Unstructured Finite Element Solver

Na Young Ahn, Manoj Cendrollu Nagesh, Mehmet Velioglu, Zhao Wu and Loic Wendling

German Research School for Simulation Sciences GmbH
Chair for Computational Analysis of Technical Systems, RWTH Aachen

Abstract: This project aims to develop and implement a finite element solver for heat diffusion problems. We mainly use FEniCS, an intuitive python, and C++ based library to translate the weak formulation of heat equation into an efficient finite element code. First, we verify the solver on simple meshes with known analytical solutions. We then extend to the real-life application by exploring the 3D piston case with various boundary conditions and see how it compares with the simplified 2D problem. Similar temperature distribution of the 2D and 3D piston infers the potential to simulate a simplified geometry as accurately as the full-scaled model but with significantly lower computational time and cost. We further optimize the solution by adapting the mesh and time-step sizes and parallelize the solver by running on the cluster. Running the parallel solver on multiple processors does not always produce better results. Problem size, communication costs, and parallel efficiency should be taken into account. Lastly, we apply algorithmic differentiation to perform parameter sensitivity analysis for simple and complex cases. Parameter estimation analysis with the algorithmic differentiation shows the importance of choosing the optimal step size α to find the balance between runtime and convergence. A possible approach would be to change the step size α for every iteration. Future studies may be done to better understand how to regulate the step size and to prevent overshooting.

1 Introduction

1.1 Motivation

Heat transfer problems arise in all fields of science and engineering. The study of heat transfer not only helps us understand the natural environment but also provides economical and efficient solutions for critical problems encountered in many engineering items of equipment[1], such as degradation of material properties due to overheating. Such real-world industrial problems, however, can be challenging due to complex geometries and lack of exact closed-form solutions. Therefore, we rely on numerical techniques - finite difference, finite volume, and finite element method - to approximate solutions.
The objective of this project is to develop a parallel finite element solver on an unstructured mesh. The finite element method allows us to arbitrarily form the elements, hence we can get a close representation of the boundaries of complicated domains. Therefore we rely on the finite element method for its flexibility dealing with irregular geometries and some unusual specification of boundary conditions. The solver will be developed, implemented and optimized with the help of FEniCS, a C++/Python library that provides an automated programming environment for differential equations.[2]
We first verify the solver on simple meshes with analytical solutions by looking at 2D steadystate and transient problems. After optimizing the serial solver by adapting the mesh and time-step sizes, we expand to the complex 3D piston case with various boundary conditions. We further parallelize the solver by running on a shared memory environment. Lastly, we apply algorithmic differentiation to perform parameter sensitivity analysis for simple and complex cases.[3]

1.2 Theoretical Background - Finite Element Method

Before we investigate different cases of the project, a brief overview of the mathematical foundation will be given in this section. In this project, the main focus is to solve the unsteady heat equation

with the following strong form[3]:

$$\frac{\partial T}{\partial t} - \alpha \nabla^2 T = 0 \quad in\ \Omega \tag{1}$$

where α is the thermal diffusivity and T is the temperature distribution. Ω is the domain where the equation is solved. In some cases, the steady equation is solved with the first term on the left-hand side dropped. Throughout the project, there is no heat source used, thus the right-hand side is equal to zero. Below, the general boundary conditions are given:

$$\begin{aligned} T &= T_D \quad on\ \Gamma_D \\ -k\nabla T &= q \quad on\ \Gamma_N \end{aligned} \tag{2}$$

where T_D is a prescribed temperature value on the boundary, q is a prescribed heat flux and k is the thermal conductivity. Here the domain boundary is split into two general cases. Γ_D is the Dirichlet boundary and Γ_N is the Neumann boundary. However, in the project, different cases have different boundary conditions: some only with Dirichlet and some with both Dirichlet and Neumann boundary conditions. Also, each type of boundary condition might be multiple, namely with different temperatures and heat flux values for different boundaries. Next, a relation between the thermal conductivity and thermal diffusivity is given, which is important for the integration of the Neumann boundary condition for the weak form[4].

$$\alpha = \frac{k}{c_p \rho} \tag{3}$$

c_p denotes the specific heat capacity and ρ denotes the density of the material. The time discretization is done with backward-Euler method[2]:

$$\frac{\partial T}{\partial t} = \frac{T^{n+1} - T^n}{\Delta t} + O(\Delta t) \tag{4}$$

This transforms the strong form as follows:

$$T^{n+1} = T^n + \Delta t\,(\alpha \nabla^2 T) \tag{5}$$

After time discretization, the equation can be converted into the weak form. First, a trial function and test function space should be defined:

$$\begin{aligned} \mathcal{S} &:= \{T \in \mathcal{H}^1(\Omega) \mid T = T_D \quad on\ \Gamma_D\} \\ \mathcal{V} &:= \{w \in \mathcal{H}^1(\Omega) \mid w = 0 \quad on\ \Gamma_D\} \end{aligned} \tag{6}$$

where $\mathcal{S}$ is the trial function space for functions T and $\mathcal{V}$ is the test functions space for functions w. Leaving out $T = T_D$ on the boundary, this function spaces lead to the Galerkin form, since the spaces $\mathcal{S}$ and $\mathcal{V}$ are equivalent except on the boundary. The equation (5) is then multiplied with the test function w and integrated over the domain Ω. Lastly, integration by parts is done on the second term of equation (5) to reach the following weak form[3]:

$$\int_\Omega w T^{n+1} d\Omega - \int_\Omega w T^n d\Omega + \int_\Omega \Delta t(\alpha \nabla w \cdot \nabla T^{n+1}) d\Omega - \int_{\Gamma_N} \Delta t \frac{\alpha}{k} (\hat{n} \cdot q) w\, d\Gamma = 0 \tag{7}$$

which then concludes the theoretical background part. This variational form can be used directly in FEniCS without the integrals. No further manipulations to reach the matrix form or defining shape functions are needed in FEniCS.

1.3 Software Requirements

1.3.1 FEniCS

FEniCS is a C++/Python library, initiated in 2003 in Chicago and licensed under the GNU LGPL, that provides an automated programming environment for differential equations using the finite element method. Generation of basis functions, evaluation of variational forms, finite element assembly, and adaptive error control are automated. FEniCS distinguishes itself from other automated finite element solvers as it provides intuitive and compact code that is very close to the mathematical formulation despite increased mathematical and algorithmic complexity when running on a high-performance compute server.[2]
However, given FEniCs is a complex software library with many dependencies to other scientific software libraries, we use the Docker containers, that not only works on all operating systems but also saves us from the hassle of building and configuring the software. FEniCS may also be installed using other methods, including Conda packages and building from source.

1.3.2 Workflow

We mainly use FEniCS for our finite element solvers. But having to deal with complex geometries, we enhance upon the existing technology by importing geometry, meshes, element connectivity from GMSH, a meshing software. It comes with both a GUI and a scripting language interface.
For Postprocessing, we use Paraview, a multi-platform data visualization application. It can not only produce publication-quality figures, provide sophisticated data analysis functions but also can handle very large (and parallelized) data sets better than VTK and Matplotlib. The following data file formats are used by FEniCS: ParaView files (.pvd), VTK files (.vtu), XDMF files (.xdmf).[2]

1.3.3 dolfin-adjoint with FEniCS

Dolfin-adjoint project can automatically derive the discrete adjoint and tangent linear models from a forward model written in the Python interface to FEniCS. These adjoint and tangent linear models are key ingredients in many important algorithms, such as data assimilation, optimal control, sensitivity analysis, design optimization, and error estimation. Such models have made an enormous impact in fields such as meteorology and oceanography, but their use in other scientific fields has been hampered by the great practical difficulty of their derivation and implementation. Also, the automatic generation of optimal adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing.[5]

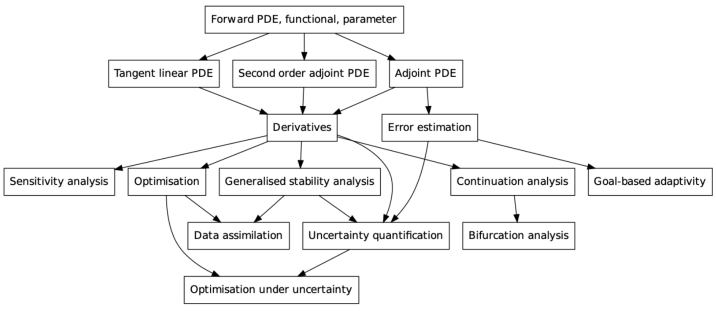


Figure 1: Structure of FEniCS

Features Dolfin-adjoint works for both steady and time-dependent problems and for both linear and nonlinear problems. And it is very easy to use, just giving a differentiable forward model, employing dolfin-adjoint involved. The adjoint and tangent linear models of dolfin-adjoint present optimal theoretical efficiency. When storing every forward variable, the adjoint takes 0.2-1.0 times the runtime

of the forward model, depending on the precise details of the structure of the forward problem. The adjoint and tangent linear models can also run in parallel with no modification. When the adjoint model is instructed, it can automatically employ optimal check pointing schemes to mitigate storage requirements for long nonlinear runs. Rigorous verification routines are provided in dolfin-adjoint, so that users can easily verify for themselves the correctness of the derived models. Also, dolfin-adjoint solves optimization problems constrained by partial differential equations by interfacing with powerful optimization algorithms.

Limitations Dofin-adjoint works only with the Python interface of FEniCS and Firedrake. The discretization of the problem must be differentiable to make the adjoint be consistent. All changes to object values (matrices, vectors, functions) must happen through the FEniCS/Firedrake interface, though custom operations can be recorded manually.

Advantages compared to traditional algorithms The traditional approach to deriving adjoint and tangent linear models is called algorithmic differentiation. The fundamental idea of algorithmic differentiation is to treat the model as a sequence of elementary instructions. Elementary instruction is a simple operation such as addition, multiplication, or exponentiation. Each one of these operations is differentiated individually, and the derivative of the whole model is then composed of the chain rule. The dolfin-adjoint project is instead based on a very different approach. The model is considered as a graph of high-level operations. This abstraction is similar to the fundamental abstraction of algorithmic differentiation but operates at a much higher level of abstraction. This idea is implemented in a software library, pyadjoint. When this new idea is combined with the high-level abstraction of the FEniCS system, many of the difficult problems associated with algorithmic differentiation dissolve.

2 Case 1 - 2D Square

We begin with a time dependent heat equation with dirichlet boundary conditions on a uniform mesh. By discretizing time into small time intervals and applying standard time-stepping methods, we can solve the heat equation by solving a sequence of variational problems.

Our model problem for time-dependent PDEs reads

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla^2 u + f \qquad && \text{in } \Omega \times (0,T], \\ u &= u_{\text{D}} && \text{on } \partial\Omega \times (0,T], \\ u &= u_0 && \text{at } t = 0\,. \end{aligned} \tag{8}$$

Here, u varies with space and time, e.g., $u = u(x, y, t)$ if the spatial domain Ω is two-dimensional. The source function f and the boundary values u_{D} may also vary with space and time. The initial condition u_0 is a function of space only.

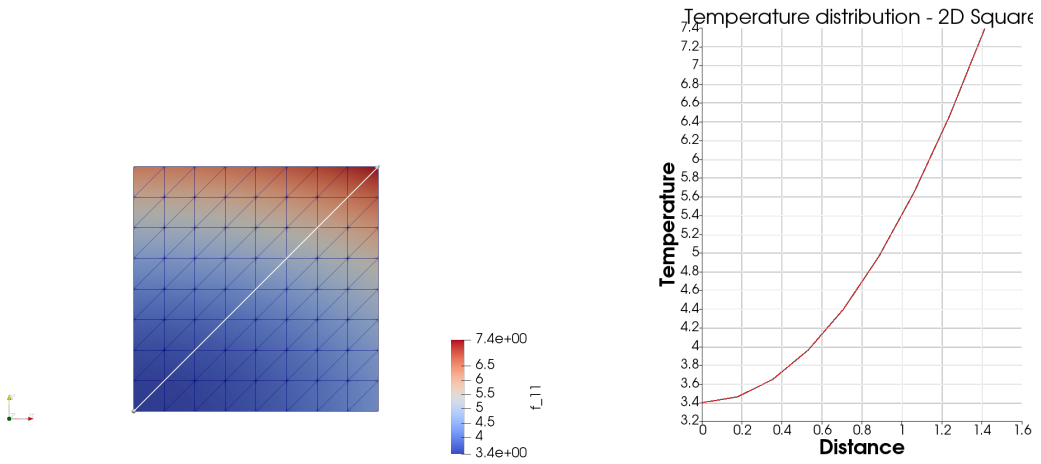
We then choose a test problem validate the calculations. Since we know that our first-order time-stepping scheme is exact for linear functions, we create a test problem which has a linear variation in time. We combine this with a quadratic variation in space. We thus take

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{9}$$

which yields a function whose computed values at the nodes will be exact, regardless of the size of the elements and Δt, as long as the mesh is uniformly partitioned. By inserting (9) into the heat equation (8), we find that the right-hand side f must be given by $f(x, y, t) = \beta - 2 - 2\alpha$. The boundary value is $u_{\text{D}}(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$ and the initial value is $u_0(x, y) = 1 + x^2 + \alpha y^2$.[2]

2.1 Result and Analysis

The simulation was first run on a 8x8 uniform mesh with 128 elements and 81 nodes for $T = 2s$ with 20 time steps, which gives time step size $\Delta t = 0.1$. The uniform temperature distribution can be seen in the Fig.2(a). The results are in very good agreement with the analytical solution.



(a) Contour plot **(b)** Line plot

Figure 2: 2D Square - Results

3 Case 2 - Cylinder

Before we use the solver in any practical heat transfer application, it is necessary to verify its accuracy, consistency, and stability. This can be accomplished by validating the diffusion equation solver developed in the previous case using the data available in the literature and a few mesh convergence studies. Once the solver is validated, we can further use it to solve heat transfer in more complex scenarios. In forthcoming sections, we will validate the solver for two generic boundary conditions that we encounter in most of the problems.

We start with the case of heat conduction between concentric cylinders with simple dirichlet boundary conditions on inner and outer surfaces. The setup and the applied temperature conditions are shown in Fig.3(a). The discretized domain with a refinement level of $0.05m$, 3572 elements and 1864 nodes is shown in Fig.3(b). The solution for this problem reduces to a one-dimensional solution with respect to the radius r and can be computed analytically. The exact solution for temperature distribution is given as[6]

$$c_{cyl}^{exact}(r(x)) = c_i - (\ln r - \ln r_i)\frac{c_i - c_a}{\ln r_a - \ln r_i} \tag{10}$$

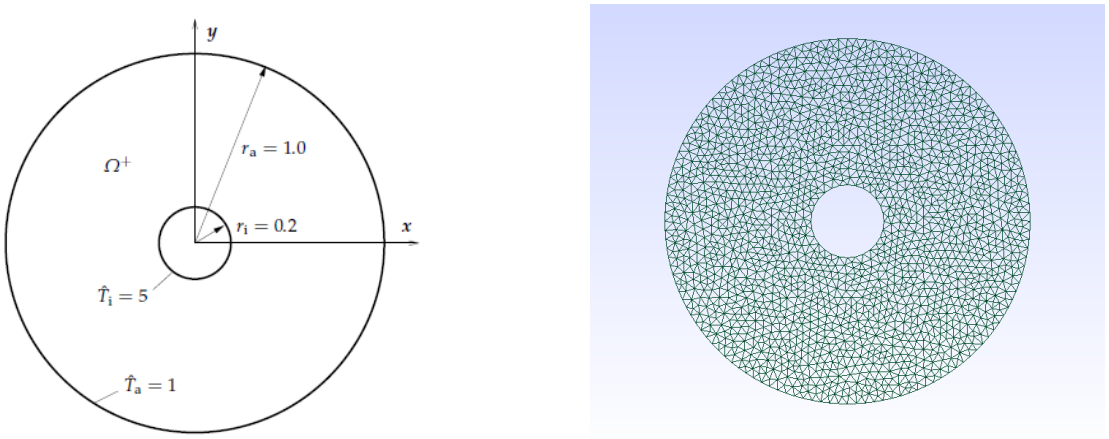
The two cylinder surfaces are aligned with the z-axis, hence the cylinder radius is defined as

$$r(x) = \sqrt{x^2 + y^2} \tag{11}$$

The boundary conditions used are shown in the following table 1

Boundary ID	Comment	Type	Value	Unit
T_a	Outer	Dirichlet	1	°C
T_i	Inner	Dirichlet	5	°C

Table 1: Boundary conditions

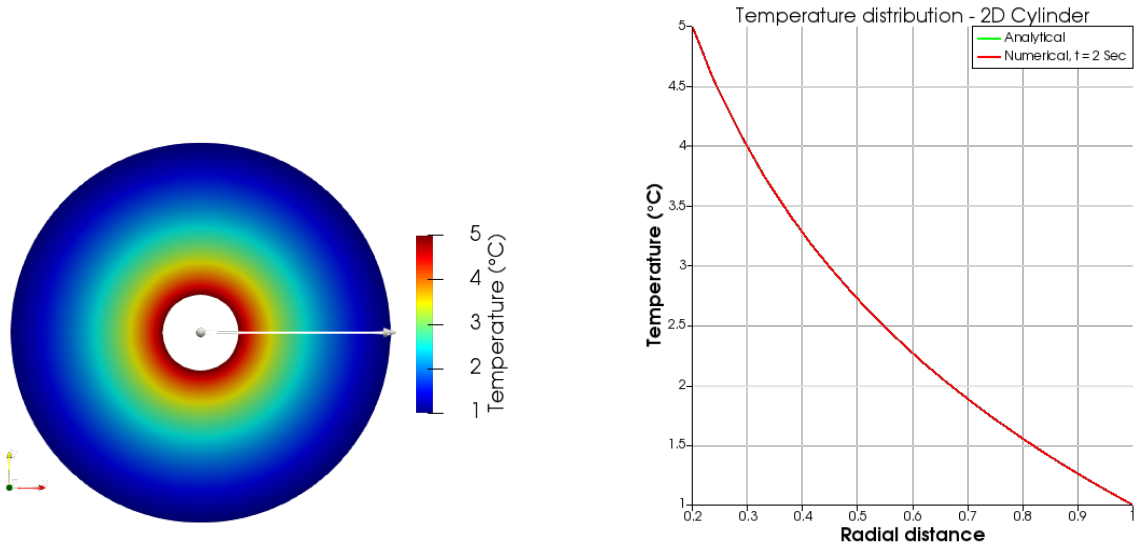


(a) 2D Setup **(b)** Meshed domain

Figure 3: 2D Cylinder

3.1 Result and Analysis

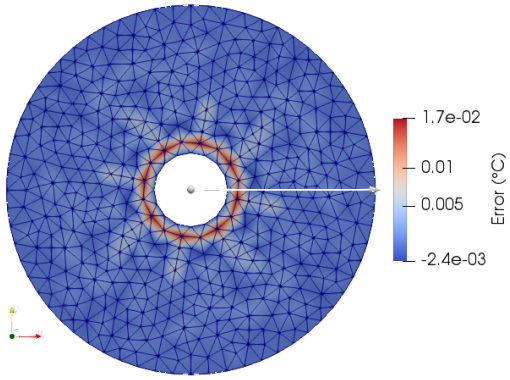
The simulation was run until the steady-state was reached and no further changes in temperatures were noticed. The uniform temperature distribution could be seen in Fig.4(a). The results are in very good agreement with the analytical solution. An overplot of both the solutions along the radial direction can be seen in Fig.4(b).



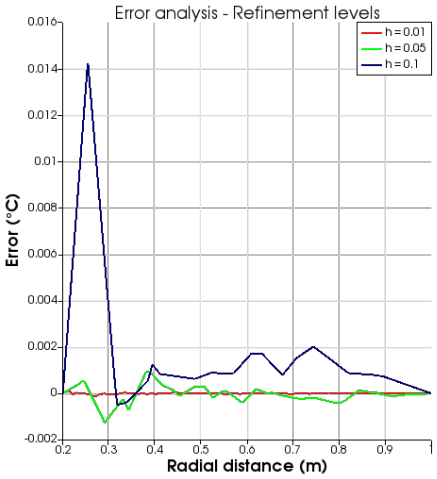
(a) Contour plot **(b)** Line plot

Figure 4: 2D Cylinder - Results

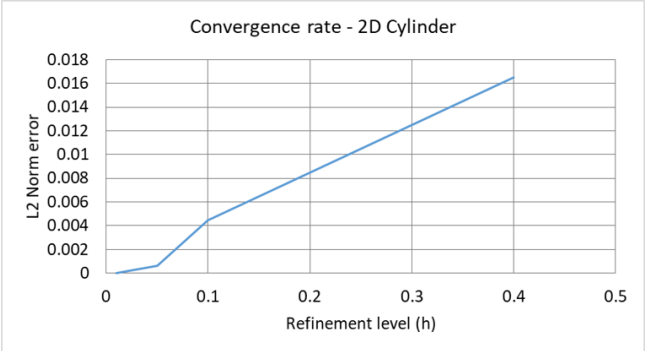
A mesh refinement study was done to see how the error scales with finer meshes. The L2 norm of the error is plotted over the entire domain as shown in Fig.5(a). We see a higher error close to the boundary due to a higher temperature gradient. A line plot of the error along the radial direction for different refinement levels is shown in Fig.5(b). The convergence rate is shown in Fig.5(c)



(a) Error contour plot



(b) Error line plot

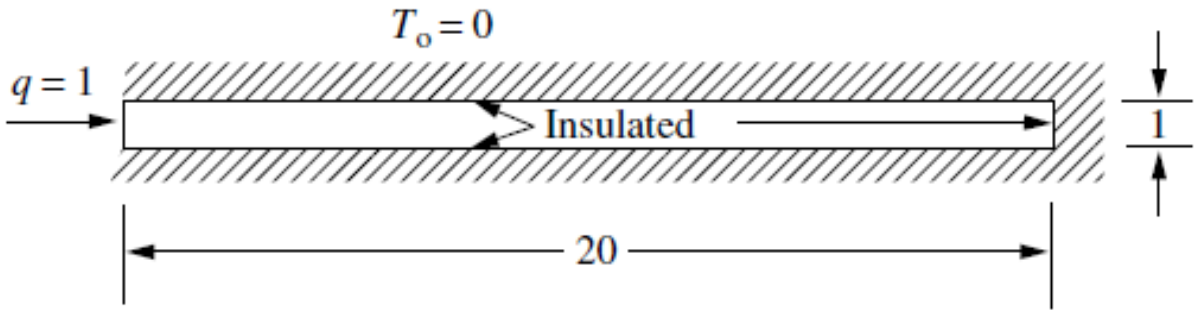


(c) Convergence

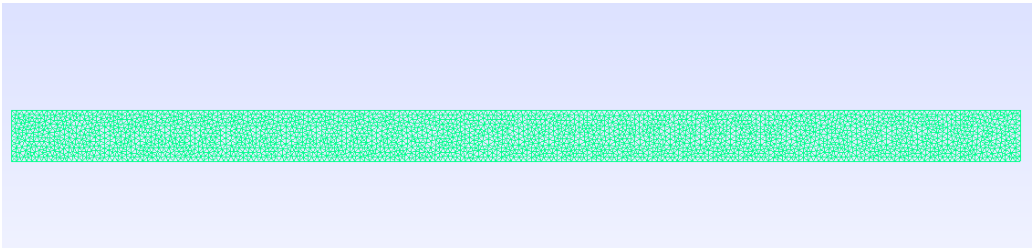
Figure 5: Mesh refinement study

4 Case 3 - Rod

In this validation case, we consider an unsteady configuration which also includes the Neumann boundary condition. Here the left side of the domain is subjected to constant heat flux and all other walls are insulated (heat flux is zero) as shown in Fig.6(a). A refinement level of ($h = 0.1m$) was used to mesh the physical domain with 5220 elements and 2821 Nodes as shown in Fig.6(b).



(a) 2D setup



(b) Meshed domain

Figure 6: 2D Rod

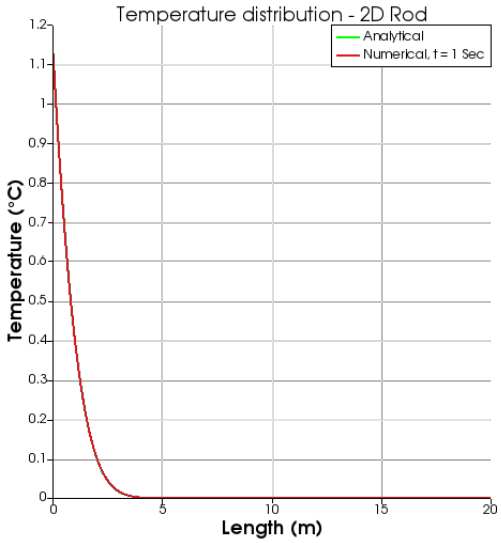
The corresponding exact temperature distribution is given as a function of time and x coordinate as [1]

$$T(x,t) = 2(t/\pi)^{(1/2)} \left[\exp\left(-x^2/4t\right) - (1/2)x\sqrt{\frac{\pi}{t}}erfc\left(\frac{x}{2\sqrt{t}}\right)\right] \tag{12}$$

4.1 Result and Analysis

The temperature distribution along the rod at time $t = 1$sec is shown in the Fig.7(a) along with the line plot in Fig.7(b).

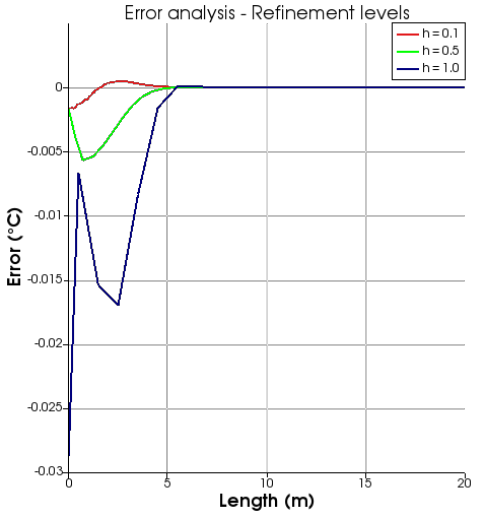
(a) Contour plot



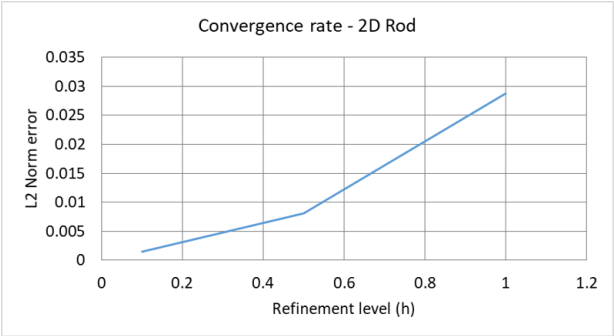
(b) Line plot

Figure 7: 2D Rod - Results

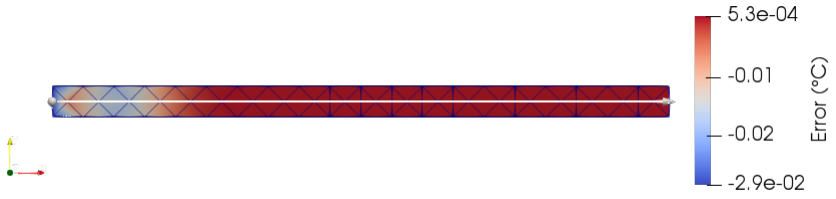
A mesh convergence study similar to the previous case was done using different refinement levels. The error distribution over the domain and the convergence rates are shown in the following figures.



(a) Error line plot



(b) Convergence



(c) Error contour plot

Figure 8: Mesh refinement study

5 Case 4 - Piston

After validating the solver in the previous test cases, a piston will be studied to understand the challenges of the design phase of an extremely important industrial component, the internal combustion engine. The studied piston will be a reciprocating piston and it should be designed for durability and performance. In real-life applications, pistons are designed such that it avoids overheating and shows satisfactory cooling performance. The piston in our case will be a simplified piston, 2D and 3D simulations will be made to decide which one is more feasible to use in our study. Overall temperature distribution and the accuracy of our study through mesh discretization will be observed and discussed in the upcoming part. Moreover, a further study will be done in order to observe how the thickness of the head of the piston affects temperature distribution.
The material used in this piston is assumed to be a special aluminum alloy. Tabulated values below show the assumed characteristic constants of this metal. These values are assumed average values for aluminum alloys with the advisor of the project and are not values from a specific aluminum alloy which are just used to obtain reasonable, comparable results.

Thermal Diffusivity	α	$5.4 \times 10^5 \ m^2/s$
Density	ρ	$2710 \ kg/m^3$
Specific Heat	c_p	$920 \ J/kgK$
Thermal Conductivity	k	$134.6 \ W/mK$

Table 2: Material properties of the assumed aluminum alloy[7]

Three different subcases will be investigated for the piston simulation, both for 2D and 3D. Each case differs by the specification of the boundary condition of the top surface. The first subcase prescribes a fixed temperature on the top boundary, the second subcase is constant heat flux on the top boundary and the third one is a time-varying heat flux, reflecting the power cycle of a piston. The boundary conditions for each subcase is tabulated below[8]. Moreover, the figures of the geometries used are given to show the boundaries clearly.

Figure 9: Piston Geometry

Boundary ID	Comment	Type	Value	Unit
0	Top	Dirichlet	300	$^\circ C$
1	Liner	Dirichlet	100	$^\circ C$
2	Bottom	Dirichlet	30	$^\circ C$
3 (for 3D)	Top	Neumann	0	MW/m^2

Table 3: Boundary conditions for subcase 1[8]

Boundary ID	Comment	Type	Value	Unit
0	Top	Neumann	2	MW/m^2
1	Liner	Dirichlet	100	$^\circ C$
2	Bottom	Dirichlet	30	$^\circ C$
3 (for 3D)	Top	Neumann	0	MW/m^2

Table 4: Boundary conditions for subcase 2[8]

Boundary ID	Comment	Type	Value	Unit
0	Top	Neumann	$\dot{q}$	MW/m^2
1	Liner	Dirichlet	100	$^\circ C$
2	Bottom	Dirichlet	30	$^\circ C$
3 (for 3D)	Top	Neumann	0	MW/m^2

Table 5: Boundary conditions for subcase 3[8]

The third subcase seeks to mimic a piston in a four stroke engine, which has the following phases[8]:

1. 0° to 180°: Intake
2. 180° to 360°: Compression
3. 360° to 540°: Power
4. 540° to 720°: Exhaust

Here an engine which is rotating at 4000 RPM is used. A complete cycle of the crankshaft is 0.06 seconds for a 720° cycle. Then the $\dot{q}$ in Table 5 is given as[8]:

$$\dot{q} = \frac{4 \times 10^6}{((\alpha - 550)^2 + 1 \times 10^3)^2} \tag{13}$$

where α is the crankshaft angle. The plot of the function for a piston cycle is given below:

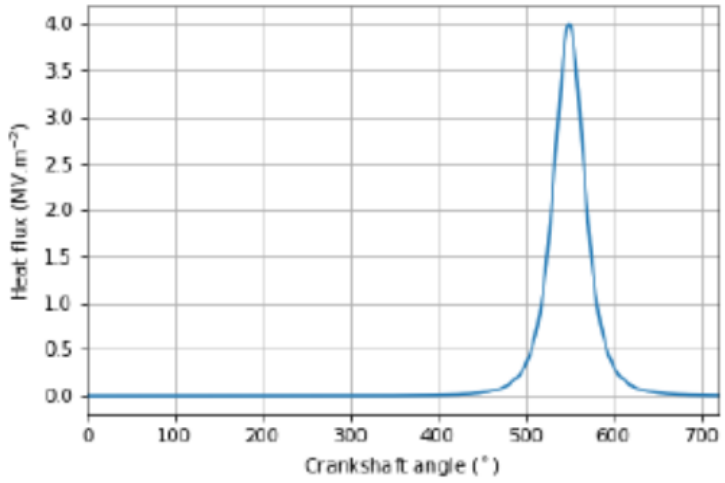


Figure 10: Heat flux for each engine cycle[8]

As a final note, all the following simulations are done with a time-step $dt = 0.005\,s$. This time-discretization is chosen since it produces reasonable results for all subcases presented. After representing all the conditions for the simulations, the results can be discussed next.

5.1 2D Piston

In this part, the 2D simulation of the piston will be discussed. Throughout the discussion, temperature distribution will be presented on a vertical line on the piston, as shown below. For all runs and subcases, the initial temperature of the domain is taken as 50 °C and the time-step size is taken to be $dt = 0.005\,s$.

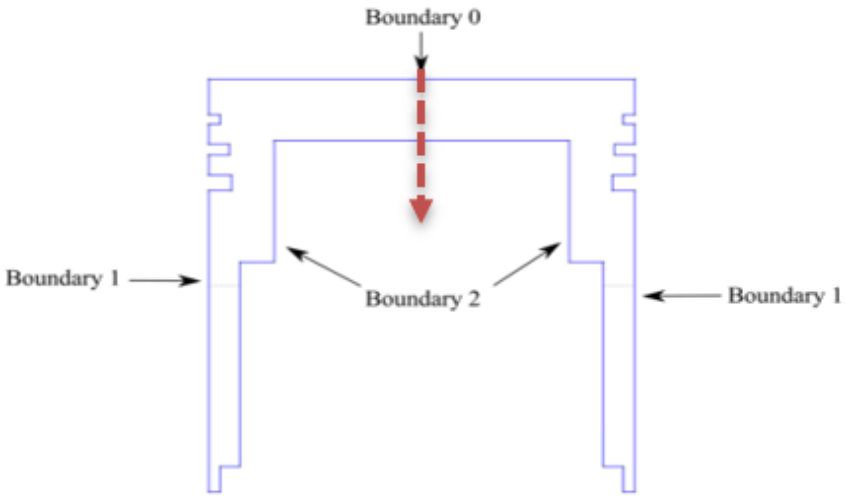


Figure 11: 2D Piston with the vertical line for observing temperature distribution

The meshing of the geometry is done with the software *GMSH*. An example mesh with element size h = 1 mm with 2810 elements and 5126 nodes is shown below.

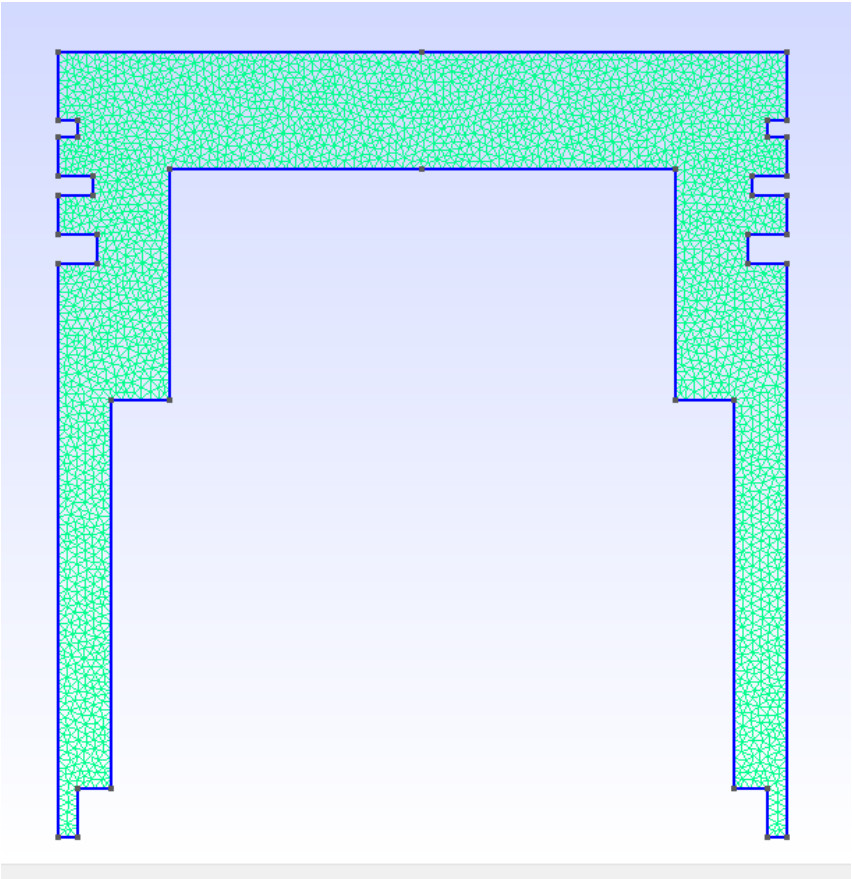
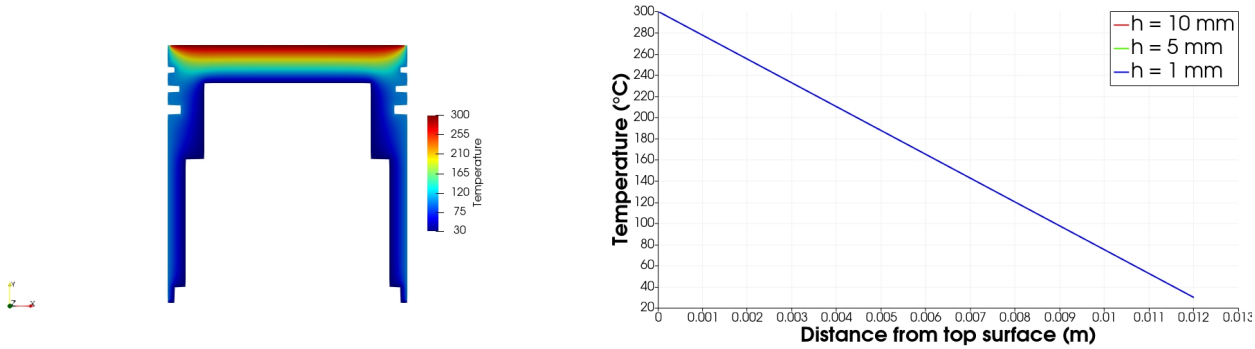


Figure 12: 2D Piston - Meshed domain

5.1.1 Subcase 1 - Dirichlet Boundary Condition on Top Boundary

The boundary conditions for subcase 1 was given in Table 3. Time-step dt = 0.005 s was used in the simulations. In the figures below, the temperature distribution at T = 2 s and the mesh convergence

study plots are given. The temperature distribution on the vertical line shown at the beginning of the section is used to study mesh convergence. Element sizes 10 mm, 5 mm and 1 mm are used for the mesh for this case.



(a) Temperature Distribution, h = 1 mm, T = 2 s **(b)** Mesh Convergence Study, T = 2 s

Figure 13: Subcase 1 - Results

As observed in the mesh convergence plot, the different mesh sizes are over-plotted, meaning that these levels of discretization don't produce differing results for this case. Thus, the mesh is converged and any size of the elements above is suitable for an accurate result. This also validates the consistency of the solver. However, the result of the finest mesh is used to plot temperature distribution, since it is fairly cheap to compute and will provide a better overview when the different subcases are compared. For the temperature distribution, the top surface dominates, since it is locked with a Dirichlet boundary condition. A gradual transition to lower temperatures happens from top to bottom.

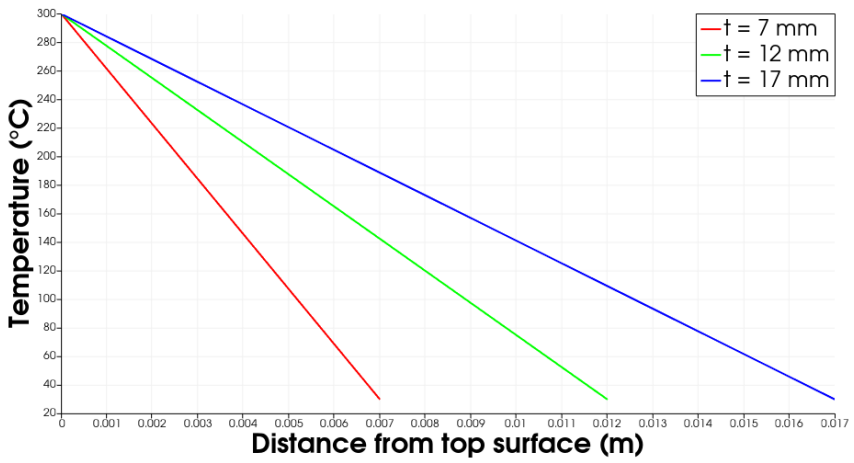
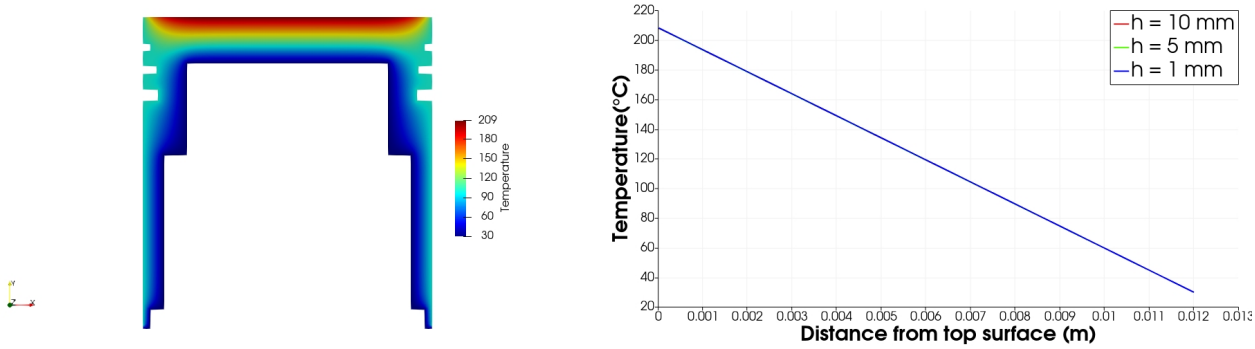


Figure 14: Subcase 1 - Thickness study

When the thickness study is observed, where the thickness depicted is the head of the piston (t - in the plot) it is observed that the thicker piston becomes, overall temperature increases. This is a physically unrealistic solution and the reason is that all boundaries are locked with a Dirichlet boundary condition. The temperature gradually decreases from a fixed boundary temperature on the top to the bottom. Since the heat flux is equal to the temperature gradient the Dirichlet boundary conditions cause an increases supply of heat flow to the thicker piston, as can be seen from the slopes, which is unrealistic since thickness has nothing to do with the heat flux supplied.

5.1.2 Subcase 2 - Constant Neumann Boundary Condition on Top Boundary

The boundary conditions for subcase 2 was given in Table 4. Time-step dt = 0.005 s was used in the simulations. In the figures below, the temperature distribution at T = 8 s and the mesh convergence study plots are given. The temperature distribution on the vertical line shown at the beginning of the section is used to study mesh convergence. Element sizes 10 mm, 5 mm and 1 mm are used for the mesh for this case.



(a) Temperature Distribution, h = 1 mm, T = 8 s **(b)** Mesh Convergence Study, T = 8 s

Figure 15: Subcase 2 - Results

As observed in the mesh convergence plot, again the different mesh sizes are over-plotted, meaning that these levels of discretization don't produce differing results for this case. However, the result of the finest mesh is used to plot temperature distribution, since it is fairly cheap to compute and will provide a better overview when the different subcases are compared. For the temperature distribution, the top surface dominates, since the bottom boundary is locked with a Dirichlet boundary condition and constant heat flux is supplied from the top which is enough to increase the temperature there to a value higher than the side walls.

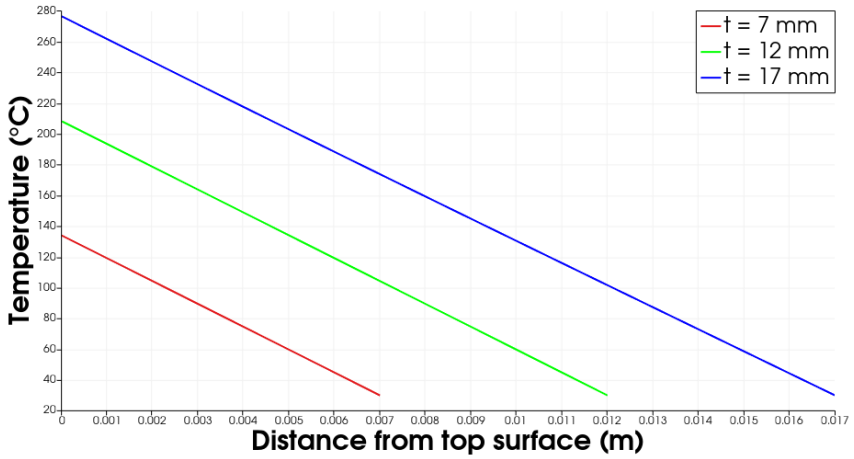


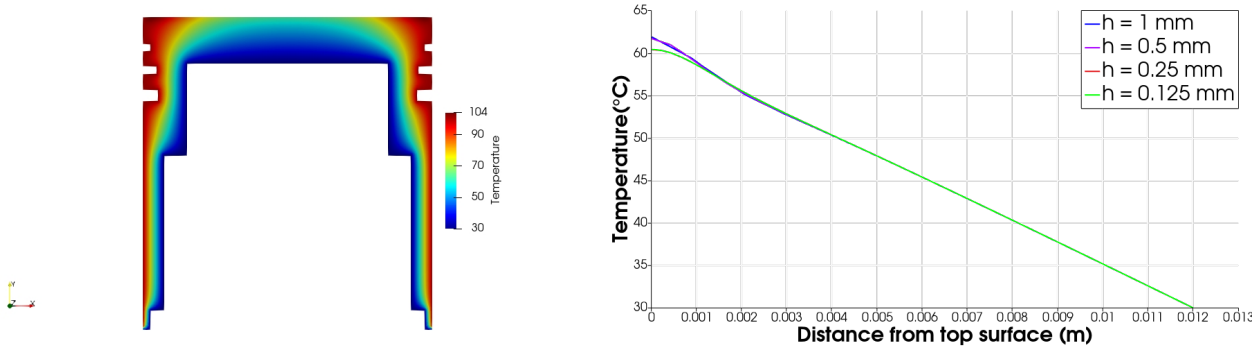
Figure 16: Subcase 2 - Thickness study

When the thickness study is observed, where the thickness depicted is the head of the piston (t - in the plot) it is observed that the thicker piston becomes, again the overall temperature increases. The constant Neumann boundary condition on the top means a constant heat flux, which means a constant temperature gradient. Moreover, the bottom boundary is at a fixed temperature of 30°. Thus, for the thicker piston, the overall temperature increases to satisfy this constant heat flux value, given that the bottom boundary temperature is fixed. This is again an unrealistic solution, and the bottom Dirichlet

boundary condition is the issue here. This setup of the simulation does not involve any heat loss through the boundaries, which causes the problem.

5.1.3 Subcase 3 - Time-Varying Neumann Boundary Condition on Top Boundary

The boundary conditions for subcase 3 was given in Table 5. Time-step dt = 0.005 s was used in the simulations. In the figures below, the temperature distribution at T = 6 s and the mesh convergence study plots are given. The temperature distribution on the vertical line shown at the beginning of the section is used to study mesh convergence. Element sizes 1 mm, 0.5 mm, 0.25 mm and 0.125 mm are used for the mesh for this case.



(a) Temperature Distribution, h = 0.25 mm, T = 6 s **(b)** Mesh Convergence Study

Figure 17: Subcase 3 - Results, T = 6 s

Here a difference in solutions can be observed when decreasing the mesh size. This means that the coarser meshes are not enough to accurately simulate this case. This is largely due to the time-varying heat flux making the problem more complex. However, as shown in the plot, the element size of 0.25 mm and 0.125 mm cause over-plotted solutions, which means that the mesh convergence study is successful and the element size 0.25 mm can be used as an accurate result. The contour plot of this mesh size is presented above and since the heat flux over a cycle is not as high as the constant heat flux case, the side walls generally have a higher temperature compared to the top boundary.

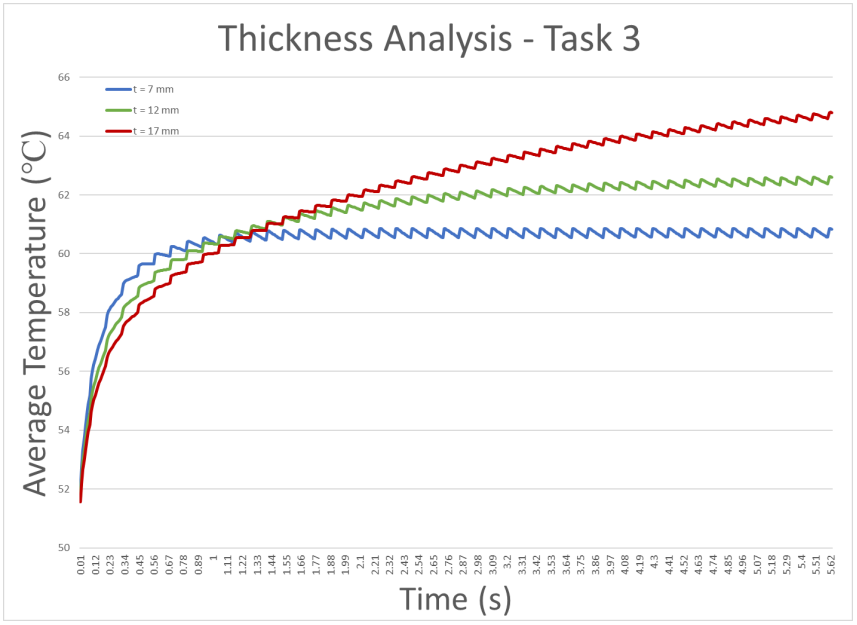
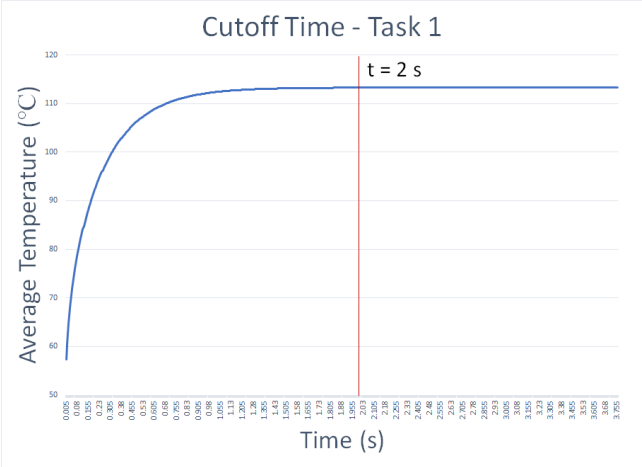


Figure 18: Subcase 3 - Thickness study

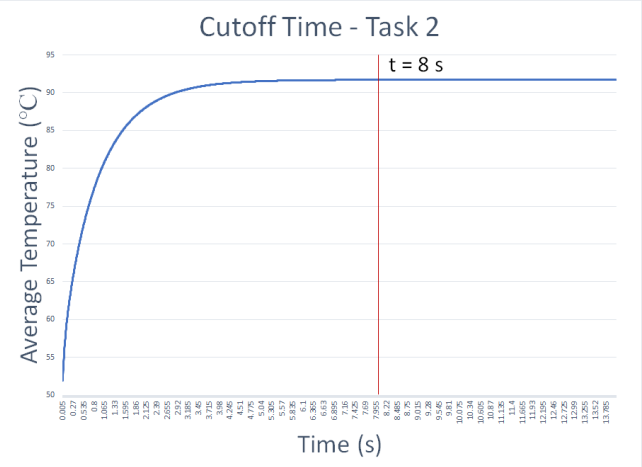
This time thickness study is done showing average temperatures. Since a constant total value of heat flux is supplied for each cycle, the result when plotting the thickness study wouldn't be any different than in subcase 2, if the temperature distribution on the vertical line is used, except the slopes. Again the thicker piston has a higher overall temperature in this case, because of the defect explained in the previous section for subcase 2. The Dirichlet boundary conditions of the other walls should be changed to allow the piston to give off heat to attain more realistic results. Here the results of the thicker piston have not yet converged, which actually attains a converged value in the simulation but is not shown here to have a better comparison. Also, this shows that the simulation with a thicker piston needs more time to converge.

5.1.4 Summary of 2D Piston

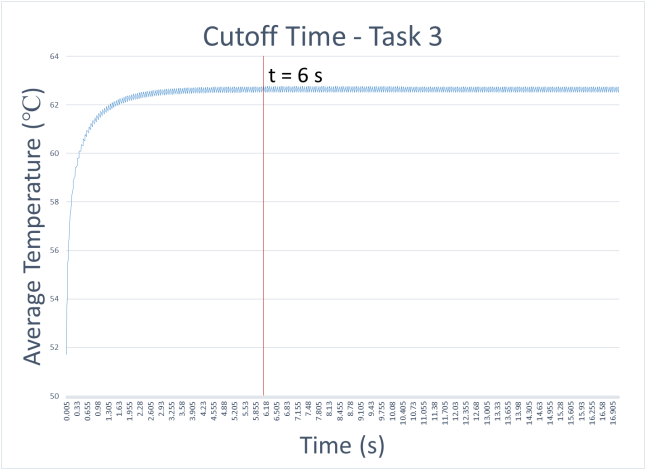
In the previous sections, the results were shown at different times for each case. The reason for this is that the solutions reach convergence after some point and to simulate them further would be computational time loss without any gains. Thus, each simulation is cut at a certain time, T = 2 s, T = 8 s, T = 6 s for subcases 1, 2 and 3 respectively. The reasoning for the choice of these times is given in the plots below, where at each time stamp the respective simulation reaches a stable, converged solution. Moreover, these times do not change when the mesh sizes are changed.



(a) Subcase 1



(b) Subcase 2



(c) Subcase 3

Figure 19: 2D Piston - Cutoff Times

Lastly, the maximum, average and minimum temperatures for each run done is given below, as requested.

Element Size	**Max. T** ($^{\circ}C$)	**Avg. T** ($^{\circ}C$)	**Min. T** ($^{\circ}C$)
h = 10 mm	300.000	108.992	30.000
h = 5 mm	300.000	111.831	30.000
h = 1 mm	300.000	113.251	30.000

Table 6: Average, Maximum and Minimum Temperatures for Subcase 1

Element Size	**Max. T** ($^{\circ}C$)	**Avg. T** ($^{\circ}C$)	**Min. T** ($^{\circ}C$)
h = 10 mm	208.197	89.718	30.000
h = 5 mm	208.233	91.111	30.000
h = 1 mm	208.780	91.695	30.000

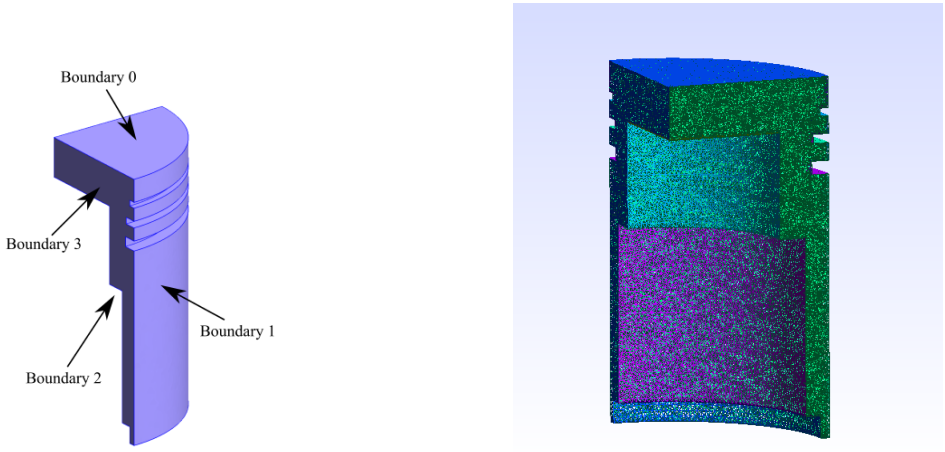
Table 7: Average, Maximum and Minimum Temperatures for Subcase 2

Element Size	**Max. T** ($^{\circ}C$)	**Avg. T** ($^{\circ}C$)	**Min. T** ($^{\circ}C$)
h = 1 mm	102.324	62.700	30.000
h = 0.5 mm	102.332	62.729	30.000
h = 0.25 mm	102.323	62.729	30.000

Table 8: Average, Maximum and Minimum Temperatures for Subcase 3

5.2 3D Piston

Having established the accuracy and the stability of the heat equation solver using the previous test cases, we move on to the practical applications. A rigorous mesh refinement study on 2D piston showed promising results with discretization being the only source of error that goes down with finer meshes. A heat diffusion study on a 2D piston would suffice in the case of uniform heat distribution over the piston surface. But that is never really the case in an actual engine cycle. Rapid combustion would often lead to a non-uniform heat flux which highlights the necessity to study the heat diffusion in a fully-scaled 3D piston.
In the following subcases we would consider a sector of the full-scale 3D piston as shown in Fig.20(a). The domain was meshed using a refinement level of ($h = 1mm$) with 166681 elements and 35977 nodes as shown in Fig.20(b). The heat diffusion equation was solved for the same boundary conditions as in the 2D case. The results were compared at similar final times and the temperature distribution was found to be very much similar in both the cases due to the uniformity in boundary conditions.

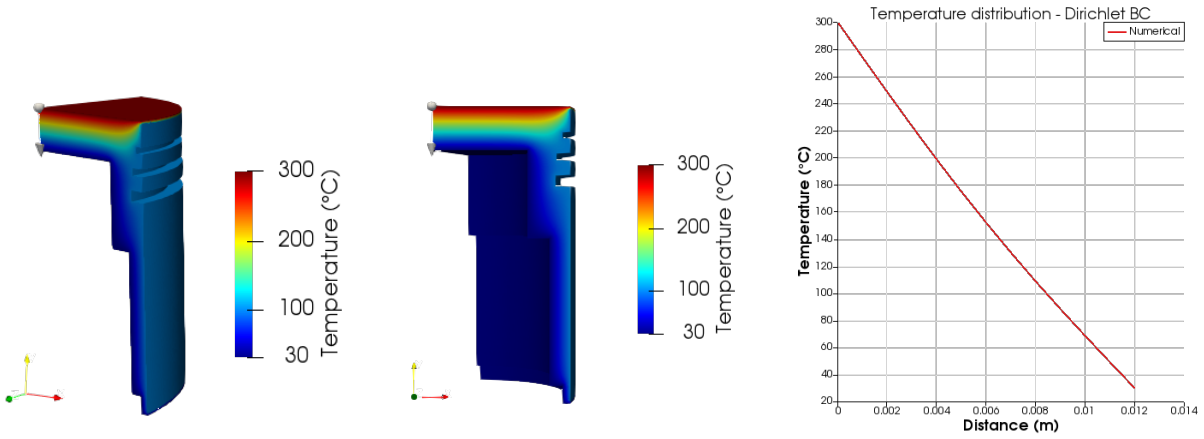


(a) 3D piston **(b)** Meshed domain

Figure 20: 3D Piston

5.2.1 Subcase 1 - Dirichlet Boundary Condition on Top Boundary

In this subcase we assign constant temperatures on all the faces of the piston similar to the 2D case as in the table 3. The temperature distribution over the cross-section is shown in the Figures below along with the line plot at the center of the piston.

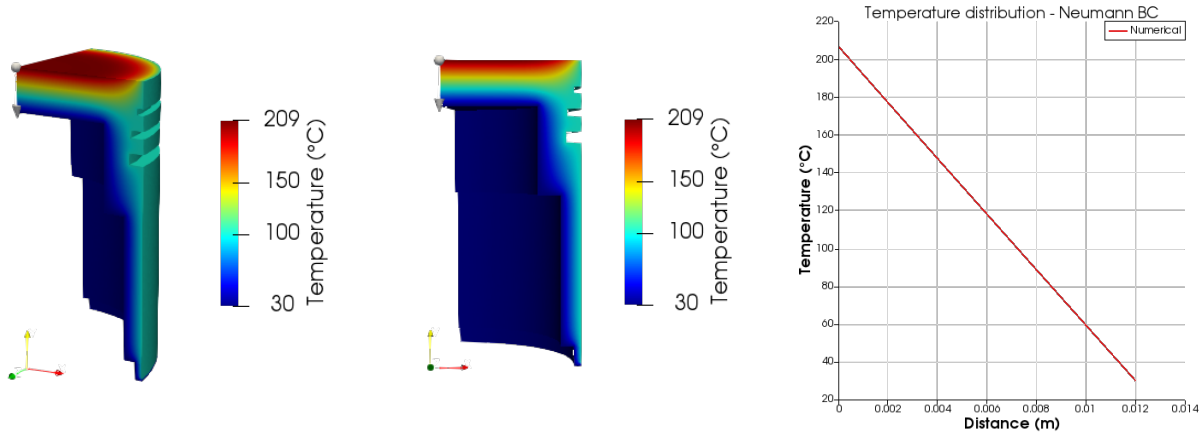


(a) 3D contour **(b)** Cross-section view **(c)** Line plot along the piston centre

Figure 21: 3D Piston - Dirichlet BC results

5.2.2 Subcase 2 - Constant Neumann Boundary Condition on Top Boundary

The constant temperature boundary condition on the top surface (boundary 0) of the piston was changed to a constant heat flux of $2MW/m^2$ as given in the table 4. This resulted in a lower peak temperature on the piston surface with the overall distribution as shown in the figures below.

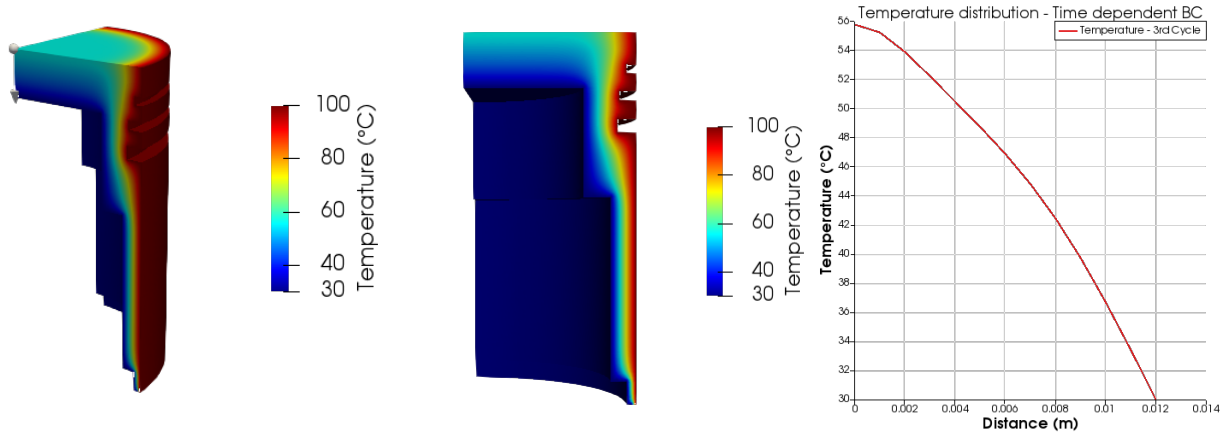


(a) 3D contour **(b)** Cross-section view **(c)** Line plot along the piston centre

Figure 22: 3D Piston - Neumann BC results

5.2.3 Subcase 3 - Time-Varying Neumann Boundary Condition on Top Boundary

In this case, the constant heat flux has been replaced with the heat release rate corresponding to an actual engine cycle. The time-varying heat flux is shown in Fig.10 of 2D Piston section. Since the peak flux lasts only a few milliseconds in each cycle, we see a much lower peak temperature on the piston surface. The temperature distribution during the third cycle is shown in the figures below.



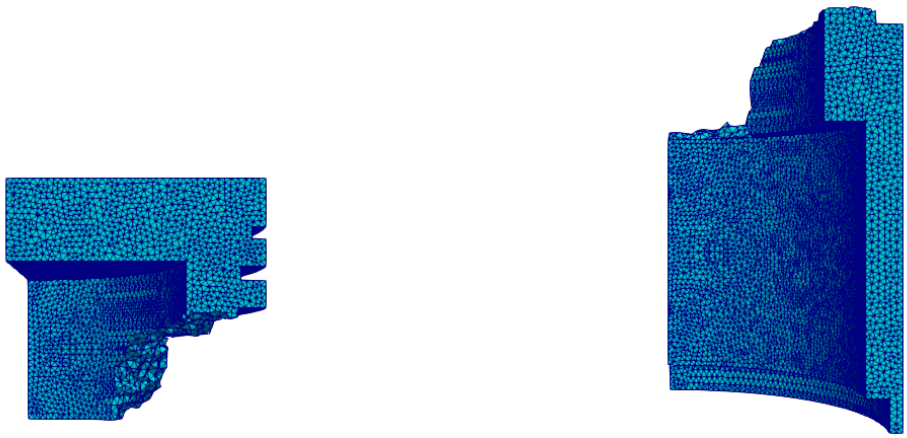
(a) 3D contour **(b)** Cross-section view **(c)** Line plot along the piston centre

Figure 23: 3D Piston - Time varying Neumann BC results

The temperature in all the above subcases was found to be in very good agreement with the 2D case.

5.3 Parallelization - Message Passing Interface (MPI)

The serial computing does not serve the purpose as we move on to bigger physical domains which would result in higher mesh count. With MPI offering us the possibility to split the tasks among multiple processors, we can lower the computation time significantly. Naive use of a higher number of processors could lead to communication overkill resulting in higher computation times. With a higher mesh count, the 3D piston was a suitable case for parallelization. The partition of the mesh domain in the case of two processors is shown in Fig.24. The computation timings, speedup and the efficiencies for different processors are tabulated below.



(a) Mesh partition 1 **(b)** Mesh partition 2

Figure 24: Domain decomposition for 2 processors

No. of processors	Time(Sec)	Speedup	Parallel efficiency
Serial	285	-	-
2	192	1.48	0.74
3	157	1.815	0.60
4	138	2.06	0.51
8	121	2.35	0.29
16	109	2.61	0.16

Table 9: MPI timings

5.3.1 Summary of 3D Piston

The temperature distribution along the section planes in all the above subcases was found to be very similar to that of the 2D piston. No significant differences were noticed in the results with the mesh refinement. As a result, mesh convergence study was just limited to the 2D piston.
An important takeaway from this study is to consider the simplification of complex 3D problems down to lower dimensions with appropriate boundary conditions and still achieve the same accuracy in results with lower computational time and cost.

6 Case 5 - Algorithmic Differentiation

6.1 Problem description

This case is based on the results of case 1 and case 4. The goal of case 5 is to reverse engineer a solution field to find the parameter, in our case the diffusion coefficient (ν), used in the original solution using Algorithmic Differentiation(AD). This is an inverse problem for parameter estimation.

6.1.1 Steps

First, we generate a reference solution using an arbitrary diffusion coefficient and save the reference value. We then load in the reference solution previously generated and check that the loaded one is correct. Afterwards, we define an initial diffusion coefficient and step size (α) and solve the temperature field for the current diffusion coefficient.

To minimize the function: $J(u) = \int_\Omega < u_{red}(T) - u(T), u_{red}(T) - u(T) > d\Omega$, the derivative of the Jacobian $\frac{dJ}{d\nu}$ has to be calculated using a build-in function in fenics-adjoint. We update the diffusion coefficient: $\nu = \nu - \alpha\frac{dJ}{d\nu}$ and loop until the coefficient matches the original one.

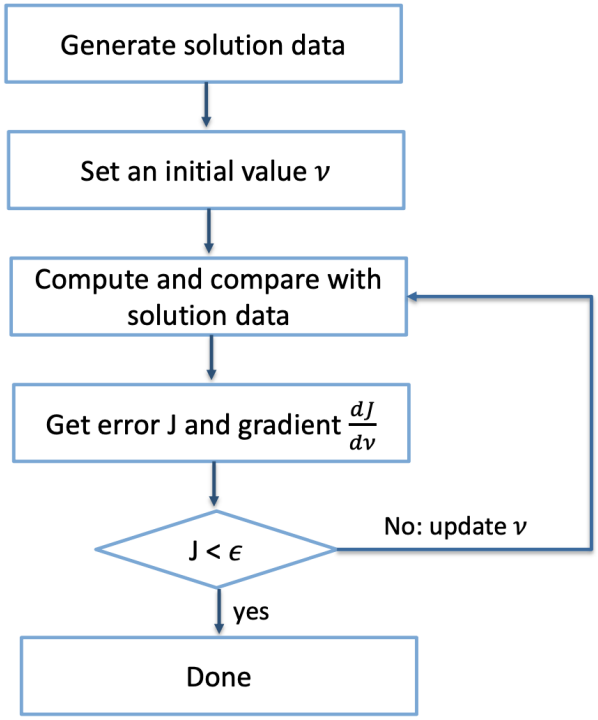


Figure 25: AD flow chart

6.1.2 Tasks

We should have a working code for a simple coarse rectangular test case and confirm that the resulting reverse engineered diffusion coefficient corresponds to the original one. Then we find the parameter sensitivity analysis of the initial value (ν_0), step size (α) and mesh size.
Our code needs to be confirmed also for the 2D piston in case 4. Parameter sensitivity test of the initial value (ν_0) and step size (α) is also applied to this case.

6.2 Results and analysis for the 2D Square

After running the code for different initial values, step size and mesh size, we obtained the following results.

6.2.1 Confirmation and initial value

As shown in Figure 26, the resulting reverse engineered diffusion coefficient corresponds to the original one. We choose two initial values separately from both sides to exact value (two bigger than exact ν and the other two smaller than exact ν), and they all converge to exact value. Meanwhile, it is obvious that if the initial value is closer to exact value, its results will be more accurate with the same number of iteration.

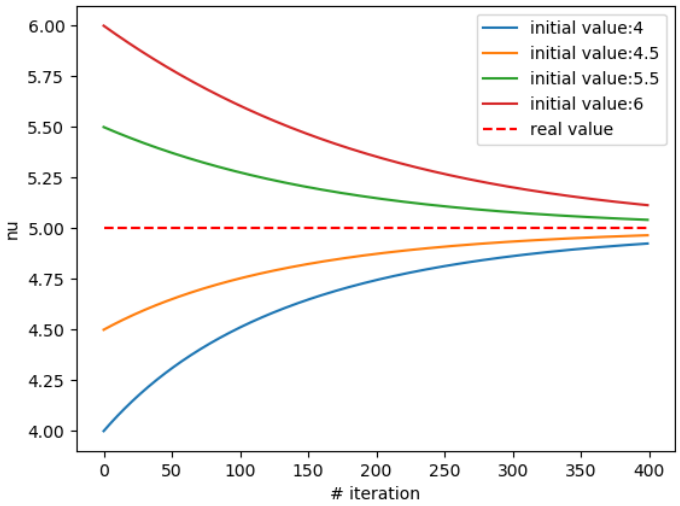


Figure 26: 2D Square: Convergence of different initial values

6.2.2 Step size

From the plot, it is clear that if the step size α is very big, the converging process is oscillatory at first and then converge. When α decreases, the oscillation disappears and just converges to the exact value. Obviously, larger alpha makes calculation faster. And with the same number of iteration, the larger step size α is, more accurate the value will be.

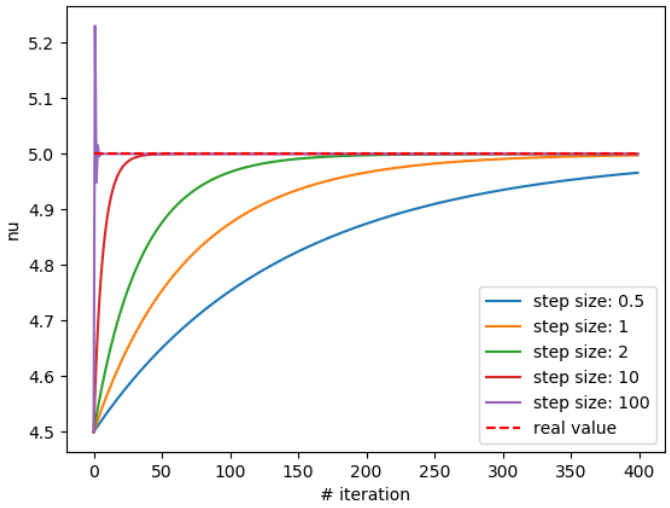


Figure 27: 2D Square: Step size (α) effect for convergence

6.2.3 Mesh size

We change the nx and ny in the unit square mesh to change the mesh size. We increase mesh number, which indicates decrease in mesh size, to increase the speed of the calculation. When the iteration number is limited, smaller mesh size will have more accurate result.

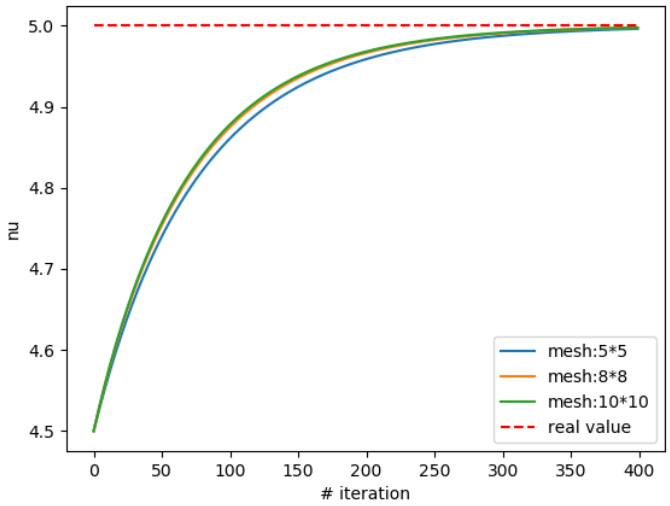


Figure 28: 2D Square: Mesh size effect for convergence

6.3 Results and analysis for 2D piston

6.3.1 Confirmation and initial value

With the same approach as the 2D square, we choose two initial values separately from both sides to the exact value. As shown in Figure 29, all initial values converge with the iteration approaching the exact value. The initial value gets closer to the exact value which gives more accurate results with same number of iterations. We observe similar behavior in the 2D square case.

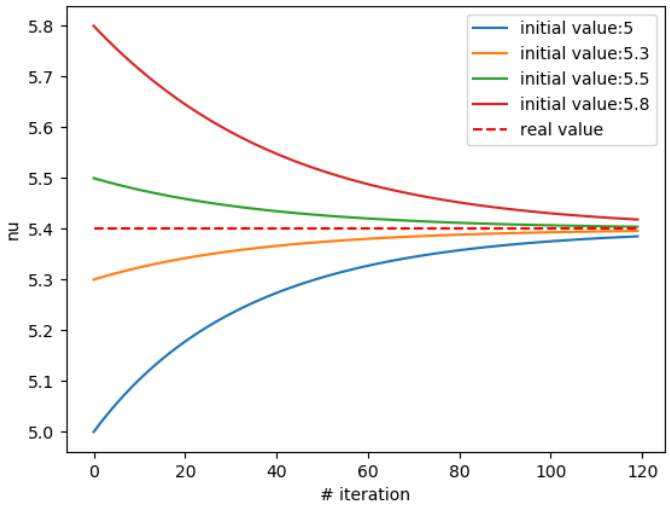


Figure 29: 2D Piston: Convergence of different initial values

6.3.2 Step size

As the step size α goes up, the converging process becomes faster and faster. As shown in Figure 30, given that we run the same number of iterations, higher step size can achieve better results. That means higher step size can result in faster and more accurate parameter estimation at the same time.

However, there are also some problems when the step size α is too big. As the cyan line shows in Figure 30, when the step size α is 100, the iteration process will not converge anymore, which means this step size α fail to deal with the problem properly.

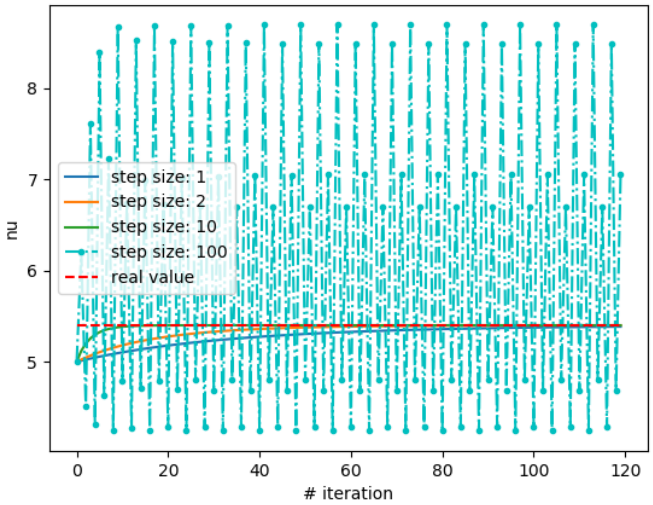


Figure 30: 2D Piston: Step size (α) effect for convergence

6.4 Summary of AD

After testing for different cases, we can draw the following conclusions.

6.4.1 Advantages

Firstly, fenics-adjoint can solve algorithmic differentiation problems corresponding to heat diffusive equation very well. Also, there are many choices for users when they want to analyze different aspects. Compared to the first-order finite difference method, the algorithm in fenics-adjoint makes the converging process much faster and stable, which means less oscillation, so that users can choose a bigger step size to save time and energy.

6.4.2 Problems and further work

When we choose a relatively small initial value, it is really challenging to adjust a proper step size α to make a balance between running time and convergence. That is because if step size α is relatively large, the loop will overshoot and not converge at all. On the other hand, if choose relatively small step size, the Jacobian $\frac{dJ}{d\nu}$ will drop very quickly just after a few iterations, which makes updating ν almost useless.
To deal with this problem, changing step size α in every iteration is a feasible solution. However, it is hard to say how to regulate the step size every time and whether it can work correctly without overshooting halfway.

7 Conclusion

In this project, we developed an unstructured finite element solver by exploring various cases with different geometries and boundary conditions with the help of FEniCS. We established the basic structure of the finite element solver and verified it to an analytical solution in the first 2D square case. Then we expanded the solver to the 2D cylinder and 2D rod geometry with Dirichlet and Neumann boundary

conditions. We further optimized the serial solver via mesh and time step refinement study on these cases. Our results showed good agreement between the numerical approximation and the analytical solution which gave us the confidence to apply our solver on the complex piston cases without a known analytical solution. For the piston case, we explored three different boundary conditions for both 2D and 3D pistons. Interestingly, the temperature distribution along the section planes for 2D and 3D piston exhibited a very similar behavior for all different boundary conditions. No significant difference was noticed in the results with the mesh refinement. An important takeaway from this study is to consider the simplification of complex 3D problems to lower dimensions with appropriate boundary conditions and still achieve the same accuracy in results with lower computational time and cost. We further parallelized our solver with the MPI to efficiently handle bigger and more complex physical domains. However, we observed that while MPI offers us the possibility to split the tasks among multiple processors, simply running on more processors could lead to communication overkill resulting in higher computation times. With a higher mesh count, the 3D piston was a suitable case for parallelization. Finally, the algorithmic differentiation was implemented on the 2D square and 2D piston case to reverse engineer the solution field to estimate the parameter - diffusion coefficient (ν). The test revealed the importance of choosing the optimal step size α to find the balance between runtime and convergence. A possible solution would be to change the step size α for every iteration. Future studies may be done in understanding how to regulate the step size and ensuring that it works without overshooting.

References

[1] Perumal Nithiarasu Roland Wynne Lewis and Kankanhalli Seetharamu. *Fundamentals of the finite element method for heat and fluid flow.* John Wiley Sons, 2004.

[2] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python - The FEniCS Tutorial.* Springer, 2017.

[3] Loic Wendling. Unstructured fem solver project. *Lecture Notes*, 2019.

[4] R Lightfoot, Byron Bird, Warren E. Stewart, and Edwin N. *Transport Phenomena.* John Wiley & Sons, 1960.

[5] Simon W. Funke Sebastian K. Mitusch and JÃ¸rgen S. Dokken. dolfin-adjoint 2018.1: automated adjoints for fenicsand firedrake. *Journal of Open Source Software, 4(38), 1292, doi:10.21105/joss.01292*, 2019.

[6] Axel Gerstenberger. An xfem based fixed-grid approach to fluid-structure interaction. Ph. D. Thesis, 2010.

[7] Wikipedia. Aluminium, 2020.

[8] Loic Wendling. Unstructured fem solver project. *Piston for internal combustion engine - Task Sheet*, 2018.

Appendix

Case 1

```
from __future__ import print_function
from fenics import *
import numpy as np

T = 2.0              # final time
num_steps = 10  # number of time steps
dt = T / num_steps  # time step size
alpha = 3            # parameter alpha
beta = 1.2           # parameter beta

# Create mesh and define function space
nx = ny = 8
mesh = UnitSquareMesh(nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t', degree=2, alpha=alpha,
    beta=beta, t=0)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define initial value
u_n = interpolate(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx #u_n+1 is u
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V) # different u; solution u not for the variational form.
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
    u_D.t = t

    # Compute solution
    solve(a == L, u, bc)

    # Plot solution
    plot(u)

    # Compute error at vertices
    u_e = interpolate(u_D, V) #reference solution
    error = np.abs(u_e.vector().get_local() - u.vector().get_local()).max()
    print('t = %.2f: error = %.3g' % (t, error))
```

```
    # Update previous solution
    u_n.assign(u)

# Hold plot
vtkfile = File('solution.pvd')
vtkfile << u
```

Case 2

```
from __future__ import print_function
from fenics import *
import numpy as np

T = 2              # final time
num_steps = 50     # number of time steps
dt = T / num_steps  # time step size
alpha = 3             # parameter alpha
beta = 1.2            # parameter beta

mesh = Mesh("cylinder.xml");
V = FunctionSpace(mesh, 'P', 1)

subdomains = MeshFunction("size_t", mesh, "cylinder_physical_region.xml")
boundaries = MeshFunction("size_t", mesh, "cylinder_facet_region.xml")

u_inner = 5
u_outer = 1
r_inner = 0.5
r_outer = 1.0

bc1 = DirichletBC(V, 5, boundaries, 3)
bc2 = DirichletBC(V, 1, boundaries, 2)

boundary_nodes = len(bc1.get_boundary_values()) + len(bc2.get_boundary_values())
bcs = [bc1, bc2]

u_D = Expression("10", degree = 1)

u_analytical = Expression('u_inner - (std::log(sqrt(x[0]*x[0] + x[1]*x[1])) -
    std::log(r_inner))*(u_inner - u_outer)*1/(std::log(r_outer) - std::log(r_inner))',
    degree=1, u_outer=u_outer, u_inner=u_inner, r_outer=r_outer, r_inner=r_inner)

# Define initial value
u_n = interpolate(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = 0

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
```

```
    t += dt
    #u_analytical.t = t

    # Compute solution
    solve(a == L, u, bcs)

    # Plot solution
    plot(u)

    # Compute error at vertices
    u_e=interpolate(u_analytical, V)
    error = (u - u_e)**2*dx
    err = sqrt(assemble(error))
    print('t = %.2f: L2 norm of error = %.3g' % (t, err) )

    error = np.abs(u_n.vector().get_local() - u.vector().get_local()).max()
    if t == dt:
        init_res = error
    print('t = %.2f: residual = %.3g' % (t, error/init_res))

    # Update previous solution
    u_n.assign(u)

    # Hold plot
    vtkfile = File('solution.pvd')
    vtkfile << u_e
    vtkfile << u
```

Case 3

```
from __future__ import print_function
from fenics import *
import numpy as np

T = 1              # final time
num_steps = 100    # number of time steps
dt = T / num_steps  # time step size
alpha = 3            # parameter alpha
beta = 1.2           # parameter beta

mesh = Mesh("rod.xml");
V = FunctionSpace(mesh, 'P', 1)

subdomains = MeshFunction("size_t", mesh, "rod_physical_region.xml")
boundaries = MeshFunction("size_t", mesh, "rod_facet_region.xml")

#Define neumann bcs
g_left = Constant(1.0)
g_rest = Constant(0.0)

#define measures
ds = Measure("ds")[boundaries]
u_D = Expression("0", degree = 1)
u_analytical = Expression('2.0*sqrt(t/pi)*(exp(-x[0]*x[0]/(4*t)) -
    0.5*x[0]*sqrt(pi*t)*erfc(x[0]/(2*sqrt(t))))', degree=1, t=0)

# Define initial value
u_n = interpolate(u_D, V)
```

```
# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = 0

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx - dt*(g_left*v*ds(2) -
    g_rest*v*ds(3))
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
    t += dt
    u_analytical.t = t

    # Compute solution
    solve(a == L, u)

    # Plot solution
    plot(u)

    # Compute error at vertices
    u_e=interpolate(u_analytical, V)
    error = (u - u_e)**2*dx
    err = sqrt(assemble(error))
     print('t = %.2f: L2 norm of error = %.3g' % (t, err) )

    #Compute residual
    error = np.abs(u_n.vector().get_local() - u.vector().get_local()).max()
    if t == dt:
        init_res = error
    print('t = %.2f: residual = %.3g' % (t, error/init_res))

    # Update previous solution
    u_n.assign(u)

# Hold plot
vtkfile = File('solution.pvd')
vtkfile << u_e
vtkfile << u
```

Case 4 - 2D

```
from __future__ import print_function
from fenics import *
import numpy as np

T = 0.12                    # final time
dt = 0.0012                 # time step size
num_steps = int(T/dt)  # number of time steps
alpha = 5.4E-5        # thermal diffusivity of the material(m^2/s)

# Import the mesh
mesh = Mesh("piston-2d.xml");
```

```
# Define the function space
V = FunctionSpace(mesh, 'P', 1)

# Import subdomains and boundaries
subdomains = MeshFunction("size_t", mesh, "piston-2d_physical_region.xml")
boundaries = MeshFunction("size_t", mesh, "piston-2d_facet_region.xml")

# Calculating average h value for the mesh
hmax = mesh.hmax();
hmin = mesh.hmin();
h = (hmax + hmin) * 0.5

# Temperature values for boundaries
T1 = Constant(300)
T2 = Constant(100)
T3 = Constant(30)

# Define Dirichlet boundary conditions
bc1 = DirichletBC(V,T1,boundaries,1)
bc2 = DirichletBC(V,T2,boundaries,2)
bc3 = DirichletBC(V,T3,boundaries,3)
bcs = [bc1, bc2, bc3]

# Initial condition
u_D = Constant(0)

# Define initial value by interpolation
u_n = interpolate(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = 0

F = u*v*dx + dt*alpha*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0

# Conergence Value
converged = 1E-9

# Open a vtk file to save the solution
vtkfile_u = File('task1_dt=%.3g_h=%.3g/u.pvd' % (dt,h))

for n in range(num_steps):

    # Update current time
    t += dt

    # Compute solution
    solve(a == L, u, bcs)

    # Compute difference of the previous and current time step - max value
    diff = np.abs(u_n.vector().get_local() - u.vector().get_local()).max()
    if t == dt:
```

```
        init_res = diff

    # Compute residual
    res = diff/init_res

    # Print t and res
    print('t = %.2f: residual = %.3g' % (t, res))

    # Break if convergence is achieved
    if res < converged:
        break

    # Update previous solution
    u_n.assign(u)

    # Write the solution to the file
    vtkfile_u << (u,t)
```

Case 4 - 3D

```
from __future__ import print_function
from fenics import *
import numpy as np

T = 10.0           # final time
num_steps = 500    # number of time steps
#dt = T / num_steps  # time step size
dt = 0.02

mesh = Mesh("piston-3d.xml");
V = FunctionSpace(mesh, 'P', 1)

subdomains = MeshFunction("size_t", mesh, "piston-3d_physical_region.xml")
boundaries = MeshFunction("size_t", mesh, "piston-3d_facet_region.xml")

alpha = 5.4E-5
rho = 2710.0
cp = 920.0
k = alpha*rho*cp

#Define neumann bcs
g_1= Constant(2E+06)
g_2 = Constant(100.0)
g_3 = Constant(30.0)
g_4 = Constant(0.0)
g_5 = Constant(0.0)

bc2 = DirichletBC(V, g_2, boundaries, 2)
bc3 = DirichletBC(V, g_3, boundaries, 3)
bcs = [bc2, bc3]

#define measures
ds = Measure('ds', domain=mesh, subdomain_data=boundaries)
dx = Measure('dx', domain=mesh, subdomain_data=subdomains)

u_D = Constant(50.0)

# Define initial value
```

```
u_n = interpolate(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = 0.0

F = u*v*dx + alpha*dt*dot(grad(u), grad(v))*dx - (u_n + f*dt)*v*dx -
    dt*(alpha/k)*(g_4*v*ds(4) + g_5*v*ds(5) + g_1*v*ds(1))
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
    t += dt

    # Compute solution
    solve(a == L, u, bcs)

    #Compute residual
    error = np.abs(u_n.vector().get_local() - u.vector().get_local()).max()
    if t == dt:
        init_res = error
    print('t = %.5f: residual = %.8f' % (t, error))

    # Update previous solution
    u_n.assign(u)

    vtkfile = File("solution_neumann"+str(n)+".pvd")
    vtkfile << u
```

Case 5 - 2D Square

```
from __future__ import print_function
from fenics import *
from fenics_adjoint import *
import numpy as np
import matplotlib.pyplot as plt

T = 2                       # final time
num_steps = 20        # number of time steps
dt = T / num_steps    # time step size
nu = Constant(4.5)    # diffusion coefficient nu
beta = 1.2                  # parameter beta
alpha = 1                   # step size

# Create mesh and define function space
nx = ny = 5           # mesh size parameter
mesh = UnitSquareMesh(nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Read data from case1
case1_ref = Function(V)
hdf = HDF5File(mesh.mpi_comm(), "case1_5.h5", "r")
attr = hdf.attributes("case1")
dataset = "case1/vector_0"
```

```
attr = hdf.attributes(dataset)
hdf.read(case1_ref, dataset)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + nu*x[1]*x[1] + beta*t', degree=2, nu=nu, beta=beta, t=0)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define variational problem
f = Expression('beta - 2 - 2*nu', degree=2, nu=nu, beta=beta, t=0)

# Time-stepping

control = Control(nu) # make nu as a variable

arr_nu = []           # save every nu
arr_J = []            # save every J
arr_dJ = []           # save every gradient dJdnu

vtkfile = File('case5.pvd')
for i in range(400):

    u = TrialFunction(V)
    v = TestFunction(V)

    # define initial value
    u_n = interpolate(u_D, V)

    # define variational problem
    F = u*v*dx + nu*dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
    a, L = lhs(F), rhs(F)

    # initialization
    u = Function(V)
    t = 0
    for n in range(num_steps):

        # Update current time
        t += dt
        u_D.t = t

        # Compute solution
        solve(a == L, u, bc)

        # Update previous solution
        u_n.assign(u)

    #vtkfile << u

    # compute gradient with AD
    J = assemble(inner(case1_ref-u, case1_ref-u)*dx)
    dJdnu = compute_gradient(J, control)

    arr_nu.append(nu)
    arr_J.append(J)
```

```
    arr_dJ.append(dJdnu)

    # update nu
    nu = nu - alpha*dJdnu

    # update conditions in the problem
    u_D.nu = Constant(nu)
    u_D.t = 0
    bc = DirichletBC(V, u_D, boundary)
    del u_n
    f.nu = Constant(nu)

# plot
np.savetxt('mesh5_alpha_1_nu_4_5.txt', np.column_stack((arr_nu, arr_J, arr_dJ)), header='nu
    J dJdnu', delimiter=' ')
plt.figure()
plt.plot(range(400),arr_J,label='J')
plt.plot(range(400),arr_dJ,label='dJdnu')
plt.legend()
plt.show()
plt.savefig('mesh5_alpha_1_nu_4_5.png')
```

Case 5 - 2D Piston

```
from __future__ import print_function
from fenics import *
from fenics_adjoint import *
import numpy as np
import matplotlib.pyplot as plt


T = 0.12                       # final time
dt = 0.0012                 # time step size
num_steps = int(T/dt)       # number of time steps

nu = Constant(5)      # thermal diffusivity (m^2/s)
ro = 2710             # density of aluminum (kg/m^3)
cp = 920                 # specific heat of aluminum (J/kgK)
k = Expression('nu * ro * cp * 1E-5', degree=1, nu=nu, ro=ro, cp=cp) # thermal
    conductivity(W/mC)

alpha = 1

# Import the mesh
mesh = Mesh("piston-2d.xml");

# Define the function space
V = FunctionSpace(mesh, 'P', 1)

# read reference data
case4_ref = Function(V)
hdf = HDF5File(mesh.mpi_comm(), "case4.h5", "r")
attr = hdf.attributes("case4")
dataset = "case4/vector_0"
attr = hdf.attributes(dataset)
hdf.read(case4_ref, dataset)

# Import subdomains and boundaries
```

```
subdomains = MeshFunction("size_t", mesh, "piston-2d_physical_region.xml")
boundaries = MeshFunction("size_t", mesh, "piston-2d_facet_region.xml")

# Calculating average h value for the mesh
hmax = mesh.hmax();
hmin = mesh.hmin();
h = (hmax + hmin) * 0.5

# Measure ds values again to fit the boundaries
ds = Measure('ds', domain=mesh, subdomain_data=boundaries)
qdot = 2E6 # Given qdot value for boundary 1

# Temperature values for boundaries
T2 = Constant(100)
T3 = Constant(30)

# Define Dirichlet boundary conditions
bc2 = DirichletBC(V,T2,boundaries,2)
bc3 = DirichletBC(V,T3,boundaries,3)

bcs = [bc2, bc3]

# Initial condition
u_D = Constant(0)

# Conergence Value
converged = 1E-9

# loop
control = Control(nu)

arr_nu = []
arr_J = []
arr_dJ = []

for i in range(120):
    # Define initial value by interpolation
    u_n = interpolate(u_D,V)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    f = 0

    F = u*v*dx + dt*nu*1E-5*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx -
        dt*nu*1E-5*(qdot/k)*v*ds(1)
    a, L = lhs(F), rhs(F)

    u = Function(V)
    t = 0

    for n in range(num_steps):

        # Update current time
        t += dt

        # Compute solution
        solve(a == L, u, bcs)
```

```
        # Compute difference of the previous and current time step - max value
        diff = np.abs(u_n.vector().get_local() - u.vector().get_local()).max()
        if t == dt:
            init_res = diff

        # Compute residual
        res = diff/init_res

        # Print t and res
        #print('t = %.2f: residual = %.3g:' % (t, res))

        # Break if convergence is achieved
        if res < converged:
            break

        # Update previous solution
        u_n.assign(u)

        # Write the solution to the file
        #vtkfile_u << (u,t)

    # calculate gradient with AD
    J = assemble(inner(case4_ref-u, case4_ref-u)*dx)
    dJdnu = compute_gradient(J, control)

    # save data
    arr_nu.append(nu)
    arr_J.append(J)
    arr_dJ.append(dJdnu)

    # update nu
    nu = nu - alpha*dJdnu

    # update parameter
    k.nu = Constant(nu)

# save data and Plot
np.savetxt('alpha_1_nu_5.txt', np.column_stack((arr_nu, arr_J, arr_dJ)), header='nu J
    dJdnu', delimiter=' ')
plt.figure()
plt.plot(range(120),arr_J,label='J')
plt.plot(range(120),arr_dJ,label='dJdnu')
plt.legend()
plt.show()
plt.savefig('fig.png')
```