# Finite Difference PDE Solver with Adaptable FD Scheme Implementation

Ari Gillman, Chemical and Biological Engineering,
James Roggeveen, Mechanical and Aerospace Engineering,
Kendra Bergstedt, Plasma Physics,
Ximing Li, Chemistry,
Princeton University

January 2020

## 1 Introduction

In this project we seek to design and develop software to solve PDEs using Finite Difference methods. In particular, we are interested in PDEs which may be solved by the method of lines. Our goal is to produce code that allows for particular finite difference schemes to be chosen by the user at runtime and to use these different discretizations to solve the PDE with no user manipulation of the governing equations. We produce code which allows for runtime implementation of a particular finite difference scheme to solve a given PDE but which still requires proper interface construction for the user to access these methods.

## 2 Background

A partial differential equation (PDE) is a relationship between an unknown function $u(x_1, x_2, \ldots, x_n)$ and its derivatives with respect to the variables $x_1, x_2, \ldots, x_n$. PDEs occur naturally in many scientific applications; they model the rate of change of a physical quantity with respect to both space variables and time variables. Some well-known examples of PDEs include Laplace's equation (an elliptic linear second-order PDE), wave function (a hyperbolic linear second-order PDE), and heat equation (a parabolic linear second-order PDE). In most cases the solution of a PDE is not unique, with a boundary of the region where the solution is defined on which additional constraints should be placed. These equations may be numerically solved using finite difference schemes to approximate the various spatial derivatives of the unknown function. Standard ODE solving schemes may be employed in parallel to solve for sequential time steps. Finally, the boundary values are fixed subject to the boundary conditions. This is then repeated for the desired number of timesteps.

There are a large variety of finite difference schemes which may be employed to approximate the spatial derivatives. These schemes vary in terms of accuracy, stability, and computational cost. Our goal is to build software that is sufficiently modular that the user can specify the particular finite difference scheme at runtime. In addition, we want to allow the user relatively simple control over the process of implementing a new form of PDE as well as adjusting the geometry of the given system. Object oriented software development seems not to be the norm for Finite Difference PDE solvers and computational researchers in general. A new code base for each new PDE is generally developed from scratch in a script-like fashion. This makes it difficult to maintain, reuse, and extend existing code. There are some existing libraries for object oriented PDE solvers, but they tend to be for finite element or finite volume solution methods. Since we could not find

a widely used, object oriented Python library for solving PDEs with finite difference methods, we thought it would be an interesting and productive exercise to build our own. In particular, our implementation works for parabolic PDEs, which are better behaved numerically than hyperbolic PDEs.

# 3 UML Diagram

(See Figure 1)

# 4 Overall Design

The blueprint of this project consists of three sections:

1. problem setup, including:
   (a) problem
   (b) linear operators
   (c) grid
   (d) initial and boundary conditions
2. driver section for both spatial and time parts
3. output

In the first section, there should be an abstract problem class with concrete subclasses defining the constant parameters and RHS function operators. Specifically, a model problem is the conductive heat equation in the form

$$\frac{\partial T}{\partial t} = \nabla^2 T \tag{4.1}$$

where $\nabla^2$ is the Laplacian operator defined as $\nabla^2 = \sum_{i=1}^{n} \frac{\partial}{\partial x_n}$. The representation of this problem in code as a function of an operator object `laplacian` should sit in a child class of Problem (`conduct_heat_eqn.py`). An operator object is either a `GridOperator`, which when called returns an object of the same size operated on by an operator, or a `VectorOperator` which when called returns an object of a modified size. Vector operators are made up of individual grid operators. An example of a grid operator would be $\frac{\partial}{\partial x}$ or a Laplacian while a vector operator would be like Grad or Div.

All FD linear operators may be expressed as an `OperatorMatrix`, where if the entire N-dimensional statevector would be flattened into a single 1-D vector then the operator matrix $L$ approximates the operator as

$$\mathcal{L}[\underline{\mathbf{u}}] \approx L \cdot \underline{\mathbf{u}} \tag{4.2}$$

Generating these `OperatorMatrix` for a particular implementation of a finite difference scheme occupies a significant portion of our code base.

A grid operator contains a `NDOperatorScheme` (`op_nd_scheme.py`) which contains the particular FD encoding of the grid operator. The operator scheme in turn contains an `OperatorND` which defines the FD method to be used in the center of the grid and `EdgeOperator`s which define a FD method to use at the edges of a grid where the stencil of the interior method will no longer fit on the grid. There can be either constant edge operators, which define problems where the boundary is a value, or periodic edge operators, which handle cases where the boundary wraps back around to the other side of an underlying grid.

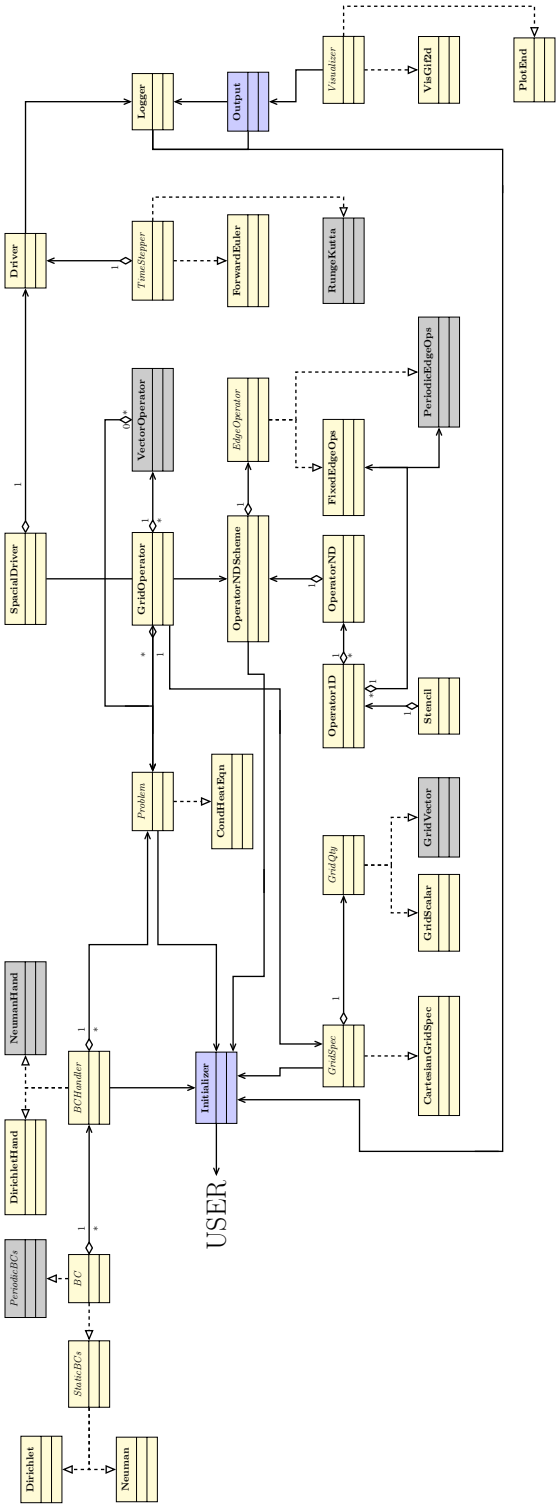



Figure 1: Implemented structure of our software. Yellow are classes implemented in our code base. Grey are classes which have not yet been implemented. Blue are classes which are partially implemented through hardcoding and would need further work before a final release.

Both ND operators and edge operators consist of `Operator1D` objects which contain a `Stencil`, which is the set of points given in indexes relative to the current point to use in a FD calculation and a set of weights for the given points. The weights are calculated for an arbitrary stencil per the algorithm given in [6]. An ND scheme needs enough operators in both the interior ND operator and the edge operators to match the dimensions of the `Grid` that will be evaluated.

For the purpose of defining and solving the given PDE equation, establishment of a proper domain in space is also one of our key considerations. In our design, two classes are responsible: a `GridSpec` which contains information about the spatial domain of the problem (we have implemented concretely `CartesianGridSpec` subclass) and a `GridQty` which holds values for each grid point (we have implemented a concrete `GridScalar` subclass to hold scalar values, though our code should be robust for higher dimensional grid quantities). With a specific grid, the grid operators necessary for te problem may be constructed by calling `populator.py` to generate the `OperatorMatrix` that can act on the underlying grid.

The last piece of this section, incorporating IC and BCs into our PDE solving, is designed to be realized by a series of boundary condition classes. First of all, a boundary condition handler class is addressed here to apply PDE problem's boundary conditions (`BC Handler.py`, `BCs.py`), which for now only contains a Dirichlet boundary condition class (`dirichlet_hand.py`), but we also leave room for a possible Neumann boundary condition class (planned) under it. A static boundary condition class could also be necessary in this part (`static_bcs.py`).

In the second section, we focus on the driver classes needed for both spatial and time part of our target PDE equation. We first created a package driver class, as the main driver class for solving the PDE problem, containing methods to call the main methods of the spatial driver and time stepper (`driver.py`), followed by the spatial driver class (`spatial_driver.py`) and abstract time stepper class (`time_stepper.py`). Currently, we aim to set Forward Euler method as its subclass (`forward_euler.py`). It uses a simple forward finite difference approach to estimate a value at the next time step, which is done with the approximation

$$u(t + dt) \approx u(t) + dt * \text{RHS} \tag{4.3}$$

where RHS is the right hand side of some parabolic PDE.

Finally, a logger class is used as a list wrapper appending time and grid values to a list stored in the object (`logger.py`), which subsequently output to visualization objects that handle making plots of the data.

# 5 Designed Functionality

In this project our goal was to create an object oriented library for solving parabolic partial differential equations. For a function $u(x, t)$, these are equations in the form

$$Au_{xx} + 2Bu_{xt} + Cu_{tt} + Du_x + Eu_t + F = 0 \tag{5.1}$$

that satisfy the condition

$$B^2 - AC = 0. \tag{5.2}$$

The prototypical example of this class of PDE is the 1D heat equation,

$$u_t = \alpha u_{xx}. \tag{5.3}$$

We wanted to design a modular, object oriented library that would solve this equation using user-defined finite difference formulae. Our design is meant to be extensible to other PDEs as long as the spatial and temporal terms in the equation can be appropriately separated. With this in mind the specific equation to be solved was designed to be implemented as a concrete child class of the abstract parent class Problem. In this way the equations can be easily exchanged by simply writing a new class implementation. We also

attempted to design the project in such a way as to allow the user to simply define what order spatial derivatives they desired and a set of stencil points to be used to approximate those spatial derivatives. The code was then intended to take those inputs and generate a set of operators that could be applied to the grid of function values. With all of these inputs as well as appropriate initial and boundary conditions, the library is meant to approximate the solution to the PDE and output the result either as a set of snapshots or as an animation of the time evolution of the system.

# 6 Git Workflow

For this project we attempted to use a forking workflow with GitHub. In this type of workflow, there is one main repository on GitHub which contains the most updated version of the functioning code base. From this main repository, each person on the development team creates what is known as a fork of the main repository and each fork is stored on the GitHub of the respective team members. When a member wants to add or further develop a feature of the code base, they create a new branch on their fork that is used for the development process. During development, the team member commits and pushes to their branch only. Once the development is complete and the developer is satisfied with their new code, they create a pull request to merge their development branch with the master branch of the original main repository. These changes are then reviewed and, assuming the changes will not break anything, the team members branch is pulled into the master branch of the main repository. All team members keep their own forks up to date by fetching the changes from the upstream repository, merging them with their local master branch, and then pushing the changes to their fork on GitHub.

As a team, we managed to stick pretty closely to this workflow prescription. Each team member did work on their own fork of the main repository. However, we did not realize until later that pull requests were only supposed to come from branches other than master. This led to an issue where a part of the code's functionality desperately needed to be implemented in the main repo, but there wasn't a branch that had the needed functionality that didn't also contain incomplete, outdated, broken code. This was solved by making a new branch with the desired functionality, deleting the broken code for that branch, and making a pull request from that branch.

We also selected one member of the team who would retain control of the main upstream repository and be in charge of reviewing and merging pull requests. This project is relatively small which made this a manageable job, but ideally these responsibilities would be divided up so that each member is responsible for reviewing changes to different sections of the code base.

# 7 Testing

We used the python `unittest` module to implement our tests. Unittest provides a logical testing structure (a test class for every implemented class), and is easily integrable with automated testing suites. We employed an as-you-go style of unit testing, with more tests written as needed afterwards. By unit testing our code as we individually built our separate features, we ensured that the separate pieces of the code worked. Then, when we slotted them together, we could focus our integration testing on the problematic interfaces between classes, rather than having to continue debugging the classes themselves. There were some hurdles; in order to be effective, unit testing has to be thorough, and with our immovable deadline it's possible that the tests we wrote don't fully cover all implemented cases. We also didn't properly utilize the `unittest.Mock` library, which hampered our ability to write unit tests for code which depended on classes not yet implemented. We were unable to use Mock objects due to confusion about their implementation, and the time constraints preventing us from fully learning their implementation. Finally, while we designed our code to be agnostic to implementation our testing code was not always so flexible and changing some implementation details required a somewhat extensive overhaul of the test code. Despite this, the unit testing helped keep our code

as functional as possible.

Continuous integration of our unit and integration tests was managed by Travis CI. It served as the primary screening for code before it was integrated into the main repo- while everything still needed human inspection, the automated testing made the process more efficient and less prone to human error. There were some minor struggles with configuring the .travis.yml file, such as when we realized that we needed to install some of the python libraries our code needed on the virtual machine, which required learning new syntax.

In order to test the scientific validity of our code, we tested our fully integrated code on a parabolic PDE with a well known behavior, the 2d heat equation with Dirichlet boundary conditions enforcing zero temperature along the boundary. In particular, we utilized the same initial conditions (up to a constant) as Step 7 in the 12 steps to Navier-Stokes (see references). This allowed us to qualitatively confirm the behavior of our simulation. In particular, we were able to observe the expected diffusion effect.

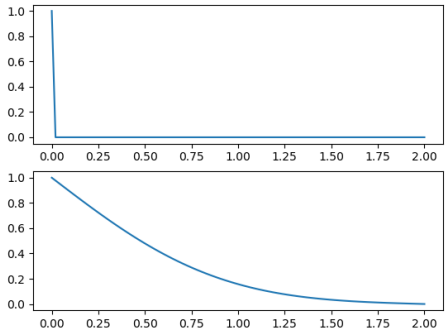
# 8 Lessons Learned from the Collaborative Object-Oriented Design Process

There were several issues that made development more difficult. Despite our significant efforts in designing the code before implementation, we found it difficult to determine the exact handshakes and exchange of information that would be needed between the various classes, and the means by which they would occur. The difficulty arose both due to inexperience with designing such interfaces as well as a significant time crunch forcing us to start implementing before each piece had been thoroughly considered. This led to confusion, especially in places where one person's feature would interface with another's. Integration testing proved valuable for ensuring that this communication difficulty didn't impact the functionality of our code, though the actual assembly of the integrated pieces took a lot of bug fixing to ensure that variable names, etc. were consistent across our code base. In the future we will recognize the importance of pre-planning and having solid APIs in place before any code is written and undertake to produce such specifications so that when it comes time to actually implement functions it is a relatively clear and painless experience.

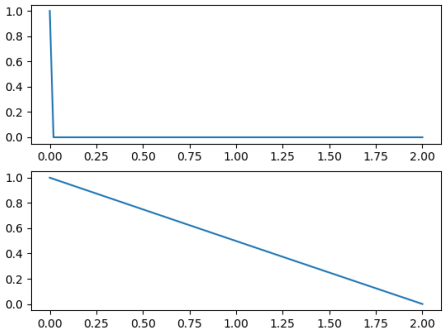
We also experienced a downside of delegating interdependent features to separate individuals. We often experienced bottlenecks, where team members would be waiting for the implementation of a particular feature which was proving more challenging than expected. Due to our compartmentalization into our own features, it was often difficult for team members to assist others on their work. The linear operator and scheme implementation in particular was nontrivial, as was the amount of time necessary for other team members to learn about it sufficiently to be able to help with it. These problems were further compounded by inadequate initial planning which meant that the design details of the operator functions were often in a heavy state of flux, and it was difficult to out source problems to other team members. The close collaboration on the same complex feature also made it more likely for git-related mistakes to occur, but with strong communication, our team was able to avoid catastrophic errors.

# 9 Achieved Functionality

In the end we were able to achieve a surprising amount of the functionality that we set out to create. While the construction of the approximate derivative operations still requires significant input from the user, the framework now exists to produce essentially infinite finite difference approximations of spatial derivatives from relatively few inputs. We were also able to create the abstract data containers, termed GridSpecs (for grid specifications) and GridQtys (grid quantities), which hold all the physical information about the system. These data containers also expose methods that wrap the underlying implementation details such that the data can be passed around and manipulated by the rest of the objects without necessarily needing to know exactly how the containers work. We were also able to successfully implement the standard

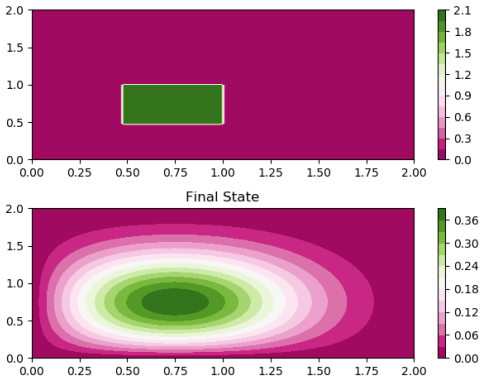


(a) First and last frame of results for a 1D heat equation run with 100 grid steps and a dt of 0.001 seconds for five seconds, with an $\alpha = 0.05$. Note that the initial sharp profile is smoothed by the Laplace operator. Code evaluated with a first order central difference scheme in the center, a first order forward difference scheme on the left boundary, and a first order backward difference scheme on the right boundary. This figure was generated using the PlotEnd class.

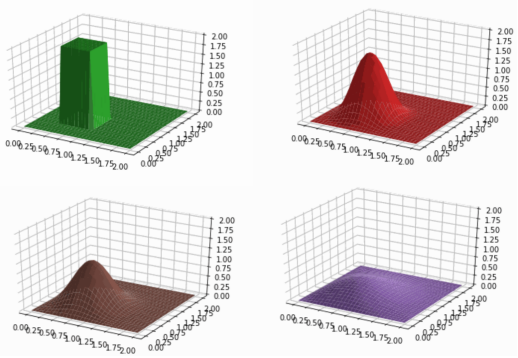


(b) First and last frame of results for a 1D heat equation run with 100 grid steps and a dt of 0.001 seconds for five seconds, with an $\alpha = 0.05$. Note that the initial sharp profile goes to a stead-state linear profile as expected for the heat equation. Code evaluated with a first order central difference scheme in the center, a first order forward difference scheme on the left boundary, and a first order backward difference scheme on the right boundary. This figure was generated using the PlotEnd class.

Figure 2




(a) First and last frame of results for a 2D heat equation run with 100 grid steps in both directions and a dt of 0.001 seconds for five seconds, with an $\alpha = 0.05$. The initial tophat distribution is smoothed over time as the heat relaxes down to the boundaries. This figure was generated using the PlotEnd class.



(b) Results for a run of the 2D heat equation with 31 grid steps, a time step of 0.01 for 10 seconds, and an $\alpha = 0.05$. Code evaluated with a first order central difference scheme in the center, a first order forward difference scheme on the left boundary, and a first order backward difference scheme on the right boundary. This figure was generated using the VisGif2D class.

Figure 3

algorithm for solving PDEs with a finite difference approximation which consists of separating the spatial and temporal aspects of the problem, approximating the spatial derivative, and then using the values of those approximations to advance the solution one step in time. The only ODE method that was implemented is the Forward Euler scheme for solving the temporal problem, but this could easily be switched out by writing another subclass of TimeStepper or potentially wrapping some existing ODE integrator. We tested the full functionality of the code with 1D and 2D heat equations and the results can be seen in the figures above. The diffusive behavior of the temperature is certainly captured qualitatively, although the quantitative accuracy has not been evaluated. It is also evident from the figures that basic data logging and visualization have been implemented, as well.

Other notable functionality includes the ability to define the spatial domain over which to solve the problem, as well as the spatial discretization, the start and end times for the solution, and the temporal discretization. The initial function values and the boundary conditions are also able to be specified by the user. Currently only constant Dirichlet type boundary conditions have been implemented, but the design framework allows for extensibility to Neumann and/or periodic boundary conditions as well.

The code does not address the issue of allowing the user the ability to build a solver suite at runtime. All of our classes are structured to allow this to happen with the implmentation of suitable Factory classes. However, due to time constraints, we prioritized getting the functionality as a PDE solver working rather than developing the interface for the user. However, we have implemented pieces of code, such as having a `Problem` store the linear operators it needs, that will enable creation of Factory classes with relative ease in the future.

# 10 Profiling

Our implementation of a Finite Difference solver requires the creation and manipulation of extremely large matricies. If a given problem has a Grid with dimensions $q_1, q_2, \ldots, q_n$ then if we define $N = \prod_{i=1}^{n} q_i$ the dimensions of a single `OperatorMatrix` will be $N \times N$. Thus, the creation of these operators as well as the method by which they were evaluated on a grid stood out to us as a potential hotspot in our code as well as an extremely large memory hog.

We had orginally implemented `OperatorMatrix` as a dense `numpy` array. The results for creating and using a `GridOperator` with this implementation are given in Figure 4a. The grid operators `__init__` method, which is where the operator matrix is defined and populated, took a total of 6.7 seconds of time and the whole operation took 7.4 seconds. In addition, the other highest time intensive processes are all related to the population of the underlying matrix.

To address this, we switched operator matrix to being implemented as a `scipy` sparse `lil_matrix`. This sparse matrix is optimized for populating. Figure 4b shows the results of profiling with this test. The total time as been cut nearly in half to 3.7 seconds and the creation of the grid operator now only takes 3.2 seconds. The code was well structured so making this change only required changing a few lines. However, the test code was more reliant on numpy methods to evaluate proper outputs and so had to be somewhat overhauled for this change.

Finally, in the interest of further imporovement, we implemented a method to turn the operator matrix into a `crs_sparse` matrix after it had been populated. This also allows us to check if a matrix has been populated before we use it. This implementation of a sparse matrix is better suited to performing matrix operations and the resuts of profiling are show in Figure 4c. It did not have a huge improvement in this round of profiling but would perhaps be more apparent when the operator is applied to a grid many more times, as it is in the main code.

Another major source of time saving would be to overhaul the populator functions with kroenecker commands to help eliminate things like calls to `_get_single_index`. However, such changes will have to wait for another

```
       590568 function calls (587175 primitive calls) in 7.412 seconds

 Ordered by: cumulative time
 List reduced from 1222 to 10 due to restriction <10>

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  489/1    0.003    0.000    7.414    7.414 {built-in method builtins.exec}
      1    0.248    0.248    7.414    7.414 genprofile.py:1(<module>)
      1    0.000    0.000    6.693    6.693 genprofile.py:9(main)
      1    0.165    0.165    6.655    6.655 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:37(__init__)
      2    4.356    2.178    4.356    2.178 /home/james/apc524/Main_PDE_Repo/src/operatormatrix.py:32(__add__)
      6    0.000    0.000    2.134    0.356 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:98(_apply_op)
      6    0.157    0.026    2.131    0.355 /home/james/apc524/Main_PDE_Repo/src/populator.py:30(populate_op)
  20000    0.626    0.000    1.686    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:48(_set_row)
  80000    0.590    0.000    0.918    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:61(_get_single_index)
  210/8    0.003    0.000    0.473    0.059 <frozen importlib._bootstrap>:978(_find_and_load)
```

(a) Profiling of creation of a GridOperator and applying it to a Grid as originally implemented.

```
       1180560 function calls (1176671 primitive calls) in 3.683 seconds

 Ordered by: cumulative time
 List reduced from 1410 to 10 due to restriction <10>

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  537/1    0.004    0.000    3.684    3.684 {built-in method builtins.exec}
      1    0.006    0.006    3.684    3.684 genprofile.py:1(<module>)
      1    0.000    0.000    3.214    3.214 genprofile.py:9(main)
      1    0.001    0.001    3.211    3.211 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:37(__init__)
      6    0.000    0.000    3.089    0.515 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:98(_apply_op)
      6    0.137    0.023    3.087    0.514 /home/james/apc524/Main_PDE_Repo/src/populator.py:30(populate_op)
  20000    0.542    0.000    2.706    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:48(_set_row)
  60000    0.156    0.000    1.519    0.000 /home/james/apc524/Main_PDE_Repo/src/operatormatrix.py:27(__setitem__)
  60000    0.466    0.000    1.363    0.000 /home/james/anaconda3/lib/python3.7/site-packages/scipy/sparse/lil.py:336
_setitem__)
  80000    0.497    0.000    0.774    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:61(_get_single_index)
```

(b) Profiling of GridOperator when the operator matrix has been switched to a sparse matrix.

```
       1180696 function calls (1176807 primitive calls) in 3.596 seconds

 Ordered by: cumulative time
 List reduced from 1411 to 10 due to restriction <10>

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  537/1    0.003    0.000    3.598    3.598 {built-in method builtins.exec}
      1    0.000    0.000    3.598    3.598 genprofile.py:1(<module>)
      1    0.000    0.000    3.139    3.139 genprofile.py:9(main)
      1    0.001    0.001    3.137    3.137 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:37(__init__)
      6    0.000    0.000    3.016    0.503 /home/james/apc524/Main_PDE_Repo/src/gridoperator.py:100(_apply_op)
      6    0.135    0.022    3.014    0.502 /home/james/apc524/Main_PDE_Repo/src/populator.py:30(populate_op)
  20000    0.535    0.000    2.644    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:48(_set_row)
  60000    0.152    0.000    1.484    0.000 /home/james/apc524/Main_PDE_Repo/src/operatormatrix.py:37(__setitem__)
  60000    0.454    0.000    1.331    0.000 /home/james/anaconda3/lib/python3.7/site-packages/scipy/sparse/lil.py:336
_setitem__)
  80000    0.479    0.000    0.747    0.000 /home/james/apc524/Main_PDE_Repo/src/populator.py:61(_get_single_index)
```

(c) Profiling of a GridOperator when the operator matrix is optimized for calculation.

Figure 4

time.

# 11 Conclusion

In conclusion, creating a finite difference PDE solver library was a valuable and interesting project. We managed to build something fairly complex that functions as expected with a physically meaningful system and which could continue to be maintained and updated if desired. We also learned a great deal about the software engineering process and the importance of thorough design steps and purposeful Git workflows. We gained hands-on experience in how scientific programming can benefit from the practice of object-oriented programming, which is still underappreciated and underutilized. So, as we continue in our scientific careers we can make the body of scientific programming slightly better, more efficient, more modular and easier to use and understand.

# References


[1] K. Ahlander. "An Object-oriented Approach to Construct PDE Solvers". In: *Scientific Report, Dept. of Scientific Computing, Uppsala University* (1996).

[2] J. D. Anderson. *Computation Fluid Dynamics: The Basics with Applications*. McGraw-Hill, Inc., 2005.

[3] L.A. Barba and G.F. Forsyth. "CFD Python: the 12 steps to Navier-Stokes equations". In: *Journal of Open Source Education* 1.10 (), p. 21. DOI: https://doi.org/10.21105/jose.00021.

[4] Stefan Bilbao. "Grid Functions and Finite Difference Operators in 2D". In: John Wiley Sons, 2009. Chap. 10.

[5] Emmett Dudley. *Updating z data on a surface$_p$lotin$Matplotlib$animation*. URL: https://stackoverflow.com/questions/45712099/updating-z-data-on-a-surface-plot-in-matplotlib-animation. (accessed: 08.16.2017).

[6] *Finite difference coefficient*. URL: https://en.wikipedia.org/wiki/Finite_difference_coefficient. (accessed: 01.14.2020).

[7] T.B. Hua. *PyCFD*. URL: http://ohllab.org/CFD_course/index.html.

[8] Marina Mele. *Modifying the $_{a}dd_{methodofaPythonClass}$*. URL: http://www.marinamele.com/2014/04/modifying-add-method-of-python-class.html. (accessed: 01.14.2020).

[9] *pcolormesh*. URL: https://matplotlib.org/3.1.0/gallery/images_contours_and_fields/pcolormesh_levels.html#sphx-glr-gallery-images-contours-and-fields-pcolormesh-levels-py. (accessed: 01.14.2020).

[10] *Pillow: the friendly PIL fork*. URL: https://python-pillow.org/. (accessed: 01.14.2020).

[11] M. et al. Thune. "Modern Software Tools for scientific computing". In: Springer Science+ Business Media, LLC, 1997. Chap. 9: Object-Oriented Construction of Parallel PDE Solvers. Book edited by Arge, E. et al.