

# User guide for the software

* Before launching the program
  + File with data names DATA.csv must be placed in the same directory as the .exe program.
  + It must contain comma-separated rows with measurement data regarding:
    - Time from the beginning of measurement [seconds]
    - Temperature measured inside the body [Celsius]
    - Heat flux applied to the body [Watt]
    - Ambient Temperature in the room [Celsius]
  + First line is reserved for the header, and is therefore skipped
* After launching the program
  + On the very top, there is always a short informational message, describing the current state of the application.
  + The whole left side is dedicated for users to specify their preferences.
    - Checkboxes on the top offer saving the results after the simulation – both the plots in .png format, and the calculated data in .csv format.
    - Below users can choose the algorithm they want from the radio button group (meaning only one button can be checked). “Classic” algorithm means we will be determining the temperatures in the body from the knowledge of heat fluxes. “Inverse” algorithm, on the other hand, means we will be trying to estimate the heat fluxes from the knowledge of the temperature.
      * The word “estimate” is there for a reason, instead of the word “determine”, because of the nature of the inverse problems – see the Inverse Problems chapter
    - Below the choice of algorithm users can choose the material from a dropdown menu. The menu consists of all metals, and each material has its own properties, that the calculation will be run with (rho, cp, lambda)
      * Users will also have a choice of defining their own materials, with their custom properties
    - Finally, there is a long list of numerical parameters that users can influence, that in turn influences the simulation. Hovering over the parameter name will show a short description of the parameter.
      * Inverse algorithm is taking two more parameters than the classic one – window span and tolerance.
      * The exhaustive list of all parameters with their description can be found in the Parameters inputted from user chapter
  + Buttons on top are responsible for controlling the simulation – starting it, pausing it or stopping it. When the button is active, it will be highlighted by a thick black margin around the edges.
  + Two labels between the buttons and input parameters are showing the time the simulation is in progress, as well as the final error after the simulation finishes.
  + The main elements on the screen are two graphs, which the results will be plotted into. The upper graph is responsible for showing temperature data, the bottom one shows heat flux values.

# 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
  + One way would be for calculation thread to output results to a file, where they would be discovered and processed by the plotting thread

# Parameters inputted from user

* + - **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
      * theta = 0.0 - fully explicit 1st order, numerically unstable.
      * theta = 0.5 - midpoint (Crank-Nicolson) 2nd order, numerically stable (probably the best choice).
      * theta = 1.0 - fully implicit 1st order, numerically stable.
    - **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

# 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:

* Laptop Acer Aspire E5-575G, processor Intel® Core™ i5-6200U @ 2.30 GHz, 8GB RAM, 64bitový operační systém

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 has the advantage of less other running processes. These running processes could increase the volatility of the results and make it less valuable.

## Methodology of the tests:

* It would certainly be possible to run all the tests by hand, manually inputting the changing parameters we want to test, and recording the resulting simulation time and error.
* However, this process would be very tedious, time consuming and prone to mistakes. Also, there is a direct correlation between the number of tests and the quality of the result, as it is usually much better to obtain multiple values and then average them out, instead of relying just on one value, that can be hindered by a mistake of whatever character.
* Therefore, a whole **testing framework** was developed to improve the speed and the quality of the tests, increase the reproducibility of the results, and offer quick visualisation and feedback.
  + Testing framework has the form of multiple python scripts. They are responsible for defining the parameters and their values we want to test, for running the tests themselves and for evaluating and visualizing the results.
  + All what is needed to use it is to define all possible values of a certain parameter. Simulation will be run with all these values and the results will be saved into a file and also into a graph.
  + We can also choose the number of times the tests will be run – the higher the number, the more stable the results should be, as the average of more tests is more precise than only one test alone. However, with the higher number of tests the overall time will also increase in a linear fashion.
  + There are multiple possibilities how to **define the list of values**, and each of those is best in a specific situation:
    - Manually creating the list
      * values = [1, 2, 3, 4, 5]
      * This way can be efficient when there is a small number of values or we want to choose only specific values, that are not easy to define programmatically
    - Using standard python
      * values = range(start=1, stop=6, step=1)
      * This is the most basic definition of a range of numbers – start at 1, continue until we are lower than 6, and take steps of 1. It creates a range of natural numbers from 1 to 5.
      * It has the disadvantage that it can operate only with integers (whole numbers), and does not support floats (decimal numbers), which is often needed
      * <https://pynative.com/python-range-function/>
    - Using numpy library
      * Numpy offers multiple functions to define a sequence of number values, and each of them has its unique use
      * numpy.linspace(start=1, stop=50, num=100)
        + “Return evenly spaced numbers over a specified interval”
        + We can specify how many numbers we want to get inside the interval – and these numbers will have the same (linear) space between them.
        + Our example returns 100 decimal numbers in the interval from 1 to 50
        + <https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linspace.html>
      * numpy.logspace(start=-5, stop=-1, num=100, base=10)
        + “Return numbers spaced evenly on a log scale”
        + Works the same as numpy.linspace(), but operates on a logarithmic scale – all numbers inside that interval have the same distance between each other from the logarithmic point of view, not from the linear (numerical) point of view
        + This example is returning with 100 numbers in interval from 10-5 to 10-1, which are gradually increasing their numerical distance between each other as they grow
        + This is extremely helpful when we want to evenly span interval containing multiple orders, as that from 10-5 to 10-1, because classical linear spacing would cause the majority of generated numbers being close to the higher value (in this case almost all numbers would be between 10-2 to 10-1, which is not desirable)
        + <https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.logspace.html>
      * numpy.arange(start=0.5, stop=1, step=0.01)
        + “Return evenly spaced values within a given interval”
        + Behaves the same way as classical range() function, but is also working for decimal numbers (floats), which makes it ideal for generating intervals e.g. between 0 and 1
        + <https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.arange.html>

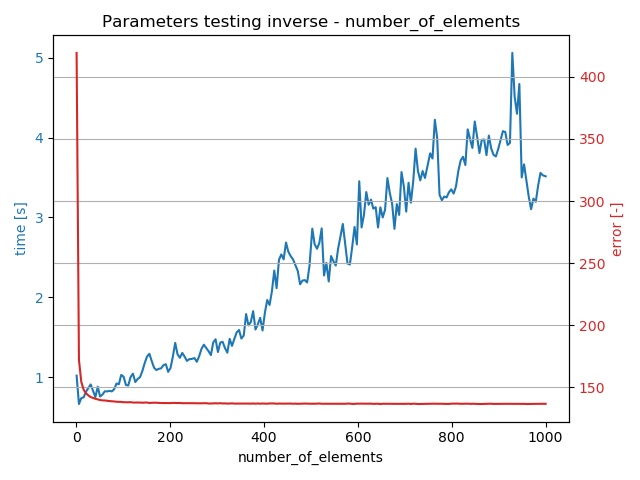
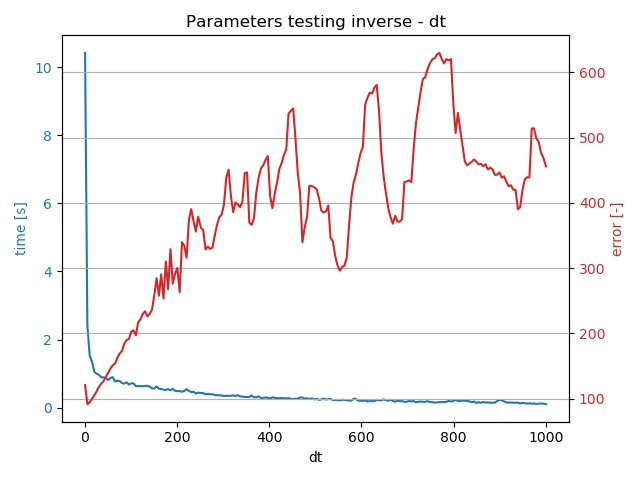
## Results of the parameter testing:

* **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.
    - With the computer analysis of the smallest product of simulation time and error margin, we found out that the optimal number of elements is **36.**
    - **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 steps 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.
    - Computer analysis showed that the optimal value of dt is **90**.
    - **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 error value is decreasing consistently with increasing theta – however, because of rounding the error on 3 decimals, it creates this diagram reminding steps) The time does not look like being dependant on theta at all.
* **Inverse problems**
  + **Number of elements**
    - Here we can observe the same dependence of both simulation time and error to the number of elements as in the classical problem.
    - **Discussion**: The optimal number of elements seem to lie around **40-60**.
    - Computer analysis showed that the optimal value is **15**.
  + **Dt**
    - The same general behaviour as in classical problem can be seen here – higher time steps result in higher error and lower simulation time.
    - What differs, however, is the rate in which the error margin is increasing at the very beginning. Because of this rapid increase in error margin, it seems that increasing time step just to speed the simulation up a little bit makes no sense.
    - **Discussion**: Therefore, we can recommend the time step being lower than in classical problems, around **20**. It should be also mentioned that the very smallest values of time step (1-5) are causing the heat flux plot looking very badly, with a lot of sharp edges.
    - According to computer calculations, the optimal value lies at **33**.
  + **Theta**



* + - In contrast with classical problems, where nothing seemed to depend on theta so much, here in inverse problem, we see an apparent correlation in error margin – it goes up with increasing theta.
    - Simulation time seems to be quite unpredictable, but its minimal values are going down with increase in theta.
    - **Discussion**: Because with increasing theta the time is generally getting better and error is getting worse, it is hard to recommend the optimal value of theta. However, because the error is rising consistently and time is fluctuating a lot, it could be worth recommending the smallest theta possible here – **0.5**.
  + **Window span**



* + - Very interesting and seemingly counterintuitive situation can be found here – with the increase of window span we see both the simulation time and error margin to rise. Also, interestingly, they are rising almost at the very same, linear pace.
    - Explaining the rising simulation time seems easy – if we need to match more windows into the future, it will take longer.
    - It is not straightforward to reason the increase in error margin. It could be explained by “overfitting” (<https://en.wikipedia.org/wiki/Overfitting>) – we were creating a heat flux profile that was matching lot of windows to the future, but this involved creating unnecessarily complex functions that were spoiling the overall result. (??? - VERIFY)
    - **Discussion**: There seems to be an easy conclusion, that it is not worth increasing the window span, and that the recommended value is **1**.
  + **Tolerance**



* + - Beware that the graph has a logarithmic x-axis, because the values range from 1e-8 to 1
    - When changing tolerance, we can observe two straightforward trends – the increase causes the error margin to rise, and the simulation time to fall.
    - This is caused by needing less calculation when the tolerance of the matching is higher – it causes the simulation to be quicker. On the other hand, the result becomes less precise, therefore the error margin keeps increasing.
    - **Discussion**: It seems reasonable to search for the optimal point somewhere **between 1e-4 and 1e-2**, because the error margin has not yet started rising dramatically, and the time is not decreasing so much anymore.
    - According to computer, the optimal spot lies at **0.0014.**
* Theoretically we could automate the determination of the optimal point by searching for a point that yields the lowest result of multiplying time by error (as we want both values to be as low as possible). Because we have all the data from graphs available in a file, it is very easy to do. (However, in this case we should also disregard cases when the simulation is maybe lightning quick but yields spoiled results.)
  + It was done by parameters\_testing\_optimum\_finding.py
  + We just specify the json file with results, and it will analyse all the tested parameters and return the values that yield the smallest product of multiplying simulation time by the error margin
  + However, it is debatable, if the approach of just multiplying time by error is the best comparison, and the error should not be accounted for more. We can paraphrase the saying that “The good feeling from a good price is quickly gone, but the bad feeling from a bad quality still persists” to say that “The immediate satisfaction with quick results is not lasting long, but the superior and more precise results are long lasting.”
* It should be also pointed out that there the word “optimal” is subjective here – every experiment can be different, some require higher level of precision, no matter what the time cost is, others are not so crucial, and some precision can be sacrificed to speed the simulation up to yield quick results.
* What can be also important, is the graphical representation of the result – in the case of higher dt (simulation step) for example, the graph does not look very smooth, as the space between the steps is filled with a linear dependency (a straight line). Therefore, it could be more desirable to rather decrease the value of dt just from this reason.
* Logarithmic axes could be used to visualise the results better, when showing very diverse data

# TODOS AND IDEAS

* Allow for theta less than 0.5, which is currently not possible
* Allow users to specify the location of data file – do not have the requirement of DATA.csv
  + When the software will be used “in production”, there will be no need for the heat flux column, as it will be our goal to determine it, without knowing it
* Allow for custom user material
* Incorporate metals\_properties.csv into the .exe build
* Convert the application to PySide2 instead of PyQt5
* Create a .py file from .ui file at the very end
* Contemplate the visualising of Parameters testing
* Create a diagram of all software components
  + Maybe also a diagram of the functions in the computational engine