FEniCS for Solving PDE's and Generating Meshes

Morgan Buterbaugh, Casey Sharek, Sky Semone

May 2023

1 Introduction

The finite element method is a general technique for numerical solutions to differential equations. The idea is to start by finding an equation variational formulation of a problem. To solve a partial differential equation using the finite element method, you first set up a variational formulation and then discretize the problem and construct a finite-dimensional space. Then you solve that system and implement the solution on a computer.

The program we have chosen to work with is FEniCS. FEniCS is an open-source library that will implement the finite element method.

2 What is FEniCS?

FEniCS is a computing program developed by NumFOCUS that is used to help solve partial differential equations with finite element methods. The FEniCS project has been sponsored by NumFOCUS since 2016. NumFOCUS is a company whose main goal is to sponsor open-source projects for research, data, and scientific computing. The purpose of this program is to make solving partial differential equations simple; it allows users to type in their problems that have no analytical solution, which FEniCS translates into its corresponding computer code and uses finite element methods to solve. FEniCS uses Python and C++ coding

languages. Some of the features offered in the package are modeling, high-performance computing, and numerical computing.

FEniCS is comprised of multiple smaller libraries and functionalities: Basix, DOLFINx, FFCx, and UFL. Basix can be used in both C++ and Python. The Basix library creates finite elements on triangles, tetrahedra, quadrilaterals, hexahedra, pyramids, and prisms; it allows its users to evaluate functions using finite element basis functions, use operations to map data between reference and physical cells, and interpolate into and between finite element spaces. The FEniCS Form Compiler (FFC) is the compiler that generates C++ code for finite element variational forms by translating mathematical notations into C++ code. The Unified Form Language (UFL) is the language used for declaring finite element discretizations of variational forms. The FInite element Automatic Tabulator (FIAT) is a package for Python that automatically generates finite element basis functions. DOLFIN is a library that automatically computes the solution of PDEs using finite element methods.

Most of the libraries have two versions: UFL and UFLx, FFC and FFCx, and DOFLIN and DOLFINx. The two different versions of these libraries correspond to the version of FEniCS in use: Legacy FEniCS, which is the older software, and FEniCSx which is the up-to-date software.

3 Installing FEniCS

According to their site, FEniCS is available for Linux, OSX, and Windows systems, but this is simply not practical for most users, since many of the dependencies require Linux-based tools for installation. This section will cover the installation process and many of the problems that may arise while trying to configure and use the software for the first time.

3.1 FEniCS vs FEniCSx

In late 2022, the FEniCS team released an updated version of the package called FEniCSx to streamline development and absorb many of the external tools that have become essential to the core use of the program. Now that there is a new version of the program, FEniCS updates have slowed and it is no longer maintained for most platforms. Having a newer version with a widely used legacy version simultaneously can lead to many issues, as you will see in our installation notes.

FEniCSx is compromised of the core FEniCS library with UFL, Basix, FCCx, and DOLFINx, all rolled into one place. Legacy FEniCS is typically compiled alongside UFL, FIAT, FFC, and DOLFIN. Most of the resources online are for the Legacy version so our first attempt at installing was for that version.

3.2 FEniCS on Linux

The Windows operating system now allows for a Linux partition to be installed and used like a virtual machine with dynamic resource pooling via Windows Subsystems for Linux (WSL). Previous versions of WSL and WSL2, which have been shelved for the current version, allowing for Linux commands to be accessible through the Windows terminals and PowerShell, but the new version creates an entirely new operating system terminal that can be accessed like a Windows application.

A successful installation on a Linux partition via Windows is presented below. For computers that already have a Debian-based (Ubuntu) system, just ignore the first step.

1. **Install WSL**

   On Windows 10 and 11, WSL can be installed through an app in the Application Store, or through "wsl –install" in the PowerShell terminal. Note that you have to run PowerShell as an Administrator to have the proper access to install the new operating system. It will install Ubunutu by default, but other distributions can be installed,

including Debian, Kali, openSUSE, and SLES.

2. **Install FEniCS through PPA**

   There is a Ubuntu PPA (personal package archive) that captures FEniCS via DOLFINx, but leaves out many of the dependencies. To install this, use the following code:

```
sudo add-apt-repository ppa:fenics-packages/fenics
sudo apt update
sudo apt install fenicsx
```

3. **Installing for Python Use**

   For building an optional Python interface of DOLFIN and MSHR, pybind11 is needed since version 2018.1.0. To install it:

```
PYBIND11_VERSION=2.2.3
wget -nc --quiet https://github.com/pybind/pybind11/archive/v${
    PYBIND11_VERSION}.tar.gz
tar -xf v${PYBIND11_VERSION}.tar.gz && cd pybind11-${
    PYBIND11_VERSION}
mkdir build && cd build && sudo cmake -DPYBIND11_TEST=off .. &&
     sudo make install
```

   Note that unless specified, the user's home directory should be used for the installation location. Installing these as the root user might cause issues since the root "home" folder is a different location in Ubuntu (and most other distributions).

   Now, the Python package that handles the FEniCS binary can be installed through

```
pip3 install fenics-ffc --upgrade
```

4. **Installing Other FEniCS Components**

   This installation method leaves out two very powerful packages that even the simplest

scripts will need when using the program today. DOLFIN and MSHR can be installed with the correct Python links through

```
FENICS_VERSION=$(python3 -c"import ffc; print(ffc.__version__)")
git clone --branch=$FENICS_VERSION https://bitbucket.org/fenics-project/dolfin
git clone --branch=$FENICS_VERSION https://bitbucket.org/fenics-project/mshr
mkdir dolfin/build && cd dolfin/build && sudo cmake .. && sudo make install && cd ../..
mkdir mshr/build && cd mshr/build && sudo cmake .. && sudo make install && cd ../..
cd dolfin/python && pip3 install . && cd ../..
cd mshr/python && pip3 install . && cd ../..
```

At this point, searching through the packages installed for the base Python3 user will result in FEniCS/MSHR/DOLFIN being installed in at least one version (though depending on the Python and Ubuntu version, these might be any combination of 2021, 2019, and 2017 versions). Binary installation of these worked for the 2019 version only. This will be addressed along with other issues in the next step.

5. **Additional Advice if Things Don't Work**

   On my machine, this MSHR installation did not work, so the MSHR directory in "home" was deleted to remove that package. It was reinstalled from source via:

   ```
   git clone https://bitbucket.org/fenics-project/mshr
   ```

   Python3 is the default version for most things now, but sometimes, an error will occur saying that the command "python" does not exist in the PATH, so an alias must be made so that any references to "python" will go to "python3" The following line can be added to the file " /.bashrc"

```
alias python='python3'
```

Alternatively, the corresponding line of the cmake/make file that is causing the error can be edited to give the correct command for the version of Python installed on the computer.

```
whereis python3
whereis python
```

The above will return the location of the executable file that is run when the command is called.

3.3 Trying to Install FEniCS On Anaconda

1. **Install WSL**

   On Windows 10 and 11, WSL can be installed through the PowerShell terminal by using the command "wsl –install". Note that you have to run PowerShell as an Administrator to have the proper access to install the new operating system. This will install Ubunutu by default.

2. **Install Anaconda**

   There are many different ways to install Anaconda, but we will focus on the Anaconda installer for Linux. The first thing you will need to do is download the Anaconda installer for Linux in your browser. Once the application is installed, you will need to open the terminal. In the terminal, run the following codes:

   ```
   shasum -a 256 /PATH/FILENAME
   bash ~/Downloads/Anaconda3-2020.05-Linux-x86_64.sh
   ```

   and answer 'yes' to all of the prompts. This will successfully install Python 3.7 through Anaconda on Linux.

3. **Install Anaconda Python Packages**

   To use the prebuilt Anaconda Python Packages in your Linux terminal, you will need to run the following:

   ```
   conda create -n fenicsproject -c conda-forge fenics
   source activate fenicsproject
   ```

   After these have successfully been downloaded and installed, you are then ready to start working on examples using FEniCS on Anaconda.

It should be noted that the installation of FEniCS on Anaconda was a failure for us.

We were able to install WSL to obtain Ubuntu and install Anaconda for Windows. We tried to download the Anaconda installer for Linux but were unsuccessful. There were many errors that arose from this; some we were able to troubleshoot and others we were not able to. Hence, we were not able to install the prebuilt Anaconda Python Packages in the Ubuntu terminal.

3.4 Trying to Install FEniCS On Docker

1. **Install WSL**

   On Windows 10 and 11, WSL can be installed through the PowerShell terminal by using the command "wsl –install". Note that you have to run PowerShell as an Administrator to have the proper access to install the new operating system. This will install Ubuntu.

2. **Install Docker** On Windows 10 and 11, Docker can be installed through the app Docker Desktop in the Application Store, or through WLS. For this project, we will focus on the installation through WLS. To install Docker, open the Ubuntu terminal and use the following code:

   ```
   sudo apt install gnome-terminal
   sudo apt remove docker-desktop
   rm -r $HOME/.docker/desktop
   ```

```
sudo rm /usr/local/bin/com.docker.cli
sudo apt purge docker-desktop
```

Next, you will have to set up Docker's package repository by running the following code:

```
sudo apt-get update
sudo apt-get install \ ca-certificates \ curl \ gnupg \ lsb-release
sudo mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo \ "deb [arch=$(dpkg --print-architecture)
        signed-by=/etc/apt/keyrings/docker.gpg]
        https://download.docker.com/linux/ubuntu \
        $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

After you have done this, you need to download the latest DEB package, which you can obtain from https://docs.docker.com/desktop/install/ubuntu/. Then, you need to install the following packages in the terminal:

```
sudo apt-get update
sudo apt-get install ./docker-desktop-<version>-<arch>.deb
```

In order to then use Docker Desktop, you should open a new terminal and run:

```
systemctl --user start docker-desktop
```

Finally, you should run the following commands in the Ubuntu terminal:

```
curl -s https://get.fenicsproject.org | bash
fenicsproject run
docker run -ti -p 127.0.0.1:8000:8000 -v
    $(pwd):/home/fenics/shared -w /home/fenics/shared
```

```
quay.io/fenicsproject/stable:current
```

which uses prebuilt, high-performance Docker images and starts a container.

It should be noted that the installation of FEniCS on Docker was a failure.

We were able to install WSL to obtain Ubuntu and install Docker within Linux. We were having some trouble trying to get the Docker's package repository to run correctly. There were many errors that arose from this; some we were able to troubleshoot and others we were not able to. Thus, we were not able to get the FEniCS on Docker to work.

3.5 Updates for FEniCsx

During the additions needed for the time-dependent simulations, we ran into further installation issues. Since the only machine we could get FEniCs to run on was a shared machine that had to have the Python/Conda version updated for another project, we lost the working combination of dependencies required for the working version we had. For newer versions of Anaconda, none of the previously mentioned instructions will work, so we had to start new with a just downloading Dolfinx without any of the supporting programs. The binaries for Dolfinx had to be downloaded, along with all of the dependencies, which dated back to previous versions of various things circa 2019. This required use of a unique terminal environment, to not interfere with the alternative versions installed on the machine.

The MSHR package was critical to the previous study on FEniCs, and no reasonible combination of Dolfinx and MSHR could be installed with each other on this machine. This proved to be the main incentive to use our own meshes and external software to replace MSHR.

4 Using FEniCS with Exmaples

Since there are many versions of the program and its associated packages, many of the examples need tailoring to work on the more recent versions of the software. Going through the forums for FEniCS and related software reveals that the examples given in the code itself have not been updated since the mid-2010s and did not work out of the box.

4.1 General advice for getting examples to work

Some basic modifications should be noted first before trying to run the examples:

1. Instead of using **interactive()**, use

   ```
   import matplotlib.pyplot as plt
   plt.show()
   ```

   The import command only needs to occur once in the program and should be placed at the beginning of the script.

2. Instead of using **.array()**, use

   ```
   u.vector().get_local()
   ```

   These are the same function but are called different things for different versions. In some versions of Python/FEniCS, the only way around this is to replace **.array()** with

   ```
   np.array(X.vector())
   ```

   Troubleshooting these by printing the variable type makes fixing these relatively easy.

3. Any lines of the script used to generate or update a terminal progress bar should be deleted. According to several users, these have not worked since the early 2010s and on newer installations, do not have support at all.

4. Note what type of object is being plotted with the **plot()** command. In many instances online, both DOLFIN (or DOLFINx) and matplotlib are being imported, so the plot

function might be coming from either. Matplotlib can only plot things like vectors and arrays, while the DOLFIN package has tools for processing many of the FEniCS objects. However, in the downloaded version, the "show" command does not work, so you have to import matplotlib anyway for **plt.show()**.

4.2 Example One: Poisson problem

This code solves the Poisson equation with Dirichlet conditions. The test problem is chosen to give an exact solution at all nodes of the mesh.

$$-\Delta u = f \in \omega$$
$$u_D = 1 + x^2 + 2y^2$$
$$f = -6$$

To create the mesh for the Poisson problem, the command **UnitSquareMesh()** can be used. The arguments for this function are nx which specifies the number of horizontal cells, and ny which specifies the number of vertical cells. For the purpose of this example, we will choose $nx = 8$ and $ny = 8$. The function space also needs to be defined with the **FunctionSpace()** command; FunctionSpace takes the defined mesh, the element family (quadrature, Lagrange, real, Arnold-Winther, etc.), and the degree of the elements. For our problem, we chose a first-degree Lagrange family of elements (called with 'P' within the code).

There is also a function that defines Dirichlet boundary conditions called **DirichletBC()**, which takes in the function space, the solution values on the boundary, and the points that belong to the boundary. The solution values on the boundary can be defined with the **Expression()** function that takes the a string for the mathematical expression and the degree. Our expression here is $1 + x^2 + 2y^2$, which must be written with C++ syntax in the code.

With the mesh, function space, and boundary conditions defined, the next step is to define the variational problem. This can be done by using the TrialFunction() and TestFunction() commands; both commands take in the function space as its argument. The source term is also defined ($f = -6$), and the partial differential equation. The left hand side and right hand side must be defined separately. In the code, the line **a=dot(grad(u),grad(v))*dx** defines the LHS, and the line **L=f*v*dx** defines the RHS. Together, this represents the varitional formula $\int u'v'dx = \int fvdx$.

The last step is to solve and plot the PDE. The **solve()** will take care of that by taking in the PDE (**a==L**), the known solution, and the boundary conditions. Figure 1 is the grid generated using this method, and Figure 2 is the solution to the Poisson problem.

The code for this problem can be found in the Appendix.

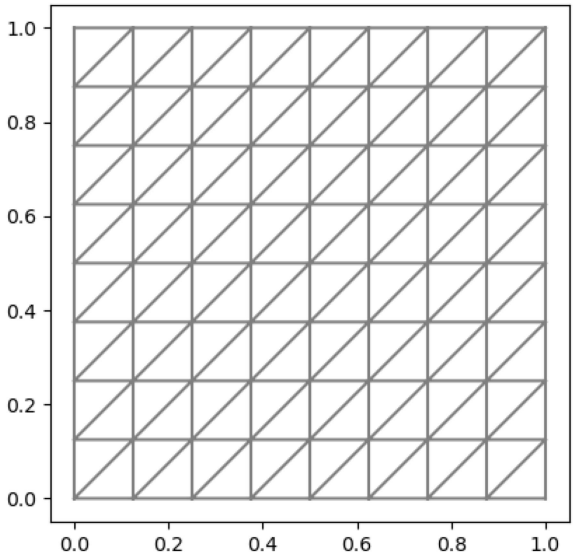


Figure 1: Simple grids can be made easily within FEniCs

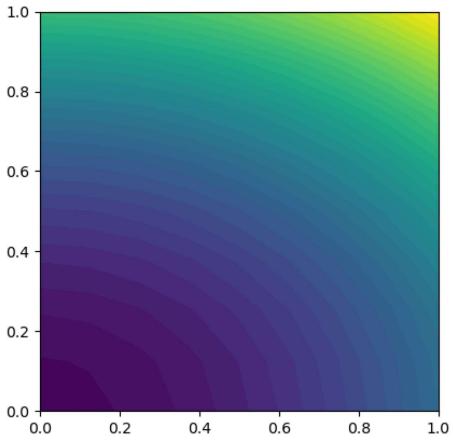


Figure 2: The solution of a Poisson problem with known solution

4.3 Example Two: Heat Dispersion in 2D

The next problem we will solve is as follows:

$$\frac{\partial u}{\partial t} = \Delta^2 u + f \in \Omega \times (0, T]$$
$$u_D = 1 + x^2 + \alpha y^2 + \beta t$$
$$f = \beta - 2 - 2\alpha$$

where the values of $\alpha = 3$ and $\beta = 1.2$ will be used.

Solving this PDE starts off very similarly to the Poisson example, where the mesh, function space, solution, and boundary conditions are defined the using the same steps and commands outlined above. The only difference is that the solution must be $1 + x^2 + \alpha y^2 + \beta t$ written in C++ syntax.

The heat equation introduces time to the system, so time-stepping becomes a necessary element to the code, which means the final time, number of time steps, and time steps size need to be defined. For our purpose, we chose a final time of two ($T = 2$), and 10 time steps. Time-stepping this equation just solves the PDE multiple times (the number of time steps defined).

An inital value is necessary since time is part of this PDE, so the initial value was coded using the **project()** function. The test function and trial function are also defined in the same manor as above, but the source term can be defined as $f = \beta - 2 - 2\alpha$. The left hand side and right hand side of the variational formula must be defined to be **a = u*v*dx + dt *dot (grad (u) , grad (v)) *dx** and **L = (u n + dt * f) *v*dx** to correspond with the variational form of $\int(uv + \Delta t u'v')dx = \int(u^n + \Delta t f^{n+1})vdx$. The time-stepping will just solve this PDE on a loop for each time step.

The code for solving this PDE can be found in the Appendix.

4.3.1 Finding the Error of a Simple System

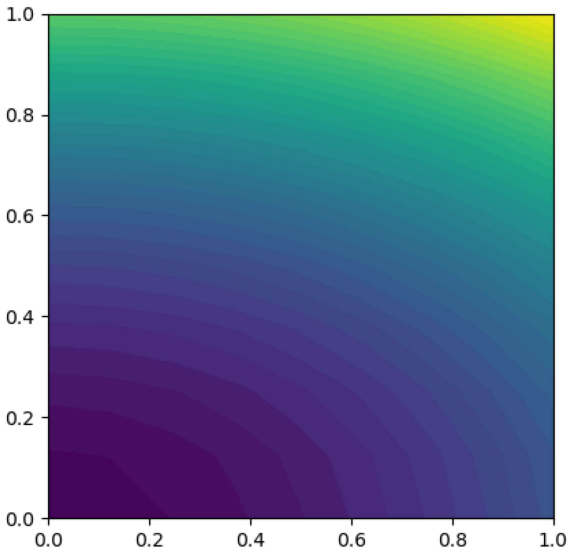


Figure 3: Heat equation applied to a known solution

```
skys@skys-laptop:~$ python3 ex1.py
Solving linear variational problem.
t = 0.20: error = 3.11e-15
Solving linear variational problem.
t = 0.40: error = 2.22e-15
Solving linear variational problem.
t = 0.60: error = 3.55e-15
Solving linear variational problem.
t = 0.80: error = 4e-15
Solving linear variational problem.
t = 1.00: error = 4e-15
Solving linear variational problem.
t = 1.20: error = 4.88e-15
Solving linear variational problem.
t = 1.40: error = 4.44e-15
Solving linear variational problem.
t = 1.60: error = 4.88e-15
Solving linear variational problem.
t = 1.80: error = 5.33e-15
Solving linear variational problem.
t = 2.00: error = 4.44e-15
```

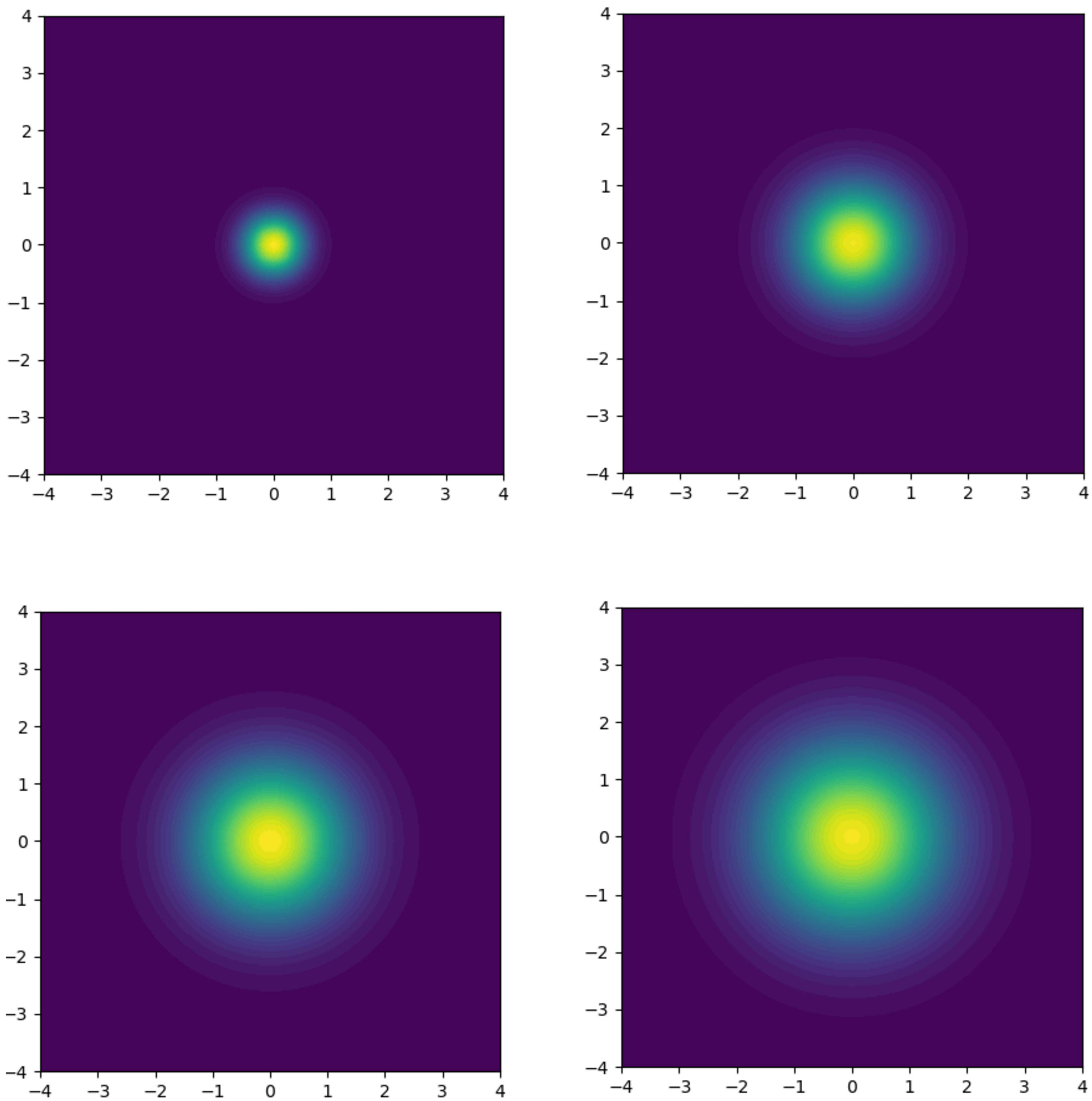
Figure 4: Absolute error as time propagates for heat equation

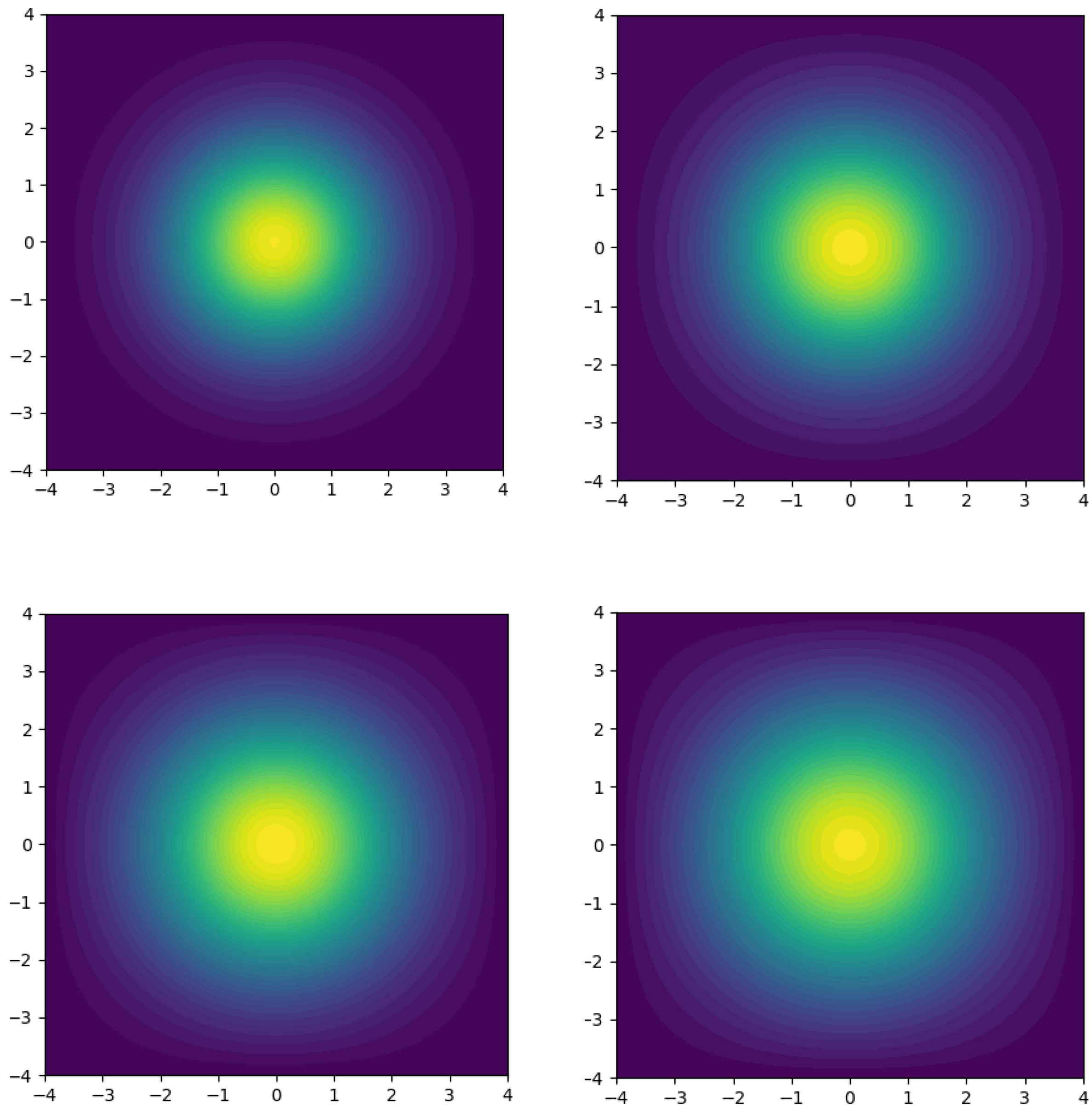
4.3.2 Solving a Gaussian Heat Blob

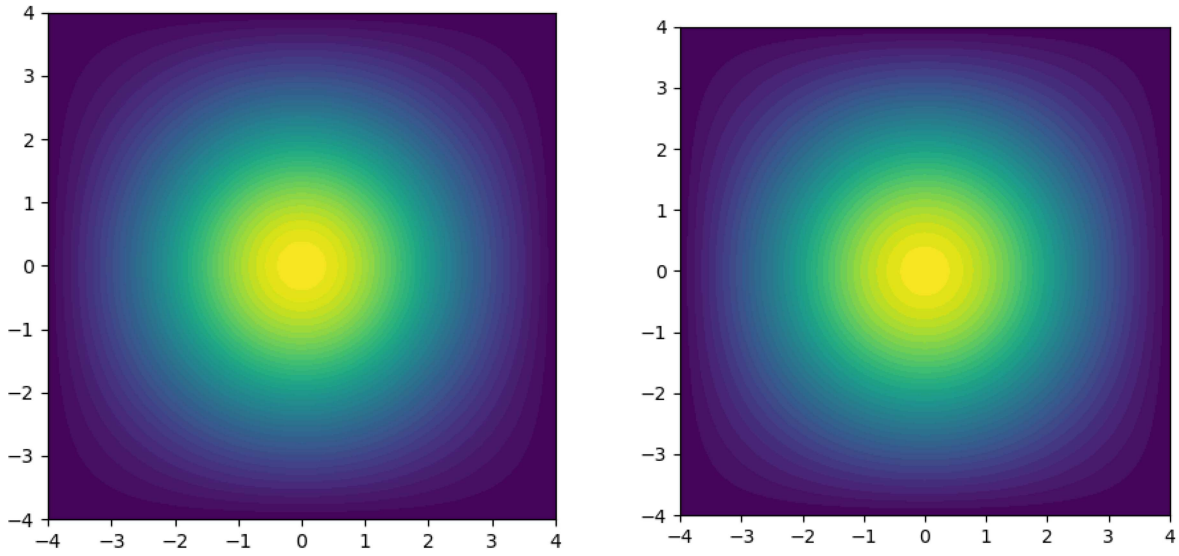
We will solve a Gaussian Heat Blob using the following conditions:

$$\frac{\partial u}{\partial t} = \Delta^2 u + f \in \Omega \times (0, T]$$
$$u_D = exp\left(-ax^2 - ay^2\right)$$
$$f = 0$$

The same steps must be taken in order to start this problem, but with a different mesh. Instead of a square mesh, a rectangle mesh will be used with the **RectangleMesh** function and a higher number of cells (30 for both horizontal and vertical). Time-stepping will be used in the same manor as above. The code for this example can be found below in the Appendix.


4
3
2
1
0
−1
−2
−3
−4
−4 −3 −2 −1 0 1 2 3 4





4.4 Example Three:

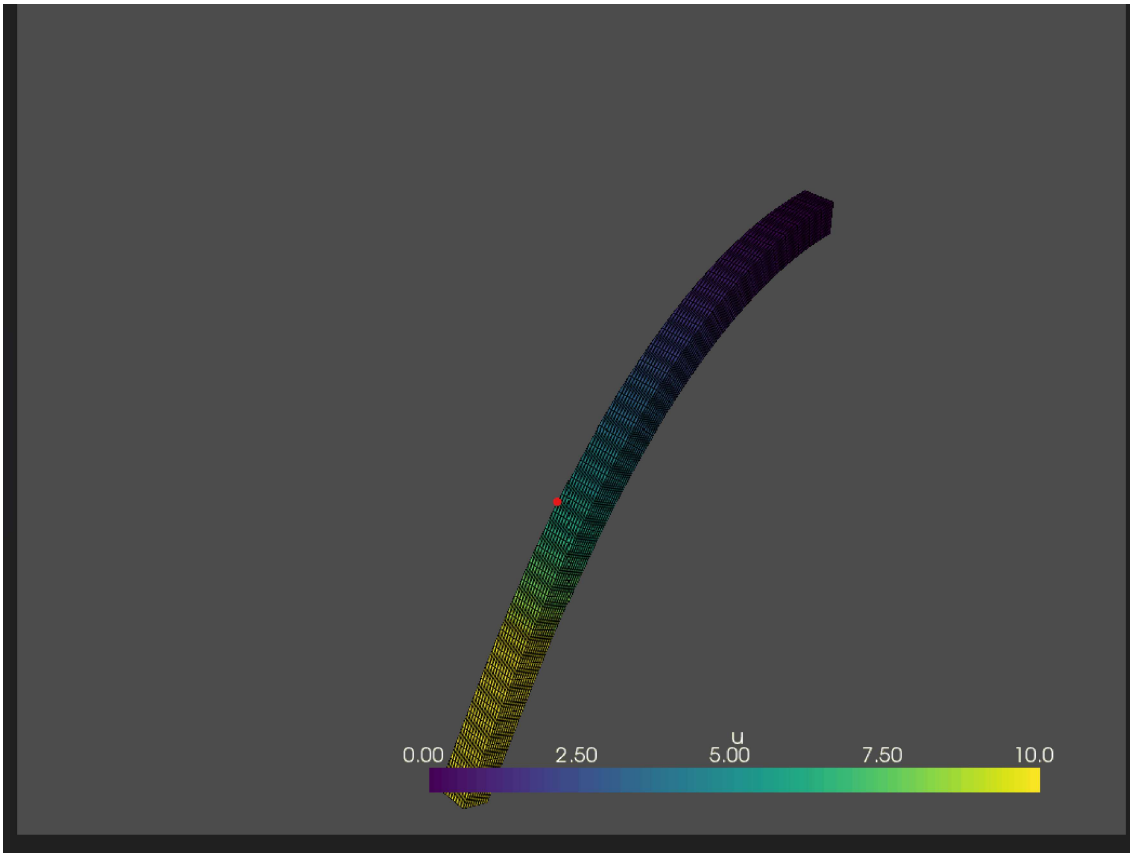


Figure 5: A 3D beam stress under external force.

While we had a solid understanding of the physics and code of the first two simulations, we wanted to see how the program responded to more complex examples, and if the code that generates it was easy to read and comprehend. We chose a reasonably complicated example using a 3D mesh and exporting a video. While we eventually got the code to run (after much editing,) it was exponentially harder to understand what the code was doing from line to line. Having worked with large finite-element method packages before, it's often the case that the language features some nice functions that can lead to elegant examples. However, as more boundary conditions and more complicated structures are required, the complexity of the program increases and its readability diminishes.

5 Fluid Flow Example

For this project, we wanted to code up an example that would determine the properties of a fluid flowing through a channel with a simple barrier in the center. This would reinforce many of the concepts learned through the FEniCS tutorials and texts, and also allow us to experiment with creating meshes more complex than simple ones used in the literature and in class.

5.1 Creating the Barrier Mesh

To do this, first a pencil sketch was created on graph paper, since without a sophisticated CAD program or vector graphics package with non-trivial post-processing, this is usually the simplest way for creating a geometrical mesh of small size. Next, GMSH was used to create the structures. With each square representing a relative distance (I used 0.01 units) the edge points are entered into the program. For curved surfaces, the center and radii were used instead of end nodes. Next, 1D lines and arcs are created from the individual points that form the boundary of the structure. This can be done either in the GUI, or by editing the .geo file: usually once the pattern for the mesh is discernable, it is faster to just edit

the text file instead of using the point and click method. There are many formats for saving meshes, so copy/pasting previous meshes might be an issue when the structure is imported later.

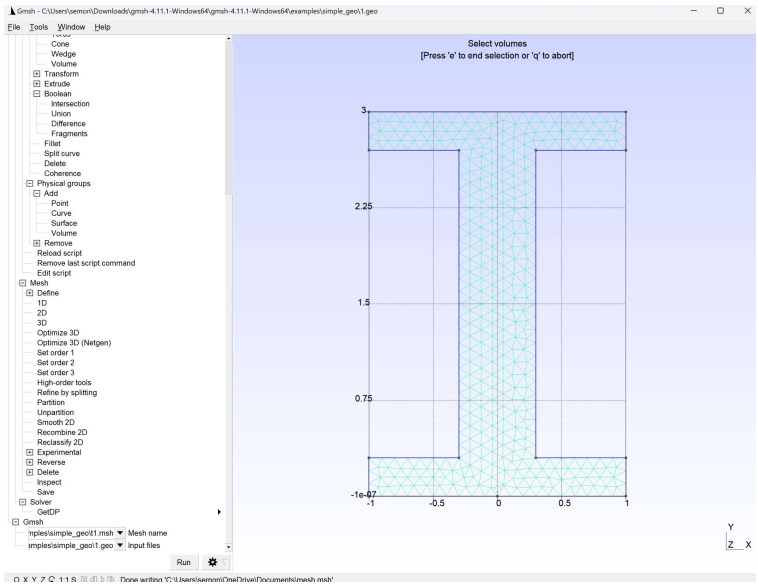


Figure 6: Using GMSH to make a mesh of the IUP

Next, a single surface is made from the combined boundaries (this is much easier in the GUI than manually). A "physical surface" must be created from the lone surface as we discovered later on when getting errors loading in the mesh in FEniCs. Once this is done and a 2D triangular mesh is created, it is saved as a ".msh" file that must be post-processed into an XML file.

We tried several methods of converting mesh formats to be used in FEniCs based on the methods given in the documentation for new and older version of the programs, but none of them work. Some people on the forums found a two step work around that involve using the program "meshio" that can be used within python. The following python script was used to make an intermediary xdmf file:

```
import meshio
msh = meshio.read("mesh.msh")
def create_mesh(mesh, cell_type, prune_z=False):
    cells = mesh.get_cells_type(cell_type)
    cell_data = mesh.get_cell_data("gmsh:physical", cell_type)
    out_mesh = meshio.Mesh(points=mesh.points, cells={
```

```
                                cell_type: cells}, cell_data={"name_to_read": [cell_data]})
    if prune_z:
        out_mesh.prune_z_0()
    return out_mesh
tetra_mesh = create_mesh(msh, "tetra")
triangle_mesh = create_mesh(msh, "triangle")
meshio.write("mesh.xdmf", tetra_mesh)
meshio.write("mf.xdmf", triangle_mesh)
```

Now, when we tried to use this mesh in the program, we got several errors, until we used the terminal commands to GMSH and Dolfin:

```
gmsh -2 mesh.geo -format msh2
dolfin-convert mesh.msh mesh.xml
```

Finally, we had about 7 copies of the original mesh, one of which could be read by the FEniCs.

5.2 Simulation Results and Interpretation

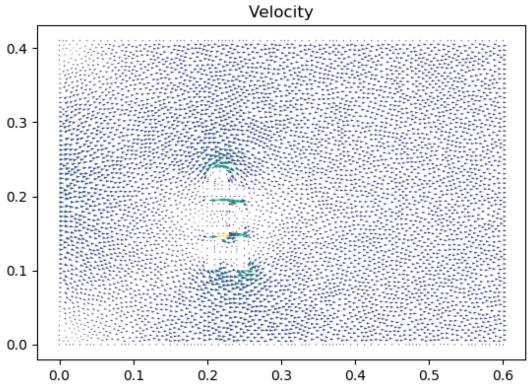


Figure 7: Initial velocity

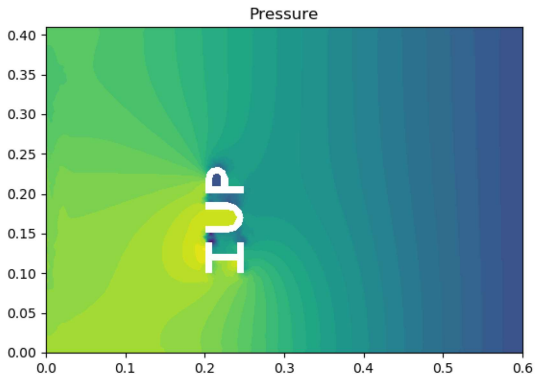


Figure 8: Initial pressure

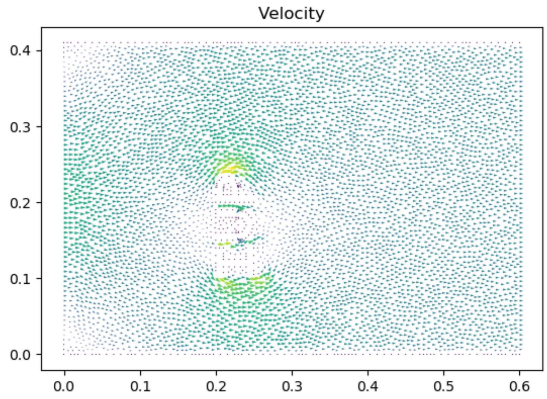


Figure 9: velocity after 0.5s

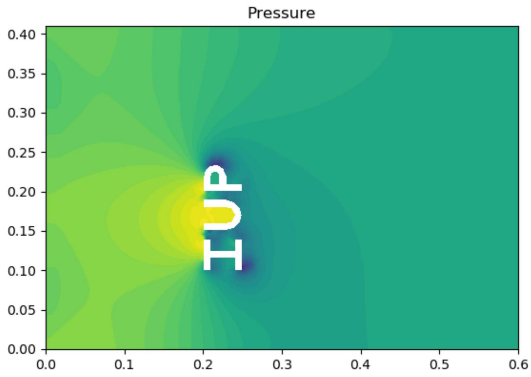


Figure 10: Pressure after 0.5s

We were able to create the mesh and the boundary conditions successfully. Even though the mesh for the channel was small, I thought it was impressive that we were able to mix the naive fenics mesh with the handmade GMSH one. While many details of the fenics physics engine lays behind smooth syntax, we also thought it was impressive how stable the simulation was, even with rough boundary conditions established. Usually, more effort is taken to ensure a smooth and fine-grid mesh on points with sharp boundaries. In the smallest portion of the channel, there was only one row of points, which most likely lead to the large velocity errors seen in figure 11. Given more time and a greater understanding of the FEniCSx problem translation, I think we could have worked these issues out and imported our solution.

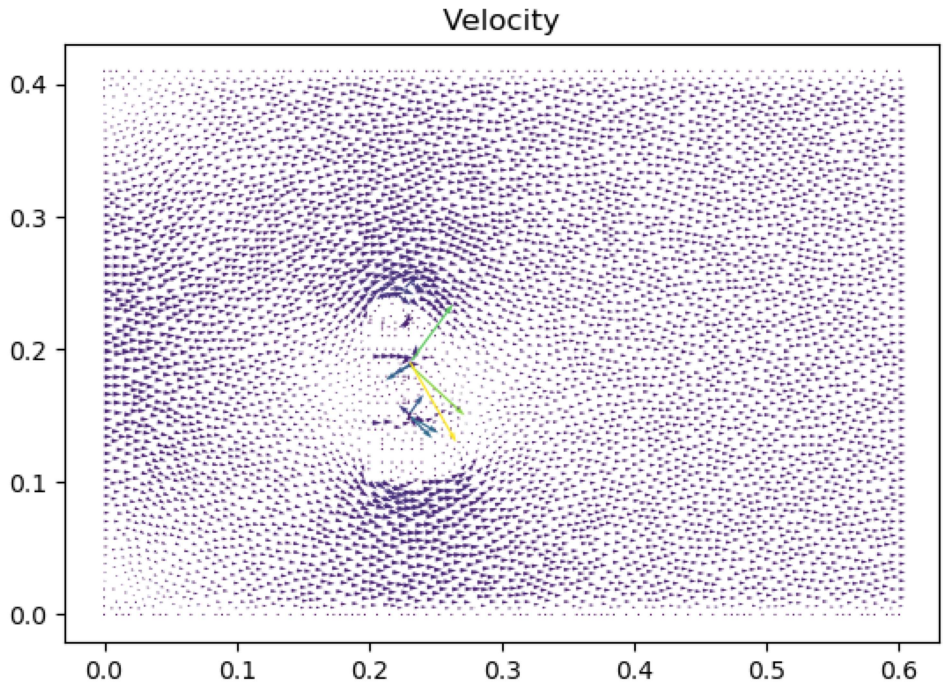


Figure 11: After 2000 time steps, the small geometry breaks the stability and large errors in the velocity arise.

6 Time-Dependant Examples

In the previous part of the study, we focused experimenting with FEM software and meshes. These examples will highlight our new work with FEM software in time-dependent simulations. In this section, we will try to focus on implementing what we learned in class and applying it to customize FEniCS software. This study will prove that we can apply what we've learned in class, and make a great case for the adaptability of the FEniCs software. It is often the case that open-source software has a high learning curve, especially for some of the very specific requirements of FEM software; however, we have determined that the FEniCs software is very robust and with the help of the manual and other examples online, it is relatively easy to implement a time-dependent example based on what we did in class.

In the following sections, we will explore and modify existing time dependent examples, and use these tools to make a different implementation of the heat solver that can be

compared to the one explored in the first part of this report.

6.1 Basics of Time in FEniCs

One unique feature of FEniCs is that time and spatial entities are treated entirely different. The base object for a differential equation is an "expression", which has spatial variables and parameters. Time is treated as a parameter in these equations and other methods must be created to solve the equations through time.

Many of the examples in the documentation were first order implicit or explicit Euler, which function well for most applications. The Crank-Nicholson method is of second order, and is derived from taking the first and second order differences as follows:

$$\frac{\partial u}{\partial t} - A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial u}{\partial x} = f$$

$$\frac{1}{2}\Big(\frac{u^{n+1}-u^n}{\Delta t}, w\Big) + \frac{A}{2}(u_x^n, w') + \frac{B}{2}(u_x^n, w) = \frac{1}{2}(f^n, w) \text{ explicit method}$$

$$\frac{1}{2}\Big(\frac{u^{n+1}-u^n}{\Delta t}, w\Big) + \frac{A}{2}(u_x^n, w') + \frac{B}{2}(u_x^{n+1}, w) = \frac{1}{2}(f^{n+1}, w) \text{ implicit method}$$

$$(u^{n+1}, w) - (u^n, w) + \frac{A\Delta t}{2}((u_x^{n+1}, w_x) + (u_x^n, w_x)) + \frac{B\Delta t}{2}((u_x^{n+1}, w) + (u_x^n, w)) = \Delta t(f^{n+\frac{1}{2}}, w)$$

$$(u^{n+1}, w) + \frac{A\Delta t}{2}(u_x^{n+1}, w_x) + \frac{B\Delta t}{2}(u_x^{n+1}, w) = (u^n, w) - \frac{A\Delta t}{2}(u_x^n, w_x) - \frac{B\Delta t}{2}(u_x^n, w) + \Delta t(f^{n+\frac{1}{2}}, w)$$

$$(\phi\phi + \frac{A\Delta t}{2}\phi'\phi' + \frac{B\Delta t}{2}\phi'\phi)u^{n+1} = (\phi\phi - \frac{A\Delta t}{2}\phi'\phi' - \frac{B\Delta t}{2}\phi'\phi)u^n + \Delta t(f^{n+\frac{1}{2}}, w)$$

$$(M + \frac{\Delta t}{2}A)u^{n+1} = (M - \frac{\Delta t}{2}A)u^n + \Delta t(f^{n+\frac{1}{2}}, w)$$

Because time-stepping methods must be created for each differential equation/boundary condition pair, there are several objects, methods, and tools in the FEniCs software that makes this relatively easy. Further, solving methods are adaptable to optimize performance of various time-stepping methods. In the software, applying the finite element method is usually done with "solve" function calls, where time-stepping can be done through "timestep" procedures. Further, projecting and interpolating are techniques that can be applied to real solutions so we have an accurate baseline when testing the solutions.

6.2 Stress-Strain in 3D Revisited

For the second example, we focused on performing time integration of an elastodynamic equation. We looked at a metallic bar that is clamped on one end and loaded by a vertical traction on the opposite end. This accounts for stress, strain, and dispersive forces throughout the bar.

We first formulated mass and damping forms of the elastodynamics equation, then we implemented the generalized α method and its influence on the solution, and finally performed computation of stresses using 'LocalSolver' in FEniCS.

The elastodynamics equation we took a look at is

$$\nabla \cdot \sigma + \rho b = \rho \ddot{u}$$

where u is the displacement vector field, $\ddot{u}$ is the acceleration, ρ is the material density, b is a given body of force, and σ is the stress tensor related to the displacement.

The weak form of this equation is found by using a test function $v \in V$ being a suitable function space that satisfies the displacement boundary conditions. So, we want to find

$$u \in V \text{ such that } m(\ddot{u}, v) + k(u, v) = L(v) \text{ for all } v \in V$$

where m is the symmetric bilinear form associated with the mass matrix and k is the stiffness matrix.

Once you have the finite element space interpolation, you then obtain the discretized evolution equation

$$m(\ddot{u}, v) + c(\dot{u}, v) + k(u, v) = L(v)$$

where c is a symmetric bilinear damping matrix. For this model, we used Rayleigh damping since there is very little known about the origin of damping in the structure.

For the time discretization using the generalized α method, we worked on the interval $[0, T]$ in $N + 1$ time steps. This method consists of solving the dynamic evolution equation at intermediate time between t_n and t_{n+1} to get the following

$$[M]\ddot{u}_{n+1}\alpha m + [C]\dot{u}_{n+1}\alpha f + [K]u_{n+1}\alpha f = F(t_{n+1}\alpha f).$$

We used the following equations for approximation for the displacement and velocity at t_{n+1}:

$$u_{n+1} = u_n + \delta t\dot{u}_n + \tfrac{\delta t^2}{2}((1-2\beta)\ddot{u}_n + 2\beta\ddot{u}_{n+1})$$
$$_{n+1} = \dot{u}_n + \delta t(1-\gamma)\ddot{u}_n + \gamma\ddot{u}_{n+1}).$$

If we set $\alpha_f = \alpha_m = 0$, then our problem can now be formulated in terms of unknown displacement at time t_{n+1}. For the parameters, we chose to set $\alpha_m, \alpha_f \leq \frac{1}{2}$, $\gamma = \frac{1}{2} + \alpha_m - \alpha_f$, and $\beta = \frac{1}{4}(\gamma + \frac{1}{2})^2$. Using FEniCS we can then plug in the known terms and solve the linear system for u_{n+1}.

The output result from FEniCS for the metallic bar that's clamped on one end and loaded by a vertical traction on the opposite end is given in Figure 12.

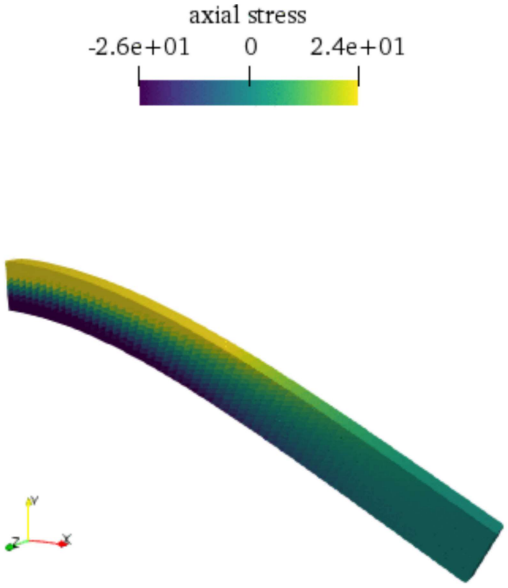


Figure 12

6.3 Cahn-Hillard Equations

We chose this because it uses a slightly different type of time-stepping, and its interesting to think about how this would be formulated for more than two types of fluids. In this example,

multiple concepts and usage cases are covered:

1. The built-in Newton solver
2. Advanced use of the base class NonlinearProblem
3. Automatic linearisation
4. A mixed finite element method
5. The Δ-method for time-dependent equations
6. User-defined Expressions as Python classes
7. Form compiler options
8. Interpolation of functions

It ended up being indispensable in creating our own code, as seen in the last example.

The Cahn-Hilliard equation is a parabolic equation and is typically used to model phase separation in binary mixtures. It involves first-order time derivatives, and second- and fourth-order spatial derivatives. The equation reads:

$$\begin{aligned} \frac{\partial c}{\partial t} - \nabla \cdot M \left(\nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \right) = 0 \quad & \text{in } \Omega, \\ M \left(\nabla \left(\frac{df}{dc} - \lambda \nabla^2 c \right) \right) = 0 \quad & \text{on } \partial\Omega, \\ M \lambda \nabla c \cdot n = 0 \quad & \text{on } \partial\Omega. \end{aligned}$$

where c is the unknown concentration at a point, f is a functional relationship in terms of c, and n is the outward direction boundary normal, M is a scalar parameter.

As a fourth order equation, it is best to express it as a coupled system of second order equations.

$$\frac{\partial c}{\partial t} - \nabla \cdot M \nabla \mu = 0 \quad \text{in } \Omega,$$
$$\mu - \frac{df}{dc} + \lambda \nabla^2 c = 0 \quad \text{in } \Omega.$$

Fields c and μ are now the unknowns. The weak variational form is written:

$$\int_\Omega \frac{\partial c}{\partial t} q \, \mathrm{d}x + \int_\Omega M \nabla \mu \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall \, q \in V,$$
$$\int_\Omega \mu v \, \mathrm{d}x - \int_\Omega \frac{df}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall \, v \in V.$$

Finally, applying the Δ-method to the weak form,

$$\int_\Omega \frac{c_{n+1} - c_n}{dt} q \, \mathrm{d}x + \int_\Omega M \nabla \mu_{n+\theta} \cdot \nabla q \, \mathrm{d}x = 0 \quad \forall \, q \in V$$
$$\int_\Omega \mu_{n+1} v \, \mathrm{d}x - \int_\Omega \frac{df_{n+1}}{dc} v \, \mathrm{d}x - \int_\Omega \lambda \nabla c_{n+1} \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall \, v \in V$$

There are many interesting things about this simulation, and one is how you can change the function in terms of c. We attempted to do this for other variations, but the results were confusing, and we switched back to an equation nearly identical to the original.

The results of the simulation are given here:

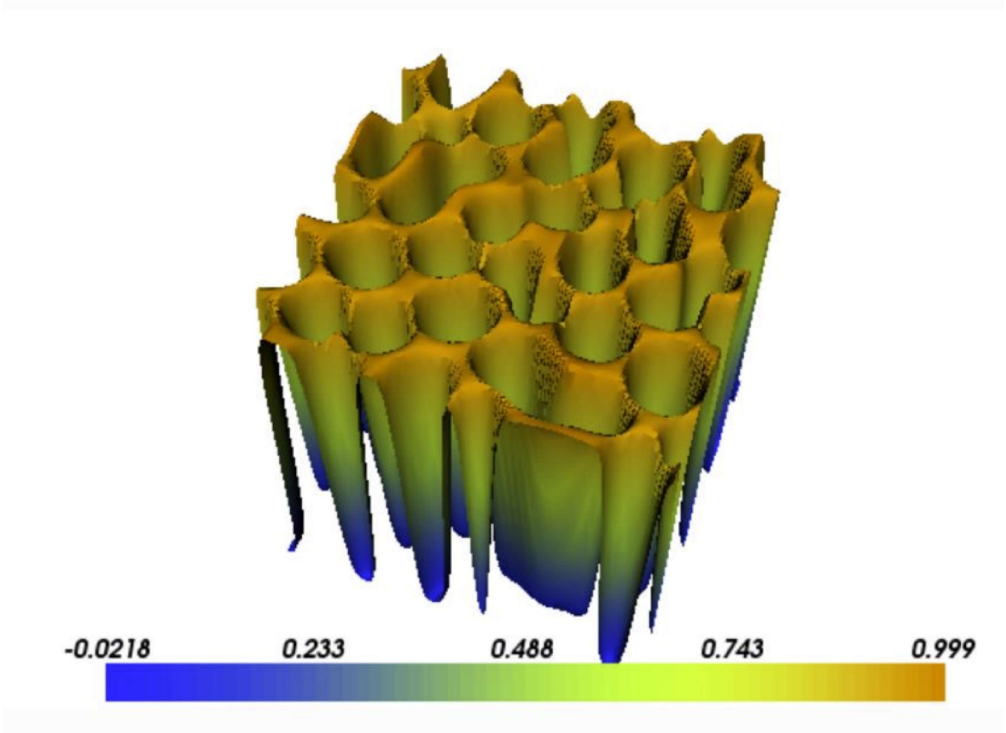


Figure 13: The default view of the output is in a 3D chart.

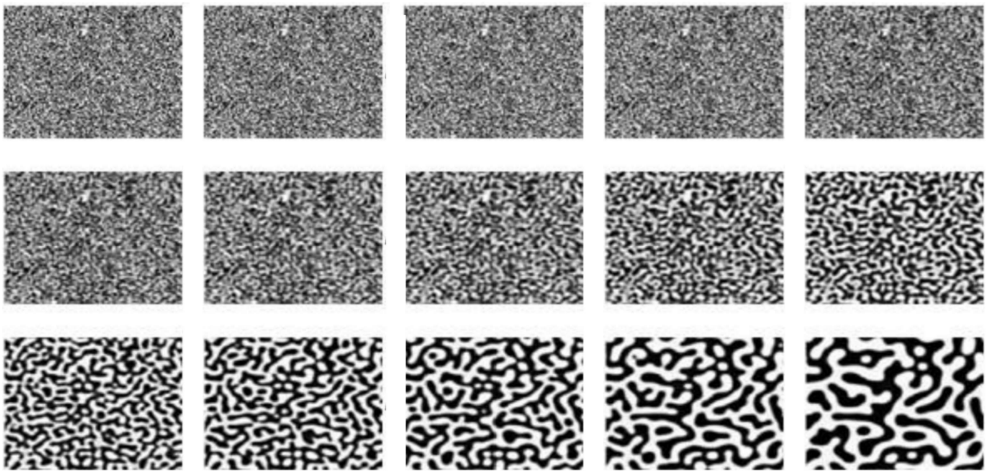


Figure 14: since the concentration boundaries are relatively sharp, a cutoff at value zero can separate the fluid into its two types.

6.4 Home-brewed Heat Equation

In our final example, we wanted to try to implement as much of our own code as possible. To ensure it worked properly, we needed something to compare it to, so we chose the simple heat

equation, since it is relatively simple over a square grid and the usual boundary conditions. It was also very similar to one of the first FEniCs examples we ran in the first part of the study.

While there was already a heat equation example in the FEniCs literature, we wanted to make our own mesh, and also make our own methods for handling the time dependent solution. Based on our last homework question, we thought that a Crank-Nicholson method would be interesting and a good exercise in familiarizing ourselves with the software.

6.4.1 Making a mesh

In class, we made a code that will generate the list of nodes and points needed to make a simple, regular grid filled with a number of triangles. We expanded this code so it could be used for any number of triangles in the x and y direction, and starting and ending at any x and y coordinate. For this to be used in serious FEM codes, it would need to be formatted like other popular meshes generated by GMSH, or similar programs. To accomplish this, I made a GMSH grid that was identical to the one I was to make in my own code, then I made a formatting script that was able to convert the node-vector and coordinate-vector into a full mesh file that included nodes, edges, planes, and surfaces. This will only work for flat 2D-examples, but it still a solid start for DIY mesh creation.

```python
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(10,6))

# how many triangles
nx  = 16
ny  = 6
# upper/lower bounds
x1 = 1.2
x2 = 9
y1 = 0.8
y2 = 7.9

# array of points used in x and y direction
xvals = np.linspace(x1,x2,nx+1)
yvals = np.linspace(y1,y2,ny+1)

def makeNodeTri(nx,ny):
    #this makes the nodes (in class assignment)
    def vrep(v,nx=nx,ny=ny):
        return np.array(ny*list(v))+(nx+1)*np.repeat(range(0,ny),nx*2)
    c1 = np.repeat(range(0,nx+1),2)[1:-1]
    c2 = np.repeat(range(0,nx),2)+1+np.array(nx*[0,nx+1])
    c3 = np.repeat(range(0,nx),2)+1+nx
    return np.transpose([vrep(c1),vrep(c2),vrep(c3)])

def node2Cor(nlist,nx,xvals,yvals):
    #convert to "coordinates"
    yc,xc = np.divmod(nlist,nx+1)
    return np.stack([xvals[xc],yvals[yc]],axis=-1)

def trianglePlot(clist):
    #added later to prove it works
    fig,ax = plt.subplots()
    ax.set_xlim([0,10]);ax.set_ylim([0,10])
    for cord in  clist:
        p=plt.Polygon(cord,color=np.random.rand(3,))
        ax.add_patch(p)
    plt.show()

################################################################################
nlist = makeNodeTri(nx,ny)                # size -> [2*nx*ny, 3]    node #'s
clist = node2Cor(nlist, nx,xvals,yvals)   # size -> [2*nx*ny, 3, 2] x,y vals
trianglePlot(clist)                       # this is just the plot
```

Figure 15: Code used to create meshes

The code for generating the nodes and coordinates is very short and efficient, while the formatting script is not. In the following figures, two mesh examples are given, varying in size and range.

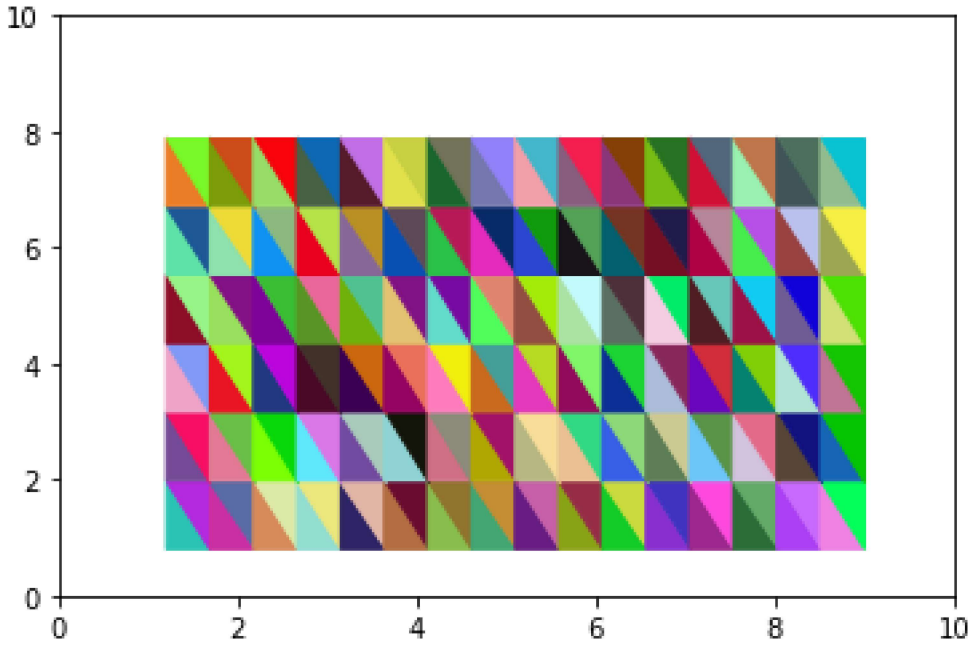


Figure 16: Example of a mesh

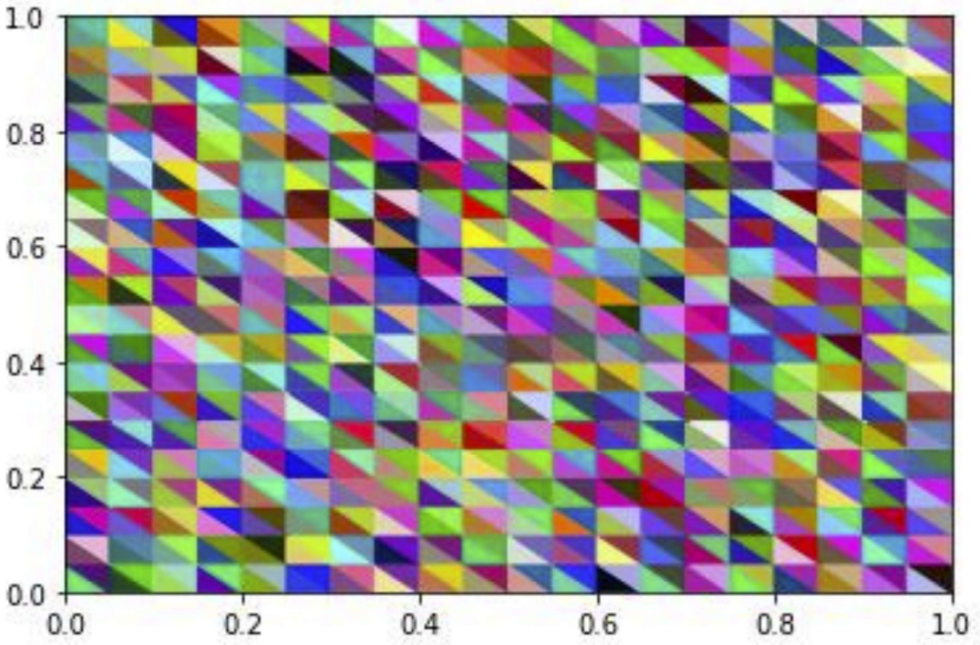


Figure 17: The specific mesh used in this study, scaled from 0 to 1, with 20 intervals in each x and y direction.

These can be compared to the GMSH/MSHR grid made in Figure 1.

6.4.2 Developing a Θ-Method

The Θ-method for solving first order time dependent differential equations relies on a forward difference. This method has parameter Θ set from values $(0, 1)$, where 0 is the explicit Euler method, 1 is implicit, and the Crank-Nicholson method is when $\Theta = \frac{1}{2}$ Implementing the Crank-Nicholson method is of interest because it is similar to a centered difference in time, resulting in a convergence rate of 2. With reference to the work explained in the first part of this section, this was the code that we developed for the Θ-method.

```
def create_timestep_solver(get_data, dsN, theta, u_old, u_new):
    # Initialize coefficients
    f_n, g_n = get_data(0)
    f_np1, g_np1 = get_data(0)
    idt = Constant(0)

    # Extract function space
    V = u_new.function_space()

    # Prepare weak formulation
    u, v = TrialFunction(V), TestFunction(V)
    theta = Constant(theta)
    F = ( idt*(u - u_old)*v*dx
        + inner(grad(theta*u + (1-theta)*u_old), grad(v))*dx
        - (theta*f_np1 + (1-theta)*f_n)*v*dx
        - (theta*g_np1 + (1-theta)*g_n)*v*dsN)
    a, L = lhs(F), rhs(F)

    def solve_(t, dt):
        # Update coefficients to current t, dt
        get_data(t, (f_n, g_n))
        get_data(t+dt, (f_np1, g_np1))
        idt.assign(1/dt)

        # Run the solver
        solve(a == L, u_new)

        # Pop log level
        set_log_level(old_level)
    return solve_
```

Figure 18: Code for the Θ-method time-stepping.

Now, applying this code to the heat equation example given earlier, we see the same exact results! Although this is not exciting on its own, it leads to many potential projects

that require more accurate time-stepping for a coarser grid.

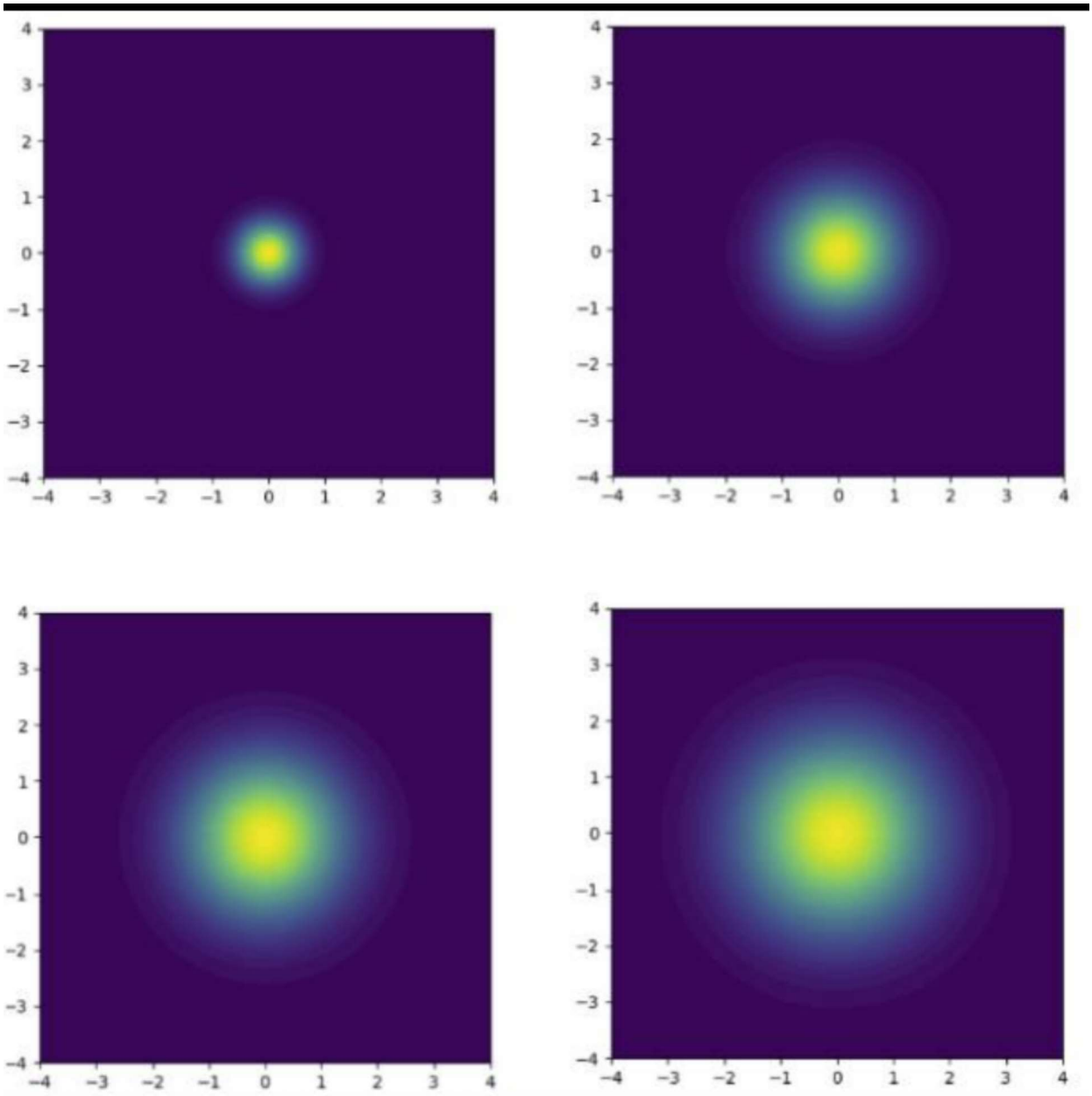


Figure 19: Heat starts in the center of grid and disperses in time.

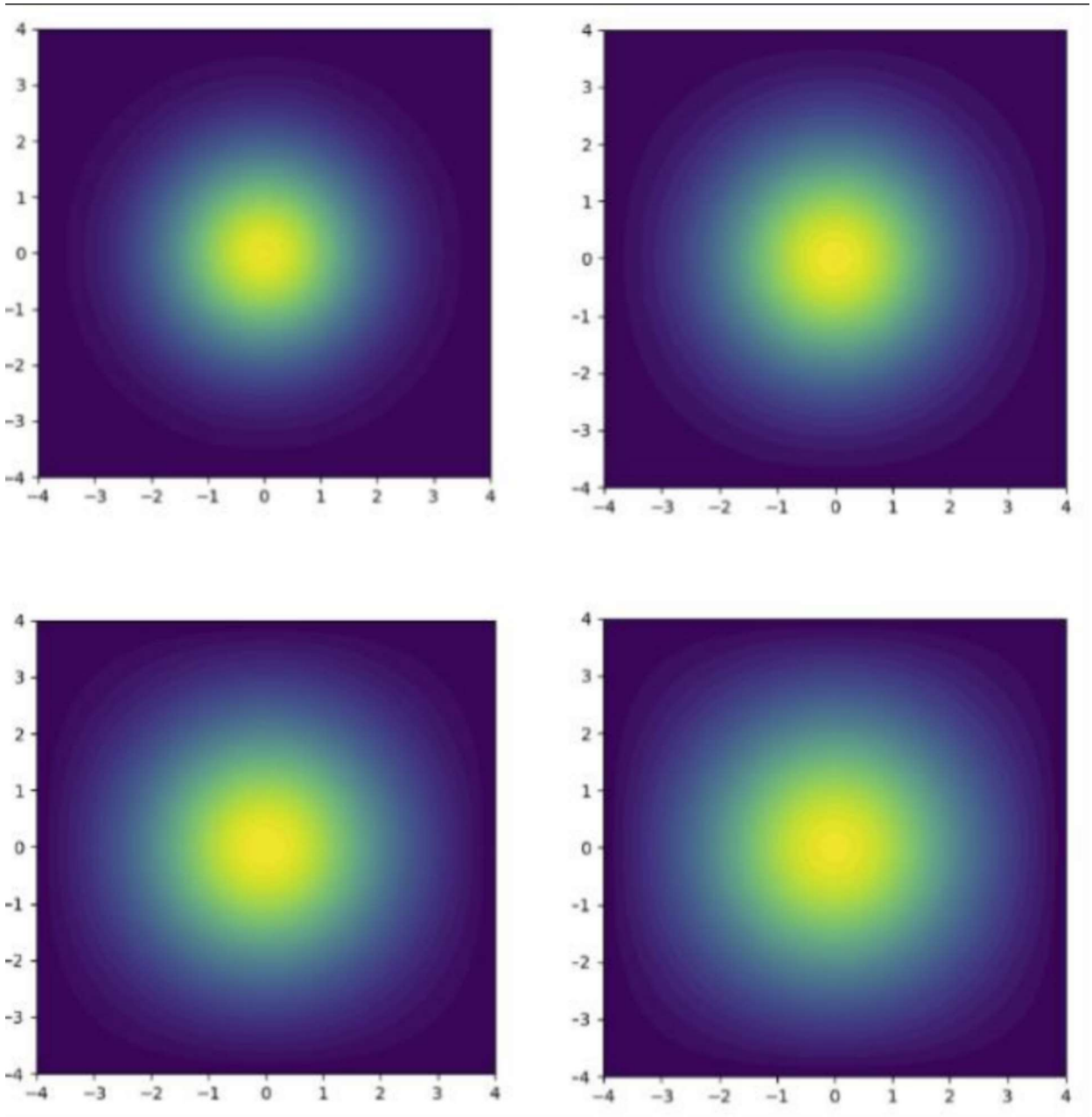


Figure 20: Continuation of heat dispersing.

Unfortunately, we ran into issues when trying to add a convergence rate piece to the initial FEniCs example, and the modified code. Part of this can be attributed to the complexity of the code, and the number of things "under-the-hood" in the solving process.

Another preventing issue was creating all the grids, which we would have to do in our cumbersome manual process. While we are satisfied with the overall results of this study,

much more in-depth research would be required to make sure that it functions the way we want it to.

7 Appendix

7.1 Code for Example One

```
from __future__ import print_function
from fenics import *
import dolfin
import matplotlib.pyplot as plt

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

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = dot(grad(u), grad(v))*dx
L = f*v*dx

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

# Plot solution and mesh
dolfin.plot(u)
plt.show()
dolfin.plot(mesh)
plt.show()
# Save solution to file in VTK format
vtkfile = File('poisson/solution.pvd')
vtkfile << u

# Compute error in L2 norm
error_L2 = errornorm(u_D, u, 'L2')

# Compute maximum error at vertices
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
```

```
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))

# Print errors
print('error_L2  =', error_L2)
print('error_max =', error_max)
```

7.2 Code for Example Two

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

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

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

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

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

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

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)

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

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

    # Update current time
```

```
    t += dt
    u_D.t = t

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

    # Plot solution
    dolfin.plot(u)
    plt.show()

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

    # Update previous solution
    u_n.assign(u)

% Part two of example two where a gaussian heat blob gets dispersed
%
%
from __future__ import print_function
from fenics import *
import matplotlib.pyplot as plt
import dolfin
import time

T = 2.0            # final time
num_steps = 100    # number of time steps
dt = T / num_steps # time step size

# Create mesh and define function space
nx = ny = 100
mesh = RectangleMesh(Point(-4, -4), Point(4, 4), nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0), boundary)

# Define initial value
u_0 = Expression('exp(-a*pow(x[0], 2) - a*pow(x[1], 2))',
                 degree=2, a=5)
u_n = interpolate(u_0, V)

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

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

# Create VTK file for saving solution
```

```
vtkfile = File('heat_gaussian/solution.pvd')

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

    # Update current time
    t += dt

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

    # Save to file and plot solution
    vtkfile << (u, t)
    if n%10==0:
        dolfin.plot(u)
        plt.show()

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

7.3 Code for Example Three

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

T = 1.0             # final time
num_steps = 1000    # number of time steps
dt = T / num_steps # time step size
mu = 0.001          # dynamic viscosity
rho = 1             # density

# Create mesh
channel = Rectangle(Point(0, 0), Point(0.6, 0.41))
mesh1 = Mesh("meshIUP1.xml")
domain = channel - mesh1
mesh = generate_mesh(domain, 64)

# Define function spaces
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)

# Define boundaries
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 0.6)'
walls    = 'near(x[1], 0) || near(x[1], 0.41)'
cylinder = 'on_boundary && x[0] > 0.1 && x[0] < 0.3 && x[1] > 0.1 && x[1] < 0.3'

# Define inflow profile
```

```
inflow_profile = ('4.0*1.5*x[1]*(0.41 - x[1]) / pow(0.41, 2)', '0')

# Define boundary conditions
bcu_inflow = DirichletBC(V, Expression(inflow_profile, degree=2), inflow)
bcu_walls = DirichletBC(V, Constant((0, 0)), walls)
bcu_cylinder = DirichletBC(V, Constant((0, 0)), cylinder)
bcp_outflow = DirichletBC(Q, Constant(0), outflow)
bcu = [bcu_inflow, bcu_walls, bcu_cylinder]
bcp = [bcp_outflow]

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)

# Define functions for solutions at previous and current time steps
u_n = Function(V)
u_  = Function(V)
p_n = Function(Q)
p_  = Function(Q)

# Define expressions used in variational forms
U  = 0.5*(u_n + u)
n  = FacetNormal(mesh)
f  = Constant((0, 0))
k  = Constant(dt)
mu = Constant(mu)
rho = Constant(rho)

# Define symmetric gradient
def epsilon(u):
    return sym(nabla_grad(u))

# Define stress tensor
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))

# Define variational problem for step 1
F1 = rho*dot((u - u_n) / k, v)*dx \
   + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Define variational problem for step 2
a2 = dot(nabla_grad(p), nabla_grad(q))*dx
L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx

# Define variational problem for step 3
a3 = dot(u, v)*dx
L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx

# Assemble matrices
```

```
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)

# Apply boundary conditions to matrices
[bc.apply(A1) for bc in bcu]
[bc.apply(A2) for bc in bcp]

# Create XDMF files for visualization output
xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')

# Create time series (for use in reaction_system.py)
timeseries_u = TimeSeries('navier_stokes_cylinder/velocity_series')
timeseries_p = TimeSeries('navier_stokes_cylinder/pressure_series')

# Save mesh to file (for use in reaction_system.py)
File('navier_stokes_cylinder/cylinder.xml.gz') << mesh

# Create progress bar
#progress = Progress('Time-stepping')
#set_log_level(PROGRESS)

# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1, 'bicgstab', 'hypre_amg')

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p_.vector(), b2, 'bicgstab', 'hypre_amg')

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u_.vector(), b3, 'cg', 'sor')

    # Plot solution
    dolfin.plot(u_, title='Velocity')
    plt.show()
    dolfin.plot(p_, title='Pressure')
    plt.show()

    # Save solution to file (XDMF/HDF5)
    xdmffile_u.write(u_, t)
    xdmffile_p.write(p_, t)

    # Save nodal values to file
    timeseries_u.store(u_.vector(), t)
```

```
    timeseries_p.store(p_.vector(), t)

    # Update previous solution
    u_n.assign(u_)
    p_n.assign(p_)

    # Update progress bar
   # progress.update(t / T)
    print('u_max:', u_.vector().get_local().max())
```

7.4 Code for IUP Fluid Flow Example

```
import numpy as np
import ufl

from petsc4py import PETSc
from mpi4py import MPI
from dolfinx import fem, mesh, plot


L = 20.0
domain = mesh.create_box(MPI.COMM_WORLD,[[0.0,0.0,0.0], [L, 1, 1]], [20, 5, 5], mesh.CellType.hexahedron)
V = fem.VectorFunctionSpace(domain, ("CG", 2))

def left(x):
    return np.isclose(x[0], 0)

def right(x):
    return np.isclose(x[0], L)

fdim = domain.topology.dim -1
left_facets = mesh.locate_entities_boundary(domain, fdim, left)
right_facets = mesh.locate_entities_boundary(domain, fdim, right)

# Concatenate and sort the arrays based on facet indices. Left facets marked with 1, right facets with two
marked_facets = np.hstack([left_facets, right_facets])
marked_values = np.hstack([np.full_like(left_facets, 1), np.full_like(right_facets, 2)])
sorted_facets = np.argsort(marked_facets)
facet_tag = mesh.meshtags(domain, fdim, marked_facets[sorted_facets], marked_values[sorted_facets])

u_bc = np.array((0,) * domain.geometry.dim, dtype=PETSc.ScalarType)

left_dofs = fem.locate_dofs_topological(V, facet_tag.dim, facet_tag.find(1))
bcs = [fem.dirichletbc(u_bc, left_dofs, V)]

B = fem.Constant(domain, PETSc.ScalarType((0, 0, 0)))
T = fem.Constant(domain, PETSc.ScalarType((0, 0, 0)))

v = ufl.TestFunction(V)
u = fem.Function(V)

# Spatial dimension
d = len(u)
```

```
# Identity tensor
I = ufl.variable(ufl.Identity(d))

# Deformation gradient
F = ufl.variable(I + ufl.grad(u))

# Right Cauchy-Green tensor
C = ufl.variable(F.T * F)

# Invariants of deformation tensors
Ic = ufl.variable(ufl.tr(C))
J  = ufl.variable(ufl.det(F))

# Elasticity parameters
E = PETSc.ScalarType(1.0e4)
nu = PETSc.ScalarType(0.3)
mu = fem.Constant(domain, E/(2*(1 + nu)))
lmbda = fem.Constant(domain, E*nu/((1 + nu)*(1 - 2*nu)))
# Stored strain energy density (compressible neo-Hookean model)
psi = (mu / 2) * (Ic - 3) - mu * ufl.ln(J) + (lmbda / 2) * (ufl.ln(J))**2
# Stress
# Hyper-elasticity
P = ufl.diff(psi, F)

# P = 2.0 * mu * ufl.sym(ufl.grad(u)) + lmbda * ufl.tr(ufl.sym(ufl.grad(u))) * I

metadata = {"quadrature_degree": 4}
ds = ufl.Measure('ds', domain=domain, subdomain_data=facet_tag, metadata=metadata)
dx = ufl.Measure("dx", domain=domain, metadata=metadata)
# Define form F (we want to find u such that F(u) = 0)
F = ufl.inner(ufl.grad(v), P)*dx - ufl.inner(v, B)*dx - ufl.inner(v, T)*ds(2)

problem = fem.petsc.NonlinearProblem(F, u, bcs)


from dolfinx import nls
solver = nls.petsc.NewtonSolver(domain.comm, problem)

# Set Newton solver options
solver.atol = 1e-8
solver.rtol = 1e-8
solver.convergence_criterion = "incremental"

import pyvista
import matplotlib.pyplot as plt
pyvista.start_xvfb()
plotter = pyvista.Plotter()
plotter.open_gif("deformation.gif", fps=3)

topology, cells, geometry = plot.create_vtk_mesh(u.function_space)
function_grid = pyvista.UnstructuredGrid(topology, cells, geometry)

values = np.zeros((geometry.shape[0], 3))
values[:, :len(u)] = u.x.array.reshape(geometry.shape[0], len(u))
function_grid["u"] = values
```

```
function_grid.set_active_vectors("u")

# Warp mesh by deformation
warped = function_grid.warp_by_vector("u", factor=1)
warped.set_active_vectors("u")

# Add mesh to plotter and visualize
actor = plotter.add_mesh(warped, show_edges=True, lighting=False, clim=[0, 10])

# Compute magnitude of displacement to visualize in GIF
Vs = fem.FunctionSpace(domain, ("Lagrange", 2))
magnitude = fem.Function(Vs)
us = fem.Expression(ufl.sqrt(sum([u[i]**2 for i in range(len(u))])), Vs.element.interpolation_points())
magnitude.interpolate(us)
warped["mag"] = magnitude.x.array

from dolfinx import log
log.set_log_level(log.LogLevel.INFO)
tval0 = -1.5
for n in range(1, 10):
    T.value[2] = n * tval0
    num_its, converged = solver.solve(u)
    assert(converged)
    u.x.scatter_forward()
    print(f"Time_step_{n},_Number_of_iterations_{num_its},_Load_{T.value}")
    function_grid["u"][:, :len(u)] = u.x.array.reshape(geometry.shape[0], len(u))
    magnitude.interpolate(us)
    warped.set_active_scalars("mag")
    warped_n = function_grid.warp_by_vector(factor=1)
    plotter.update_coordinates(warped_n.points.copy(), render=False)
    plotter.update_scalar_bar_range([0, 10])
    plotter.update_scalars(magnitude.x.array)
    plotter.write_frame()
plotter.close()
```

7.5 Code for Time-Dependent Example One

```
from dolfin import *
from ufl import Jacobian, diag
from plotting import plot

mesh = Mesh()
filename = "shell.xdmf"
f = XDMFFile(filename)
f.read(mesh)

def tangent(mesh):
    t = Jacobian(mesh)
    return as_vector([t[0,0], t[1, 0], t[2, 0]])/sqrt(inner(t,t))

t = tangent(mesh)
```

```
# We now compute the section local axis. As mentioned earlier, $a_1$ will be perpendicular to $t$ and the
    vertical direction $e_z=(0,0,1)$. After normalization, $a_2$ is built by taking the cross product
    between $t$ and $a_1$, $a_2$ will therefore belong to the plane made by $t$ and the vertical direction
    .

# In[2]:


ez = as_vector([0, 0, 1])
a1 = cross(t, ez)
a1 /= sqrt(dot(a1, a1))
a2 = cross(t, a1)
a2 /= sqrt(dot(a2, a2))


# We now define the material and geometrical constants which will be used in the constitutive relation. We
     consider the case of a rectangular cross-section of width $b$ and height $h$ in directions $a_1$ and
    $a_2$. The bending inertia will therefore be $I_1 = bh^3/12$ and $I_2=hb^3/12$. The torsional inertia
    is $J=\beta hb^3$ with $\beta\approx 0.26$ for $h=3b$. Finally, the shear areas are approximated by
    $S_1=S_2=\kappa S$ with $\kappa=5/6$.

# In[3]:


thick = Constant(0.3)
width = thick/3
E = Constant(70e3)
nu = Constant(0.3)
G = E/2/(1+nu)
rho = Constant(2.7e-3)
g = Constant(9.81)

S = thick*width
ES = E*S
EI1 = E*width*thick**3/12
EI2 = E*width**3*thick/12
GJ = G*0.26*thick*width**3
kappa = Constant(5./6.)
GS1 = kappa*G*S
GS2 = kappa*G*S


# We now consider a mixed $\mathbb{P}_1/\mathbb{P}_1$-Lagrange interpolation for the displacement and
    rotation fields. The variational form is built using a function 'generalized_strains' giving the
    vector of six generalized strains as well as a function 'generalized_stresses' which computes the dot
    product of the strains with the above-mentioned constitutive matrix (diagonal here). Note that since
    the 1D beams are embedded in an ambient 3D space, the gradient operator has shape (3,), we therefore
    define a tangential gradient operator 'tgrad' by taking the dot product with the local tangent vector
    $t$.
#
# Finally, similarly to Reissner-Mindlin plates, shear-locking issues might arise in the thin beam limit.
    To avoid this, reduced integration is performed on the shear part $Q_1\widehat{\gamma}_1+Q_2\widehat{\
    gamma}_2$ of the variational form using a one-point rule.

# In[6]:
```

```
Ue = VectorElement("CG", mesh.ufl_cell(), 1, dim=3)
W = FunctionSpace(mesh, Ue*Ue)

u_ = TestFunction(W)
du = TrialFunction(W)
(w_, theta_) = split(u_)
(dw, dtheta) = split(du)

def tgrad(u):
    return dot(grad(u), t)
def generalized_strains(u):
    (w, theta) = split(u)
    return as_vector([dot(tgrad(w), t),
                      dot(tgrad(w), a1)-dot(theta, a2),
                      dot(tgrad(w), a2)+dot(theta, a1),
                      dot(tgrad(theta), t),
                      dot(tgrad(theta), a1),
                      dot(tgrad(theta), a2)])
def generalized_stresses(u):
    return dot(diag(as_vector([ES, GS1, GS2, GJ, EI1, EI2])), generalized_strains(u))

Sig = generalized_stresses(du)
Eps =  generalized_strains(u_)

dx_shear = dx(scheme="default",metadata={"quadrature_scheme":"default", "quadrature_degree": 1})
k_form = sum([Sig[i]*Eps[i]*dx for i in [0, 3, 4, 5]]) + (Sig[1]*Eps[1]+Sig[2]*Eps[2])*dx_shear
l_form = Constant(-rho*S*g)*w_[2]*dx


# Clamped boundary conditions are considered at the bottom $z=0$ level and the linear problem is finally
    solved.

# In[7]:


def bottom(x, on_boundary):
    return near(x[2], 0.)
bc = DirichletBC(W, Constant((0, 0, 0, 0, 0, 0)), bottom)

u = Function(W)
solve(k_form == l_form, u, bc)


# Matplotlib functions cannot plot functions defined on a 1D mesh embedded in a 3D space so that we output
    the solution fields to XDMF format for vizualisation with Paraview for instance. We also export the
    bending moments by projecting them on a suitable function space.

# In[8]:


ffile = XDMFFile("shell-results.xdmf")
ffile.parameters["functions_share_mesh"] = True
v = u.sub(0, True)
v.rename("Displacement", "")
ffile.write(v, 0.)
```

```
theta = u.sub(1, True)
theta.rename("Rotation", "")
ffile.write(theta, 0.)

V1 = VectorFunctionSpace(mesh, "CG", 1, dim=2)
M = Function(V1, name="Bending_moments_(M1,M2)")
Sig = generalized_stresses(u)
M.assign(project(as_vector([Sig[4], Sig[5]]), V1))
ffile.write(M, 0)

ffile.close()
```

7.6 Code for Time-Dependent Example Two

```
import random
from dolfin import *

# Class representing the intial conditions
class InitialConditions(Expression):
    def __init__(self):
        random.seed(2 + MPI.rank(mpi_comm_world()))
    def eval(self, values, x):
        values[0] = 0.63 + 0.02*(0.5 - random.random())
        values[1] = 0.0
    def value_shape(self):
        return (2,)

# Class for interfacing with the Newton solver
class CahnHilliardEquation(NonlinearProblem):
    def __init__(self, a, L):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
    def F(self, b, x):
        assemble(self.L, tensor=b)
    def J(self, A, x):
        assemble(self.a, tensor=A)

# Model parameters
lmbda  = 1.0e-02  # surface parameter
dt     = 5.0e-06  # time step
theta  = 0.5      # time stepping family, e.g. theta=1 -> backward Euler, theta=0.5 -> Crank-Nicolson

# Form compiler options
parameters["form_compiler"]["optimize"]     = True
parameters["form_compiler"]["cpp_optimize"] = True
parameters["form_compiler"]["representation"] = "quadrature"

# Create mesh and define function spaces
mesh = UnitSquareMesh(96, 96)
V = FunctionSpace(mesh, "Lagrange", 1)
ME = V*V
```

```
# Define trial and test functions
du     = TrialFunction(ME)
q, v   = TestFunctions(ME)

# Define functions
u   = Function(ME)  # current solution
u0  = Function(ME)  # solution from previous converged step

# Split mixed functions
dc, dmu = split(du)
c,  mu  = split(u)
c0, mu0 = split(u0)

# Create intial conditions and interpolate
u_init = InitialConditions()
u.interpolate(u_init)
u0.interpolate(u_init)

# Compute the chemical potential df/dc
c = variable(c)
f    = 100*c**2*(1-c)**2
dfdc = diff(f, c)

# mu_(n+theta)
mu_mid = (1.0-theta)*mu0 + theta*mu

# Weak statement of the equations
L0 = c*q*dx - c0*q*dx + dt*dot(grad(mu_mid), grad(q))*dx
L1 = mu*v*dx - dfdc*v*dx - lmbda*dot(grad(c), grad(v))*dx
L = L0 + L1

# Compute directional derivative about u in the direction of du (Jacobian)
a = derivative(L, u, du)

# Create nonlinear problem and Newton solver
problem = CahnHilliardEquation(a, L)
solver = NewtonSolver()
solver.parameters["linear_solver"] = "lu"
solver.parameters["convergence_criterion"] = "incremental"
solver.parameters["relative_tolerance"] = 1e-6

# Output file
file = File("output.pvd", "compressed")

# Step in time
t = 0.0
T = 50*dt
while (t < T):
    t += dt
    u0.vector()[:] = u.vector()
    solver.solve(problem, u.vector())
    file << (u.split()[0], t)

plot(u.split()[0])
interactive()
```

7.7 Code for Time-Dependent Example Three

```
from fenics import *
import numpy as np
T = 2.0 # final time
num_steps = 10 # number of time steps
dt = T / num_steps # time step size
alpha = 3 # parameter alpha
beta = 1.2 # parameter beta
# Create mesh and define function space
nx = ny = 8
mesh = importMesh('skyTest1.txt')
V = FunctionSpace(mesh, 'P', 1)
# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
degree=2, alpha=alpha, beta=beta, t=0)
def boundary(x, on_boundary):
    return on_boundary
bc = DirichletBC(V, u_D, boundary)
# Define initial value
u_n = interpolate(u_D, V)
#u_n = project(u_D, V)
# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)
# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
    t += dt
    u_D.t = t
    # Compute solution
    solve(a == L, u, bc)
    # Plot solution
    plot(u)
    # Compute error at vertices
    u_e = interpolate(u_D, V)
    error = np.abs(u_e.vector().array() - u.vector().array()).max()
    print('t = %.2f: error = %.3g' % (t, error))
    # Update previous solution
    u_n.assign(u)


def create_timestep_solver(get_data, dsN, theta, u_old, u_new):
    f_n, g_n = get_data(0)
    f_np1, g_np1 = get_data(0)
    idt = Constant(0)

    # Extract function space
    V = u_new.function_space()

    # Prepare weak formulation
```

```
    u, v = TrialFunction(V), TestFunction(V)
    theta = Constant(theta)
    F = ( idt*(u - u_old)*v*dx
        + inner(grad(theta*u + (1-theta)*u_old), grad(v))*dx
        - (theta*f_np1 + (1-theta)*f_n)*v*dx
        - (theta*g_np1 + (1-theta)*g_n)*v*dsN
    )
    a, L = lhs(F), rhs(F)

    def solve_(t, dt):
        """Update problem data to interval (t,t+dt) and
________run the solver"""

        # Update coefficients to current t, dt
        get_data(t, (f_n, g_n))
        get_data(t+dt, (f_np1, g_np1))
        idt.assign(1/dt)

        # Push log level
        old_level = get_log_level()
        warning = LogLevel.WARNING if cpp.__version__ > '2017.2.0' else WARNING
        set_log_level(warning)

        # Run the solver
        solve(a == L, u_new)

        # Pop log level
        set_log_level(old_level)

    return solve_


def timestepping(V, dsN, theta, T, dt, u_0, get_data):
    u = Function(V)
    solver = create_timestep_solver(get_data, dsN, theta, u, u)
    u.interpolate(u_0)
    t = 0
    while t < T:
        energy = assemble(u*dx)
        solver(t, dt)
        t += dt
```