BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

ENERGY INSTITUTE

ENERGETICKÝ ÚSTAV

SOFTWARE FOR INVERSE HEAT TRANSFER PROBLEMS

SOFTWARE PRO ŘEŠENÍ INVERZNÍCH ÚLOH PŘENOSU TEPLA

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR — Bc. Jiří Musil

AUTOR PRÁCE

SUPERVISOR — doc. Ing. Jiří Pospíšil, Ph.D.

VEDOUCÍ PRÁCE

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Energetický ústav
Student:	**Bc. Jiří Musil**
Studijní program:	Strojní inženýrství
Studijní obor:	Energetické inženýrství
Vedoucí práce:	**doc. Ing. Jiří Pospíšil, Ph.D.**
Akademický rok:	2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Software pro řešení inverzních úloh přenosu tepla

Stručná charakteristika problematiky úkolu:

Práce je zaměřena na zpracování matematické formulace inverzní úlohy, umožňující ze zaznamenaného průběhu teplot v konkrétních místech tělesa identifikovat nestacionární tepelné toky. Inverzní úloha bude zpracována ve zvoleném softwarovém nástroji a vytvořeno bude uživatelsky přívětivé GUI. Dosažené výsledky budou porovnány s daty dostupných experimentů.

Cíle diplomové práce:

1. Představení možností řešení úloh přenosu tepla.
2. Popis inverzních metod řešení úloh přenosu tepla.
3. Návrh řešiče a GUI pro řešení inverzní úlohy nestacionárního přenosu tepla na zadané geometrii.
4. Porovnání vypočtených výsledků s experimentem.

Seznam doporučené literatury:

PAVELEK, M. Termomechanika. Brno: Akademické nakladatelství CERM, 2011, 192 s., ISBN 978-8--214-4300-6.

JÍCHA, M. Přenos tepla a látky. Brno: CERM, 2001, 160 s., ISBN 80-214-2029-4.

BERGMAN, T. a Frank P. INCROPERA. Fundamentals of heat and mass transfer. 7th ed. /. Hoboken, NJ: Wiley, c2011, xxiii.

NELLIS, G. a S. A. KLEIN. Heat transfer. New York: Cambridge University Press, 2009. ISBN 0521881072.

OZISIK, M. N. a H. R. B. ORLANDE. Inverse heat transfer : fundamentals and applications. B.m.: Taylor & Francis, 2000. ISBN 9781560328384.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

doc. Ing. Jiří Pospíšil, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRACT

This thesis is focused on creating a software tool for the simulation of heat transfer, with the focus on the inverse problem. The first part describes the basic theory of inverse problems and heat transfer, as well as the derivation of a numerical solution of heat transfer equation, that is suitable for a computer simulation.
The main part of the thesis deals with the design and implementation of the software solution itself. Apart from the computational engine, that is responsible for the simulation, also the graphical user interface (GUI) is created, that enables for convenient interaction with the computational engine.
The last part serves as a presentation of achieved results and their comparison with the real experiment, as well as analyzing the influence of incoming parameters on the quality of the simulation.

Keywords:

Inverse problem, heat transfer, software development, Python

ABSTRAKT

Tato práce se zabývá vytvořením softwarového nástroje pro simulaci přenosu tepla se zaměřením na využití inverzní úlohy. Je zde popsána základní teorie inverzních úloh a přenosu tepla, na kterou navazuje odvození numerické rovnice přenosu tepla, vhodné pro počítačovou simulaci.
Hlavní část práce se věnuje návrhu a samotné implementaci softwarového řešení, s ohledem jak na funkčnost, tak na uživatelskou přívětivost. Kromě výpočtového modelu, který je zodpovědný za průběh simulace, je vytvořeno také plnohodnotné uživatelské rozhraní (GUI), umožňující jednoduchou interakci s výpočtovým modelem.
Závěrem práce je prezentování dosažených výsledků a jejich porovnání s reálným experimentem, stejně jako zjištění vlivu vstupních parametrů na kvalitu simulace.

Klíčová slova:

Inverzní úloha, přenos tepla, vývoj softwaru, Python

BIBLIOGRAPHIC CITATION

MUSIL, Jiří. *Software for inverse heat transfer problems* [online]. Brno, 2020. Also available at: https://www.vutbr.cz/studenti/zav-prace/detail/124707. Master's thesis. Brno University of Technology, Faculty of Mechanical Engineering, Energy Institute. Supervisor Jiří Pospíšil.

DECLARATION

I declare that I have personally created the master's thesis **Software for inverse heat transfer problems** under the supervision of doc. Ing. Jiří Pospíšil, Ph.D. and using resources listed in the bibliography.

---------- ----------

Date Bc. Jiří Musil

ACKNOWLEDGEMENT

I would like to express my thanks and gratitude to the supervisor of this thesis, doc. Ing. Jiří Pospíšil, Ph.D., for allowing me to choose this topic and combine the energy field with the software engineering field.

During the whole process of making the thesis, I was enjoying the collaboration and consultations with Ing. Libor Kudela. His technical and mathematical knowledge was very helpful for me, especially when creating the numerical simulation. I would like to thank him for all his time and energy dedicated to this thesis.

The thesis would not be possible without the data from a real experiment, for which I would like to thank Ing. Ladislav Šnajdárek, Ph.D.

The biggest acknowledgement, however, is pointed towards my parents and family overall, who supported me endlessly during the endless 19 years of my school studies.

Content

Introduction

Nowadays, the demand for automation and the use of software is growing in all industry segments, heat transfer and engineering in general are no exceptions. Therefore, a good command of computing devices (computers) has become almost a necessity in this world.

The gap between programming and engineering is getting smaller, and the extensive usage of software has the potential of quickening the engineering process, and also eliminating some human errors.

A lot of engineers have a good command of their engineering field but are lacking knowledge and experience in software development, therefore lacking even the confidence in programming. The goal of this thesis is to show that only a basic knowledge of programming is enough to create a useful piece of software that can be helpful when solving engineering problems.

Experience says that having bad software is almost worse than having no software. The damage caused by improperly written or untested software can be huge, as using potentially flawed or inconsistent results in real engineering projects is dangerous. One example being the numerous cases of bridges that have recently collapsed, not only in Italy.

Therefore, this thesis tries to create a reference implementation of a software solving engineering problems. This implementation should reflect the best practices of software development and utilise concepts like source control, modularity or testing, which are less-known outside of the software community. These best practices can greatly improve the quality and reliability of the created software solution, and therefore increase its usage in solving engineering challenges.

It is worth mentioning that the heat transfer we are modelling is just a representative for any engineering problem – the same core software could be used for any other simulation, be it a mechanical or physical one.

Inverse heat transfer problems were chosen as an appropriate engineering problem to simulate because from their nature, inverse problems are quite demanding on computational power they require. They are one of the proofs that advances in engineering are strongly relying on advances in computer science, as the research in inverse problems started to really grow only after computing devices were made broadly available.

As the thesis was meant to resemble an open-source software project from the very beginning, the choice of English as its main language was not very hard. When projects are created and documented in English, they get a 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.

1 Inverse problems

This chapter describes the basic concepts behind inverse problems. Information in this chapter come from literature [1] - [4].

In the majority of scientific fields, the classical (direct) problems mean the cause of some action (boundary condition) is given, and our goal is to determine the effect that this action will have on the object.

Inverse problems are being solved when we know the final results of an action, but we are searching for the (boundary) conditions that led to this result. In the context of heat transfer, it could be translated into knowing the temperature distribution in the object after heating it but being interested in the heat flux applied on the object. Temperature distribution inside the object is usually determined by measuring the temperature below the object's surface with an appropriate method.

1.1 Classical vs inverse problems

"In the direct problem the causes are given, the effect is determined; whereas in the inverse problem the effect is given, the cause (or causes) is estimated." [1]

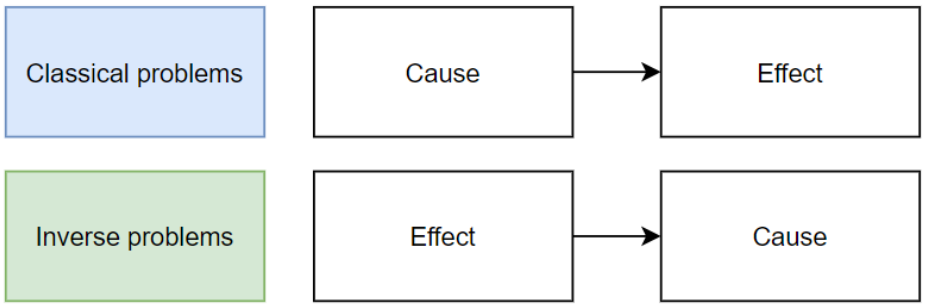

Figure 1 **Classical and inverse problems**

The difference between classical and inverse problems is also in the precision and correctness of the result. It can be said that the result of the classical problem can be unambiguously determined. The solution to the inverse problem, on the other hand, can be only estimated, and there will always be some uncertainty.

From a mathematical perspective, they differ in the aspect called posedness. Classical problems are described as "well-posed", in contrast to inverse problems, which are "ill-possed".

Well-posedness is defined as a mathematical model of a physical process that has three main characteristics – a solution exists, it is unique, and its behaviour changes continuously with the initial conditions (is stable). Examples of this include the heat equation with clearly defined initial and boundary conditions.

Ill-possed problems' main difference lies in the stability of the result – the third point. The solution of an ill-posed problem can be highly unstable and sensitive to small changes in the input data (random errors). It means that even small errors encountered in measurements can have a big impact on the result.

It is also almost impossible to prove the uniqueness of the solution resulting from the inverse problem.

There is also a terminology difference in describing the way the results are created between classical and inverse problems. In the case of classical problems, the results are *determined*, and in the case of inverse problems, the results are *estimated*. This is because while solving the inverse problem, errors in the measurements can influence the final result much more than while solving the classical problem.

1.2 Application of inverse problems

This discipline has numerous practical usages, as usually, the measurement of boundary conditions applied on an object is harder than measuring the temperature distribution inside the body. For example, placing sensors on the desired place would interfere with the measurement itself, as the sensors would influence the experiment in an undesirable way.

What is always necessary, however, is the known solution for a forward problem.

One of the most widespread usages comes from the manufacturing system. The temperature cycle for a component to gain the desired characteristics is usually well described. The boundary conditions like heat flux or pressure are to be determined, either by a trial and error or by solving an inverse problem.

The solution of heat transfer problems contributed to the space exploration programs in the 1950s and 1960s – as engineers were unable to measure the temperature at the outside of the spaceships during their re-entry from the space in the atmosphere because the heat flux there was enormously high. Therefore, temperature measurements were taken not on the very outside, but below the surface, where the temperature could be measured more easily. With this knowledge, the temperature at the surface of the spaceship could be estimated.

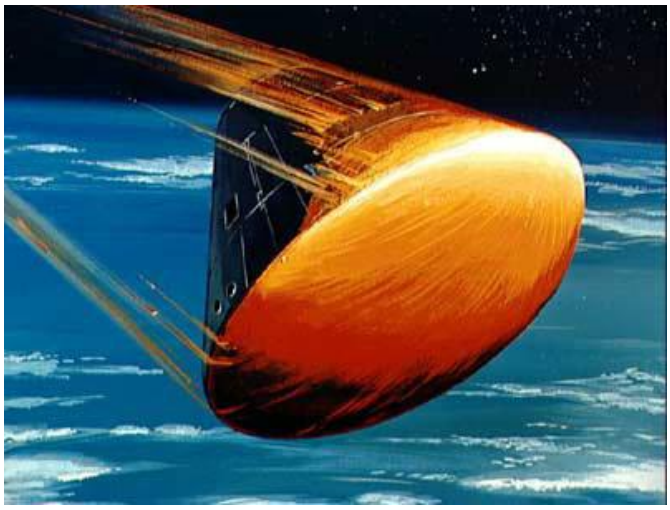

Figure 2 **Heatshield of a spaceship re-entering the atmosphere [38]**

Modern materials have their thermophysical properties dependent on the temperature and the position in the object, and therefore it is harder to determine their actual properties at

the exact moment. With the use of inverse problems, they can be estimated at almost any given point.

With the use of inverse problems, industrial devices and their properties can be more easily evaluated during real operating conditions, not having to depend only on simulation or other estimates.

"*The principal advantage of the IHTP is that it enables to conduct experiments as close to the real conditions as possible.*" [3]

Research and advancements in inverse problems are tightly connected with the advancements in computer science, as more powerful computers allowed for better usage of numerical methods and computing, which is broadly used in this area.

Inverse Heat Transfer Problems (IHTP) do not limit themselves to estimating heat flux on the boundary from the temperature distribution, they can also be used for estimating object material properties like specific heat capacity (c_p) or thermal conductivity (λ).

IHTP are usually connected with conduction, however, they are also applicable in other heat transfer mechanisms – convection and radiation, and even in mixes problems. One-dimensional problems are also the most usual ones, in recent years, however, even multi-dimensional problems are being solved inversely.

1.3 Hurdles of inverse problems

With the many benefits and newly opened possibilities that inverse problems bring and offer, there are also inherent issues with their use, that makes their usage far from being straightforward.

The temperature response for applied heat flux is delayed, and this delay is higher in points further from the source. Therefore, if we want to estimate current heat flux, we need to measure the temperatures not only in the current moment but also in the near future.

The measurement errors are also magnified more with the increasing distance from the source of heat flux. This leads to the effort of measuring the temperature (or any other value) in the closest proximity as possible. However, the closer we are, the more we are influencing the measurement results by the measurement itself – there needs to be a balance.

2 Heat transfer

This chapter is describing the very basic concepts regarding heat transfer, without trying to be an extensive resource of detailed knowledge. Information in this chapter come from the literature [5] - [8], and readers wanting to know more about heat transfer are advised to use these resources.

In our physical world energy can be transferred from one system to another either by work or by heat.

In contrast to thermodynamics, which is only concerned with the initial and final state of these systems, the field of heat transfer also determines detailed information about the way this transfer of energy is occurring, like time or rate.

"Heat transfer (or heat) is thermal energy in transit due to a spatial temperature difference. Whenever a temperature difference exists in a medium or between media, heat transfer must occur." [7]

2.1 Modes of heat transfer

There are three main types of heat transfer, so-called modes. They include conduction, convection, and radiation.

On the *Figure* below there are all three modes of heat transfer described in the context of a cooking pot on the stove.

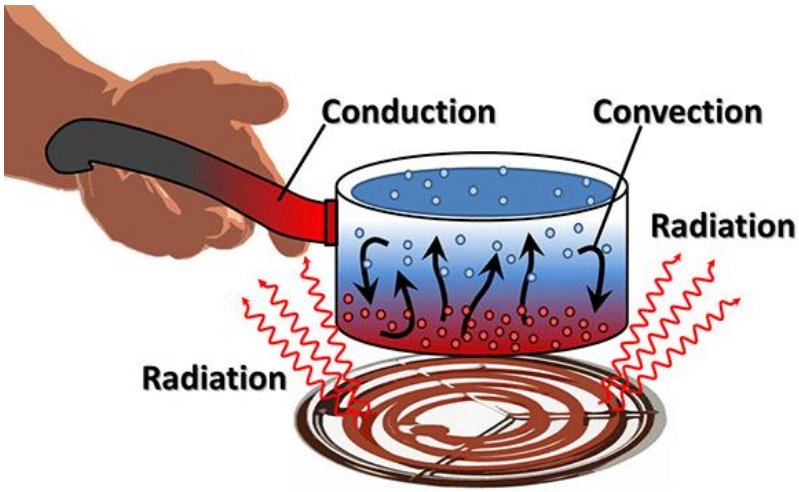

Figure 3 **Three modes of heat transfer [39]**

Conduction

Pure conduction happens inside stationary solid or fluid objects when there is a temperature gradient inside that object.

"Conduction may be viewed as the transfer of energy from the more energetic to the less energetic particles of a substance due to interactions between the particles." [7]

Particles with higher temperature (higher energy) are randomly colliding with less energetic particles. During the collisions, energy in the form of heat is transferred from the more

energetic particles to the less ones. Therefore, the heat goes from the warmer region of the object to the colder region, and this is also the direction of the heat transfer.

Random molecular motion, which is contributing to the heat transfer by conduction, is also called diffusion.

When comparing the conduction in gas, liquid and solid objects, the mechanism is almost exactly the same. However, there is a difference in the particle density (distance between particles in the object) between these three states – in case of gas, particles are much more far away from each other than in case of solid. This contributes to the fact that conduction in solid materials (metals) is much more intensive and quicker compared to gases.

The rate equation for conduction, which is used to calculate the transferred energy, is called Fourier's law.

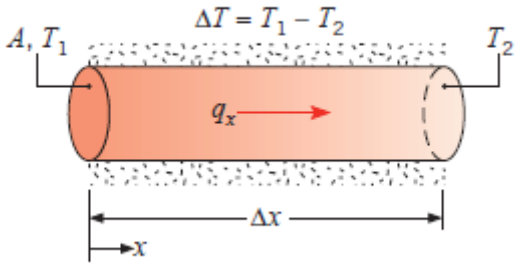

Figure 4 **Conduction in an insulated rod [7]**

Figure above shows the conduction heat transfer in a rod, whose round surface is insulated. The left side of the rod has a higher temperature than the right side of the rod. Therefore, the heat transfer is occurring from the hotter left side to the colder right side.

Convection

Convection is described as a heat transfer between a surface and moving fluid when these have different temperatures.

One mechanism of convection is already known from conduction – the heat transfers due to random movements and collisions between particles which are of different temperatures (diffusion).

The other mechanism causing convection is the bulk motion of the fluid, which is represented by a large number of molecules moving collectively. This bulk movement is also known as advection.

The final heat transfer by convection is then determined as an aggregation of these two mechanisms – because even in the large bulk of fluid, which is flowing "as one piece", there exists a random movement of molecules in the bulk.

The rate equation for convection is known as Newton's law of cooling.

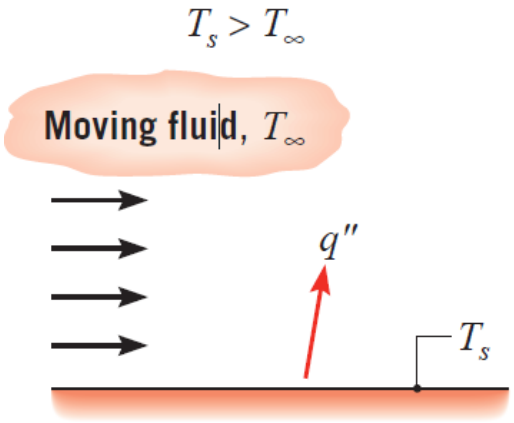

Figure 5 **Moving fluid around a hot surface [7]**

Figure above pictures the flow of fluid (air) around a hotter surface, which causes the convection heat transfer between the surface and the flowing fluid.

Radiation

Radiation means the emission of heat by all objects that have a non-zero temperature (in Kelvins).

The emission of energy by radiation happens thanks to electromagnetic waves.

Opposed to conduction and convection, where there needs to be a material medium for the heat transfer to happen, radiation does not require that. On the contrary, heat transfer by radiation happens most efficiently in a vacuum, when there is no medium at all.

The rate equation for radiation is called Stefan-Boltzmann law.

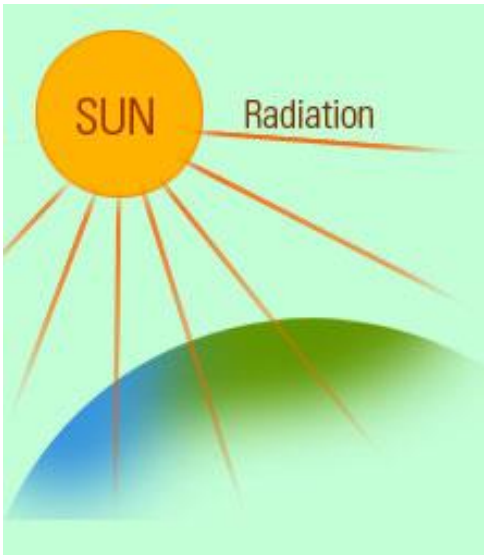

Figure 6 **Radiation from the Sun [40]**

Figure above shows the example of radiation heat transfer from the Sun to Earth – an example of radiation heat transfer occurring through the vacuum that separates Sun and Earth.

2.2 PDE and its linearization

We will be considering 1D object where variable heat flux is applied on the left side, and the right side is being exposed to the surrounding (ambient) temperature, therefore convection cooling can happen.

Figure below depicts this scenario with all the known variables, including the material characteristics and thickness of the wall (L), point of interest (x_0) where we want to get the temperature and the boundary conditions on the left and the right side.

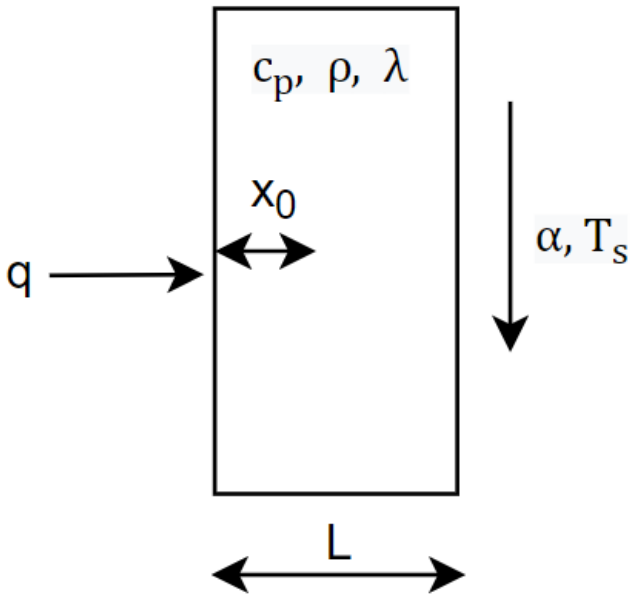

Figure 7 **Schema of an experiment wall**

To model the heat transfer efficiently using a computer, we need a numerical representation of the heat equation. This can be done by taking the PDE (partial differential equation) of the heat transfer and transforming that into the form of matrices, that computer can operate with.

After having the equations, an algorithm must be also created to actually perform the simulation, step by step.

The mathematical approach to the linearization of the PDE was inspired by [44] with the big help of [45] and [46].

2.2.1 Transforming the PDE

Following lines are describing a series of steps and equations of the transformation. Their goal is to transform PDE into a matrix form AT=b, where A is a known matrix, b is a known vector, and T is a vector of temperatures that we are interested in, and that can be calculated by solving this matrix equation.

Starting at the basic PDE of heat transfer.

$$\frac{\partial T}{\partial t} \cdot c_p \cdot \rho = \nabla \cdot (\lambda \cdot \nabla \cdot \mathrm{T}) \qquad (1)$$

Discretization in time using theta-scheme yields[1].

$$\frac{T^k - T^{k-1}}{dt} c_p \rho = \theta \cdot \nabla \cdot (\lambda \cdot \nabla \cdot T^k) + (1-\theta) \cdot \nabla (\lambda \cdot \nabla \cdot T^{k-1}) \tag{2}$$

Multiply by a test function v(x).

$$\frac{T^k - T^{k-1}}{dt} c_p \rho v = \theta \nabla (\lambda \nabla T^k) v + (1-\theta) \nabla (\lambda \nabla T^{k-1}) v \tag{3}$$

Integrate over the whole body domain.

$$\int_\Omega \frac{T^k - T^{k-1}}{dt} c_p \rho v dx = \theta \int_\Omega \nabla (\lambda \nabla T^k) v dx + (1-\theta) \int_\Omega \nabla (\lambda \nabla T^{k-1}) v dx \tag{4}$$

Apply Gauss-Ostrogradsky theorem.

$$\int_\Omega \frac{T^k - T^{k-1}}{dt} c_p \rho v dx = \theta \left[\int_{\partial\Omega} \vec{n} \mathrm{v} \lambda \nabla T^k dS - \int_\Omega \nabla \mathrm{v} \lambda \nabla T^k dx \right] + (1-\theta) \left[\int_{\partial\Omega} \vec{n} \mathrm{v} \lambda \nabla T^{k-1} dS - \int_\Omega \nabla \mathrm{v} \lambda \nabla T^{k-1} dx \right] \tag{5}$$

Weaken the form in integrals by dS.

$$\vec{n} \nabla T^k = \frac{\partial T^k}{\partial n} \tag{6}$$

$$\int_\Omega \frac{T^k - T^{k-1}}{dt} c_p \rho v dx = \theta \left[\int_{\partial\Omega} \mathrm{v} \lambda \frac{\partial T^k}{\partial n} dS - \int_\Omega \nabla \mathrm{v} \lambda \nabla T^k dx \right] + (1-\theta) \left[\int_{\partial\Omega} \mathrm{v} \lambda \frac{\partial T^{k-1}}{\partial n} dS - \int_\Omega \nabla \mathrm{v} \lambda \nabla T^{k-1} dx \right] \tag{7}$$

We have two boundary conditions (BC):

On the left side (Γ_L) we have Neumann BC – second order.

$$\frac{\partial T}{\partial n} = \frac{q}{\lambda} \tag{8}$$

On the right side (Γ_R) we have Robin BC – third order (T_s = surrounding temperature = ambient temperature).

$$\frac{\partial T}{\partial n} = -\frac{\alpha}{\lambda} (T - T_s) \tag{9}$$

[1] Theta in this case is setting the implicitness/explicitness of the solution and will be further described in later chapters.

$$\int_{\partial\Omega} \mathrm{v}\lambda \frac{\partial T^k}{\partial n} dS = \int_{\Gamma_L} \mathrm{v}q dS - \int_{\Gamma_R} \mathrm{v}\alpha(T - T_s) dS \tag{10}$$

$$\int_\Omega \frac{T^k - T^{k-1}}{dt} c_p \rho \mathrm{v} dx = \theta \left[\int_{\Gamma_L} \mathrm{v} q^k dS - \int_{\Gamma_R} \mathrm{v}\alpha(T^k - {T_s}^k) dS - \int_\Omega \nabla \mathrm{v}\lambda \nabla T^k dx \right] + (1-\theta) \left[\int_{\Gamma_L} \mathrm{v} q^{k-1} dS - \int_{\Gamma_R} \mathrm{v}\alpha(T^{k-1} - {T_s}^{k-1}) dS - \int_\Omega \nabla \mathrm{v}\lambda \nabla T^{k-1} dx \right] \tag{11}$$

Separate entries with T^k on the left and rest on the right.

$$\int_\Omega T^k c_p \rho \mathrm{v} dx + dt\theta \int_\Omega \nabla \mathrm{v}\lambda \nabla T^k dx + dt\theta \int_{\Gamma_R} \mathrm{v}\alpha T^k dS = \int_\Omega T^{k-1} c_p \rho \mathrm{v} dx - dt(1-\theta) \int_\Omega \nabla \mathrm{v}\lambda \nabla T^{k-1} dx + dt(1-\theta) \int_{\Gamma_L} \mathrm{v} q^{k-1} dS + dt\theta \int_{\Gamma_L} \mathrm{v} q^k dS - dt(1-\theta) \int_{\Gamma_R} \mathrm{v}\alpha T^{k-1} dS + dt(1-\theta) \int_{\Gamma_R} \mathrm{v}\alpha {T_s}^{k-1} dS + dt\theta \int_{\Gamma_R} \mathrm{v}\alpha {T_s}^k dS \tag{12}$$

Replacing functions (v and T) with finite elements (later omitting the notation of sums over i for simplicity).

$$v = \sum_{j=1}^{N} \widehat{\phi_j} \tag{13}$$

$$T = \sum_{i=1}^{N} \phi_i T_i \tag{14}$$

$$\int_\Omega \phi_i \widehat{\phi_j} c_p \rho dx T_i^k + dt\theta \int_\Omega \nabla \phi_i \nabla \widehat{\phi_j} \lambda dx T_i^k + dt\theta \int_{\Gamma_R} \phi_i \widehat{\phi_j} \alpha dS T_i^k = \int_\Omega \phi_i \widehat{\phi_j} c_p \rho dx T^{k-1} - dt(1-\theta) \int_\Omega \nabla \phi_i \nabla \widehat{\phi_j} \lambda dx T^{k-1} + dt(1-\theta) \int_{\Gamma_L} \widehat{\phi_j} dS q^{k-1} + dt\theta \int_{\Gamma_L} \widehat{\phi_j} dS\, q^k - dt(1-\theta) \int_{\Gamma_R} \phi_i \widehat{\phi_j} \alpha dS T^{k-1} + dt(1-\theta) \int_{\Gamma_R} \widehat{\phi_j} \alpha dS T_s^{k-1} + dt\theta \int_{\Gamma_R} \widehat{\phi_j} \alpha dS\, T_s^k \tag{15}$$

Equation (15) above will be the basis for creating an iterative (step by step) algorithmic approach of simulating the progress of heat transfer.

As all the elements are already in place, we can increase the readability by replacing equation elements with matrixes, aiming to get a matrix form AT=b. The logic of this replacement is to let everything on the left side be matrix A (as the T is already there), and everything on the right side will become vector b.

$$\int_{\Omega} \phi_i \widehat{\phi_J} c_p \rho dx = M \tag{16}$$

$$\int_{\Omega} \nabla\phi_i \widehat{\nabla\phi_J} \lambda dx = K \tag{17}$$

$$M + dt\theta K = A \tag{18}$$

$$(M - dt(1-\theta)K).T = b \tag{19}$$

$$A.T = b \tag{20}$$

It is apparent that the equation (19) defining the vector b is not complete, as we have not accounted for all the boundary elements represented by integrals over Γ_L and Γ_R domains, representing the edges, the very left and very end of the body. As these operations modifying the first and last element in the b vector must be done for each step, depending on outside conditions, they are covered in the further part, when describing the step by step algorithm.

Also, even the equation (18) is not complete, as there is one extra element on the left side of equation (15) – the integral over Γ_R. This will need to be accounted for when creating the resulting A matrix by modifying its very last element on the right side.

Constructing numerical matrixes

We see that the final matrix form AT=b relies on matrixes M and K, which need to be derived further. The goal is to replace the elements $\int_{\Omega} \phi_i \widehat{\phi_J}\, dx$ and $\int_{\Omega} \nabla\phi_i \widehat{\nabla\phi_J}\, dx$ in equations (16) and (17) with their numerical matrix representations. These matrix representations, together with other equation components – c_p and ρ in the case of (16), and λ in the case of (17) – will then form the matrixes M and K.

Numerical representations of these matrixes were determined from the first order Galerkin elements. Galerkin method is a common approach to describing the problem in the finite element method space.

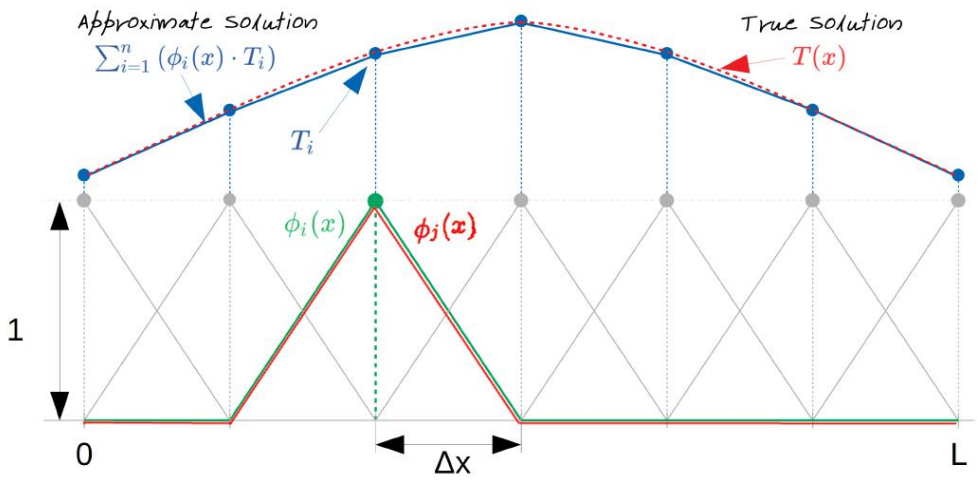

Figure 8 **Galerkin elements, situation with i = j, inspired by [45] and [46]**

The *Figure* above shows the representation of first order Galerkin elements. The main parts are the functions Φ_i and Φ_j, which are part of our matrixes M and K. These functions have a triangular shape with a value 1 at one point and value 0 in all other points. Triangular shape was chosen to make the linearization easier, but any arbitrary shape could have been chosen.

We can visually represent the integral of the product of these two functions in different indexes (i and j). It has a form of integral over the area where these two functions overlap each other[2]. The indexes i and j then represent the indexes in the resulting matrix.

There are three possibilities that can occur – either the indexes i and j are the same (as depicted above), they are neighbours (their difference is 1), or they are not touching each other (their difference is more than 1). We can, therefore, expect that the resulting matrix will be filled with three different numeric values, depending on the relationship between indexes i and j in the matrix.

All three possibilities are depicted in the *Figure* below, with the common overlap being cross-hatched in blue.

[2] Theoretically the integral spans the whole object domain, but its value is zero where these two functions do not overlap, because at least one of them has zero value.

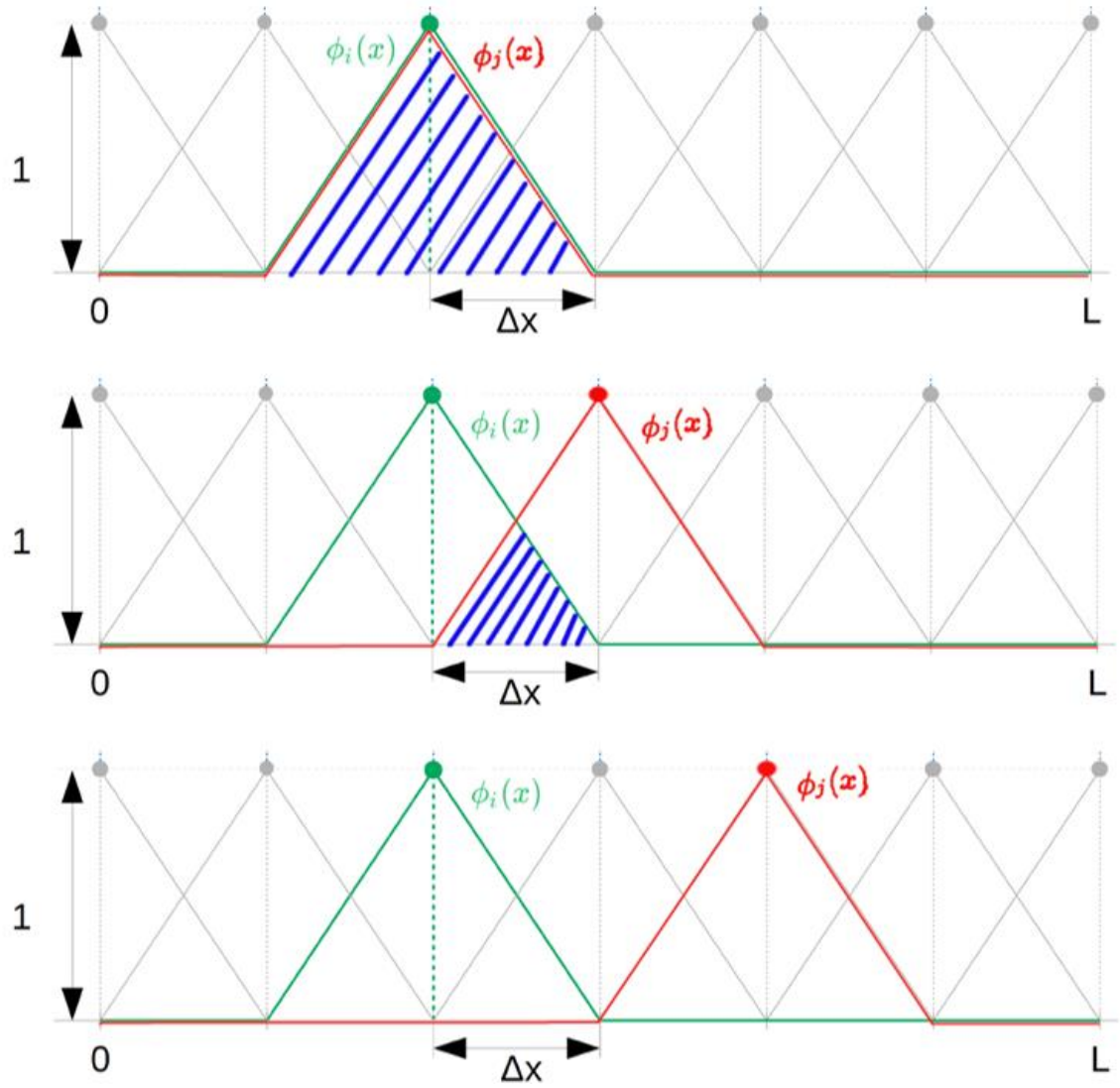

Figure 9 **Galerkin elements – possible i and j relationships, inspired by [45] and [46]**

It is apparent, that the more overlap both functions have, the bigger will be the resulting integral of their product. Numerical meaning of this is that the highest values will be found on the main diagonal, where the indexes i and j are identical (in the graph the functions lay on each other). On both sides from the main diagonal, we expect lower numbers because of the lower overlapping area. And finally, two and more spaces away from the main diagonal, the matrix will contain zeros, as the functions that are this much apart are not sharing any common overlap, and their product will always be zero.

Starting with the M matrix simplifications, when we perform the integration on the diagonal, with i and j being the same, we will get the value $(2/3) \cdot \Delta x$. Calculations with i and j shifted for one yield the value $(1/6) \cdot \Delta x$, while combinations of more shifted indexes are always zero. To unite the denominator, we will transfer the diagonal value to $(4/6)$, and we can factor the Δx from the matrix so that only numbers occur there.

There is however one more thing to do, as the situation is a little bit different on the very edges of the system (on the boundaries). The values associated with the boundary nodes

need to be cut in half, as only one half of the triangle lies inside the body domain. This causes the numerical values in the very first and very last node be divided by two (in this case 4/6 becomes 2/6). The result can be seen below in equation (21).

$$\frac{M}{c_p\rho} = \int_\Omega \phi_i \widehat{\phi_j}\, dx = \begin{pmatrix} 2/6 & 1/6 & 0 & 0 \\ 1/6 & 4/6 & 1/6 & 0 \\ 0 & 1/6 & 4/6 & 1/6 \\ 0 & 0 & 1/6 & 2/6 \end{pmatrix} \Delta x \tag{21}$$

In the case of the K matrix, the functions are derived before being multiplied and integrated, therefore different numbers will occur in the matrix. The diagonal values will result in $2/\Delta x$, and the ones shifted by one will become $-1/\Delta x$. Same as above, we need to apply special treatment to the boundary nodes and divide their values by two. The final result is described below in equation (22).

$$\frac{K}{\lambda} = \int_\Omega \nabla\phi_i \nabla\widehat{\phi_j}\, dx = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix} \frac{1}{\Delta x} \tag{22}$$

The final matrixes can be then seen below, together with their computer representation, describing how do the matrixes look in the code. The variable N is determining the number of elements that we cut the object into, therefore influencing the matrix size. It should be also pointed out that index 0 represents the first element in the array, and index -1 represents the last element – this is connected with the concrete implementation in Python.

Matrix M

Matrix M can be described as a tridiagonal sparse mass matrix. It contains information about the heat capacity of the elements and how their temperatures react to incoming heat.

- $M = \begin{pmatrix} 2/6 & 1/6 & 0 & 0 \\ 1/6 & 4/6 & 1/6 & 0 \\ 0 & 1/6 & 4/6 & 1/6 \\ 0 & 0 & 1/6 & 2/6 \end{pmatrix} c_p\rho\Delta x$
- M = csr_matrix(dx*rho*cp*diags([1/6, 4/6, 1/6], [-1, 0, 1], shape=(N+1, N+1)))
- M[0, 0] /= 2
- M[-1, -1] /= 2

Matrix K

Matrix K is being described as a tridiagonal sparse stiffness matrix or conduction matrix. It contains information about heat conductivity and how the elements affect each other.

- $K = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix} \frac{\lambda}{\Delta x}$
- K = csr_matrix((1/dx)*lmbd*diags([-1, 2, -1], [-1, 0, 1], shape=(N+1, N+1)))
- K[0, 0] /= 2
- K[-1, -1] /= 2

Matrix A

Matrix A, which is a combination of both M and K, is also a tridiagonal sparse matrix. Its very last element must be reflecting the convection heat transfer to the surroundings – caused by the third element on the left side of equation (15). Matrix A will be then constant for the whole simulation.

- A = M + dt*theta*K
- A[-1, -1] += dt*theta*robin_alpha

Vector b

Vector b is a vector of known coefficients. It has an implied unit of J. With performance in mind, its determination was divided into two steps, in which the first one will be done only once before the simulation starts (b_base matrix), as its value will not change during the simulation. The actual vector b will be determined for each step of the calculation separately, as it depends on the current vector of temperatures (T).

This vector will also need to be further modified, as we already mentioned – in each step, there will be a different situation on the object boundaries. Therefore, the first and last element of this vector will be different each step. The exact modifications will be shown in the numerical part.

- b_base =M - dt*(1-theta)*K
- b =b_base.dot(T)

2.2.2 Numerical simulation using matrixes

The section above shows how do the matrixes look before the start of the simulation, with the most important equation governing everything else being (15). This is however not enough, as we need some logic (algorithm) of actually running the simulation in time, where all these matrixes will be interacting together and will create resulting temperature distribution in the body.

Our goal is to simulate temperature in time and focusing only on one specific place in the body – we call this place x0. Therefore, we need an array in which we will be storing the gradual progress of that one temperature in time. The length of this array is dependent on the number of time steps, in which we will be evaluating the simulation. We call this array **T_x0**, as the x0 index represents that this temperature is connected with a specific place in the body.

However, even if we are only concerned about one specific place, we cannot just determine the temperature at this place, we need to simulate the temperature distribution in the whole body because all the elements are influencing each other. Only from this complete temperature distribution can we focus on the place of our interest. This temperature distribution in the whole has also a form of an array and we call it **T**. Its size is influenced by the number of elements in the body. Its content will be changing each step when we will use its previous value (previous temperature distribution) to simulate a new value. There is no need to store historic values, apart from the one temperature at x0 place (which is done in T_x0 array).

The abovementioned array (vector) T, the temperature distribution, is the same T as we were setting up in the AT=b matrix form. T can be set as unknown in this equation, and considering we know the current matrix A and current vector b, we can determine vector T, the current temperature distribution.

The prerequisite of determining T is, therefore, knowing the exact form of A and b in the current time. We already know that matrix A remains constant during the whole simulation and that the vector b needs to be recalculated for each and every step.

Finally, we have a chance of completing the equation (19) and defining all the operations that need to be performed on the vector b in every step of the simulation. All these operations are nothing but modifications of the right side from equation (15), where we take all the integrals over Γ_L and Γ_R and apply it numerically on the first element (left side) of the vector and on the last element (right side) of the vector respectively.

Below we can see the new instantiation of a vector b and all five modifications that need to happen at each step because all those operations are depending on current values of temperature distribution (T) and the index of the current step (current_step_idx). Modifications are also connected with their counterparts from equation (15) to make their origin clear.

- b = b_base.dot(T)
- b[0] += dt*(1-theta)*HeatFlux[current_step_idx-1]
  - $+dt(1-\theta)\int_{\Gamma_L}\widehat{\phi_j}dSq^{k-1}$
- b[0] += dt*theta*HeatFlux[current_step_idx]
  - $+dt\theta\int_{\Gamma_L}\widehat{\phi_j}dS\,q^{k}$
- b[-1] -= dt*(1-theta)*robin_alpha*T[-1]
  - $-dt(1-\theta)\int_{\Gamma_R}\phi_i\widehat{\phi_j}\alpha dST^{k-1}$
- b[-1] += dt*(1-theta)*robin_alpha*T_amb[current_step_idx-1]
  - $+dt(1-\theta)\int_{\Gamma_R}\widehat{\phi_j}\alpha dST_s^{k-1}+dt\theta$
- b[-1] += dt*theta*robin_alpha*T_amb[current_step_idx]
  - $+dt\theta\int_{\Gamma_R}\widehat{\phi_j}\alpha dS\,T_s^{k}$

In the code above, we can see two new arrays, HeatFlux and T_amb, which are storing time values of the heat flux applied to the body and also the ambient temperature in the room. These data usually come in the form of measurements from the experiment.

Figure below describes the simplified logic flow of performing the whole numerical simulation and sums up all the information above.

```
prepare_all_variables()
while current_step < final_step:
    increment_step()
    assemble_and_modify_vector_b()
    solve_the_matrix_equation_and_get_temperature_distribution()
    save_temperature_at_x0()
```

Figure 10 **Pseudocode for the numerical simulation**

Conclusion of the numerical solution

In layman's terms, the whole numerical solution using the matrix equation AT=b could be described as following:

- Matrix A stores material properties of the object.
- Vector b reflects the current situation according to previous temperature and the boundary conditions in the current moment.
- Vector T when solved, will reflect the temperature distribution in the current moment.
- The ability to perform the simulation is conditioned by having time values of heat flux and ambient temperatures at the moment we want to simulate.

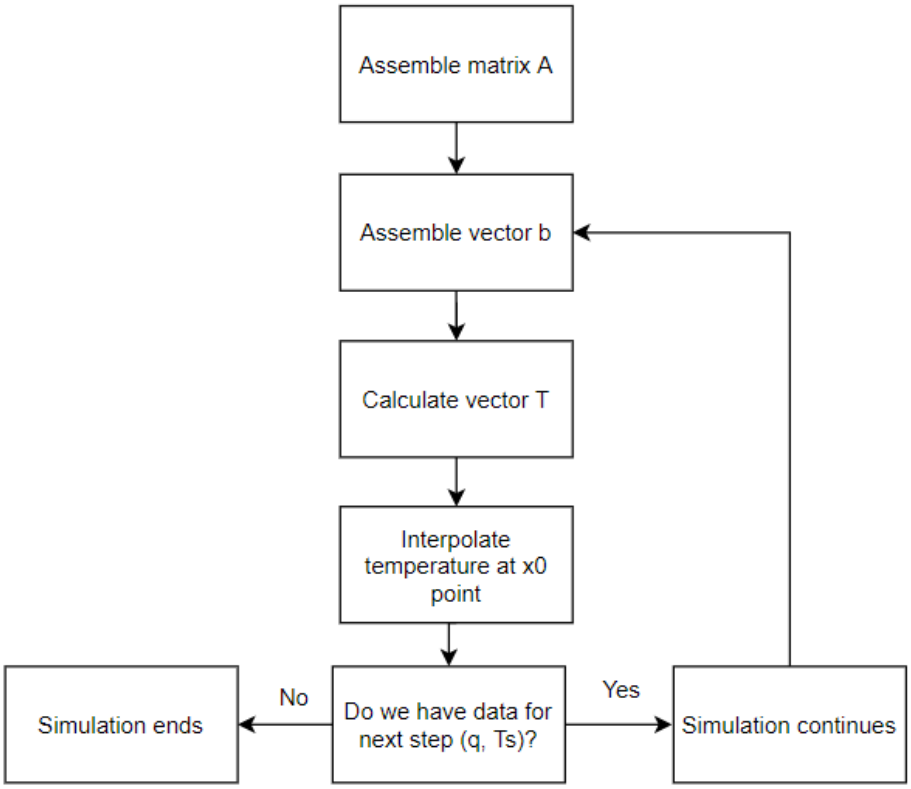

Figure 11 **Steps of the numerical solution**

3 Description of the real experiment

Data, that were used extensively when developing and testing the computational engine, come from the real experiment, that was conducted as a part of the doctoral thesis [35]. The resulted data is describing 1D heat transfer in a wall.

3.1 Schema of the experiment

The experiment consisted of a stator and rotor, and a heat transmission medium flowing between them. Thermocouples mounted into the stator and rotor were responsible for measuring the temperatures and heat fluxes caused by the flowing medium. *Figure* below depicts the main parts of the experiment.

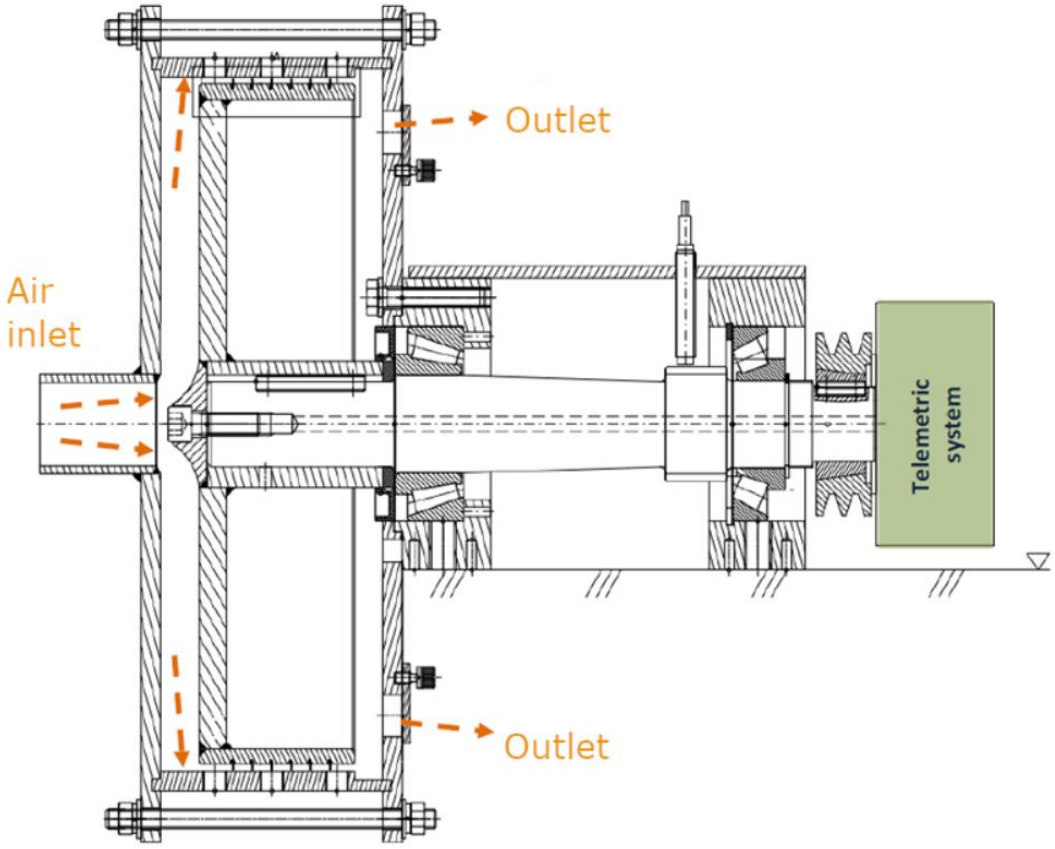

Figure 12 **Schema of the experimental stand, taken from [35]**

Rotor has a diameter of 350 mm and is powered by an electromotor. The shaft of the rotor is made hollow so that the cabling from thermocouples could be transmitted to the telemetry system, where the data is processed.

The working medium, in this case, is the compressed air, that is heated up by electric heating.

In the *Figure* below, there are three places in the stator (1, 2, 3), where thermocouples were inserted. Our data come from position 1. Probably more precise results could be gained using the data from position 2, as the heat transfer in this place is more resembling of the

real 1D heat transfer that we want to picture, with as little 2D conduction influence as possible.

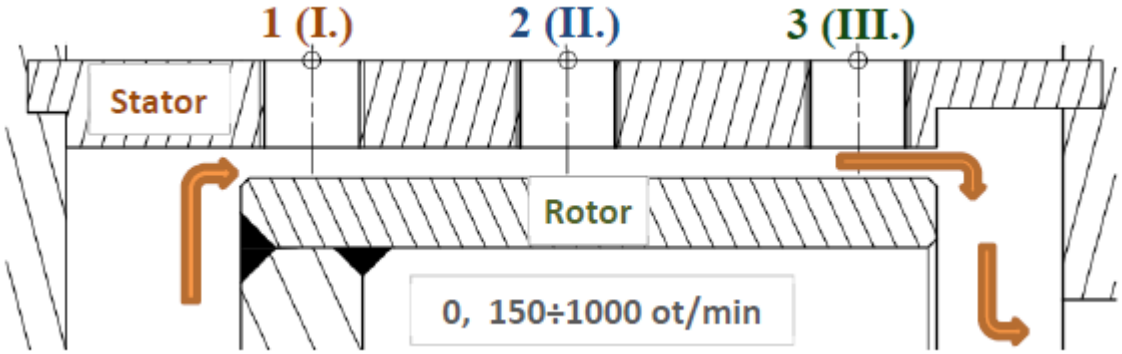

Figure 13 **Cross-section showing the positioning of thermocouples, taken from [35]**

We got the following details from the experiment, that were used as boundary conditions for our computation engine, together with all the time-dependant data:

- The material of the stator was stainless steel, with the density ρ = 7850 kg/m^3, specific heat capacity c = 490 J/kgK and heat conductivity λ = 13,5 W/mK.
- The convective heat transfer coefficient of the surrounding air was determined to be α = 13,5 W/m^{2}K.
- The width of the stator was 1 cm, and the temperature was measured 0,445 cm deep in the stator from the flowing-air side.

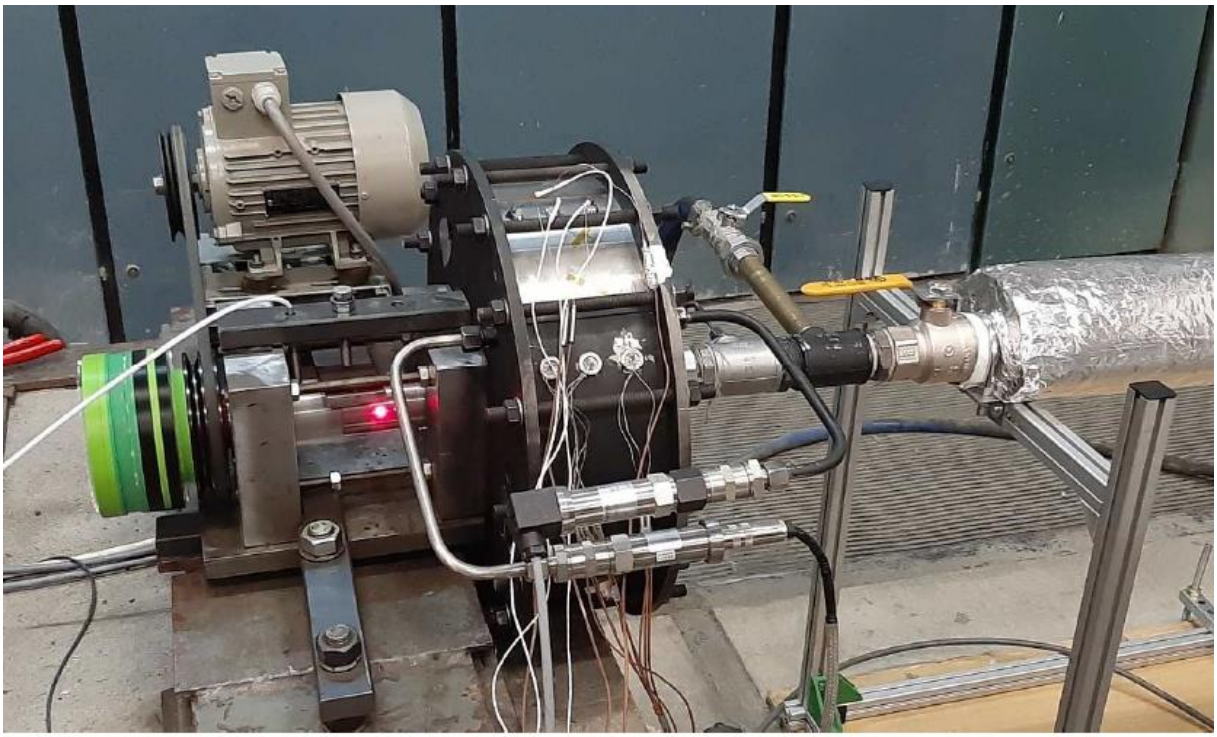

Figure 14 **View at the real experimental stand, taken from [35]**

3.2 Data from the experiment

The main benefits of the experiment for our thesis were the time-flow data of measured temperatures and corresponding heat fluxes.

We were using these data extensively when developing the computation engine. These data served the purpose of validating the correctness of the simulations – in both the classical heat transfer simulations and the inverse simulations.

It also played a crucial role in the testing phase, when we were determining the impact of various parameters on the final result – comparing their error (deviation) from these experimental results gave us the measure of quality for the result.

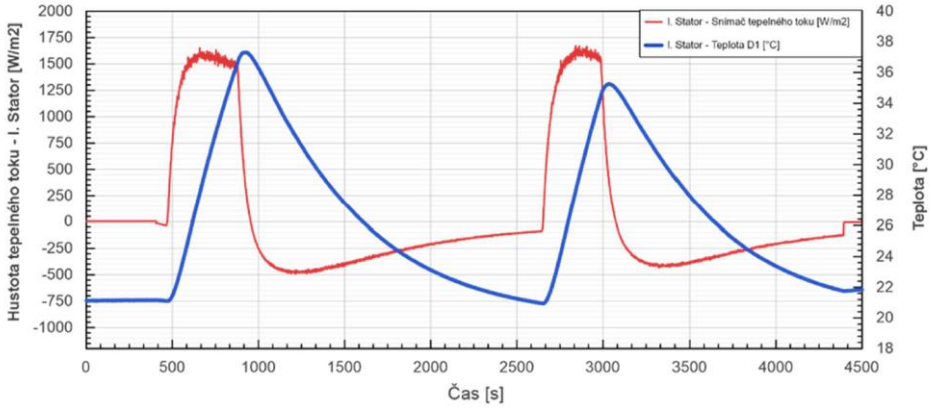

Figure 15 **Experimental data, taken from [35]**

Figure above shows the time progress of both the temperature and heat flux that was measured during the experiment. *Figures* below show the same data in our application.

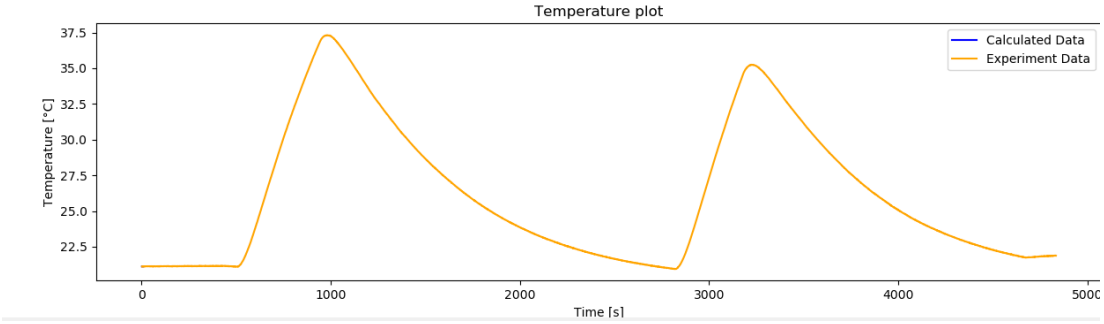

Figure 16 **Temperature data rendered in our app**

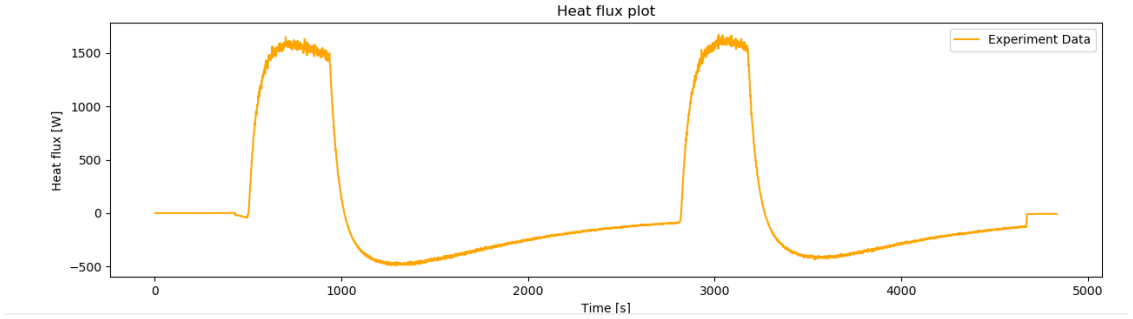

Figure 17 **Heat flux data rendered in our app**

4 Software requirements and analysis

This chapter is dedicated to gathering the requirements for the software, laying out the architecture and choosing the best tool (programming language) for the project.

4.1 Architecture of the solution

As in other fields, software project should start with some thorough planning and considerations about the architecture. To design a good architecture, we must first gather all the requirements, meaning what should be achieved and what features are necessary. Having this knowledge at the very start really helps with architectural decisions.

These are the basic requirements for the software, what it should do:

- take data from the user
- run the simulation
- show results to the user

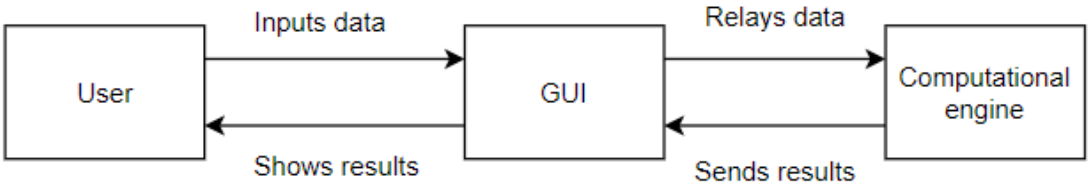

Figure 18 **Basic data flow diagram**

The *Figure* above shows the most basic diagram, describing the high-level flow of information. It starts with the user inputting data into the GUI, these data are passed on to the computational engine, which creates the simulation. Simulation data are then passed back to the GUI, where they are processed, and the results are displayed to the user.

After defining this high-level diagram, we will be further defining the responsibilities of the main components – the GUI and the computational engine. And as we will see, there is also a need for a third component, that connects these two main components together in an efficient manner.

GUI

Its main purpose is to be a middleman between the user and the computational engine. It needs to gather all necessary input data from the user, send all this data to the engine, and after it receives back the data about resulted simulation, it must show these results correctly.

It can be divided into the following parts:

User input service

One part of the GUI needs to be responsible for gathering all the necessary information from the user. This user input service will define, what data can be inputted in the GUI, specify their data type or specific restrictions.

Material service

Users need to be able to choose a custom material for the simulation or even create their own material. These responsibilities will be handled by a material service.

Plotting service
After the simulation finishes, we want to display the results in a graph. The plotting service prepares and configures these graphs which will be embedded in the GUI.

Computational engine
The main purpose of the engine is to take all necessary user input and perform a numerical simulation according to these input data. After the simulation is performed, it should return the results.

It will consist of these parts:

Simulation
The main part of the engine will be responsible for storing all the data and functions necessary for the simulation to be run. It will be further divided between the simulation object, implementing the specific logic for the simulation, and the simulation controller, which oversees the simulation and calls all its functions in the right order.

Experimental data service
As experimental data supplied by the user are a crucial part of the software, we will dedicate its own service for this purpose. It will be making sure the data from the experiment are well interpreted and parsed.

Multithreaded infrastructure
As we want both the main parts, the GUI and the computational engine, being connected, but not being dependent on each other, we want to dedicate a single thread to each of those components. This way, the components are not so tightly coupled, and both can be active simultaneously, which would not be possible when they would be sharing the same thread.

Therefore, the multithreaded infrastructure is the above-mentioned third component, that glues together the GUI and the engine. Its responsibility is to wait until the GUI will want to call the computational engine, and in this case, it creates a new thread and runs the engine inside it.

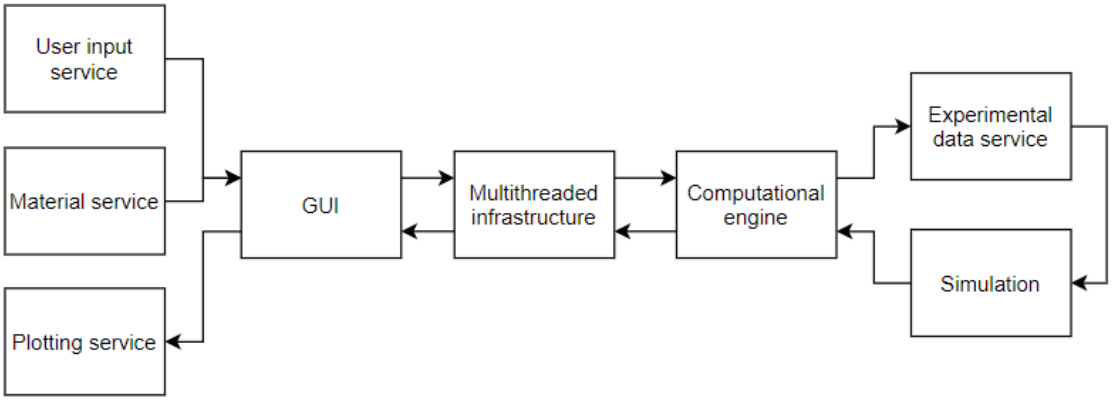

Figure 19 **Diagram of all software components**

The *Figure* above shows the final architecture diagram and the flow of data between these components, starting from left to right.

4.2 Choice of the programming language

From the planning done in the previous chapter, we know that our programming language should support three main areas (programming paradigms):

- Scientific computation
- GUI creation
- General control flow

After some consideration, Python[3] was chosen as the language for the project. Apart from Python, there were other two candidates, namely C++ and Matlab.

4.2.1 Python

Figure 20 **The official logo of Python [41]**

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 the 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, which leads to a deeper understanding of the subject.

GUI creation

The second area which needs to be satisfied is the ability to create GUI in the language. Python offers multiple libraries for creating GUIs, the basic one called Tkinter is even

[3] https://python.org/

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

The last requirement for our programming language used is good 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.

There are certainly possible alternatives to Python, which all have their own advantages and disadvantages.

4.2.2 C++

One of the other suitable languages for this application would be **C++**, lower-level language, that is known for its high speed, but also its complex code. [9]

Figure 21 **The official logo of C++ [42]**

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 an 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, etc. – 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.).

4.2.3 Matlab/Octave

Matlab, and its open-source version Octave, are computation software for scientists and engineers.

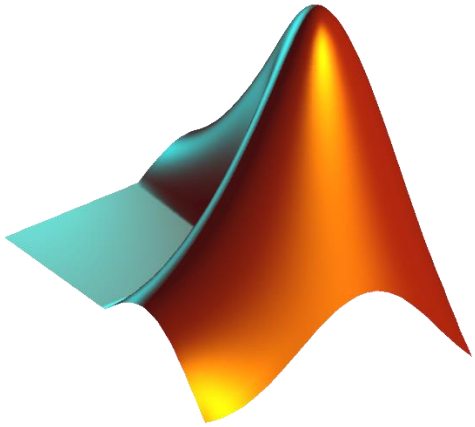

Figure 22 **The official logo of Matlab [43]**

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.

Information about the advantages and disadvantages of Matlab found in the literature [10], [11].

5 Software creation and description

This chapter is focused on the creation of the software. It is structured according to the proposed architecture, and it describes all the parts of the overall solution. The main parts of the software are the GUI, multithreaded infrastructure, and computational engine.

It also includes mentions about performance testing and optimizing, dependencies being used, and the process of data smoothing.

5.1 GUI

The main purpose of the GUI is to be a middleman between the user and the computational engine. It gathers all the data supplied by the user and relays them to the engine. After the engine finishes with the simulation, GUI should display the results.

5.1.1 Choice of GUI framework

As already mentioned, Python provides multiple frameworks for creating GUI applications, where the most prominent two are Tkinter and PyQt5. The first prototype of the application was created in Tkinter, but as the application grew on features and size, it turned out almost necessary to rewrite it using PyQt5.

Tkinter

Tkinter is a basic library for building Python GUIs, as it already comes packaged in a standard library, therefore there is no need for its installation. It is very easy to use but is not very suitable for bigger applications. [12]

PyQt5

PyQt5 is a more complete framework and offers multiple benefits over Tkinter. These benefits include areas like performance, functionality or the ease of UI development.

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.

It has a richer library of available widgets and behaviours that can be easily implemented.

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. [13]

Qt as a platform for creating GUIs is not used only in Python but in a variety of widely used languages and platforms[4]. Both the knowledge of it and the possibility of transforming the app into a different language, if necessary, can prove to be very useful.

[4] https://en.wikipedia.org/wiki/Qt_(software)

PySide2

There exists another Python GUI framework, PySide2, which offers almost exactly the same functionality as PyQt5. The only major difference is its licensing conditions, as software using PyQt5 must contribute some money to its maintainers if being monetized, not being the case by PySide2.

As we can read in the literature [14], "*PyQt5 is available under a GPL or commercial license, and PySide2 under an LGPL license"*.

Therefore, theoretically, after completing the project, the transition from PyQt5 to PySide2 can be beneficial.

5.1.2 GUI creation

GUI features:

- Information panel for the user on the top
- Custom material choice window for user-defined materials
- 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

GUI layout

The basic structure of the GUI is defined by **heat_transfer_gui.ui** file. This file contains information about the UI, its basic layout and elements. It can then be transformed into a Python file (**heat_transfer_gui_window.py**) 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.

UI file is created with the use of the QtDesigner application. It offers user-friendly creation of GUIs, which does not require almost any programming skills.

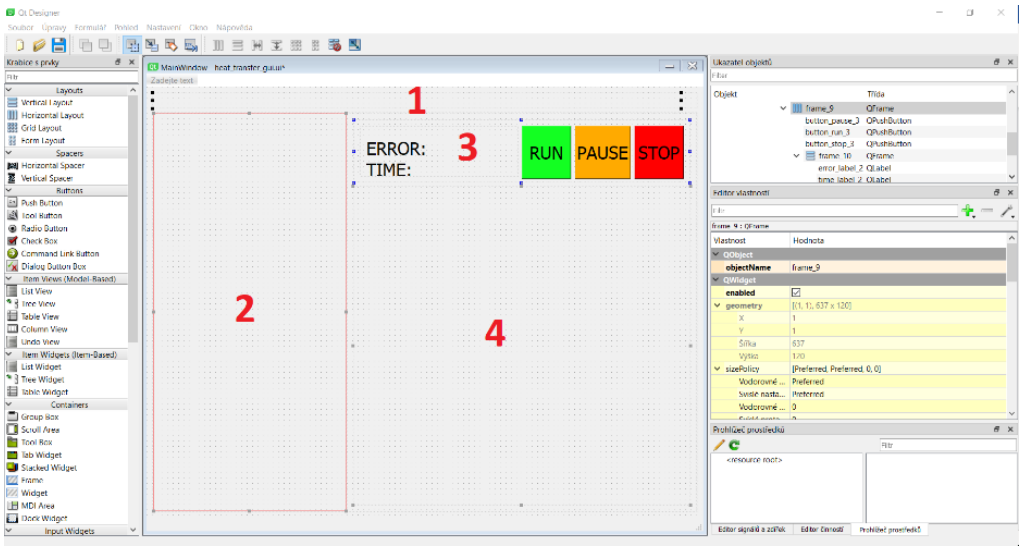

Figure 23 **Working window in QtDesigner showing the basic layout of the app**

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. Numbers at the beginning of the paragraphs below are describing the same numbered areas in the *Figure* above.

(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 of 120 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.

Filling the GUI with data

In **heat_transfer_gui.py**, the main and callable module for the GUI, 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

Left-side user input is being filled by multiple functions responsible for rendering the appropriate information:

- **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_data_file_choices()** is including the functionality for users to choose the custom file with experimental data, so they are not dependent on naming the file DATA.csv and placing it in the same directory as the main program. It enables users to choose a file anywhere in their file system, is storing its whole path, and displays just the name of the file, to confirm the user we are using a particular file.
- **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. Also, a custom material choice was included, so that users can define their own materials with custom properties. A new dialogue is created for this purpose, which is collecting the user input, and sending it to be processed by the application.
- **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.
- **add_smoothing_options()** is responsible for rendering the smoothing options after the end of the inverse simulation.
- As there is a difference in input parameters for classical and inverse simulation, we have to dynamically respond to the current algorithm choice and reflect the situation in showing the appropriate user inputs. When a change in algorithm is determined, we delete the whole layout with user inputs and replace it with newly rendered layout with updated information.

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.

All the components have the smallest possible sizes, so even in the smallest possible window size, everything will be proportionate and visible.

5.1.3 User input service

User input service is responsible for gathering all the necessary information from the user.

It provides a general infrastructure for including new user inputs. Code itself does not need to be changed to add a new input field into the GUI. 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**).

```
{
    "name": "Object length",
    "description": "The length of the 1D object we are simulating.",
    "input_name": "object_length_input_centimetres",
    "variable_name": "object_length",
    "default_value": 1,
    "parse_function": float,
    "multiplicate_to_SI": 0.01,
    "unit": "centimeters",
    "unit_abbrev": "cm"
}
```

Figure 24 **Example record of defining an input field**

Figure above shows an example defining the input field for the object length. This record will be rendered into the GUI as an input row, and the results can be seen in the *Figure* below.

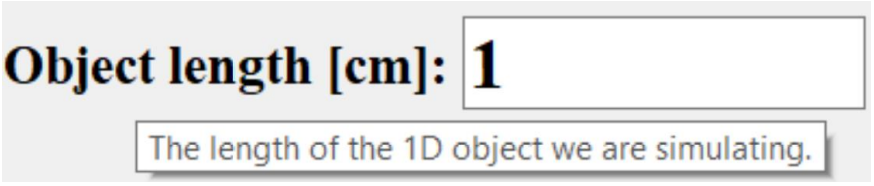

Figure 25 **Rendered user input record in the GUI**

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 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.

It offers an easy possibility of declaring default values so that the user does not have to input everything from scratch before every simulation.

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. Furthermore, the validation does not even permit inputting nothing else than numbers.

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.

5.1.4 Material service

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. We found one material library[5], but it only includes a very limited choice of materials, which is not suitable for us.

The material database exists in the form of a CSV file with material properties, **metals_properties.csv**.

[5] https://pypi.org/project/materials/

```
NAME,T_MELT [C],RHO [kg/m3],C_P [J/(kg*K)],LAMBDA [W/(m*K)]
Lithium,453,534,3582,84
Beryllium,1560,1850,1825,200
Sodium,370,968,1228,142
Magnesium,923,1738,1022,156
Aluminium,933,2700,897,237
Potassium,336,862,757,102
Calcium,1115,1550,647,201
Scandium,1814,2985,568,15
```

Figure 26 **Sample data from the material database**

It was generated by **fetch_elements.py** script, which is taking material data from Wikipedia. Material data can be refreshed anytime just by running this script.

The script is visiting the Wikipedia page of all pure metal elements from the periodic table and takes the properties we need for the simulations - rho, cp, and lambda.

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[6].

Apart from offering a database of possible materials, there is a functionality for creating new custom materials by users, which are saved in a special location and are not depending on the default materials.

5.1.5 Plotting service

Plotting service has the goal of preparing and configuring the graphs for displaying the time progress of temperature and heat flux. We are taking big advantage of the matplotlib library, which will be described later in the software dependencies.

Both of these plots are defined in their specific files, **heat_transfer_plot_temperature.py,** and **heat_transfer_plot_heatflux.py** respectively, to increase the possible customization of both plots. They are created as individual classes, that are then imported and instantiated by the GUI.

Graphs have the **plot()** method that takes both the experimental and calculated data and displays them with the appropriate description on the graph. This method is then being called by the computational engine that will be sending its numerical results here to be displayed.

Both graphs also have a **save_results_to_png_file()** method, which when called will save the current situation in the graph into a PNG file. It is being called by the GUI when users choose to save the plots.

[6] Programmatically getting content from a website - https://en.wikipedia.org/wiki/Web_scraping

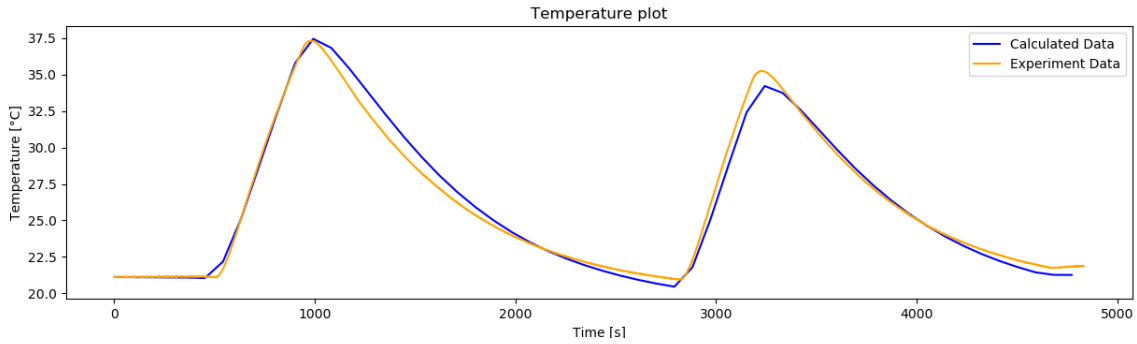

Figure 27 **Customized graph to display the temperatures in time**

5.2 Multithreaded infrastructure

This component is in place to be a connector between the GUI and the computational engine.

Specifics of GUI development

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.

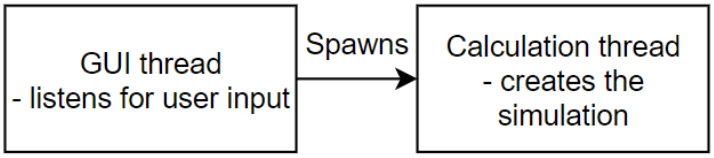

Figure 28 **Schema of multithreaded GUI application**

Tkinter situation

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. [15]

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.

PyQt5 situation

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. [16]

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 the 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[7], which we took advantage of. The queue is a data structure reminding a list, that allows for the communication between two software components (like threads). Both components can insert information on the back of the queue (put() method), or retrieve the information from the beginning of the queue (get() method).

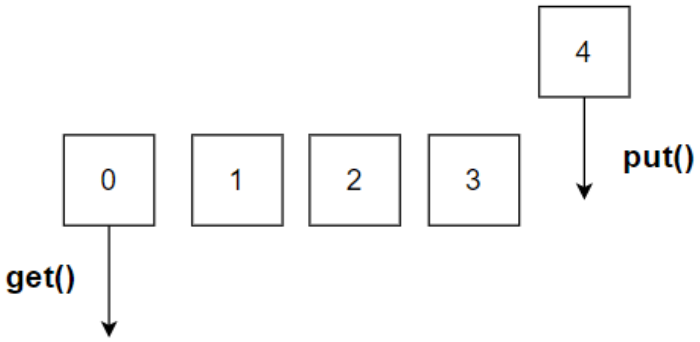

Figure 29 **Schema of the queue and its methods**

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.

[7] https://docs.python.org/3/library/queue.html

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.

5.3 Computational engine

The computational engine is responsible for defining simulation logic and running the simulation. It consists of two services (handling experimental data and interpolation) and the simulation itself.

5.3.1 Experimental data service

Its purpose is to analyse and process the experimental data supplied from the user in the form of CSV file.

The file 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]

We have a standalone module for this purpose, **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. This transformed data is then used by the simulation.

```
Time,Temperature,HeatFlux,T_amb
0,21.1339,1.3009,20.732
3.189,21.1143,1.1495,20.7312
4.239,21.1229,1.154,20.7314
5.291,21.1117,1.2747,20.729
6.342,21.1256,1.2633,20.7295
7.397,21.1189,1.2405,20.7289
8.449,21.1155,1.1358,20.7284
9.505,21.1296,1.2815,20.7281
10.558,21.1142,1.1426,20.7278
11.614,21.1134,1.0994,20.7271
12.666,21.1275,1.2337,20.7265
13.72,21.1126,1.2451,20.727
```

Figure 30 **Example of CSV experimental data**

We had multiple possibilities of how to parse the data from the CSV file, all of them having some advantages and disadvantages. At first, we were recognizing the columns according to their position in the row (time comes first, temperature second, etc.). This, however, had the disadvantage of strictly setting the structure of CSV file (the order of the columns), which can become problematic when users do not have the influence over their format.

With the assumption, that it is much easier to just include one header row into a CSV file than changing the order of the columns in the whole file, we decided to identify the data according to the header row, where users specify which column contains which data.

The header values identifying the specific columns can be seen in the *Figure* above, being Time for time, Temperature for temperature, HeatFlux for heat flux and T_amb for the ambient temperature.

5.3.2 Interpolation service

During the development, we discovered a need for another service, dealing with a quick interpolation. In the whole simulation, we were using exactly the same kind of interpolation in determining the temperature at an exact spot, given the array of temperatures in the whole body.

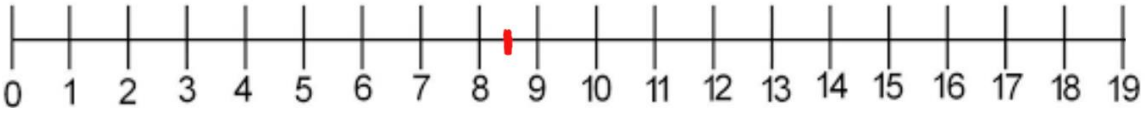

Figure 31 **Array of elements in the body, point of interest being red**

Figure above shows the example of this situation. We have divided the body into 20 elements (indexed from 0 to 19), and we have simulated temperatures in these 20 points. Our place of interest, in which we want to know the temperature, is highlighted by a red marker, situated between indexes 8 and 9. We only know the temperature at index 8 and temperature at index 9, so the value inside this interval comes from linear interpolation.

Because the position of interest stays the same during the whole simulation, we can save a lot of computing by not determining the relative position of our point in the array on every step – we do it just once in the beginning. Whenever we want to get a temperature at this point later, we just take the interpolated value at the already calculated spot, "*somewhere between indexes 8 and 9*".

This solution turned out to be much quicker than the initial interpolation, and credit for it goes to Mr Kudela. The functionality was included in a separate module, **interpolations.py**. One benefit of the solution is that it can also be used for multiple points of interest, not only for one.

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 (temperatures), 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, and we were deciding which one to use in the main code of our computation engine.

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 main code, so we moved that deciding logic to the interpolation module itself, using the factory design pattern (described later).

5.3.3 Simulation

The whole simulation will be divided into multiple parts.

The first one being the simulation controller, responsible for defining the common interface for all underlying simulations and creating the infrastructure for the simulation to be run.

After that, we have the simulations themselves, in our case the simulation for the classical approach, and then for the inverse approach.

The classical (or forward) approach in this situation means *determining* the temperature distribution inside the object knowing the boundary heat flux – according to the definition of the classical problem. Inverse approach, on the other hand, deals with *estimating* the heat flux on the boundaries knowing the temperature distribution inside the object. And as was already mentioned, to use the inverse approach, we need a solution of the classical (forward) problem first. Therefore, the inverse simulation will be depending on, and extending, the classical simulation.

The relationship between these components is described in the *Figure* below. Simulation controller sits above the individual simulations and controls them. After we create the classical simulation, which solves the underlying forward problem, the inverse simulation will use its logic while creating extra logic – effectively extending the classic simulation.

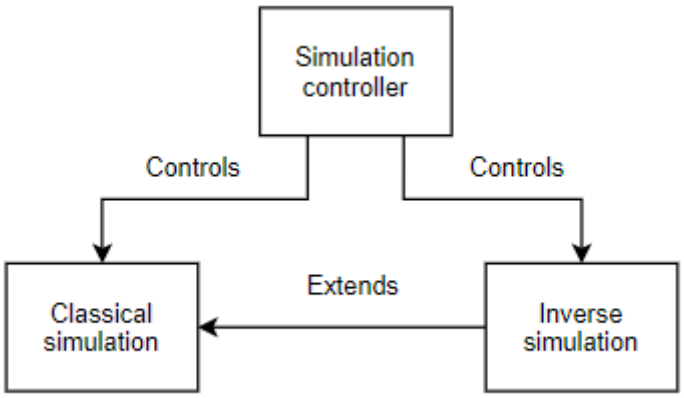

Figure 32 **Relationships between Simulation components**

Simulation controller

Simulation controller is implemented in **heat_transfer_simulation_utilities.py** module, which defines the infrastructure for the simulations to be run.

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

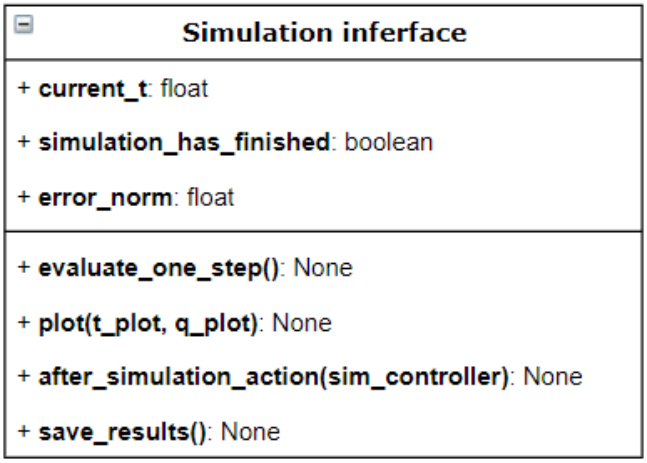

Figure 33 **API of the Simulation interface, showing its properties and methods**

The *Figure* above defines the API (external-facing attributes and methods) that the simulation controller is expecting. It can control literally any simulation that implements this interface.

Simulation controller consists of two classes – Callback class, and the SimulationController class.

Callback class

Callback class is responsible for updating the plots with calculated data, and also serves as a communication channel between the simulation and the main GUI thread in case of simulation through GUI.

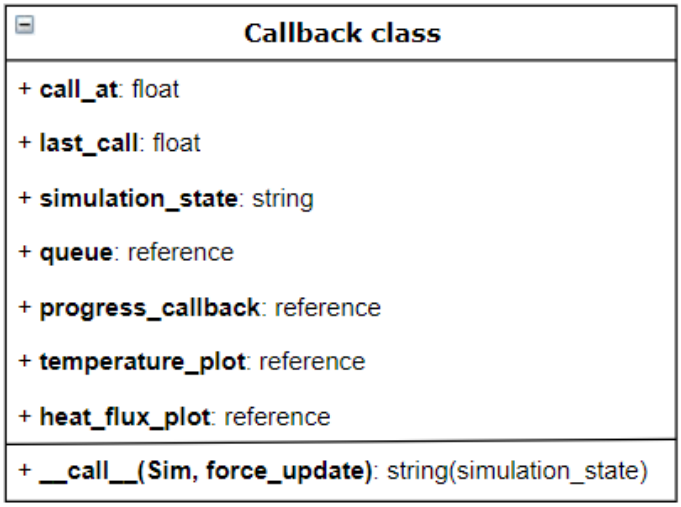

Figure 34 **Attributes and methods of the Callback class**

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__()**[8], which is responsible for updating the plot at regular intervals.

[8] __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 - https://stackoverflow.com/questions/5824881/python-call-special-method-practical-example

- It is 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 controller.
- 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 accordingly.
- It returns the current simulation state that resulted from the user input from GUI (if any), so the higher function calling it (the simulation controller in this case) can act accordingly to this simulation state.

SimulationController class

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.

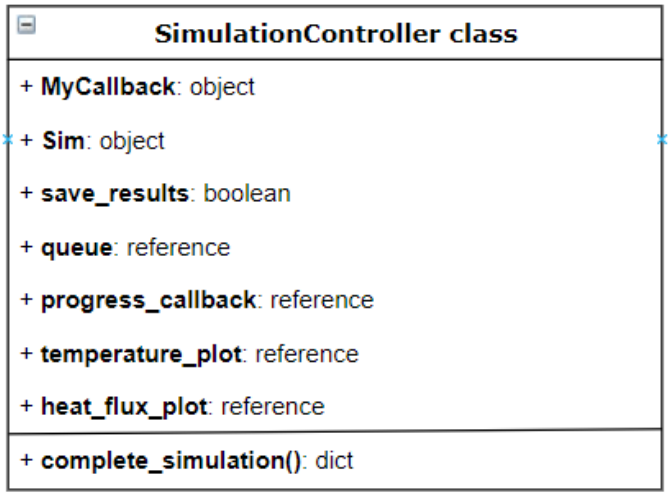

Figure 35 **Attributes and methods of the SimulationController class**

It takes quite a big number 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 the 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.

Classical simulation

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.

It implements the Simulation API, therefore needs to expose *at least* the attributes and methods defined by it.

+ **A**: array	+ **current_t**: float
+ **Exp_data**: object	+ **dt**: float
+ **HeatFlux**: array	+ **dx**: float
+ **K**: array	+ **error_norm**: float
+ **M**: array	+ **heat_flux_already_plotted**: boolean
+ **N**: int	+ **length**: float
+ **T**: array	+ **lmbd**: float
+ **T_amb**: array	+ **max_step_idx**: int
+ **T_data**: array	+ **rho**: float
+ **T_x0**: array	+ **robin_alpha**: float
+ **T_x0_interpolator**: callable	+ **simulation_has_finished**: boolean
+ **b**: array	+ **t**: array
+ **b_base**: array	+ **theta**: float
+ **cp**: float	+ **x**: array
+ **current_step_idx**: int	+ **x0**: float

Figure 36 **Attributes of the Simulation class**

As can be seen in the *Figure* above, this class has an enormous number of attributes, which is usually not desired. Therefore, we should consider splitting this class into more pieces. However, most of these attributes are used for internal purposes only, as the numerical calculation requires a lot of them.

Input parameters

It needs to be initialized with multiple parameters, which are describing the characteristics of the current simulation. These parameters are well described in the Testing chapter.

- 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)

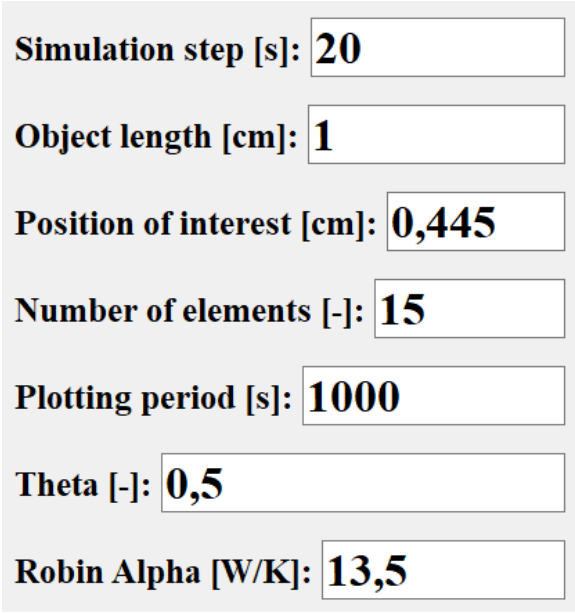

Figure 37 **Parameters for the simulation**

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[9] function, to evaluate the measured data at the time values the simulation will be using.
    - 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 element (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.

[9] https://docs.scipy.org/doc/numpy/reference/generated/numpy.interp.html

- 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 equations.
    - All the main parameters use so-called sparse matrixes[10].
        - 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).

Methods

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 this function 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[11] function.

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 for 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.

[10] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html
[11] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html

Calculate_final_error() Is calculating the error margin at the end of the simulation. 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.

Property **simulation_has_finished** 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 (simulation controller in this case).

Running the simulation

The Simulation class and the SimulationController class are connected together in a separate module, which is used for actually running the simulation. In our case, it 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.

Its **create_and_run_simulation()** method creates the Simulation object from inputted parameters, uses it to instantiate SimulationController object, and then calls its complete_simulation() method, which equals running the whole simulation.

simulate_from_gui() method is a connector between GUI and create_and_run_simulation() method. It takes all the parameters from the GUI and relays them further, returning back the results.

Inverse simulation

For defining the simulation 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).

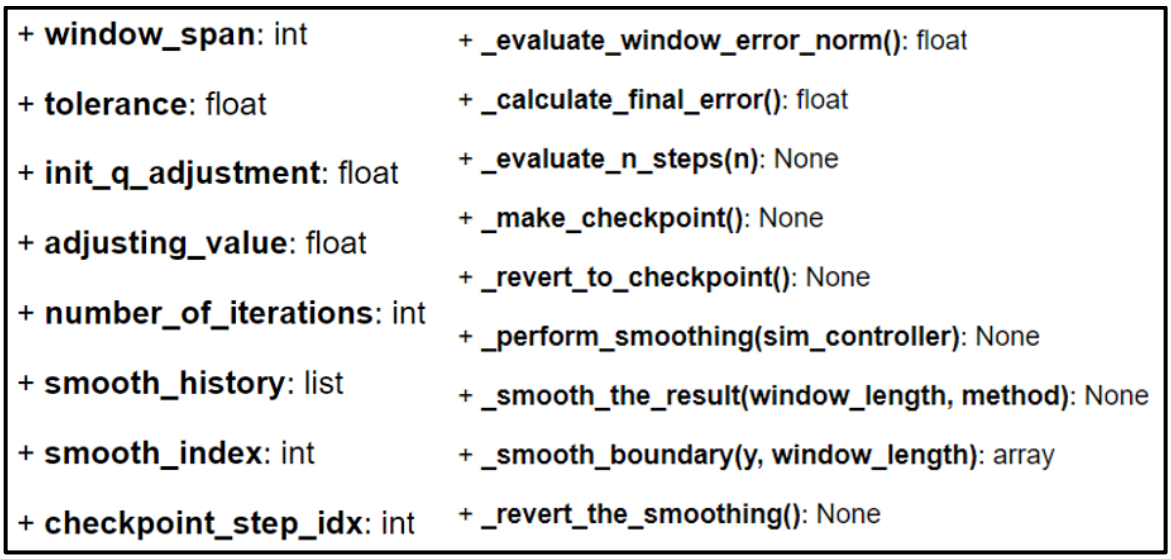

Figure 38 **Additional attributes and methods of the InverseSimulation class**

It takes multiple parameters on initialisation (including the same parameters as the parent Simulation class):

- window_span
- tolerance
- init_q_adjustment
- adjusting_value

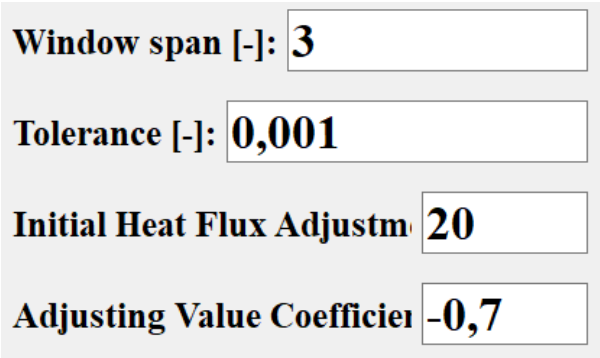

Figure 39 **Extra parameters for the inverse simulation**

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 between 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 window_span steps.

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 parent evaluate_one_step() function and according to the resulting error value, it is adjusting the heat flux little bit and runs the parent evaluate_one_step() again, until the error value is acceptable. We accept the heat flux when 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 (schemas are included afterwards):

- 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 or subtraction of this value to the current heat flux.
- Before running the evaluating function, we will assign the previously calculated heat flux value to all the heat flux points in the window_span we will be simulating - 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.
- 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 tolerance, we accept the estimated heat fluxes and perform the parent evaluate_one_step() function, with which we finalize the decision to use these heat fluxes.
        - We also decided to check if the value of the error itself is smaller than the tolerance so that we catch the cases when we get a good solution between two bad solutions – which would not be caught by comparing only the differences in error margin.
    - 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 a decimal number between -1 and 0.
- 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 only 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 (defined by the absolute difference between two consecutive values being less than our tolerance), we will accept the current estimation of heat flux and we will finish the step.

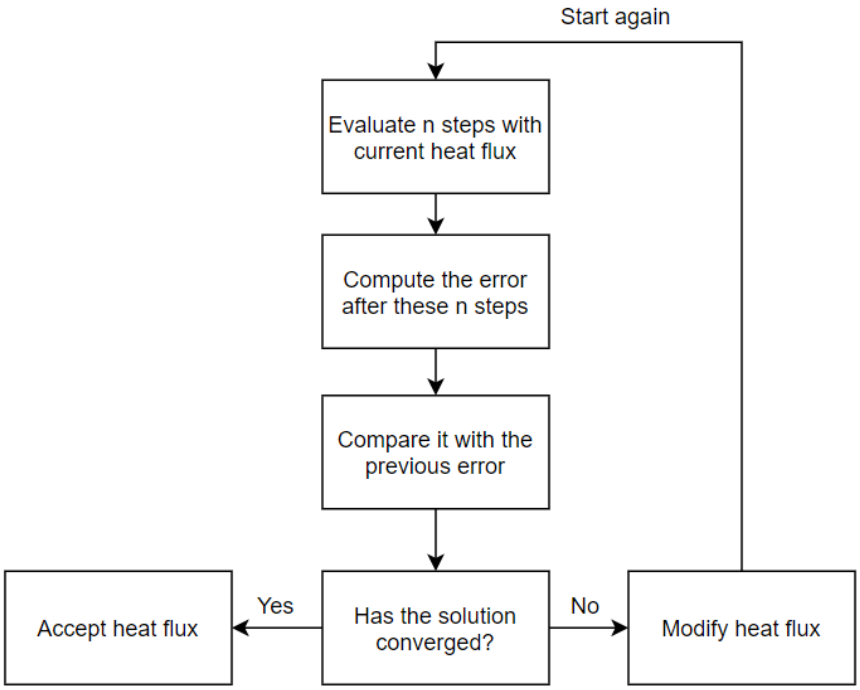

Figure 40 **Schema and a decision tree of determining the right heat flux**

The *Figure* above shows the logic of determining the heat flux by guessing „random" values of it, analysing the simulation and either accepting this heat flux, or adjusting its value and trying again.

The *Figure* below shows the decision tree when modifying the heat flux. When we find that we are converging to the right solution, we will continue adjusting in the same directions. Otherwise, we will switch the direction and decrease the adjusting value.

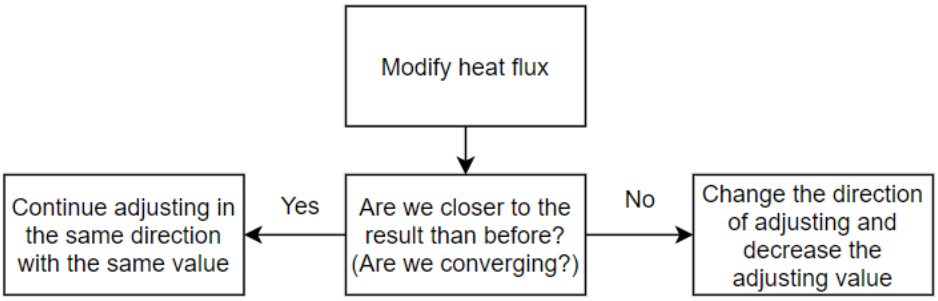

Figure 41 **Decision tree when modifying the heat flux**

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.

Method **after_simulation_action()** contains more logic than in the classic Simulation class, especially because it supports the possibility of **data smoothing**, in addition to determining the error margin. There are multiple methods for smoothing the final result after simulation finishes:

- **perform_smoothing()** is an overarching method for the whole smoothing process. It has a one-way connection with the GUI through the shared queue and is listening for the user input. According to the string value sent from the GUI, it will either finish the smoothing, revert it one step back, or perform a specified smoothing.
- **smooth_the_result()** is there to perform a specified smoothing when called. It implements all the possible smoothing mechanisms, at this moment the SavGol smoothing and the moving average smoothing.
    - **smooth_boundary()** is a helper method for the moving average smoothing. It is designed to finish the smoothing on the boundaries of the interval, which is by default not handled properly by a moving average smoothing we have in place.
- **revert_the_smoothing()** offers the possibility of going back in the smoothing history. Can be very useful when we want to experiment with different smoothing approaches, or we have accidentally performed a smoothing that looks visually bad, so we can always revert it to the previous state. It is remembering the whole smoothing history, so it is possible to always go back to the very beginning (to the state after the simulation finished)

Running the inverse simulation
To instantiate everything necessary for the inverse simulation and to actually run it, there is a module **heat_transfer_simulation_inverse.py**. The logic there is the same as already described for a classical simulation.

5.4 Performance testing and optimizing

The point of performance testing and optimizing is to make the code run as quick as possible.

Rules of optimization

It can be said that there are 3 rules of optimization, that should be abided by:

- **1. Don't.** Most of the time the performance of production software is already good enough, and it is not worth investing time into the improvements, that would not create a big impact on the whole service. However, there are some cases there execution speed can be the selling point of the software solution.
- **2. Don't ... yet.** First, the code should be completed from the logical point of view, and tests should be in place to make sure the software will perform exactly the same things even after the optimization.
- **3. Profile.** There is no way we can correctly judge where the most time is spent, as the behaviour of programming languages is very complex. Therefore, it is beneficial to use profilers, which can identify the places in the code that are adepts for optimizations.

[17], [18]

Process of optimization

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.

Results

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.

It should be also mentioned, that running the code on a more powerful computer was able to increase the speed **2,5-fold**, without any struggles with code optimization.

Discussion of performance improvements

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

Line profiler we used - kernprof[12] - 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[13] "@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 function – nicely identifying possible performance bottlenecks. [19]

```
Wrote profile results to heat_transfer_simulation.py.lprof
Timer unit: 1e-06 s

Total time: 1.08408 s
File: /mnt/d/Programování/Diploma-Thesis---Inverse-Heat-Transfer/NumericalForward.py
Function: evaluate_one_step at line 146

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   146                                           @profile
   147                                           def evaluate_one_step(self) -> None:
   148                                               """
   149                                               Simulating one step of the simulation
   150                                               Is using already initialised instance variables
   151                                               """
   152
   153                                               # Assemble vector b (dot() is matrix multiplication)
   154      4831     108967.0     22.6     10.1          self.b = self.b_base.dot(self.T)
   155                                                   # Apply explicit portion of HeatFlux (Neumann BC 1st node)
   156      4831      24927.0      5.2      2.3          self.b[0] += self.dt*(1-self.theta)*self.HeatFlux[self.current_step_idx]
   157                                                   # Apply implicit portion of HeatFlux (Neumann BC 1st node)
   158      4831      18495.0      3.8      1.7          self.b[0] += self.dt*self.theta*self.HeatFlux[self.current_step_idx+1]
   159                                                   # Apply explicit contribution of the body temperature (Robin BC Nth node)
   160      4831      19061.0      3.9      1.8          self.b[-1] -= self.dt*(1-self.theta)*self.robin_alpha*self.T[-1]
   161                                                   # Apply explicit contribution of the ambient temperature (Robin BC Nth node)
   162      4831      17751.0      3.7      1.6          self.b[-1] += self.dt*(1-self.theta)*self.robin_alpha*self.T_amb[self.current_step_idx]
   163                                                   # Apply implicit contribution of the ambient temperature (Robin BC Nth node)
   164      4831      17889.0      3.7      1.7          self.b[-1] += self.dt*self.theta*self.robin_alpha*self.T_amb[self.current_step_idx+1]
   165
   166                                                   # solve the equation self.A*self.T=b
   167      4831     816966.0    169.1     75.4          self.T = spsolve(self.A, self.b)  # solve new self.T for the new step
   168      4831      15606.0      3.2      1.4          self.current_step_idx += 1  # move to new timestep
   169      4831      31698.0      6.6      2.9          self.T_x0[self.current_step_idx] = self.T_x0_interpolator(self.T)  # type: ignore
   170
   171                                                   # Incrementing time information for the callback
   172      4831      12722.0      2.6      1.2          self.current_t = self.dt * self.current_step_idx
```

Figure 42 **Output from performance profiling with kernprof**

Figure above shows the sample output from a performance profiling of the evaluate_one_step() method, which is the main method of our computation engine. As can be seen in the "% Time" column, the majority of the execution time (75 %) is being spent on solving the system of linear equations.

5.5 Data smoothing

Smoothing the data is helping to get rid of random errors that happened during the simulation. We used data smoothing as a tool to increase the quality of the inverse solution.

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,

[12] https://github.com/rkern/line_profiler
[13] https://www.geeksforgeeks.org/decorators-in-python/

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.

We implemented two possibilities of performing the smoothing – pure moving average method, and Savitzky-Golay filter.

5.5.1 Moving average method

This function is evaluating the average value of a certain number of points around a certain point, including that certain point, and it is assigning this value as a new value of that point. Un its implementation, it uses numpy.convolve function[14]. [33]

When the number of points is odd, 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 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 *Figure* below, the final result, in other words, the smoothness, is highly depending on the window span. The higher the window span, the more points it averages, and the smoother the line is.

[14] https://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html

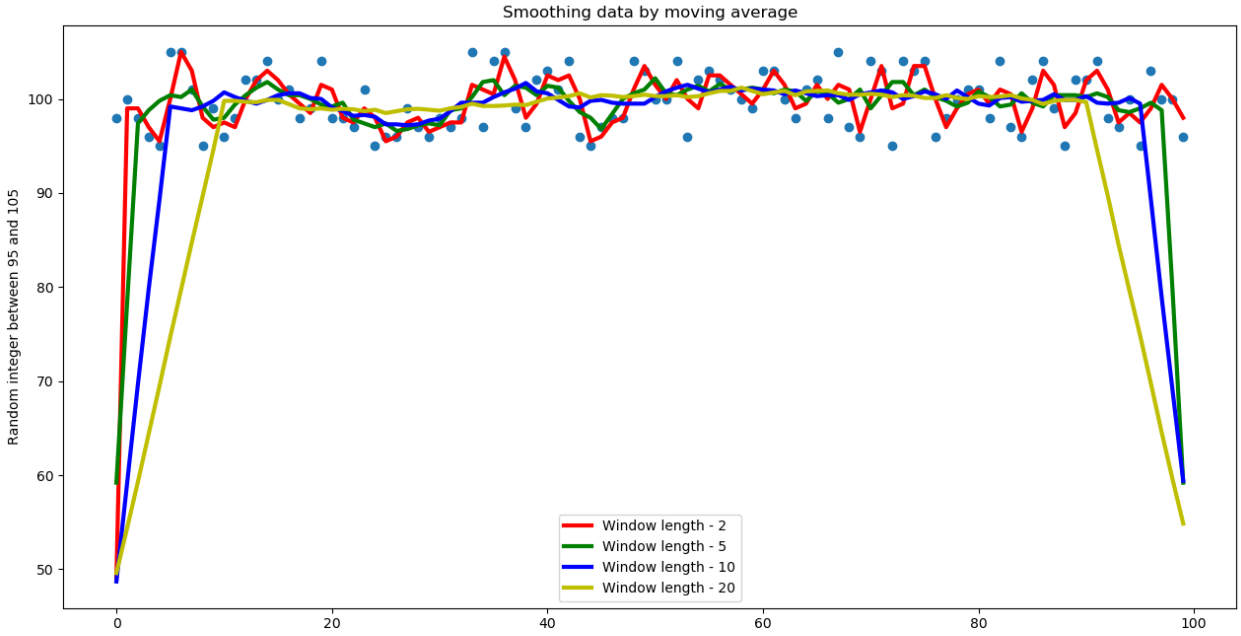

Figure 43 **Smoothing data by moving average – initial method**

The basic 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 *Figure* 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).

Analysing the resulting *Figure* below, we see that the higher window spans have higher smoothing effect – smoothing with window span of only two still creates a lot of sudden spikes; the smoothing with values of 10 or 20 already shows much more stable progress of the smoothed result.

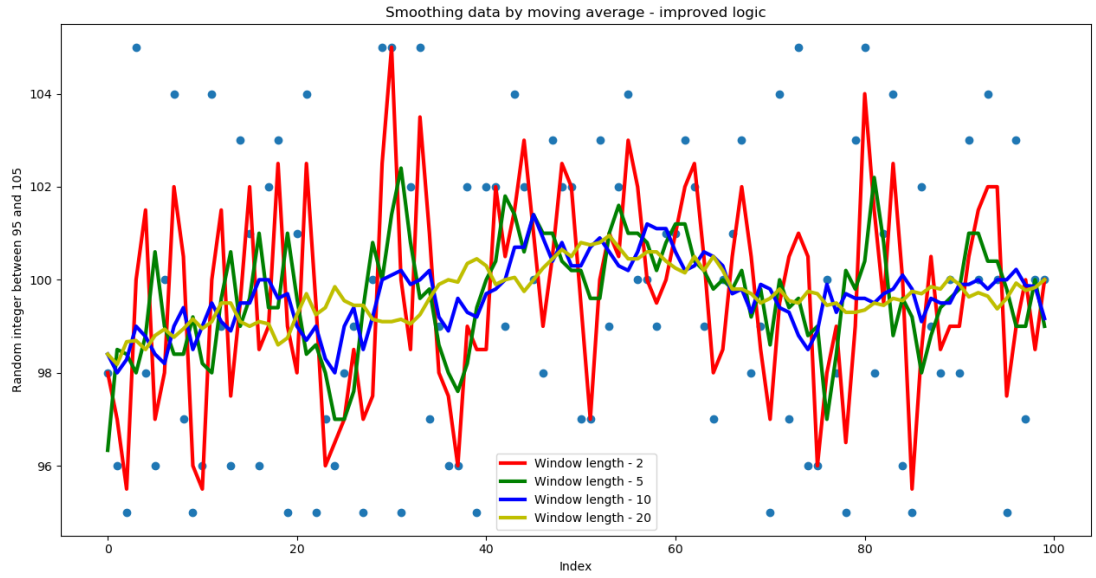

Figure 44 **Smoothing data by moving average – improved method**

5.5.2 Savgol filter (Savitzky-Golay filter)

This method achieves very similar results as the moving-average method, and it also uses np.convolve in its implementation. [34]

However, in contrast to the initial implementation of the moving average method, it deals with boundary points in an expected manner by default.

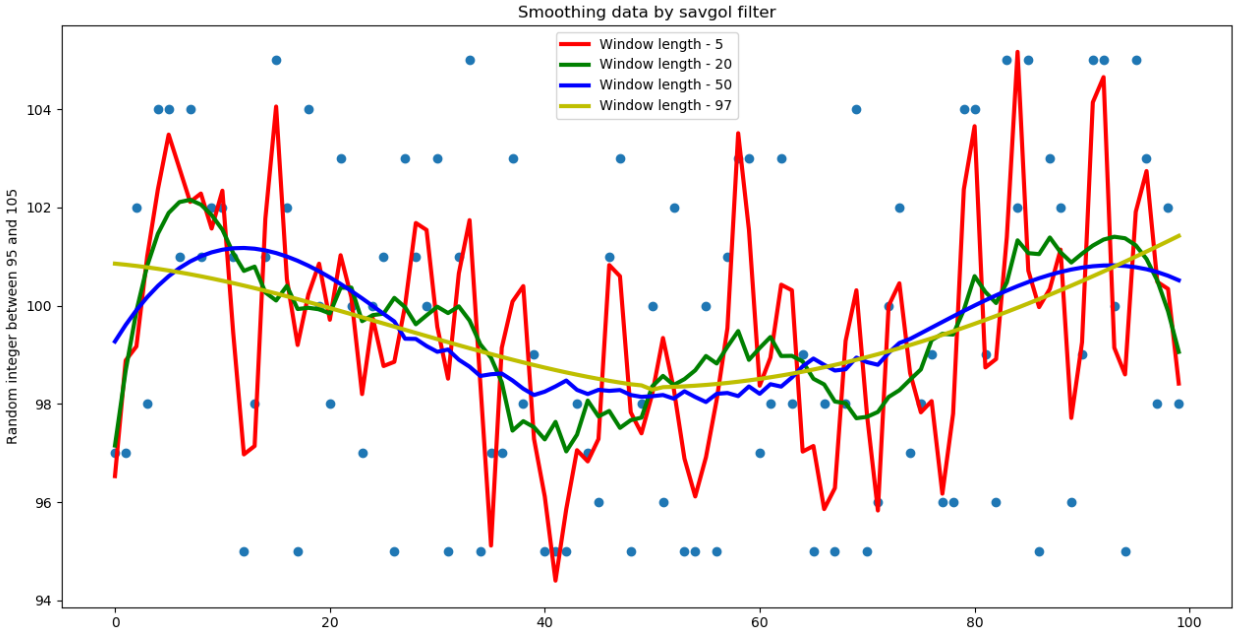

Figure 45 **Smoothing data by Savitzky-Golay filter**

5.6 Dependences and libraries

The whole project could not be created just by "pure" Python, as the standard library does not include some more advanced functionality used for computing and visualising the results. Therefore, we needed to use external (third-party) libraries (dependencies).

Special file, *requirements.txt*[15], 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[16] for Python must be available, but it comes packaged with all newer Python versions - 3.4 or higher.

It can be beneficial to set up a virtual environment[17] just for this project so that the specific libraries for this project are not "polluting" the general Python environment on the machine, and also the versions of libraries can be specified freely for each project.

Matplotlib

It is the most widely used Python library for plotting graphs[18]. 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 compared with other data visualisation frameworks, like plotly[19], which is otherwise also very user-friendly.

Numpy

It stands for Numerical Python[20]. It focuses mostly on defining data structures and functions to efficiently work with large collections of objects, called arrays. Because most of its functionality is written in C language, it usually performs operations with numbers much more quickly than native Python.

[15] https://medium.com/@boscacci/why-and-how-to-make-a-requirements-txt-f329c685181e
[16] https://www.w3schools.com/python/python_pip.asp
[17] https://pipenv.readthedocs.io/en/latest/
[18] https://matplotlib.org/
[19] https://plot.ly/python/
[20] https://numpy.org/

Scipy

It stands for Scientific Python[21]. It provides a lot of tools for mathematics, science and engineering applications of programming. It is dependent on numpy, as it internally works with its arrays. Examples of use-cases include integration, statistics or signal processing.

In our case, Scipy had to be downgraded from the current version 1.3.3 to version 1.2.1 to overcome problems with compatibility[22].

PyQt5

It was already described well in the part about the GUI choice.

[21] https://www.scipy.org/

[22] https://github.com/scipy/scipy/issues/11062

6 Software results and conclusions

This chapter presents the final piece of software, includes a manual for its use, and discusses interesting pieces of development.

6.1 The basic flow of the software

When the user chooses all the desired parameters at the left side of the GUI 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).
- References to both plots and 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.
- During the simulation, the calculating thread is updating the graphs in the GUI, according to the inputted frequency of updates. Simulation time is also being incremented in GUI while the simulation is running. Pause and Stop buttons are active and are performing appropriate actions on the simulation.
- After the simulation finishes, it sends back the error margin of the simulation, and the internal state in GUI will be reflected.
- In the case of the inverse simulation the smoothing options will show up, that can influence the final result.
- When specified by the user, the results of the simulation can be saved in a data file and as a picture.

6.2 User guide for the software

This chapter is describing the necessary information users need to know in order to use the software as intended and to avoid possible confusions.

Before launching the program

CSV file with data from the experiment must be prepared.

The first line is reserved for the header, which is describing the columns in the file. These columns must have a defined name, for the specific data to be recognized (what is time, what is temperature, etc.).

The naming convention, together with a description of that data, can be seen in the table below, and an example of a real data is given in the picture below.

Table 1 **Experimental data**

Value	Unit	Column name	Description
Time	Seconds	Time	Time from the beginning of the measurement
Temperature	Degrees celsius	Temperature	Temperature measured inside the body
Heat Flux	Watts	HeatFlux	Heat flux applied to the body
Ambient Temperature	Degrees celsius	T_amb	Ambient Temperature in the room

```
Time,Temperature,HeatFlux,T_amb
0,21.1339,1.3009,20.732
3.189,21.1143,1.1495,20.7312
4.239,21.1229,1.154,20.7314
5.291,21.1117,1.2747,20.729
6.342,21.1256,1.2633,20.7295
7.397,21.1189,1.2405,20.7289
8.449,21.1155,1.1358,20.7284
9.505,21.1296,1.2815,20.7281
10.558,21.1142,1.1426,20.7278
11.614,21.1134,1.0994,20.7271
12.666,21.1275,1.2337,20.7265
13.72,21.1126,1.2451,20.727
```

Figure 46 **Sample experimental data**

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 of the plots in .png format and the calculated data in .csv format.

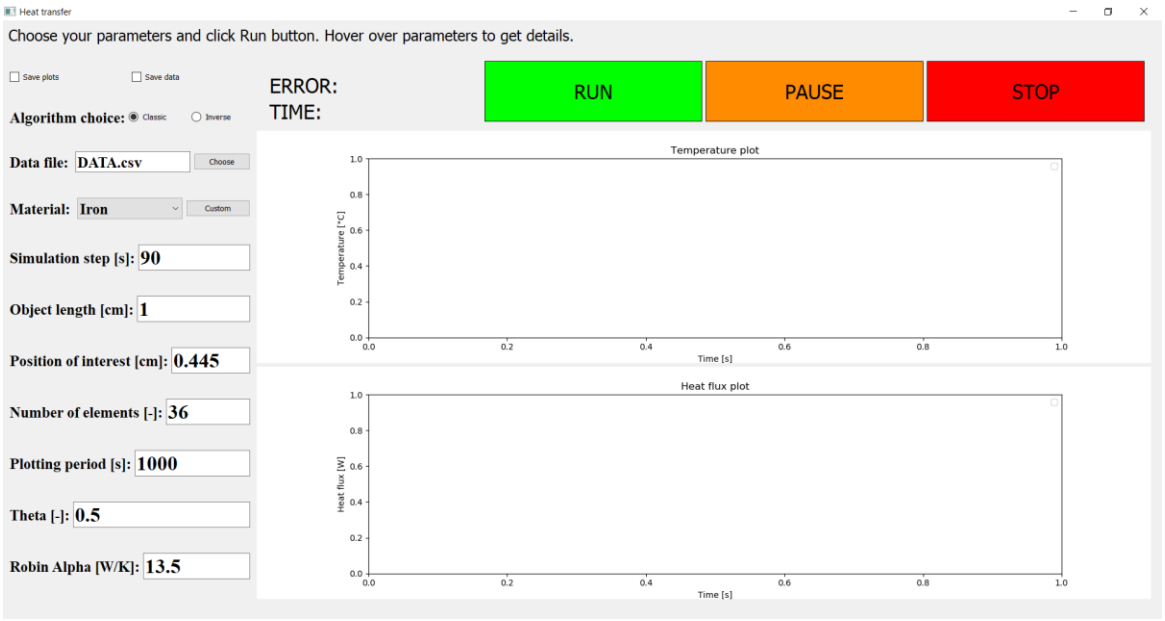

Figure 47 **GUI after starting the application**

Below the checkboxes, 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.)

Location of CSV file with data can be specified by choosing a file from a file explorer.

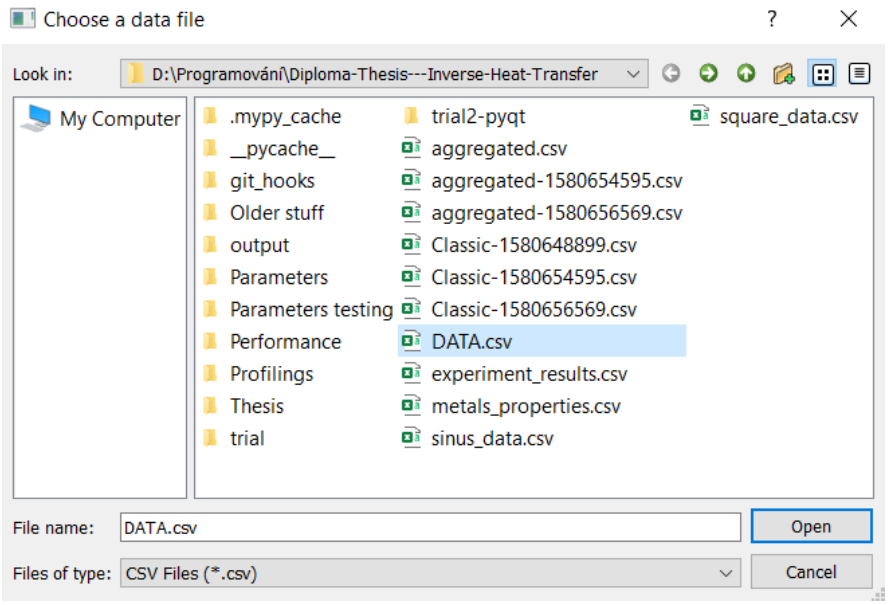

Figure 48 **Choosing a location of a data file**

Below that 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).

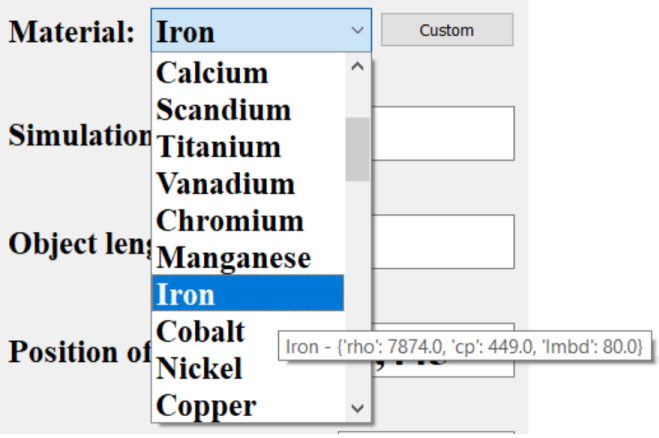

Figure 49 **Material choice dropdown**

Users also have a choice of defining their own materials, with custom properties.

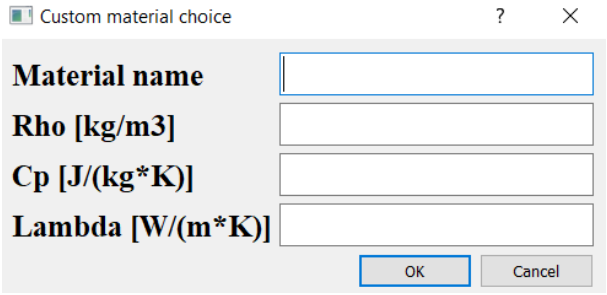

Figure 50 **Custom material choice window**

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 more parameters than the classic one because inverse problems are more complex and allow for higher customization. (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.

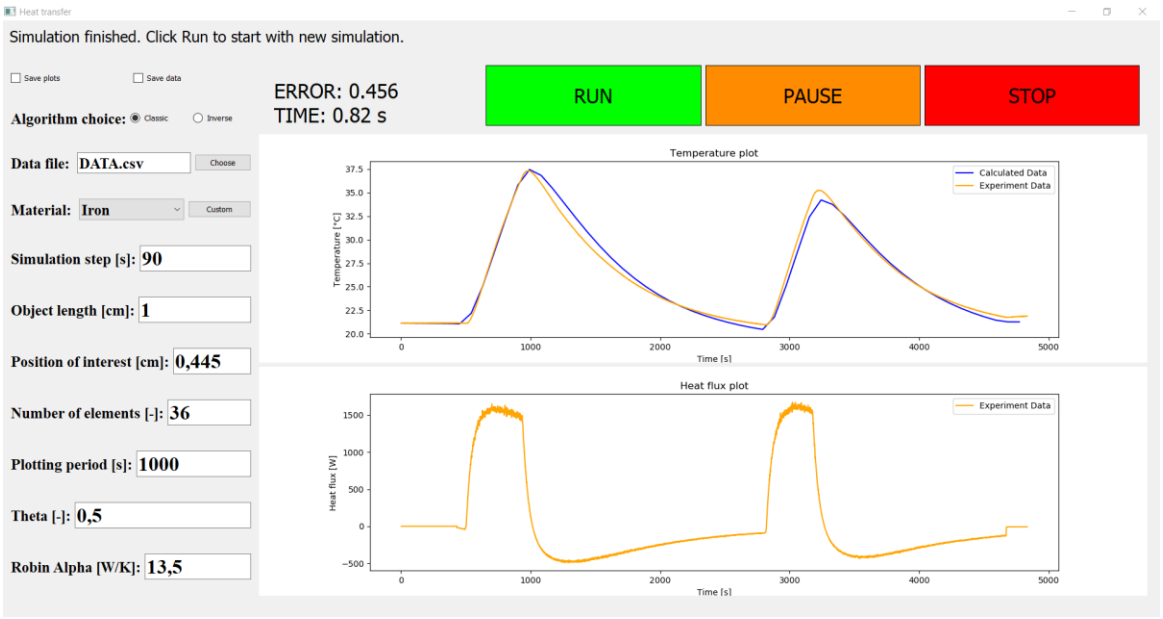

Figure 51 **GUI at the end of the classic simulation**

After the simulation finishes

In case of an inverse problem, there is a possibility of smoothing the results appearing in the left menu - clicking “Smooth” will perform the action.

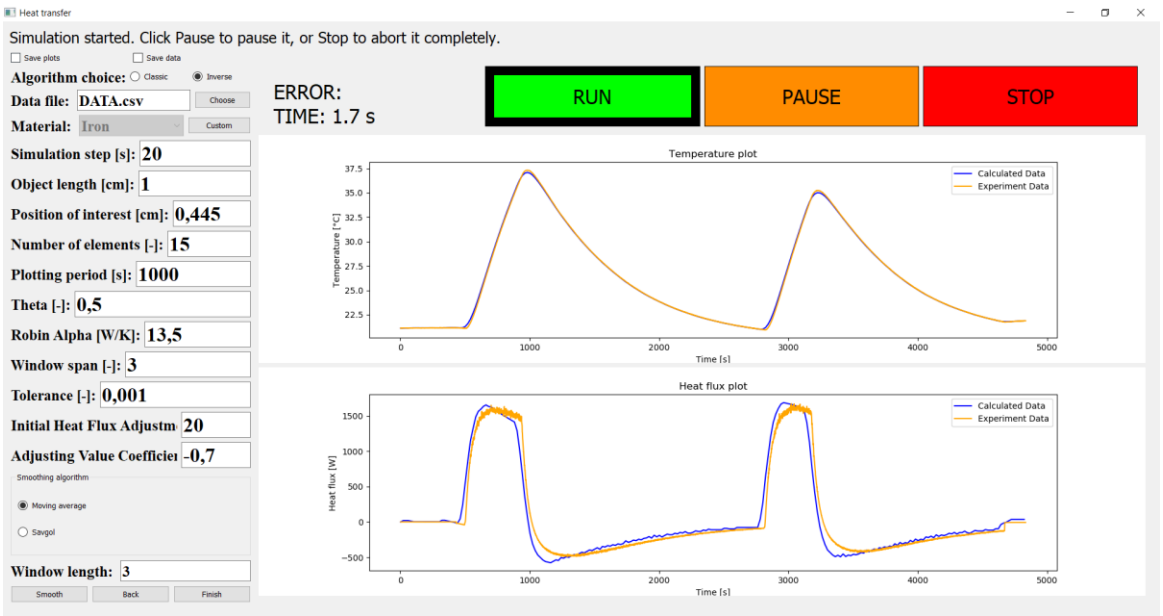

Figure 52 **Smoothing panel at the bottom left after the inverse simulation**

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.

6.3 Programming concepts worth pointing out

During the development, we encountered some of the basic programming paradigms, that are worth exploring more. Combination of these concepts and practices can significantly increase the quality of the software, both from the readability and reliability perspectives.

Some other concepts, like multithreading, performance profiling, or specifics of GUI creation, were already described in other parts of the thesis.

Using classes

Classes are used for aggregating common properties and behaviours of a certain software component, which can be completely isolated from the rest of the code. We are using classes extensively throughout the whole project, because it allows us to unite common functionality into its own modules, so the behaviour of the application can be understood by smaller pieces, as each module is responsible for a certain action (defining user inputs, providing material data, parsing experimental data, etc.). [20]

All these modules are then imported in the main module, which creates individual instances (objects) out of these classes and combines them all together.

One instructive and very minimalistic use of a class is to hold a small set of data, that together have some higher meaning. An example could be storing material properties inside one simple class, holding just three attributes – density, heat capacity and heat conductivity:

```
class Material:
    def __init__(self, rho, cp, lmbd):
        self.rho = rho  # Mass density
        self.cp = cp  # Specific heat capacity
        self.lmbd = lmbd  # Heat conductivity
steel = Material(7850, 520, 50) # Instantiating the steel material
```

Especially during the development process, it can be very beneficial to create a textual representation of the class, defining how the class instance should be displayed when using print() on it. The debugging process regarding the class arguments can be made easier by

this. It is being done by implementing the __repr__()[23] magic method, which has access to all instance variables and can display them in an arbitrary way.

Class inheritance and method overriding

We used class inheritance in our computation engine, where the InverseSimulation class is inheriting from the Simulation class.

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). [21]

Version control

Version control gives us the ability to store the whole history of the project, piece by piece (commit by commit), and allows for easier collaboration of many people at the same project.

It is a priceless tool in software development, that makes sharing code between more people much easier and transparent than other methods (sending emails, using Google Disk, etc.).

Using some online repository for storing code has also the benefit of acting as a backup of the code – the code is stored not only at one local place, where it can be vulnerable to loss of data or its corruption. [22]

We are using git[24] as a version control system, and Github[25] as a service for hosting git repositories.

Design patterns

Design patterns can be described as well-known and well-tested solutions to commonly occurring problems. Their use in software creation is highly encouraged, as it usually increases the overall readability of the solution, as it is probable, that the reader of the code is already familiar with this pattern and understands its purpose. It also creates a solution that already stood the test of time, and the probability of bugs or unexpected behaviours can be highly reduced by implementing them. [23]

In the interpolation module of the computational engine, we encountered a problem of how to dynamically create instances of interpolation class. The reason for that is sometimes we

[23] https://docs.python.org/3/reference/datamodel.html#object.__repr__
[24] https://git-scm.com/
[25] https://github.com/

need to interpolate value only in one point, but sometimes in more places, and special custom classes were designed for both of these purposes. Dynamic creating of new objects, and hiding its implementation, is the purpose of factory design pattern. Therefore, in the interpolation module, we created our factory function, that is determining which object to create, according to the current need. The main part of the code can just call this function, be returned with an object, but does not have to know about the implementation details. [24]

Testing

As the software projects grow bigger, there is an increasing need to make sure all the functionality is always kept in a working state, and new features will not introduce problems in already created behaviour. Making sure all the software components are always compatible with each other, and everything is working as expected, is the main goal of testing. [26]

There happen to be two kinds of testing, which differ in the scope they are testing, meaning the amount and complexity of the tested code.

Unit tests are mostly interested in individual functions, testing only a very small and specific piece of code – supplying some input to the function, and awaiting specified output.

Integration tests, on the other hand, are focused on testing bigger parts of the code and are usually making sure more software components are cooperating together in the way they should.

In the project, we use mostly integration tests, to make sure the whole computation engine as a whole is yielding right results when well-known and well-tested simulation parameters are defined. We are expecting the simulation not to throw any errors and return a certain amount of precision. If any errors are encountered during the whole simulation, or the returned error margin at the end of the simulation is suspiciously high, it is a red flag to us, that something is not working properly, and that we should closely examine all the changes we have done recently.

Having well-defined and deterministic test cases (returning either success or failure) can be advantageously integrated into the version control system. Git, our choice of a version control system, is offering mechanisms called git hooks[26], which can make sure, that code not passing the tests (and therefore most probably containing mistakes) cannot be even committed or pushed to the repository. Before every commit, it can automatically run all the defined tests, and in case of any problems, will reject the commit and notify the user to check the code. It is a very convenient tool, that can significantly reduce the number of bugs, that are accidentally committed to the shared repository, and therefore highly increasing the consistency and reliability of the software. [25]

[26] https://githooks.com/

Using the naming convention for methods

Dividing functions and methods into public and private ones contributes to the easier readability of the code and mainly of the cooperation between multiple parts of the code.

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 the 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.

[27]

Static analysis of the code

Python is by nature interpreted programming language, which means the lines of code are encountered by interpreter only at runtime, the code does not have to be compiled beforehand, as opposed to languages like Java or C++. The distinction can be also described as interpreted languages are usually dynamically typed (Python), whereas compiled languages are mostly statically typed (Java).

One of the advantages of interpreted languages is the superior portability of the code, which can be moved easily as a text file, and modified without issues. It is in contrast to compiled languages, where modifying even a single line of code must be followed by the compilation of the whole application.

Drawbacks of dynamic typing 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, that is not extensively covered by tests).

In the project, we are using the mypy library[27] to perform static analysis of the Python code – we aim to combine the benefits of both approaches, having a dynamically typed code with static analysis of it.

Using mypy requires placing type hints in the code, most often to the function declarations, to define the types of variables that are being inputted and outputted. These types include numbers, text or list of values. Annotating a function with these type declarations will cause mypy to analyse all the usages of the function, and determines if it follows the rules we have defined.

However, static analysis of the code with mypy does not bring only benefits. Sometimes the analysis uncovers “errors”, that are not actual errors, and these have to be manually silenced in the code (through a specific comment), which can make the code less readable and clear. Also, the majority of third party libraries do not contain these type checks, their imports usually have to be silenced as well, not to cause unexpected errors.

[27] https://mypy.readthedocs.io/en/stable/introduction.html

[28]

Error handling + creating custom exceptions

Because there is quite a lot of logic in the whole application, and many arguments are being inputted from the user, there is a need for a solid error handling, in order to avoid confusion, and, most importantly, for uncaught errors not to crash the whole application. It is also a good practice to tell users, why the error happened, so they can avoid causing it again in the future if it was caused by their actions or inputs.

As the application is divided into two parts – GUI and a computation engine – error handling needs to be set up in both parts individually.

Implementing proper error handling in computation engine is not so crucial, because it is being run in its own thread (separated from the GUI thread), and its failure itself cannot crash the GUI. In the event of an unexpected failure, the simulation would just seemingly stop, which could be a good sign itself for the user, that something has gone wrong, and they should, for example, examine the inputted parameters. However, a much more user-friendly way is to propagate the error to the user, because it can precisely point out the root cause of the error.

One example of propagating specific errors from computation engine to the GUI is the inability to parse CSV file with experimental data. When the computation engine is unable to correctly process the file, or it finds out some necessary data is missing there, it will throw our custom-made exception, and the user will be notified of this problem, with precise steps how to improve the situation.

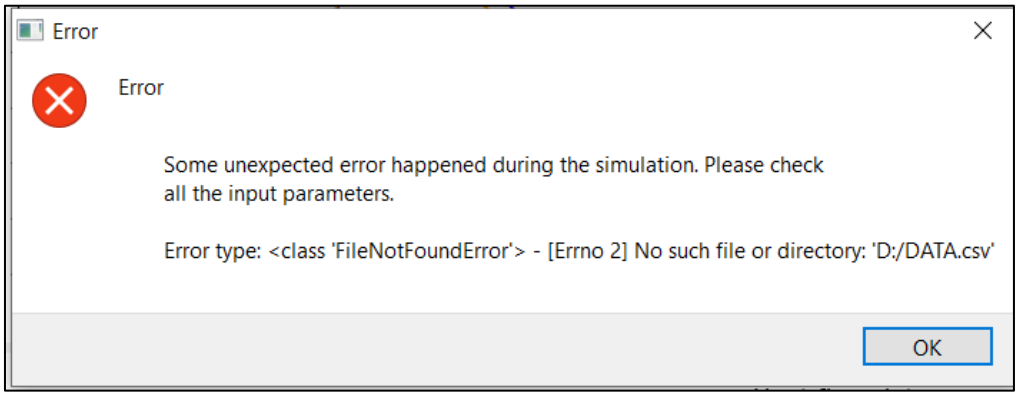

Figure 53 **Example of the error caused by a missing data file**

Example of catching GUI-specific errors include checking, if a file with material data is present, and if not, the users are notified and advised to include it.

All the errors eventually appear in GUI as popup windows with describing the cause of the error, and possible steps, how to resolve it.

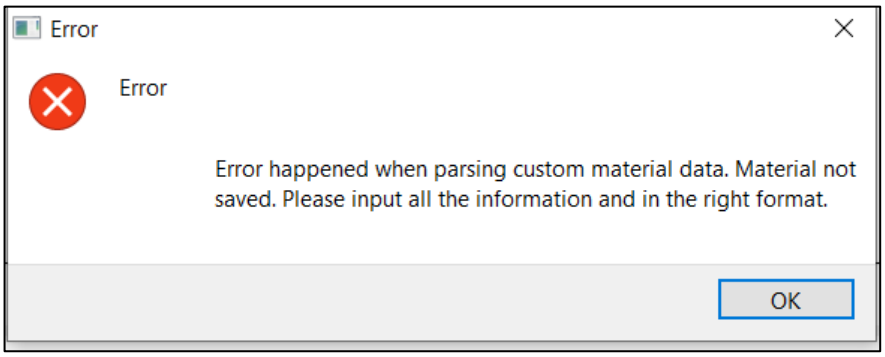

Figure 54 **Example of the error caused by bad material data input**

6.4 Interesting bugs worth pointing out

After being involved in coding for some time, experience says that there can be bugs without programming, but there can be no programming without bugs[28]. Below is a couple of them that arose during the development of the app – and almost certainly there are many others, not uncovered yet.

At some point, we were not 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.

There are slight differences in Operating Systems between Windows and Linux - and these 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.

In the GUI, 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.

When parsing and processing the CSV file with experimental values, we are using the csv library, which is in the standard library. First, we were using the csv.reader object for that, which is processing the lines in CSV file according to their position (index) in the row. Because the first line of a file was including headers to describe the columns in the file, we were skipping this first line, so we are taking only the experimental (number) values. Later, we replaced the csv.reader with csv.DictReader object, which is processing the file according to column names defined in the header row, as opposed to the index (as it has various benefits for the users). It also has a different behaviour, in that it is not regarding the header row as data it should read, it skips it automatically. However, we still included the manual skipping of the one line, and therefore we were missing the first row of experimental data.

[28] By bugs we mean mistakes in code, that cause the program to not work exactly as wanted, or performing some unexpected behaviour.

7 Testing of the software

The goal of this chapter is to verify the precision of the designed simulations, and also use the simulation to explore the influence of its various parameters on the final result.

7.1 Comparing the results with the real experiment

As a measure of correctness, we compared the achieved results with the theoretically right results, coming from the real experiment.

In both graphs (temperature and heat flux), we are plotting both the calculated data (blue line) and the real experimental data (orange line). The more similar the calculated results are to the experimental results, the more precise the simulation is.

Figure below describes the result of a forward (classical) simulation, determining the temperature from the knowledge of heat flux.

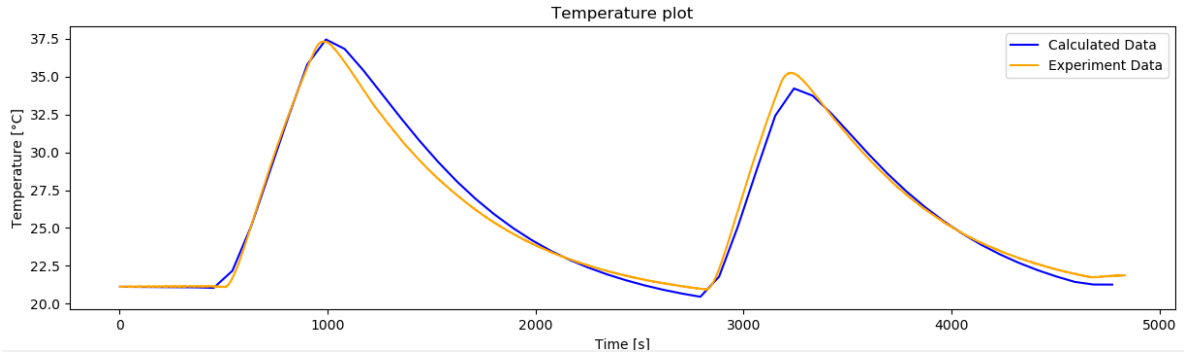

Figure 55 **Compared results of the forward simulation**

Figure below, on the other hand, shows the result of the inverse simulation, determining the heat flux from the knowledge of temperature.

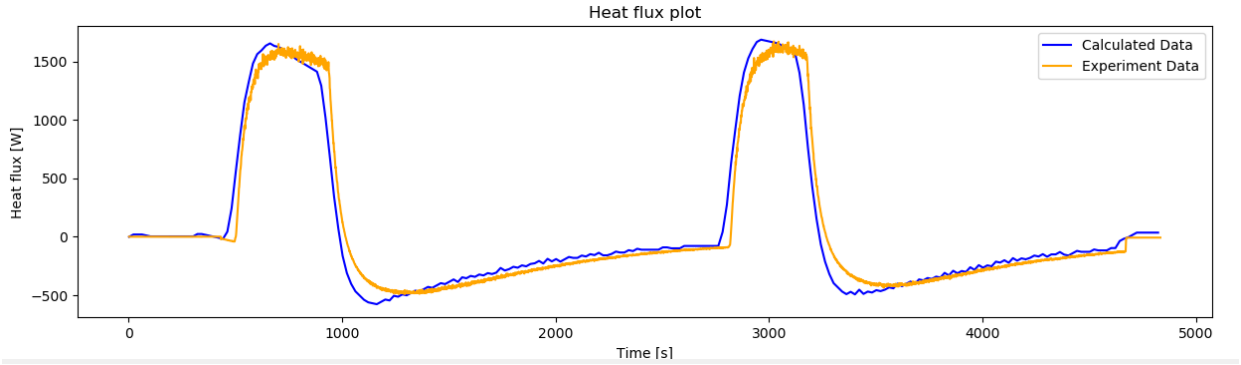

Figure 56 **Compared results of the inverse simulation**

Both results above can be regarded as satisfactory, considering the relative simplicity of our calculation models.

To express the precision numerically, we were also calculating the error value for these comparison simulations. This error value was calculated as the average absolute difference between the calculated value and the experimental value in all the simulated points.

The error value is always expressed in units on the y-axis, therefore for the forward simulations, its units can be regarded as °C, and in case of inverse simulation, it will have units of W.

In the forward simulation, we were achieving error values around **0,45 °C**. It means that the calculated temperature in each point is on average about half a degree different than its theoretical value. However, the absolute value alone cannot determine the precision – we also need to account for the temperature range of the simulation (the lowest and the highest temperature in the simulation). As the lowest temperature was about 20 °C, and the highest 37 °C, the level of precision we achieved was **0,45 °C on 17 °C range**. Transferring this into the percentage count, the average absolute error is **2,65 %**.

In the case of the inverse simulation, the error value was about **80 W**. It means that on average the calculated values are 80 W different than the experimental ones. In the terms of relative and more reflective terms, it achieved the precision of **80 W on 2000 W range**. In percentage terms, the average absolute error is **4 %**.

7.2 Parameters inputted from the user

The simulation of heat transfer is highly customizable. Users are free to choose a lot of parameters themselves and therefore influence the simulation. Following part describes these parameters and makes a preparation for the following testing of all these parameters.

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.

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 should 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 decimal number. Is expressed in seconds.

Theta

Determines explicitness (theta being 0) and implicitness (theta being 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.

There are three distinct values of theta. Value 0 means the approach is fully explicit, offers converge of 1^{st} order, and this approach is numerically unstable. Value 1 signs fully implicit approach, also has the convergence in the 1^{st} order, and in contrast to the fully explicit approach, is numerically stable. The third interesting point lies at value 0.5, which is called midpoint, or Crank-Nicolson point – with this approach, the convergence is in 2^{nd} order, and also the solution is 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 also 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, that can be calculated.

Can be an arbitrary decimal value (float) between 0 and 1 and has no units.

[36], [37]

Robin Alpha

Defines the convection heat transfer coefficient at 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 and should have a positive impact on precision.

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. Defines 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. Defines 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.

Can be a decimal value between -1 and 0 and has no units.

7.3 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 these parameters.

```
parameters = {
    "rho": 7850,
    "cp": 520,
    "lmbd": 50,
    "dt": 3,
    "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,
    "experiment_data_path": "DATA.csv"
}
```

Figure 57 **Default parameters used for testing purposes**

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.

7.3.1 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.

Testing framework

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 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, to have a visual representation of the outcome.

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.

```
amount_of_tests = 5
input_values = [1, 2, 3, 4, 5]
for _ in range(amount_of_tests):
    for value in input_values:
        run_simulation(value)
```

Figure 58 **Pseudocode describing the testing framework**

The whole methodology can be summed by the *Figure* above, showing pseudocode for the testing framework. On a high level, we are only defining the number of tests and the input values we want to test – the rest is handled by the framework itself, and is, therefore, an implementation detail.

Defining lists of values

In order to make the tests as automated as possible, so that they require user input only once at the very beginning of testing, it is useful to have the ability to define more test scenarios at once – instead of manually changing the arguments after each test simulation. This can be achieved by creating a list of values, that will be looped over, and simulation will be carried out with each value in the list.

There are multiple possibilities for defining the list of values in Python, and each of those can be 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. [29]

Using numpy library

Numpy offers multiple functions to define a sequence of number values, and each of them has its unique use.

Linspace

numpy.linspace(start=1, stop=50, num=100)

"Return evenly spaced numbers over a specified interval." [30]

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.

Logspace

numpy.logspace(start=-5, stop=-1, num=100, base=10)

"Return numbers spaced evenly on a log scale." [31]

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).

Arange

numpy.arange(start=0.5, stop=1, step=0.01)

"Return evenly spaced values within a given interval." [32]

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.

7.3.2 Results of the parameter testing

Classic problems

Number of elements

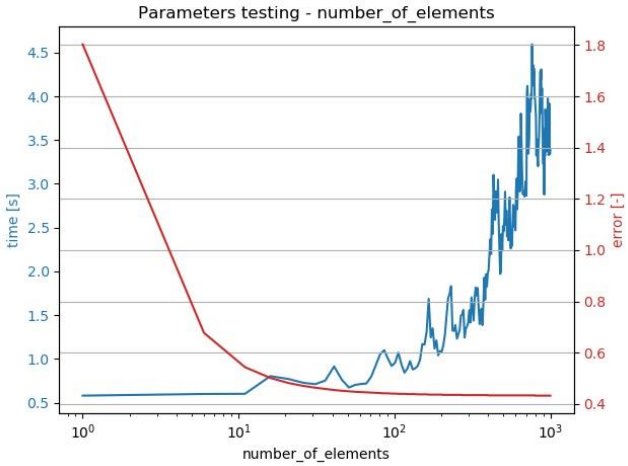

Figure 59 **Influence of number of elements – logarithmic view**

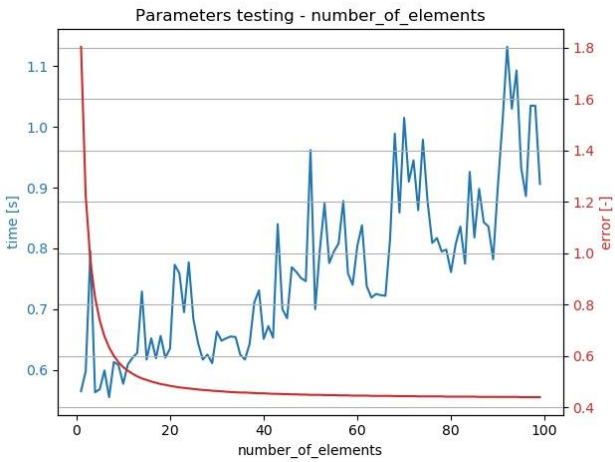

Figure 60 **Influence of number of elements – optimal range**

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 – therefore the smaller error margin.

It can be observed that we already get a reasonable result by using only around 10 elements. The error margin is also decreasing further even when we are in higher orders.

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 (timestep)

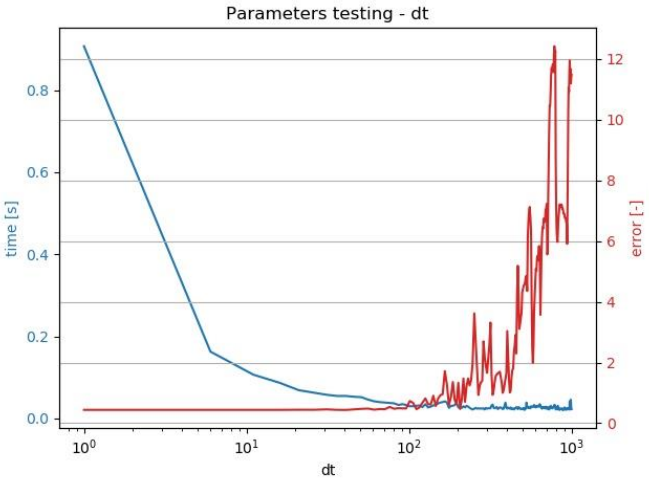

Figure 62 **Influence of timestep – logarithmic view**

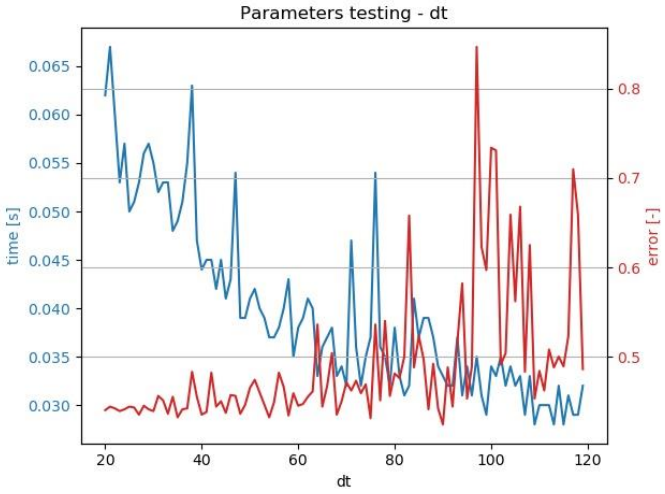

Figure 61 **Influence of timestep – optimal range**

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

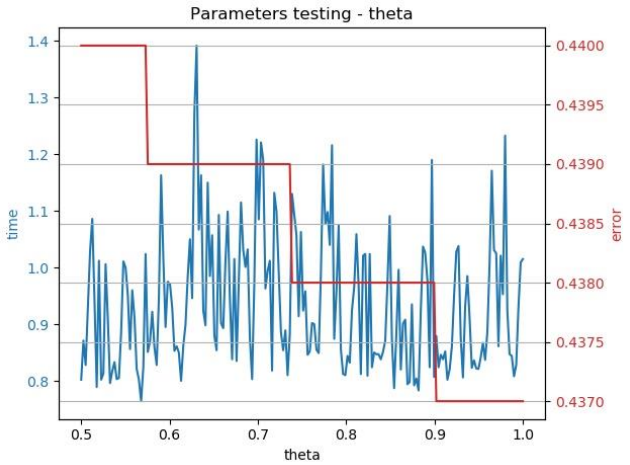

Figure 63 **Influence of theta**

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 simulation time does not look like being dependant on theta at all.

Inverse problems

Number of elements

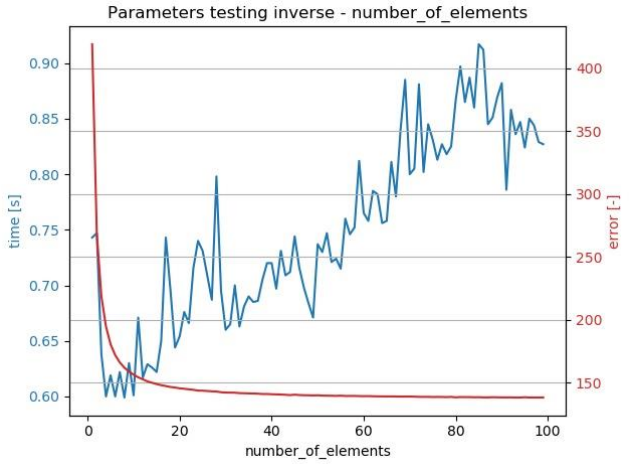

Figure 64 **Influence of number of elements in inverse**

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

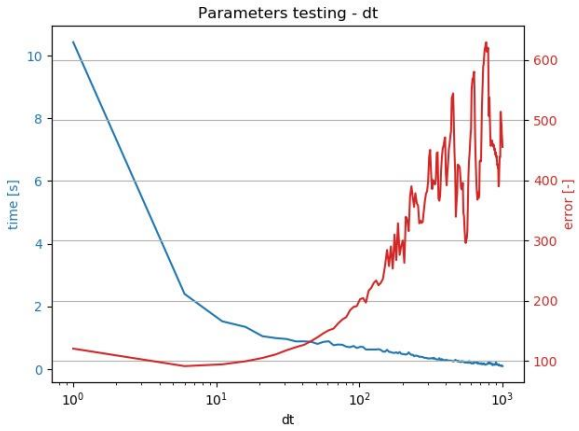

Figure 66 **Influence of dt in inverse – logarithmic view**

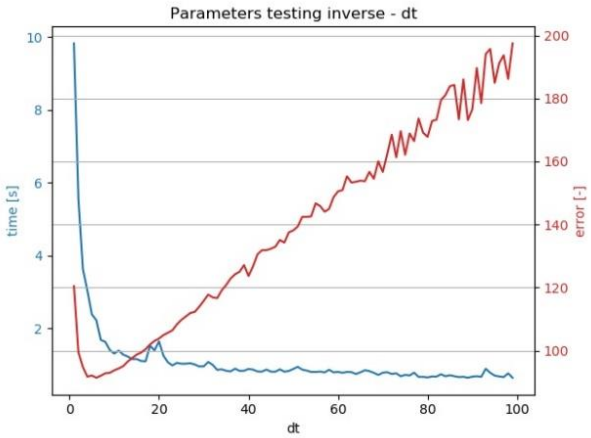

Figure 65 **Influence of dt in inverse – optimal range**

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.

When we carried out tests with higher window span (7), 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.

Discussion: Therefore, we can recommend the time step being lower than in classical problems, around **20**. It should be also mentioned that the very smallest values of time step (1-5) are causing the heat flux plot looking very badly, with a lot of sharp edges. According to computer calculations, the optimal value lies at **33**.

Theta

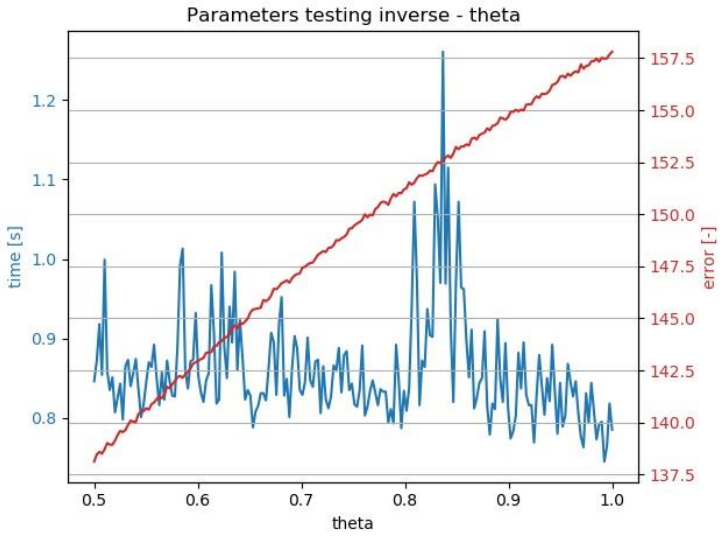

Figure 67 **Influence of theta in inverse**

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

Figure 69 **Influence of window span in inverse – initial results**

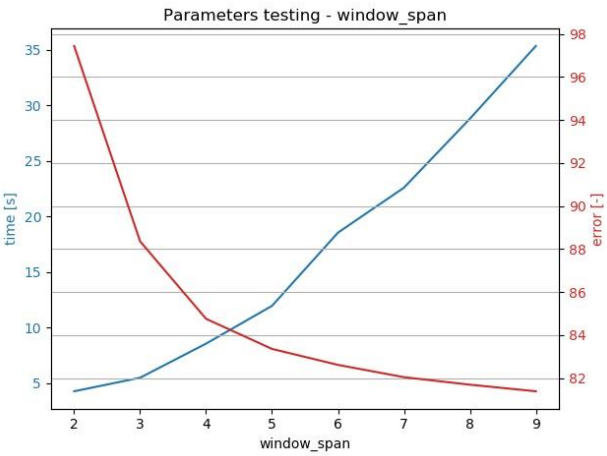

Figure 68 **Influence of window span in inverse – results after fix**

At first, the very interesting and seemingly counterintuitive situation could be found here – with the increase of window span we saw both the simulation time and error margin to rise. Also, interestingly, they were rising almost at the very same, linear pace.

Explaining the rising simulation time seemed easy – if we need to match more windows into the future, it will take longer.

It was not straightforward to reason the increase in error margin. It could have been explained by overfitting[29] – 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.

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.

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**.

Tolerance

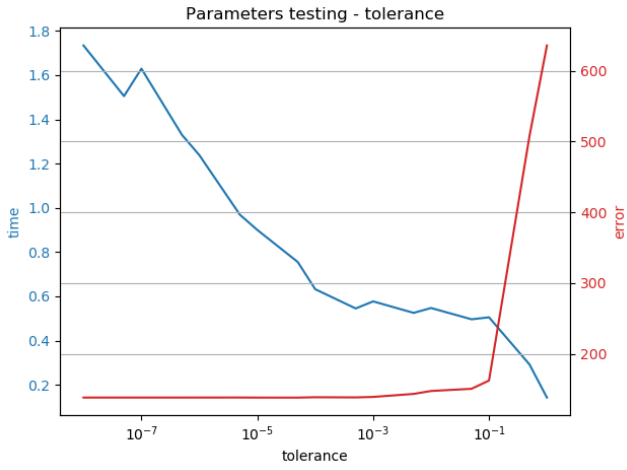

Figure 71 **Influence of tolerance in inverse – whole range**

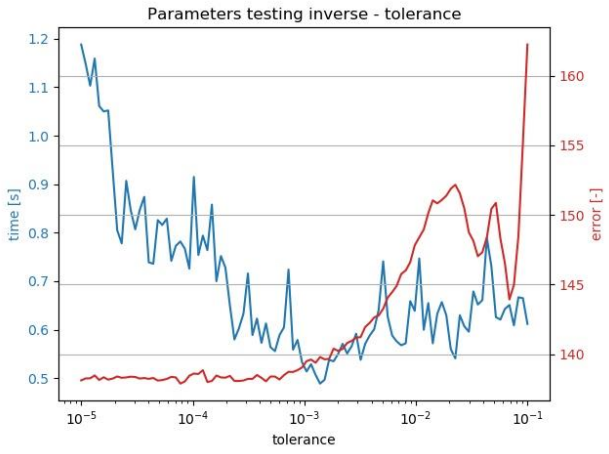

Figure 70 **Influence of tolerance in inverse – optimal range**

[29] https://en.wikipedia.org/wiki/Overfitting

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

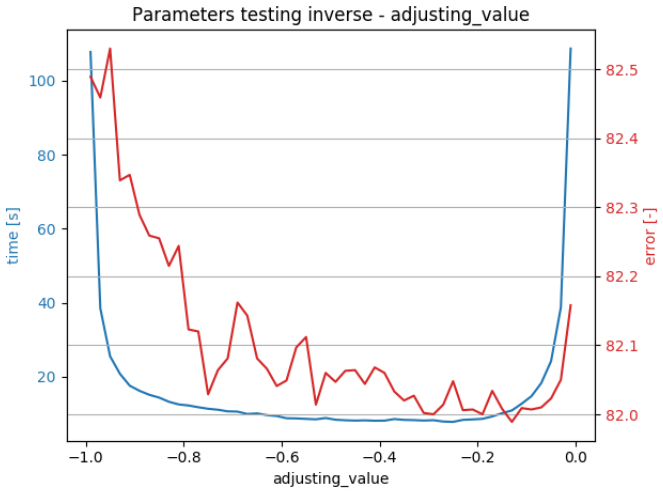

Figure 72 **Influence of adjusting value in inverse**

The dependence nicely shows how lengthy the simulation is when we choose the adjusting value close to the boundaries of possible values (-1 and 0).

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.

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.

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

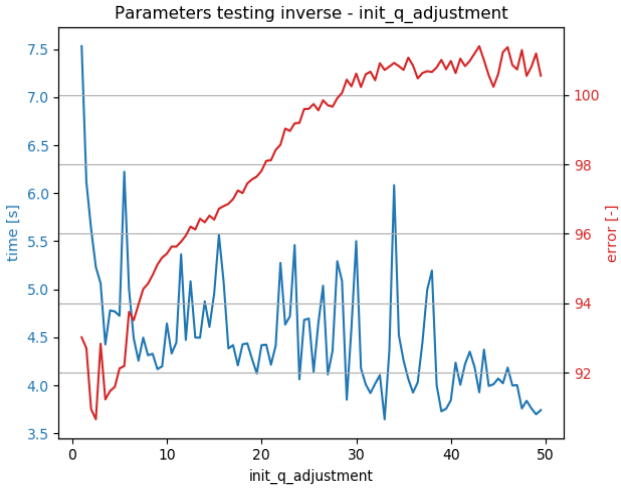

Figure 73 **Influence of initial heat flux adjustment – dt = 3 s**

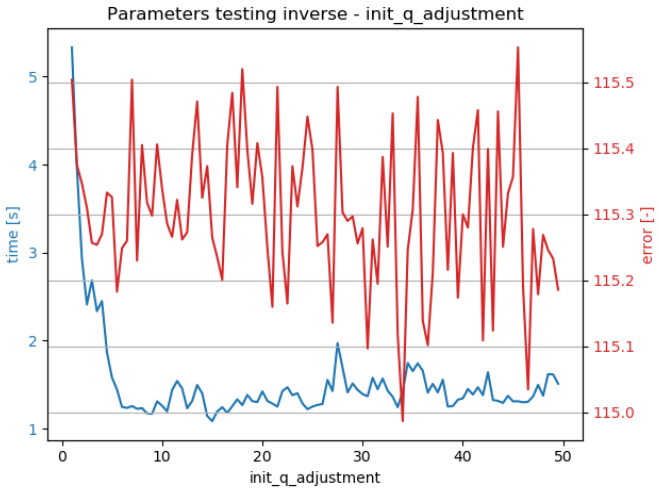

Figure 74 **Influence of initial heat flux adjustment – dt = 30 s**

In the first experiment with a timestep of 3 seconds, it is apparent that higher initial adjustment speeds up the simulation but at the cost of the higher error margin.

The second experiment with timestep being equal to 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 number of steps needed to traverse bigger differences in heat flux when the adjustment steps are lower.

Discussion: It was shown that one of the factors influencing the dependency is the timestep. Generally, in our case, the initial adjusting value around **10-15 W** can be suggested.

7.3.3 Testing observations and conclusions

Apart from all the results and diagrams above, we found out that in the inverse problem, the lower dt (time step) decreases the number of iterations needed to locate appropriate heat flux. 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.

Logarithmic axes could be used to visualise the results better when showing very diverse data.

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.)

Automatic optimal point determination was done by *parameters_testing_optimum_finding.py* script. It analyses the tested parameters and returns 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, apart from hard numbers, is the graphical representation of the result. In the case of higher dt 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 and sacrifice the precision on paper to get a more realistic result.

8 Possible improvements for the future

As software creation is an endless and iterative process, there is always something that could be improved. This short chapter describes some of these improving steps and points out some problems with the current state.

However, these suggestions can be subjective, and the best feedback and suggestions for new features always come from the customers of the software, who really use it on a regular basis.

- The whole GUI could be converted to PySide2 instead of PyQt5, so it could be used without any restrictions (as commercial use of PyQt5 is monetized).
- Currently, the choice of data file gets reset to its default value when changing the algorithm. It should be remembering the user's choice and preserve it even when the algorithm is changed.
- Tooltips providing an explanation of all user actions could be created on buttons and labels, as a way of user help. Also having the whole window dedicated for a user manual could be beneficial.
- Support for Excel files could be created instead of only accepting CSV files with the experimental data. The parsing logic would stay almost the same.
- GUI styling could be more addressed because some of the elements are not looking really well – longer input labels are not fully visible, on a small window size the Time label on graphs is not fully visible, etc.
- Smoothing panel could be given its own window so that it is more visible than appearing in the left panel, where it could be overlooked.
- The simulation could be made quicker by giving the task of graph plotting to its own thread. The simulation thread would not have to deal with the plotting, it would only output the simulation results for example to a file, where it would be picked by plotting thread.
- The logic of finding the right heat flux in the inverse simulation could utilize different methods of converging into the right solutions, one possibility could be the halving of the intervals.
- The whole inverse approach and the parameters testing could be tried on different data sources, as the results would be then more representative.
- Smoothing the data with the moving average with the window length being even $(2, 4\ldots)$ causes the data to shift to the right side, which is most probably undesirable. This should be researched more thoroughly and in case it is manipulating the results rather than smoothing them, the restriction of the smoothing window length being odd should be implemented, the same way as is being done in SavGol filter.

9 Conclusion

The goal of this thesis was to create a software tool for simulating heat transfer and implementing the inverse approach. At the beginning, it laid out the basic theory behind inverse problems and heat transfer. The main takeaway of these chapters is that the inverse problems are starting to play a very important role in the engineering process, and its importance will probably only increase in the future, with more widespread usage of computers, which are crucial for computationally demanding inverse calculations.

The main part of the thesis, dealing with software implementation, started with careful planning. We were analyzing the requirements needed to complete this task and came up with three possible programming languages (Python, C++ and Matlab), from which Python was used in the end. The main reasoning behind this choice was Python's ease of development and the abundance of powerful libraries. This choice turned out quite well and can be a proof that engineering problems are not only a domain of Matlab, which requires an expensive license. Even these free and open-source development platforms like Python can offer everything necessary for fully-fledged engineering simulation.

The whole process of software planning and creation is filled with architectural and other various diagrams. This, together with the thorough description of all software parts, increases the understandability of the solution and creates a high-level overview of the code. The whole codebase of the project is then included in the attachment, together with the link to its online public repository, where the project is stored.

The software creation part, together with the explanation of programming concepts used there, can be used as a guide of basic software development in engineering. It shows examples of using software tools and methodologies like version control or testing, that can greatly improve the quality and reliability of the created software.

The resulting software has the form of a GUI connected with the heat transfer simulation. Its architecture is highly modular, so the heat transfer simulation can be easily replaced by any other engineering simulation. This creates the opportunity of reusing the code in other projects.

In the heat transfer simulation, we developed a very lightweight approach of solving the inverse problem, depending only on the knowledge of the forward solution and basic logic. The results were then compared with the data from the real experiment, showing that the precision of this approach is good enough, with the average error value of only 4 %.

After creating a reliable heat transfer simulation for both the forward problem and the inverse problem, we decided to test the influence of all the various input parameters to the resulting simulation. The results of this testing include the visual representation of the relationship between the parameters and the quality of the simulation, as well as the recommendation about the optimal values of these parameters.

At the end, we were also outlining some possibilities on how to further increase the reliability and usability of the software, which can be useful if this software would be used or built upon in the future.

List of used symbols and units

Symbol	Quantity	Unit
ρ	Density	kg/m^3
c_p	Specific heat capacity	J/kgK
λ	Heat conductivity	W/mK
α	Convective heat transfer coefficient	W/m^2K
L, x	Length	cm, m
x_0	Position of interest	cm, m
t	Time	s
Q	Heat flux	W
q	Heat flux density	W/m^2
T	Temperature	°C, K
T_s	Ambient temperature	°C, K
θ	Implicitness/explicitness coefficient	-

List of figures

List of used literature

[1] OZISIK, Necati. *Inverse Heat Transfer: Fundamentals and Applications*. CRC Press, 2000. ISBN 978-1560328384.

[2] TABRIZI, Ardeshir a Yogesh JALURIA. *Solution of an inverse problem to determine heat source strength and location* [online]. 2017 [cit. 2020-04-04]. Available at: https://www.researchgate.net/publication/325221371_SOLUTION_OF_AN_INVERSE_PROBLEM_TO_DETERMINE_HEAT_SOURCE_STRENGTH_AND_LOCATION

[3] TABRIZI, Ardeshir a Yogesh JALURIA. *An optimization strategy for the inverse solution of a convection heat transfer problem* [online]. 2018 [cit. 2020-04-04]. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0017931017344617?via%3Dihub#ab010

[4] Well-posed problem. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-04-04]. Available at: https://en.wikipedia.org/wiki/Well-posed_problem

[5] PAVELEK, Milan. *Termomechanika*. Brno: Akademické nakladatelství CERM, 2011. ISBN 978-80-214-4300-6.

[6] JÍCHA, Miroslav. *Přenos tepla a látky*. Brno: Akademické nakladatelství CERM, 2001. ISBN 80-214-2029-4.

[7] BERGMAN, Theodore a Frank INCROPERA. *Fundamentals of heat and mass transfer*. 7th. Hoboken, NJ: Wiley, 2011. ISBN 978-0-470-50197-9.

[8] NELLIS, Gregory a Sanford KLEIN. *Heat transfer*. New York: Cambridge University Press, 2009. ISBN 978-0521881074.

[9] *Python vs. C++: Let's Compare* [online]. [cit. 2020-04-04]. Available at: https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/

[10] *MATLAB vs Python: Why and How to Make the Switch* [online]. [cit. 2020-04-04]. Available at: https://realpython.com/matlab-vs-python/

[11] *MATLAB vs. Python: Top Reasons to Choose MATLAB* [online]. [cit. 2020-04-04]. Available at: https://www.mathworks.com/products/matlab/matlab-vs-python.html

[12] *Tkinter — Python interface to Tcl/Tk* [online]. [cit. 2020-04-04]. Available at: https://docs.python.org/3.8/library/tkinter.html

[13] *Qt Designer Manual* [online]. [cit. 2020-04-04]. Available at: https://doc.qt.io/qt-5/qtdesigner-manual.html

[14] *PyQt5 vs PySide2: What's the difference between the two Python Qt libraries?* [online]. [cit. 2020-04-04]. Available at: https://www.learnpyqt.com/blog/pyqt5-vs-pyside2/

[15] *Tkinter understanding mainloop* [online]. [cit. 2020-04-04]. Available at: https://stackoverflow.com/questions/29158220/tkinter-understanding-mainloop

[16] *Multithreading PyQt applications with QThreadPool* [online]. [cit. 2020-04-04]. Available at: https://www.learnpyqt.com/courses/concurrent-execution/multithreading-pyqt-applications-qthreadpool/

[17] *Profiling and optimizing Python code* [online]. [cit. 2020-04-04]. Available at: https://osf.io/upav8/

[18] *The Rules of Optimization: Why So Many Performance Efforts Fail* [online]. [cit. 2020-04-04]. Available at: https://hackernoon.com/the-rules-of-optimization-why-so-many-performance-efforts-fail-cf06aad89099

[19] *How do I use line_profiler (from Robert Kern)?* [online]. [cit. 2020-04-04]. Available at: https://stackoverflow.com/questions/23885147/how-do-i-use-line-profiler-from-robert-kern

[20] *Python Classes and Objects* [online]. [cit. 2020-04-04]. Available at: https://www.w3schools.com/python/python_classes.asp

[21] *Method overriding in Python* [online]. [cit. 2020-04-04]. Available at: https://www.thedigitalcatonline.com/blog/2014/05/19/method-overriding-in-python/

[22] *What is version control* [online]. [cit. 2020-04-04]. Available at: https://www.atlassian.com/git/tutorials/what-is-version-control

[23] *Python Design Patterns: For Sleek And Fashionable Code* [online]. [cit. 2020-04-04]. Available at: https://www.toptal.com/python/python-design-patterns

[24] *The Factory Method Pattern and Its Implementation in Python* [online]. [cit. 2020-04-04]. Available at: https://realpython.com/factory-method-python/

[25] *Git Hooks* [online]. [cit. 2020-04-04]. Available at: https://www.atlassian.com/git/tutorials/git-hooks

[26] *What is software testing?* [online]. [cit. 2020-04-04]. Available at: https://www.atlassian.com/continuous-delivery/software-testing

[27] *The Meaning of Underscores in Python* [online]. [cit. 2020-04-04]. Available at: https://dbader.org/blog/meaning-of-underscores-in-python

[28] *Python Type Checking (Guide)* [online]. [cit. 2020-04-04]. Available at: https://realpython.com/python-type-checking/

[29] *Python range() function explained with examples* [online]. [cit. 2020-04-04]. Available at: https://pynative.com/python-range-function/

[30] *Numpy.linspace* [online]. [cit. 2020-04-04]. Available at: https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linspace.html

[31] *Numpy.logspace* [online]. [cit. 2020-04-04]. Available at: https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.logspace.html

[32] *Numpy.arange* [online]. [cit. 2020-04-04]. Available at: https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.arange.html

[33] *How to smooth a curve in the right way?* [online]. [cit. 2020-04-04]. Available at: https://stackoverflow.com/questions/20618804/how-to-smooth-a-curve-in-the-right-way

[34] Savitzky–Golay filter. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-04-04]. Available at: https://en.wikipedia.org/wiki/Savitzky%E2%80%93Golay_filter

[35] ŠNAJDÁREK, Ladislav. *Přestup tepla v kanálech malých průřezů s rotující stěnou*. Brno, 2019. Dizertační práce. Vysoké učení technické v Brně.

[36] *What is the difference between the implicit and explicit formulation in heat transfer analysis?* [online]. [cit. 2020-04-05]. Available at: https://www.researchgate.net/post/What_is_the_difference_between_the_implicit_and_explicit_formulation_in_heat_transfer_analysis

[37] Crank–Nicolson method. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-04-05]. Available at: https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method

[38] *How the Apollo Spacecraft Worked* [online]. [cit. 2020-04-10]. Available at: https://science.howstuffworks.com/apollo-spacecraft7.htm

[39] *What's the Difference Between Conduction, Convection, and Radiation?* [online]. [cit. 2020-04-10]. Available at: https://www.machinedesign.com/learning-resources/whats-the-difference-between/document/21834474/whats-the-difference-between-conduction-convection-and-radiation

[40] *Heat transfer by radiation* [online]. [cit. 2020-04-10]. Available at: https://www.eschooltoday.com/energy/kinds-of-energy/what-is-radiation.html

[41] *The Python Logo* [online]. [cit. 2020-04-10]. Available at: https://www.python.org/community/logos/

[42] C++. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-04-10]. Available at: https://en.wikipedia.org/wiki/C%2B%2B

[43] *Mathworks* [online]. [cit. 2020-04-10]. Available at: https://www.mathworks.com/

[44] LOGG, Anders, Kent-Andre MARDAL a Garth N. WELLS. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1.

[45] Consultations with Ing. Libor Kudela and his own shared documents.

[46] *Finite Elements in 1D, MIT course, Prof. Gilbert Strang* [online]. [cit. 2020-05-10]. Available at https://ocw.mit.edu/courses/mathematics/18-085-computational-science-and-engineering-i-fall-2008/video-lectures/lecture-18-finite-elements-in-1d-part-2/

List of attachments

[1] InverseHeatTransfer.zip

- Archive including the latest version of the application code as for 31. 5. 2020.

[2] https://github.com/grdddj/Diploma-Thesis---Inverse-Heat-Transfer

- Link to the github repository, where the whole incremental project history can be viewed. Future changes, if any, will be still done in this repository. Everybody is also free to contribute to this codebase via pull requests.