FEM Heat App - Documentation and Manual

Samuel Selleck (sa2421se-s)

May 28, 2021

1 Program Use Case

The program could for example be used for estimating heat distribution in a square house with a central heating element such as a furnace.

2 Architecture

The program is devided into two files, one with all the model and data definitions, as well as the computation done with these. The other the gui application logic connecting the graphical interface defined in a separate file with the execution of logic in the model file. The model contains the following classes:

- Solver - responsible for solving the fem problem
- InputData responsible for handling input and geometry generation
- OutputData - collects all output from a solved problem
- Visualization - used to visualize the results

The GUI contains the following classes:

- SolverThread - class to handle threading for the solver, to not block main gui
- MainWindow - the main PyQt5 window, connects all buttons to their action.

3 Assessment of Correctness

The temperature distribution is strictly decreasing from the edge of high temerature to the outside edge with lower temperature, and does so in a non-linear fassion. The temerature distribution between long flat edges is uniform and almost unaffected by the heat value at either sides. The total flux on the two boundaries was verified to sum to zero along the boundaries for a few different parameters.

4 Program Manual

Bellow follows a series of figures explaining the layout of the program and how to use it.

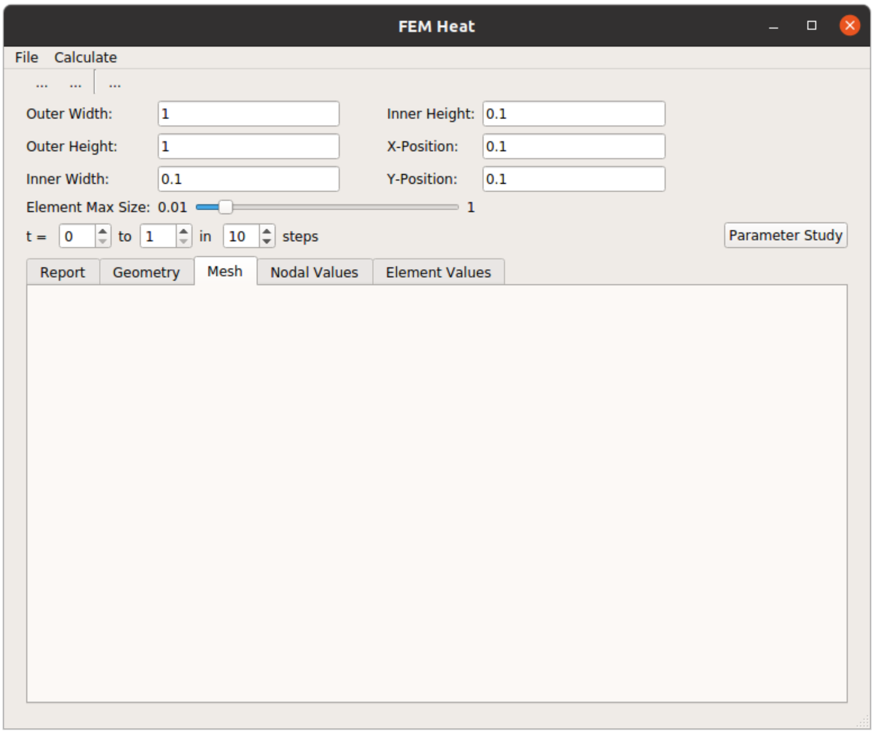

Figure 1: The layout of the program. Use the File menu at the top to save or load a configuration.

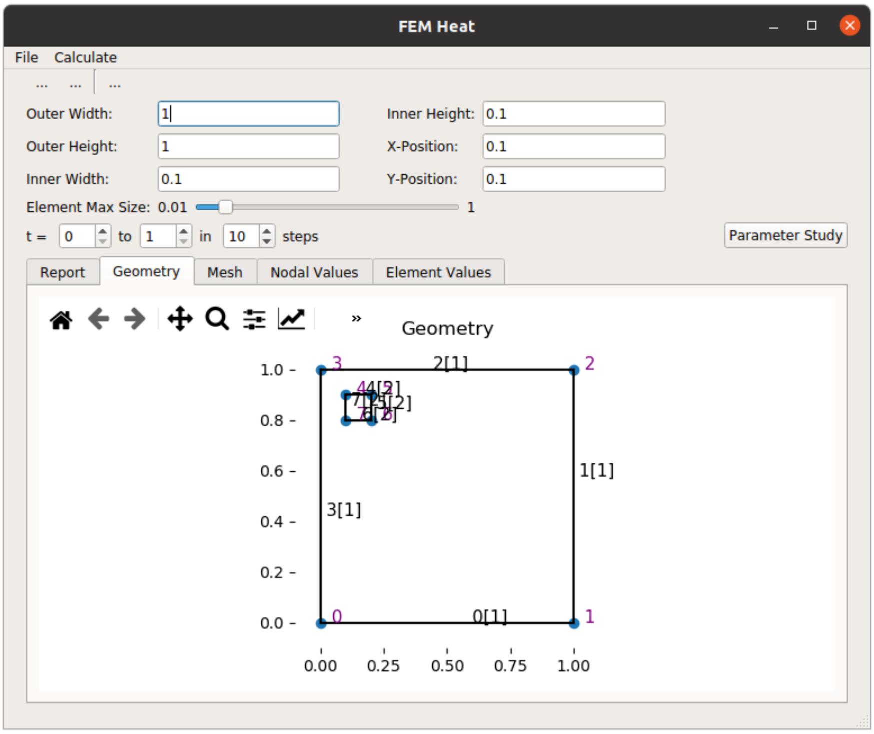

Figure 2: The geometry tab. Shows in real time the geometry of the defined problem, using the parameters at the top of the page.

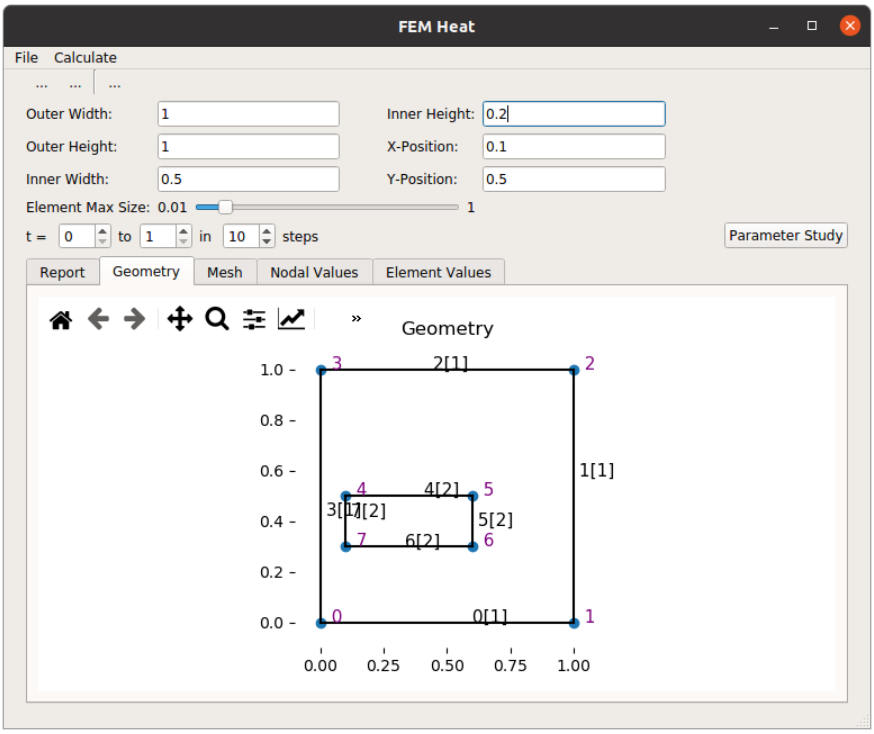

Figure 3: When the desired geometry has been created, the finite element computation can be run by nativating to Calculate → Execute.

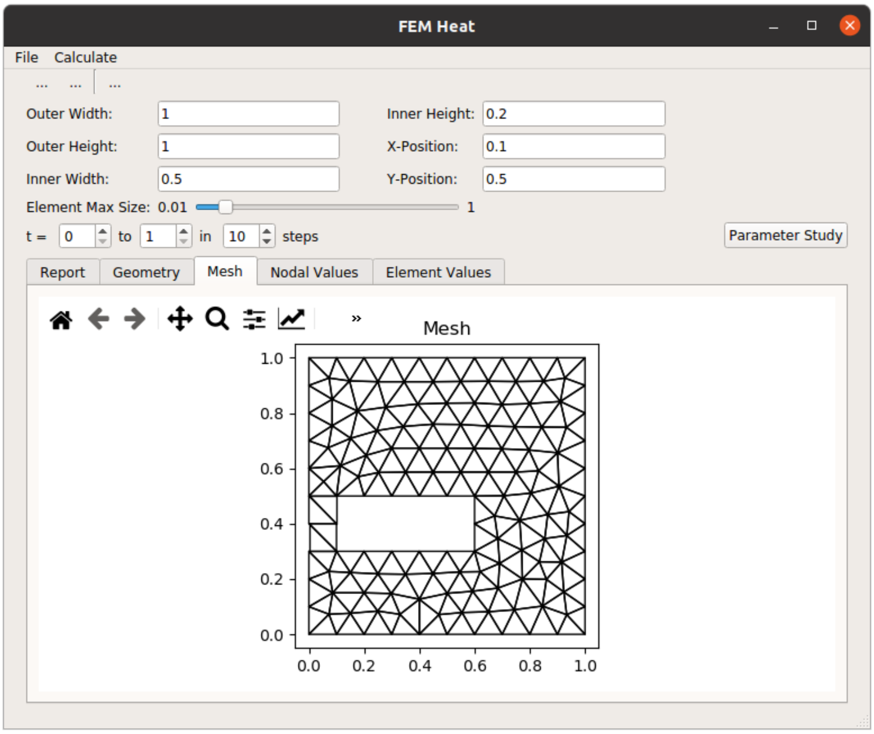

Figure 4: The mesh view. Mesh size can be increased and decreased using the slider.

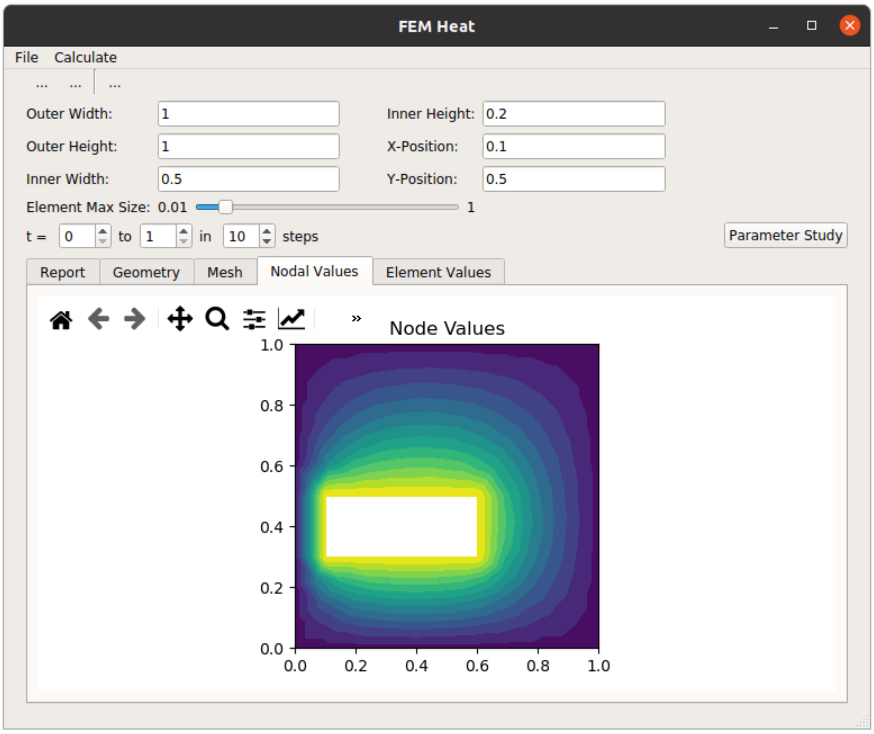

Figure 5: The node values view.

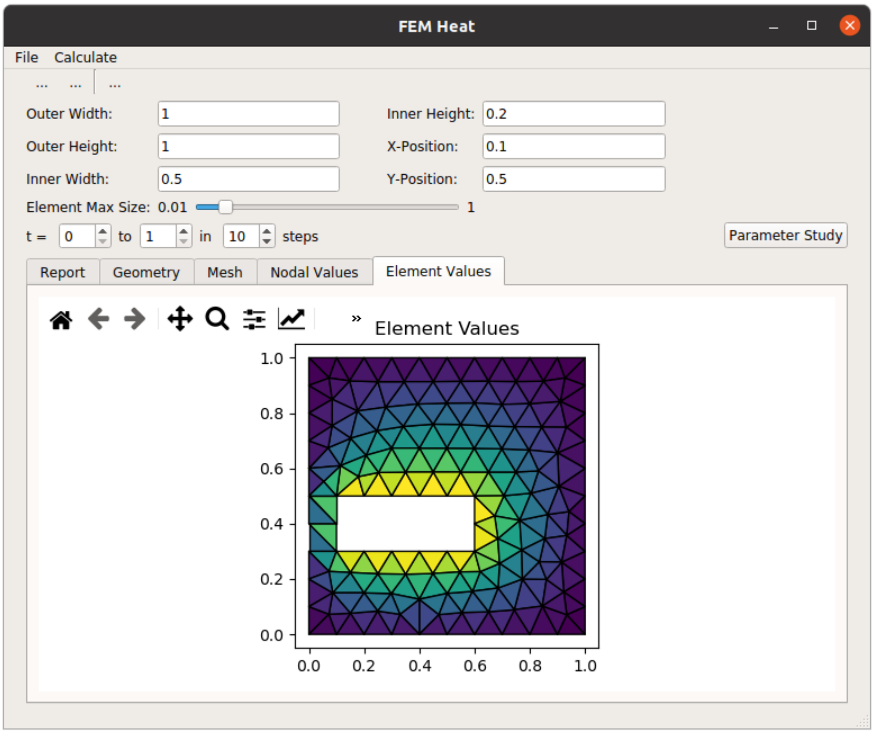

Figure 6: The element values view.

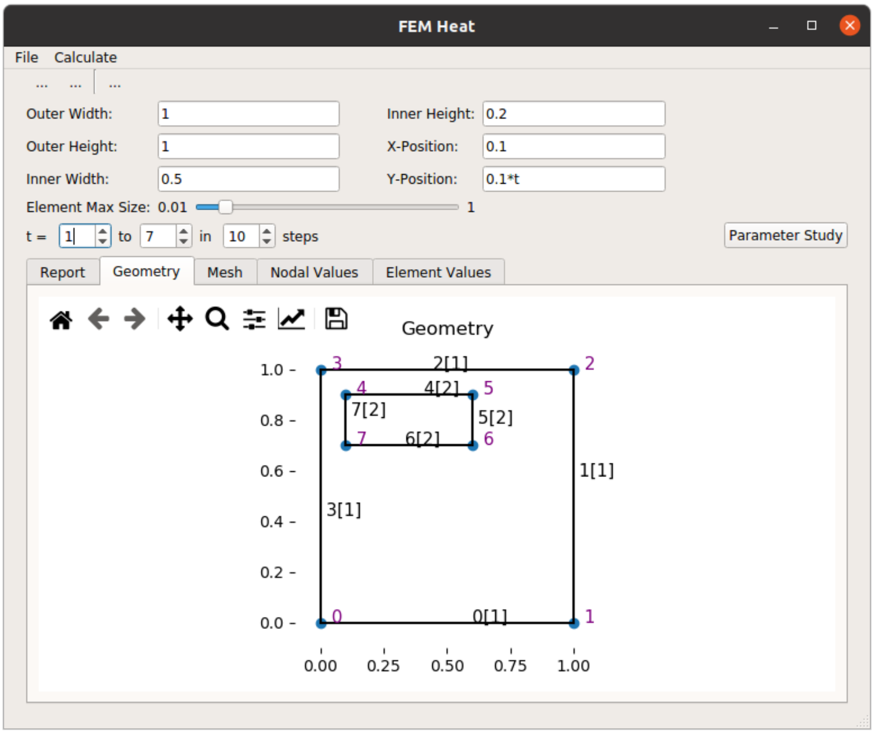

Figure 7: Parameter studys are done by letting the parameters vary with the variable t. The geometry for normal executions is used with the value of t defined in the first box.

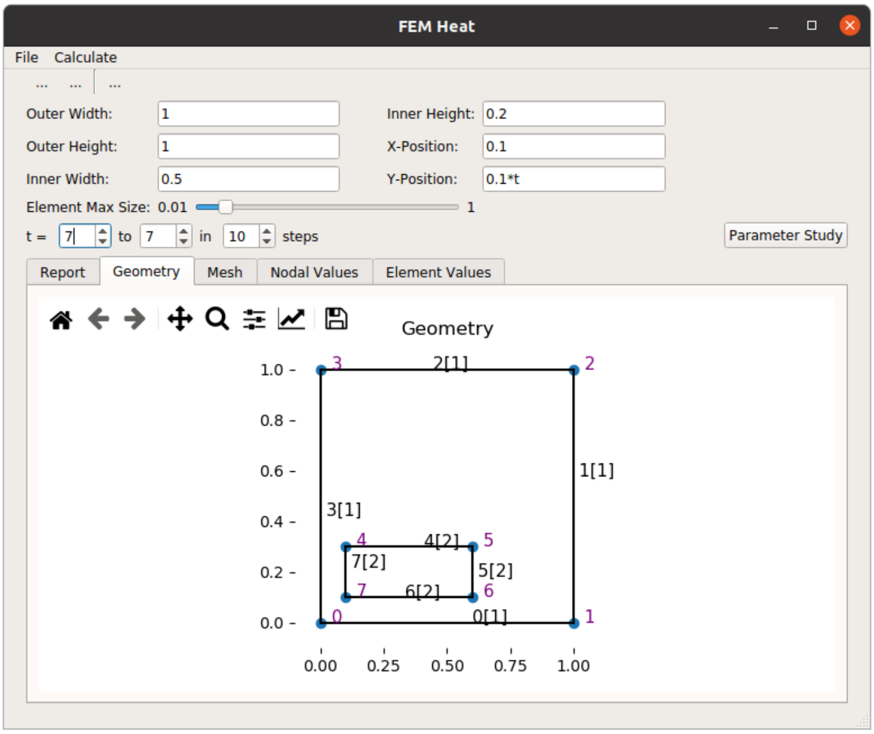

Figure 8: Increasing t in this case moves the box downwards. Remember to put the value back before executing the parameter study.

5 Appendix

Listing 1: Model

```
import numpy as np
import json

import calfem.core as cfc
import calfem.geometry as cfg # <-- Geometrirutiner
import calfem.mesh as cfm # <-- NÃďtgenerering
import calfem.vis_mpl as cfv # <-- Visualisering
import calfem.utils as cfu # <-- Blandade rutiner

import pyvtk as vtk
import matplotlib.pyplot as plt


MARKERS = {
    "outer": 1,
    "inner": 2,
}

class Solver(object):
    """Klass fÃűr att hantera lÃűsningen av vÃěr berÃďkningsmodell
        ."""
    def __init__(self, input_data, output_data):
        self.input_data = input_data
        self.output_data = output_data

    def execute(self, param_study):

        # -- constants
        cond = np.asarray(self.input_data.conduction)
        outer_temp = self.input_data.outer_temp
        inner_temp = self.input_data.inner_temp
        ep = [self.input_data.thickness]

        t_from = self.input_data.t_from
        t_to = self.input_data.t_to
        t_steps = self.input_data.t_steps
```

```
space = np.linspace(t_from, t_to, t_steps) if param_study
   else [t_from]

for index, t_param in enumerate(space):
    geometry = self.input_data.geometry(t_param)

    mesh = cfm.GmshMeshGenerator(geometry)
    mesh.el_size_factor = self.input_data.element_max_size
       /100
    mesh.el_type = 2
    mesh.dofs_per_node = 1
    mesh.return_boundary_elements = True

    coords, edof, dofs, bdofs, elementmarkers,
       boundaryElements = mesh.create()

    n = len(coords);
    K = np.zeros([n, n])
    f = np.zeros([n, 1])
    bc_prescr, bc_val = [], []

    # --- Applicera randvillkor/laster
    bc_prescr, bc_val = cfu.applybc(bdofs, bc_prescr, bc_val
       , MARKERS["outer"] , outer_temp)
    bc_prescr, bc_val = cfu.applybc(bdofs, bc_prescr, bc_val
       , MARKERS["inner"], inner_temp)
    bc_prescr = bc_prescr.astype(int)

    ex, ey = cfc.coordxtr(edof, coords, np.arange(1, n + 1)
       [..., None])

    D = np.array([
        [cond, 0],
        [0, cond],
    ]);

    for elx, ely, eltopo, in zip(ex, ey, edof):
        Ke = cfc.flw2te(elx, ely, ep, D)
        cfc.assem(eltopo, K, Ke)
```

```
        t, r = cfc.solveq(K, f, bc_prescr, bc_val)

        flow = [cfc.flw2ts(ex[i], ey[i], D, np.array(t[el-1]).
           reshape(-1,))[0].tolist()[0]
               for i, el, in enumerate(edof)]

        max_flow = [np.linalg.norm(flw) for flw in flow]

        element_t = [np.mean(np.array(t[el-1]).reshape(-1,)) for
            el in edof]

        if param_study:
           Solver.export_vtk(param_study, coords, edof, t, flow,
              max_flow, index)
        else:
            self.output_data.update({
                "element_t": element_t,
                "t": t.tolist(),
                "r": r,
                "coords": coords,
                "edof": edof,
                "geometry": geometry,
                "el_type": mesh.el_type,
                "dofs_per_node": mesh.dofs_per_node,
                "max_flow": max_flow,
                "flow": flow,
            })

def export_vtk(filename, coords, edof, t, flow, max_flow, index)
    :
    points = [[*coord, 0] for coord in coords.tolist()]
    polygons = (edof-1).tolist()
    flow_3d = [[*flw, 0] for flw in flow]

    point_data = vtk.PointData(vtk.Scalars(t.tolist(), name="
       temperature"))
    cell_data = vtk.CellData(vtk.Scalars(max_flow, name="maxflow
       "), vtk.Vectors(flow_3d, "flow"))
```

```
        structure = vtk.PolyData(points = points, polygons =
            polygons)

        vtk_data = vtk.VtkData(structure, point_data, cell_data)
        filename = f"{filename}_{index:02d}.vtk"
        vtk_data.tofile(filename, "ascii")
        print(f"saved␣file:␣{filename}")



class InputData(dict):
    """Klass fÃűr att definiera indata fÃűr vÃěr modell."""
    def __init__(self):
        pass

    def save(self, filename):
        with open(filename, "w") as ofile:
            json.dump(vars(self), ofile, sort_keys = True, indent =
                4)

    def load(self, filename):
        with open(filename, "r") as ifile:
            self.update(json.load(ifile))

    def update(self, attrs):
        for key, val in attrs.items():
            setattr(self, key, val)

    def __str__(self):
       attr = vars(self)
       aliases = {
           "thickness": "Thickness",
           "x_position": "X-position",
           "y_position": "Y-position",
           "outer_width": "Outer␣Width",
           "outer_height": "Outer␣Height",
           "inner_width": "Inner␣Width",
           "inner_height": "Inner␣Height",
           "conduction": "Conduction",
       }
```

```
    return "\n".join(f'{val}:\n{attr[key]}' for key, val in
        aliases.items())

def geometry(self, t_param):
    """Skapa en geometri instans baserat pÃě definierade
        parametrar"""

    glob = {"__builtins__": {}}
    loc = {"t": t_param}

    try:
        w = eval(self.outer_width, glob, loc)
        h = eval(self.outer_height, glob, loc)
        a = eval(self.inner_width, glob, loc)
        b = eval(self.inner_height, glob, loc)
        x = eval(self.x_position, glob, loc)
        y = eval(self.y_position, glob, loc)
    except Exception:
        return

    points = [
        [0, 0], [w, 0], [w, h], [0, h],
        [x, h - y], [x + a, h - y], [x + a, h - y - b], [x, h -
            y - b]
    ]

    boundaries = {
        "outer": [[0, 1], [1, 2], [2, 3], [3, 0]],
        "inner": [[4, 5], [5, 6], [6, 7], [7, 4]]
    }

    g = cfg.Geometry()

    for point in points:
        g.point(point)

    for marker_name, splines in boundaries.items():
        for spline in splines:
            g.spline(spline, marker = MARKERS[marker_name])
```

```
        g.surface([0,1, 2, 3], [[4, 5, 6, 7]])

        return g


class OutputData:
    """Klass fÃűr att lagra resultaten frÃěn berÃďkningen."""

    def __init__(self):
        pass

    def update(self, attrs):
        for key, val in attrs.items():
            setattr(self, key, val);

    def __str__(self):
        attr = vars(self)
        aliases = {
           "t": "Temperatures",
        }
        return "\n".join(f'{val}:\n{attr[key]}' for key, val in
           aliases.items())

class Report:
    """Klass fÃűr presentation av indata och utdata i rapportform.
        """
    def __init__(self, input_data, output_data):
        self.input_data = input_data
        self.output_data = output_data


    def __str__(self):
        return f'''
-------------- Model input ----------------------------------
{self.input_data}
-------------- Model output ---------------------------------
{self.output_data}
        '''

class Visualisation(object):
```

```python
def __init__(self, input_data, output_data):
    self.input_data = input_data
    self.output_data = output_data

    plt.ioff()
    self.geom_fig = None
    self.mesh_fig = None
    self.el_value_fig = None
    self.node_value_fig = None

def geometry(self):

    self.geom_fig = cfv.figure(self.geom_fig)
    cfv.clf()
    cfv.draw_geometry(self.output_data.geometry, title="Geometry
        ")
    return self.geom_fig

def mesh(self):
    self.mesh_fig = cfv.figure(self.mesh_fig)

    cfv.clf()
    cfv.draw_mesh(
        self.output_data.coords,
        self.output_data.edof,
        self.output_data.dofs_per_node,
        self.output_data.el_type,
        title="Mesh"
    )

    return self.mesh_fig

def nodal_values(self):
    """Visa geometri visualisering"""

    self.node_value_fig = cfv.figure(self.node_value_fig)

    cfv.clf()
    cfv.draw_nodal_values(
        self.output_data.t,
```

```
            self.output_data.coords,
            self.output_data.edof,
            12, "Node␣Values",
            self.output_data.dofs_per_node,
            self.output_data.el_type,
            draw_elements=False
        )

        return self.node_value_fig

    def element_values(self):
        self.el_value_fig = cfv.figure(self.el_value_fig)

        cfv.clf()
        cfv.draw_element_values(
            self.output_data.element_t,
            self.output_data.coords,
            self.output_data.edof,
            self.output_data.dofs_per_node,
            self.output_data.el_type,
            title="Element␣Values"
        )

        return self.el_value_fig
```

Listing 2: GUI

```
import sys

from PyQt5.QtCore import pyqtSlot, pyqtSignal, QThread
from PyQt5.QtWidgets import QApplication, QDialog, QWidget,
   QMainWindow, QFileDialog, QMessageBox, QVBoxLayout
from PyQt5.uic import loadUi

import heatmodel as hm
import calfem.ui as cfui
import calfem.vis_mpl as cfv

class SolverThread(QThread):
    """Klass fÃűr att hantera berÃďkning i bakgrunden"""
```

```
    def __init__(self, solver, callback, param_study = ""):
        """Klasskonstruktor"""
        QThread.__init__(self)
        self.solver = solver
        self.param_study = param_study
        self.finished.connect(callback)

    def __del__(self):
        self.wait()

    def run(self):
        self.solver.execute(self.param_study)

class MainWindow(QMainWindow):
    """MainWindow-klass som hanterar vÃĕrt huvudfÃűnster"""

    def __init__(self):
        """Constructor"""

        # --- Init UI
        super(QMainWindow, self).__init__()

        self.app = app
        self.ui = loadUi('mainwindow.ui', self)

        self.components = {
            "triggered": {
            "action_new",
            "action_open",
            "action_save",
            "action_save_as",
            "action_exit",
            "action_execute"
            },
            "clicked": {
                "parameter_study_button",
                "save_tool_button",
                "open_tool_button",
                "execute_tool_button"
            },
```

```
    "textfields": {
        "outer_width",
        "outer_height",
        "inner_width",
        "inner_height",
        "x_position",
        "y_position",
    },
    "numfields": {
        "t_from",
        "t_to",
        "t_steps",
        "element_max_size"
    }
}

# --- Bind controls to methods
for to_connect in ["triggered", "clicked"]:
    for component_name in self.components[to_connect]:
        component = getattr(self.ui, component_name)
        getattr(component, to_connect).connect(getattr(self,
            f'on_{component_name}'))

# --- Bind controls to update geometry
for component_name in self.components["textfields"]:
    component = getattr(self.ui, component_name)
    component.returnPressed.connect(self.update_geometry)

self.ui.t_from.valueChanged.connect(self.update_geometry)

# --- Init model
self.input_data = hm.InputData()
self.output_data = hm.OutputData()
self.solver = hm.Solver(self.input_data, self.output_data)
self.report = hm.Report(self.input_data, self.output_data)
self.visualization = hm.Visualisation(self.input_data, self.
    output_data)
self.calc_done = True
self.init_input_data()
```

```
    # --- Show
    self.ui.show()
    self.ui.raise_()
    self.update_geometry()

def init_input_data(self):
    self.filename = None
    self.input_data.update({
        "version": 1,
        "outer_width": "1",
        "outer_height": "1",
        "inner_width": "0.1",
        "inner_height": "0.1",
        "x_position": "0.1",
        "y_position": "0.1",
        "t_from": 0,
        "t_to": 1,
        "t_steps": 10,
        "element_max_size": 10,
        "thickness": 1,
        "conduction": 1.7,
        "outer_temp": 20,
        "inner_temp": 120,
    })
    self.update_ui(self.input_data)

def on_save_tool_button(self):
    self.on_action_save()

def on_open_tool_button(self):
    self.on_action_open()

def on_execute_tool_button(self):
    self.on_action_execute()

def on_action_new(self):
    self.init_input_data()

def on_action_open(self):
    self.filename, _ = QFileDialog.getOpenFileName(self.ui,
```

```
        "ÃŰppna␣modell", "", "Modell␣filer␣(*.json)")

        if self.filename!="":
            self.input_data.load(self.filename)
            self.update_ui(self.input_data)
            self.update_geometry()

    def on_action_save(self):
        if self.filename:
            self.update_model(self.input_data)
            self.input_data.save(self.filename)
        else:
            self.on_action_save_as()

    def on_action_save_as(self):
        self.update_model(self.input_data)

        self.filename, _ = QFileDialog.getSaveFileName(self.ui,
            "Spara␣modell", "", "Modell␣filer␣(*.json)")

        if self.filename!="":
            self.input_data.save(self.filename)

    def on_action_exit(self):
        self.ui.close()
        self.app.quit()

    def on_action_execute(self):
        self.ui.setEnabled(False)
        self.update_model(self.input_data)

        self.solverThread = SolverThread(self.solver, self.
           on_finished_execute)
        self.calc_done = False
        self.solverThread.start()

    def on_finished_execute(self):
        self.ui.report_field.setPlainText(str(self.report))

        figure_names = ["geometry", "mesh", "nodal_values", "
```

```
           element_values"]
        for name in figure_names:
            self.update_figure(name)

        self.calc_done = True
        self.ui.setEnabled(True)

    def on_parameter_study_button(self):
        if self.filename:
            self.on_action_save()
            self.ui.setEnabled(False)
            self.update_model(self.input_data)

            self.solverThread = SolverThread(self.solver, self.
               on_finished_param_study, self.filename.split(".")[0])
            self.calc_done = False
            self.solverThread.start()
        else:
             msg = QMessageBox(
                QMessageBox.Information,
                "No␣save␣file.",
                "A␣project␣save␣file␣needs␣to␣exist␣to␣perform␣a␣
                   parameter␣study."
             )
             msg.exec_()


    def on_finished_param_study(self):
        self.calc_done = True
        self.ui.setEnabled(True)
        msg = QMessageBox(
            QMessageBox.Information,
            "Parameter␣Study␣Done",
            "The␣parameter␣study␣is␣finished,␣vtk␣files␣exported."
        )
        msg.exec_()

    def update_ui(self, model):
        for field in self.components["textfields"]:
            ui_field = getattr(self.ui, field)
```

```
            ui_field.setText(getattr(model, field))
        for field in self.components["numfields"]:
            ui_field = getattr(self.ui, field)
            ui_field.setValue(int(getattr(model, field)))

    def update_model(self, model):
        for field in self.components["textfields"]:
            ui_field = getattr(self.ui, field)
            setattr(model, field,ui_field.text())
        for field in self.components["numfields"]:
            ui_field = getattr(self.ui, field)
            setattr(model, field, ui_field.value())

    def update_figure(self, name):
        fig = getattr(self.visualization, name)()
        widget = cfv.figure_widget(fig)
        box = getattr(self.ui, f"{name}_box")
        box.takeAt(0).widget().deleteLater()
        box.addWidget(widget)

    def update_geometry(self):
        cfv.close_all()
        input_data = hm.InputData()
        self.update_model(input_data)
        self.visualization.output_data.geometry = input_data.
           geometry(input_data.t_from)
        self.update_figure("geometry")


if __name__ == "__main__":

    app = QApplication(sys.argv)

    widget = MainWindow()
    widget.show()

    sys.exit(app.exec_())
```