Content

[Inverse heat transfer software 1](#_Toc32352566)

[Why Python? 1](#_Toc32352567)

[Architecture 5](#_Toc32352568)

[Programming concepts worth pointing out 16](#_Toc32352569)

[Interesting bugs worth pointing out 18](#_Toc32352570)

[GUI choice 18](#_Toc32352571)

[GUI creation 19](#_Toc32352572)

[Computation engine description 23](#_Toc32352573)

[The basic flow of the software 35](#_Toc32352574)

[Dependences and libraries 36](#_Toc32352575)

[User guide for the software 38](#_Toc32352576)

[Possible features and improvements 40](#_Toc32352577)

[Parameters inputted from the user 40](#_Toc32352578)

[PARAMETERS TESTING 44](#_Toc32352579)

[The methodology of the tests: 45](#_Toc32352580)

[Results of the parameter testing: 47](#_Toc32352581)

[TODOS AND IDEAS 62](#_Toc32352582)

# Inverse heat transfer software

* This document is a draft of the diploma thesis describing the nuts and bolts of the software solution
* **Please excuse the lack of structure here for the time being**

# Why Python?

* Because it is extremely easy to use and has capabilities in all **three programming paradigms** that we need for this project
  + **Scientific computation**
    - As the main core of the project is its computation engine for simulating heat transfer, we need a tool that supports useful data structures (e.g. matrixes) and numerical methods to quickly work with them (e.g. interpolation or linear equations solver).
    - Through the libraries like numpy or scipy (described later), python can gain scientific and numeric capabilities matching or even exceeding other scientific software like Matlab. And opposed to Matlab, it can be used freely without any license.
    - Not only can we run scientific computations, but also the displaying of data in form of graphs can be done easily with libraries like matplotlib.
    - All three abovementioned libraries are often used together in what is known as SciPy stack.
    - Because of the abundance of these libraries, one can even choose from multiple similar implementations of the same functionality, which adds a benefit of being able to experiment with various approaches – that leads to a deeper understanding of the subject.
  + **GUI creation**
    - Python offers multiple libraries for creating GUIs, the basic one called Tkinter is even in its standard library.
    - It has a connection with the well-known Qt platform (through PyQt5 library), which is a complete solution for creating even more advanced GUIs.
  + **General control flow**
    - To allow for good user experience, the computation engine and GUI need to be connected into one piece, and this is done by defining custom application logic.
    - This is probably the area where python shines the most – the ease with which the main logic of the program can be written, thanks to its easy syntax and high readability.
* In general, it can be said that python was chosen because of its easy syntax and the high number of available libraries, that exist for almost any possible use case
* <https://www.python.org/>
* Possible alternatives to python
  + C++
    - Description
      * Lower-level language, that is known for its high speed, but also for its complex code.
    - Advantages
      * Generally, code runs much faster than in python, as the C++ code is being compiled into the machine code, which can run directly on the hardware.
        + Python, on the other hand, is interpreted language, which means it is not compiled into machine code. It allows for the python code to be run almost anywhere. The cost of this is the inferior speed of execution.
        + Also, for the language to be compiled, it needs to be statically typed (the variables cannot change their type – string, integer… - during the runtime), whereas interpreted languages can be dynamically typed (variables can change their type). Being dynamically typed allows for higher freedom, but slows down the execution speed, as the interpreter always needs to check the data type of the used variables – as opposed to statically typed languages, where the compilation makes sure the variables stay the same type, and can, therefore, optimize operations on those variables.
    - Disadvantages
      * Writing usable code in C++ takes much longer than in python, as there are much more syntax rules which to abide by, and the standard functionality of the language (standard library) is far from being so broad as in python, so one needs to write custom code even for operations that are already “built-in” in python itself. (Python is sometimes called a “battery-included language”.)
      * Also, more experience in programming is needed to use C++, because of the higher complexity it requires (programmer is in charge of memory management, etc.).
    - *“Python is much more beginner-friendly, while C++ is a more complicated, low-level language. C++ has more syntax rules and other programming conventions, while Python aims to imitate the regular English language.”*
    - <https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/>
    - *"Python where we can, C++ where we must"*
    - <https://stackoverflow.com/questions/2560310/heavy-usage-of-python-at-google/2561008#2561008>
  + Matlab/Octave
    - Description
      * Computation software for scientists and engineers
    - Advantages
      * Everything in Matlab is designed to work together easily, which is more robust and less prone to unexpected changes that in case of bundling together multiple libraries in python.
      * It comes with the full GUI (development environment) that is simplifying the process of developing and debugging.
        + However, there is a similar IDE included in Anaconda python distribution, called Spyder, which was designed specifically to resemble the Matlab environment (having a console, showing all variables, history of commands, etc.).
      * All the functionality in Matlab is arguably more rigorously tested and validated by the industry than python libraries (which are open source on the other hand, so really anybody can verify their correctness).
    - Disadvantages
      * Matlab requires a rather expensive license to be used for commercial purposes. Octave is for free, but it is lacking some features from Matlab and is generally running much slower.
      * Toolboxes to extend Matlab functionality (can be thought of as additional libraries) are also not for free
      * Matlab itself and all the toolboxes are closed source and developed and maintained by a single company (Mathworks), which makes it a single point of failure – as opposed to the true open-source philosophy of python and its libraries.
    - <https://realpython.com/matlab-vs-python/>
    - <https://www.mathworks.com/products/matlab/matlab-vs-python.html>

# 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).
    - The reason being the specific way we need the interpolated values – all the time we need to interpolate a value on the exact same place in the array.
    - The class in the module is being initialised by an array of values x (in ascending order) and a specific value x0 (or a list of them), which marks the place(s) of our interest.
    - It will determine the exact position of the x0 value in the x array.
    - When supplied an array of y values, having the same dimension as the x array it was initialised with, it will return the y0 value that is on the exact same spot in the y array as the x0 value in regard to the x array (which equals to the interpolated value).
    - At first, we had individual classes for handling the interpolation of a single value and for interpolating a list of values
      * This had a small but measurable speed improvement, as the interpolating function did not have to spend time checking this difference during the execution, thus saving time
      * However, instantiating of the custom class was unnecessary polluting the code, so we moved that deciding logic to the interpolation module itself, using the factory design pattern
        + <https://en.wikipedia.org/wiki/Factory_method_pattern>
      * TODO: reformulate to reflect current situation
* **GUI with integrated graph, menu options, and simulation controllers**
  + **Qt5 Designer**
    - .ui file will be created, which contains 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 the final release it is beneficial to transform it into its own python file
      * Reason being that when converting the application to .exe file, all the non-python files must be then present in the same directory as the .exe file for the imports to work. When creating a python file from it, it will be imported in the main module and will not have to be present after the conversion.
  + **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 customized input field into the GUI is a matter of creating a new record in a dedicated file (heat\_transfer\_user\_inputs\_classic.py)
      * We can easily specify the name of the 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 the user does not have to input everything from scratch before every simulation.
    - It allows for a custom description of each parameter, so the user can have a better idea of what that parameter means – this description will be available in GUI after hovering over the parameter label.
    - There is an input validation in place, so when the field is expecting a whole number, inputting a decimal number or some letters will cause the validation to fail. In that case, the wrong value will be replaced with a default value and the user will be notified.
    - A little challenge was not to confuse users nor computation engine with units (on small objects it is beneficial 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 the 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 could add 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 visible feedback
    - Locking the inputs when the simulation is running, not to confuse the user
    - Hovering over the input variable shows its description
    - Possibility of saving the data and/or plots from the simulation
    - Smoothing panel shown after inverse simulation finishes
* **Multithreaded infrastructure for the GUI**
  + Workers, that enable using more threads and sending signals between those threads (on progress (time), finish (error margin), 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 to produce a working solution)
  + Creating GUI applications is a 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 task is going on (like a heat transfer simulation in our case).
  + The GUI is normally 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 then 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 the 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. In our case user would be able to start the simulation by clicking a “run” button, but after that, all the clicks would be suppressed until the simulation would be over.
  + 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 those background processes), we need to incorporate second thread, whose only purpose would be to run background processes. 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 of calculating the simulation and rendering the results.
  + 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 for overcoming 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 being constantly interrupted by the listening periods, which caused the time for the whole calculation to be higher than without those listening “pauses”. Also, there was 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 it 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. In case of inverse simulation, after it finishes, we also send a signal to show smoothing panel, which can be used to smooth the final results.
    - 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, the python standard library offers a module named queue (<https://docs.python.org/3/library/queue.html>), which we took advantage of. The queue is a data structure that allows for the communication between two software components (like threads). 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 plots in GUI, so that the user can see the simulation in real-time.
      * There were multiple possible ways of achieving this (make it either 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 the 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 a multithreaded solution, there is also a multiprocessing solution
    - The multithreaded solution is generally easier and quicker to set up, it has the benefits of shared memory, but in python, only one thread can be really executing the code at the same time – meaning the speed improvement does not have to be so huge as with multiprocessing 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 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 dropdown menu is created in GUI – all materials defined in our database will be displayed there. The material properties will be loaded into the local memory, ready to be queried, when the user chooses the material for the simulation.
  + We created our own database of material properties, as there seems to be no python module that would satisfy our needs.
    - There is a library <https://pypi.org/project/materials/>, but it only includes a very limited choice of materials, which is not suitable for us.
  + Our material database exists in the form of a CSV file with material properties – **metals\_properties.csv**.
  + It was generated by **fetch\_elements.py** script, which is taking material data from Wikipedia
    - The 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 an 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 is processing the data in a .csv file, and transforming them into python data structures (lists of values) inside its own class
* **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 calculations 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 to 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 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 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 **evaluate\_one\_step ()**, 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, and letting them be instance variables)
      * 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 evaluate\_one\_step() 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 **2,5-fold**, without any struggles with code optimization.
  + TODO: INCLUDE REAL CODE EXAMPLES OF WHAT WAS OPTIMIZED
    - Deep copying is very slow – better to copy arrays with the use of assignment [:]
    - Interpolations.py is a big optimisation by itself
      * Also, the creating of a custom class for float or list to save one if statement during the execution is a nice speed improvement.
  + 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 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.
    - The reason being the high cost of the 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.
    - However, the hardware solution can reach its limits quite quickly, when no parallelisation is used. The reason being the speed of the CPU is not so drastically different when comparing a “weak” computer and “strong” computer. What differs, is the number of cores and threads these computers have in store – therefore to really squeeze the most performance out of a powerful computer, some form of parallelisation is needed (meaning running calculations on multiple threads or processes simultaneously).
  + 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 the 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)

* Creating an interpolation object according to the number of points we are interested in, to save if statements during the execution.
  + This should be covered by a factory design pattern however
* Communication between threads through the shared queue.
* Not printing so much information to the console.

import random

if random.random() < 0.1:

print("sleep")

* Defining how the class should be displayed when using print() on it through the \_\_repr\_\_() magic method.
* Class inheritance between Simulation and InverseSimulation
  + It allows us to use all the code from the Simulation class in the InverseSimulation class without the need for copying it. We can then extend this behaviour by defining new custom variables and methods on the InverseSimulation class – without affecting the parent Simulation class.
  + We also have the possibility of overriding methods from the parent class, by this, we can unite the names of public methods for both classes – even the child InverseSimulation class can have its own public evaluate\_one\_step() method. This way we have a common interface for all the simulations, which results in more readable code (it could be said that both classes implement the same interface).
  + <https://www.thedigitalcatonline.com/blog/2014/05/19/method-overriding-in-python/>
* Version control through git and Github
  + Git is a protocol (version control system) and Github is a service for hosting git repositories.
  + It is a priceless tool in software development, that makes sharing code between more people much easier than other methods (sending emails, using Google Disk, etc.).
  + <https://git-scm.com/>
  + <https://github.com/>
* Usage of pre-commit git hooks to reduce the possibility of committing bugs in the code (rejecting the commit when any of the defined tests fails).
  + <https://githooks.com/>
* Dividing functions and methods into public and private ones by using naming convention
  + Names of functions used only inside a module are starting with an underscore, to signal they are not meant to be called from the outside
  + It makes it easy to quickly tell which functions are used outside of current module, therefore paying more attention when changing their inputs or outputs, because it could break the code in some other module
  + Python natively does not support the convention of public and private methods and attributes, as languages like Java do
* Static analysis of the code through mypy
  + Python is by nature interpreted programming language, and so the lines of code are encountered by interpreter only at runtime
  + One of the advantages of it is superior portability of the code, which can be moved easily as a text file, and modified without issues, as opposed to compiled languages like Java or C++.
  + Drawbacks of it include the impossibility of discovering obvious bugs prior to runtime (like using undefined variables etc.), which can cause unexpected problems later on (especially when these bugs are hidden outside of the main code execution, which is not being well tested).
  + TODO: describe our usage of mypy and its benefits and drawbacks
  + <https://realpython.com/python-type-checking/>
  + <https://medium.com/@ageitgey/learn-how-to-use-static-type-checking-in-python-3-6-in-10-minutes-12c86d72677b>
  + <https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/>

# Interesting bugs worth pointing out

* There can be bugs without programming, but there can be no programming without bugs (own thought).
* <https://softwareengineering.stackexchange.com/questions/195571/is-it-possible-to-reach-absolute-zero-bug-state-for-large-scale-software>
* <https://www.quora.com/Are-there-programmers-who-write-virtually-bug-free-code>
* Users could choke the simulation by clicking PAUSE and STOP button when the simulation was not running, and it caused the RUN button being seemingly unresponsive because in the shared queue there were multiple commands for the simulation to go to paused or stopped state – therefore preventing the simulation from advancing until all the non-running commands were exhausted by clicking RUN button multiple times.
* Not being able to finish the simulation because our last time step exceeded the biggest time value of the simulation, so there was no interpolated value of temperatures and heat fluxes.
* Slight differences in Operating Systems between Windows and Linux caused the GUI written for Windows not working in Linux – additional code had to be written to distinguish between the platforms and call the platform-specific methods.

# GUI choice

* 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 an 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”*
  + <https://www.learnpyqt.com/blog/pyqt5-vs-pyside2/>
* Tkinter is a basic library for building python GUIs, as it already comes packaged in a standard library, therefore there 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:
    - PyQt5 allows 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 would be necessary in the case of the 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 the 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.

# GUI creation

* The basic structure of the GUI is defined by **heat\_transfer\_gui.ui** file.
  + It is created with the use of the Qt Designer application.
  + It offers user-friendly creation of GUIs, which do not require almost any programming skills.



* In the Designer, we create the basic structure of the application – defining the components (smaller windows) on the screen, where data will be rendered afterwards.
  + **(1)** On the very top, we create a label that will be used for showing custom information for the user spanning the whole width.
    - It has a fixed height of 30 pixels, and together with the fixed font size of 15 points, it means the showed text will be always fully visible.
  + **(2)** The whole left side below is occupied by a vertical layout in which all the user inputs and options will be rendered.
    - It has a maximum width of 400 pixels, so it is not increasing unnecessarily at the cost of the graphs.
  + **(3)** The upper right side is dedicated to the control and information panel
    - It has a fixed height of120 pixels, so it is not increasing unnecessarily on at cost of the graphs.
    - It hosts three buttons for controlling the simulation and two labels for getting information about the simulation.
  + **(4)** The biggest area on the screen is there to accommodate graphs of temperature and heat flux progress.
    - This vertical layout can increase as much as possible on the screen, to offer good visibility for both graphs.
* In **heat\_transfer\_gui.py**, we are building on top of the heat\_transfer\_gui.ui, and are filling custom components into the layouts.
  + Function **show\_message\_to\_user()** is created to show arbitrary text in the top info label.
  + Creation of the left user-input side
    - There are multiple functions for rendering all the options
      * **add\_saving\_choices()** is including the checkboxes for the user to choose whether to save results or not (regarding the plot graphs and the numerical data).
        + Both checkboxes are saved as instance variables, and accessing their **isChecked()** method is then indicating if these checkboxes were checked by the user or not.
      * **add\_algorithm\_choice()** is including the radio buttons for the user to choose the algorithm (classic or inverse).
        + We also need to include the label describing the purpose of these radio buttons.
        + To quickly get the current situation from anywhere in the application, function **get\_current\_algorithm()** was created to determine which radio button is currently clicked and return the appropriate algorithm name.
      * **add\_material\_choice()** is adding the dropdown menu with all the possible materials users can choose.
        + We are using **material\_service** here to supply the list of all materials.
        + We can get the currently chosen material by calling the **currentText()** method on the dropdown menu object.
      * **add\_user\_inputs()** is including the labels and input fields for all the input data that are required for a current situation (algorithm).
        + We are using **input\_service** to supply the list of all parameters that need to be rendered.
        + Because the parameters for classical and inverse simulation are not the same, we have to use the right service for this, and it is the job of **get\_current\_input\_service()**, which is returning it according to the current state of radio buttons representing the choice of algorithm.
        + We are looping over all the fields that should be rendered, saving them into instance variables to have easy access to them and setting a default value into all the input fields.
      * TODO: include the data file choice – with the obstacle of showing only the data file name, but having to store the whole path, in case it is residing in some other directory than our main code
      * TODO: add the smoothing panel function, and its obstackles – radio buttons were colliding with the algorithm ones, there was a need for modification
      * TODO: describe the way how new input fields are getting there when changing the algorithm – that we need to delete the old data and include new data
    - All the functions are displaying the information in a similar way – for each row, they are creating a horizontal layout with all the various components (labels, checkboxes, radio buttons or input fields) and including this new layout in the parent vertical layout, defined at the very beginning in the Designer.
  + All buttons are connected with their appropriate functions
    - **run\_simulation(), pause\_simulation(), stop\_simulation()**
    - Also, there is added highlight effect of creating a thick black border around the button that was clicked and is currently active.
    - Buttons are set to be responsive only when it makes sense (the PAUSE and STOP button are not performing anything at the beginning because there is no simulation to be paused or stopped; the RUN button is not working when the simulation is currently running).
  + Functions are created to change the text in the error label and time label - **update\_time\_label()** and **update\_error\_label()**.
  + Canvases are created to represent plots as objects and are inputted to their dedicated layout as widgets.
    - Both plots are defined in their specific files, **heat\_transfer\_plot\_temperature.py,** and **heat\_transfer\_plot\_heatflux.py** respectively.
  + All the components have the smallest possible sizes, so even in the smallest possible window size, everything will be proportionate and visible.

# Computation engine description

* What equations are used, how they look like in python
* The main module of the computation is **NumericalForward.py,** which contains the **Simulation class**
  + Simulation class is responsible for initializing and keeping track of all variables necessary for the computation, as well as defining the functionality of working with these variables to perform the simulation.
  + Simulation class needs to be initialized with multiple parameters, which are describing the characteristics of the current simulation. These parameters are identical to the parameters that can be specified by the user in the GUI (and they are already described in that part pretty well).
    - Number of elements in the model (N)
    - Time step (dt)
    - Theta (theta)
    - Material properties (material)
    - Convection heat transfer coefficient (robin\_alpha)
    - Path to file with experiments data (experiment\_data\_path)
    - Place where the measurements were taken (x0)
    - The overall length of the object (length)
  + After being inputted all these values, it will generate multiple internal variables, that will be used within the simulation.
    - The array of time values (self.t) in which we will be evaluating the simulation.
    - Arrays of Temperature (self.T\_data), HeatFlux (self.HeatFlux) and Ambient Temperature (self.T\_amb) with values corresponding to the time values.
      * These arrays are made by interpolating the inputted experimental data by numpy.interp function, to evaluate the measured data at the time values the simulation will be using (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.interp.html>).
      * All the interpolation is made at once in the very beginning, which is very beneficial, as the interpolation function can be called only once, and not during every calculation step, which speeds the process up considerably.
    - Size of one elements (self.dx), calculated by dividing the overall object length by the specified number of elements.
    - The array of positions in the objects (self.x), where we will be determining the temperature, according to the number of elements.
    - The array of temperatures at the measured point (self.T\_x0), which will hold the values of temperatures at the point of interest in all the time values of the simulation.
    - Interpolation class will be created and assigned locally to quickly interpolate the temperature at our point of interest (x0) from the temperature distribution in the whole object.
    - Parameters needed to construct the system of linear equation
      * M
        + Tridiagonal sparse mass matrix (contains information about heat capacity of the elements and how their temperatures react to incoming heat)
        + self.M = csr\_matrix(self.dx\*self.rho\*self.cp\*diags([1/6, 4/6, 1/6], [-1, 0, 1], shape=(N+1, N+1)))
      * K
        + Tridiagonal sparse stiffness matrix (contains information about heat conductivity and how the elements affect each other)
        + Being also described as conduction matrix

*“Stiffness matrix is a general term for a matrix of known coefficients being multiplied by unknown degrees of freedom, i.e., displacement OR temperature, etc. Thus, the element conduction matrix is often referred to as the stiffness matrix.”*

*“Finite Element Analysis is a numerical approach realistically possible only with computer processing, based on finite differences. It divides a region into smaller ‘elements’, which are individually analyzed, in order to derive an approximate response of the overall system.”*

*“The fundamental concept of FEM is that a continuous function of a continuum (given domain ) having infinite degrees of freedom is replaced by a discrete model, approximated by a set of piecewise continuous functions having a finite degree of freedom.”*

<http://www.eng.fsu.edu/~shih/eml3050/students'%20presentation/group-4/heat.ppt>

<https://abaqus-docs.mit.edu/2017/English/SIMACAEANLRefMap/simaanl-c-mtxgenerationgeneral.htm>

* + - * + self.K = csr\_matrix((1/self.dx)\*self.lmbd\*diags([-1, 2, -1], [-1, 0, 1], shape=(N+1, N+1)))
      * A
        + Tridiagonal sparse matrix
        + Unit is J/K
        + self.A = self.M + self.dt\*self.theta\*self.K
      * b
        + Vector
        + Unit is J
        + self.b\_base = self.M - self.dt\*(1-self.theta)\*self.K
        + self.b = self.b\_base.dot(self.T)
      * TODO: DESCRIBE THEM, AND SHOW HOW THEY LOOK IN THE CODE (even though they will be certainly described in the other part of the thesis regarding the math stuff)
      * TODO: describe sparse matrixes and why are they useful
        + They are matrixes that do not store zero elements, therefore make the matrixes consume less memory and make the operations with their non-zero elements faster.
        + It reduces scanning time (traversing the matrix)
        + <https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html>
        + <https://www.youtube.com/watch?v=Lhef_jxzqCg>
        + <https://www.youtube.com/watch?v=bOmAllndo-4>
        + <https://www.youtube.com/watch?v=iqZKBptJV2U>
  + Its main method is **evaluate\_one\_step()**, whose job is to simulate one step of the simulation – constructing the necessary matrixes and solving the system of linear equations.
    - The result of evaluate\_one\_step() is a new temperature value being added into array storing the temperature history at the point of interest (x0)
    - When performing the function, it will also increment the internal variable of the current step, which means when we call the function again, it will start calculating the next step.
    - The function needs some higher function to call it and to control how many times it should be called to finish the simulation (as it is only focusing on calculating that one step, and is not interested in anything else – which is exactly what a function should do).
    - The system of linear equation is being solved by scipy.sparse.linalg.spsolve() function
      * <https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html>
      * TODO: try this in some other examples and describe its workings
  + **Plot()** method
    - Is responsible for updating the graphs in the GUI
    - It takes a reference of the temperature and heat flux graphs, and is calling their plot() method with the current values of temperature, that were calculated so far
    - The heat flux graph is updated only once at the very beginning, as it contains only the experimental heat flux. This heat flux is not changing throughout the simulation, so there is no need of updating this graph again.
  + **Save\_results()** method
    - Is outputting numerical results of the simulations into a CSV file, containing the time values and their appropriate temperature values
  + **After\_simulation\_action()** method
    - Defines what should happen after the simulation finishes
    - Is calculating the error margin at the end of the simulation with the use of **calculate\_final\_error()** function
      * Error is being calculated as an average difference between the calculated temperature and the real temperature in all the time points that are being calculated
  + It has a property self.**simulation\_has\_finished,** which is always reflecting the simulation state, if it has finished or not
    - We decide on this by comparing the index of the current step with the index of the last step of the simulation
    - It is being used to determine whether to stop calling evaluate\_one\_step() method by some higher function
* Next important module for the simulation is **heat\_transfer\_simulation\_utilities.py**, which defines the infrastructure for the simulations to be run
  + It is a general module for running any simulations, that can be customized by will
    - It can be used to run any simulation as long as this simulation will expose the same public methods and properties (as long as the simulation implements a suitable interface)
      * evaluate\_one\_step() method to determine next step of the simulation
      * plot() method to update results in the graphs visible in GUI
      * save\_results() method to save the results of the simulation
      * after\_simulation\_action() method to perform custom actions or calculation after the simulation finishes (like calculating the error etc.)
      * simulation\_has\_finished property to act as a stopping point where to stop calling evaluate\_one\_step()
      * current\_t property to store the last simulation time, so that we know when to perform the graph plotting through the callback
      * error\_norm property for determining the error value after the simulation finishes
  + **Callback class** is responsible for updating the plots with calculated data, and also serves as a communication channel with the main GUI thread in case of simulation through GUI.
    - It is being initialised by multiple parameters
      * progress\_callback, which is a way of communication, by which the GUI is being informed about the progress of the simulation (if it is running or not)
      * call\_at, which specified how frequently should the plots in GUI be updated, in simulation seconds
      * temperature\_plot, which is a reference to the plot showing the temperature data
      * heat\_flux\_plot, which is a reference to the plot showing the heat flux data
      * queue, which is the communication channel through which the GUI is controlling the simulation
    - It has one method **\_\_call\_\_()**, which is responsible for updating the plot in regular intervals
      * (\_\_call\_\_ is so called magic method which is a special class method defining the actions when we directly call the object without specifying any method)
        + TODO: include source to python documentation
      * It is also reading the data from the shared queue, where it is looking for commands coming from GUI – whether to change the state of the simulation (stop it, pause it, continue) – and relays these commands to the simulation.
      * At the end of each call we are emitting values for the GUI, signalling whether the simulation is in running state or not. GUI is listening for these values and is increasing the simulation time shown accordingly.
      * We are returning the current simulation state that resulted from the user input from GUI (if any), so the higher function calling it can act accordingly to this simulation state.
  + **SimulationController class** is responsible for preparing, initialising and running both the simulation and the callback.
    - It holds both the simulation and the callback as its own variables and controls the functions that are called on them
    - It takes quite a big amount of parameters
      * The majority of parameters is the same as for the initialisation of the Callback
      * Simulation object, which is basically an instantiation of the Simulation class with all the specified parameters
      * Save\_results, which is determining whether we should be saving the results or not
    - Method **complete\_simulation()** is calling the callback and simulation’s evaluate\_one\_step() method, while the simulation is not over (until the simulation\_has\_finished property on the simulation object does not return true)
      * It is mediating the communication between the callback and the simulation (it is a one-way communication only, in this direction, when the callback is able to change the simulation state after this change is requested from the GUI).
      * After the simulation is over, it makes sure the temperature and heat flux graphs get updated with the final results
      * It also calls custom method of the simulation (after\_simulation\_action()) in which the simulation itself can define arbitrary logic, while having access to all the connections with the GUI
      * If wanted, it saves the numerical results into a file
      * It returns the error value of the simulation, so this value can be displayed in GUI
* Specifically for simulating the classic problem, there is a module **heat\_transfer\_simulation.py**
  + It uses all the behaviour defined in heat\_transfer\_simulation\_utilities.py and defines a custom simulation using the Simulation class from NumericalForward module
  + **create\_and\_run\_simulation()** method
    - From inputted parameters creates the Simulation object, uses it to instantiate SimulationController object, and then calls its complete\_simulation() method, which equals running the whole simulation
  + **simulate\_from\_gui()** method
    - Connector between GUI and create\_and\_run\_simulation() method
    - Takes all the parameters from the GUI and relays them further, returning back the results
* For the calculation of the inverse problem, there is a module **NumericalInverse.py**
  + It defines a class **InverseSimulation**, which is inheriting from Simulation class, so it can use all its internal properties describing the temperatures, heat fluxes, etc., as well as all the methods, when needed (and it is free to override this behaviour)
  + It takes multiple parameters on initialisation
    - Apart from the same parameters as the parent Simulation class
    - All of these parameters are already described elsewhere
    - window\_span
    - tolerance
    - init\_q\_adjustment
    - adjusting\_value
  + Its method **evaluate\_window\_error\_norm()** is determining current error norm, that we have achieved with our estimation of current heat flux
    - We are looking at the differences of the measured temperatures and the temperatures resulting from the simulation with our estimated heat flux
  + It has methods for saving the current index and current calculated temperature, as a form of a checkpoint - **make\_checkpoint()**, as well as for reverting to that checkpoint, when the results of the current inverse step are not satisfiable - **revert\_to\_checkpoint()**
  + For simulating multiple steps at once we have function **evaluate\_n\_steps()**, which we are calling to evaluate simulation after our window\_span
  + The main function of the class, **evaluate\_one\_step()**, is responsible for running one whole step of the inverse problem
    - To comply with our simulation interface, it has the same name as in a Simulation class.
    - It is guessing the value of heat flux at the current moment, running the evaluate\_one\_step() function and according to the resulting error value, it is adjusting the heat flux little bit and runs the evaluate\_one\_step() again, until the error value is acceptable (the difference between the previous error and the current error is less than the tolerance the class was initialised with)
    - The process of adjusting the heat flux according to the resulting error value is as follows
      * From the input argument to the function we get the value of heat flux (in Watts), that will be used to adjust the estimated heat flux, in the form of addition and subtraction of this value to the current heat flux
      * In the beginning, we assign the current value of heat flux (the explicit portion) and the next value of heat flux (implicit portion) to equal the previously calculated heat flux value, as the new values most probably lie in the approximately same region as their preceding value
      * We determine the initial error value from the default temperatures in the T\_x0 array (which is 0.0 degrees Celsius) so that we have some value to compare the errors with
        + TODO: DISCUSS THE CHANGE OF INITIAL prev\_Error TO 0.0
      * In the theoretically infinite loop, we are evaluating the number of steps according to the window\_span variable and calculating the error we got (the assigned values of heat flux are being considered in the evaluation)
      * After each simulation, we compare the new error value with the previous one, and there can be three possibilities:
        + If their difference is smaller than the chosen interval, we accept the estimated heat fluxes and perform the evaluate\_one\_step() function, with which we finalize the decision to use these heat fluxes
        + If the new error is smaller than the previous error, we know our adjustments are going the right direction

We continue with adding or subtracting the same adjusting value of the heat flux for the next step

* + - * + If the new error is higher than the previous one, we can assume we have gone too far with adjusting in the current direction

Therefore, we will switch the direction of adjusting (subtracting the adjusting value instead of adding it or vice versa)

We must also decrease the adjusting value (having it the same would just cause an infinite stepping between three points)

We do both the direction switching and adjusting value decreasing by multiplying the current adjusting value by (-0,7)

POSSIBLE IMPROVEMENT

Wouldn’t it be better to assume that the right heat flux lies between the previous value and current value when the error suddenly increased?

By this, we could easily isolate the smaller interval in which we could look for a good solution

From my simplified point of view, it looks like the halving of the intervals should be good here, but I am probably unaware of some characteristics of these functions (not having linear behaviour, etc.)

* + - * + Note: The error values do not have to be in abs() themselves, as they cannot be negative, from the way the evaluate\_window\_error\_norm() function works

It is performing the summation of squares of temperature differences in certain time points

* + - * The reason we are evaluating the difference of two successive error values, and not the error values itself, is that it may happen that error value will converge to some limit value, and will not decrease further (in that case we would be potentially creating an infinite loop)
        + Because of this possible scenario, we are monitoring the progress of the error value in time, and if we find that this value is not changing much anymore (meaning the absolute difference between two consecutive values is less than our tolerance), we will accept the current estimation of heat flux and we will finish the step
        + ??? - VALIDATE THIS
      * POSSIBLE IMPROVEMENT
        + Why not **checking for both** the difference of two errors being less than tolerance and also the errors themselves in abs() being less than the tolerance
        + Because it might happen, that we will skip a very good result (close to 0) just because previously we encountered some very bad result, and the difference between those is bigger than the tolerance

There might be some reasons we want the solution to “converge”, instead of just picking the right result (that we got randomly), so please correct me if it is the case here

I can also compare the simulations with the current method and with checking for both approaches, and evaluating the improvements if any

* + - * One possible (but rather theoretical) bottleneck of the abovementioned approach could be the case of the first calculated temperature closely matching the default temperatures in the T\_x0 array (which is 0.0)
        + It could cause the two successive errors to be almost identical, therefore causing the (probably unsuitable) heat fluxes to be accepted, which is undesirable
        + This could be avoided by making the default value of T\_x0 array some unreachable value, like -274.0
        + Also, we could make the prev\_Error at the beginning be infinity, so it could not be matched
        + All have the possible drawbacks that the very first guess can be infinitesimally correct, but it will not be accepted, because it is far away from the first (random) value

This actually leads me to the idea of **making prev\_Error 0.0 at the beginning**, as it seems to solve all our problems

It would cause the second error being higher than 0, so we would be already

* + - * + TODO: AGREE ON THIS

Even after it gets “fixed”, we can still write about it being there in the past and advocating the improvement

* + - At the very beginning of the function, it calls the **make\_checkpoint()** function on the Simulation object, which is saving the current temperature distribution (self.T).
      * The reason for this is that we want to be able to experiment with the various values of heat flux and seeing what effects it has on the temperature distribution in the next step. However, from the way we are tracking the temperature distribution, we can always access only the temperature distribution in the last step.
      * Therefore with the checkpoint, we save the temperature distribution before we will experiment with the heat fluxes, and after each unsuccessful experiment, we will revert the previous state of temperature distribution by running the **revert\_to\_checkpoint()** method on the Simulation object. After reverting, we are ready to adjust the heat flux a little bit and try to simulate again from the same starting point.
  + There are multiple methods for smoothing the final result after simulation finishes
    - \_perform\_smoothing()
    - \_smooth\_the\_result()
      * \_smooth\_boundary()
    - \_revert\_the\_smoothing()
* For the simulation of the inverse problem, there is a module **heat\_transfer\_simulation\_inverse.py**
  + The simulation there is conducted a very similar way as described above in heat\_transfer\_simulation.py
  + We are instantiating Simulation(), InverseProblem() and InverseCallback() as internal variables into InverseSimulationController class object, preparing the simulation and then running all the simulation steps

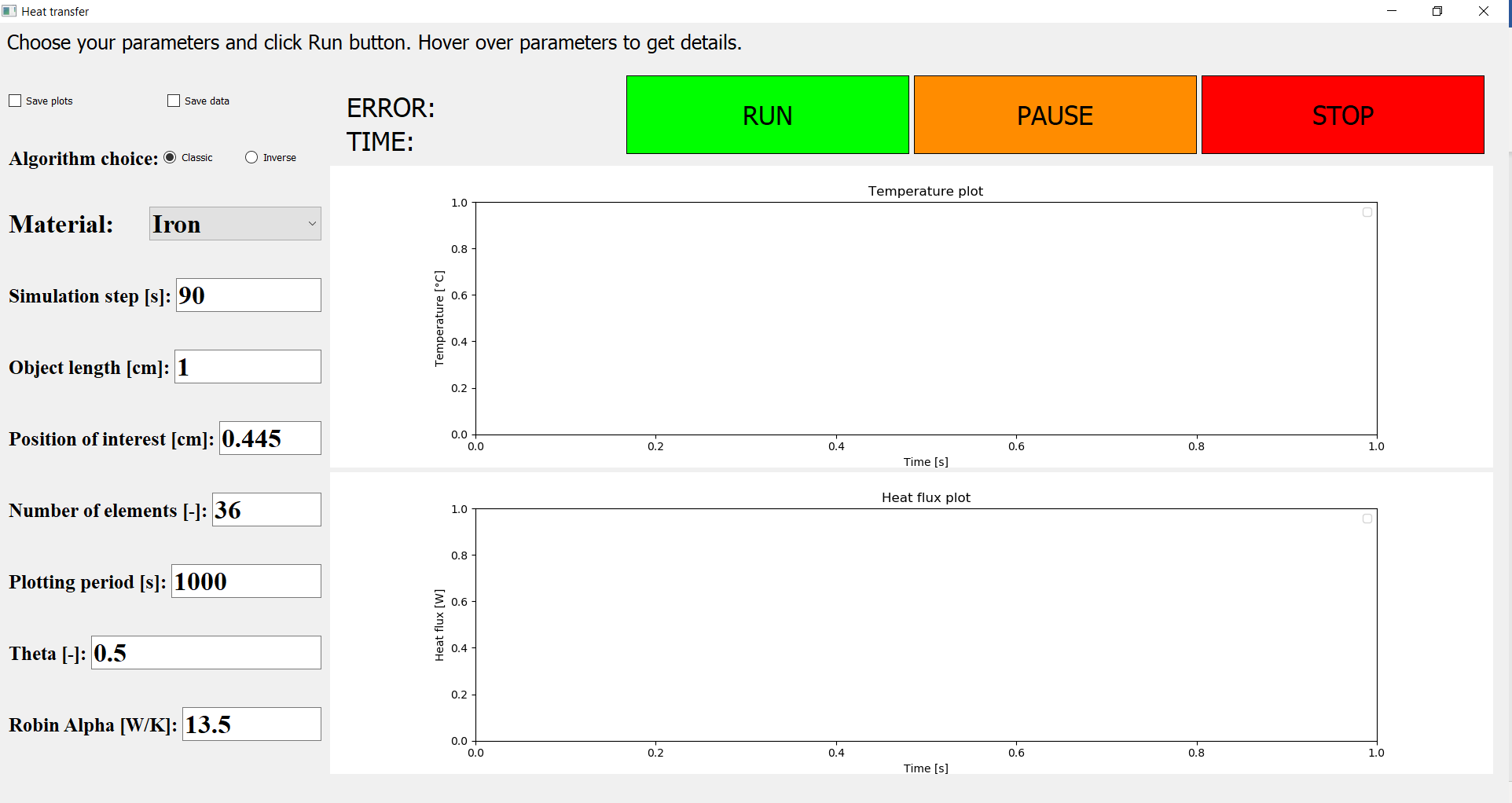
# The basic flow of the software

* How the components cooperate together
* When the user chooses all the desired parameters at the left side and clicks RUN button, the following happens:
  + The internal state of the application will be changed to a running state.
  + All internal variables will be put to their initial values before simulation (e.g. the ones tracking the simulation time).
  + The physical properties of the chosen material will be determined.
  + All the number parameters will be extracted and parsed to comply with defined structure (e.g. number of elements must be a positive whole number).
    - TODO: describe the working of get\_numbers\_from\_the\_user\_input() function that is responsible for this
  + References to both plots and to the shared queue (communication channel) will be bundled together with all the parameters in one big object.
  + Simulation thread will be initiated – the function to perform will be determined according to the type of algorithm (classical or inverse) chosen by the user. The big object created above will be inputted as an argument to this function.
  + Further communication channels with the simulation thread will be set up so that GUI can receive updates from the simulation thread.
  + The thread will be started, and the simulation begins.

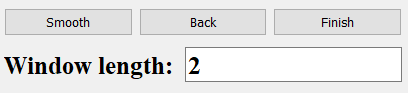
# Dependences and libraries

* 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 set up 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>
* <https://realpython.com/matlab-vs-python/> - the overview of the below mentioned scientific libraries
* 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.”*
  + *“The performance-sensitive parts of NumPy are all written in the C language, so they are very fast”*
  + [*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]
  + The 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 to 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, which in turn influences the simulation. Hovering over the parameter name will show a short description of the parameter.
      * The 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.
  + After the inverse simulation finishes
    - There is a possibility of smoothing the results appearing in the left menu



* + - We are smoothing the results by calculating the average value of x nearby points around each point and assigning this as a new value for that specific point. By this, we are getting rid of extremes and make the whole graph look smoother.
    - The window length is freely customizable, any integer can be inputted. Generally, lower numbers are fitting when the simulation step (dt) was high. When we conducted simulation with a small dt (like 1 second), there is a lot of points on the graph, therefore also the window length should be higher to achieve the desired smoothing effect.
    - There is a possibility of reverting the previous state by clicking “Back” – we store all the past results, so users can freely experiment with different window lengths and then come to the very beginning.
    - Clicking “Finish” will accept the current situation of the heat flux, finish the simulation and calculate its error.

# Possible features and improvements

* Storing user preferences
* Storing the 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 the 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 an 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 an 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 an 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 the 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 an 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 an arbitrary positive whole number
      * Is expressed in seconds
    - **Theta**
      * Determines explicitness (0) and implicitness (1) of the algorithm approach.
      * Explicitness means we are more focusing on matching already calculated values rather than on matching the future values from the measurement
      * Explicit method is much easier to implement, but tends to be unstable and requires very small timesteps
  + The implicit method, on the other hand, is more difficult to implement and usually requires more iterations per one timestep. However, they are more stable and can handle bigger timesteps, therefore compensating the calculation difficultness
    - * Optimal value is 0.5
        + With the increase, we can observe a slight decrease in 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 an 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.
      * When the simulation is explicit, there could be only a small error at the beginning, and it will be gradually increasing the magnitude (exponentially) and will cause the whole simulation to be worthless
  + There needs to be a small time\_step dt, otherwise, the solution will go out of control
  + There is a special value of dt, which it cannot exceed
* <https://www.researchgate.net/post/What_is_the_difference_between_the_implicit_and_explicit_formulation_in_heat_transfer_analysis>
  + - **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 the inverse simulation.
      * **Assumption**: Increasing the value will cause the simulation to take longer but has no big positive impact on 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 an 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 an arbitrary decimal value (float) and has no units
    - **Initial Heat Flux Adjustment**
      * Only for inverse simulation
      * How big should be the initial step when adjusting Heat Flux
      * **Assumption**: Its ideal value depends on the character of the data – when the flux is expected to change a lot throughout the experiment, higher values could be better.
      * Can be an arbitrary float value, is represented in Watts
    - **Adjusting Value Coefficient**
      * Only for inverse simulation
      * By what parameter should the adjusting value be multiplied by after the error increases in comparison with the previous step
      * **Assumption**: The ideal value will probably be different for each experiment, and will also be rather random
      * Should be a decimal value between -1 and 0 and has no units

# PARAMETERS TESTING

In order to have a 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": 10,

"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": 4,

"tolerance": 1e-05,

"init\_q\_adjustment": 20,

"adjusting\_value": -0.7

} … 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 a 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 fewer other running processes. These running processes could increase the volatility of the results and make it less valuable.

## The 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.
  + The 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 that is needed to use it is to define all possible values of a certain parameter. The 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 a higher number of tests, the overall time will also increase in a linear fashion.
  + There are multiple possibilities for **defining 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, like 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>

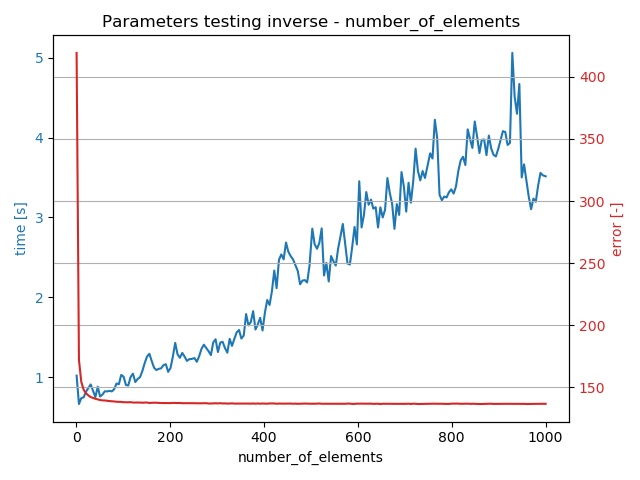
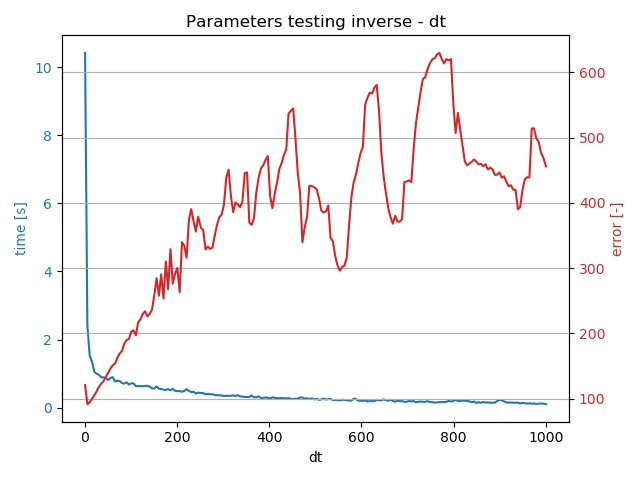
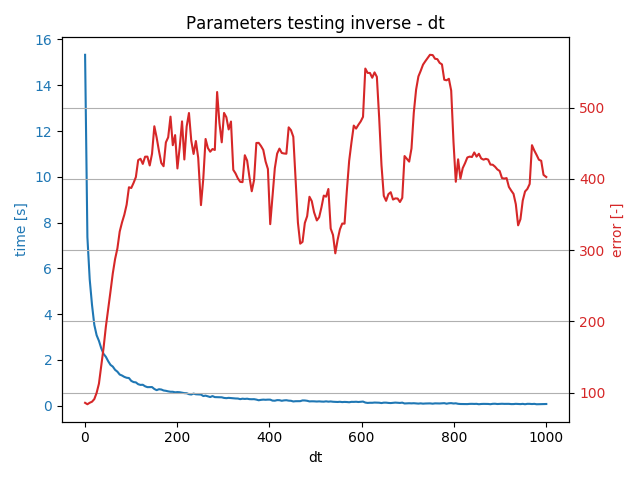
## Results of the parameter testing:

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



* + - We see that with the increasing number of elements in our grid the simulation time is increasing in an 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 a 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 a 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.
    - The reason for this behaviour is that with the higher values of dt there are fewer 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.
    - The 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**

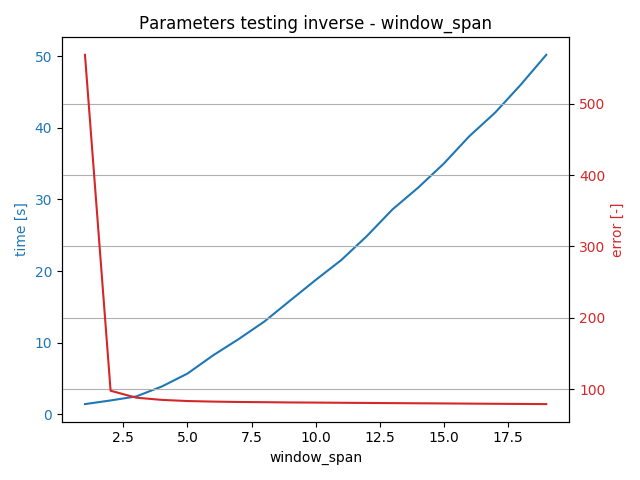


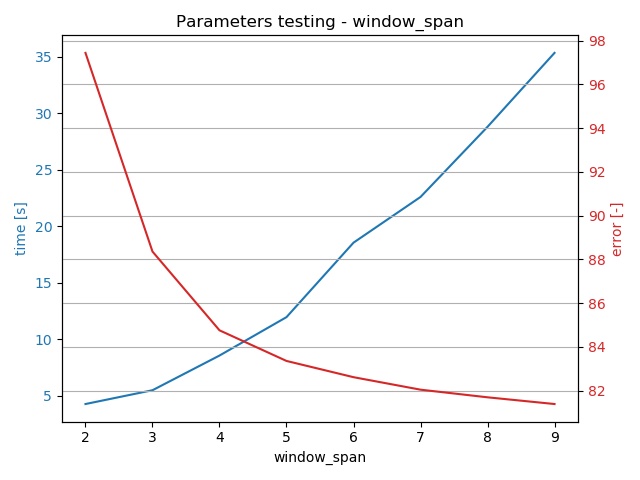
* + - The increase in theta is followed by a very small decrease in error. (The error value is decreasing consistently with increasing theta – however, because of rounding the error on 3 decimals, it creates this diagram reminding steps) The time does not look like being dependant on theta at all.
* **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 a 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 the 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**.
    - When we carried out tests with higher window span (7), as can be seen in the right picture, the response in error margin was much sharper at the beginning, than in lower window span (2). This shows that generally, the higher window span causes the error being more influenced by the rising timestep – it could be said that that higher window span requires lower timestep to function at its best.
  + **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 the 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**

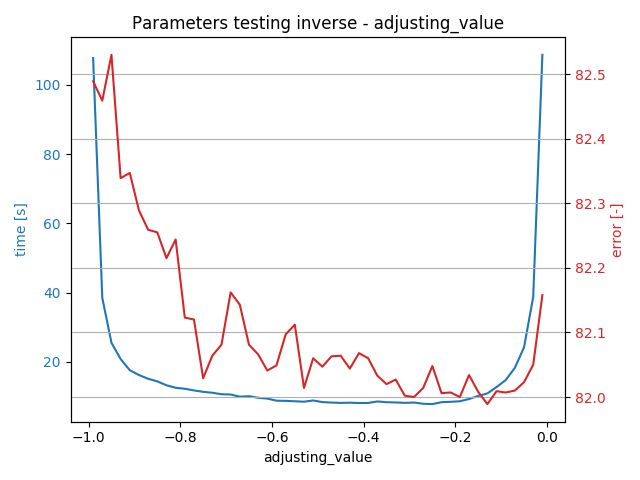
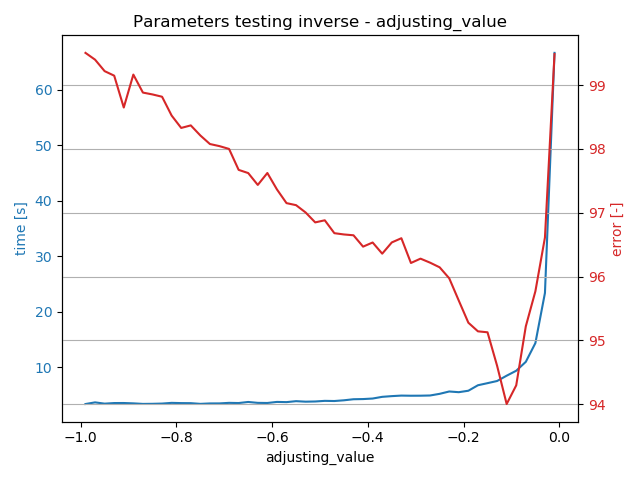


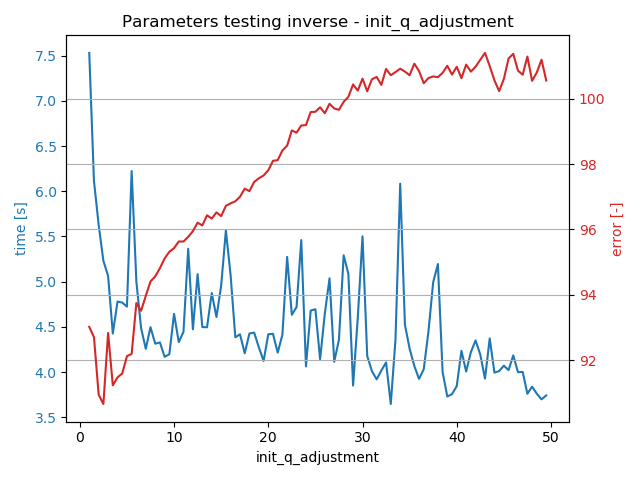
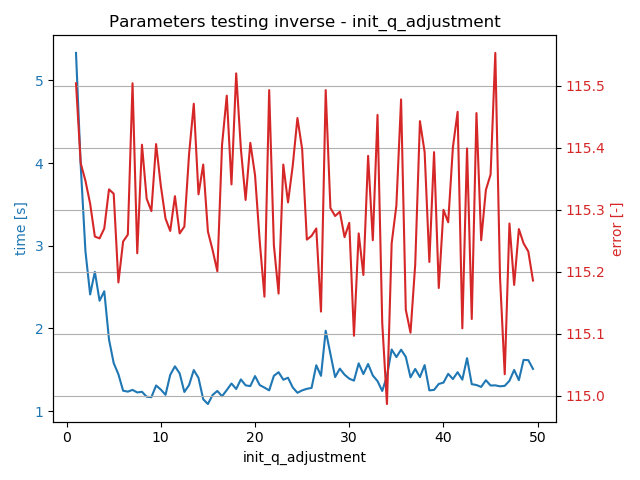
* + - The very interesting and seemingly counterintuitive situation can be found here – with the increase of window span we see both the simulation time and error margin to rise. Also, interestingly, they are rising almost at the very same, linear pace.
    - Explaining the rising simulation time seems 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 a lot of windows to the future, but this involved creating unnecessarily complex functions that were spoiling the overall result. (??? - VERIFY)
    - **Discussion**: After fixing the logic of simulating more than 2 window spans, we came to the conclusion that increasing the window span makes the calculation more precise, however, at the cost of time complexity. The ideal value of the window span could be around **6-8**.

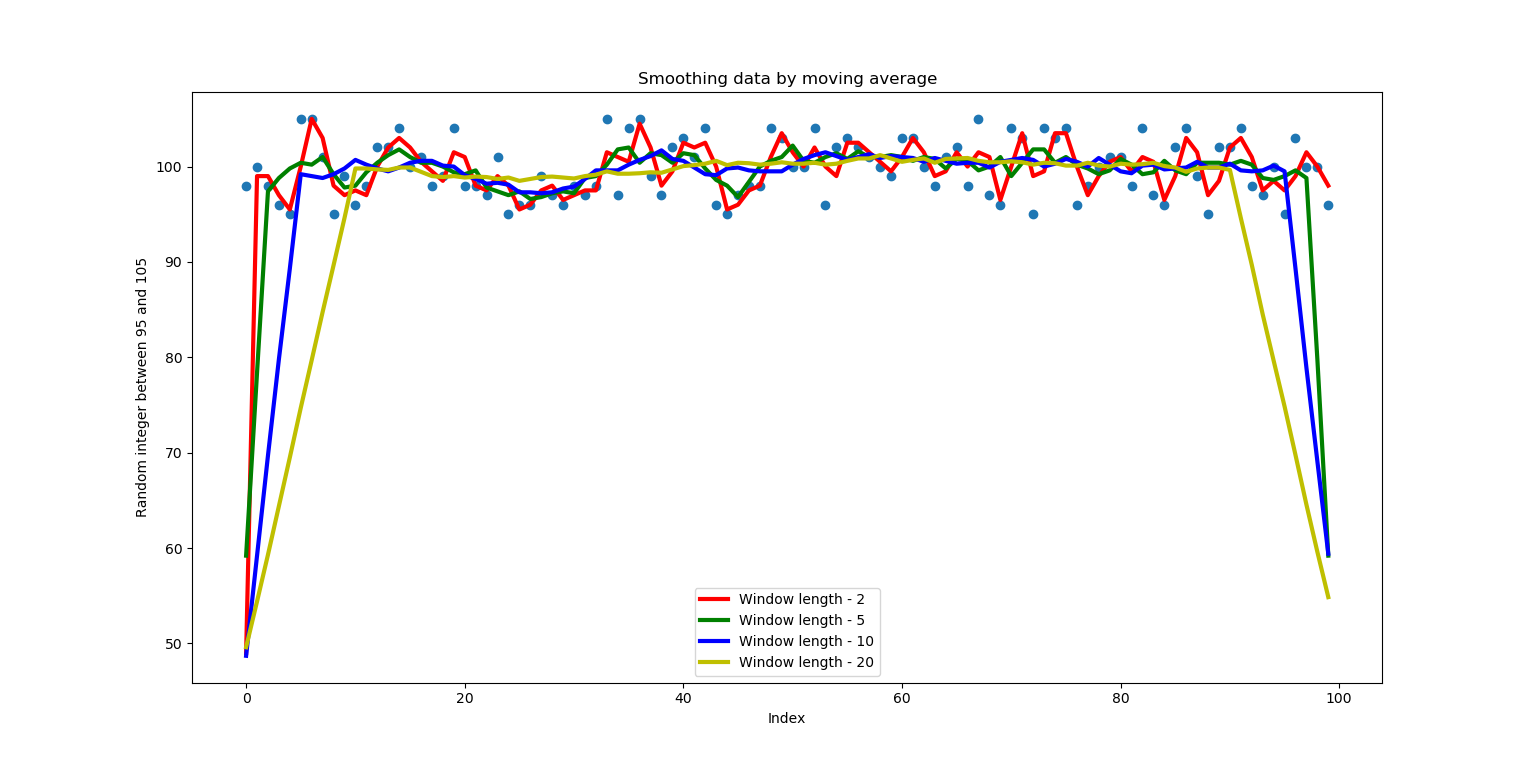


* + - The abovementioned behaviour was found very strange, so we were encouraged to review the logic for inverse simulation, and thanks to it we discovered a bug inside.
      * Even when we chose a higher window span than 2, we were modifying the heat flux values only for two first windows, and others were unassigned, which was causing the error to be higher with the rising window span.
    - After changes the dependence looked already more realistic – with the higher window span, the error was decreasing. However, the simulation time went rapidly up.
  + **Tolerance**



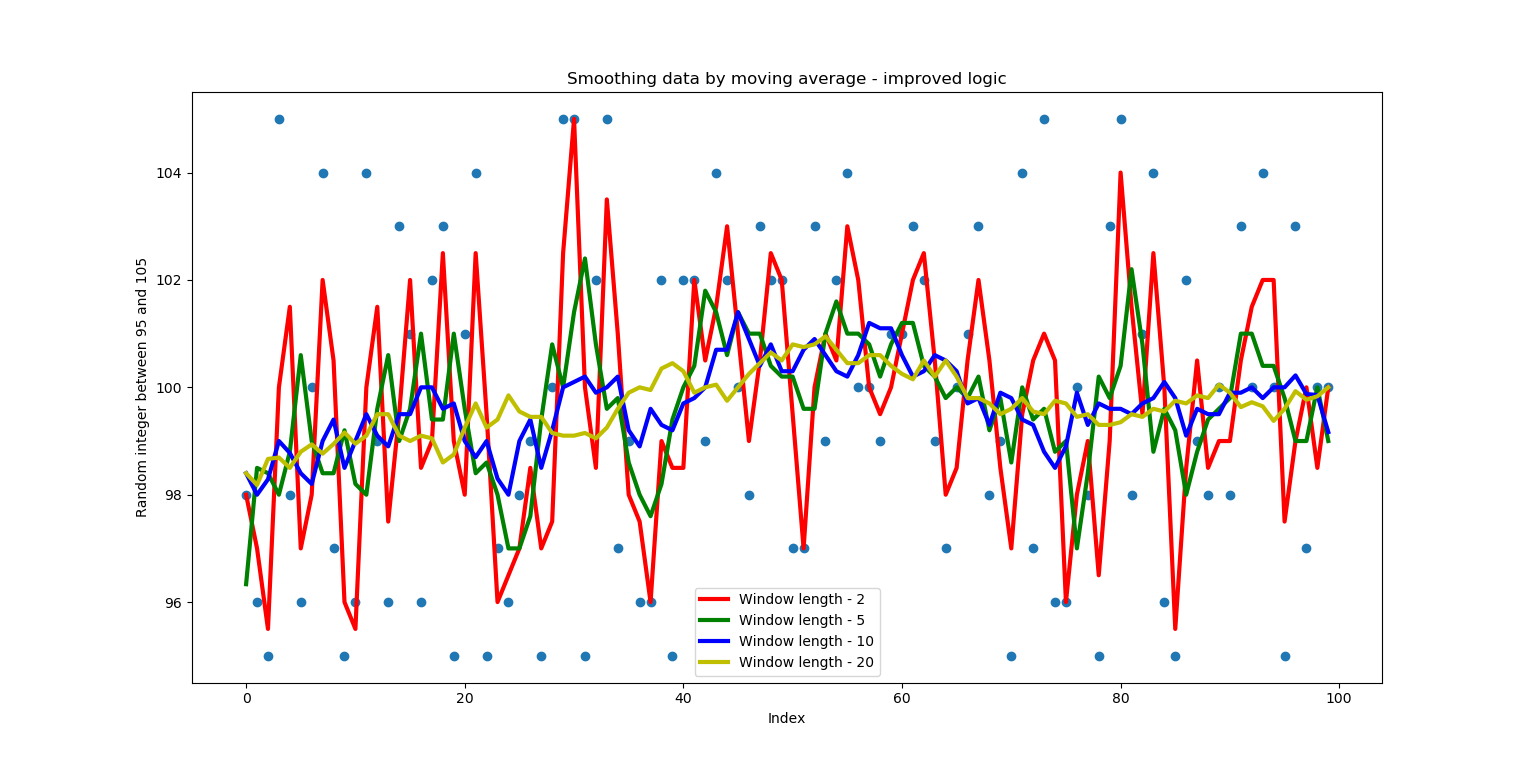
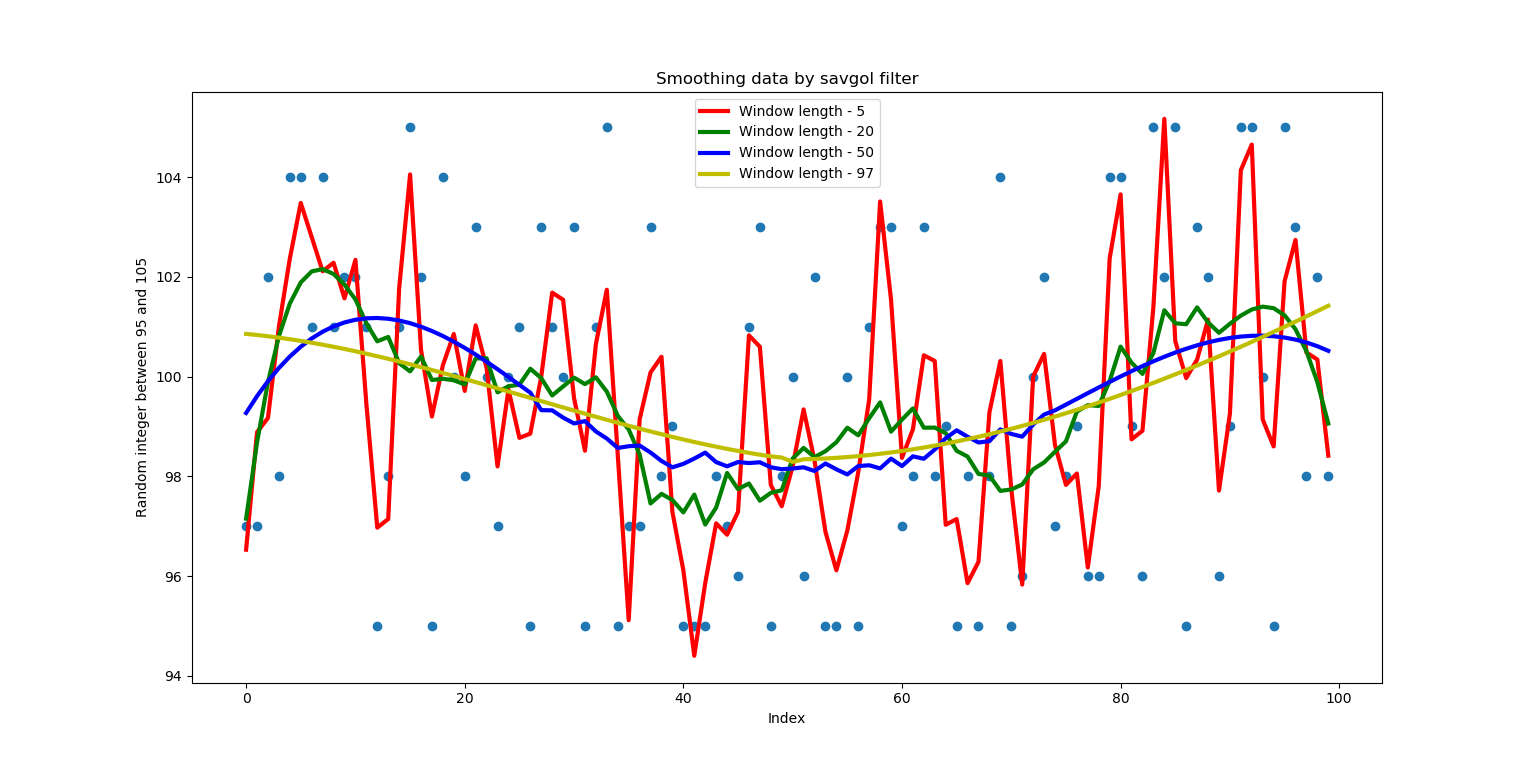
* + - Beware that the graph has a logarithmic x-axis because the values range from 1e-8 to 1
    - When changing tolerance, we can observe two straightforward trends – the increase causes the error margin to rise, and the simulation time to fall.
    - This is caused by needing less calculation when 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 the computer, the optimal spot lies at **0.0014.**
  + **Adjusting value**
    - This is with window span = 7
    - It nicely shows how lengthy the simulation is when we choose the adjusting value close to the boundaries of possible values (-1 and 0)
    - It can be also seen that adjusting value does not have a considerable effect on the error margin of the simulation, so we always come to a very similar solution, regardless of the chosen value
    - The reason is that the value close to -1 causes the solution to “converge” very slowly, as we are almost reverting to the previous points when adjusting the heat flux (we would be literally reverting to the previous points with the value equal to -1)
    - Values close to 0 have the disadvantage of sharply decreasing the absolute value of adjustment, therefore moving to the satisfiable result takes a very big number of steps
    - Practically there is very little difference in the range between (-0.7 and -0.2) in terms of time complexity
    - **Discussion:** In our test case the range of suitable adjusting value was very wide, so we cannot recommend any specific value. The mistake should not be made when choosing a value around **-0.5**, which lies around the middle of our suitable interval.
  + **Initial Heat Flux Adjustment**
    - Dt used here was 3 seconds (left), and it is apparent that higher initial adjustment speeds up the simulation, but at the cost of higher error margin
    - The second experiment with dt = 30 seconds was conducted, and results show that in this higher timestep error is practically the same regardless of the value, only the time is much higher in lower values. The reason for the time difference is the higher amount of steps needed to traverse bigger differences in heat flux when the adjustment steps are lower.



* TESTING OBSERVATIONS:
  + The lower dt, the less amount of iterations is needed
    - Meaning we probably do not need any special methods for determining the good heat flux (like bisection method), when we have a very low dt
    - The explanation can be that the difference in heat fluxes for a shorter time is lower, therefore we can reach the next value quicker
    - It is also good to decrease the init\_q\_adjustment with decreasing dt because the changes in heat flux for that shorter time period is expected to be lower
  + **Smoothing the data** helps reducing errors in some situations (best way from 85 to 65)
    - TODO: describe the process of smoothing, and its challenges
    - The moving average is turning out to be better
    - It can be advantageous to perform the smoothing multiple times
    - 2 times 2 window length yields the best results when dt is 20
    - Two possibilities of performing the smoothing:
    - **Moving average method**
      * <https://stackoverflow.com/questions/20618804/how-to-smooth-a-curve-in-the-right-way#answer-26337730>
      * Using <https://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html>
      * Python itself (through numpy library) already offers techniques on how to achieve this.
      * def smooth(y, box\_pts):

box = np.ones(box\_pts)/box\_pts

return np.convolve(y, box, mode='same')

* + - * This function is evaluating the average value of “box\_pts” points around a certain point, including that certain point, and it is assigning this value as a new value of that point
        + When the “box\_pts” is an odd number, it takes the point itself and the same amount of elements before the point and after the point (in case of it being 5, it takes that point, two previous points, and two next points)
        + In case of it being even, it takes one more point from before the point in question (in case of being it 4, it takes the point, two previous points, and one next point)
      * To try and validate the correctness of this method we generated a random sequence of numbers between 95 and 105, and used the abovementioned method to smooth this data (with the expected result of an almost straight line at the value of 100 – as it is the mean value). As can be seen on the graph above, the final result, in other words, the smoothness, is highly depending on the window span (box\_pts)
      * The basic numpy approach shown above has the disadvantage that on both ends (boundaries) of the graph the calculated averages are not corresponding to the rest because there is not enough data on the boundaries to calculate the complete average of a specified number of points around. (When it cannot find any points previous to, or after a point, it will regard that value as 0, and still includes this point to the calculation of the average – therefore the averages are so low on the boundaries.)
        + Therefore, we modified the function to also more precisely determine averages on the boundaries. It was done by multiplying the values on the boundaries with appropriate coefficients according to their position (and therefore according to the number of points in the vicinity that the previous value was calculated with). As can be seen in the picture above, the points closer to the boundary need to be multiplied by a higher coefficient. The actual logic of determining this coefficient can be reviewed in the code (or on tn.cz).
        + 
    - **Savgol filter (aka Savitzky-Golay filter)**
      * <http://scipy.github.io/devdocs/generated/scipy.signal.savgol_filter.html>
      * <https://stackoverflow.com/questions/20618804/how-to-smooth-a-curve-in-the-right-way#answer-20642478>
      * <https://scipy-cookbook.readthedocs.io/items/SavitzkyGolay.html>
      * *“The Savitzky-Golay filter removes high-frequency noise from data”*
      * Is also using np.convolve in its implementation
    - Smoothing the data is helping to get rid of random errors that happened during the simulation. It makes the graph looking smoother by eliminating the extreme results in individual points – and as these extreme results are usually oscillating around theoretically correct solution, it makes the final results more precise and helps to uncover result that is more probable in reality.
    - Obviously, the smoothing has also drawbacks, and can even worsen the result. This can happen in situations when there would really be a lot of oscillations in the real experiment, so we would be incorrectly assuming these are random errors, but in reality, they would be valid.
      * It can be also very tricky to choose the right window span, which is greatly affecting the smoothness. Usually higher window span means the smoothing will be more effective – aggregating together more points. Higher smoothing means we will more precisely highlight an ongoing trend, however, we can even smooth out some extremes in the data itself, which is undesirable.

* 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 the 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 a 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 for this reason.
* Observation: 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
  + It is possible, but the errors at the end of the simulation are going ballistic
  + With step 1 second even with theta 0.498, it is going crazy after 4000 second
* 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
  + If not possible, have it as a python module
* Convert the application to PySide2 instead of PyQt5 at the very end
* 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
* Test the impact of the whole inverse approach (changing window span, having various logic for finding satisfying heat flux, etc.) on more data sources
  + We can even randomly generate some heat flux values in time, calculate the temperature distribution through our NumericalForward, and then trying NumericalInverse for these temperature data, generating heat flux values again and comparing them with the original heat flux values
  + This way we could test all the approaches more thoroughly because so far, we can test it only on one specific simulation, which is not a very representative sample
* Reflect the last changes in architecture in the description of all software parts
* Research the behaviour of smoothing with moving average when inputting even window span (2, 4…) – it is causing probably undesirable shifting of the graph to the right side
  + Probably allow only for the input of only odd numbers, the same way as in the SavGol filter
* Formulate the main goals of the thesis
  + Software can be used for educational purposes of basic software development in engineering
    - Showing subtle examples of using software tools and methodologies like version control or unit testing, that can greatly improve the quality and reliability of the created software
  + Showing that programming such an application is nothing hard
    - Anybody with basic python knowledge can do that
  + Write and document the code in such a manner that anybody can understand it
    - Working of the software should be understandable even by those who have never written a line of code in their life
  + Create an architecture that can be easily extended
  + Create a referential implementation of connecting GUI with simulation that can be used for many other purposes outside of heat transfer (anything can be simulated there)
    - The main components of the software can be literally copy-pasted and used in a different project
  + Create a minimalistic approach for solving the inverse problem
    - Trying to keep it as simple as possible, while still performing a good-enough solution
  + Showing that python can be used instead of Matlab
  + Do it all open-source, so that everybody can benefit
  + “Github hosted collaboration of enthusiastic engineers”
* Other ideas for introduction (preface) and conclusion:
  + The gap between engineering and programming is getting smaller
  + The automation of tasks is growing at immense speed today, so a good command of computing devices (computers) is becoming almost a necessity in engineering world
  + Motivation for creating this thesis (maybe too honest and disrespectful)
    - Wanted to do something unique and exciting, and not being the hundreth person who calculates and designs the same heat turbine or boiler, where hardly anything new can be written and the whole thesis would be basically copying information and equations from some old book.
  + As the thesis was meant to resemble an open-source software project from the very beginning, the choice of English as a main language of the whole thesis was not very hard
    - When projects are created and documented in English, they immediately get much broader potential audience, and together with that goes the number of possible contributors
    - Also, to write about a software project in some other language than English is not very intuitive, as the majority of code syntax, as well as the programming terminology, is defined in English, and the translations to local languages can be quite awkward