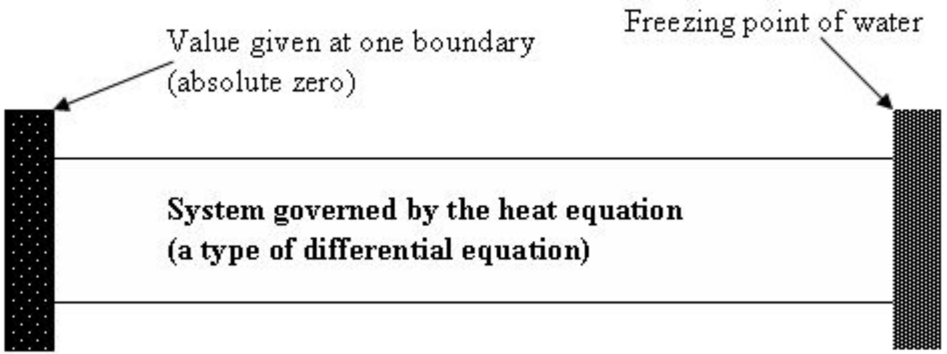
SplitFXM - Boundary Value Problems in Python

Pavan B Govindaraju

Breaking down the title - 1

Boundary Value Problem (BVP)

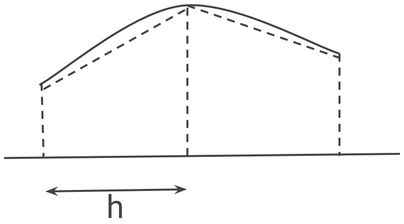


Many physical phenomena are governed by **differential equations** with **values/conditions at the boundaries**

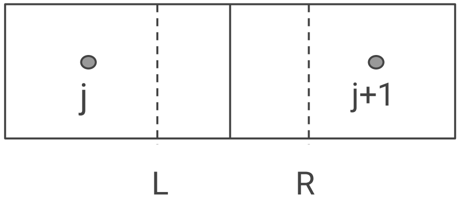
Breaking down the title - 2

<u>F-X-M</u>

- Not always possible to find analytical solutions to BVPs
- Differential equation is approximated using numerical derivatives etc.
- Two common approaches
  - Finite Difference (FDM)
  - Finite Volume

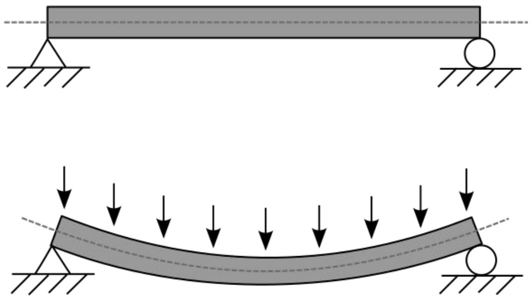


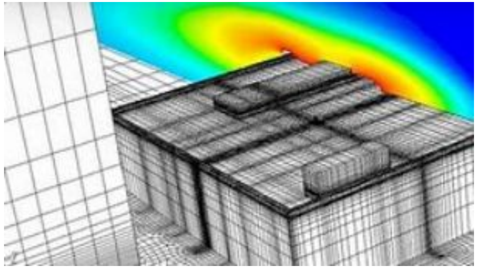
Finite Difference

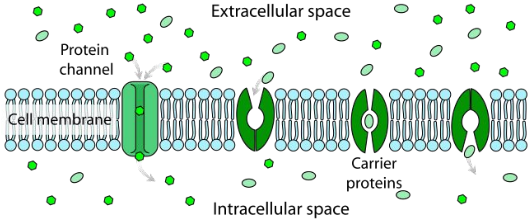


Finite Volume

Examples







Applications in various fields

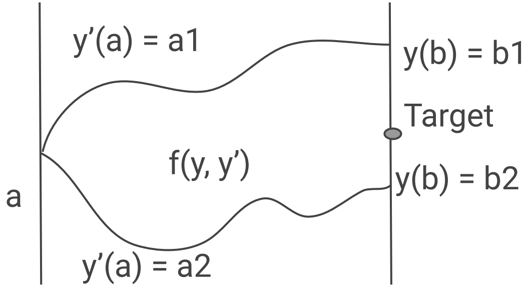
- Design Optimization
- Biology
- Supersonic flows
- Combustion

and so on

Existing Solvers in Python

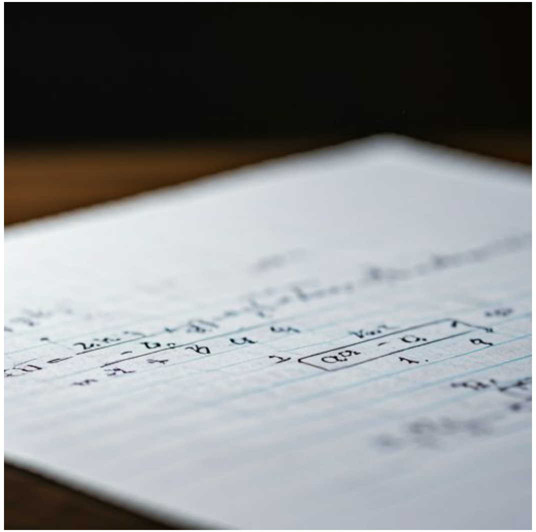
`solve_bvp` - but very limited use cases

- Need to reformulate the problem
- Support for limited boundary conditions
- Good for textbook problems



Need for better solvers

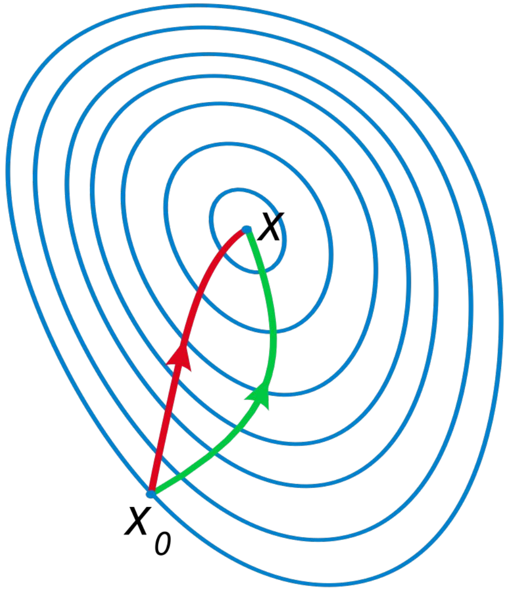
- BVPs are nonlinear many times
  - No guaranteed convergence
- Involves large matrix inversions - should support sparse matrices
- Easy interface
  - solving the problem = writing down the equation
- Support for various kinds of boundary conditions
- Mesh refinement - sharp gradients to be resolved



Newton Iteration

- Used to solve the nonlinear equation through minimizing the error

- Assumes locally quadratic and goes in the minimum direction as opposed to gradient descent (green)



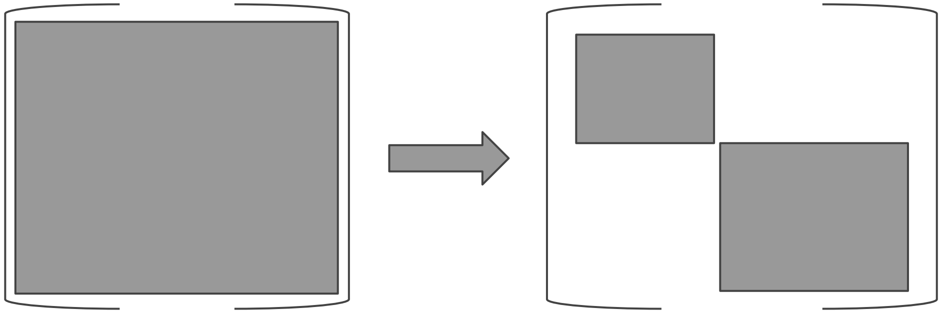
$$F(x_n + s_n) = 0 \quad \Rightarrow \quad F(x_n) = -J(x_n) \cdot s_n$$

$$s_n = \boxed{-J^{-1}(x_n)} F(x_n)$$

Expensive to invert

SplitNewton - Method

Jacobian can be approximated using two (and hence recursively) sub-systems



- Easier to invert
- More numerically stable - low condition numbers

SplitNewton - Proof

Let $\mathbf{x} := (\mathbf{y}, \mathbf{z})$ and $F(\mathbf{x}) := (G(\mathbf{y}, \mathbf{z}), H(\mathbf{y}, \mathbf{z}))$. Define a sequence

$\mathbf{x}_k := (\mathbf{y}_k, \mathbf{z}_k)$ and $\mathbf{s}_k := \mathbf{x}_{k+1} - \mathbf{x}_k = (\mathbf{y}_{k+1} - \mathbf{y}_k, \mathbf{z}_{k+1} - \mathbf{z}_k) := (\mathbf{t}_k, \mathbf{u}_k)$

where

$\mathbf{t}_k = \{\delta_k \mathbf{I} - G'(\mathbf{y}_k, \mathbf{z}_k)\} G(\mathbf{y}_k, \mathbf{z}_k)$

$\mathbf{u}_k = \{\delta_k \mathbf{I} - H'(\mathbf{y}_k, \mathbf{z}_k)\} H(\mathbf{y}_k, \mathbf{z}_k)$

where G' and H' exists and invertible $\forall (\mathbf{y}_k, \mathbf{z}_k)$

and $\delta_k = min(\delta_0 ||F_0|| / ||F_k||, \delta_{max})$

Then, $\lim_{k\to\infty} (\mathbf{y}_k, \lim_{l\to\infty} z_k^l) = \mathbf{x}^*$, where $F(\mathbf{x}^*) = 0$

Proof:

From convergence result in [1] for pseudo-transient continuation, on applying to $H(\mathbf{y}_k, \mathbf{z}_k^l) := H_{\mathbf{y}_k}(\mathbf{z}_k^l)$

$\lim_{k\to\infty} (\mathbf{y}_k, \lim_{l\to\infty} \mathbf{z}_k^l) := \lim_{k\to\infty} (\mathbf{y}_k, \mathbf{z}_k^*)$, it follows that $\lim_{k\to\infty} H_{\mathbf{y}_k}(\mathbf{z}_k^*) = 0$ and thus $\lim_{k\to\infty} H(\mathbf{y}_k, \mathbf{z}_k^*) = 0$

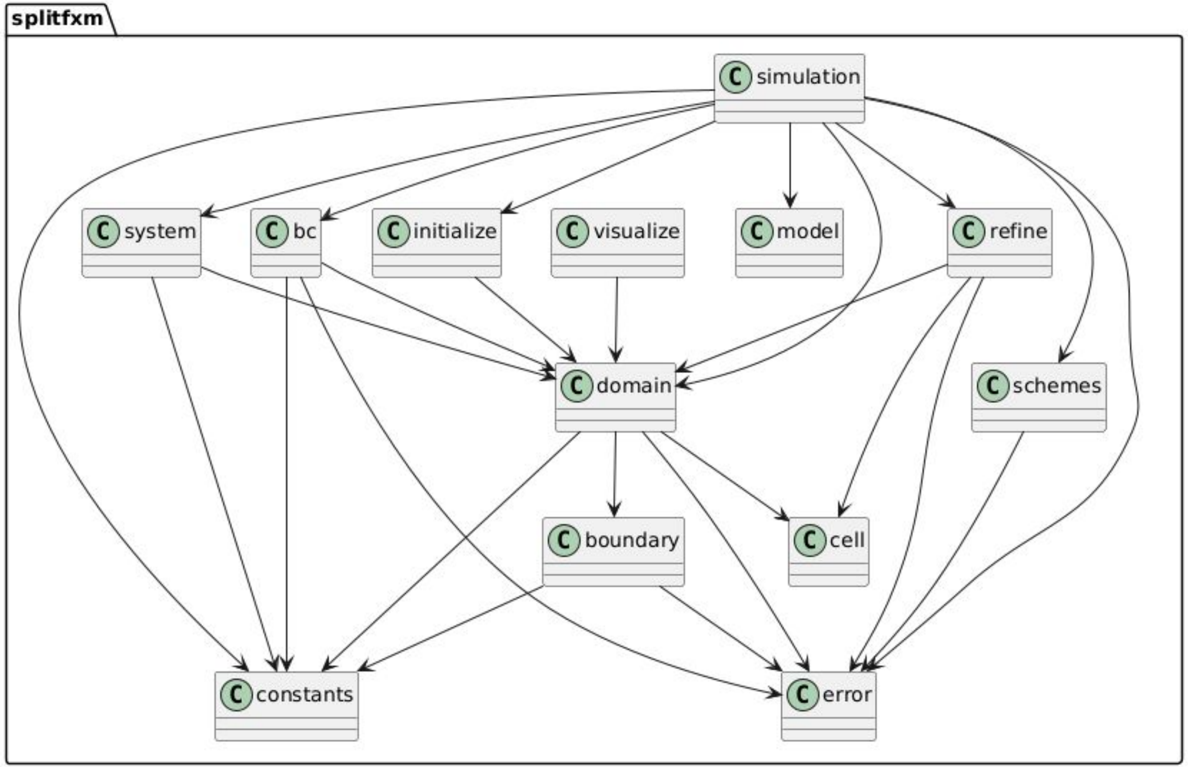
Now, $\lim_{k\to\infty} (G(\mathbf{y}_k, \mathbf{z}_k^*), H(\mathbf{y}_k, \mathbf{z}_k^*)) = \lim_{k\to\infty} (G(\mathbf{y}_k, \mathbf{z}_k^*), 0)$

Let $G(\mathbf{y}_k, \mathbf{z}_k^*) := G_{\mathbf{z}_k^*}(\mathbf{y}_k)$

From convergence result in [1] applied to $G_{\mathbf{z}_k^*}(\mathbf{y}_k)$, we have $\lim_{k\to\infty} G_{\mathbf{z}_k^*}(\mathbf{y}_k) := G_{\mathbf{z}^*}(\mathbf{y}^*) = 0$

Thus, $G(\mathbf{y}^*, \mathbf{z}^*) = 0$ and $H(\mathbf{y}^*, \mathbf{z}^*) = 0$ as $\lim_{k\to\infty} H_{\mathbf{y}_k}(\mathbf{z}^*) = 0$

SplitFXM - UML Diagram



Generated using pyreverse

SplitFXM - Basic Usage

```python
# Define the problem
method = 'FVM'
m = AdvectionDiffusion(c=0.2, nu=0.001, method=method)
# nx, nb_left, nb_right, variables
d = Domain.from_size(20, 1, 1, ["u", "v", "w"])
ics = {"u": "gaussian", "v": "rarefaction", "w": "tophat"}
bcs = {
    "u": {
        "left": "periodic",
        "right": "periodic"
    },
    "v": {
        "left": {"dirichlet": 3},
        "right": {"dirichlet": 4}
    },
    "w": {
        "left": {"dirichlet": 2},
        "right": "periodic"
    }
}
s = Simulation(d, m, ics, bcs, default_scheme(method))
```

Equation

Domain

Initial Conditions

Boundary Conditions

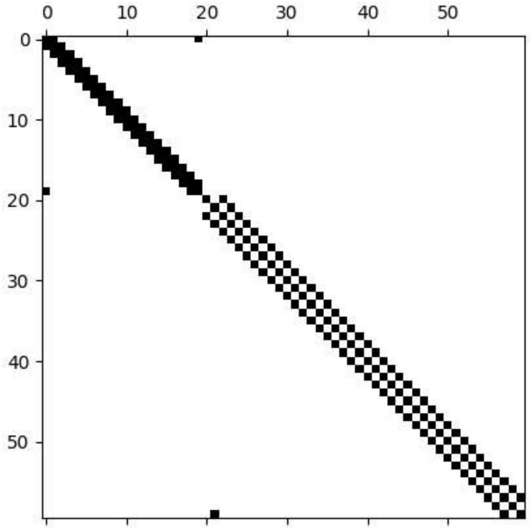
Simulation

Additional Features - Sparse Jacobians

Constructing dense Jacobians is a naive approach and makes most problems intractable

Example benchmark: Reduction from **20s to 0.5s**

Tricky to implement with split!

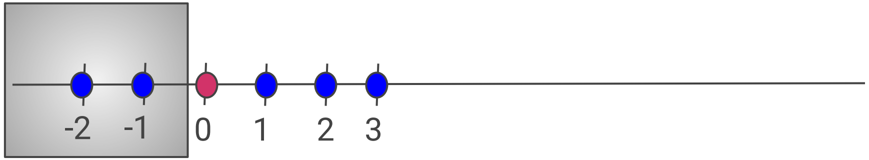


$$O(N_{\text{cell}}^3 \times N_{\text{var}}^3)$$

$$O(N_{\text{cell}} \times N_{\text{var}}^2)$$

Additional Features - Asymmetric Stencils

Ghost points allow computing derivatives at boundary locations



Stencil - points used to compute the derivative. Can be asymmetric as well

Both `lb` and `rb` were being specified in `Domain`

Additional Features - Vector Finite Volume

Finite Volume schemes require additional work for vector-valued functions

Eigenvalue decomposition

$$J = R\Lambda R^{-1}$$

Transformation to characteristic variables and evaluating flux

$$w = R^{-1}u \qquad F_{char} = F(w)$$

Convert back to actual flux

$$F = RF_{char}$$

Need for Code Optimization

On running `cProfile` for the code

ncalls	tottime	percall	cumtime	percall	
1820	0.3563	0.0001958	0.6305	0.0003464	fv_transport.py:27(residuals)

Residual computation is the most time-consuming step

Expected as Jacobian is numerically evaluated - best place to optimize

Cython

- Very much like Python but compiles to C functions
- Some parts of the code can be written in Cython and others in Python
- C is “as fast as it gets” in scientific computing
- Gain from compilation - Python is “interpreted”
- Offers 10-1000x speedups as compared to basic Python

Writing in Cython

- Provides annotation to show “Python” parts that can be sped up
- Need to specify data-types
- Loops can be parallelized using `prange`
- Variables need to be declared beforehand
- Can use C-based libraries like `numpy` (`cimport`)
- Topic on its own

```
cdef int[:] get_hp_lengths(str seq):
    ''' Calculate HP length of substring starting at each index. '''

    hp_lens_buf = np.zeros(len(seq), dtype=np.intc)
    cdef int[:] hp_lens = hp_lens_buf
    cdef Py_ssize_t start, stop

    for start in range(len(seq)):
        for stop in range(start+1, len(seq)):
            if seq[stop] != seq[start]:
                hp_lens[start] = stop - start
                break
    if len(seq):
        hp_lens[-1] += 1
    return hp_lens
```

Building Cython

Using same `setup.py` but with additional commands

```
extensions = [
    Extension(
        name="splitfxm.derivatives",
        sources=["splitfxm/derivatives.pyx"],
        include_dirs=[np.get_include()],  # Include NumPy headers
```

```
setup(ext_modules=cythonize(extensions))
```

Can specify C compilation flags as well (`-fopenmp` for example)

Cython in SplitFXM - Derivatives

Simple array-based computations

```
Fl = F[0, :]
Fr = F[2, :]
dx = cell_sub_x[2] - cell_sub_x[0]

for i in prange(n, nogil=True):
    Fans[i] = (Fr[i] - Fl[i]) / dx
return Fans
```

Double array needs to be pre-defined - helper method `allocate_double_array` written

Fast array allocation in Cython - short topic in itself

Cython in SplitFXM - Flux

Slightly more elaborate

- Defined `matvec` - Matrix-Vector multiplication helper

Program using mostly for loops and basic operations for maximum speedup

```
for i in prange(n, nogil=True):
    u_diff[i] = uc[i] - ul[i]

for i in prange(n, nogil=True):
    Fw[i] = 0.5 * (Fl[i] + Fr[i]) - 0.5 * sigma * u_diff[i]

Fl = F[1, :]
Fr = F[2, :]

for i in prange(n, nogil=True):
    u_diff[i] = ur[i] - uc[i]

for i in prange(n, nogil=True):
    Fe[i] = 0.5 * (Fl[i] + Fr[i]) - 0.5 * sigma * u_diff[i]

return Fw, Fe
```

Unit Testing

Suite in `pytest` with 100+ tests offering almost 100% coverage

Test execution time almost instantaneous

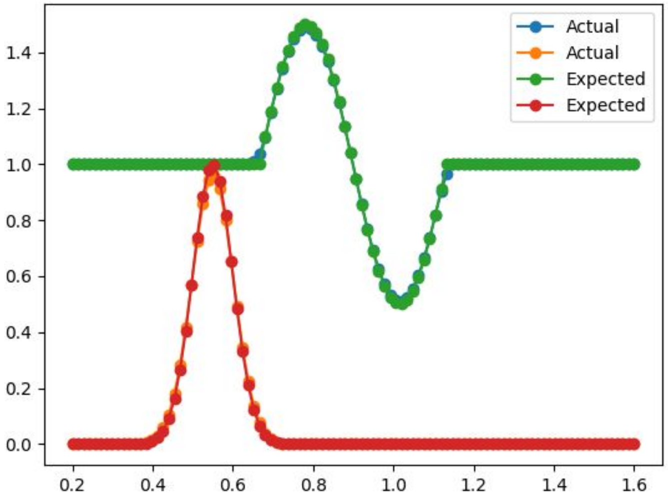
```
tests/test_bc.py ........
tests/test_boundary.py ........
tests/test_cell.py .........
tests/test_constants.py ..
tests/test_derivative.py .........
tests/test_domain.py ..................
tests/test_flux.py ...........
tests/test_generate.py ....
tests/test_initialize.py .....
```

- tests
  - test_bc.py
  - test_boundary.py
  - test_cell.py
  - test_constants.py
  - test_derivative.py
  - test_domain.py
  - test_flux.py
  - test_generate.py
  - test_initialize.py
  - test_model.py
  - test_refine.py
  - test_schemes.py
  - test_simulation.py
  - test_system.py
  - test_verification.py
  - test_visualize.py

Verification -Advection-Diffusion

Typical test where initial condition is advected in the domain for 1 cycle using periodic boundary conditions

Agreement upto machine precision

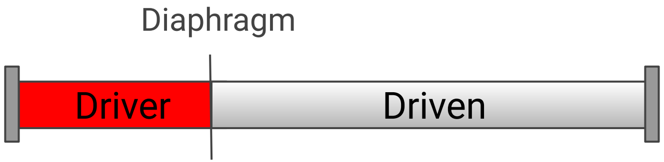


Verification - Shock Tube

Common test case for
finite-difference/volume solvers in 1D

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho \\ \rho u \\ \rho E\end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix}\rho u \\ \rho u^2 + p \\ \rho u \left(E + \frac{p}{\rho}\right)\end{pmatrix} = 0$$

Can be written as special case of
Advection-Diffusion equation



Verification - Shock Tube - 2

Using Lax-Friedrichs finite-volume scheme and RK45 time-stepping

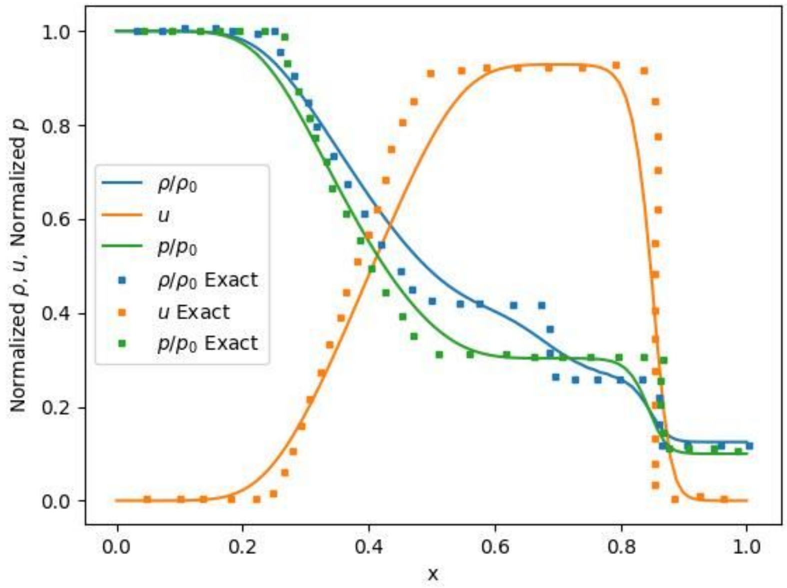
```
method = 'FVM'

s.evolve(t_diff=0.2)
```

Captures the various sections of the shock tube at t = 0.2

- Shock Location
- Rarefaction etc.

Solution accuracy limited by the scheme and its error



Documentation - mkdocs

Specification using `.yml` file

Code sections can be generated using plugins such as `mkdocstrings`

Latex equations can also be rendered from it

Single line push to GitHub
`mkdocs gh-deploy`

```yaml
site_name: splitfxm
theme:
  name: readthedocs
nav:
  - Home: index.md
  - Getting Started: start.md
  - Benchmark: benchmark.md
  - Pricing: pricing.md
  - Code Documentation:
    - Overview: api.md
    - Model: model.md
    - Domain: domain.md
    - Initial Conditions: ic.md
    - Boundary Conditions: bc.md
    - Finite-Volume Schemes: flux.md
    - Finite-Difference Schemes:
      - Overview: derivatives.md
```

Summary

- Boundary Value Problems and their applications
- Existing solvers and the need for better ones
- Newton and SplitNewton methods
- Additional features - Sparse Jacobians, Asymmetric Stencils, Vector support
- Cython and optimization
- Unit testing, verification
- Documentation generation using `mkdocs`

Thank You!

DOI 10.5281/zenodo.13882261

QR for the repo