Here's my project plan for my final year project. I have to do a progress presentation now.

Project plan:

1.1 Project Brief and Motivation

Today’s bottleneck in optimizing thermal management solutions for chip and transistor cool-

ing lies in slow and incoherent models. As components continue to shrink in size while increasing

in power density, traditional Fourier heat conduction models have proved themselves insufficient.

These models often fail to capture critical non-equilibrium effects, especially in high Knudsen

number regimes or complex geometries where the mean free path of phonons starts to become

comparable to the device dimensions. To address these limitations, this project aims to develop

a versatile mesoscale heat conduction model that leverages the Guyer-Krumhansl (G-K) equa-

tions. This framework should be capable of capturing both diffusive and ballistic behaviors, thus

providing a comprehensive approach to non-Fourier conductive heat transport. This approach ad-

dresses the limitations of Boltzmann Transport Equation (BTE) solvers, which, while accurate, are

computationally intensive and often unsuitable for complex, multimaterial topologies necessary in

real-world applications. A streamlined and parametrized model will enable a seamless generation

of bimaterial topologies for the most portent modern electronic cooling solutions.

The motivation for this work is driven by this need for scalable and adaptable thermal design

tools, especially in semiconductor industries facing growing demands for efficient, compact cool-

ing solutions. By reducing the design space of thermal structures and parameterising a heat solver,

the resulting framework should determine thermal source placement and magnitudes that optimize

cooling efficiency and minimize hotspots. This is critical for fields requiring advanced design tech-

niques capable of adapting to various material properties, source layouts, and Knudsen numbers,

as commonly found in modern transistor or chip architectures.

A versatile and modular design framework is essential for bioinspired, multimaterial structures—

these mimic natural thermal efficiencies and can help drastically reduce the combinatorial design

space. Such structures, optimized using Variational Autoencoders (VAEs) and advanced image

processing techniques, enable the framework to systematically explore and map bioinspired con-

figurations onto mesoscale domains, opening new possibilities for thermal management in cutting-

edge electronic devices [1].

1.2 Summary of Previous Work

Previous research has explored the G-K model within single-material domains and simpler

topologies. For example, Tur-Prats et al. demonstrated the validity of the G-K model in cap-

1

turing hydrodynamic-like behaviors in nanoscale domains [2]. However, the authors do not apply

their framework to multimaterial configurations and optimizations, limiting its versatility. On the

other hand, the BTE has been applied successfully to bimaterial structures, but it is computa-

tionally intensive and does not facilitate practical, large-scale optimization for complex topologies.

Tang et al. showed that topology optimization within the BTE framework improves heat spreading

near electronic junctions, yet they emphasized the computational limitations in capturing ballistic-

diffusive effects in complex geometries [3]. This project extends this foundational work by apply-

ing a VAE to generate bioinspired, tree-like thermal configurations, using image processing to fit

these structures precisely onto the simulation domain, allowing for a modular and adaptable design

space well-suited to various cooling scenarios [1].

1.3 Project Overview

This project integrates three primary research elements to meet the demands of efficient ther-

mal design: finite element methods (FEM) for the G-K equations, VAEs for bioinspired bimate-

rial configurations, and advanced optimization techniques. Building the FEM solver in FEniCSx

presents multiple challenges, including complex mesh generation, solver setup, and parameteriza-

tion for various configurations. Implementing the G-K equations introduces additional complexity,

as it requires carefully crafted variational formulations that can accurately capture both Fourier

and non-Fourier conduction in the same framework [2]. Achieving a modular, Python-based sys-

tem allows for rapid iteration, suitable for high-performance computing (HPC) deployment, where

mesh parallelization and batch processing can handle large-scale simulations efficiently.

By parameterizing the model, this framework enables future adaptability, potentially allowing

the extension to porous media structures rather than bimaterial domains, offering further flexibility

and efficiency improvements in thermal design for microelectronics.

2 Project Objectives

The objectives for this project are as follows:

1. Develop and implement a mesoscale heat conduction model using the finite element method

(FEM) in FEniCSx:

• Construct the solver based on the Guyer-Krumhansl equations for capturing non-

Fourier heat transport characteristics.

• Parameterize the model to handle multiple heat sources, optimizing their placement

and intensity.

2

• Perform a mesh refinement study to validate model accuracy in temperature distribu-

tion predictions, ensuring reliable simulation outcomes.

2. Conduct extensive optimization studies to determine optimal heat source configurations for

thermal management:

• Implement state-of-the-art algorithms, including CMA-ES and Bayesian optimiza-

tion, to explore and identify configurations that minimize thermal buildup.

• Parallelize the optimization process on HPC, utilizing both source control and struc-

tured batch processing where applicable.

3. Investigate the effects of varying Knudsen numbers on heat transport:

• Compare the non-Fourier heat conduction characteristics modeled by the Guyer-Krumhansl

equations against classical Fourier conduction.

• Assess heat transfer across a bimaterial interface under these varying conditions.

• Assess effect of volume fraction control on bimaterial topology under varying condi-

tions.

4. Investigate the effect of different source locations and sizes on optimization solutions.

• Analyze the varying constructal patterns that emerge with different configurations.

• Identify corresponding ’real-world’ problem setups and find solutions accordingly.

---

The presentation must be 10 slides. I think I will have a slide with project statement / project overview, also explaining the novelty and challenges, a few slides explaining the progress so far, then a slide with future work.

The rubric is:

INTRODUCTION: (10%) may include introduction; motivation for the project; objectives; summary of

previous work; etc….

Problem unclear; no explanation

of intentions; no objectives E D C B A A\* Identifies the problem; captures

context; clearly states intentions

PROGRESS: (30%) may include summary of progress; conceptual, software or experimental design;

initial work; results and analysis to date; explanations of delays; etc…

No progress against original plan;

exaggerated progress; unclear E D C B A A\* progress and delays

Honest & accurate review of

GANTT CHART: (10%) may include indication of progress to date; changes to project plan; modification of

objectives, milestones or deliverables; etc…

No Gantt chart; not updated;

progress & changes not marked E D C B A A\* Clear and complete plan;

updated clearly; progress clear

FUTURE WORK: (20%) may include identification of future work; changes to project plan; modification of

objectives, milestones or deliverables; identification of barriers to progress; etc…

Vague or irrelevant; unreasonable

or very under/overambitious E D C B A A\* Clear and complete; achievable

but ambitious

STYLE & TIMEKEEPING: (20%) may include communication of work using graphs, tables, photographs,

micrographs, CAD, FEA, CFD, images, screenshots, flow charts;

timekeeping; annotation of images; etc…

Inaudible; confusing; untimely;

not engaging; template not used E D C B A A\* Audible; engaging; punctual; well

explained; well paced

QUALITY OF SLIDES: (10%) may include appearance; legibility; layout; flow; graphs, tables, images

legible and well labelled; text easy to read; etc…

Illegible; disorganised; poorly laid

out; unstructured; crammed E D C B A A\* Professional; clear; attractive;

fully legible

Note that a good presentation should be grade B. (A\* ≥ 85% > A ≥ 70% > B ≥ 60% > C ≥ 50% > D ≥ 40% > E)

Here's my progress so far:

I have written a massive python code. I have focused on GUI and streamlining sending things to the Imperial HPC.

I have started getting some results, comparing different latent spaces of the VAE in solving a 3 source problem.

I have spent a lot of time and effort parametrizing the meshing of the source so I can make multiple sources with varying heats. I spent a lot of time making it so that I can use a gui to send directly to the hpc:

Process for Send directly to hpc

Write password in terminal

Choose number of nodes, cpu per nodes,

Memory per node, MPI, etc…

Do user host directory and environment.

The code makes the script for youand you can send it

Enter the password and it transfers the relevant files

And the run script and it makes it run.

When youre done, you can move the files back,

And choose the config file with “Visualize best” and

It will run the post processor and visualize gamma,

Temp, flux, profiles, and convergence.

The first step was making the fenicsx solver, which solves the Guyer Krumhansl finite element equations. I did this with fenicsx. It took significant effort to set up the formulation in the fenicsx software. I will attached the code. Once I got that working, I needed to connect it to an optimizer. I connected it to bayes first, then to CMA-ES. Bayes didnt work. Now that CMA-ES works, I parametrized the whole thing. And I made a gui. And modularized the whole code. I am going to attach some of the code and I am going to attach my notes on notion (which are incomplete). And maybe my git commits.

I am pasting all my files as a json, hopefully you understand the format:

```

[

  {

    "filePath": "../HeatToolbox/src/\_\_init\_\_.py",

    "fileContent": ""

  },

  {

    "filePath": "../HeatToolbox/src/gui/\_\_init\_\_.py",

    "fileContent": ""

  },

  {

    "filePath": "../HeatToolbox/src/gui/hpc\_frame.py",

    "fileContent": "import customtkinter as ctk\n\n\nclass HPCFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options):\n        super().\_\_init\_\_(parent)\n        self.options = options\n\n        self.grid(row=2, column=0, columnspan=2, padx=10, pady=5, sticky=\"nsew\")\n        self.grid\_columnconfigure(0, weight=1)  # Reduce weight of label column\n        self.grid\_columnconfigure(1, weight=2)  # Increase weight for entries\n        self.grid\_columnconfigure(2, weight=1)  # Ensure equal spacing for HPC user settings\n        self.grid\_columnconfigure(3, weight=2)  # Ensure equal spacing for HPC user settings\n        self.grid\_rowconfigure(99, weight=1)  # Prevent bottom empty space\n\n        self.create\_widgets()\n\n        # Attach a trace to update GUI when hpc\_enabled changes\n        self.options[\"hpc\_enabled\"].trace\_add(\"write\", self.\_on\_hpc\_enabled\_changed)\n\n    def create\_widgets(self):\n        # Enable HPC Checkbox\n        self.hpc\_checkbox = ctk.CTkCheckBox(\n            self,\n            text=\"Enable HPC\",\n            variable=self.options[\"hpc\_enabled\"]\n            # command=self.toggle\_hpc\_options,\n        )\n        self.hpc\_checkbox.grid(row=0, column=0, sticky=\"w\")\n\n        # HPC Options\n        self.nodes\_label = ctk.CTkLabel(self, text=\"Number of Nodes:\")\n        self.nodes\_entry = ctk.CTkEntry(self, textvariable=self.options[\"nodes\"], width=100)\n\n        self.ncpus\_label = ctk.CTkLabel(self, text=\"CPUs per Node (ncpus):\")\n        self.ncpus\_entry = ctk.CTkEntry(self, textvariable=self.options[\"ncpus\"], width=100)\n\n        self.mem\_label = ctk.CTkLabel(self, text=\"Memory per Node (GB):\")\n        self.mem\_entry = ctk.CTkEntry(self, textvariable=self.options[\"mem\"], width=100)\n\n        self.walltime\_label = ctk.CTkLabel(self, text=\"Walltime (HH:MM:SS):\")\n        self.walltime\_entry = ctk.CTkEntry(self, textvariable=self.options[\"walltime\"], width=100)\n\n        self.parallelize\_checkbutton = ctk.CTkCheckBox(\n            self,\n            text=\"Enable Parallelization (MPI)\",\n            variable=self.options[\"parallelize\"],\n            command=self.toggle\_mpi\_options,\n        )\n\n        self.mpiprocs\_label = ctk.CTkLabel(self, text=\"MPI Processes per Node (mpiprocs):\")\n        self.mpiprocs\_entry = ctk.CTkEntry(self, textvariable=self.options[\"mpiprocs\"], width=100)\n\n        self.conda\_path\_label = ctk.CTkLabel(self, text=\"Conda Environment Path:\")\n        self.conda\_path\_entry = ctk.CTkEntry(self, textvariable=self.options[\"conda\_env\_path\"], width=150)\n\n        self.conda\_name\_label = ctk.CTkLabel(self, text=\"Conda Environment Name:\")\n        self.conda\_name\_entry = ctk.CTkEntry(self, textvariable=self.options[\"conda\_env\_name\"], width=150)\n\n        self.hpc\_user\_label = ctk.CTkLabel(self, text=\"HPC User:\")\n        self.hpc\_user\_entry = ctk.CTkEntry(self, textvariable=self.options[\"hpc\_user\"], width=150)\n\n        self.hpc\_host\_label = ctk.CTkLabel(self, text=\"HPC Host:\")\n        self.hpc\_host\_entry = ctk.CTkEntry(self, textvariable=self.options[\"hpc\_host\"], width=150)\n\n        self.hpc\_dir\_label = ctk.CTkLabel(self, text=\"HPC Directory:\")\n        self.hpc\_dir\_entry = ctk.CTkEntry(self, textvariable=self.options[\"hpc\_dir\"], width=150)\n\n        # Initially hide all HPC-related options\n        self.hide\_hpc\_options()\n\n    def toggle\_hpc\_options(self):\n        \"\"\"Show or hide HPC options based on the Enable HPC checkbox.\"\"\"\n        if self.options[\"hpc\_enabled\"].get():\n            # Show HPC-related options\n            self.nodes\_label.grid(row=1, column=0, padx=0, pady=2, sticky=\"w\")\n            self.nodes\_entry.grid(row=1, column=1, padx=(2, 0), pady=2, sticky=\"w\")\n\n            self.ncpus\_label.grid(row=2, column=0, padx=0, pady=2, sticky=\"w\")\n            self.ncpus\_entry.grid(row=2, column=1, padx=(2, 0), pady=2, sticky=\"w\")\n\n            self.mem\_label.grid(row=3, column=0, padx=0, pady=2, sticky=\"w\")\n            self.mem\_entry.grid(row=3, column=1, padx=(2, 0), pady=2, sticky=\"w\")\n\n            self.walltime\_label.grid(row=4, column=0, padx=0, pady=2, sticky=\"w\")\n            self.walltime\_entry.grid(row=4, column=1, padx=(2, 0), pady=2, sticky=\"w\")\n\n            self.parallelize\_checkbutton.grid(row=5, column=0, sticky=\"w\")\n            self.toggle\_mpi\_options()  # Show or hide MPI options based on the checkbox\n            self.hpc\_user\_label.grid(row=1, column=2, sticky=\"w\")\n            self.hpc\_user\_entry.grid(row=1, column=3, sticky=\"w\")\n            self.hpc\_host\_label.grid(row=2, column=2, sticky=\"w\")\n            self.hpc\_host\_entry.grid(row=2, column=3, sticky=\"w\")\n            self.hpc\_dir\_label.grid(row=3, column=2, sticky=\"w\")\n            self.hpc\_dir\_entry.grid(row=3, column=3, sticky=\"w\")\n            self.conda\_path\_label.grid(row=4, column=2, sticky=\"w\")\n            self.conda\_path\_entry.grid(row=4, column=3, sticky=\"w\")\n            self.conda\_name\_label.grid(row=5, column=2, sticky=\"w\")\n            self.conda\_name\_entry.grid(row=5, column=3, sticky=\"w\")\n        else:\n            self.hide\_hpc\_options()\n\n    def hide\_hpc\_options(self):\n        \"\"\"Hide all HPC-related options.\"\"\"\n        self.nodes\_label.grid\_remove()\n        self.nodes\_entry.grid\_remove()\n        self.ncpus\_label.grid\_remove()\n        self.ncpus\_entry.grid\_remove()\n        self.mem\_label.grid\_remove()\n        self.mem\_entry.grid\_remove()\n        self.walltime\_label.grid\_remove()\n        self.walltime\_entry.grid\_remove()\n        self.parallelize\_checkbutton.grid\_remove()\n        self.mpiprocs\_label.grid\_remove()\n        self.mpiprocs\_entry.grid\_remove()\n        self.conda\_path\_label.grid\_remove()\n        self.conda\_path\_entry.grid\_remove()\n        self.conda\_name\_label.grid\_remove()\n        self.conda\_name\_entry.grid\_remove()\n        self.hpc\_user\_label.grid\_remove()\n        self.hpc\_user\_entry.grid\_remove()\n        self.hpc\_host\_label.grid\_remove()\n        self.hpc\_host\_entry.grid\_remove()\n        self.hpc\_dir\_label.grid\_remove()\n        self.hpc\_dir\_entry.grid\_remove()\n\n    def toggle\_mpi\_options(self):\n        if self.options[\"parallelize\"].get():\n            self.mpiprocs\_label.grid(row=6, column=0, sticky=\"w\")\n            self.mpiprocs\_entry.grid(row=6, column=1, sticky=\"w\")\n        else:\n            self.mpiprocs\_label.grid\_remove()\n            self.mpiprocs\_entry.grid\_remove()\n\n    def \_on\_hpc\_enabled\_changed(self, \*args):\n        \"\"\"Callback whenever self.options['hpc\_enabled'] is set (True/False).\"\"\"\n        # Just reuse your existing logic:\n        self.toggle\_hpc\_options()\n"

  },

  {

    "filePath": "../HeatToolbox/src/gui/main\_window.py",

    "fileContent": "# import tkinter as tk\nimport customtkinter as ctk\nimport traceback\nfrom tkinter import messagebox, filedialog\n\nfrom .optimization\_frame import OptimizationFrame\nfrom .material\_frame import MaterialFrame\nfrom .solving\_frame import SolvingFrame\nfrom .visualization\_frame import VisualizationFrame\nfrom .sources\_frame import SourcesFrame\nfrom .hpc\_frame import HPCFrame\n\nfrom sim\_config import SimulationConfig\nfrom main import SimulationController\nfrom hpc.script\_generator import generate\_hpc\_script\nfrom .utils\_config import initialize\_options,  get\_config\_dict, save\_config, load\_config\nfrom hpc.hpc\_utils import submit\_job, prompt\_password\nimport os\nimport json\nimport gmsh\nimport matplotlib.pyplot as plt\nfrom matplotlib.figure import Figure\nfrom matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg\n\n\nclass SimulationConfigGUI(ctk.CTk):\n    def \_\_init\_\_(self):\n        super().\_\_init\_\_()\n        \n        # Set Modern Appearance\n        ctk.set\_appearance\_mode(\"Dark\")  # Modes: \"Light\", \"Dark\", \"System\"\n        ctk.set\_default\_color\_theme(\"blue\")  # Themes: \"blue\", \"dark-blue\", \"green\"\n\n        # Ensure the main window stretches properly but doesn't leave extra space\n        self.grid\_columnconfigure(0, weight=1)\n        self.grid\_rowconfigure(0, weight=1)\n        self.grid\_rowconfigure(1, weight=1)\n        self.grid\_rowconfigure(2, weight=1)\n        self.grid\_rowconfigure(3, weight=1)\n        self.grid\_rowconfigure(4, weight=1)\n        self.grid\_rowconfigure(5, weight=1)  # Ensure the last row stretches\n        self.grid\_rowconfigure(99, weight=0)  # Ensure last row does NOT expand\n\n        # self.root = root\n        # self.root.title(\"Simulation Configuration\")\n        # self.set\_focus()\n\n        # Initialize options\n        self.options = initialize\_options()\n\n        # Initialize frames\n        self.init\_frames()\n\n        # Add buttons\n        self.add\_buttons()\n\n    def set\_focus(self):\n        \"\"\"Ensure the GUI window appears in front of other apps.\"\"\"\n        self.root.lift()\n        self.root.attributes(\"-topmost\", True)\n        self.root.update()\n        self.root.attributes(\"-topmost\", False)\n\n    def init\_frames(self):\n        \"\"\"Initialize and add all frames to the GUI.\"\"\"\n        self.optimization\_frame = OptimizationFrame(self, self.options)\n        self.material\_frame = MaterialFrame(self, self.options)\n        self.solving\_frame = SolvingFrame(self, self.options)\n        self.visualization\_frame = VisualizationFrame(\n            self, self.options, visualize\_options=[\"gamma\", \"temperature\", \"flux\", \"profiles\", \"pregamma\"]\n        )\n        self.sources\_frame = SourcesFrame(self, self.options, self.material\_frame)\n        self.hpc\_frame = HPCFrame(self, self.options)\n\n    def add\_buttons(self):\n        \"\"\"Create modern buttons using CTkButton.\"\"\"\n        button\_frame = ctk.CTkFrame(self)\n        button\_frame.grid(row=3, column=0, columnspan=3, pady=10, padx=10, sticky=\"ew\")\n\n        self.grid\_columnconfigure(0, weight=1)  # Ensure buttons expand evenly\n\n        buttons = [\n            (\"Run Simulation\", self.run\_simulation),\n            (\"Submit to HPC\", self.submit\_to\_hpc),\n            (\"Visualize Best\", self.visualize\_best\_result),\n            (\"Visualize Mesh\", self.visualize\_gmsh\_mesh),  # <--- NEW\n            (\"Save Config\", self.save\_config),\n            (\"Load Config\", self.load\_config),\n        ]\n\n        for idx, (text, command) in enumerate(buttons):\n            ctk.CTkButton(button\_frame, text=text, command=command, width=150, height=40).grid(row=0, column=idx, padx=5, pady=5, sticky=\"ew\")\n\n        # REMOVE EMPTY SPACE BELOW\n        self.grid\_rowconfigure(3, weight=0)  # Fix the bottom row to not expand\n\n    def run\_simulation(self):\n        try:\n            # Extracting the options\n            # config = get\_config\_dict(self.options)\n            log\_name = self.options[\"log\_name\"].get() or \"default\_log\"\n            log\_dir = os.path.join(\"logs\", log\_name)\n            os.makedirs(log\_dir, exist\_ok=True)\n            save\_config(self.options, log\_dir)\n            config\_path = os.path.join(log\_dir, \"config.json\")\n\n            # Initialize SimulationConfig with the extracted args\n            config = SimulationConfig(config\_path)\n            sim = SimulationController(config)\n\n            # Run the simulation\n            sim.run\_simulation()\n\n        except Exception as e:\n            traceback.print\_exc()\n            messagebox.showerror(\"Error\", str(e))\n\n    def submit\_to\_hpc(self):\n        try:\n            # Generate HPC script\n            config = get\_config\_dict(self.options)\n\n            log\_name = self.options[\"log\_name\"].get() or \"default\_log\"\n            log\_dir = os.path.join(\"logs\", log\_name)\n            os.makedirs(log\_dir, exist\_ok=True)\n            save\_config(self.options, log\_dir)\n            config\_path = os.path.join(log\_dir, \"config.json\")\n\n            script\_content = generate\_hpc\_script(config, config\_path)\n            self.save\_hpc\_script(script\_content, log\_dir)\n\n            # Prompt for password and submit the job\n            password = prompt\_password()\n            hpc\_user = self.options[\"hpc\_user\"].get()\n            hpc\_host = self.options[\"hpc\_host\"].get()\n            hpc\_path = self.options[\"hpc\_dir\"].get()\n            submit\_job(hpc\_user, hpc\_host, hpc\_path, log\_dir, password)\n\n        except Exception as e:\n            traceback.print\_exc()\n            messagebox.showerror(\"Error\", str(e))\n\n    def save\_hpc\_script(self, hpc\_script, log\_dir):\n        \"\"\"Save the HPC script in the log directory.\"\"\"\n        # Save HPC script\n        hpc\_path = os.path.join(log\_dir, \"hpc\_run.sh\")\n        with open(hpc\_path, \"w\") as f:\n            f.write(hpc\_script)\n\n    def save\_config(self):\n        log\_name = self.options[\"log\_name\"].get() or \"default\_log\"\n        log\_dir = os.path.join(\"logs\", log\_name)\n        os.makedirs(log\_dir, exist\_ok=True)\n        save\_config(self.options, log\_dir)\n\n    def load\_config(self):\n        file\_path = filedialog.askopenfilename(defaultextension=\".json\", filetypes=[(\"JSON files\", \"\*.json\")])\n        if file\_path:\n            load\_config(self.options, file\_path)\n\n    def visualize\_best\_result(self):\n        \"\"\"Load config, modify it, and run simulation on the best latent vector.\"\"\"\n        # Ask the user to select a config file\n        file\_path = filedialog.askopenfilename(\n            title=\"Select Configuration File\",\n            filetypes=[(\"JSON files\", \"\*.json\"), (\"All files\", \"\*.\*\")]\n        )\n        if file\_path:\n            try:\n                # Load the configuration from the selected file\n                with open(file\_path, \"r\") as f:\n                    config = json.load(f)\n\n                # Modify the configuration in memory\n                config[\"optim\"] = False\n                config[\"hpc\_enabled\"] = False\n                # Turn all visualization options on\n                if \"visualize\" in config:\n                    for key in config[\"visualize\"]:\n                        config[\"visualize\"][key] = True\n                else:\n                    config[\"visualize\"] = {\n                        \"gamma\": True,\n                        \"temperature\": True,\n                        \"flux\": True,\n                        \"profiles\": True,\n                        \"pregamma\": True\n                    }\n\n                # # Ensure we have the log directory\n                # if \"log\_name\" in config:\n                #     log\_dir = os.path.join(\"logs\", config[\"log\_name\"])\n                # else:\n                #     messagebox.showerror(\"Error\", \"The configuration file must contain 'log\_name'.\")\n                #     return\n\n                config[\"load\_cma\_result\"] = True\n\n                # Update options in the GUI (optional, if we want to reflect changes)\n                # self.options = {}  # Reset options\n                # set\_options\_from\_config(self.options, config)\n\n                # Create a SimulationConfig object\n                simulation\_config = SimulationConfig(config)\n\n                # Create a SimulationController and run the simulation\n                controller = SimulationController(simulation\_config)\n                controller.run\_simulation()\n\n                # Inform the user\n                messagebox.showinfo(\"Success\", \"Simulation completed and visualizations saved.\")\n\n            except Exception as e:\n                messagebox.showerror(\"Error\", f\"An error occurred: {e}\")\n\n    def visualize\_gmsh\_mesh(self):\n            \"\"\"\n            Example method to build or load the Gmsh geometry,\n            extract boundary lines and color them in a Matplotlib figure\n            embedded in Tkinter.\n            \"\"\"\n            try:\n                # 1) Build or load your Gmsh model using your config\n                #    (You could also call your 'sym\_create\_mesh' or 'create\_mesh' \n                #    but \*do not finalize gmsh yet\*, because we want to query it.)\n                \n                gmsh.initialize()\n                gmsh.model.add(\"visual\_demo\")\n\n                # For illustration, let's do something simpler:\n                # We'll just add a rectangle + a circle or so\n                # Or call your existing geometry builder. For example:\n                # my\_mesh\_gen = MeshGenerator(self.options)  # if that’s how you do it\n                # my\_mesh\_gen.sym\_create\_mesh()              # but skip finalizing\n                #\n                # Instead, as a self-contained example:\n                p0 = gmsh.model.geo.addPoint(0, 0, 0)\n                p1 = gmsh.model.geo.addPoint(2, 0, 0)\n                p2 = gmsh.model.geo.addPoint(2, 1, 0)\n                p3 = gmsh.model.geo.addPoint(0, 1, 0)\n                l0 = gmsh.model.geo.addLine(p0, p1)\n                l1 = gmsh.model.geo.addLine(p1, p2)\n                l2 = gmsh.model.geo.addLine(p2, p3)\n                l3 = gmsh.model.geo.addLine(p3, p0)\n                loop = gmsh.model.geo.addCurveLoop([l0,l1,l2,l3])\n                s = gmsh.model.geo.addPlaneSurface([loop])\n                # Physical tags\n                gmsh.model.addPhysicalGroup(1, [l0], tag=1)\n                gmsh.model.setPhysicalName(1, 1, \"Bottom\")\n                gmsh.model.addPhysicalGroup(1, [l1], tag=2)\n                gmsh.model.setPhysicalName(1, 2, \"Right\")\n                gmsh.model.addPhysicalGroup(1, [l2,l3], tag=3)\n                gmsh.model.setPhysicalName(1, 3, \"OtherBoundary\")\n                gmsh.model.addPhysicalGroup(2, [s], tag=1)\n                gmsh.model.setPhysicalName(2, 1, \"Domain\")\n\n                gmsh.model.geo.synchronize()\n                gmsh.model.mesh.generate(2)\n\n                # 2) Extract boundary line data for each Physical Group\n                dim = 1  # lines\n                phys\_groups = gmsh.model.getPhysicalGroups(dim) \n                # => [(1, tag1), (1, tag2), ...]\n\n                # We'll store the lines (list of segments) and each segment's color / name\n                boundary\_segments = []\n                color\_map = {}\n                color\_cycle = plt.cm.tab10.colors  # or any color palette\n                color\_index = 0\n\n                for (d, group\_tag) in phys\_groups:\n                    # fetch the entity tags (line IDs) in this group\n                    line\_ids = gmsh.model.getEntitiesForPhysicalGroup(d, group\_tag)\n                    # get the group name\n                    group\_name = gmsh.model.getPhysicalName(d, group\_tag)\n                    # pick a color from the color cycle\n                    this\_color = color\_cycle[color\_index % len(color\_cycle)]\n                    color\_index += 1\n                    color\_map[group\_tag] = (group\_name, this\_color)\n\n                    # For each line\_id, get the node coordinates in 2D\n                    for lid in line\_ids:\n                        # get mesh nodes for this line\n                        node\_tags, node\_coords, \_ = gmsh.model.mesh.getNodes(dim=d, tag=lid)\n                        # node\_coords is a flat list: [x1,y1,z1, x2,y2,z2, ...]\n                        # The line connectivity\n                        elem\_types, elem\_tags, elem\_node\_tags = gmsh.model.mesh.getElements(dim=d, tag=lid)\n                        # Typically there's 1 element type (e.g. 1D line segments):\n                        # we can parse elem\_node\_tags[0] in pairs (or in order) to figure out each segment\n\n                        # But for a simple line, it might be just 1 segment with 2 endpoints.\n                        # We'll parse them thoroughly in case it's subdivided for the mesh.\n\n                        # Build a mapping from nodeTag -> (x,y)\n                        node\_map = {}\n                        for i, ntag in enumerate(node\_tags):\n                            x = node\_coords[3\*i]\n                            y = node\_coords[3\*i+1]\n                            node\_map[ntag] = (x, y)\n\n                        # Now get the connectivity\n                        # e.g. for 2-node line segments, each element has 2 node tags\n                        econn = elem\_node\_tags[0]  # the actual node tags in the element\n                        # group them in pairs\n                        pairs = [econn[i:i+2] for i in range(0, len(econn), 2)]\n                        for (nd1, nd2) in pairs:\n                            x1, y1 = node\_map[nd1]\n                            x2, y2 = node\_map[nd2]\n                            boundary\_segments.append({\n                                \"group\_tag\": group\_tag,\n                                \"x\": [x1, x2],\n                                \"y\": [y1, y2],\n                            })\n\n                # 3) Now gmsh.finalize() if you like\n                gmsh.finalize()\n\n                # 4) Create a Matplotlib Figure, plot each segment in its group color\n                fig = Figure(figsize=(5, 4), dpi=100)\n                ax = fig.add\_subplot(111)\n                ax.set\_title(\"Gmsh Boundary Visualization\")\n                ax.set\_aspect(\"equal\", \"box\")\n\n                for seg in boundary\_segments:\n                    grp = seg[\"group\_tag\"]\n                    (grp\_name, c) = color\_map[grp]\n                    ax.plot(seg[\"x\"], seg[\"y\"], color=c, lw=2, label=grp\_name)\n\n                # Because multiple segments from the same group can appear,\n                # we don't want a repeating label in the legend. Let's do a trick:\n                # get existing labels, only label the first segment of each group.\n                handles, labels = ax.get\_legend\_handles\_labels()\n                # remove duplicates\n                unique = dict()\n                for h, l in zip(handles, labels):\n                    if l not in unique:\n                        unique[l] = h\n                ax.legend(unique.values(), unique.keys(), loc='best')\n\n                # 5) Embed this figure in a Tkinter Toplevel or inside your main window\n                top = ctk.CTkToplevel(self)\n                top.title(\"Mesh Visualization\")\n\n                canvas = FigureCanvasTkAgg(fig, master=top)  \n                canvas.draw()\n                canvas.get\_tk\_widget().pack(side=\"top\", fill=\"both\", expand=True)\n\n            except Exception as e:\n                traceback.print\_exc()\n                messagebox.showerror(\"Error\", str(e))"

  },

  {

    "filePath": "../HeatToolbox/src/gui/material\_frame.py",

    "fileContent": "import customtkinter as ctk\nimport tkinter as tk\nimport os\n\n\nclass MaterialFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options):\n        super().\_\_init\_\_(parent)\n        self.options = options\n        self.grid(row=0, column=1, padx=10, pady=5, sticky=\"nsew\")\n        self.create\_widgets()\n\n    def create\_widgets(self):\n        # Load CMA-ES Config Button\n        row = 0\n        # select solver type, gke vs fourier\n        ctk.CTkLabel(self, text=\"Solver Type\").grid(row=row, column=0, sticky=\"w\")\n        self.solver\_type\_menu = ctk.CTkOptionMenu(\n            self,\n            variable=self.options[\"solver\_type\"],\n            values=[\"gke\", \"fourier\", \"joule\"],\n            command=self.update\_solver\_type,\n        ).grid(row=row, column=1, padx=5, pady=5, sticky=\"ew\")\n\n        # write knudsen number\n        row += 1\n        ctk.CTkLabel(self, text=\"Knudsen Number\").grid(row=row, column=0, sticky=\"w\")\n        ctk.CTkEntry(self, textvariable=self.options[\"knudsen\"], width=100).grid(\n            row=row, column=1\n        )\n\n        # latent size\n        row += 1\n        ctk.CTkLabel(self, text=\"Latent Size\").grid(row=row, column=0, sticky=\"w\")\n        ctk.CTkOptionMenu(\n            self,\n            variable=self.options[\"latent\_size\"],\n            values=[\"2\", \"4\", \"8\", \"16\"],  # Convert numbers to strings\n            command=self.update\_latent\_size,\n        ).grid(row=row, column=1, padx=5, pady=5, sticky=\"ew\")\n\n        # Latent Method\n        row += 1\n        ctk.CTkLabel(self, text=\"Latent Method\").grid(row=row, column=0, sticky=\"w\")\n        ctk.CTkOptionMenu(\n            self,\n            variable=self.options[\"latent\_method\"],\n            values=[\"manual\", \"random\", \"preloaded\"],\n            command=self.update\_latent\_method,\n        ).grid(row=row, column=1, padx=5, pady=5, sticky=\"ew\")\n        self.options[\"latent\_method\"].set(\"preloaded\")  # Set a default value\n\n        # Latent Values Entry (will be updated based on latent size and method)\n        row += 1\n        self.latent\_frame = ctk.CTkFrame(self)\n        self.latent\_frame.grid(row=row, column=0, columnspan=2, sticky=\"w\")\n        self.update\_latent\_method()  # Initialize latent entries\n\n        row += 1\n        # \"Enable Symmetry\" Checkbutton (moved to row 3)\n        ctk.CTkCheckBox(\n            self, text=\"Enable Symmetry\", variable=self.options[\"symmetry\"]\n        ).grid(row=row, column=0, sticky=\"w\")\n\n        row += 1\n        # \"Run with Blank Image\" Checkbutton (moved to row 4)\n        ctk.CTkCheckBox(\n            self, text=\"Run with Blank Image\", variable=self.options[\"blank\"]\n        ).grid(row=row, column=0, sticky=\"w\")\n\n        row += 1\n        # \"Sources (Position, Heat)\" Label (moved to row 5)\n        ctk.CTkLabel(self, text=\"Sources (Position, Heat)\").grid(\n            row=row, column=0, sticky=\"w\"\n        )\n        self.row = row\n\n    def update\_latent\_size(self, choice):\n        \"\"\"Update the latent variables and entries when the latent size changes.\"\"\"\n        size = int(choice)  # Convert from string to integer\n        self.options[\"latent\_size\"].set(size)  # Update the variable\n        self.options[\"latent\"] = [tk.DoubleVar() for \_ in range(size)]\n        self.update\_latent\_entries()\n\n    def update\_latent\_method(self, \*args):\n        \"\"\"Update the visibility of latent entries based on the selected method.\"\"\"\n        self.update\_latent\_entries()\n\n    def update\_latent\_entries(self):\n        \"\"\"Create or update the latent entries based on the latent size and method.\"\"\"\n        # Clear the current latent entries\n        for widget in self.latent\_frame.winfo\_children():\n            widget.destroy()\n\n        method = self.options[\"latent\_method\"].get()\n        if method == \"manual\":\n            ctk.CTkLabel(self.latent\_frame, text=\"Latent Values\").grid(\n                row=0, column=0, sticky=\"w\"\n            )\n            latent\_values\_frame = ctk.CTkFrame(self.latent\_frame)\n            latent\_values\_frame.grid(\n                row=0, column=1, columnspan=self.options[\"latent\_size\"].get()\n            )\n            for i in range(self.options[\"latent\_size\"].get()):\n                ctk.CTkEntry(\n                    latent\_values\_frame, textvariable=self.options[\"latent\"][i], width=100\n                ).grid(row=0, column=i)\n        elif method == \"random\":\n            ctk.CTkLabel(\n                self.latent\_frame, text=\"Latent vector will be randomly generated\"\n            ).grid(row=0, column=0, sticky=\"w\")\n        elif method == \"preloaded\":\n            ctk.CTkLabel(\n                self.latent\_frame,\n                text=\"Latent vector will be loaded from 'best\_latent\_vector.npy'\",\n            ).grid(row=0, column=0, sticky=\"w\")\n\n    def update\_solver\_type(self, \*args):\n        solver\_type = self.options[\"solver\_type\"].get()\n\n        # Update defaults based on solver type\n        if solver\_type == \"gke\":\n            self.options[\"knudsen\"].set(1.0)\n            # Additional GKE-specific defaults\n        elif solver\_type == \"fourier\":\n            self.options[\"knudsen\"].set(None)  # Not used for Fourier\n            # Fourier-specific defaults\n        elif solver\_type == \"joule\":\n            self.options[\"knudsen\"].set(None)  # Not applicable\n            # Add Joule-specific defaults here\n\n        # Trigger updates for dependent widgets\n        self.update\_latent\_entries()\n"

  },

  {

    "filePath": "../HeatToolbox/src/gui/optimization\_frame.py",

    "fileContent": "import customtkinter as ctk\n\n\nclass OptimizationFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options):\n        super().\_\_init\_\_(parent)\n        self.options = options\n\n        self.grid(row=0, column=0, padx=10, pady=5, sticky=\"nsew\")\n        self.grid\_columnconfigure((0, 1), weight=1)\n        self.grid\_rowconfigure(99, weight=1)  # Prevent bottom empty space\n\n        self.create\_widgets()\n        # Automatically update UI when \"optim\" checkbox is changed\n        self.options[\"optim\"].trace\_add(\"write\", self.\_on\_optim\_changed)\n\n    def create\_widgets(self):\n        \"\"\"Create UI elements with CustomTkinter.\"\"\"\n        # Run Optimization Checkbox\n        self.optim\_checkbox = ctk.CTkCheckBox(\n            self, text=\"Run Optimization\", variable=self.options[\"optim\"]\n        )\n        self.optim\_checkbox.grid(row=0, column=0, columnspan=2, pady=(5, 10), sticky=\"w\")\n\n        # Optimizer Selection\n        self.optimizer\_label = ctk.CTkLabel(self, text=\"Select Optimizer:\")\n        self.optimizer\_menu = ctk.CTkOptionMenu(\n            self, variable=self.options[\"optimizer\"], values=[\"bayesian\", \"cmaes\"]\n        )\n\n        # Population Size\n        self.popsize\_label = ctk.CTkLabel(self, text=\"Population Size:\")\n        self.popsize\_entry = ctk.CTkEntry(self, textvariable=self.options[\"popsize\"], width=100)\n\n        # Bounds\n        self.bounds\_lower\_label = ctk.CTkLabel(self, text=\"Lower Bound:\")\n        self.bounds\_lower\_entry = ctk.CTkEntry(self, textvariable=self.options[\"bounds\_lower\"], width=100)\n\n        self.bounds\_upper\_label = ctk.CTkLabel(self, text=\"Upper Bound:\")\n        self.bounds\_upper\_entry = ctk.CTkEntry(self, textvariable=self.options[\"bounds\_upper\"], width=100)\n\n        # Number of Iterations\n        self.n\_iter\_label = ctk.CTkLabel(self, text=\"Number of Iterations:\")\n        self.n\_iter\_entry = ctk.CTkEntry(self, textvariable=self.options[\"n\_iter\"], width=100)\n\n        # Initially hide all optimizer-related options\n        self.hide\_optimizer\_options()\n\n    def toggle\_optimizer\_options(self):\n        \"\"\"Show or hide optimizer options based on the Run Optimization checkbox.\"\"\"\n        if self.options[\"optim\"].get():\n            # Show optimizer-related options\n            self.optimizer\_label.grid(row=1, column=0, sticky=\"w\")\n            self.optimizer\_menu.grid(row=1, column=1, padx=5, pady=5, sticky=\"w\")\n            self.popsize\_label.grid(row=2, column=0, sticky=\"w\")\n            self.popsize\_entry.grid(row=2, column=1, sticky=\"w\")\n            self.bounds\_lower\_label.grid(row=3, column=0, sticky=\"w\")\n            self.bounds\_lower\_entry.grid(row=3, column=1, sticky=\"w\")\n            self.bounds\_upper\_label.grid(row=4, column=0, sticky=\"w\")\n            self.bounds\_upper\_entry.grid(row=4, column=1, sticky=\"w\")\n            self.n\_iter\_label.grid(row=5, column=0, sticky=\"w\")\n            self.n\_iter\_entry.grid(row=5, column=1, sticky=\"w\")\n        else:\n            self.hide\_optimizer\_options()\n\n    def hide\_optimizer\_options(self):\n        \"\"\"Hide all optimizer-related options.\"\"\"\n        self.optimizer\_label.grid\_remove()\n        self.optimizer\_menu.grid\_remove()\n        self.popsize\_label.grid\_remove()\n        self.popsize\_entry.grid\_remove()\n        self.bounds\_lower\_label.grid\_remove()\n        self.bounds\_lower\_entry.grid\_remove()\n        self.bounds\_upper\_label.grid\_remove()\n        self.bounds\_upper\_entry.grid\_remove()\n        self.n\_iter\_label.grid\_remove()\n        self.n\_iter\_entry.grid\_remove()\n    \n    def \_on\_optim\_changed(self, \*args):\n        \"\"\"Callback whenever self.options['hpc\_enabled'] is set (True/False).\"\"\"\n        # Just reuse your existing logic:\n        self.toggle\_optimizer\_options()\n"

  },

  {

    "filePath": "../HeatToolbox/src/gui/solving\_frame.py",

    "fileContent": "import customtkinter as ctk\n\n\nclass SolvingFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options):\n        super().\_\_init\_\_(parent)\n        self.options = options\n        self.grid(row=1, column=0, padx=10, pady=5, sticky=\"nsew\")\n        self.create\_widgets()\n\n    def create\_widgets(self):\n        ctk.CTkLabel(self, text=\"Mesh Resolution: Length / [] \").grid(\n            row=0, column=0, sticky=\"w\"\n        )\n        ctk.CTkEntry(self, textvariable=self.options[\"res\"], width=100).grid(\n            row=0, column=1\n        )\n\n        ctk.CTkCheckBox(\n            self,\n            text=\"Enable Volume Fraction Control\",\n            variable=self.options[\"vf\_enabled\"],\n            command=self.toggle\_volume\_fraction,\n        ).grid(row=1, column=0, sticky=\"w\")\n\n        ctk.CTkLabel(self, text=\"Volume Fraction\").grid(\n            row=2, column=0, sticky=\"w\"\n        )\n        self.vf\_entry = ctk.CTkEntry(self, textvariable=self.options[\"vf\_value\"], width=100)\n        self.vf\_entry.grid(row=2, column=1)\n        self.toggle\_volume\_fraction()\n\n        # Logging Toggle\n        ctk.CTkCheckBox(\n            self,\n            text=\"Enable Logging\",\n            variable=self.options[\"logging\_enabled\"],\n        ).grid(row=3, column=0, sticky=\"w\")\n\n        ctk.CTkLabel(self, text=\"Log File Name\").grid(row=4, column=0, sticky=\"w\")\n        ctk.CTkEntry(self, textvariable=self.options[\"log\_name\"], width=100).grid(row=4, column=1, sticky=\"w\")\n\n    def toggle\_volume\_fraction(self):\n        if self.options[\"vf\_enabled\"].get():\n            self.vf\_entry.configure(state=\"normal\")  #  Use configure() instead of config()\n        else:\n            self.vf\_entry.configure(state=\"disabled\")  #  Correct way to disable"

  },

  {

    "filePath": "../HeatToolbox/src/gui/sources\_frame.py",

    "fileContent": "import tkinter as tk\nimport customtkinter as ctk\n\n\nclass SourcesFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options, material\_frame):\n        super().\_\_init\_\_(parent)\n        self.options = options\n        self.material\_frame = material\_frame\n        self.row = self.material\_frame.row\n        # self.grid(row=2, column=1, columnspan=4, sticky=\"w\")\n        # Store this frame in options so it can be accessed in utils\_config.py\n        self.options[\"sources\_frame\"] = self\n\n        self.create\_widgets()\n        self.add\_source\_row()\n\n    def create\_widgets(self):\n        ctk.CTkLabel(self.material\_frame, text=\"Sources (Position, Heat)\").grid(\n            row=self.row, column=0, sticky=\"w\"\n        )\n        self.sources\_frame = ctk.CTkFrame(self.material\_frame)\n        self.sources\_frame.grid(row=self.row + 1, column=1, columnspan=4, sticky=\"w\")\n\n        ctk.CTkButton(\n            self.material\_frame,\n            text=\"Add Source\",\n            command=self.add\_source\_row,\n        ).grid(row=self.row + 2, column=0, sticky=\"w\")\n\n    def add\_source\_row(self):\n        source\_row = {}\n        row\_frame = ctk.CTkFrame(self.sources\_frame)\n        row\_frame.pack(anchor=\"w\", pady=2)\n\n        # Position Entry\n        ctk.CTkLabel(row\_frame, text=\"Position:\").pack(side=\"left\")\n        position\_var = tk.DoubleVar(value=0.5)\n        ctk.CTkEntry(row\_frame, textvariable=position\_var, width=50).pack(side=\"left\")\n\n        # Heat Value Entry\n        ctk.CTkLabel(row\_frame, text=\"Heat:\").pack(side=\"left\")\n        heat\_var = tk.DoubleVar(value=80)\n        ctk.CTkEntry(row\_frame, textvariable=heat\_var, width=50).pack(side=\"left\")\n\n        # Remove Button\n        remove\_button = ctk.CTkButton(\n            row\_frame, text=\"Remove\", command=lambda: self.remove\_source\_row(source\_row)\n        )\n        remove\_button.pack(side=\"left\", padx=5)\n\n        # Store references\n        source\_row[\"frame\"] = row\_frame\n        source\_row[\"position\"] = position\_var\n        source\_row[\"heat\"] = heat\_var\n        source\_row[\"remove\_button\"] = remove\_button\n\n        self.options[\"sources\"].append(source\_row)\n\n    def remove\_source\_row(self, source\_row):\n        source\_row[\"frame\"].destroy()\n        self.options[\"sources\"].remove(source\_row)\n"

  },

  {

    "filePath": "../HeatToolbox/src/gui/utils\_config.py",

    "fileContent": "import tkinter as tk\nfrom tkinter import messagebox\nimport os\nimport json\n\n\ndef initialize\_options():\n    \"\"\"Initialize all Tkinter options with default values.\"\"\"\n    return {\n        \"optim\": tk.BooleanVar(),\n        \"optimizer\": tk.StringVar(value=\"cmaes\"),\n        \"latent\_size\": tk.IntVar(value=4),\n        \"latent\_method\": tk.StringVar(value=\"preloaded\"),\n        \"latent\": [tk.DoubleVar() for \_ in range(4)],\n        \"symmetry\": tk.BooleanVar(),\n        \"blank\": tk.BooleanVar(),\n        \"sources\": [],\n        \"res\": tk.DoubleVar(value=12),\n        \"visualize\": {},\n        \"vf\_enabled\": tk.BooleanVar(value=True),\n        \"vf\_value\": tk.DoubleVar(value=0.2),\n        \"plot\_mode\": tk.StringVar(value=\"screenshot\"),\n        \"logging\_enabled\": tk.BooleanVar(value=True),\n        # HPC options\n        \"nodes\": tk.IntVar(value=1),\n        \"ncpus\": tk.IntVar(value=4),\n        \"mem\": tk.IntVar(value=8),\n        \"walltime\": tk.StringVar(value=\"03:00:00\"),\n        \"parallelize\": tk.BooleanVar(value=False),\n        \"mpiprocs\": tk.IntVar(value=4),\n        \"conda\_env\_path\": tk.StringVar(value=\"~/miniforge3/bin/conda\"),\n        \"conda\_env\_name\": tk.StringVar(value=\"fenicsx\_torch\"),\n        \"log\_name\": tk.StringVar(value=\"\"),\n        \"hpc\_user\": tk.StringVar(value=\"pt721\"),\n        \"hpc\_host\": tk.StringVar(value=\"login.cx3.hpc.ic.ac.uk\"),\n        \"hpc\_dir\": tk.StringVar(value=\"~/BTE-NO\"),\n        # CMAES\n        \"popsize\": tk.IntVar(value=8),  # New parameter for population size\n        \"bounds\_lower\": tk.DoubleVar(value=-2.5),  # New lower bound for CMA-ES\n        \"bounds\_upper\": tk.DoubleVar(value=2.5),  # New upper bound for CMA-ES\n        \"n\_iter\": tk.IntVar(value=100),  # New parameter for the number of iterations\n        \"hpc\_enabled\": tk.BooleanVar(value=False),\n        \"load\_cmaes\_config\": tk.BooleanVar(value=False),\n        # \"timeout\": tk.StringVar(),\n        \"knudsen\": tk.DoubleVar(value=1),\n        \"solver\_type\": tk.StringVar(value=\"gke\"),\n    }\n\n\ndef get\_config\_dict(options):\n    \"\"\"Extract configuration from Tkinter options.\"\"\"\n    return {\n        \"optim\": options[\"optim\"].get(),\n        \"optimizer\": options[\"optimizer\"].get(),\n        \"latent\_size\": options[\"latent\_size\"].get(),\n        \"latent\_method\": options[\"latent\_method\"].get(),\n        \"latent\": [val.get() for val in options[\"latent\"]],\n        \"symmetry\": options[\"symmetry\"].get(),\n        \"blank\": options[\"blank\"].get(),\n        \"sources\": parse\_sources(options[\"sources\"]),\n        \"res\": options[\"res\"].get(),\n        \"visualize\": {k: v.get() for k, v in options[\"visualize\"].items()},  # Convert BooleanVars to actual values\n        \"vf\_enabled\": options[\"vf\_enabled\"].get(),\n        \"vf\_value\": options[\"vf\_value\"].get(),\n        \"plot\_mode\": options[\"plot\_mode\"].get(),\n        \"logging\_enabled\": options[\"logging\_enabled\"].get(),\n        \"nodes\": options[\"nodes\"].get(),\n        \"ncpus\": options[\"ncpus\"].get(),\n        \"mem\": options[\"mem\"].get(),\n        \"walltime\": options[\"walltime\"].get(),\n        # \"timeout\": options[\"timeout\"].get(),\n        \"parallelize\": options[\"parallelize\"].get(),\n        \"mpiprocs\": options[\"mpiprocs\"].get(),\n        \"conda\_env\_path\": options[\"conda\_env\_path\"].get(),\n        \"conda\_env\_name\": options[\"conda\_env\_name\"].get(),\n        \"log\_name\": options[\"log\_name\"].get(),\n        \"hpc\_user\": options[\"hpc\_user\"].get(),\n        \"hpc\_host\": options[\"hpc\_host\"].get(),\n        \"hpc\_dir\": options[\"hpc\_dir\"].get(),\n        \"popsize\": options[\"popsize\"].get(),  # New parameter\n        \"bounds\": [options[\"bounds\_lower\"].get(), options[\"bounds\_upper\"].get()],  # New parameter\n        \"n\_iter\": options[\"n\_iter\"].get(),  # New parameter\n        \"hpc\_enabled\": options[\"hpc\_enabled\"].get(),\n        \"load\_cmaes\_config\": options[\"load\_cmaes\_config\"].get(),\n        \"knudsen\": options[\"knudsen\"].get(),\n        \"solver\_type\": options[\"solver\_type\"].get(),\n    }\n\n\ndef save\_config(options, log\_dir):\n    \"\"\"Save the current configuration to a JSON file.\"\"\"\n    print(\"SAVING CONFIG\")\n    config = get\_config\_dict(options)\n    os.makedirs(log\_dir, exist\_ok=True)\n    config\_path = os.path.join(log\_dir, \"config.json\")\n    with open(config\_path, \"w\") as f:\n        json.dump(config, f, indent=4)\n    messagebox.showinfo(\"Success\", f\"Configuration saved to {config\_path}\")\n\n\ndef load\_config(options, file\_path):\n    \"\"\"Load configuration from a JSON file and update the options.\"\"\"\n    with open(file\_path, \"r\") as f:\n        config = json.load(f)\n    set\_options\_from\_config(options, config)\n    messagebox.showinfo(\"Success\", f\"Configuration loaded from {file\_path}\")\n\n\ndef set\_options\_from\_config(options, config):\n    \"\"\"Update Tkinter options with values from a configuration.\"\"\"\n    for key, value in config.items():\n        if key in options:\n            if key == \"sources\":  # Special handling for sources\n                update\_sources\_from\_config(options, value)\n            elif isinstance(options[key], list):  # Handle lists like \"latent\"\n                if isinstance(value, list):\n                    for i, val in enumerate(value):\n                        if i < len(options[key]) and hasattr(options[key][i], 'set'):\n                            options[key][i].set(val)\n                else:\n                    raise ValueError(f\"Expected a list for key '{key}', but got {type(value)}.\")\n            elif isinstance(options[key], dict):  # Handle dictionaries like \"visualize\"\n                if isinstance(value, dict):\n                    for subkey, subvalue in value.items():\n                        if subkey in options[key]:\n                            if hasattr(options[key][subkey], 'set'):\n                                options[key][subkey].set(subvalue)\n                            else:\n                                options[key][subkey] = subvalue\n                        else:\n                            # Handle new keys that might not exist in options\n                            options[key][subkey] = tk.BooleanVar(value=subvalue)\n                else:\n                    raise ValueError(f\"Expected a dict for key '{key}', but got {type(value)}.\")\n            elif hasattr(options[key], 'set'):  # For Tkinter variables\n                options[key].set(value)\n            else:  # Handle non-Tkinter variables (e.g., plain dicts or other types)\n                options[key] = value\n        else:\n            # Handle new keys that might not exist in options\n            options[key] = value\n\n\ndef parse\_sources(tk\_sources\_dict):\n    sources\_list = []\n    for source in tk\_sources\_dict:\n        pos = source[\"position\"].get()\n        heat = source[\"heat\"].get()\n        if pos == \"\" or heat == \"\":\n            continue  # Skip empty entries\n        try:\n            pos = float(pos)\n            heat = float(heat)\n            if pos < 0 or pos > 1:\n                raise ValueError(\"Source positions must be between 0 and 1 (normalized).\")\n            sources\_list.extend([pos, heat])\n        except ValueError as e:\n            raise ValueError(f\"Invalid source input: {e}\")\n    return sources\_list if sources\_list else None\n\n\ndef update\_sources\_from\_config(options, sources\_list):\n    \"\"\"Ensure the correct number of source rows are added to match the loaded config.\"\"\"\n    sources\_frame = options[\"sources\_frame\"]  # This should be a reference to the GUI's SourcesFrame\n\n    # Clear all existing sources\n    for source in options[\"sources\"]:\n        sources\_frame.remove\_source\_row(source)\n    \n    options[\"sources\"].clear()  # Reset source list\n    \n    # Add sources from the loaded config\n    for i in range(0, len(sources\_list), 2):  # Assuming sources are stored as [pos, heat, pos, heat, ...]\n        if i + 1 < len(sources\_list):\n            position, heat = sources\_list[i], sources\_list[i + 1]\n            sources\_frame.add\_source\_row()  # Add a new source row dynamically\n\n            # Update the last added source with correct values\n            options[\"sources\"][-1][\"position\"].set(position)\n            options[\"sources\"][-1][\"heat\"].set(heat)\n"

  },

  {

    "filePath": "../HeatToolbox/src/gui/visualization\_frame.py",

    "fileContent": "import customtkinter as ctk\nimport tkinter as tk\n\n\nclass VisualizationFrame(ctk.CTkFrame):\n    def \_\_init\_\_(self, parent, options, visualize\_options):\n        super().\_\_init\_\_(parent)\n        self.options = options\n        self.visualize\_options = visualize\_options\n        self.grid(row=1, column=1, padx=10, pady=5, sticky=\"nsew\")\n        self.create\_widgets()\n\n    def create\_widgets(self):\n        # Visualization checkboxes\n        ctk.CTkLabel(self, text=\"Visualization Options\").grid(row=0, column=0, sticky=\"nw\")\n        self.visualize\_frame = ctk.CTkFrame(self)\n        self.visualize\_frame.grid(row=0, column=1, sticky=\"w\")\n\n        for option in self.visualize\_options:\n            self.options[\"visualize\"][option] = tk.BooleanVar()\n            ctk.CTkCheckBox(\n                self.visualize\_frame,\n                text=option,\n                variable=self.options[\"visualize\"][option],\n                command=self.update\_plot\_mode\_visibility,  # Attach callback\n            ).pack(anchor=\"w\")\n\n        # Plotting mode\n        ctk.CTkLabel(self, text=\"Plotting Mode\").grid(row=1, column=0, sticky=\"w\")\n        self.plot\_mode\_frame = ctk.CTkFrame(self)\n        self.plot\_mode\_frame.grid(row=1, column=1, sticky=\"w\")\n        self.plot\_mode\_frame.grid\_remove()  # Hidden initially\n\n        ctk.CTkRadioButton(\n            self.plot\_mode\_frame,\n            text=\"Save Screenshots\",\n            variable=self.options[\"plot\_mode\"],\n            value=\"screenshot\",\n        ).grid(row=1, column=1, sticky=\"w\")\n\n        ctk.CTkRadioButton(\n            self.plot\_mode\_frame,\n            text=\"Interactive Plotting\",\n            variable=self.options[\"plot\_mode\"],\n            value=\"interactive\",\n        ).grid(row=2, column=1, sticky=\"w\")\n\n    def update\_plot\_mode\_visibility(self):\n        \"\"\"Show or hide the plot mode options based on visualization selections.\"\"\"\n        # Check if any visualization option is selected\n        any\_selected = any(var.get() for var in self.options[\"visualize\"].values())\n\n        if any\_selected:\n            self.plot\_mode\_frame.grid()  # Show the plotting mode options\n        else:\n            self.plot\_mode\_frame.grid\_remove()  # Hide the plotting mode options\n"

  },

  {

    "filePath": "../HeatToolbox/src/hpc/\_\_init\_\_.py",

    "fileContent": ""

  },

  {

    "filePath": "../HeatToolbox/src/hpc/hpc\_utils.py",

    "fileContent": "import os\nimport subprocess\nimport tkinter as tk\nfrom tkinter import messagebox\nimport time\n\n\ndef submit\_job(hpc\_user, hpc\_host, hpc\_path, log\_dir, password):\n\n    local\_base\_path = \".\"\n    # Transfer the log directory to the HPC\n    transfer\_files\_to\_hpc(\n        local\_base\_path, log\_dir, hpc\_user, hpc\_host, hpc\_path, password\n    )\n    wait\_for\_files(\n        hpc\_user,\n        hpc\_host,\n        os.path.join(hpc\_path, log\_dir),\n        [\"hpc\_run.sh\", \"config.json\"],\n        password\n    )\n    try:\n        # Set the SSHPASS environment variable\n        env = os.environ.copy()\n        env['SSHPASS'] = password\n\n        # SSH command with environment setup for qsub\n        ssh\_command = [\n            \"sshpass\",\n            \"-e\",  # Use the password from the environment variable\n            \"ssh\",\n            f\"{hpc\_user}@{hpc\_host}\",\n            f\"bash -l -c 'cd {hpc\_path} && qsub {log\_dir}/hpc\_run.sh'\",\n        ]\n\n        # Run the command\n        result = subprocess.run(ssh\_command, check=True, text=True, capture\_output=True, env=env)\n\n        # Extract the job ID from the result\n        job\_id = result.stdout.strip()\n        messagebox.showinfo(\"Success\", f\"Job submitted successfully!\\nJob ID: {job\_id}\")\n\n    except subprocess.CalledProcessError as e:\n        messagebox.showerror(\"Error\", f\"Job submission failed.\\n{e.stderr}\")\n\n\ndef transfer\_files\_to\_hpc(\n    local\_base\_path, log\_dir, hpc\_user, hpc\_host, hpc\_remote\_base\_path, password\n):\n    \"\"\"\n    Transfer files to HPC while maintaining the folder structure.\n\n    Args:\n        local\_base\_path (str): The local base path (e.g., \"BTE-NO\").\n        log\_dir (str): The log directory to transfer (relative to the local base path, e.g., \"logs/logfolderiwant\").\n        hpc\_user (str): HPC username.\n        hpc\_host (str): HPC hostname.\n        hpc\_remote\_base\_path (str): The remote base path on the HPC (e.g., \"~/BTE-NO\").\n        password (str): Password for HPC access.\n    \"\"\"\n    # Ensure local paths exist\n    local\_src\_path = os.path.join(local\_base\_path, \"src\")\n    local\_log\_path = os.path.join(local\_base\_path, log\_dir)\n    if not os.path.exists(local\_src\_path):\n        raise FileNotFoundError(f\"Source directory not found: {local\_src\_path}\")\n    if not os.path.exists(local\_log\_path):\n        raise FileNotFoundError(f\"Log directory not found: {local\_log\_path}\")\n\n    # Set the SSHPASS environment variable\n    env = os.environ.copy()\n    env['SSHPASS'] = password\n\n    # Use subprocess with `sshpass` for password-based file transfer\n    rsync\_command = [\n        \"sshpass\", \"-e\",\n        \"rsync\",\n        \"-avz\",\n        \"--progress\",\n        \"--relative\",\n        \"--exclude-from\",\n        os.path.join(local\_base\_path, \"hpc\_exclude.txt\"),\n        f\"{local\_src\_path}/\",  # Transfer the src directory\n        f\"{local\_log\_path}/\",  # Transfer the specific log directory\n        f\"{hpc\_user}@{hpc\_host}:{hpc\_remote\_base\_path}/\",\n    ]\n    print(rsync\_command)\n\n    # Execute the command\n    try:\n        subprocess.run(rsync\_command, check=True, env=env)\n        print(\n            f\"Successfully transferred to {hpc\_user}@{hpc\_host}:{hpc\_remote\_base\_path}\"\n        )\n    except subprocess.CalledProcessError as e:\n        raise RuntimeError(f\"File transfer failed: {e}\")\n\n\ndef wait\_for\_files(hpc\_user, hpc\_host, remote\_path, files, password, timeout=30, interval=2):\n    env = os.environ.copy()\n    env['SSHPASS'] = password\n    elapsed = 0\n    while elapsed < timeout:\n        try:\n            # Check if all files exist on HPC\n            check\_command = [\n                \"sshpass\", \"-e\",\n                \"ssh\", f\"{hpc\_user}@{hpc\_host}\",\n                f\"ls {' '.join([os.path.join(remote\_path, file) for file in files])}\"\n            ]\n            subprocess.run(check\_command, check=True, env=env)\n            return  # Files are available\n        except subprocess.CalledProcessError:\n            elapsed += interval\n            print(f\"Waiting for files to sync... ({elapsed}/{timeout} seconds)\")\n            time.sleep(interval)\n    raise RuntimeError(\"Files did not appear on HPC within the timeout period.\")\n\n\ndef prompt\_password(prompt=\"Enter HPC Password\"):\n    \"\"\"Prompt the user to enter their HPC password securely.\"\"\"\n    import getpass\n    password = getpass.getpass(prompt + \": \")\n    return password\n\n\ndef save\_hpc\_script(script\_content, log\_dir):\n    \"\"\"Save the HPC script to the specified log directory.\"\"\"\n    hpc\_path = os.path.join(log\_dir, \"hpc\_run.sh\")\n    with open(hpc\_path, \"w\") as f:\n        f.write(script\_content)\n    print(f\"HPC script saved to {hpc\_path}\")\n    return hpc\_path\n"

  },

  {

    "filePath": "../HeatToolbox/src/hpc/script\_generator.py",

    "fileContent": "\ndef generate\_hpc\_script(config, config\_path):\n    \"\"\"\n    Generate the HPC script based on a configuration dictionary.\n\n    Args:\n        config (dict): Configuration dictionary containing HPC and simulation parameters.\n        config\_path (str): Path to the configuration JSON file.\n\n    Returns:\n        str: The content of the HPC script.\n    \"\"\"\n    # Start the script with the shebang and PBS directives\n    script\_lines = [\"#!/bin/bash\"]\n\n    # Select resources\n    select\_line = f\"#PBS -l select={config['nodes']}:ncpus={config['ncpus']}:mem={config['mem']}gb\"\n\n    # If parallelization is enabled, add mpiprocs\n    if config.get(\"parallelize\", False):\n        select\_line += f\":mpiprocs={config['mpiprocs']}\"\n\n    script\_lines.append(select\_line)\n\n    # Add walltime\n    script\_lines.append(f\"#PBS -l walltime={config['walltime']}\")\n\n    # Load necessary modules and activate the conda environment\n    script\_lines.extend(\n        [\n            \"\",\n            \"module load tools/prod\",\n            f'eval \"$({config[\"conda\_env\_path\"]} shell.bash hook)\"',\n            f\"conda activate {config['conda\_env\_name']}\",\n            \"\",\n            \"# Copy input file to $TMPDIR\",\n            \"cp -r $HOME/BTE-NO $TMPDIR/\",\n            \"\",\n            \"cd $TMPDIR/BTE-NO\",\n            \"\",\n        ]\n    )\n\n    # Generate the command line using the config path\n    command = f\"python src/main.py --config {config\_path}\"\n\n    # Calculate timeout in hours\n    if config.get(\"timeout\"):\n        timeout\_parts = list(map(int, config[\"timeout\"].split(\":\")))\n        script\_timeout = (\n            timeout\_parts[0] + timeout\_parts[1] / 60 + timeout\_parts[2] / 3600\n        )\n    else:\n        timeout\_parts = list(map(int, config[\"walltime\"].split(\":\")))\n        script\_timeout = (\n            timeout\_parts[0] + timeout\_parts[1] / 60 + timeout\_parts[2] / 3600\n        )\n        script\_timeout \*= 0.98  # Use 98% of walltime as timeout buffer\n\n    timeout\_line = f\"timeout {round(script\_timeout, 2)}h {command}\"\n\n    # Use mpirun if parallelization is enabled\n    if config.get(\"parallelize\", False):\n        mpirun\_command = (\n            f\"mpirun -np {config['mpiprocs'] \* config['nodes']} {timeout\_line}\"\n        )\n        script\_lines.append(\"# Run application with MPI\")\n        script\_lines.append(mpirun\_command)\n    else:\n        script\_lines.append(\"# Run application\")\n        script\_lines.append(timeout\_line)\n\n    script\_lines.extend(\n        [\n            \"\",\n            \"# Copy required files back\",\n            \"cp -r $TMPDIR/BTE-NO/logs/\* $HOME/BTE-NO/logs/\",\n            \"\",\n        ]\n    )\n\n    # Join the script lines into a single string\n    script\_content = \"\\n\".join(script\_lines)\n\n    return script\_content\n"

  },

  {

    "filePath": "../HeatToolbox/src/image\_processing.py",

    "fileContent": "# image\_processing.py\nimport numpy as np\n# from mpi4py import MPI\nimport torch\nimport torch.nn.functional as F\nfrom scipy.ndimage import zoom\n\n\ndef img\_list\_to\_gamma\_expression(img\_list, config):\n    \"\"\"\n    Build a function gamma\_expression(x\_input) -> array\n    that, for each point x\_input in the domain, returns\n    0 or 1 depending on whether it lies in a region\n    determined by 'img\_list' or the top-extrusion mask.\n\n    This handles both non-symmetric & symmetric geometries,\n    with multiple source positions.\n    \"\"\"\n\n    # ------------------------------------------------------------\n    # 1) Define domain bounding box\n    #    Note: If config.symmetry is True, we assume L\_X is\n    #    already half of the full domain width. The rest of\n    #    the logic (0..L\_X) stays the same.\n    # ------------------------------------------------------------\n    x\_min = 0\n    x\_max = config.L\_X\n    y\_min = 0\n    y\_max = config.L\_Y + config.SOURCE\_HEIGHT\n\n    # Range in X and Y\n    x\_range = x\_max - x\_min\n    y\_range = y\_max - y\_min\n\n    # For the images, we shift them upward by some offset:\n    y\_min\_im = y\_min + 1.5 \* config.SOURCE\_HEIGHT  # To make image higher\n\n    # ------------------------------------------------------------\n    # 2) Handle the source positions in absolute coordinates\n    # ------------------------------------------------------------\n    num\_sources = len(config.source\_positions)\n    # Convert fractional positions [0..1] => [0..L\_X]\n    # If symmetry is enabled, we assume the sources are\n    # already in the left half of the domain. We need to double\n    # the source positions to cover the full domain.\n    if config.symmetry:\n        source\_positions = np.array(config.source\_positions) \* config.L\_X \* 2\n    else:\n        source\_positions = np.array(config.source\_positions) \* config.L\_X\n\n    # ------------------------------------------------------------\n    # 3) Define how wide each image “strip” is\n    #    (Your original approach: x\_range / num\_sources)\n    # ------------------------------------------------------------\n    image\_mesh\_width = x\_range / num\_sources\n\n    # ------------------------------------------------------------\n    # 4) Build an array of x-ranges for each image in img\_list\n    # ------------------------------------------------------------\n    image\_x\_ranges = []\n    for x\_pos in source\_positions:\n        x\_min\_image = x\_pos - image\_mesh\_width / 2\n        x\_max\_image = x\_pos + image\_mesh\_width / 2\n\n        # Ensure x\_min\_image and x\_max\_image are within domain bounds\n        x\_min\_image = max(x\_min\_image, x\_min)\n        x\_max\_image = min(x\_max\_image, x\_max)\n\n        image\_x\_ranges.append([x\_min\_image, x\_max\_image])\n\n    # ------------------------------------------------------------\n    # 5) Define the “bottom” of the top extrusion region\n    #    This is for masking purposes.\n    # ------------------------------------------------------------\n    y\_min\_extrusion = config.L\_Y - config.SOURCE\_HEIGHT\n\n    # ------------------------------------------------------------\n    # 6) Construct the actual callback function for gamma\n    # ------------------------------------------------------------\n    def gamma\_expression(x\_input):\n        \"\"\"\n        x\_input.shape -> (gdim, N)\n        Feturn an array of shape (N,) with 0 or 1.\n        \"\"\"\n        # x\_input is of shape (gdim, N)\n        x\_coords = x\_input[0, :]\n        y\_coords = x\_input[1, :]\n\n        # Initialize gamma\_values with zeros\n        gamma\_values = np.zeros\_like(x\_coords)\n\n        # -------------------------------\n        # A) “Paint” the images\n        # -------------------------------\n        for img, (x\_min\_image, x\_max\_image) in zip(img\_list, image\_x\_ranges):\n            # img is a 2D numpy array of shape (height, width) with 0/1\n            img\_height, img\_width = img.shape\n\n            # Determine which points are within the image's x-range\n            in\_image = np.logical\_and(\n                x\_coords >= x\_min\_image, x\_coords <= x\_max\_image\n            )\n\n            # Map y as well, assuming the image spans from y\_min to y\_max\n            in\_image = np.logical\_and(\n                in\_image,\n                np.logical\_and(y\_coords >= y\_min, y\_coords <= y\_max)\n            )\n\n            # Normalize x and y within the image's range\n            x\_norm = (x\_coords[in\_image] - x\_min\_image) / image\_mesh\_width\n            y\_norm = (y\_coords[in\_image] - y\_min\_im) / y\_range\n\n            x\_indices = np.clip(\n                (x\_norm \* (img\_width - 1)).astype(int), 0, img\_width - 1\n            )\n            y\_indices = np.clip(\n                ((1 - y\_norm) \* (img\_height - 1)).astype(int), 0, img\_height - 1\n            )\n\n            # Get gamma values from the image\n            gamma\_values\_in\_image = img[y\_indices, x\_indices]\n\n            # Update gamma\_values array:\n            # if gamma\_values\_in\_image == 1, set gamma\_values to 1\n            gamma\_values\_current = gamma\_values[in\_image]\n            gamma\_values\_new = np.where(\n                gamma\_values\_in\_image == 1, 1, gamma\_values\_current\n            )\n            gamma\_values[in\_image] = gamma\_values\_new\n\n        # -------------------------------\n        # B) Optional: mask extrusions\n        # -------------------------------\n        if config.mask\_extrusion:\n            # We do multiple extrusions for multiple source\_positions\n            # whether or not we are in symmetry mode.\n            #\n            # If symmetry = True, we typically have L\_X “halved,” but\n            # the logic is the same, as we still place extrusions\n            # at each x\_pos ± source\_width/2 (with clamp).\n            for x\_pos in source\_positions:\n                # define a slightly bigger region than the width for safety\n                half\_w = 1.1 \* config.SOURCE\_WIDTH / 2.0\n                x\_min\_ex = x\_pos - half\_w\n                x\_max\_ex = x\_pos + half\_w\n\n                # clamp to [0, L\_X]\n                if x\_min\_ex < x\_min:\n                    x\_min\_ex = x\_min\n                if x\_max\_ex > x\_max:\n                    x\_max\_ex = x\_max\n\n                # all points above y\_min\_extrusion, within x\_min\_ex..x\_max\_ex\n                in\_extrusion\_x = np.logical\_and(x\_coords >= x\_min\_ex, x\_coords <= x\_max\_ex)\n                in\_extrusion\_y = y\_coords > y\_min\_extrusion\n                in\_extrusion = np.logical\_and(in\_extrusion\_x, in\_extrusion\_y)\n\n                # set gamma=1 in that top region\n                gamma\_values[in\_extrusion] = 1.0\n            # DEPRECIATE THE FOLLOWING:\n            # if config.symmetry:\n            #     x\_min\_extrusion = config.L\_X - config.SOURCE\_WIDTH\n            #     x\_max\_extrusion = config.L\_X\n\n            #     in\_extrusion = np.logical\_and(\n            #         np.logical\_and(\n            #             y\_coords > y\_min\_extrusion, x\_coords >= x\_min\_extrusion\n            #         ),\n            #         x\_coords <= x\_max\_extrusion,\n            #     )\n            #     gamma\_values[in\_extrusion] = 1.0\n            # else:\n            #     for x\_pos in source\_positions:\n            #         x\_min\_extrusion = x\_pos - 1.1 \* config.SOURCE\_WIDTH / 2\n            #         x\_max\_extrusion = x\_pos + 1.1 \* config.SOURCE\_WIDTH / 2\n\n            #         in\_extrusion = np.logical\_and(\n            #             np.logical\_and(\n            #                 y\_coords > y\_min\_extrusion, x\_coords >= x\_min\_extrusion\n            #             ),\n            #             x\_coords <= x\_max\_extrusion,\n            #         )\n            #         gamma\_values[in\_extrusion] = 1.0\n\n        return gamma\_values\n\n    return gamma\_expression\n\n\ndef generate\_images(config, latent\_vectors, model):\n    # Generate image from latent vector\n    img\_list = []\n    for z in latent\_vectors:\n        if config.blank:\n            img = np.zeros((128, 128))\n        else:\n            # Ensure z is reshaped correctly if needed\n            img = z\_to\_img(z.reshape(1, -1), model, config.vol\_fraction)\n        img\_list.append(img)\n\n    # Apply symmetry to each image if enabled\n    if config.symmetry:\n        # only keep the left half of the image if source\_position is 0.5\n        for i, img in enumerate(img\_list):\n            if config.source\_positions[i] == 0.5:\n                img\_list[i] = img[:, : img.shape[1] // 2]\n\n    return img\_list\n\n\ndef gaussian\_blur(img, sigma=1):\n    \"\"\"Apply Gaussian blur using PyTorch.\"\"\"\n    # Create a 2D Gaussian kernel\n    kernel\_size = int(4 \* sigma + 1)\n    x = torch.arange(-kernel\_size // 2 + 1.0, kernel\_size // 2 + 1.0)\n    gauss = torch.exp(-(x\*\*2) / (2 \* sigma\*\*2))\n    gauss = gauss / gauss.sum()\n    gauss\_kernel = gauss[:, None] \* gauss[None, :]\n    gauss\_kernel = gauss\_kernel.expand(\n        1, 1, \*gauss\_kernel.shape\n    )  # Shape to match convolution input\n\n    # Add batch and channel dimensions to the image for convolution\n    img\_tensor = torch.from\_numpy(img).unsqueeze(0).unsqueeze(0).float()\n\n    # Apply 2D convolution for Gaussian blur\n    blurred\_img = F.conv2d(img\_tensor, gauss\_kernel, padding=kernel\_size // 2)\n\n    # Remove extra dimensions\n    return blurred\_img.squeeze().numpy()\n\n\ndef apply\_volume\_fraction(img, vf):\n    # Flatten the image to a 1D array\n    img\_flat = img.flatten()\n    # Sort the pixels\n    sorted\_pixels = np.sort(img\_flat)\n    # Determine the threshold that will result in the desired volume fraction\n    num\_pixels = img\_flat.size\n    k = int((1 - vf) \* num\_pixels)\n    threshold = (sorted\_pixels[k] + sorted\_pixels[k - 1]) / 2.0\n    # Apply the threshold\n    img\_binary = (img >= threshold).astype(np.float32)\n    return img\_binary\n\n\ndef upsample\_image(img, zoom\_factor):\n    return zoom(img, zoom\_factor, order=3)  # Cubic interpolation\n\n\ndef z\_to\_img(z, model, vf, device=torch.device(\"cpu\"), sigma=1.5, zoom\_factor=3):\n    z = torch.from\_numpy(z).float().unsqueeze(0).to(device)  # Add batch dimension\n    model.eval()\n    with torch.no\_grad():\n        sample = model.decode(z)\n        # Remove batch and channel dimensions\n        img = sample.squeeze().squeeze().cpu().numpy()\n        # Flip the image\n        img = img[::-1, :]\n        # Take the left half of the image and resymmetrize\n        img = img[:, : img.shape[1] // 2]\n        img = np.concatenate((img, img[:, ::-1]), axis=1)\n\n    # Apply Gaussian blur to smoothen the image\n    img\_smoothed = gaussian\_blur(img, sigma=sigma)\n    # Upsample the image before thresholding\n    img\_upsampled = upsample\_image(img\_smoothed, zoom\_factor=zoom\_factor)\n\n    # Apply volume fraction control or default thresholding\n    if vf is None:\n        # Default binary thresholding\n        img\_binary = (img\_upsampled >= 0.5).astype(np.float32)\n    else:\n        img\_binary = apply\_volume\_fraction(img\_upsampled, vf)\n\n    return img\_binary\n\n\ndef img\_to\_gamma\_expression(img, config):\n    # Compute global min and max coordinates\n    x\_min = 0\n    x\_max = config.L\_X\n    y\_min = 0\n    y\_max = config.L\_Y + config.SOURCE\_HEIGHT\n\n    img\_height, img\_width = img.shape\n\n    x\_range = x\_max - x\_min\n    y\_range = y\_max - y\_min\n\n    y\_min += 1.5 \* config.SOURCE\_HEIGHT  # To make image higher\n\n    # Avoid division by zero in case the mesh or image has no range\n    if x\_range == 0:\n        x\_range = 1.0\n    if y\_range == 0:\n        y\_range = 1.0\n\n    y\_min\_extrusion = config.L\_Y - config.SOURCE\_HEIGHT\n    # Define the extrusion region\n    if config.symmetry:\n        x\_min\_extrusion = config.L\_X - config.SOURCE\_WIDTH\n        x\_max\_extrusion = config.L\_X\n    elif len(config.source\_positions) > 1:\n        x\_min\_extrusion\_l = config.L\_X \* 0.25 - config.SOURCE\_WIDTH / 2\n        x\_max\_extrusion\_l = config.L\_X \* 0.25 + config.SOURCE\_WIDTH / 2\n        x\_min\_extrusion\_r = config.L\_X \* 0.75 - config.SOURCE\_WIDTH / 2\n        x\_max\_extrusion\_r = config.L\_X \* 0.75 + config.SOURCE\_WIDTH / 2\n    else:\n        x\_min\_extrusion = config.L\_X / 2 - config.SOURCE\_WIDTH / 2\n        x\_max\_extrusion = config.L\_X / 2 + config.SOURCE\_WIDTH / 2\n\n    def gamma\_expression(x\_input):\n        # x\_input is of shape (gdim, N)\n        x\_coords = x\_input[0, :]\n        y\_coords = x\_input[1, :]\n\n        # Initialize gamma\_values with zeros\n        gamma\_values = np.zeros\_like(x\_coords)\n\n        # For all points in the mesh, scale the image to the full size of the mesh\n        x\_norm = (x\_coords - x\_min) / x\_range  # Normalize x coordinates\n        y\_norm = (y\_coords - y\_min) / y\_range  # Normalize y coordinates\n\n        x\_indices = np.clip((x\_norm \* (img\_width - 1)).astype(int), 0, img\_width - 1)\n        y\_indices = np.clip(\n            ((1 - y\_norm) \* (img\_height - 1)).astype(int), 0, img\_height - 1\n        )\n\n        gamma\_values = img[y\_indices, x\_indices]  # Map image values to the mesh\n\n        if config.mask\_extrusion:\n            # Mask the top extrusion if requested\n            if len(config.source\_positions) > 1:\n                # use the left and the right\n                in\_extrusion\_l = np.logical\_and(\n                    np.logical\_and(\n                        y\_coords > y\_min\_extrusion, x\_coords >= x\_min\_extrusion\_l\n                    ),\n                    x\_coords <= x\_max\_extrusion\_l,\n                )\n                in\_extrusion\_r = np.logical\_and(\n                    np.logical\_and(\n                        y\_coords > y\_min\_extrusion, x\_coords >= x\_min\_extrusion\_r\n                    ),\n                    x\_coords <= x\_max\_extrusion\_r,\n                )\n                gamma\_values[in\_extrusion\_l] = 1.0\n                gamma\_values[in\_extrusion\_r] = 1.0\n            else:\n                in\_extrusion = np.logical\_and(\n                    np.logical\_and(\n                        y\_coords > y\_min\_extrusion, x\_coords >= x\_min\_extrusion\n                    ),\n                    x\_coords <= x\_max\_extrusion,\n                )\n                gamma\_values[in\_extrusion] = 1.0\n\n        return gamma\_values\n\n    return gamma\_expression\n"

  },

  {

    "filePath": "../HeatToolbox/src/log\_utils.py",

    "fileContent": "# import os\nimport numpy as np\n\n\ndef read\_last\_latent\_vector(cma\_log\_file, z\_dim, num\_sources):\n    \"\"\"\n    Read the last latent vector(s) from the CMA-ES log file.\n\n    Parameters:\n    - cma\_log\_file (str): Path to the CMA-ES log file.\n    - z\_dim (int): Dimension of the latent vector.\n    - num\_sources (int): Number of sources (latent vectors).\n\n    Returns:\n    - latent\_vectors (list of numpy arrays): List containing the latent vectors.\n    \"\"\"\n    with open(cma\_log\_file, 'r') as f:\n        lines = f.readlines()\n\n    # Filter out comment lines (starting with '%')\n    data\_lines = [line for line in lines if not line.strip().startswith('%') and line.strip()]\n    if not data\_lines:\n        raise ValueError(\"No data lines found in the CMA-ES log file.\")\n\n    # Take the line with the best solution\n    # Need to find the minimum of the third column in the data\_lines\n    best\_line = min(data\_lines, key=lambda x: float(x.strip().split()[2]))\n\n    # Split the line into components\n    parts = best\_line.strip().split()\n    # The latent vector starts from the 6th element (index 5)\n    latent\_values = [float(x) for x in parts[5:]]\n\n    # Check if the length matches expected size\n    expected\_length = z\_dim \* num\_sources\n    if len(latent\_values) != expected\_length:\n        raise ValueError(f\"Expected {expected\_length} latent values, but got {len(latent\_values)}.\")\n\n    # Split the latent\_values into individual latent vectors\n    latent\_vectors = []\n    for i in range(num\_sources):\n        start\_idx = i \* z\_dim\n        end\_idx = (i + 1) \* z\_dim\n        latent\_vector = np.array(latent\_values[start\_idx:end\_idx])\n        latent\_vectors.append(latent\_vector)\n\n    return latent\_vectors\n"

  },

  {

    "filePath": "../HeatToolbox/src/logging\_module.py",

    "fileContent": "import os\nimport json\nimport datetime\nimport numpy as np\nfrom mpi4py import MPI\n\n\nclass LoggingModule:\n    def \_\_init\_\_(self, config):\n        self.rank = MPI.COMM\_WORLD.rank\n        self.config = config\n\n        # Determine the log directory name\n        if config.log\_name:\n            # Use the provided log name\n            self.log\_dir = os.path.join(\"logs\", config.log\_name)\n        else:\n            # Generate a unique directory name based on timestamp and descriptors\n            timestamp = datetime.datetime.now().strftime(\"%Y%m%d\_%H%M%S\")\n            if config.optim:\n                optimizer = config.optimizer\n            else:\n                optimizer = \"solve-only\"\n            num\_sources = (\n                len(config.source\_positions)\n                if hasattr(config, \"source\_positions\")\n                else \"unknown\"\n            )\n            self.log\_dir = f\"logs/{timestamp}\_{optimizer}\_sources\_{num\_sources}\"\n\n        # Create the log directory if it doesn't exist\n        if self.rank == 0:\n            os.makedirs(self.log\_dir, exist\_ok=True)\n            # Initialize logging\n            self.\_initialize\_logging()\n\n    def \_initialize\_logging(self):\n        # Save the configuration file in the log directory\n        config\_path = os.path.join(self.log\_dir, \"config.json\")\n        # Only create/write config.json if it doesn't already exist\n        if not os.path.exists(config\_path):\n            with open(config\_path, \"w\") as f:\n                json.dump(self.config.config, f, indent=4)\n        else:\n            print(f\"Skipping config.json write; already exists at {config\_path}\")\n\n        # Save a command-line script to reproduce the run\n        run\_script\_path = os.path.join(self.log\_dir, \"run\_simulation.sh\")\n        command = f\"python src/main.py --config {config\_path}\"\n        with open(run\_script\_path, \"w\") as f:\n            f.write(command + \"\\n\")\n\n        # Save the HPC script if generated\n        if hasattr(self.config, \"hpc\_script\_content\"):\n            hpc\_script\_path = os.path.join(self.log\_dir, \"hpc\_run.sh\")\n            with open(hpc\_script\_path, \"w\") as f:\n                f.write(self.config.hpc\_script\_content)\n\n    def log\_generation\_data(self, generation, data):\n        \"\"\"Log generation data during optimization.\"\"\"\n        if self.rank == 0:\n            with open(os.path.join(self.log\_dir, \"generation\_data.txt\"), \"a\") as f:\n                timestamp = datetime.datetime.now().isoformat()\n                f.write(f\"{timestamp} - Generation {generation}: {data}\\n\")\n\n    def save\_optimized\_latent\_vectors(self, latent\_vectors):\n        \"\"\"Save the best latent vectors from optimization.\"\"\"\n        if self.rank == 0:\n            # Save the latent vectors in the optimization\_results subdirectory\n            opt\_results\_dir = os.path.join(self.log\_dir, \"optimization\_results\")\n            os.makedirs(opt\_results\_dir, exist\_ok=True)\n            latent\_vectors\_path = os.path.join(opt\_results\_dir, \"best\_latent\_vectors.npy\")\n            np.save(latent\_vectors\_path, latent\_vectors)\n\n    def save\_image(self, fig, filename, latent\_vector\_name=None):\n        \"\"\"Save visualization images in the appropriate directory.\"\"\"\n        if self.rank == 0:\n            if latent\_vector\_name:\n                # Create a subdirectory for the specific latent vector\n                vis\_dir = os.path.join(self.log\_dir, \"visualization\", latent\_vector\_name)\n            else:\n                vis\_dir = os.path.join(self.log\_dir, \"visualization\")\n            os.makedirs(vis\_dir, exist\_ok=True)\n            filepath = os.path.join(vis\_dir, filename)\n            fig.savefig(filepath)\n            print(f\"Visualization saved to {filepath}\")\n"

  },

  {

    "filePath": "../HeatToolbox/src/main.py",

    "fileContent": "# main.py\n\n# import libraries\nimport torch\nfrom mpi4py import MPI\nimport argparse\nimport numpy as np\nimport dolfinx.io\nimport os\n\n# import modules\nfrom mesh\_generator import MeshGenerator\nfrom vae\_module import load\_vae\_model, VAE, Flatten, UnFlatten\nfrom image\_processing import z\_to\_img, generate\_images\nfrom opts.cmaes import CMAESModule\nfrom opts.bayes import BayesianModule\nfrom post\_processing import PostProcessingModule\nfrom solver\_gke\_module import GKESolver\nfrom solver\_fourier\_module import FourierSolver\nfrom logging\_module import LoggingModule\nfrom sim\_config import SimulationConfig\nfrom log\_utils import read\_last\_latent\_vector\n\n\nclass SimulationController:\n    def \_\_init\_\_(self, config):\n        self.config = config\n        self.comm = MPI.COMM\_WORLD\n        self.rank = self.comm.rank\n        self.logger = LoggingModule(config) if config.logging\_enabled and self.rank == 0 else None\n        self.model = load\_vae\_model(self.rank, self.config.latent\_size)\n\n    def run\_simulation(self):\n        # Mesh generation\n        mesh\_generator = MeshGenerator(self.config)\n        time1 = MPI.Wtime()\n\n        # Create solver instance\n        if self.rank == 0:\n            if self.config.symmetry:\n                mesh\_generator.sym\_create\_mesh()\n            else:\n                mesh\_generator.create\_mesh()\n        else:\n            msh = None\n            cell\_markers = None\n            facet\_markers = None\n\n        # Broadcast mesh data to all processes\n        self.comm.barrier()\n        msh, cell\_markers, facet\_markers = dolfinx.io.gmshio.read\_from\_msh(\n            \"domain\_with\_extrusions.msh\", MPI.COMM\_SELF, gdim=2\n        )\n\n        # create solver instance\n        if self.config.solver\_type == \"gke\":\n            solver = GKESolver(msh, facet\_markers, self.config)\n        elif self.config.solver\_type == \"fourier\":\n            solver = FourierSolver(msh, facet\_markers, self.config)\n        else:\n            raise ValueError(f\"Unknown solver type: {self.config.solver\_type}\")\n\n        if self.config.optim:\n            # Run optimization\n            optimizer = None\n            if self.config.optimizer == \"cmaes\":\n                optimizer = CMAESModule(\n                    solver,\n                    self.model,\n                    torch.device(\"cpu\"),\n                    self.rank,\n                    self.config,\n                    logger=self.logger,\n                )\n                best\_z\_list = optimizer.optimize(\n                    n\_iter=100\n                )  # Adjust iterations as needed\n            elif self.config.optimizer == \"bayesian\":\n                optimizer = BayesianModule(\n                    solver,\n                    self.model,\n                    torch.device(\"cpu\"),\n                    self.rank,\n                    self.config,\n                    logger=self.logger,\n                )\n                best\_z\_list = optimizer.optimize(init\_points=10, n\_iter=100)\n\n            latent\_vectors = best\_z\_list\n            # Optional: Save the best\_z to a file for future solving\n            # if self.rank == 0:\n            #     self.logger.save\_optimized\_latent\_vectors(latent\_vectors)\n        elif self.config.load\_cma\_result:\n            # Load latent vectors from CMA-ES log file\n            latent\_vectors = self.load\_latent\_vectors\_from\_cma\_log()\n            print(latent\_vectors)\n            # Proceed to generate images and solve\n            img\_list = generate\_images(self.config, latent\_vectors, self.model)\n\n            if self.config.visualize['pregamma']:\n                self.plot\_image\_list(img\_list, self.config, logger=self.logger)\n\n            avg\_temp\_global = solver.solve\_image(img\_list)\n            time2 = MPI.Wtime()\n            if self.rank == 0:\n                print(f\"Average temperature: {avg\_temp\_global} K\")\n                print(f\"Time taken to solve: {time2 - time1:.3f} seconds\")\n\n            # Check if visualize list is not empty\n            if self.config.visualize:\n                import cma\n                cma\_log\_dir = os.path.join(self.config.log\_dir, \"cma\_logs\")\n                cma.plot(os.path.join(cma\_log\_dir, \"outcma\_\"))\n                cma.s.figsave(os.path.join(cma\_log\_dir, 'convergence\_plots.png'))\n\n                post\_processor = PostProcessingModule(\n                    self.rank, self.config, logger=self.logger\n                )\n \n                if self.config.solver\_type == \"gke\":\n                    q, T = solver.U.sub(0).collapse(), solver.U.sub(1).collapse()\n                    V1, \_ = solver.U.function\_space.sub(1).collapse()\n                elif self.config.solver\_type == \"fourier\":\n                    q, T = None, solver.T\n                    V1 = solver.V\n\n                post\_processor.postprocess\_results(q, T, V1, solver.msh, solver.gamma)\n        else:\n            latent\_vectors = self.get\_latent\_vectors()\n\n            # Generate image from latent vector\n            img\_list = generate\_images(self.config, latent\_vectors, self.model)\n            # check if pregamma key is true in the visualize dictinoary\n            if self.config.visualize['pregamma']:\n                self.plot\_image\_list(img\_list, self.config, logger=self.logger)\n\n            avg\_temp\_global = solver.solve\_image(img\_list)\n            time2 = MPI.Wtime()\n            if self.rank == 0:\n                print(f\"Average temperature: {avg\_temp\_global} K\")\n                print(f\"Time taken to solve: {time2 - time1:.3f} seconds\")\n\n            # Check if visualize list is not empty\n            if self.config.visualize:\n                post\_processor = PostProcessingModule(\n                    self.rank, self.config, logger=self.logger\n                )\n                if self.config.solver\_type == \"gke\":\n                    q, T = solver.U.sub(0).collapse(), solver.U.sub(1).collapse()\n                    V1, \_ = solver.U.function\_space.sub(1).collapse()\n                elif self.config.solver\_type == \"fourier\":\n                    q, T = None, solver.T\n                    V1 = solver.V\n\n                post\_processor.postprocess\_results(q, T, V1, solver.msh, solver.gamma)\n\n    def get\_latent\_vectors(self):\n        # Handle latent vector based on the selected method\n        latent\_vectors = []\n        if self.config.latent\_method == \"manual\":\n            # Use the latent vector provided in args.latent\n            z = np.array(self.config.latent)\n            if len(z) != self.config.latent\_size:\n                raise ValueError(\n                    f\"Expected latent vector of size {self.config.latent\_size}, got {len(z)}.\"\n                )\n            latent\_vectors = [z] \* len(self.config.source\_positions)\n        elif self.config.latent\_method == \"random\":\n            # Generate random latent vectors\n            for \_ in range(len(self.config.source\_positions)):\n                z = np.random.randn(self.config.latent\_size)\n                latent\_vectors.append(z)\n        elif self.config.latent\_method == \"preloaded\":\n            # Load latent vectors from file\n            try:\n                best\_z\_list = np.load(\"best\_latent\_vector.npy\", allow\_pickle=True)\n                print(\"Opening best vector from file\")\n                latent\_vectors = best\_z\_list\n            except FileNotFoundError:\n                raise FileNotFoundError(\n                    \"No saved latent vectors found. Please provide a valid file.\"\n                )\n        return latent\_vectors\n\n    def plot\_image\_list(self, img\_list, config, logger=None):\n        import matplotlib.pyplot as plt\n\n        fig, axs = plt.subplots(1, len(img\_list), figsize=(15, 5))\n        if len(img\_list) == 1:\n            axs.imshow(img\_list[0], cmap=\"gray\")\n            axs.axis(\"off\")\n        else:\n            for i, img in enumerate(img\_list):\n                axs[i].imshow(img, cmap=\"gray\")\n                axs[i].axis(\"off\")\n        if config.plot\_mode == \"screenshot\":\n            if logger:\n                logger.save\_image(fig, \"image\_list.png\")\n            else:\n                plt.savefig(\"image\_list.png\")\n        else:\n            plt.show()\n\n    def load\_latent\_vectors\_from\_cma\_log(self):\n        \"\"\"Load the most recent CMA-ES result and update latent vectors.\"\"\"\n        cma\_log\_file = os.path.join(self.config.log\_dir, \"cma\_logs\", \"outcma\_xrecentbest.dat\")\n        z\_dim = self.config.latent\_size\n        num\_sources = len(self.config.source\_positions)\n        print(f\"Zdim and num sources: {z\_dim}, {num\_sources}\")\n        latent\_vectors = read\_last\_latent\_vector(cma\_log\_file, z\_dim, num\_sources)\n        return latent\_vectors\n\n\ndef parse\_arguments():\n    # Command-line arguments to determine the modes\n    parser = argparse.ArgumentParser()\n    parser.add\_argument(\n        \"--config\", type=str, required=True, help=\"Path to the configuration JSON file.\"\n    )\n    return parser.parse\_args()\n\n\nif \_\_name\_\_ == \"\_\_main\_\_\":\n    args = parse\_arguments()\n    config = SimulationConfig(args.config)\n    controller = SimulationController(config)\n    controller.run\_simulation()\n"

  },

  {

    "filePath": "../HeatToolbox/src/mesh\_generator.py",

    "fileContent": "import gmsh\nfrom mpi4py import MPI\nfrom dolfinx import io\n\n\nclass MeshGenerator:\n    def \_\_init\_\_(self, config):\n        self.config = config\n\n    def create\_mesh(self):\n        comm = MPI.COMM\_SELF\n        rank = comm.rank\n\n        gmsh.initialize()\n        gmsh.option.setNumber(\"General.Terminal\", 0)\n        gmsh.model.add(\"domain\_with\_extrusions\")\n\n        L\_X = self.config.L\_X\n        L\_Y = self.config.L\_Y\n        res = self.config.RESOLUTION\n        source\_width = self.config.SOURCE\_WIDTH\n        source\_height = self.config.SOURCE\_HEIGHT\n\n        # Define points for the base rectangle\n        p0 = gmsh.model.geo.addPoint(0, 0, 0, meshSize=res)  # Bottom left\n        p1 = gmsh.model.geo.addPoint(L\_X, 0, 0, meshSize=res)  # Bottom right\n        p2 = gmsh.model.geo.addPoint(L\_X, L\_Y, 0, meshSize=res)  # Top right\n        p3 = gmsh.model.geo.addPoint(0, L\_Y, 0, meshSize=res)  # Top left\n\n        # Define lines for the base rectangle\n        l0 = gmsh.model.geo.addLine(p0, p1)  # Bottom\n        l1 = gmsh.model.geo.addLine(p1, p2)  # Right\n        l2 = gmsh.model.geo.addLine(p2, p3)  # Top\n        l3 = gmsh.model.geo.addLine(p3, p0)  # Left\n\n        # Initialize list to hold all extrusion curve loops\n        extrusion\_loops = []\n\n        # Physical groups for boundaries\n        slip\_boundaries = [l1, l3]\n        top\_boundaries = []\n        connect\_rect\_extrusion\_point = p3\n        rect\_extr\_lines = []\n        # Iterate over each source position to create extrusions\n        for idx, pos in enumerate(self.config.source\_positions):\n            x\_pos = pos \* L\_X\n            x\_min = x\_pos - source\_width / 2\n            x\_max = x\_pos + source\_width / 2\n            y\_min = L\_Y\n            y\_max = L\_Y + source\_height\n\n            # Define points for the extrusion (source region)\n            p4 = gmsh.model.geo.addPoint(x\_min, y\_min, 0, meshSize=res)  # Bottom left of extrusion\n            p5 = gmsh.model.geo.addPoint(x\_max, y\_min, 0, meshSize=res)  # Bottom right of extrusion\n            p6 = gmsh.model.geo.addPoint(x\_max, y\_max, 0, meshSize=res)  # Top right of extrusion\n            p7 = gmsh.model.geo.addPoint(x\_min, y\_max, 0, meshSize=res)  # Top left of extrusion\n\n            # Define lines for the extrusion\n            # l4 = gmsh.model.geo.addLine(p4, p5)  # Bottom of extrusion\n            l5 = gmsh.model.geo.addLine(p5, p6)  # Right of extrusion\n            l6 = gmsh.model.geo.addLine(p6, p7)  # Top of extrusion\n            l7 = gmsh.model.geo.addLine(p7, p4)  # Left of extrusion\n\n            # To connect extrusion to the base rectangle (For curve loop and slip line)\n            l\_connect = gmsh.model.geo.addLine(connect\_rect\_extrusion\_point, p4)\n            connect\_rect\_extrusion\_point = p5\n            rect\_extr\_lines.append(l\_connect)\n\n            # Define curve loops for the extrusion (for curve loop)\n            extrusion\_loop = [l5, l6, l7]\n            extrusion\_loops.append(extrusion\_loop)  # Use this for curve loop later\n\n            # Physical groups for boundaries\n            slip\_boundaries.extend([l5, l7, l\_connect])\n            top\_boundaries.append(l6)  # Top of each extrusion as a separate boundary\n\n        l\_connect = gmsh.model.geo.addLine(connect\_rect\_extrusion\_point, p2)\n        slip\_boundaries.append(l\_connect)\n        rect\_extr\_lines.append(l\_connect)\n        all\_loops = [l0, l1]\n        # reverse order of extrusion loop and rect\_extr\_lines:\n        rect\_extr\_lines = rect\_extr\_lines[::-1]\n        extrusion\_loops = extrusion\_loops[::-1]\n        for i in range(len(extrusion\_loops)):\n            all\_loops.append(-rect\_extr\_lines[i])\n            all\_loops.extend(extrusion\_loops[i])\n        # add last rect\_extr\_line\n        all\_loops.extend([-rect\_extr\_lines[-1], l3])\n\n        # create gmsh loop and surface\n        loop\_combined = gmsh.model.geo.addCurveLoop(all\_loops)\n        surface = gmsh.model.geo.addPlaneSurface([loop\_combined])\n\n        gmsh.model.geo.synchronize()\n\n        # Define physical groups\n        # Isothermal Boundary\n        gmsh.model.addPhysicalGroup(1, [l0], tag=1)\n        gmsh.model.setPhysicalName(1, 1, \"IsothermalBoundary\")\n\n        # Slip Boundaries\n        gmsh.model.addPhysicalGroup(1, slip\_boundaries, tag=2)\n        gmsh.model.setPhysicalName(1, 2, \"SlipBoundary\")\n\n        # Top Boundaries\n        for idx, l\_top in enumerate(top\_boundaries):\n            tag = 3 + idx  # Starting from tag 3\n            gmsh.model.addPhysicalGroup(1, [l\_top], tag=tag)\n            gmsh.model.setPhysicalName(1, tag, f\"TopBoundary\_{idx}\")\n\n        # Domain\n        gmsh.model.addPhysicalGroup(2, [surface], tag=1)\n        gmsh.model.setPhysicalName(2, 1, \"Domain\")\n\n        # Generate the mesh\n        gmsh.model.mesh.generate(2)\n\n        # save the mesh because of mpi\n        gmsh.write(\"domain\_with\_extrusions.msh\")\n\n        # Convert to Dolfinx mesh\n        # msh, cell\_markers, facet\_markers = io.gmshio.model\_to\_mesh(\n        #     gmsh.model,\n        #     comm=comm,\n        #     rank=rank,\n        #     gdim=2\n        # )\n\n        gmsh.finalize()\n\n        # return msh, cell\_markers, facet\_markers\n\n    def sym\_create\_mesh(self):\n        \"\"\"\n        Symmetric version: \n        - The right boundary (x = L\_X) is flagged as 'Symmetry'.\n            This boundary is set as L\_X / 2 in sim\_config.py\n        - We create a single source 'extrusion' region at the top-right corner.\n        \"\"\"\n        comm = MPI.COMM\_SELF\n        rank = comm.rank\n\n        gmsh.initialize()\n        gmsh.option.setNumber(\"General.Terminal\", 0)\n        gmsh.model.add(\"domain\_with\_extrusions\")\n\n        L\_X = self.config.L\_X\n        L\_Y = self.config.L\_Y\n        res = self.config.RESOLUTION\n        source\_width = self.config.SOURCE\_WIDTH\n        source\_height = self.config.SOURCE\_HEIGHT\n\n        # if ther is source at x=0.5:\n        y\_max = L\_Y + self.config.SOURCE\_HEIGHT\n\n        # -------------------------------------------------\n        # 1) Define base rectangle (0 <= x <= L\_X, 0 <= y <= L\_Y)\n        # -------------------------------------------------\n        #                         --p2\n        #    p3-------- l2? ------\n        #    |                       |\n        #   l3                      l1  (Symmetry)\n        #    |                       |\n        #    p0--------- l0 --------p1\n        #        \n        p0 = gmsh.model.geo.addPoint(0, 0, 0, meshSize=res)  # Bottom left\n        p1 = gmsh.model.geo.addPoint(L\_X, 0, 0, meshSize=res)  # Bottom right\n        p2 = gmsh.model.geo.addPoint(L\_X, y\_max, 0, meshSize=res)  # Top right\n        p3 = gmsh.model.geo.addPoint(0, L\_Y, 0, meshSize=res)  # Top left\n\n        # Define lines for the base rectangle\n        l0 = gmsh.model.geo.addLine(p0, p1)  # Bottom\n        l1 = gmsh.model.geo.addLine(p1, p2)  # Right (SYMMETRY LINE)\n        # l2 = gmsh.model.geo.addLine(p2, p3)  # Top\n        l3 = gmsh.model.geo.addLine(p3, p0)  # Left\n\n        # -------------------------------------------------\n        # 2) Extrusions: from y=L\_Y up to y=L\_Y+source\_height\n        #    for each source in 'source\_positions'\n        # -------------------------------------------------\n        #\n        # We'll connect them from left to right, building lines that\n        # run along the top of the base rectangle (p3 => first block => ... => p2).\n        #\n        # - \"connect\_rect\_extrusion\_point\" is the last x on the top row that we reached\n        #   (starting from p3).\n        # - For each source:\n        #       x\_pos  = fraction \* L\_X\n        #       x\_min  = x\_pos - source\_width/2\n        #       x\_max  = x\_pos + source\_width/2\n        #   Then create p4, p5, p6, p7 for that block\n        #   Connect previous chunk to p4 with a new line\n        # -------------------------------------------------\n\n        # We store the vertical/diagonal lines that connect from one block to the next\n        rect\_extr\_lines = []\n        # Initialize list to hold all extrusion curve loops\n        extrusion\_loops = []\n\n        # Physical groups for boundaries\n        isothermal\_boundaries = [l0]  # bottom\n        symmetry\_boundaries = [l1]  # right\n        slip\_boundaries = [l3]  # left\n        top\_boundaries = []  # top\n\n        # Start with the top-left corner (p3) of the base rectangle\n        connect\_rect\_extrusion\_point = p3\n\n        # Sort sources by ascending x-position (if needed)\n        sorted\_positions = sorted(self.config.source\_positions)\n        # Iterate over each source position to create extrusions\n        for idx, pos in enumerate(sorted\_positions):\n            x\_pos = pos \* L\_X \* 2  # x2 because source positions are to [0,1] and domain is [0,0.5]\n            x\_min = x\_pos - source\_width / 2\n            x\_max = x\_pos + source\_width / 2\n\n            # If you want to clamp to [0, L\_X] so blocks cannot go out of domain:\n            if x\_min < 0.0:\n                x\_min = 0.0\n            if x\_max > L\_X:\n                x\_max = L\_X\n\n            y\_min = L\_Y\n            y\_max = L\_Y + source\_height\n\n            # Define points for the extrusion (source region)\n            p4 = gmsh.model.geo.addPoint(x\_min, y\_min, 0, meshSize=res)  # Bottom left of extrusion\n            p5 = gmsh.model.geo.addPoint(x\_max, y\_min, 0, meshSize=res)  # Bottom right of extrusion\n            if abs(x\_max - L\_X) > 1e-14:  # if we are not on the symmetry line\n                p6 = gmsh.model.geo.addPoint(x\_max, y\_max, 0, meshSize=res)  # Top right of extrusion\n            else:  # we are on symmetry line\n                p6 = p2  # set the top right of extrusion to be the top right of the base rectangle\n            p7 = gmsh.model.geo.addPoint(x\_min, y\_max, 0, meshSize=res)  # Top left of extrusion\n\n            # Define lines for the extrusion\n            # l4 = gmsh.model.geo.addLine(p4, p5)  # Bottom of extrusion\n            l5 = gmsh.model.geo.addLine(p5, p6)  # Right of extrusion\n            l6 = gmsh.model.geo.addLine(p6, p7)  # Top of extrusion\n            l7 = gmsh.model.geo.addLine(p7, p4)  # Left of extrusion\n\n            # To connect extrusion to the base rectangle (For curve loop and slip line)\n            # First run: p3 to p4 is top left of rect to bottom left of first block\n            # Second run: p5 (bottom right of previous) to p4 (bottom left of current)\n            l\_connect = gmsh.model.geo.addLine(connect\_rect\_extrusion\_point, p4)\n            # Update the \"current\" top reference to the bottom-right corner of this block\n            connect\_rect\_extrusion\_point = p5\n            rect\_extr\_lines.append(l\_connect)\n\n            # if we are not on the symmetry line:\n            if abs(x\_max - L\_X) > 1e-14:\n                # Define curve loops for the extrusion (for curve loop)\n                # add right (l5) and top (l6) if we are not on symmetry line\n                extrusion\_loops.append([l5, l6, l7])\n                # Extend: Right edge l5\n                slip\_boundaries.append(l5)\n            else:  # we are on symmetry line\n                # dont add l5 (right)\n                extrusion\_loop = [l6, l7]\n                extrusion\_loops.append(extrusion\_loop)  # Use this for curve loop later\n\n            # Left edge l7 (probably always slip unless x\_min = 0)\n            slip\_boundaries.append(l7)\n            # The connecting line from the last top corner to the new block's left corner\n            slip\_boundaries.append(l\_connect)  # or symmetry if it touches x= L\_X, etc.\n            top\_boundaries.append(l6)  # Top of each extrusion as a separate boundary\n\n        # -------------------------------------------------\n        # 3) Build the final curve loop\n        # -------------------------------------------------\n        # Following the same orientation logic as your original code:\n        #\n        #   all\_loops starts with [l0, l1] => bottom & right\n        #   then we reverse rect\_extr\_lines and extrusion\_loops\n        #   for each block we do -l\_connect, +[l5, l6, l7]\n        #   at the end we add -rect\_extr\_lines[-1], l3\n        #\n        all\_loops = [l0, l1]   # bottom + right\n        # reverse order of extrusion loop and rect\_extr\_lines:\n        rect\_extr\_lines = rect\_extr\_lines[::-1]\n        extrusion\_loops = extrusion\_loops[::-1]\n        # for each block, add -l\_connect, +[l5, l6, l7]\n        for i in range(len(extrusion\_loops)):\n            # FIRST DO EXTRUSION IN MIDDLE\n            all\_loops.extend(extrusion\_loops[i])\n            # THEN CONNECT TO NEXT\n            all\_loops.append(-rect\_extr\_lines[i])\n        # add LEFT side l3\n        all\_loops.append(l3)\n        # create gmsh loop and surface\n        loop\_combined = gmsh.model.geo.addCurveLoop(all\_loops)\n        surface = gmsh.model.geo.addPlaneSurface([loop\_combined])\n\n        gmsh.model.geo.synchronize()\n\n        # Define physical groups\n        # Isothermal Boundary\n        gmsh.model.addPhysicalGroup(1, isothermal\_boundaries, tag=1)\n        gmsh.model.setPhysicalName(1, 1, \"IsothermalBoundary\")\n\n        # Slip Boundaries\n        gmsh.model.addPhysicalGroup(1, slip\_boundaries, tag=2)\n        gmsh.model.setPhysicalName(1, 2, \"SlipBoundary\")\n\n        # Top Boundaries\n        for idx, l\_top in enumerate(top\_boundaries):\n            tag = 3 + idx  # Starting from tag 3\n            gmsh.model.addPhysicalGroup(1, [l\_top], tag=tag)\n            gmsh.model.setPhysicalName(1, tag, f\"TopBoundary\_{idx}\")\n\n        gmsh.model.addPhysicalGroup(1, symmetry\_boundaries, tag=99)\n        gmsh.model.setPhysicalName(1, 99, \"Symmetry\")\n\n        # Domain\n        gmsh.model.addPhysicalGroup(2, [surface], tag=1)\n        gmsh.model.setPhysicalName(2, 1, \"Domain\")\n        # -------------------------\n        #  5) Generate and convert\n        # -------------------------\n        gmsh.model.mesh.generate(2)\n\n        # save the mesh because of mpi\n        gmsh.write(\"domain\_with\_extrusions.msh\")\n\n        gmsh.finalize()\n\n    def old\_sym\_create\_mesh(self):\n        comm = MPI.COMM\_SELF\n        if comm.rank == 0:\n            gmsh.initialize()\n            gmsh.model.add(\"domain\_with\_extrusion\")\n\n            L\_X = self.config.L\_X\n            L\_Y = self.config.L\_Y\n            res = self.config.RESOLUTION\n\n            y\_max = L\_Y + self.config.SOURCE\_HEIGHT\n            # Define points for the base rectangle\n            p0 = gmsh.model.geo.addPoint(0, 0, 0, meshSize=res)  # bottom left\n            p1 = gmsh.model.geo.addPoint(L\_X, 0, 0, meshSize=res)  # bottom right\n            p2 = gmsh.model.geo.addPoint(L\_X, y\_max, 0, meshSize=res)  # top right\n            p3 = gmsh.model.geo.addPoint(0, L\_Y, 0, meshSize=res)  # tl\n\n            # Define lines for the base rectangle\n            l0 = gmsh.model.geo.addLine(p0, p1)  # bottom of base rectangle\n            l1 = gmsh.model.geo.addLine(p1, p2)  # right of base rectangle\n            l3 = gmsh.model.geo.addLine(p3, p0)  # left of base rectangle\n\n            # Define points for the extrusion (source region)\n            x\_min = L\_X - self.config.SOURCE\_WIDTH\n            # bottom left of extrusion:\n            p4 = gmsh.model.geo.addPoint(x\_min, L\_Y, 0, meshSize=res)\n            # top left of extrusion:\n            p7 = gmsh.model.geo.addPoint(x\_min, y\_max, 0, meshSize=res)\n\n            # Define lines for the extrusion\n            l6 = gmsh.model.geo.addLine(p2, p7)  # top of extrusion\n            l7 = gmsh.model.geo.addLine(p7, p4)  # left of extrusion\n\n            # Connect the extrusion to the base rectangle\n            l8 = gmsh.model.geo.addLine(p3, p4)  # top of base rectangle\n\n            # Define curve loops\n            loop\_combined = gmsh.model.geo.addCurveLoop([l0, l1, l6, l7, -l8, l3])\n            surface = gmsh.model.geo.addPlaneSurface([loop\_combined])\n\n            gmsh.model.geo.synchronize()\n            # Physical groups for boundaries\n            gmsh.model.addPhysicalGroup(1, [l0], tag=1)\n            gmsh.model.setPhysicalName(1, 1, \"IsothermalBoundary\")\n            gmsh.model.addPhysicalGroup(1, [l6], tag=3)\n            gmsh.model.setPhysicalName(1, 3, \"TopBoundary\")\n            gmsh.model.addPhysicalGroup(1, [l7, l8, l3], tag=2)\n            gmsh.model.setPhysicalName(1, 2, \"SlipBoundary\")\n            # gmsh.model.addPhysicalGroup(1, [l7, l6], tag=3)\n            # gmsh.model.setPhysicalName(1, 3, \"TopBoundary\")\n            # gmsh.model.addPhysicalGroup(1, [l8, l3], tag=2)\n            # gmsh.model.setPhysicalName(1, 2, \"SlipBoundary\")\n            gmsh.model.addPhysicalGroup(1, [l1], tag=4)\n            gmsh.model.setPhysicalName(1, 4, \"Symmetry\")\n            gmsh.model.addPhysicalGroup(2, [surface], tag=1)\n            gmsh.model.setPhysicalName(2, 1, \"Domain\")\n\n            gmsh.model.mesh.generate(2)\n\n        msh, cell\_markers, facet\_markers = io.gmshio.model\_to\_mesh(gmsh.model, comm, rank=0, gdim=2)\n\n        if comm.rank == 0:\n            gmsh.finalize()\n\n        return msh, cell\_markers, facet\_markers\n"

  },

  {

    "filePath": "../HeatToolbox/src/mesh\_refinement\_study.py",

    "fileContent": "# mesh\_refinement\_study.py\n\nimport numpy as np\nimport subprocess\nimport matplotlib.pyplot as plt\nimport os\n\ndef main():\n    # Define the length scale from your config or set it directly\n    MEAN\_FREE\_PATH = 0.439e-6  # meters\n    KNUDSEN = 1\n    LENGTH = MEAN\_FREE\_PATH / KNUDSEN  # meters\n\n    # List of mesh resolutions to test (from coarser to finer)\n    resolutions = [\n        LENGTH / 0.1,    # Coarser mesh\n        LENGTH / 0.5,    # Coarser mesh\n        LENGTH / 1,    # Coarser mesh\n        LENGTH / 2,\n        LENGTH / 3,\n        LENGTH / 5,    # Default mesh\n        LENGTH / 8,\n        LENGTH / 10,\n        LENGTH / 12,\n        LENGTH / 15,\n        LENGTH / 20,\n        LENGTH / 30,    # Finer mesh\n        # LENGTH / 40,    # Finer mesh\n    ]\n\n    # Number of MPI processes per simulation\n    mpi\_processes = 1  # Adjust based on your system\n\n    # Initialize a list to store average temperatures\n    avg\_temps = []\n\n    for res in resolutions:\n        # Construct the command with mpirun\n        command = [\n            \"python3\",\n            \"src/main.py\",\n            \"--res\",\n            str(res),\n            \"--visualize\",\n            \"none\",\n            \"--sym\",\n            # \"--blank\",  # Uncomment if needed\n        ]\n\n        print(f\"\\nRunning simulation with resolution: {res:.2e} m\")\n        \n        # Define log file name\n        log\_file = f\"simulation\_res\_{res:.2e}.log\"\n        \n        try:\n            with open(log\_file, \"w\") as f:\n                result = subprocess.run(\n                    command,\n                    stdout=f,\n                    stderr=subprocess.STDOUT,\n                    text=True,\n                    check=True  # Raise an error if the simulation fails\n                )\n        except subprocess.CalledProcessError as e:\n            print(f\"Simulation failed at resolution {res:.2e} m.\")\n            print(f\"Check log file {log\_file} for details.\")\n            avg\_temps.append((res, None))\n            continue\n\n        # Read the log file to extract average temperature\n        avg\_temp = None\n        with open(log\_file, \"r\") as f:\n            for line in f:\n                if \"Average temperature\" in line:\n                    try:\n                        temp\_str = line.split(\":\")[1].strip().split()[0]\n                        avg\_temp = float(temp\_str)\n                        avg\_temps.append((res, avg\_temp))\n                    except (IndexError, ValueError):\n                        print(f\"Error parsing temperature from line: {line}\")\n                        avg\_temps.append((res, None))\n                    break\n            else:\n                print(f\"Average temperature not found in log file for resolution {res:.2e} m.\")\n                avg\_temps.append((res, None))\n\n        print(f\"Average Temperature for resolution {res:.2e} m: {avg\_temp} K\")\n\n        # Optionally, delete the log file after parsing to save space\n        # os.remove(log\_file)\n\n    # After running all simulations, print and save the results\n    print(\"\\nMesh Refinement Study Results:\")\n    for res, temp in avg\_temps:\n        if temp is not None:\n            print(f\"Resolution: {res:.2e} m, Average Temperature: {temp:.6f} K\")\n        else:\n            print(f\"Resolution: {res:.2e} m, Average Temperature: N/A\")\n\n    # Convert resolutions and temperatures to numpy arrays for plotting\n    res\_array = np.array([res for res, temp in avg\_temps if temp is not None])\n    temp\_array = np.array([temp for res, temp in avg\_temps if temp is not None])\n\n    # Plot average temperature vs. mesh resolution\n    plt.figure(figsize=(8, 6))\n    plt.loglog(res\_array, temp\_array, marker='o', linestyle='-')\n    plt.xlabel('Mesh Resolution (m)')\n    plt.ylabel('Average Temperature (K)')\n    plt.title('Mesh Refinement Study: Average Temperature vs. Mesh Resolution')\n    plt.grid(True, which=\"both\", ls=\"--\")\n    plt.gca().invert\_xaxis()  # Finer meshes (smaller resolutions) on the right\n    plt.tight\_layout()\n    plt.savefig(\"mesh\_refinement\_study\_results.png\")\n    plt.show()\n\n\nif \_\_name\_\_ == \"\_\_main\_\_\":\n    main()\n"

  },

  {

    "filePath": "../HeatToolbox/src/opts/bayes.py",

    "fileContent": "import numpy as np\nfrom image\_processing import z\_to\_img\n\n# external optimizers:\n# implement: only import if you want to use them\ntry:\n    from bayes\_opt import BayesianOptimization\nexcept ImportError:\n    BayesianOptimization = None\n\n\nclass BayesianModule:\n    def \_\_init\_\_(self, solver, model, device, rank, config, logger=None):\n        self.logger = logger\n        self.model = model\n        self.device = device\n        self.rank = rank\n        # Initialize the solver\n        self.solver = solver\n        self.config = config\n        self.N\_sources = len(config.source\_positions)\n\n    def evaluate(self, \*\*kwargs):\n        # Extract latent variables for each source\n        latent\_vectors = []\n        for i in range(self.N\_sources):\n            z = np.array(\n                [\n                    kwargs[f\"z{i}\_1\"],\n                    kwargs[f\"z{i}\_2\"],\n                    kwargs[f\"z{i}\_3\"],\n                    kwargs[f\"z{i}\_4\"],\n                ]\n            ).reshape(1, 4)\n            latent\_vectors.append(z)\n\n        if self.rank == 0:\n            img\_list = []\n            for z in latent\_vectors:\n                sample = z\_to\_img(z, self.model, self.config.vol\_fraction)\n                # If symmetry is enabled, take left half of the image\n                if self.config.symmetry:\n                    sample = sample[:, : sample.shape[1] // 2]\n                img\_list.append(sample)\n        else:\n            img\_list = None\n\n        # Solve the images\n        obj = self.solver.solve\_image(img\_list)\n\n        # Return the objective function value\n        # Assuming you want to maximize 1 / obj\n        return 1 / obj\n\n    def optimize(self, init\_points=10, n\_iter=60):\n        # Prepare bounds for all latent variables\n        pbounds = {}\n        for i in range(self.N\_sources):\n            pbounds.update(\n                {\n                    f\"z{i}\_1\": (-2.5, 2.5),\n                    f\"z{i}\_2\": (-2.5, 2.5),\n                    f\"z{i}\_3\": (-2.5, 2.5),\n                    f\"z{i}\_4\": (-2.5, 2.5),\n                }\n            )\n\n        optimizer = BayesianOptimization(\n            f=self.evaluate, pbounds=pbounds, random\_state=1\n        )\n        optimizer.maximize(init\_points=init\_points, n\_iter=n\_iter)\n\n        # Log each iteration's results\n\n        for i, res in enumerate(optimizer.res):\n            generation\_data = {\n                \"iteration\": i,\n                \"target\": res[\"target\"],\n                \"params\": res[\"params\"],\n            }\n            if self.logger:\n                self.logger.log\_generation\_data(i, generation\_data)\n\n        # Extract best latent vectors\n        best\_params = optimizer.max[\"params\"]\n        best\_z\_list = []\n        for i in range(self.N\_sources):\n            best\_z = np.array(\n                [\n                    best\_params[f\"z{i}\_1\"],\n                    best\_params[f\"z{i}\_2\"],\n                    best\_params[f\"z{i}\_3\"],\n                    best\_params[f\"z{i}\_4\"],\n                ]\n            )\n            best\_z\_list.append(best\_z)\n\n        return best\_z\_list\n"

  },

  {

    "filePath": "../HeatToolbox/src/opts/cmaes.py",

    "fileContent": "# optimization\_module.py\n\n# from bayes\_opt import BayesianOptimization\n# import PIL\n# from PIL import Image\n# from solver\_module import Solver\nimport numpy as np\nfrom image\_processing import z\_to\_img, generate\_images\n\n# external optimizers:\nimport cma\nfrom mpi4py import MPI\nimport os\nfrom post\_processing import PostProcessingModule\n\n\nclass CMAESModule:\n    def \_\_init\_\_(self, solver, model, device, rank, config, logger=None):\n        \"\"\"\n        Initialize the CMAESModule.\n\n        Parameters:\n        - solver: An instance of your solver class that can evaluate a solution.\n        - model: The VAE model used for decoding latent vectors to images.\n        - device: The device (CPU or GPU) to run computations on.\n        - config: The simulation configuration containing settings and parameters.\n        - logger: Optional logger instance for logging optimization progress.\n        \"\"\"\n        self.solver = solver\n        self.model = model\n        self.device = device\n        self.config = config\n        self.logger = logger\n        self.rank = rank\n\n        self.comm = MPI.COMM\_WORLD\n        self.size = self.comm.Get\_size()  # Total number of MPI processes\n\n        self.N\_sources = len(config.source\_positions)\n        self.z\_dim = config.latent\_size\n        # Initialize CMA-ES parameters\n        self.init\_z = np.zeros(self.z\_dim \* self.N\_sources)  # Initial latent vector\n        self.sigma0 = 0.5  # Initial standard deviation\n\n        # Directory for CMA logs\n        self.cma\_log\_dir = os.path.join(self.logger.log\_dir, \"cma\_logs\") if self.logger else \"cma\_logs\"\n        if self.rank == 0:\n            os.makedirs(self.cma\_log\_dir, exist\_ok=True)\n\n        timeout\_parts = list(map(int, config.walltime.split(\":\")))\n        self.timeout = (\n            timeout\_parts[0]\*3600 + timeout\_parts[1] \* 60 + timeout\_parts[2]\n        )\n        self.timeout \*= 0.98  # Use 98% of walltime as timeout buffer\n\n    def evaluate\_candidate(self, latent\_vectors):\n        \"\"\"\n        Evaluate a candidate solution.\n\n        Parameters:\n        - z: The latent vector representing the candidate solution.\n\n        Returns:\n        - fitness: The fitness value of the candidate (e.g., average temperature).\n        \"\"\"\n        # Decode the latent vector to an image only in the root process\n        img\_list = generate\_images(self.config, latent\_vectors, self.model)\n\n        # Solve the problem using the solver with the generated image\n        fitness = self.solver.solve\_image(img\_list)\n\n        return fitness\n\n    def optimize(self, n\_iter=100):\n        \"\"\"\n        Run the CMA-ES optimization.\n\n        Parameters:\n        - n\_iter: Number of iterations to run the optimization.\n\n        Returns:\n        - best\_z: The best latent vector found during optimization.\n        \"\"\"\n        # Initialize CMA-ES optimizer\n        if self.rank == 0:\n            on\_hpc = self.config.hpc\_enabled\n            cma\_options = {\n                'verb\_filenameprefix': os.path.join(self.cma\_log\_dir, \"outcma\_\"),\n                'verb\_disp': 0 if on\_hpc else 1,  # 100 #v verbosity: display console output every verb\_disp iteration\n                'verb\_log': 1 if on\_hpc else 0,  # verbosity: write data to files every verb\_log iteration\n                'verb\_append': 1,  # initial evaluation counter, if append, do not overwrite output files\n                'timeout': self.timeout,  # Stop after timeout seconds\n                'popsize': self.config.popsize,  # Population size\n                'bounds': self.config.bounds,  # Bounds\n            }\n            es = cma.CMAEvolutionStrategy(self.init\_z, self.sigma0, cma\_options)\n        else:\n            es = None\n\n        for generation in range(self.config.n\_iter):\n            start\_time = MPI.Wtime()\n            if self.rank == 0:\n                # Ask for candidate solutions\n                candidate\_solutions = es.ask()\n            else:\n                candidate\_solutions = None\n\n            candidate\_solutions = self.comm.bcast(candidate\_solutions, root=0)\n\n            # Determine the workload for each process\n            num\_candidates = len(candidate\_solutions)\n            counts = [num\_candidates // self.size] \* self.size\n            for i in range(num\_candidates % self.size):\n                counts[i] += 1\n            offsets = np.cumsum([0] + counts[:-1])\n\n            # Each process selects its subset of candidate solutions\n            start = offsets[self.rank]\n            end = offsets[self.rank] + counts[self.rank]\n            local\_candidates = candidate\_solutions[start:end]\n\n            # Each process evaluates its local candidates\n            local\_fitnesses = []\n\n            for x in local\_candidates:\n                # Split latent vectors per source\n                latent\_vectors = [\n                    x[i \* self.z\_dim : (i + 1) \* self.z\_dim]\n                    for i in range(self.N\_sources)\n                ]  # Split latent vectors per source\n                fitness = self.evaluate\_candidate(latent\_vectors)\n                local\_fitnesses.append(fitness)\n\n            all\_fitness = self.comm.gather(local\_fitnesses, root=0)\n\n            if self.rank == 0:\n                fitnesses = [fitness for sublist in all\_fitness for fitness in sublist]\n                # Tell CMA-ES the fitnesses of the candidates\n                es.tell(candidate\_solutions, fitnesses)\n                # Log results for this generation\n                es.logger.add()  # Add generation data to the log files\n\n                # Pair each candidate solution with its fitness\n                results = list(zip(candidate\_solutions, fitnesses))\n\n                # Find the candidate with the minimum fitness\n                best\_solution, min\_fitness = min(results, key=lambda s: s[1])\n                generation\_time = MPI.Wtime() - start\_time  # Measure time for the generation\n\n                # Logging and displaying progress\n                es.disp()\n                if self.logger:\n                    self.logger.log\_generation\_data(\n                        generation, {\n                            \"best\_value\": min(fitnesses),\n                            \"generation\_time\": generation\_time,\n                            \"population\_size\": len(candidate\_solutions),\n                        }\n                    )\n\n        # After optim, get the best solution found\n        if self.rank == 0:\n            result = es.result\n            best\_z = result.xbest  # Best latent vector\n            self.save\_cma\_plots()  # Save CMA-ES plots\n        else:\n            best\_z = None\n\n        best\_z = self.comm.bcast(best\_z, root=0)\n\n        return best\_z\n\n    def save\_cma\_plots(self):\n        \"\"\"\n        Generate and save plots from the CMA-ES optimization.\n\n        Parameters:\n        - es: The CMAEvolutionStrategy instance.\n        \"\"\"\n        import matplotlib\n        matplotlib.use('Agg')  # Ensure we're using a non-interactive backend\n        import matplotlib.pyplot as plt\n\n        # Plot the data using cma's logger\n        cma.plot(os.path.join(self.cma\_log\_dir, \"outcma\_\"))\n\n        # Save the figure\n        plot\_path = os.path.join(self.cma\_log\_dir, \"cmaes\_convergence.png\")\n        cma.s.figsave(plot\_path)  # Save all current figures to a file\n\n        print(f\"CMA-ES convergence plot saved to {plot\_path}\")\n\n"

  },

  {

    "filePath": "../HeatToolbox/src/opts/nsga.py",

    "fileContent": "\n''' module docstring'''\nfrom pymoo.core.problem import Problem\nfrom pymoo.algorithms.moo.nsga2 import NSGA2\nfrom pymoo.core.termination import TerminateIfAny\nfrom pymoo.optimize import minimize\nfrom pymoo.termination.max\_time import TimeBasedTermination\nfrom pymoo.termination.default import DefaultMultiObjectiveTermination\nfrom pymoo.util.display.multi import MultiObjectiveOutput\nfrom pymoo.core.callback import Callback\nfrom pymoo.core.population import Population\nfrom pymoo.operators.mutation.pm import PolynomialMutation\n# from pymoo.algorithms.moo.nsga2 import RankAndCrowdingSurvival\nfrom pymoo.operators.survival.rank\_and\_crowding import RankAndCrowding\n\n\nimport torch\nimport numpy as np\nimport logging\nimport json\nimport os\nimport pickle\nimport copy\nfrom image\_processing import z\_to\_img\n# from memory\_profiler import profile\n# from .utils.logging\_utils import generate\_log\_name, initialize\_logger\_folder\n\n\nclass CloakProblem(Problem):\n    '''setting up optimization problem for a bi-objective thermal cloak'''\n    def \_\_init\_\_(self, settings):\n        N\_OBJ = len(self.z\_dim \* self.N\_sources)\n        # print(N\_OBJ)\n        super().\_\_init\_\_(n\_var=self.ass.ngen,\n                         n\_obj=N\_OBJ,  # how many objectives considering\n                         n\_constr=0,  # zero constraints being used\n                         xl=-10,  # Lower bound for variables\n                         xu=10)  # Upper bound for variables\n\n    def \_evaluate(self, x, out, \*args, \*\*kwargs):\n        '''parameters: x is the population filled with genes\n        '''\n        losses = []\n        for i, gene in enumerate(x):\n            # Convert to torch and reshape\n            gene\_torch = torch.from\_numpy(np.array(gene)).cuda()\n            gene\_torch = torch.reshape(gene\_torch, (1, -1))\n\n            # Calculate objective\n            loss = self.ass.sample\_eval(gene\_torch)\n\n            # Append loss to list\n            losses.append(loss)\n\n            log\_entry = {\n                \"subject\": i,\n                \"loss\": loss\n            }\n            logging.info(json.dumps(log\_entry))\n            # Decode the latent vector to an image only in the root process\n            img\_list = []\n            for z in gene\_torch:\n                img = z\_to\_img(z.reshape(1, -1), self.model, self.config.vol\_fraction)\n                # Apply symmetry if enabled\n                if self.config.symmetry:\n                    img = img[:, : img.shape[1] // 2]\n                img\_list.append(img)\n            # Solve the problem using the solver with the generated image\n            fitness = self.solver.solve\_image(img\_list)\n\n        out[\"F\"] = np.array(losses)\n\n    def \_\_deepcopy\_\_(self, memo):\n        # Create a shallow copy\n        cls = self.\_\_class\_\_\n        new\_instance = cls.\_\_new\_\_(cls)\n        memo[id(self)] = new\_instance\n\n        # Copy other attributes, but skip `self.ass`\n        for k, v in self.\_\_dict\_\_.items():\n            if k != \"ass\":\n                setattr(new\_instance, k, copy.deepcopy(v, memo))\n\n        return new\_instance\n\n    # Modify pickling behavior\n    def \_\_reduce\_\_(self):\n        # Return a callable (usually a class or function) and a tuple of\n        # arguments to pass to the callable. The callable is used to recreate\n        # the object when deserializing. The '\_recreate' method can be a\n        # @staticmethod or another external function.\n        return (self.\_recreate, (self.n\_var, self.n\_obj, self.n\_constr,\n                                 self.xl, self.xu))\n\n    @staticmethod\n    def \_recreate(n\_var, n\_obj, n\_constr, xl, xu):\n        # This method will be called when deserializing.\n        obj = CloakProblem.\_\_new\_\_(CloakProblem)  # Create a new instance\n        # obj.n\_var = n\_var\n        # obj.n\_obj = n\_obj\n        # obj.n\_constr = n\_constr\n        # obj.xl = xl\n        # obj.xu = xu\n        # NOTE: self.ass is not set here. If you need to reinitialize it after\n        # deserialization, you should do it outside of the pickling process or\n        # add additional logic.\n        return obj\n\n\nclass CustomOutput(MultiObjectiveOutput):\n    def update(self, algorithm):\n        super().update(algorithm)\n\n        # Log the data to a file\n        log\_entry = {\n            \"generation\": algorithm.n\_gen,\n            \"n\_non\_dom\": len(algorithm.opt),\n            \"eps\": self.eps.value,\n            \"indicator\": self.indicator.value,\n        }\n        logging.info(json.dumps(log\_entry))\n\n\nclass MyCallback(Callback):\n\n    def \_\_init\_\_(self) -> None:\n        super().\_\_init\_\_()\n        self.n\_evals = []\n        self.opt = []\n\n    def notify(self, algorithm):\n        # Append row of \"F\" with best sum of objectives\n        self.n\_evals.append(algorithm.evaluator.n\_eval)\n        self.opt.append(algorithm.opt.get('F'))\n\n\nclass CustomNSGA2(NSGA2):\n\n    def \_advance(self, infills=None, \*\*kwargs):\n        '''calls this instead of \_advance in GeneticAlgorithm class\n        Normalize objective values before survival and setting pareto front\n        '''\n        # the current population\n        pop = self.pop\n\n        # merge the offsprings with the current population\n        if infills is not None:\n            pop = Population.merge(self.pop, infills)\n\n        # This part is new\n        # Get the ideal and nadir points from the Pareto front\n        pf = self.opt.get('F')\n        nadir = np.max(pf, axis=0)\n        ideal = np.min(pf, axis=0)\n\n        # Log ideal and nadir\n        log\_entry = {\n            \"ideal\": list(ideal),\n            \"nadir\": list(nadir)\n        }\n        logging.info(json.dumps(log\_entry))\n\n        # Normalize the objective values of the merged population\n        F = pop.get('F')\n        F\_normalized = (F - ideal) / (nadir - ideal)\n        pop.set('F', F\_normalized)\n        # self.pop = ?\n        # end new part\n\n        # execute the survival to find the fittest solutions\n        self.pop = self.survival.do(self.problem, pop, n\_survive=self.pop\_size, algorithm=self, \*\*kwargs)\n\n\ndef optim\_main(config, log\_fold\_path=None):\n    '''main function for optimization'''\n\n    # Set up problem and algorithm\n    problem = CloakProblem(config)\n\n    algorithm = NSGA2(\n        pop\_size=config['genetic']['nsubjects'],\n        n\_offsprings=config['genetic']['noffsprings'],\n        # sampling=FloatRandomSampling(),\n        # crossover=SBX(),\n        mutation=PolynomialMutation(prob=0.95, eta=10),\n        survival=RankAndCrowding(crowding\_func=\"ce\"),\n        eliminate\_duplicates=True,\n        output=CustomOutput()\n    )\n\n    # RSNGA2\n    # from pymoo.algorithms.moo.rnsga2 import RNSGA2\n    # # Define reference points\n    # ref\_points = np.array([[0.5, 0.5, 0.5]])\n\n    # # Get Algorithm\n    # algorithm = RNSGA2(\n    #     ref\_points=ref\_points,\n    #     pop\_size=100, #config['genetic']['nsubjects'],\n    #     epsilon=0.01,\n    #     normalization='front',\n    #     extreme\_points\_as\_reference\_points=False,\n    #     weights=np.array([0.5, 0.5, 0.5]),\n    #     output=CustomOutput()\n    # )\n\n    # Termination criterion\n    TimeTerm = TimeBasedTermination(config['genetic']['maxtime'])\n    DefaultTerm = DefaultMultiObjectiveTermination(n\_max\_evals=config['genetic']['niterations'])\n    termination = TerminateIfAny(DefaultTerm, TimeTerm)\n\n    print(\"Starting minimization...\")\n    res = minimize(problem,\n                   algorithm,\n                   termination=termination,\n                   callback=MyCallback(),\n                   save\_history=False,\n                   verbose=True)\n\n    print(\"Best solution found: \\nX = %s\\nF = %s\" % (res.X, res.F))\n\n    # For post-processing\n    print(\"Copying to pickle object...\")\n    with open(os.path.join(log\_fold\_path, \"ResultObject.pk1\"), \"wb\") as f:\n        pickle.dump(res, f)\n\n    # End of main\n    print(\"Optim over.\")\n"

  },

  {

    "filePath": "../HeatToolbox/src/post\_processing.py",

    "fileContent": "# post\_processing.py\n\nimport numpy as np\nfrom dolfinx import fem, la, plot, geometry\nimport matplotlib.pyplot as plt\nimport ufl\n# from mpi4py import MPI\nimport os\ntry:\n    import pyvista as pv\nexcept ImportError:\n    pv = None\n\n\nclass PostProcessingModule:\n    def \_\_init\_\_(self, rank, config, logger=None):\n        self.rank = rank\n        self.config = config\n        self.logger = logger\n        self.is\_off\_screen = self.config.plot\_mode == \"screenshot\"\n\n        # make sure self.logger.log\_dir / visualization exists\n        if self.rank == 0:\n            os.makedirs(os.path.join(self.logger.log\_dir, \"visualization\"), exist\_ok=True)\n\n    def postprocess\_results(self, q, T, V, msh, gamma):\n        global\_top, global\_geom, global\_ct, global\_vals = self.gather\_mesh\_on\_rank0(\n            msh, V, T\n        )\n        \_, \_, \_, global\_gamma = self.gather\_mesh\_on\_rank0(msh, V, gamma)\n\n        viz = self.config.visualize\n        if self.rank == 0:\n            # print(f\"(D) Norm of flux coefficient vector (monolithic, direct): {norm\_q}\")\n            # print(f\"(D) Norm of temp coefficient vector (monolithic, direct): {norm\_T}\")\n\n            if viz[\"gamma\"] and global\_gamma is not None:\n                self.plot\_scalar\_field(\n                    global\_top,\n                    global\_ct,\n                    global\_geom,\n                    global\_gamma,\n                    field\_name=\"gamma\",\n                    show\_edges=False,\n                )\n\n            if viz[\"temperature\"] and global\_vals is not None:\n                self.plot\_scalar\_field(\n                    global\_top,\n                    global\_ct,\n                    global\_geom,\n                    global\_vals,\n                    field\_name=\"T\",\n                    # clim=(0, 0.5),\n                )\n\n        # Code for flux plotting (only runs when not in parallel)\n        if msh.comm.size == 1:\n            if viz[\"flux\"]:\n                self.plot\_vector\_field(q, msh)\n            if viz[\"profiles\"]:\n                self.plot\_profiles(q, T, msh)\n        else:\n            if self.rank == 0:\n                print(\"Flux and profiles plotting is disabled when running in parallel.\")\n                print(\"Please run the code in serial to enable flux and profiles plotting.\")\n\n    def plot\_profiles(self, q, T, msh):\n        q\_x, q\_y = q.split()  # extract components\n        curl\_q = self.calculate\_curl(q, msh, plot\_curl=False)\n        eff\_cond = self.calculate\_eff\_thermal\_cond(q, T, msh)\n        # GET TEMPERATURE and FLUX PROFILES #\n        x\_char = self.config.L\_X if self.config.symmetry else self.config.L\_X / 2\n        # horizontal line:\n        x\_end = x\_char\n        y\_val = self.config.L\_Y - self.config.LENGTH / 8\n        (x\_vals, T\_x) = self.get\_temperature\_line(T, msh, \"horizontal\", start=0, end=x\_end, value=y\_val)\n        (\_, q\_x\_vals\_horiz) = self.get\_temperature\_line(q\_x, msh, \"horizontal\", start=0, end=x\_end, value=y\_val)\n        (\_, curl\_vals\_horiz) = self.get\_temperature\_line(curl\_q, msh, \"horizontal\", start=0, end=x\_end, value=y\_val)\n\n        # vertical line:\n        y\_end = self.config.L\_Y + self.config.SOURCE\_HEIGHT\n        x\_val = x\_char - self.config.LENGTH \* 3 / 8\n        (y\_vals, T\_y) = self.get\_temperature\_line(T, msh, \"vertical\", start=0, end=y\_end, value=x\_val)\n        (\_, q\_y\_vals\_vert) = self.get\_temperature\_line(q\_y, msh, \"vertical\", start=0, end=y\_end, value=x\_val)\n        (\_, curl\_vals\_vert) = self.get\_temperature\_line(curl\_q, msh, \"vertical\", start=0, end=y\_end, value=x\_val)\n        (\_, eff\_cond\_vals) = self.get\_temperature\_line(eff\_cond, msh, \"vertical\", start=0, end=y\_end, value=x\_val)\n        # normalize x and y vals by config.ell\_si\n        x\_vals = (x\_vals[-1] - x\_vals) / self.config.ELL\_SI\n        y\_vals = (y\_vals[-1] - y\_vals) / self.config.ELL\_SI\n\n        # PLOT TEMP PROFILES #\n        fig, ax = plt.subplots(figsize=(10, 5))\n        fig.suptitle(\"Temperature profiles\")\n        ax.plot(x\_vals, T\_x, color='red', label=\"T(x) - Horizontal Line\")\n        ax.plot(y\_vals, T\_y, color='blue', label=\"T(y) - Vertical Line\")\n        ax.set\_xlabel(\"Position (normalized)\")\n        ax.set\_ylabel(\"Temperature (T)\")\n        ax.legend()\n        if self.is\_off\_screen:\n            if self.logger:\n                self.logger.save\_image(fig, \"temperature\_profiles.png\")\n            else:\n                plt.savefig(\"temperature\_profiles.png\")\n        else:\n            plt.show()\n\n        # PLOT FLUX PROFILES #\n        q\_x\_vals\_horiz \*= -1  # flip the sign of the flux\n        # 2 subplot:\n        fig, axs = plt.subplots(1, 2, figsize=(10, 4))\n        axs[0].plot(y\_vals, q\_y\_vals\_vert, color='blue', label=\"Vert flux - Vertical Line\")\n        axs[1].plot(x\_vals, q\_x\_vals\_horiz, color='red', label=\"Horiz flux - Horizontal Line\")\n        # make axis titles\n        axs[0].set\_xlabel(\"Position (normalized)\")\n        axs[0].set\_ylabel(\"Heat flux vertical [W/m^2]\")\n        axs[1].set\_xlabel(\"Position (normalized)\")\n        axs[1].set\_ylabel(\"Heat flux horizontal [W/m^2]\")\n        if self.config.plot\_mode == \"screenshot\":\n            if self.logger:\n                self.logger.save\_image(fig, \"flux\_profiles.png\")\n            else:\n                plt.savefig(\"flux\_profiles.png\")\n        else:\n            plt.show()\n\n        # PLOT CURL AND EFF CONDUCTIVITY PROFILES #\n        fig, axs = plt.subplots(1, 2, figsize=(10, 4))\n        curl\_vals\_horiz \*= -1  # flip the sign of the curl\n        curl\_vals\_vert \*= -1  # flip the sign of the curl\n        axs[0].plot(x\_vals, curl\_vals\_horiz, color='red', label=\"Curl(q) - Horizontal Line\")\n        axs[0].plot(y\_vals, curl\_vals\_vert, color='blue', label=\"Curl(q) - Vertical Line\")\n        axs[1].plot(y\_vals, eff\_cond\_vals, color='blue', label=\"Effective conductivity - Vertical Line\")\n        if self.config.plot\_mode == \"screenshot\":\n            if self.logger:\n                self.logger.save\_image(fig, \"curl\_and\_cond\_profiles.png\")\n            else:\n                plt.savefig(\"curl\_and\_cond\_profiles.png\")\n        else:\n            plt.show()\n\n    def gather\_mesh\_on\_rank0(self, mesh, V, function, root=0):\n        \"\"\"\n        Gathers mesh data (topology, cell types, geometry) and solution data (u) from all ranks to rank 0.\n\n        Parameters:\n        mesh: dolfinx.Mesh\n            The distributed mesh.\n        V: dolfinx.FunctionSpace\n            The function space associated with the function.\n        function: dolfinx.Function\n            The function whose data needs to be gathered (e.g., temperature field).\n        root: int\n            The rank on which to gather data (default is 0).\n\n        Returns:\n        On rank 0:\n            - root\_top: np.ndarray (global topology)\n            - root\_geom: np.ndarray (global geometry)\n            - root\_ct: np.ndarray (global cell types)\n            - root\_vals: np.ndarray (global function values)\n        On other ranks:\n            - None, None, None, None\n        \"\"\"\n        comm = mesh.comm\n        rank = comm.rank\n\n        # Create local VTK mesh data structures\n        topology, cell\_types, geometry = plot.vtk\_mesh(mesh, mesh.topology.dim)\n\n        # Get the number of cells and DOFs (degrees of freedom) for local partition\n        num\_cells\_local = mesh.topology.index\_map(mesh.topology.dim).size\_local\n        num\_dofs\_local = V.dofmap.index\_map.size\_local \* V.dofmap.index\_map\_bs\n\n        # Number of DOFs per cell (assuming uniform cells)\n        num\_dofs\_per\_cell = topology[0]\n\n        # Get the DOF indices from the topology array\n        topology\_dofs = (np.arange(len(topology)) % (num\_dofs\_per\_cell + 1)) != 0\n\n        # Map to global DOF indices\n        global\_dofs = V.dofmap.index\_map.local\_to\_global(topology[topology\_dofs].copy())\n\n        # Replace local DOF indices with global DOF indices\n        topology[topology\_dofs] = global\_dofs\n\n        # Gather mesh and function data on the root process\n        global\_topology = comm.gather(\n            topology[: (num\_dofs\_per\_cell + 1) \* num\_cells\_local], root=root\n        )\n        global\_geometry = comm.gather(\n            geometry[: V.dofmap.index\_map.size\_local, :], root=root\n        )\n        global\_ct = comm.gather(cell\_types[:num\_cells\_local], root=root)\n        global\_vals = comm.gather(function.x.array[:num\_dofs\_local], root=root)\n\n        if rank == root:\n            # Stack the data from all ranks on the root process\n            root\_geom = np.vstack(global\_geometry)\n            root\_top = np.concatenate(global\_topology)\n            root\_ct = np.concatenate(global\_ct)\n            root\_vals = np.concatenate(global\_vals)\n\n            return root\_top, root\_geom, root\_ct, root\_vals\n\n        return None, None, None, None\n\n    def plot\_scalar\_field(\n        self,\n        global\_top,\n        global\_ct,\n        global\_geom,\n        values,\n        field\_name=\"field\",\n        clim=None,\n        show\_edges=False,\n    ):\n        grid = pv.UnstructuredGrid(global\_top, global\_ct, global\_geom)\n        grid.point\_data[field\_name] = values.real\n        grid.set\_active\_scalars(field\_name)\n\n        # Plot the scalar field\n        plotter = pv.Plotter(off\_screen=self.is\_off\_screen)\n        plotter.add\_mesh(grid, cmap=\"coolwarm\", show\_edges=show\_edges, clim=clim)\n        plotter.view\_xy()\n        if self.is\_off\_screen:\n            if self.logger:\n                filepath = os.path.join(self.logger.log\_dir, \"visualization\", f\"{field\_name}\_field.png\")\n                plotter.screenshot(\n                    filepath,\n                    transparent\_background=False,\n                    window\_size=[1000, 1000],\n                )\n            else:\n                plotter.screenshot(\n                    f\"{field\_name}\_field.png\",\n                    transparent\_background=False,\n                    window\_size=[1000, 1000],\n                )\n        else:\n            print(\"Plotting the scalar field interactively.\")\n            # Display interactively if interactive mode\n            plotter.show()\n\n    def plot\_vector\_field(self, q, msh):\n        gdim = msh.geometry.dim\n        V\_dg = fem.functionspace(msh, (\"DG\", 2, (gdim,)))\n        q\_dg = fem.Function(V\_dg)\n        q\_copy = q.copy()\n        q\_dg.interpolate(q\_copy)\n\n        V\_cells, V\_types, V\_x = plot.vtk\_mesh(V\_dg)\n        V\_grid = pv.UnstructuredGrid(V\_cells, V\_types, V\_x)\n        Esh\_values = np.zeros((V\_x.shape[0], 3), dtype=np.float64)\n        Esh\_values[:, : msh.topology.dim] = q\_dg.x.array.reshape(\n            V\_x.shape[0], msh.topology.dim\n        ).real\n        V\_grid.point\_data[\"u\"] = Esh\_values\n\n        plotter = pv.Plotter(off\_screen=self.is\_off\_screen)\n        plotter.add\_text(\"magnitude\", font\_size=12, color=\"black\")\n        plotter.add\_mesh(V\_grid.copy(), show\_edges=False)\n        plotter.view\_xy()\n        plotter.link\_views()\n        if self.is\_off\_screen:\n            if self.logger:\n                filepath = os.path.join(self.logger.log\_dir, \"visualization\", \"flux\_field.png\")\n                plotter.screenshot(\n                    filepath,\n                    transparent\_background=False,\n                    window\_size=[1000, 1000],\n                )\n            else:\n                plotter.screenshot(\n                    \"flux\_field.png\",\n                    transparent\_background=False,\n                    window\_size=[1000, 1000],\n                )\n        else:\n            print(\"Plotting the vector field interactively.\")\n            # Display interactively if interactive mode\n            plotter.show()\n\n    def get\_temperature\_line(\n        self,\n        T\_results,\n        msh,\n        line\_orientation=\"horizontal\",\n        start=0.0,\n        end=1.0,\n        value=0.5,\n        num\_points=200,\n    ):\n        # tol = 1e-9  # Small tolerance to avoid hitting boundaries exactly\n        tol = self.config.LENGTH \* 1e-3  # Small tolerance to avoid hitting boundaries exactly\n        points = np.zeros((3, num\_points))\n\n        if line\_orientation == \"horizontal\":\n            x\_coords = np.linspace(start + tol, end - tol, num\_points)\n            points[0] = x\_coords\n            points[1] = value\n        elif line\_orientation == \"vertical\":\n            y\_coords = np.linspace(start + tol, end - tol, num\_points)\n            points[0] = value\n            points[1] = y\_coords\n        else:\n            raise ValueError(\n                \"line\_orientation must be either 'horizontal' or 'vertical'\"\n            )\n\n        # Create bounding box tree for the mesh\n        bb\_tree = geometry.bb\_tree(msh, msh.topology.dim)\n\n        # Find cells whose bounding-box collide with the the points\n        cell\_candidates = geometry.compute\_collisions\_points(bb\_tree, points.T)\n\n        # Choose one of the cells that contains the point\n        colliding\_cells = geometry.compute\_colliding\_cells(msh, cell\_candidates, points.T)\n\n        cells = []\n        points\_on\_proc = []\n        # Get the temperature values at the points\n        for i, point in enumerate(points.T):\n            if len(colliding\_cells.links(i)) > 0:\n                points\_on\_proc.append(point)\n                cells.append(colliding\_cells.links(i)[0])\n\n        points\_on\_proc = np.array(points\_on\_proc, dtype=np.float64)\n        T\_values = T\_results.eval(points\_on\_proc, cells)\n\n        if line\_orientation == \"horizontal\":\n            # Return the coordinates and their corresponding temperature values\n            return (points[0], T\_values)\n        elif line\_orientation == \"vertical\":\n            # Return the coordinates and their corresponding temperature values\n            return (points[1], T\_values)\n\n    def calculate\_curl(self, q, msh, plot\_curl=False):\n\n        def curl\_2d(q: fem.Function):\n            # Returns the z-component of the 2D curl as a scalar\n            return q[1].dx(0) - q[0].dx(1)\n\n        V\_curl = fem.functionspace(msh, (\"DG\", 1))  # DG space for scalar curl\n        curl\_function = fem.Function(V\_curl)\n\n        curl\_flux\_calculator = fem.Expression(curl\_2d(q), V\_curl.element.interpolation\_points())\n        curl\_function.interpolate(curl\_flux\_calculator)\n\n        if plot\_curl:\n            V\_cells, V\_types, V\_x = plot.vtk\_mesh(V\_curl)\n            curl\_values = curl\_function.x.array\n            curl\_grid = pv.UnstructuredGrid(V\_cells, V\_types, V\_x)\n            curl\_grid.point\_data[\"Curl\"] = curl\_values\n            curl\_grid.set\_active\_scalars(\"Curl\")\n            # Plot the curl field\n            plotter = pv.Plotter()\n            plotter.add\_mesh(curl\_grid, cmap=\"coolwarm\", show\_edges=False)\n            plotter.view\_xy()\n            plotter.show()\n\n        return curl\_function\n\n    def calculate\_eff\_thermal\_cond(self, q, T, msh):\n        def heat\_flux\_magnitude(q):\n            q\_x, q\_y = q[0], q[1]\n            return ufl.sqrt(q\_x\*\*2 + q\_y\*\*2)\n\n        def temperature\_gradient\_magnitude(T):\n            grad\_T = ufl.grad(T)\n            return ufl.sqrt(grad\_T[0]\*\*2 + grad\_T[1]\*\*2)\n\n        def k\_cond(q, T):\n            q\_magnitude = heat\_flux\_magnitude(q)\n            grad\_T\_magnitude = temperature\_gradient\_magnitude(T)\n            return q\_magnitude / (grad\_T\_magnitude)\n\n        V\_cond = fem.functionspace(msh, (\"DG\", 1))  # DG space for scalar curl\n        cond\_function = fem.Function(V\_cond)\n\n        curl\_flux\_calculator = fem.Expression(k\_cond(q, T), V\_cond.element.interpolation\_points())\n        cond\_function.interpolate(curl\_flux\_calculator)\n\n        return cond\_function\n"

  },

  {

    "filePath": "../HeatToolbox/src/run\_gui.py",

    "fileContent": "# from tkinter import Tk\nfrom gui.main\_window import SimulationConfigGUI\n\n\ndef main():\n    # Create the root Tkinter window\n    # root = Tk()\n    # Set up the main GUI window\n    app = SimulationConfigGUI()\n    # Run the Tkinter main loop\n    app.mainloop()\n\n\nif \_\_name\_\_ == \"\_\_main\_\_\":\n    main()\n"

  },

  {

    "filePath": "../HeatToolbox/src/sim\_config.py",

    "fileContent": "import json\nimport os\nfrom petsc4py import PETSc\nimport numpy as np\nfrom mpi4py import MPI\n\n\nclass SimulationConfig:\n    def \_\_init\_\_(self, config):\n        \"\"\"\n        Initialize the simulation configuration.\n\n        Parameters:\n        - config (str or dict): Path to a JSON configuration file or a dictionary containing configuration data.\n        \"\"\"\n        self.comm = MPI.COMM\_WORLD\n        self.size = self.comm.Get\_size()  # Total number of MPI processes\n\n        if isinstance(config, str):  # If a file path is provided\n            if self.comm.rank == 0:\n                print(\"Loading config from file on rank 0\")\n                with open(config, \"r\") as f:\n                    loaded\_config = json.load(f)\n            else:\n                loaded\_config = None\n            # Broadcast the config to all ranks\n            loaded\_config = self.comm.bcast(loaded\_config, root=0)\n            self.config = loaded\_config\n        elif isinstance(config, dict):  # If a dictionary is provided\n            # If a dictionary was directly passed in, just broadcast that\n            if self.comm.rank == 0:\n                loaded\_config = config\n            else:\n                loaded\_config = None\n\n            loaded\_config = self.comm.bcast(loaded\_config, root=0)\n            self.config = loaded\_config\n        else:\n            raise TypeError(\"Config must be either a file path (str) or a dictionary.\")\n\n        # Physical properties\n        self.C = PETSc.ScalarType(1.0)  # Slip parameter\n        self.T\_ISO = PETSc.ScalarType(0.0)\n        self.Q\_L = PETSc.ScalarType(80)\n\n        self.MEAN\_FREE\_PATH = 0.439e-6\n        self.KNUDSEN = self.config.get(\"knudsen\", 1)\n        self.solver\_type = self.config.get(\"solver\_type\", \"gke\")\n\n        # Volume fraction\n        self.vol\_fraction = (\n            self.config.get(\"vf\_value\", 0.2)\n            if self.config.get(\"vf\_enabled\", True)\n            else None\n        )\n\n        # Geometric properties\n        # if solver type is fourier, set length manually to 1\n        if self.solver\_type == \"fourier\":\n            self.LENGTH = 1\n        elif self.solver\_type == \"gke\":\n            self.LENGTH = self.MEAN\_FREE\_PATH / self.KNUDSEN\n        self.L\_X = 25 \* self.LENGTH\n        self.L\_Y = 12.5 \* self.LENGTH\n        self.SOURCE\_WIDTH = self.LENGTH\n        self.SOURCE\_HEIGHT = self.LENGTH \* 0.25\n        self.mask\_extrusion = True\n        self.blank = self.config.get(\"blank\", False)\n\n        # Mesh resolution\n        res = self.config.get(\"res\", 12.0)\n        self.RESOLUTION = self.LENGTH / res if res > 0 else self.LENGTH / 12\n\n        # Material properties\n        self.ELL\_SI = PETSc.ScalarType(self.MEAN\_FREE\_PATH / np.sqrt(5))\n        self.ELL\_DI = PETSc.ScalarType(196e-8)\n        self.KAPPA\_SI = PETSc.ScalarType(141.0)\n        self.KAPPA\_DI = PETSc.ScalarType(600.0)\n\n        # Sources\n        self.sources = self.config.get(\"sources\", [0.5, self.Q\_L])\n        self.process\_sources()\n\n        # Symmetry\n        self.symmetry = self.config.get(\"symmetry\", False)\n        if self.symmetry:\n            self.L\_X /= 2\n\n        # Visualization\n        self.visualize = self.config.get(\"visualize\", [])\n        self.plot\_mode = self.config.get(\"plot\_mode\", \"screenshot\")\n\n        # Optimization parameters\n        self.optim = self.config.get(\"optim\", False)\n        self.optimizer = self.config.get(\"optimizer\", \"cmaes\")\n        self.latent = self.config.get(\"latent\", None)\n        self.latent\_size = self.config.get(\"latent\_size\", 4)\n        self.latent\_method = self.config.get(\"latent\_method\", \"preloaded\")\n        self.walltime = self.config.get(\"walltime\", \"03:00:00\")\n        self.popsize = self.config.get(\"popsize\", 8)\n        self.bounds = self.config.get(\"bounds\", [-2.5, 2.5])\n        self.n\_iter = self.config.get(\"n\_iter\", 100)\n\n        # Logging\n        self.logging\_enabled = self.config.get(\"logging\_enabled\", True)\n        self.log\_name = self.config.get(\"log\_name\", None)  # New: user-defined log name\n        self.log\_dir = os.path.join(\"logs\", self.log\_name)\n        # Ensure the log directory  exists\n        if not os.path.exists(self.log\_dir):\n            os.makedirs(self.log\_dir, exist\_ok=True)\n\n        self.hpc\_enabled = self.config.get(\"hpc\_enabled\", True)\n        self.load\_cma\_result = self.config.get(\"load\_cma\_result\", False)\n\n    def process\_sources(self):\n        sources = self.sources\n        if len(sources) % 2 != 0:\n            raise ValueError(\"Each source must have a position and a heat value.\")\n        source\_pairs = [(sources[i], sources[i + 1]) for i in range(0, len(sources), 2)]\n        self.source\_positions = []\n        self.Q\_sources = []\n        for pos, Q in source\_pairs:\n            if pos < 0 or pos > 1:\n                raise ValueError(\"Source positions must be between 0 and 1.\")\n            self.source\_positions.append(pos)\n            self.Q\_sources.append(PETSc.ScalarType(Q))\n        # Sort sources\n        combined\_sources = sorted(zip(self.source\_positions, self.Q\_sources))\n        self.source\_positions, self.Q\_sources = list(zip(\*combined\_sources))\n"

  },

  {

    "filePath": "../HeatToolbox/src/solver\_fourier\_module.py",

    "fileContent": "import ufl\nfrom mpi4py import MPI\nfrom petsc4py import PETSc\nfrom dolfinx import fem, mesh\nimport dolfinx.fem.petsc  # ghost import\nfrom image\_processing import img\_list\_to\_gamma\_expression\nimport numpy as np\n\n\nclass FourierSolver:\n    def \_\_init\_\_(self, msh, facet\_markers, config):\n        self.msh = msh\n        self.config = config\n        self.facet\_markers = facet\_markers\n\n        # Set up function space and function for temperature T\n        self.V = fem.functionspace(msh, (\"CG\", 1))\n        self.T = fem.Function(self.V)\n        self.T.name = \"Temperature\"\n        self.T.x.array[:] = 0.0  # Initialize T\n\n        # Set up boundary conditions and measures\n        self.define\_boundary\_conditions()\n\n        # Function for material property gamma\n        V\_gamma = fem.functionspace(msh, (\"CG\", 1))\n        self.gamma = fem.Function(V\_gamma)\n\n    def define\_boundary\_conditions(self):\n        # Define boundary conditions and measures\n        self.ds = ufl.Measure(\"ds\", domain=self.msh, subdomain\_data=self.facet\_markers)\n        self.ds\_bottom = self.ds(1)  # Isothermal Boundary\n        self.ds\_slip = self.ds(2)    # Slip Boundary\n\n        # Collect top boundary measures based on source positions\n        num\_sources = len(self.config.source\_positions)\n        top\_tags = list(range(3, 3 + num\_sources))  # Tags start at 3\n        self.ds\_tops = [self.ds(tag) for tag in top\_tags]\n\n        if self.config.symmetry:\n            self.ds\_symmetry = self.ds(4)\n\n        # Set up Dirichlet boundary condition at the bottom (isothermal boundary)\n        T\_D\_bottom = fem.Constant(self.msh, 0.0)\n        facets\_bottom = mesh.locate\_entities\_boundary(\n            self.msh, self.msh.topology.dim - 1, lambda x: np.isclose(x[1], 0.0, atol=1e-8)\n        )\n        dofs\_bottom = fem.locate\_dofs\_topological(self.V, self.msh.topology.dim - 1, facets\_bottom)\n        self.bc\_bottom = fem.dirichletbc(T\_D\_bottom, dofs\_bottom, self.V)\n\n        # Collect all Dirichlet boundary conditions\n        self.bcs = [self.bc\_bottom]\n\n    def define\_variational\_form(self):\n        T = self.T  # Current temperature solution\n        v = ufl.TestFunction(self.V)\n        n = ufl.FacetNormal(self.msh)\n\n        def ramp(gamma, a\_min, a\_max, qa=200):\n            return a\_min + (a\_max - a\_min) \* gamma / (1 + qa \* (1 - gamma))\n\n        ramp\_kappa = ramp(self.gamma, self.config.KAPPA\_SI, self.config.KAPPA\_DI)\n\n        # Variational form for Fourier's heat conduction\n        F = ramp\_kappa \* ufl.inner(ufl.grad(T), ufl.grad(v)) \* ufl.dx\n\n        # Include Neumann boundary conditions (source terms) on top boundaries\n        source\_term = sum(\n            Q\_i \* v \* ds\_top\_i\n            for Q\_i, ds\_top\_i in zip(self.config.Q\_sources, self.ds\_tops)\n        )\n        F += source\_term\n\n        return F\n\n    def solve\_image(self, img\_list):\n        gamma\_expr = img\_list\_to\_gamma\_expression(img\_list, self.config)\n        self.gamma.interpolate(gamma\_expr)\n\n        # Define variational forms\n        F = self.define\_variational\_form()\n        residual = fem.form(F)\n        J = ufl.derivative(F, self.T)\n        jacobian = fem.form(J)\n\n        # Solve the problem\n        self.solve\_problem(residual, jacobian)\n\n        temp\_form = fem.form(self.T \* ufl.dx)\n        temp\_local = fem.assemble\_scalar(temp\_form)\n        temp\_global = temp\_local\n        area = self.config.L\_X \* self.config.L\_Y + self.config.SOURCE\_WIDTH \* self.config.SOURCE\_HEIGHT\n        avg\_temp\_global = temp\_global / area\n        return avg\_temp\_global\n\n    def solve\_problem(self, residual, jacobian):\n        # Create matrix and vector\n        A = fem.petsc.create\_matrix(jacobian)\n        L = fem.petsc.create\_vector(residual)\n\n        solver = PETSc.KSP().create(self.T.function\_space.mesh.comm)\n        solver.setOperators(A)\n        solver.setType(\"cg\")  # Conjugate Gradient for symmetric positive-definite\n        solver.setTolerances(rtol=1e-6, atol=1e-13, max\_it=1000)\n        pc = solver.getPC()\n        pc.setType(\"ilu\")  # Incomplete LU factorization\n\n        solver.setFromOptions()\n\n        # Assemble system\n        with L.localForm() as loc\_L:\n            loc\_L.set(0)\n        A.zeroEntries()\n        fem.petsc.assemble\_matrix(A, jacobian, bcs=self.bcs)\n        A.assemble()\n        fem.petsc.assemble\_vector(L, residual)\n        fem.petsc.apply\_lifting(L, [jacobian], [self.bcs])\n        L.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)\n        fem.petsc.set\_bc(L, self.bcs)\n        L.ghostUpdate(addv=PETSc.InsertMode.INSERT\_VALUES, mode=PETSc.ScatterMode.FORWARD)\n\n        # Solve linear problem\n        solver.solve(L, self.T.x.petsc\_vec)\n        self.T.x.scatter\_forward()\n"

  },

  {

    "filePath": "../HeatToolbox/src/solver\_gke\_module.py",

    "fileContent": "import ufl\nfrom mpi4py import MPI\nfrom petsc4py import PETSc\nfrom dolfinx import fem, mesh\nimport dolfinx.fem.petsc  # ghost import\nfrom image\_processing import img\_list\_to\_gamma\_expression\nimport basix.ufl\nimport numpy as np\n\n\nclass GKESolver:\n    def \_\_init\_\_(self, msh, facet\_markers, config):\n        self.msh = msh\n        self.config = config\n        self.facet\_markers = facet\_markers\n\n        # Set up function spaces and functions\n        P2 = basix.ufl.element(\"CG\", msh.basix\_cell(), 2, shape=(msh.geometry.dim,))\n        P1 = basix.ufl.element(\"CG\", msh.basix\_cell(), 1)\n        TH = basix.ufl.mixed\_element([P2, P1])\n        self.W = fem.functionspace(msh, TH)\n\n        self.U = fem.Function(self.W)  # Current solution (q, T)\n        self.dU = ufl.TrialFunction(self.W)  # Increment (delta q, delta T)\n        (self.v, self.s) = ufl.TestFunctions(self.W)\n        self.U.x.array[:] = 0.0  # Initialize U\n\n        # Set up boundary conditions and measures\n        self.define\_boundary\_conditions()\n        self.perform\_mesh\_checks()\n\n        V\_gamma = fem.functionspace(msh, (\"CG\", 1))\n        self.gamma = fem.Function(V\_gamma)\n\n    def define\_boundary\_conditions(self):\n        # Define boundary conditions and measures\n        self.ds = ufl.Measure(\"ds\", domain=self.msh, subdomain\_data=self.facet\_markers)\n        self.ds\_bottom = self.ds(1)  # Isothermal Boundary\n        self.ds\_slip = self.ds(2)    # Slip Boundary\n\n        # Collect top boundary measures based on source positions\n        num\_sources = len(self.config.source\_positions)\n        top\_tags = list(range(3, 3 + num\_sources))  # Tags start at 3\n        self.ds\_tops = [self.ds(tag) for tag in top\_tags]\n\n        if self.config.symmetry:\n            self.ds\_symmetry = self.ds(99)\n\n        # Set up boundary condition functions and Dirichlet boundary conditions\n        W1 = self.W.sub(1)\n        Q1, \_ = W1.collapse()\n        noslip = fem.Function(Q1)  # Set a constant function for noslip condition\n\n        rtol = self.config.LENGTH \* 1e-3\n        facets = mesh.locate\_entities(self.msh, 1, lambda x: np.isclose(x[1], 0.0, rtol=rtol))\n        dofs = fem.locate\_dofs\_topological((W1, Q1), 1, facets)\n        self.bc0 = fem.dirichletbc(noslip, dofs, W1)\n\n    def define\_variational\_form(self):\n        q, T = ufl.split(self.U)\n        n = ufl.FacetNormal(self.msh)\n\n        def ramp(gamma, a\_min, a\_max, qa=200):\n            return a\_min + (a\_max - a\_min) \* gamma / (1 + qa \* (1 - gamma))\n\n        ramp\_kappa = ramp(self.gamma, self.config.KAPPA\_SI, self.config.KAPPA\_DI)\n        ramp\_ell = ramp(self.gamma, self.config.ELL\_SI, self.config.ELL\_DI)\n\n        viscous\_term = (\n            ramp\_ell \*\* 2\n            \* (\n                ufl.inner(ufl.grad(q), ufl.grad(self.v))\n                + 2 \* ufl.inner(ufl.div(q) \* ufl.Identity(2), ufl.grad(self.v))\n            )\n            \* ufl.dx\n        )\n\n        flux\_continuity = -ufl.div(q) \* self.s \* ufl.dx\n        pressure\_term = - ramp\_kappa \* T \* ufl.div(self.v) \* ufl.dx\n        flux\_term = ufl.inner(q, self.v) \* ufl.dx\n\n        # Source term at the top boundaries\n        source\_term = sum(\n            Q\_i \* ufl.dot(self.v, n) \* ds\_top\_i\n            for Q\_i, ds\_top\_i in zip(self.config.Q\_sources, self.ds\_tops)\n        )\n\n        def u\_t(q):\n            return q - ufl.dot(q, n) \* n\n\n        def t(q, T):\n            return (\n                ramp\_ell \*\* 2 \* (ufl.grad(q) + 2 \* ufl.div(q) \* ufl.Identity(2))\n                - ramp\_kappa \* T \* ufl.Identity(2)\n            ) \* n\n\n        F = (\n            flux\_continuity  # Continuity: ∇⋅q = 0\n            + flux\_term  # Flux term: q ⋅ v\n            + viscous\_term  # Viscous-like term\n            + pressure\_term  # Pressure-like term from ∇T\n            - ufl.dot(n, t(q, T)) \* ufl.dot(self.v, n) \* self.ds\_slip  # Slip boundary condition term\n            - ufl.dot(q, n) \* ufl.dot(n, t(self.v, self.s)) \* self.ds\_slip  # Slip boundary condition term\n            + ramp\_ell \* ufl.dot(u\_t(q), u\_t(self.v)) \* self.ds\_slip  # Slip boundary stabilization term\n            + ufl.dot(q, n) \* ufl.dot(self.v, n) \* self.ds\_slip  # Additional stabilization\n            + source\_term\n        )\n\n        # Symmetry boundary condition if enabled\n        if self.config.symmetry:\n            F += ufl.dot(q, n) \* ufl.dot(self.v, n) \* self.ds\_symmetry\n\n        return F\n\n    def solve\_image(self, img\_list):\n        gamma\_expr = img\_list\_to\_gamma\_expression(img\_list, self.config)\n        # gamma\_expr = img\_to\_gamma\_expression(img\_list[0], self.config)\n        self.gamma.interpolate(gamma\_expr)\n\n        # Define variational forms\n        F = self.define\_variational\_form()\n\n        # Solve the problem\n        self.solve\_problem(F)\n\n        q, T = self.U.sub(0).collapse(), self.U.sub(1).collapse()\n\n        temp\_form = fem.form(T \* ufl.dx)\n        temp\_local = fem.assemble\_scalar(temp\_form)\n        # temp\_global = self.msh.comm.allreduce(temp\_local, op=MPI.SUM)\n        temp\_global = temp\_local\n        area = self.config.L\_X \* self.config.L\_Y + self.config.SOURCE\_WIDTH \* self.config.SOURCE\_HEIGHT\n        avg\_temp\_global = temp\_global / area\n        return avg\_temp\_global\n\n    def solve\_problem(self, F):\n        residual = fem.form(F)\n        J = ufl.derivative(F, self.U, self.dU)\n        jacobian = fem.form(J)\n        # Create matrix and vector\n        A = dolfinx.fem.petsc.create\_matrix(jacobian)\n        L = dolfinx.fem.petsc.create\_vector(residual)\n\n        solver = PETSc.KSP().create(self.U.function\_space.mesh.comm)\n        solver.setOperators(A)\n        solver.setType(\"minres\")\n        solver.setTolerances(rtol=1e-6, atol=1e-13, max\_it=1000)\n        pc = solver.getPC()\n        pc.setType(\"lu\")\n        pc.setFactorSolverType(\"mumps\")\n\n        # Set MUMPS options\n        factor\_mat = pc.getFactorMatrix()\n        factor\_mat.setMumpsIcntl(24, 1)  # Support solving singular matrix\n        factor\_mat.setMumpsIcntl(25, 0)  # Support solving singular matrix\n        factor\_mat.setMumpsIcntl(13, 1)  # Enable parallel root node factorization\n        factor\_mat.setMumpsIcntl(28, 2)  # Use parallel analysis\n        factor\_mat.setMumpsIcntl(14, 100)  # Increase MUMPS working memory\n        factor\_mat.setMumpsIcntl(22, 0)  # Disable out-of-core factorization\n        factor\_mat.setMumpsCntl(1, 1e-9)  # Relative pivoting scale\n        factor\_mat.setMumpsCntl(3, 1e-9)  # Absolute pivoting scale\n        factor\_mat.setMumpsIcntl(1, -1)  # Print all error messages\n        factor\_mat.setMumpsIcntl(2, 3)  # Enable diagnostic printing stats and warnings\n        factor\_mat.setMumpsIcntl(4, 0)  # Set print level verbosity (0-4)\n\n        solver.setFromOptions()\n\n        du = fem.Function(self.W)\n\n        i = 0\n        max\_iterations = 5\n        while i < max\_iterations:\n            # Assemble Jacobian and residual\n            with L.localForm() as loc\_L:\n                loc\_L.set(0)\n            A.zeroEntries()\n            fem.petsc.assemble\_matrix(A, jacobian, bcs=[self.bc0])\n            A.assemble()\n            fem.petsc.assemble\_vector(L, residual)\n            L.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)\n            L.scale(-1)\n\n            # Apply boundary conditions\n            fem.petsc.apply\_lifting(L, [jacobian], [[self.bc0]], x0=[self.U.x.petsc\_vec], alpha=1)\n            fem.petsc.set\_bc(L, [self.bc0], self.U.x.petsc\_vec, 1.0)\n            L.ghostUpdate(\n                addv=PETSc.InsertMode.INSERT\_VALUES, mode=PETSc.ScatterMode.FORWARD\n            )\n\n            # Solve linear problem\n            solver.solve(L, du.x.petsc\_vec)\n            du.x.scatter\_forward()\n\n            # Update solution\n            self.U.x.array[:] += du.x.array\n            i += 1\n\n            # Check convergence\n            correction\_norm = du.x.petsc\_vec.norm(0)\n            if correction\_norm < 1e-5:\n                break\n\n    def perform\_mesh\_checks(self):\n        # Check measures over specific boundaries\n        checks = [\n            # (\"source line\", self.ds\_top, self.config.SOURCE\_WIDTH),\n            (\"isothermal line\", self.ds\_bottom, self.config.L\_X),\n            (\"slip line\", self.ds\_slip, self.config.L\_Y + (self.config.L\_X - self.config.SOURCE\_WIDTH) + self.config.SOURCE\_HEIGHT)\n        ]\n\n        for name, ds\_measure, expected\_value in checks:\n            check\_form = fem.form(PETSc.ScalarType(1) \* ds\_measure)\n            check\_local = fem.assemble\_scalar(check\_form)  # Assemble over cell\n            total\_check = self.msh.comm.allreduce(check\_local, op=MPI.SUM)\n            # if rank == 0:\n            #     # perform assertions\n            #     assert np.isclose(total\_check, expected\_value, rtol=1e-6), f\"Mesh check failed for {name}\"\n"

  },

  {

    "filePath": "../HeatToolbox/src/solver\_joule\_module.py",

    "fileContent": "import ufl\nfrom mpi4py import MPI\nfrom petsc4py import PETSc\nfrom dolfinx import fem, mesh\nimport dolfinx.fem.petsc  # ghost import\nfrom image\_processing import img\_list\_to\_gamma\_expression\nimport numpy as np\n\n\nclass JouleSolver:\n    def \_\_init\_\_(self, msh, facet\_markers, config):\n        self.msh = msh\n        self.config = config\n        self.facet\_markers = facet\_markers\n\n        # Set up function space and function for temperature T\n        self.V = fem.functionspace(msh, (\"CG\", 1))\n        self.T = fem.Function(self.V)\n        self.T.name = \"Temperature\"\n        self.T.x.array[:] = 0.0  # Initialize T\n\n        # Set up boundary conditions and measures\n        self.define\_boundary\_conditions()\n\n        # Function for material property gamma\n        V\_gamma = fem.functionspace(msh, (\"CG\", 1))\n        self.gamma = fem.Function(V\_gamma)\n\n    def define\_boundary\_conditions(self):\n        # Define boundary conditions and measures\n        self.ds = ufl.Measure(\"ds\", domain=self.msh, subdomain\_data=self.facet\_markers)\n        self.ds\_bottom = self.ds(1)  # Isothermal Boundary\n        self.ds\_slip = self.ds(2)    # Slip Boundary\n\n        # Collect top boundary measures based on source positions\n        num\_sources = len(self.config.source\_positions)\n        top\_tags = list(range(3, 3 + num\_sources))  # Tags start at 3\n        self.ds\_tops = [self.ds(tag) for tag in top\_tags]\n\n        if self.config.symmetry:\n            self.ds\_symmetry = self.ds(4)\n\n        # Set up Dirichlet boundary condition at the bottom (isothermal boundary)\n        T\_D\_bottom = fem.Constant(self.msh, 0.0)\n        facets\_bottom = mesh.locate\_entities\_boundary(\n            self.msh, self.msh.topology.dim - 1, lambda x: np.isclose(x[1], 0.0, atol=1e-8)\n        )\n        dofs\_bottom = fem.locate\_dofs\_topological(self.V, self.msh.topology.dim - 1, facets\_bottom)\n        self.bc\_bottom = fem.dirichletbc(T\_D\_bottom, dofs\_bottom, self.V)\n\n        # Collect all Dirichlet boundary conditions\n        self.bcs = [self.bc\_bottom]\n\n    def define\_variational\_form(self):\n        T = self.T  # Current temperature solution\n        v = ufl.TestFunction(self.V)\n        n = ufl.FacetNormal(self.msh)\n\n        def ramp(gamma, a\_min, a\_max, qa=200):\n            return a\_min + (a\_max - a\_min) \* gamma / (1 + qa \* (1 - gamma))\n\n        ramp\_kappa = ramp(self.gamma, self.config.KAPPA\_SI, self.config.KAPPA\_DI)\n\n        # Variational form for Fourier's heat conduction\n        F = ramp\_kappa \* ufl.inner(ufl.grad(T), ufl.grad(v)) \* ufl.dx\n\n        # Include Neumann boundary conditions (source terms) on top boundaries\n        source\_term = sum(\n            Q\_i \* v \* ds\_top\_i\n            for Q\_i, ds\_top\_i in zip(self.config.Q\_sources, self.ds\_tops)\n        )\n        F += source\_term\n\n        return F\n\n    def solve\_image(self, img\_list):\n        gamma\_expr = img\_list\_to\_gamma\_expression(img\_list, self.config)\n        self.gamma.interpolate(gamma\_expr)\n\n        # Define variational forms\n        F = self.define\_variational\_form()\n        residual = fem.form(F)\n        J = ufl.derivative(F, self.T)\n        jacobian = fem.form(J)\n\n        # Solve the problem\n        self.solve\_problem(residual, jacobian)\n\n        temp\_form = fem.form(self.T \* ufl.dx)\n        temp\_local = fem.assemble\_scalar(temp\_form)\n        temp\_global = temp\_local\n        area = self.config.L\_X \* self.config.L\_Y + self.config.SOURCE\_WIDTH \* self.config.SOURCE\_HEIGHT\n        avg\_temp\_global = temp\_global / area\n        return avg\_temp\_global\n\n    def solve\_problem(self, residual, jacobian):\n        # Create matrix and vector\n        A = fem.petsc.create\_matrix(jacobian)\n        L = fem.petsc.create\_vector(residual)\n\n        solver = PETSc.KSP().create(self.T.function\_space.mesh.comm)\n        solver.setOperators(A)\n        solver.setType(\"cg\")  # Conjugate Gradient for symmetric positive-definite\n        solver.setTolerances(rtol=1e-6, atol=1e-13, max\_it=1000)\n        pc = solver.getPC()\n        pc.setType(\"ilu\")  # Incomplete LU factorization\n\n        solver.setFromOptions()\n\n        # Assemble system\n        with L.localForm() as loc\_L:\n            loc\_L.set(0)\n        A.zeroEntries()\n        fem.petsc.assemble\_matrix(A, jacobian, bcs=self.bcs)\n        A.assemble()\n        fem.petsc.assemble\_vector(L, residual)\n        fem.petsc.apply\_lifting(L, [jacobian], [self.bcs])\n        L.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)\n        fem.petsc.set\_bc(L, self.bcs)\n        L.ghostUpdate(addv=PETSc.InsertMode.INSERT\_VALUES, mode=PETSc.ScatterMode.FORWARD)\n\n        # Solve linear problem\n        solver.solve(L, self.T.x.petsc\_vec)\n        self.T.x.scatter\_forward()\n"

  },

  {

    "filePath": "../HeatToolbox/src/vae\_module.py",

    "fileContent": "# vae\_model.py\n\nimport torch\nfrom torch import nn\nfrom mpi4py import MPI\n\n# REMOVE LATER?\nimage\_size = 128\nhidden\_size = 1024\nlatent\_size = 4\n\ndevice = torch.device(\"cpu\")  # Change to \"cuda\" if using GPU\n\n\ndef load\_vae\_model(rank, z\_dim=latent\_size):\n    model = VAE(z\_dim=z\_dim)\n    # model = torch.load(\"./model/model\", map\_location=torch.device(\"cpu\"))\n    model\_path = f\"VAEs/models/128latent{z\_dim}epochs200Alldict\" if z\_dim != 16 else f\"VAEs/models/128latent{z\_dim}epochs500Alldict\" \n    model.load\_state\_dict(torch.load(model\_path, map\_location=device))\n    model.eval()\n    return model\n\n\nclass Flatten(nn.Module):\n    def forward(self, input):\n        return input.view(input.size(0), -1)\n\n\nclass UnFlatten(nn.Module):\n    def forward(self, input, size=hidden\_size):\n        return input.view(input.size(0), size, 1, 1)\n\n\nclass VAE(nn.Module):\n    def \_\_init\_\_(self, image\_channels=1, h\_dim=hidden\_size, z\_dim=latent\_size):\n        super(VAE, self).\_\_init\_\_()\n        self.encoder = nn.Sequential(\n            nn.Conv2d(image\_channels, 16, kernel\_size=4, stride=2),\n            nn.ReLU(),\n            nn.Conv2d(16, 32, kernel\_size=4, stride=2),\n            nn.ReLU(),\n            nn.Conv2d(32, 64, kernel\_size=4, stride=2),\n            nn.ReLU(),\n            nn.Conv2d(64, 128, kernel\_size=4, stride=2),\n            nn.ReLU(),\n            nn.Conv2d(128, 256, kernel\_size=4, stride=2),\n            nn.ReLU(),\n            Flatten(),\n        )\n\n        self.fc1 = nn.Linear(h\_dim, z\_dim)\n        self.fc2 = nn.Linear(h\_dim, z\_dim)\n        self.fc3 = nn.Linear(z\_dim, h\_dim)\n\n        self.decoder = nn.Sequential(\n            UnFlatten(),\n            nn.ConvTranspose2d(h\_dim, 128, kernel\_size=5, stride=2),\n            nn.ReLU(),\n            nn.ConvTranspose2d(128, 64, kernel\_size=5, stride=2),\n            nn.ReLU(),\n            nn.ConvTranspose2d(64, 32, kernel\_size=5, stride=2),\n            nn.ReLU(),\n            nn.ConvTranspose2d(32, 16, kernel\_size=6, stride=2),\n            nn.ReLU(),\n            nn.ConvTranspose2d(16, image\_channels, kernel\_size=6, stride=2),\n            nn.Sigmoid(),\n        )\n\n    def reparameterize(self, mu, logvar):\n        std = logvar.mul(0.5).exp\_()\n        # eps = torch.randn\_like(std)\n        esp = torch.randn(\*mu.size()).to(device)\n        z = mu + std \* esp\n        return z\n\n    def bottleneck(self, h):\n        mu, logvar = self.fc1(h), self.fc2(h)\n        z = self.reparameterize(mu, logvar)\n        return z, mu, logvar\n\n    def encode(self, x):\n        h = self.encoder(x)\n        z, mu, logvar = self.bottleneck(h)\n        return z, mu, logvar\n\n    def decode(self, z):\n        z = self.fc3(z)\n        z = self.decoder(z)\n        return z\n\n    def forward(self, x):\n        z, mu, logvar = self.encode(x)\n        z = self.decode(z)\n        return z, mu, logvar\n\n"

  }

]

```

My notion notes (incomplete):

```

# FYP

27/1/25

- project statement

    - challenges

    - gantt chart

    - novelty

- progress to date

    - longest part

    - do differnet latent spaces

    - do cmaes vs nsga

- future

25/1/25

- do symmetry.

    - had lots of errors

    - trying to debug T = 0

    - source widths are half what they shuold be.

- do nsga.

- make image go further down.

13/1/25

- met with Ricardo

    - lack of symmetry problem.

    - how to fix.

    - vae doesnt seem to be set up for other than 4 latent vector

- why is it still saving config for visualization

- why is load sometimes not working in hpc

-

24/12/24

- Made wfc work knida

- wfc: I need help with a complex task. I have a code for Wave Function Collapse tileset generation in python. I will provide it on a second. This is for PCB design, so the tiles I provide are for pcbs (tracks and components etc). The problem is, the WFC algorithm does not know how to connect.

    - either use pathfinding to guide it or connectivity aware entropy heuristics

19/12/24

reorganize structure for joule heating

27/11/24

results are not symmetric.

adding fourier by doing low knudsen.

but need to add actual fourier

using my previous code (GAOPT) and this: [https://jsdokken.com/dolfinx-tutorial/chapter2/ns\_code1.html](https://jsdokken.com/dolfinx-tutorial/chapter2/ns\_code1.html)

18/11/24

To do

- [ ]  \*\*Optim / batch processing of cmaes\*\*

    - [ ]  Plotting

    - [ ]  memory usage / memory profiling

        - to check its worth it

    - [ ]  Do modularize the parallelization of MUMPS. check if it makes a difference.

    - [ ]  Remove MPI traces for FENCISX now? Then make it \*\*\*Open MP?\*\*\*

- [ ]  \*\*HPC\*\*

    - [ ]  maybe add checkpointing

- [ ]  \*\*GUI and params\*\*:

    - [ ]  \*\*Fix load config\*\*

    - [ ]  \*\*Function to read the best x from the log files of a folder and output the results\*\*

        - [ ]  checks load cma-es confing (new button), unchecks run optim, unchecks enable hpc, doesn’t create config file, has all visualization options on, and runs simulation.

    - [ ]  add mask extrusion

    - [ ]  make optim depend on latent given (low priority, stick with 4 for now)

    - [ ]  toggles:

        - [ ]  cannot have latent vector in args if optim is on. separate the parts of the gui better (low priority)

        - [ ]  turns off visualation frame if parallel on.

        - [ ]  If hpc turned on, viz is not an option?.

    - [ ]  decouple gui loading

17/11/24

fixed issue with timeout. needed the “h”

My bigger goal, besides refactoring the gui/hpc, is to have a better logging and data holding system.

This is what I mean:

- I want configurations to be saved in easy to use formats, like json or yaml.

- I dont think I should have the python file for config in the way that I do (See Code)

- I want hpc scripts to be added to their respective log folder, then sent from there in the hpc.

- I want in the gui:

    - to be able to load a config or log file.

    - I want to be able to open a config file and rerun it.

    - I want to be able to name the log file

The whole logging system will have to change.

I want the log file to:

- if in optimization, to have a optimization results sub folder

- after optimized, we should be able to use this log file and its config file to run on the a specific latent vector (from the optimization). This would save the images in a visualize or something folder with a name for that latent vector.

16/11/24

works on batch parallel. now add to do:

- [ ]  \*\*batch processing of cmaes\*\*

    - [x]  MPI

    - [ ]  memory usage / memory profiling

        - to check its worth it

    - [x]  \*\*Add outputs in full to log file.better logging of optimization: timing and everything\*\*

    - [ ]  Remove MPI traces for FENCISX now? Then make it \*\*\*Open MP?\*\*\*

        - or do i make it all MPI

    - [ ]  Do modularize the parallelization of MUMPS. check if it makes a difference.

    - [ ]  Plotting

- [ ]  \*\*HPC\*\*

    - [ ]  maybe add checkpointing

    - [x]  \*\*add password and sending integration\*\*

    - [x]  \*\*Use Environment Variables:\*\* Prompt the user for their password and store it temporarily in an environment variable that is then used by sshpass.

    - [x]  \*\*Add hpc script to log folder instead of gen command\*\*

    - [x]  Make sshd for hpc.  (folder on mac)

- [ ]  \*\*GUI and params\*\*:

    - [ ]  \*\*Fix load config\*\*

    - [ ]  add mask extrusion

    - [ ]  make optim depend on latent given (low priority, stick with 4 for now)

    - [ ]  cannot have latent vector in args if optim is on. separate the parts of the gui better (low priority)

    - [ ]  turns off visualation frame if parallel on.

    - [x]  Only show Optim and HPC if turned on.

        - [ ]  If hpc turned on, viz is not an option?.

    - [ ]  decouple gui loading

    - [x]  \*\*more optim parameters for CMAES\*\*

    - [x]  use / load previous configs memories.

    - [x]  name log files

    - [x]  make it so gui opens in centre

- [x]  General refactoring

    - [x]  \*\*Use Data Classes:\*\* Replace the inner Args class with Python’s dataclasses for cleaner and more structured configuration management.

        - Maybe just switch to yaml.

- [ ]  implement Globus CLI (low priority)

    - [x]  implement a better workflow for HPC

    - add to gui: post process into same folder.

- [ ]  what about GPU?

- [ ]  decouple imports, in gui and main code.?

add to do:

- add cmaes output to logging

- add cmaes parameters to gui,

- add memory profiling

- optimize hpc workflow

- refractor main

- then hit run

15/11/24

fixed parallelization: was issue with loading mesh

need to save mesh with config on one process then load it on all processes.

do memory profiling

fixing gui, making more organized

- [x]  \*\*batch processing of cmaes\*\*

    - [x]  MPI

        - memory usage / memory profiling

        - check its worth it

    - [ ]  Remove MPI traces for FENCISX now? Then make it \*\*\*Open MP?\*\*\*

        - or do i make it all MPI

    - [ ]  Do modularize the parallelization of MUMPS

- [x]  \*\*create a modularity with the HPC scripts\*\*

    - [ ]  maybe add checkpointing

- [ ]  \*\*GUI and parmas\*\*:

    - [x]  toggle parallelization with number of processes.

    - [ ]  turns off visualation frame if parallel on.

    - [ ]  cannot have latent vector in args if optim is on. separate the parts of the gui better (low priority)

    - [ ]  \*\*more optim parameters for CMAES\*\*

    - [x]  add HPC frame

    - [ ]  make optim depend on latent given (low priority, stick with 4 for now)

- [ ]  \*\*better logging of optimization: timing and everything\*\*

- [ ]  implement Globus CLI (low priority)

    - [ ]  implement a better workflow for HPC

    - add to gui: post process into same folder.

- [ ]  what about GPU?

- [ ]  decouple imports, in gui and main code.

- [ ]  Make sshd for hpc.

14/11/24

Priority

1. Setup fenicsx on 0.9.0

2. Set up batch processing with CMAES

3. Clean up code, especially gui and main.

To do from yesterday

- [ ]  \*\*batch processing of cmaes\*\*

    - MPI

        - memory usage / memory profiling

        - check its worth it

    - Remove MPI traces for FENCISX now? Then make it \*\*\*Open MP?\*\*\*

        - or do i make it all MPI

    - Do modularize the parallelization of MUMPS

- [ ]  \*\*create a modularity with the HPC scripts\*\*

    - add this to gui like parameters and all that

    - and maybe checkpointing

    - [ ]  \*\*more optim parameters for CMAES\*\*

- [ ]  \*\*GUI\*\*:

    - [ ]  toggle parallelization with number of processes.

    - [ ]  turns off viz if parallel on.

    - [ ]  cannot have latent vector in args if optim is on. separate the parts of the gui better (low priority)

- [ ]  \*\*better logging of optimization: timing and everything\*\*

- [ ]  implement Globus CLI (low priority)

    - [ ]  implement a better workflow for HPC

    - add to gui: post process into same folder.

- [ ]  make optim depend on latent given (low priority, stick with 4 for now)

- [ ]  what about GPU?

- [ ]  decouple imports, in gui and main code.

 batch processing with fenicsx is really difficult. borderline impossible. need to make sure its actually worth it….

- do i load the mesh onto each process?

13/11/24

- setup hpc, need to add cmaes

    - made a script

    - [HPC](https://www.notion.so/HPC-a9392ec101d241b2aaef517c8b7529aa?pvs=21)

    - using globus for file management and sending mroe than one file. otherwise use rsync

- new VAEs from matei

- 2, 4, 8, 16 size latent space

- need to map it properly

- added a lot more to GUI

- to do

    - \*\*batch processing of cmaes\*\*

        - do this internal or myself?

        - very important

    - \*\*better logging of otpimization\*\*

    - \*\*more optim parameters\*\*

    - \*\*make optim depend on latent given (low priorityt stick with 4 for now)\*\*

    - \*\*create a modularity with the HPC scripts\*\*

        - add this to gui like parameters and all that

        - and maybe checkpointing

    - \*\*remove MPI traces for FENCISX now… modularize the parallelization of MUMPS\*\*

    - cannot have latent vector in args if optim is on

    - \*\*implement Globus CLI\*\*

- maybe do: memory profiler / fenicsx profiler

- not parallelizing with mpi because not worth it…

- FENICSX IS ON 0.9.0 on hpc. functions stuff changed since 0.7.0 (as on my mac). darn.

    - clone the environment and update it on mac. dont want to remake an environment for hpc.

    - conda create -n fenicsx-torch090 -c conda-forge fenics-dolfinx=0.9.0 mpich pyvista pytorch::pytorch torchvision torchaudio -c pytorch

    - pip install gmsh matplotlib cma

    - conda install scipy

    - conda install numpy=1.26.4

        - to get the right version working

12/11/24

- Using hpc:

    - conda create -n fenicsx-torch -c conda-forge fenics-dolfinx mpich pyvista pytorch::pytorch torchvision torchaudio -c pytorch

    - pip install gmsh matplotlib cma

    - need scipy…. conda or pip? too late for conda now? still try.

2/11/24

- need to make documentation  / a read me

- need to start adding stuff to the teams and do the project plan

1/11/24

- working on logging module

31/10/24

- make vf toggleable

- make the post processing more modular as per gui

- IMPLEMENT LOGGING

- effect of vf sweep?

- hpc

I want to implement a logging functionality for this code. I need the following:

- Configurations saved for easy replication (as a command in the command line)

- Generation data if optimizing

- pictures saved if toggled.

Ok now I need to expand the gui:

- I want to make volume fraction toggeable instead of -1 if I want it off

- I need to fix the post processing:

    - no more "all", just select in gui which plots I want

    - a toggle for saving screenshots vs interactive (remove ssh, interactive and say screenshot vs interactive choice)

- sections in the gui, so one section for optimization (bayes vs cmaes and in the future optimization parameters), one section for material topologies (eg, blank image, symmetry, sources), one section for solving (mesh resolution)

27/10/24

- actually going to just use my Mac and HPC instead of home PC.

- order of commands to install environment:

    - conda create -n fenicsx-torch -c conda-forge fenics-dolfinx mpich pyvista pytorch::pytorch torchvision torchaudio -c pytorch

    - pip install gmsh matplotlib

- set up GUI:

    - [x]  Sources add and remove at will

        - [ ]  Sources: make sure it's in ascending order

    - [ ]  Add option in GUI so that it prints the command for a terminal, so that it can be pasted into the HPC. It shouldn't run the code if this option is selected

    - [ ]  Add ssh button so that it activates the XFVB buffer and prints off screen (should toggle off screen)

        - [ ]  add a visualize interactively button. ( should be impossible to have both ssh and visualize interactive on)

        - [ ]  default is to have no ssh, and off screen printing (Screenshot in pyvista).

![image.png](FYP%2011348d1292e78063a22afd6de8fa4f60/image.png)

- [ ]  CMAES

26/10/24

- trying to implement vf control. did it by upscaling and gaussian blur.

- now refractoring the code. i want a gui

    - use Tkinter

23/10/24

\*\*Potential Ways to Streamline the Code\*\*

1. \*\*Adhere to PEP8 Standards\*\*:

• \*\*Consistency\*\*: Ensure consistent naming conventions, indentation, and spacing.

• \*\*Imports\*\*: Group imports logically (standard libraries, third-party libraries, local modules).

• \*\*Comments and Docstrings\*\*: Add docstrings to classes and functions for better documentation.

2. \*\*Refactor Repetitive Code\*\*:

• \*\*Utility Functions\*\*: Identify repetitive code blocks and encapsulate them into utility functions or classes.

• \*\*Configuration Handling\*\*: Use a configuration file (like YAML or JSON) to manage default parameters and avoid hardcoding values.

3. \*\*Simplify Command-Line Argument Parsing\*\*:

• \*\*Argument Groups\*\*: Organize related arguments into groups for clarity.

• \*\*Validation\*\*: Move argument validation logic into a separate function or method.

4. \*\*Optimize Data Structures\*\*:

• \*\*Numpy Arrays\*\*: Ensure all numerical computations use NumPy arrays for efficiency.

• \*\*Avoid Unnecessary Copies\*\*: Be cautious with data copying; use views when possible.

5. \*\*Remove Unused Code and Imports\*\*:

• \*\*Clean Up\*\*: Remove commented-out code and any imports or variables that are not used.

\*\*Improving Solving Speed for the Optimization\*\*

The primary bottleneck in your optimization process is the expensive evaluation of the objective function, which involves solving a PDE for each iteration. Here are strategies to speed this up:

1. \*\*Parallelizing Function Evaluations\*\*:

• \*\*Batch Evaluations\*\*: Modify the optimization algorithm to perform batch evaluations where possible.

• \*\*MPI Sub-Communicators\*\*: Split MPI.COMM\_WORLD into sub-communicators to run multiple PDE solves in parallel.

• \*\*Asynchronous Evaluations\*\*: Use asynchronous computation patterns to overlap communication and computation.

2. \*\*Use of Surrogate Models\*\*:

• \*\*Meta-Modeling\*\*: Build a surrogate model (e.g., Gaussian Processes, Neural Networks) to approximate the PDE solver.

• \*\*Active Learning\*\*: Use the surrogate model to predict expensive evaluations and update it iteratively.

3. \*\*Optimize the Solver Configuration\*\*:

• \*\*Solver Parameters\*\*: Fine-tune PETSc KSP solver parameters (e.g., preconditioners, convergence tolerances).

• \*\*Linear Algebra Libraries\*\*: Ensure you’re using optimized BLAS/LAPACK libraries (like Intel MKL).

4. \*\*Algorithmic Improvements\*\*:

• \*\*Gradient-Based Optimization\*\*: If possible, compute gradients and use more efficient optimization algorithms.

• \*\*Dimensionality Reduction\*\*: Reduce the dimensionality of the latent space if feasible.

5. \*\*Caching and Memoization\*\*:

• \*\*Result Caching\*\*: Store results of expensive computations to avoid redundant evaluations.

• \*\*Hashing Inputs\*\*: Use a hash of the inputs to check if a computation has been performed before.

6. \*\*Profile the Code\*\*:

• \*\*Profiling Tools\*\*: Use cProfile or line\_profiler to identify and optimize hotspots.

• \*\*Benchmarking\*\*: Measure performance before and after changes to ensure improvements are effective.

\*\*Parallelization Strategies\*\*

1. \*\*Fully Utilize MPI for Parallelism\*\*:

• \*\*Solver Parallelization\*\*: Ensure that the PDE solver fully exploits MPI across all available cores.

• \*\*Distributed Data Structures\*\*: Use distributed arrays and data structures where appropriate.

2. \*\*Parallel Optimization Algorithms\*\*:

• \*\*Parallel Bayesian Optimization\*\*: Use libraries like dragonfly or GPyOpt that support parallel evaluations.

• \*\*Alternative Algorithms\*\*: Consider optimization algorithms inherently suitable for parallelization (e.g., evolutionary algorithms).

3. \*\*Hybrid Parallelism\*\*:

• \*\*MPI + OpenMP\*\*: Combine MPI with multi-threading (OpenMP) within each node for shared-memory parallelism.

• \*\*GPU Acceleration\*\*: Offload computationally intensive tasks to GPUs using CUDA or OpenCL if hardware permits.

4. \*\*Asynchronous Execution\*\*:

• \*\*Non-blocking Communications\*\*: Use MPI’s non-blocking communication routines to overlap communication and computation.

• \*\*Task Scheduling\*\*: Implement a task scheduler to manage and distribute workloads dynamically.

5. \*\*Parallel Post-Processing\*\*:

• \*\*Parallel I/O\*\*: Use parallel file formats (like HDF5) to handle large data outputs efficiently.

• \*\*Distributed Visualization\*\*: Employ tools like ParaView with parallel capabilities for large-scale data.

\*\*Building a GUI for Configuration\*\*

1. \*\*Choose a GUI Framework\*\*:

• \*\*PyQt5/PySide2\*\*: Rich feature set and cross-platform compatibility.

• \*\*Tkinter\*\*: Simpler but sufficient for basic interfaces.

• \*\*Web-Based GUIs\*\*: Use Flask or Django for a web interface accessible via a browser.

2. \*\*Design the Interface\*\*:

• \*\*Parameter Inputs\*\*: Create input fields, sliders, and dropdowns for all configurable parameters.

• \*\*Visualization\*\*: Embed plots and visual feedback within the GUI.

• \*\*Progress Monitoring\*\*: Display real-time progress bars and logs.

3. \*\*Event Handling and Validation\*\*:

• \*\*Input Validation\*\*: Check user inputs and provide immediate feedback on invalid configurations.

• \*\*Responsive Design\*\*: Ensure the GUI remains responsive during long computations (use threading or multiprocessing).

4. \*\*Integration with the Backend\*\*:

• \*\*API Layer\*\*: Abstract the computational backend behind an API that the GUI interacts with.

• \*\*Asynchronous Calls\*\*: Implement asynchronous function calls to prevent the GUI from freezing.

5. \*\*Deployment Considerations\*\*:

• \*\*Standalone Application\*\*: Package the GUI and backend into a standalone application using tools like PyInstaller.

• \*\*Cross-Platform Support\*\*: Test on different operating systems to ensure compatibility.

\*\*Logging and Saving Configurations\*\*

1. \*\*Implement Logging\*\*:

• \*\*Python Logging Module\*\*: Use the built-in logging module to log events at various severity levels.

• \*\*Log Files\*\*: Write logs to files with timestamps, and organize them per run or per optimization iteration.

• \*\*Console Output\*\*: Configure loggers to output critical information to the console for real-time monitoring.

2. \*\*Save Configurations\*\*:

• \*\*Configuration Files\*\*: Use JSON, YAML, or INI files to save configurations in a human-readable format.

• \*\*Serialization\*\*: Serialize complex objects (like models) using pickle with caution (consider security implications).

3. \*\*Result Tracking\*\*:

• \*\*Database Storage\*\*: Store results and configurations in a lightweight database (like SQLite) for easy querying.

• \*\*Versioning\*\*: Implement a versioning system to keep track of different runs and their parameters.

4. \*\*Automatic Saving\*\*:

• \*\*Checkpointing\*\*: Periodically save the state of long-running computations to allow resuming in case of failures.

• \*\*Callbacks/Hooks\*\*: Use optimization callbacks to save intermediate results and configurations.

5. \*\*Metadata Inclusion\*\*:

• \*\*Comprehensive Records\*\*: Include all relevant metadata (timestamp, user, system info, seed values) in the saved configurations.

• \*\*Data Provenance\*\*: Ensure that results can be traced back to their originating parameters and code versions.

\*\*Additional Recommendations\*\*

1. \*\*Code Documentation and Comments\*\*:

• \*\*Docstrings\*\*: Provide clear docstrings for all modules, classes, and functions.

• \*\*Inline Comments\*\*: Explain non-obvious code segments to aid future maintenance.

2. \*\*Error Handling\*\*:

• \*\*Exceptions\*\*: Use try-except blocks to handle potential errors gracefully.

• \*\*User Feedback\*\*: Provide meaningful error messages to help diagnose issues.

3. \*\*Unit Testing\*\*:

• \*\*Test Cases\*\*: Write unit tests for critical functions to ensure code reliability.

• \*\*Continuous Integration\*\*: Set up CI/CD pipelines to automatically run tests on code changes.

4. \*\*Code Optimization\*\*:

• \*\*Efficient Algorithms\*\*: Review algorithms for computational efficiency (e.g., avoid nested loops when vectorization is possible).

• \*\*Memory Management\*\*: Monitor memory usage and optimize data structures to prevent leaks.

5. \*\*Community Resources\*\*:

• \*\*Leverage Libraries\*\*: Stay updated with the latest versions of libraries which might offer performance improvements.

• \*\*Open-Source Collaboration\*\*: Consider open-sourcing parts of your code to benefit from community feedback.

21/10/24

outlining objective, having built the framework:

\*\*Primary Objective:\*\*

• \*\*Investigate\*\* how different heat source configurations (positions and intensities) affect the development of thermal transport structures in materials, comparing predictions from the Guyer-Krumhansl (G-K) equations and Fourier’s law across varying Knudsen numbers, including the effects of bimaterial interfaces. The aim is to derive insights that can guide the design of efficient cooling systems for transistors and similar devices.

\*\*specific:\*\*

1. \*\*Model Comparison:\*\*

• \*\*Objective 1:\*\* Compare the thermal transport predictions of the G-K equations and Fourier’s law at different Knudsen numbers to understand the transition from Fourier conduction to non-Fourier (hydrodynamic) behavior at the mesoscale.

• \*\*Objective 2:\*\* Examine how these differences influence the formation of tree-like heat flow structures under various heat source configurations.

2. \*\*Heat Source Optimization:\*\*

• \*\*Objective 3:\*\* Optimize the placement and intensity of heat sources to minimize thermal resistance or maximize heat dissipation efficiency, using both the G-K equations and Fourier’s law.

• \*\*Objective 4:\*\* Identify how optimal configurations vary between the two models and across different Knudsen numbers.

3. \*\*Bimaterial Interface Effects:\*\*

• \*\*Objective 5:\*\* Investigate the impact of bimaterial interfaces on thermal transport and optimal heat source configurations, assessing differences predicted by the G-K equations and Fourier’s law.

• \*\*Objective 6:\*\* Provide design guidelines for incorporating bimaterial interfaces to enhance cooling performance in microelectronic devices.

20/10/24

- ran some optimizations

- realized i need to have it save the configuration when i want to see the results (sources and heats)

19/10/24.

got code to run. but image not showing correctly. need to compare what worked before and what works now for control scenario (’python3 src/main.py --source\_positions 0.5’)

not implementing extrusions yet

18/10/24

- Changing it to be modular with the sources.

- lots of changes.

- big questions re image processing

- need to make sure symmetry works again

14/10/24

- [ ]  mesh resolution sweep

- [ ]  bimaterial

- [ ]  multiple sources each with own tree

- [ ]  implement fourier to compare

- [ ]  alpha and interface

5/10/24

- [x]  redo parallel

    - [x]  heat flux should not be plotted when in parallel

- [ ]  how to do bimaterial?

    - [ ]  ell changes?

- [ ]  refractor mumps to depend on parallel

- [ ]  implement fourier to compare

- [x]  get vorticity plot somehow.

- [ ]  alpha: how the shape changes if we penalize the interface in ATP problem

- [ ]  transient??

[https://www.sciencedirect.com/science/article/pii/S2451904921001001#f0030](https://www.sciencedirect.com/science/article/pii/S2451904921001001#f0030) good application on transistors.

4/10/24

for kndusen sweep / temperatuer line,

- should non local length change by knudsden number?

- should normalization in temp line change by material ie non local length??

testing on blank topology

need to do same as temperature line for flux

made the temperature line for blank (made it easy to just activate blank or gamma =0 with —blank when you run), but have issues. mesh is fine already.

![image.png](FYP%2011348d1292e78063a22afd6de8fa4f60/image%201.png)

- fixed this by making it a finer mesh (length/20)

making flux profiles: use split

~~need to be able to make vorticity plots and profiles too~~

~~and effective thermal conductivity~~

reimplement paralellliztion

things to make more modular:

- Nano:

    - X Not blocking heat source. Use filter function.

    - X Move tree structure up to source directly

    - X optimize the latent space with bayes.py from phase change

    - Go to paper geometry / aspect ratio?

    - Maybe make the image stretch all the way to the mask extrusion

    - so in terms of to do list for publishing:

        - Knudsen sweep

        - Multiple extrusions. Distance between. Each its own tree from vae.

        - Size of source/extrusion

        - thermal interface (alpha function from other file?). low priority

        - Bayesian/VAE vs TO for all

        - change vae

        - Extrusion no extrusion?

        - Make sure temperature line follows…

- Phase change. Adaptive mesh in fenicsx. Jan 7 .

- Cold plate: put in fenics. Transient problem

    - Neural operator. do a sweep of all available ones

    - implement fenics adjoint in fenicsx

- Application for GA

Refractoring code:

- different files for each module

    - bcs

    - config

    - extrusion

    - logger

    - main

    - mesh generator

    - optimization

    - post processing

    - solver module

    - vae module

Organize:

- Github for code.

- Notion for notes. plus ipad

- Zotero for papers and notes on papers

-

3/10/24

how did i get here?

- parallelization, rank stuff

- into class format

- trying different solvers and methods. Taylor Hood Elements.

    - considered discontinuous galerkin methods

-

-

Upgrading from fenics to fenicsx:

Reasons:

- up to date code library

- faster (much faster)

- parallel processing

- gmsh software for complex meshes

To do it:

- lots of trial and error

- wouldn’t work at small scale

- wouldn’t converge

- Using mumps and configuring the parameters to adapt to small scale

Things i’ve done:

- make code extremely modular to build as a library/toolbox for optimization

```

Git commit logs:

\* 23576ab (HEAD -> main, origin/main, origin/HEAD) Feature: symmetry works. next: NSGA

\* cee4a12 new gui works. implement symmetry again. then implement nsga.

\* f3c8475 moved to ctkinter

\* 1c26cab fixing gui. still need to fix hpc and viz changing config when it shouldn't

| \* a04b369 (origin/joule, joule) NOT WORKING: Refactor GUI initialization and solver selection

|/

\* d392b1b (origin/refactor-log-and-hpc, refactor-log-and-hpc) Starting to make Joule functionality

\* 4c34286 Add fourier solving module and make gui changes

\* abd5e4b Add CMA plotting and add fourier solver.

\* 5abcaf6 Add visualize best from hpc config

\* abbc938 Add cma plotting

\* af0eba8 HPC enable toggle

\* efe9552 Add CMAES params

\* 3a079cd logging changes, fix gui, fix hpc, change cmaes. todo: implement cmaes params

\* 93511a5 trying to implement

\* 7607317 Small changes to main and sim config

\* 0989bc4 Refactor config class into new file

\* 1c9c355 (origin/batch-cmaes, batch-cmaes) refractor main

\* b1f9358 parallelize and refractor gpu. add hpc run script

\* dc54bbb attempting mpi for cmaes. fenics does not work

\* e70e2d0 (origin/fenicsx090, fenicsx090) Implemented 090

\* f7cd411 Refactor .gitignore to include logs directory

\* 2f4b8a5 Refactor hpc\_run.sh and fix stuff for hpc to run

\* 6bac3c6 Refactor CMA-ES module to split latent vectors per source

\* e5487f6 refractoring cma-es

\* 0838560 Refactor hpc\_run.sh and change CMA used

\* 8814e13 Add missing files and remove unused files

\* 17e1302 vae changes

\* 7c0ed12 logging and gui changes

\* 950fc7e implement cma-es

\* 3f896b7 remove cached

\* 1882f96 gui

\* 0e89b72 volume fraction + refractor

\* 2329541 modular sources

\* 5fc456c git ignore

\* 8efdefb feature: source in cmd line

\* 0885713 fixing image to gamma

\* d19e486 implementing two sources

\* 47feddc feat: reparallelize code

\* 27a1001 implement vorticity and eff cond post proc

\* 69c3258 implement knudsen and temp line plots

\* 6b618fb feat: new class organization

\* ac407d1 small changes

\* 0f631f3 implement sym and optim

\* 11fa0e6 optim and alpha tests

\* 0a87def clean up

\* 2765680 img works

\* 68a5786 sep into functions v4

\* 52658a5 implement gmsh

\* 1d1f042 implement parallelization with mumps (v3)

\* 9d5d8ec v2 works for correct scale

\* 20b68fd first commit