MOLE User Manual

Release 1.1.0

Valeria Barra **Angel Boada** **Jared Brzenski**
Jose Castillo **Prit Chakalasiya**
Surinder Chhabra Singh **Johnny Delgado Corbino**
Tony Drummond **Miguel Dumett** **Joe Hellmers**
Arshia Ilaty **Katayoon Kaviani** **Oluchi Nzerem**
Giulia Pagallo **Christopher Paolini**
Valentina Rosano **Aneesh Murthy Srinivas**
Janani Priyadharshini Srinivasan **Manuel Valera**

Nov 03, 2025

INDEX

About MOLE

A high-order mimetic differential operators library for solving PDEs

High-order Accuracy Discrete analogs of vector calculus operators that satisfy local and global conservation laws

Structured Grids Works with uniform, non-uniform, and curvilinear staggered grids

Operator Set Includes Gradient, Divergence, and Laplacian operators with various boundary conditions

Dual Implementation Available in both C++ and MATLAB/ Octave with consistent interfaces

CHAPTER

ONE

INTRODUCTION

MOLE is a high-quality (C++ & MATLAB/Octave) library that implements high-order mimetic operators to solve partial differential equations. It provides discrete analogs of the most common vector calculus operators: Gradient, Divergence, Laplacian, Bilaplacian, and Curl. These operators (highly sparse matrices) act on staggered grids (uniform, non-uniform, curvilinear) and satisfy local and global conservation laws.

Mathematics is based on the work of Corbino and Castillo. However, the user may find helpful previous publications, such as Castillo and Grone, in which similar operators were derived using a matrix analysis approach.

MOLE comes with comprehensive documentation:

- **API Reference & User Guide**: Access our online Documentation
- **Building Documentation**: To build documentation locally, follow our Documentation Guide.

  Important Note: Performing non-unary operations involving operands constructed over different grids may lead to unexpected results. While MOLE allows such operations without throwing errors, users must exercise caution when manipulating operators across different grids.

1.1 Licensing

MOLE is distributed under a GNU General Public License; please refer to the *LICENSE* file for more details.

1.2 Community Guidelines

We welcome contributions to MOLE, including:

- Adding new functionalities
- Providing examples
- Addressing existing issues
- Reporting bugs
- Requesting new features

Please refer to our Contribution Guidelines for more details.

1.3 How to Cite

Please cite our work if you use MOLE in your research or software. Citations are helpful for the continued development and maintenance of the library.

```bibtex
@article{Corbino2024,
   doi = {10.21105/joss.06288},
   url = {https://doi.org/10.21105/joss.06288},
   year = {2024},
   publisher = {The Open Journal},
   volume = {9},
   number = {99},
   pages = {6288},
   author = {Corbino, Johnny and Dumett, Miguel A. and Castillo, Jose E.},
   title = {MOLE: Mimetic Operators Library Enhanced},
   journal = {Journal of Open Source Software} }
```

The archival copy of the MOLE User Manual is maintained on Zenodo. To cite the User Manual please use:

```bibtex
@misc{MOLE_user_manual,
   author       = {Barra, Valeria and
                   Boada, Angel and
                   Brzenski, Jared and
                   Castillo, Jose and
                   Chakalasiya, Prit and
                   Singh, Surinder Chhabra and
                   Corbino, Johnny Delgado and
                   Drummond, Tony and
                   Dumett, Miguel and
                   Hellmers, Joe and
                   Ilaty, Arshia and
                   Kaviani, Katayoon and
                   Nzerem, Oluchi and
                   Pagallo, Giulia and
                   Paolini, Christopher and
                   Rosano, Valentina and
                   Srinivas, Aneesh Murthy and
                   Srinivasan, Janani Priyadharshini and
                   Valera, Manuel},
   title        = {{MOLE User Manual}},
   month        = aug,
   year         = 2025,
   publisher    = {Zenodo},
   version      = {1.1.0},
   doi          = {10.5281/zenodo.16898575},
   url          = {https://doi.org/10.5281/zenodo.16898575},
}
```

CHAPTER

TWO

GETTING STARTED

Welcome to the getting started guide for MOLE (Mimetic Operators Library Enhanced). This guide will help you set up and run your first MOLE project.

2.1 Platform and Compiler Compatibility

Refer to the table below for compiler support across different operating systems when building MOLE.

OS / Compiler	GCC 13.2.0	AppleClang	IntelLLVM (icpx)
Linux	Yes	No	Yes
macOS	No	Yes	Yes

2.2 Prerequisites

To install the MOLE library, you'll need the following packages:

- CMake (Minimum version 3.10)
- OpenBLAS (Minimum version 0.3.10)
- Eigen3
- LAPACK (Mac only)
- libomp (Mac only)

For documentation build requirements, please refer to the Documentation Guide.

2.3 Package Installation by OS

2.3.1 Ubuntu/Debian Systems

```
# Install all required packages
sudo apt install cmake libopenblas-dev libeigen3-dev
```

2.3.2 macOS Systems

Install Homebrew if you don't have it already, then run:

```
# Install all required packages
brew install cmake openblas eigen libomp lapack
```

Troubleshooting Homebrew: If you encounter installation errors, try these steps:

```
# Fix permissions issues
sudo chown -R $(whoami) /usr/local/Cellar
# Fix shallow clone issues
git -C /usr/local/Homebrew/Library/Taps/homebrew/homebrew-core fetch --
↪unshallow
# Remove Java dependencies if they cause conflicts
brew uninstall --ignore-dependencies java
brew update
```

2.3.3 RHEL/CentOS/Fedora Systems

```
# Install all required packages
sudo yum install cmake openblas-devel eigen3-devel lapack
```

2.4 Building and Installing

1. Clone the repository:

   ```
   git clone https://github.com/csrc-sdsu/mole.git
   cd mole
   ```

2. Build the library:

   ```
   mkdir build && cd build
   cmake ..
   make
   ```

3. Install the library:

   - For a custom location:

     ```
     cmake --install . --prefix /path/to/location
     ```

   - For a system location (requires privileges):

     ```
     sudo cmake --install .
     ```

     Or

     ```
     sudo cmake --install . --prefix /path/to/privileged/location
     ```

Note: Armadillo and SuperLU will be automatically installed in the build directory during the build process.

2.5 Testing

Run from the `build` directory:

2.5.1 C++

A suite of four automatic tests that verify MOLE's installation and dependencies. These tests run automatically during the C++ library construction.

```
make run_tests
```

2.5.2 MATLAB/Octave

MATLAB/Octave equivalent of the C++ test suite. We recommend running these tests before using MOLE to ensure proper setup.

```
make run_matlab_octave_tests
```

2.6 Examples

Many of the examples require 'gnuplot' to visualize the results. You can get gnuplot on macOSX with

```
brew install gnuplot
```

and on Windows downlaoding and running the file from here

2.6.1 C++

Four self-contained, well-documented examples demonstrating typical PDE solutions. These are automatically built with `make` and serve as an excellent starting point for C++ users.

2.6.2 MATLAB/Octave Examples

A collection of over 30 examples showcasing various PDE solutions, from simple linear one-dimensional problems to complex nonlinear multidimensional scenarios.

2.7 Quick Start Examples

Here are some simple examples to help you get started with MOLE:

C++

```cpp
// transport1D.cpp - 1D advection-reaction-dispersion equation

#include "mole.h"
#include <iostream>

int main() {
  int k = 2;              // Operators' order of accuracy
  Real a = 0;             // Left boundary
  Real b = 130;           // Right boundary
  int m = 26;             // Number of cells
  Real dx = (b - a) / m; // Cell's width [m]

  // Get 1D mimetic operators
  Gradient G(k, m, dx);
```

(continues on next page)

(continued from previous page)

```
  Divergence D(k, m, dx);
  Interpol I(m, 0.5);

  // Allocate fields
  vec C(m + 2); // Scalar field (concentrations)
  vec V(m + 1); // Vector field (velocities)

  // Time integration loop (simplified)
  for (int i = 0; i <= iter; i++) {
    // First-order forward-time scheme
    C += dt * (D * (dis * (G * C)) - D * (V % (I * C)));
  }

  cout << C;
  return 0;
}
```

MATLAB/Octave

```
% elliptic1D.m - 1D Poisson's equation with Robin boundary conditions

addpath('../../src/matlab_octave')

west = 0;  % Domain's limits
east = 1;

k = 6;     % Operator's order of accuracy
m = 2*k+1; % Minimum number of cells for desired accuracy
dx = (east-west)/m;  % Step length

L = lap(k, m, dx);  % 1D Mimetic laplacian operator

% Impose Robin BC on laplacian operator
a = 1;
b = 1;
L = L + robinBC(k, m, dx, a, b);

% 1D Staggered grid
grid = [west west+dx/2 : dx : east-dx/2 east];

% RHS
U = exp(grid)';
U(1) = 0;      % West BC
U(end) = 2*exp(1);  % East BC

U = L\U;  % Solve a linear system of equations

% Plot result
plot(grid, U, 'o')
hold on
plot(grid, exp(grid))
```

For full examples, see:

- C++: transport1D.cpp
- MATLAB: elliptic1D.m

2.8 Next Steps

- Check out more C++ examples in the examples/cpp/ directory
- Explore the MATLAB/Octave examples in the examples/matlab_octave/ directory
- Join our community and contribute

Performing non-unary operations involving operands constructed over different grids may lead to unexpected results. While MOLE allows such operations without throwing errors, users must exercise caution when manipulating operators across different grids.

2.9 Licensing

MOLE is distributed under a GNU General Public License; please refer to the *LICENSE* file for more details.

CHAPTER

THREE

MOLE DOCUMENTATION GUIDE

This is the definitive guide for building and maintaining the documentation for the MOLE.

3.1 Documentation Structure

The MOLE documentation consists of two main components:

1. **API Documentation** (Doxygen)
   - C++ API reference
   - MATLAB/Octave API reference
   - Implementation details
   - Code documentation
2. **User Manual** (Sphinx)
   - Tutorials
   - Examples
   - Usage guides
   - Theory background

3.2 Building Documentation

3.2.1 Prerequisites

Before building the documentation, ensure you have:

1. **Doxygen** installed on your system (for API documentation)

```
# Ubuntu/Debian
sudo apt install doxygen

# macOS with Homebrew
brew install doxygen

# RHEL/CentOS/Fedora
sudo dnf install doxygen
```

2. **Graphviz** installed for generating diagrams

```
# Ubuntu/Debian
sudo apt install graphviz

# macOS with Homebrew
brew install graphviz

# RHEL/CentOS/Fedora
sudo dnf install graphviz
```

3. **Inkscape** installed for high-quality SVG to PDF conversion (required for PDF output)

```
# Ubuntu/Debian
sudo apt install inkscape

# macOS with Homebrew
brew install inkscape

# RHEL/CentOS/Fedora
sudo dnf install inkscape
```

4. **LaTeX** (required for PDF generation):

```
# Ubuntu/Debian
sudo apt install texlive-latex-base texlive-fonts-recommended texlive-latex-extra

# RHEL/CentOS/Fedora
sudo dnf install texlive-scheme-medium

# macOS
brew install --cask mactex
```

5. **Python dependencies**: First, ensure Python 3 and pip are installed:

```
# Ubuntu/Debian
sudo apt install python3 python3-pip python3-venv

# RHEL/CentOS/Fedora
sudo dnf install python3 python3-pip python3-virtualenv
```

Then set up the Python environment:

```
# Navigate to the documentation directory
cd doc/sphinx

# Create and activate a virtual environment (Required)
python3 -m venv .venv
source .venv/bin/activate  # On Unix/MacOS
# or
.venv\Scripts\activate     # On Windows

# Install required dependencies
# Either use:
make doc-deps
```

(continues on next page)

(continued from previous page)

```
# Or install directly:
python3 -m pip install -r requirements.txt
```

Some additional packages have been added to enhance documentation features:

- `sphinx-togglebutton`: For collapsible content
- `sphinx-fontawesome`: For FontAwesome icons
- `sphinxcontrib-mermaid`: For Mermaid.js diagrams
- `sphinxext-altair`: For Altair chart integration
- `Pillow`: For image processing including dark mode logo

3.2.2 Building Steps

All commands should be run from the `doc/sphinx` directory:

```
# Generate API Documentation
make doc-doxygen

# Build HTML documentation
make doc-html

# PDF output (requires LaTeX and Inkscape)
make doc-latexpdf

# Clean build files (Doxygen + Sphinx)
make doc-clean

# Build all documentation
make doc-all
```

The documentation will be generated in:

- HTML: `doc/sphinx/build/html/`
- PDF: `doc/sphinx/build/latex/MOLE-docs.pdf` and at the project root (`MOLE-docs.pdf`)
- API Docs: `doc/doxygen/`

3.2.3 PDF Generation Process

The PDF generation process does the following:

1. Runs Sphinx to generate LaTeX files
2. Converts SVG figures to high-quality PDF using Inkscape (2400 DPI)
3. Applies special fixes to LaTeX files for proper math rendering
4. Compiles the LaTeX document into a PDF
5. Copies the final PDF to the project root directory

This process is handled by the `build.sh` script, which is called by the `doc-latexpdf` make target.

3.2.4 Image Handling

Images are automatically handled when building the documentation using the Makefile targets:

The image handling process:

- Copies images from `doc/assets/img/` to `doc/sphinx/build/html/_static/img/`
- Fixes image paths in the HTML output
- For PDF output, converts SVG files to high-quality PDFs (using Inkscape)

If you're running Sphinx directly without the Makefile, you'll need to run the image copy script separately:

```
# Run after building documentation manually
./copy_images.sh
```

3.3 Development Workflow

When contributing to documentation:

1. **API Documentation**
   - Add C++ documentation in source code using Doxygen syntax
   - Build with `make doc-doxygen` to verify
2. **User Manual**
   - Edit `.rst` or `.md` files in `doc/sphinx/source/`
   - Build with `make doc-html` to preview changes
   - Use `make doc-clean` to force full rebuild

3.4 Documentation Standards

- Use clear, concise language
- Include working code examples
- Follow Sphinx reST syntax
- Follow Doxygen conventions for API docs

CHAPTER

FOUR

C++

This section documents the C++ implementation of the MOLE toolkit.

4.1 Overview

The C++ implementation provides high-performance operators and boundary conditions for numerical computations, with a focus on mimetic methods.

4.2 Class Structure

The MOLE C++ API consists of several key components:

Base Classes:

- **sp_mat**: Armadillo sparse matrix base class
- **Operator**: Abstract base class for all operators

Core Operators:

- **Interpol**: Interpolation operator for grid transformations
- **Gradient**: Gradient operator with support for various grid types
- **Divergence**: Divergence operator with support for various grid types
- **Laplacian**: Laplacian operator composed of Gradient and Divergence

Boundary Conditions:

- **RobinBC**: Implements Robin, Neumann, and Dirichlet boundary conditions
- **MixedBC**: Implements mixed-type boundary conditions

Utility Components:

- **Utils**: Core utility functions for matrix operations and grid handling
- **Grid Functions**: Functions for grid generation and manipulation
- **Weight Functions**: Functions for computing operator weights

Key Features:

- All operators inherit from the abstract Operator class
- Boundary conditions can modify operator behavior
- Utility functions support both operators and boundary conditions
- Grid and weight functions configure operator behavior

- Laplacian implementation uses both Gradient and Divergence

4.2.1 Operators

MOLE provides a comprehensive set of mimetic operators for numerical computations. These operators are designed to preserve important mathematical properties of the continuous operators they approximate.

Gradient Operator

The Gradient operator computes the gradient of a scalar field in the MOLE library.

API Reference

class **Gradient** : public sp_mat

Mimetic *Gradient* operator.

Public Functions

Gradient(u16 k, u32 m, Real dx)

1-D Mimetic *Gradient* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells
- **dx** – Spacing between cells

Gradient(u16 k, u32 m, u32 n, Real dx, Real dy)

2-D Mimetic *Gradient* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction

Gradient(u16 k, u32 m, u32 n, u32 o, Real dx, Real dy, Real dz)

3-D Mimetic *Gradient* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **o** – Number of cells in z-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction
- **dz** – Spacing between cells in z-direction

vec **getP**()

Returns the weights used in the Mimeitc *Gradient* Operators.

> **Note**
>
> for informational purposes only, can be used in constructing new operators.

Divergence Operator

The Divergence operator computes the divergence of a vector field in the MOLE library.

API Reference

class **Divergence** : public sp_mat

Mimetic *Divergence* operator.

Public Functions

Divergence(u16 k, u32 m, Real dx)

1-D Mimetic *Divergence* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells
- **dx** – Spacing between cells

Divergence(u16 k, u32 m, u32 n, Real dx, Real dy)

2-D Mimetic *Divergence* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction

Divergence(u16 k, u32 m, u32 n, u32 o, Real dx, Real dy, Real dz)

3-D Mimetic *Divergence* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **o** – Number of cells in z-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction
- **dz** – Spacing between cells in z-direction

vec **getQ**()

Returns the weights used in the Mimeitc *Divergence* Operators.

> **Note**
>
> for informational purposes only, can be used in constructing new operators.

Laplacian Operator

The Laplacian operator computes the Laplacian of a scalar field in the MOLE library.

API Reference

class **Laplacian** : public sp_mat

Mimetic *Laplacian* operator.

Public Functions

Laplacian(u16 k, u32 m, Real dx)

1-D Mimetic *Laplacian* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells
- **dx** – Spacing between cells

Laplacian(u16 k, u32 m, u32 n, Real dx, Real dy)

2-D Mimetic *Laplacian* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction

Laplacian(u16 k, u32 m, u32 n, u32 o, Real dx, Real dy, Real dz)

3-D Mimetic *Laplacian* Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **o** – Number of cells in z-direction
- **dx** – Spacing between cells in x-direction
- **dy** – Spacing between cells in y-direction
- **dz** – Spacing between cells in z-direction

Interpolation Operator

The Interpol class performs interpolation operations in the MOLE library.

API Reference

class **Interpol** : public sp_mat

Mimetic Interpolator operator.

Public Functions

Interpol(u32 m, Real c)

1-D Mimetic Interpolator Constructor

Parameters

- **m** – Number of cells
- **c** – Weight for ends, can be any value from 0.0<=c<=1.0

Interpol(u32 m, u32 n, Real c1, Real c2)

2-D Mimetic Interpolator Constructor

Parameters

- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **c1** – Weight for ends in x-direction, can be any value from 0.0<=c<=1.0
- **c2** – Weight for ends in y-direction, can be any value from 0.0<=c<=1.0

Interpol(u32 m, u32 n, u32 o, Real c1, Real c2, Real c3)

3-D Mimetic Interpolator Constructor

Parameters

- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **o** – Number of cells in z-direction
- **c1** – Weight for ends in x-direction, can be any value from 0.0<=c<=1.0
- **c2** – Weight for ends in y-direction, can be any value from 0.0<=c<=1.0
- **c3** – Weight for ends in z-direction, can be any value from 0.0<=c<=1.0

Interpol(bool type, u32 m, Real c)

1-D Mimetic Interpolator Constructor

Parameters

- **type** – Dummy holder to trigger overloaded function
- **m** – Number of cells
- **c** – Weight for ends, can be any value from 0.0<=c<=1.0

Interpol(bool type, u32 m, u32 n, Real c1, Real c2)

2-D Mimetic Interpolator Constructor

Parameters

- **type** – Dummy holder to trigger overloaded function
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **c1** – Weight for ends in x-direction, can be any value from 0.0<=c<=1.0
- **c2** – Weight for ends in y-direction, can be any value from 0.0<=c<=1.0

Interpol(bool type, u32 m, u32 n, u32 o, Real c1, Real c2, Real c3)

3-D Mimetic Interpolator Constructor

Parameters

- **type** – Dummy holder to trigger overloaded function
- **m** – Number of cells in x-direction
- **n** – Number of cells in y-direction
- **o** – Number of cells in z-direction
- **c1** – Weight for ends in x-direction, can be any value from 0.0<=c<=1.0
- **c2** – Weight for ends in y-direction, can be any value from 0.0<=c<=1.0
- **c3** – Weight for ends in z-direction, can be any value from 0.0<=c<=1.0

Usage Examples

Transport Example (Gradient & Divergence)

Listing 1: Transport 1D Example using Gradient and Divergence (examples/cpp/transport1D.cpp)

```cpp
/**
 * This example uses MOLE to solve the 1D advection-reaction-dispersion
 * equation:
 * https://wwwbrr.cr.usgs.gov/projects/GWC_coupled/phreeqc/html/final-22.html
 */

#include "mole.h"
#include <iostream>

int main() {

  int k = 2;             // Operators' order of accuracy
  Real a = 0;            // Left boundary
  Real b = 130;          // Right boundary
  int m = 26;            // Number of cells
  Real dx = (b - a) / m; // Cell's width [m]
  Real t = 4;            // Simulation time [years]
  int iter = 208;        // Number of iterations
  Real dt = t / iter;    // Time step
  Real dis = 5;          // Dispersivity [m]
```

(continues on next page)

(continued from previous page)

```
  Real vel = 15;          // Pore-water flow velocity [m/year]
  Real R = 2.5;           // Retardation (Cl^- = 1, Na^+ = 2.5)
  Real C0 = 1;            // Displacing solution concentration [mmol/kgw]

  // Get 1D mimetic operators
  Gradient G(k, m, dx);
  Divergence D(k, m, dx);
  Interpol I(m, 0.5);

  // Allocate fields
  vec C(m + 2); // Scalar field (concentrations)
  vec V(m + 1); // Vector field (velocities)

  // Impose initial conditions
  C(0) = C0;
  V.fill(vel);

  // Hydrodynamic dispersion coefficient [m^2/year]
  dis *= vel; // 75

  // dt = dt/R (retardation)
  dt /= R;

  // Time integration loop
  for (int i = 0; i <= iter; i++) {

    // First-order forward-time scheme
    C += dt * (D * (dis * (G * C)) - D * (V % (I * C)));

    // Right boundary condition (reflection)
    C(m + 1) = C(m);
  }

  // Spit out the new concentrations!
  cout << C;

  return 0;
}
```

Elliptic Example (Laplacian)

Listing 2: Elliptic 2D Example using Laplacian (examples/cpp/elliptic2D.cpp)

```
/**
 * This example uses MOLE to solve a 2D BVP
 * It is the equivalent to examples_MATLAB/minimal_poisson2D.m
 * The output can be plotted via:
 * gnuplot> plot 'solution' matrix with image
 */

```

(continues on next page)

(continued from previous page)

```cpp
#include "mole.h"
#include <iostream>

int main() {
  int k = 2; // Operators' order of accuracy
  int m = 9; // Vertical resolution
  int n = 9; // Horizontal resolution

  // Get mimetic operators
  Laplacian L(k, m, n, 1, 1);
  RobinBC BC(k, m, 1, n, 1, 1, 0); // Dirichlet BC
  L = L + BC;

  // Build RHS for system of equations
  mat rhs(m + 2, n + 2, fill::zeros);
  rhs.row(0) = 100 * ones(1, n + 2); // Known value at the bottom boundary

  // Solve the system of linear equations
#ifdef EIGEN
  // Use Eigen only if SuperLU (faster) is not available
  vec sol = Utils::spsolve_eigen(L, vectorise(rhs));
#else
  vec sol = spsolve(L, vectorise(rhs)); // Will use SuperLU
#endif

  // Print out the solution
  cout << reshape(sol, m + 2, n + 2);

  return 0;
}
```

Schrödinger Example (Complex Operators)

Listing 3: Schrödinger 1D Example (examples/cpp/schrodinger1D.cpp)

```cpp
/**
 * This example uses MOLE to solve the 1D Schrodinger equation
 */

#include "mole.h"
#include <algorithm>
#include <iostream>

int main() {

  int k = 4;   // Operators' order of accuracy
  Real a = -5; // Left boundary
  Real b = 5;  // Right boundary
  int m = 500; // Number of cells
  vec grid = linspace(a, b, m);
  Real dx = grid(1) - grid(0); // Step size
```

(continues on next page)

(continued from previous page)

```
  // Get mimetic Laplacian operator
  Laplacian L(k, m - 2, dx);

  std::transform(grid.begin(), grid.end(), grid.begin(),
                 [](Real x) { return x * x; });

  sp_mat V(m, m); // Potential energy operator
  V.diag(0) = grid;

  // Hamiltonian
  sp_mat H = -0.5 * (sp_mat)L + V;

  cx_vec eigval;
  eig_gen(eigval, (mat)H); // Compute eigenvalues

  eigval = sort(eigval);

  cout << "Energy levels = [ ";
  for (int i = 0; i < 4; ++i)
    cout << real(eigval(i) / eigval(0)) << ' ';
  cout << "]\n";

  return 0;
}
```

Notes and Considerations

4.2.2 Boundary Conditions

MOLE supports a variety of boundary conditions for solving PDEs with different physical constraints.

Mixed Boundary Conditions

The MixedBC class implements mixed boundary conditions in the MOLE library.

API Reference

class **MixedBC** : public sp_mat

Mimetic Mixed Boundary Condition operator.

Public Functions

MixedBC(u16 k, u32 m, Real dx, const std::string &left, const std::vector<Real> &coeffs_left, const std::string &right, const std::vector<Real> &coeffs_right)

1-D Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells
- **dx** – Spacing between cells

- **left** – Type of boundary condition at the left boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_left** – Coefficients for the left boundary condition
- **right** – Type of boundary condition at the right boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_right** – Coefficients for the right boundary condition

MixedBC(u16 k, u32 m, Real dx, u32 n, Real dy, const std::string &left, const std::vector<Real> &coeffs_left, const std::string &right, const std::vector<Real> &coeffs_right, const std::string &bottom, const std::vector<Real> &coeffs_bottom, const std::string &top, const std::vector<Real> &coeffs_top)

2-D Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells along x-axis
- **dx** – Spacing between cells along x-axis
- **n** – Number of cells along y-axis
- **dy** – Spacing between cells along y-axis
- **left** – Type of boundary condition at the left boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_left** – Coefficients for the left boundary condition
- **right** – Type of boundary condition at the right boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_right** – Coefficients for the right boundary condition
- **bottom** – Type of boundary condition at the bottom boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_bottom** – Coefficients for the bottom boundary condition
- **top** – Type of boundary condition at the top boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_top** – Coefficients for the top boundary condition

MixedBC(u16 k, u32 m, Real dx, u32 n, Real dy, u32 o, Real dz, const std::string &left, const std::vector<Real> &coeffs_left, const std::string &right, const std::vector<Real> &coeffs_right, const std::string &bottom, const std::vector<Real> &coeffs_bottom, const std::string &top, const std::vector<Real> &coeffs_top, const std::string &front, const std::vector<Real> &coeffs_front, const std::string &back, const std::vector<Real> &coeffs_back)

3-D Constructor

Parameters

- **k** – Order of accuracy
- **m** – Number of cells along x-axis
- **dx** – Spacing between cells along x-axis
- **n** – Number of cells along y-axis
- **dy** – Spacing between cells along y-axis
- **o** – Number of cells along z-axis
- **dz** – Spacing between cells along z-axis

- **left** – Type of boundary condition at the left boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_left** – Coefficients for the left boundary condition
- **right** – Type of boundary condition at the right boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_right** – Coefficients for the right boundary condition
- **bottom** – Type of boundary condition at the bottom boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_bottom** – Coefficients for the bottom boundary condition
- **top** – Type of boundary condition at the top boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_top** – Coefficients for the top boundary condition
- **front** – Type of boundary condition at the front boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_front** – Coefficients for the front boundary condition
- **back** – Type of boundary condition at the back boundary ('Dirichlet', 'Neumann', 'Robin')
- **coeffs_back** – Coefficients for the back boundary condition

Robin Boundary Conditions

The RobinBC class implements Robin boundary conditions in the MOLE library.

API Reference

class **RobinBC** : public sp_mat

Mimetic Robin Boundary Condition operator.

Public Functions

RobinBC(u16 k, u32 m, Real dx, Real a, Real b)

1-D Robin boundary constructor

Parameters

- **k** – mimetic order of accuracy
- **m** – number of cells in x-dimension
- **dx** – cell width in x-direction
- **a** – Coefficient of the Dirichlet function
- **b** – Coefficient of the Neumann function

RobinBC(u16 k, u32 m, Real dx, u32 n, Real dy, Real a, Real b)

2-D Robin boundary constructor

> **Note**
>
> Uses 1-D Robin to build the 2-D operator

Parameters

- **k** – mimetic order of accuracy
- **m** – number of cells in x-dimension
- **dx** – cell width in x-direction
- **n** – number of cells in y-dimension
- **dy** – cell width in y-direction
- **a** – Coefficient of the Dirichlet function
- **b** – Coefficient of the Neumann function

RobinBC(u16 k, u32 m, Real dx, u32 n, Real dy, u32 o, Real dz, Real a, Real b)

3-D Robin boundary constructor

Note

Uses 1-D Robin to build the 3-D operator

Parameters

- **k** – mimetic order of accuracy
- **m** – number of cells in x-dimension
- **dx** – cell width in x-direction
- **n** – number of cells in y-dimension
- **dy** – cell width in y-direction
- **o** – number of cells in z-dimension
- **dz** – cell width in z-direction
- **a** – Coefficient of the Dirichlet function
- **b** – Coefficient of the Neumann function

Usage Examples

Elliptic Problem with Mixed Boundary Conditions

Listing 4: Elliptic 1D Example with Mixed Boundary Conditions (examples/cpp/elliptic1D.cpp)

```cpp
/**
 * This example uses MOLE to solve a 1D BVP
 */

#include "mole.h"
#include <iostream>

int main() {

  int k = 6;              // Operators' order of accuracy
  Real a = 0;             // Left boundary
  Real b = 1;             // Right boundary
```

(continues on next page)

(continued from previous page)

```
  int m = 2 * k + 1;      // Number of cells
  Real dx = (b - a) / m; // Step size

  // Get mimetic operators
  Laplacian L(k, m, dx);
  Real d = 1; // Dirichlet coefficient
  Real n = 1; // Neumann coefficient
  RobinBC BC(k, m, dx, d, n);
  L = L + BC;

  // 1D Staggered grid
  vec grid(m + 2);
  grid(0) = a;
  grid(1) = a + dx / 2.0;
  int i;
  for (i = 2; i <= m; i++) {
    grid(i) = grid(i - 1) + dx;
  }
  grid(i) = b;

  // Build RHS for system of equations
  vec rhs(m + 2);
  rhs = exp(grid); // rhs(0) = 1
  rhs(0) = 0;
  rhs(m + 1) = 2 * exp(1); // rhs(1) = 2e

  // Solve the system of linear equations
#ifdef EIGEN
  // Use Eigen only if SuperLU (faster) is not available
  vec sol = Utils::spsolve_eigen(L, rhs);
#else
  vec sol = spsolve(L, rhs); // Will use SuperLU
#endif

  // Print out the solution
  cout << sol;

  return 0;
}
```

Complex Problem with Advanced Boundary Handling

Listing 5: 3D Convection-Diffusion Example with Complex Boundary Conditions (examples/cpp/convection_diffusion3D.cpp)

```
/**
 * @file convection_diffusion3D.cpp
 * @brief Solves a 3D Convection-Diffusion equation using Mimetic Operators for Linear␣
↪Equations (MOLE).
 *
 * This program numerically solves the 3D convection-diffusion equation using MOLE␣
```

(continues on next page)

(continued from previous page)

```
techniques.
 * It simulates the distribution of CO2 concentration over time within a porous medium,
 * considering both diffusion and advection effects.
 *
 * If OUTPUT_FRAME_DATA is set to 1, slices (2D cross-sections) of the concentration field are extracted
 * and saved to a text file for visualization.
 */

#include <iostream>
#include <fstream>
#include <mole.h>

#define OUTPUT_FRAME_DATA 0

// Helper for linear indexing in 3D arrays (0-based indexing)
inline size_t idx3D(size_t i, size_t j, size_t k, size_t m, size_t n, size_t o) {
    return i + (m+2)*j + (m+2)*(n+2)*k;
}

int main() {
    // Parameters
    unsigned short k = 2;
    unsigned int m = 101;
    unsigned int n = 51;
    unsigned int o = 101;

    double a = 0.0, b = 101.0;
    double c = 0.0, d = 51.0;
    double e = 0.0, f = 101.0;

    double dx = (b - a) / m;
    double dy = (d - c) / n;
    double dz = (f - e) / o;

    // Construct operators
    sp_mat D = Divergence(k, m, n, o, dx, dy, dz);
    sp_mat G = Gradient(k, m, n, o, dx, dy, dz);
    sp_mat I = Interpol(m, n, o, 1, 1, 1);

    size_t scalarSize = (m+2)*(n+2)*(o+2);
    size_t vectorSize = G.n_rows;

    // Allocate fields
    std::vector<double> V(vectorSize, 0.0);
    vec C(scalarSize, fill::zeros);

    // Initial conditions
    int bottom = 10;
    int top = 15;
    int seal = 40;
    int seal_idx = seal - 1;
```

(continues on next page)

(continued from previous page)

```
    int seal5_idx = (seal + 5) - 1;

    // Construct the velocity field
    size_t yCount = m*(n+1)*o;
    std::vector<double> y(yCount, 1.0);

    // Apply shale conditions on velocity field
    for (int i_ = 0; i_ < (int)m; i_++) {
        for (int k_ = 0; k_ < (int)o; k_++) {
            y[i_ + m*seal_idx + m*(n+1)*k_] = 0.0;
            y[i_ + m*seal5_idx + m*(n+1)*k_] = 0.0;
        }
    }

    // Assign y into V at the correct offset
    size_t offset = (m+1)*n*o;
    for (size_t i_ = 0; i_ < yCount; ++i_) {
        if (offset + i_ < V.size()) {
            V[offset + i_] = y[i_];
        }
    }

    // Set initial density
    int mid_x = (int)std::ceil((m+2)/2.0) - 1;
    int mid_z = (int)std::ceil((o+2)/2.0) - 1;
    for (int j = bottom - 1; j <= top - 1; j++) {
        size_t idx = idx3D(mid_x, j, mid_z, m, n, o);
        C(idx) = 1.0;
    }

    // Well indices where C=1
    std::vector<size_t> wellIndices;
    for (size_t i_ = 0; i_ < scalarSize; ++i_) {
        if (C(i_) == 1.0) {
            wellIndices.push_back(i_);
        }
    }

    // Diffusivity and porosity
    double diff = 1.0;
    double porosity = 1.0;
    diff *= porosity;

    // Build K
    std::vector<double> K(vectorSize, diff);
    std::vector<double> kk(yCount, diff);
    for (int i_ = 0; i_ < (int)m; i_++) {
        for (int k_ = 0; k_ < (int)o; k_++) {
            kk[i_ + m*seal_idx + m*(n+1)*k_] = diff/10.0;
            kk[i_ + m*seal5_idx + m*(n+1)*k_] = diff/40.0;
        }
    }
```

(continues on next page)

(continued from previous page)

```
    for (size_t i_ = 0; i_ < yCount; ++i_) {
        if (offset + i_ < K.size()) {
            K[offset + i_] = kk[i_];
        }
    }

    // Time step calculation
    double dt1 = dx*dx/(3*diff)/3.0;
    double maxV = 0.0;
    for (auto val : V) {
        if (val > maxV) maxV = val;
    }
    double dt2 = (maxV > 0.0) ? (dx/maxV)/3.0 : 1e-3;
    double dt = std::min(dt1, dt2);
    int iters = 120;

    // Convert V and K to arma::vec
    arma::vec K_arma(K);
    arma::vec V_arma(V);

    arma::ivec offsets_vec(1);
    offsets_vec(0) = 0;

    sp_mat Kdiag = spdiags(K_arma, offsets_vec, K_arma.n_elem, K_arma.n_elem);
    sp_mat Vdiag = spdiags(V_arma, offsets_vec, V_arma.n_elem, V_arma.n_elem);

    SizeMat size_identity(D.n_rows, D.n_rows);
    sp_mat I_sp = speye(size_identity);

    // Operators: L and Dadv
    sp_mat L = dt * D * Kdiag * G + I_sp;
    sp_mat Dadv = dt * D * Vdiag * I;

    #if OUTPUT_FRAME_DATA
    // Open a single file to store selected frames
    std::ofstream frameFile("frames.txt");
    if(!frameFile) {
        std::cerr << "Error opening frames.txt for writing.\n";
        return 1;
    }
    #endif

    // Time-stepping loop
    for (int i_ = 1; i_ <= iters*3; ++i_) {
        // Diffusion step
        vec Cnew = L * C;
        for (auto w : wellIndices) {
            Cnew(w) = 1.0;
        }
        C = Cnew;

        // Advection step
```

(continues on next page)

(continued from previous page)

```
        vec Cadv = Dadv * C;
        Cadv = C - Cadv;
        for (auto w : wellIndices) {
            Cadv(w) = 1.0;
        }
        C = Cadv;

        #if OUTPUT_FRAME_DATA
        // Write only selected frames to a single file
        frameFile << "FRAME " << i_ << "\n";
        for (int j = 0; j < (int)(n+2); j++) {
            for (int k_ = 0; k_ < (int)(o+2); k_++) {
                size_t idx = idx3D(seal_idx, j, k_, m, n, o);
                frameFile << C(idx);
                if (k_ < (int)(o+1)) frameFile << " ";
            }
            frameFile << "\n";
        }
        frameFile << "\n"; // Blank line between frames
        #endif
    }

    #if OUTPUT_FRAME_DATA
    frameFile.close();
    #endif

    // Display minimum and maximum concentration values
    std::cout << "Minimum CO2 concentration: " << C.min() << "\n";
    std::cout << "Maximum CO2 concentration: " << C.max() << "\n";

    return 0;
}
```

Importance of Boundary Conditions

Boundary conditions are critical for ensuring that differential equation solutions are unique and physically meaningful. They specify constraints at the boundaries of the computational domain.

4.2.3 Utilities

MOLE provides a set of utility functions and classes to simplify common tasks when working with mimetic operators and boundary conditions.

API Reference

class **Utils**

Utility Functions.

Public Functions

void **meshgrid**(const vec &x, const vec &y, mat &X, mat &Y)

An analog to the MATLAB/Octave 2D meshgrid operation.

returns 2-D grid coordinates based on the coordinates contained in vectors x and y. X is a matrix where each row is a copy of x, and Y is a matrix where each column is a copy of y. The grid represented by the coordinates X and Y has length(y) rows and length(x) columns. Key here is the rows is the y-coordinate, and the columns are the x-coordinate.

Parameters

- **x** – a vector of x-indices
- **y** – a vector of y-indices
- **X** – a sparse matrix, will be filled by the function
- **Y** – a sparse matrix, will be filled by the function

void **meshgrid**(const vec &x, const vec &y, const vec &z, cube &X, cube &Y, cube &Z)

An analog to the MATLAB/Octave 3D meshgrid operation.

meshgrid(x,y,z,X,Y,Z) returns 3-D grid coordinates defined by the vectors x, y, and z. The grid represented by X, Y, and Z has size length(y)-by-length(x)-by-length(z).

Parameters

- **x** – a vector of x-indices
- **y** – a vector of y-indices
- **z** – a vector of z-indices
- **X** – a sparse matrix, will be filled by the function
- **Y** – a sparse matrix, will be filled by the function
- **Z** – a sparse matrix, will be filled by the function

Public Static Functions

static sp_mat **spkron**(const sp_mat &A, const sp_mat &B)

A wrappper for implementing a sparse Kroenecker product.

Note

This is available in Armadillo >8.0

Parameters

- **A** – a sparse matrix
- **B** – a sparse matrix

static sp_mat **spjoin_rows**(const sp_mat &A, const sp_mat &B)

An in place operation for joining two matrices by rows.

Note

This is available in Armadillo >8.0

Parameters

- **A** – a sparse matrix
- **B** – a sparse matrix

static sp_mat **spjoin_cols**(const sp_mat &A, const sp_mat &B)

An in place operation for joining two matrices by columns.

> **Note**
>
> This is available in Armadillo >=8.5

Parameters

- **A** – a sparse matrix
- **B** – a sparse matrix

static vec **spsolve_eigen**(const sp_mat &A, const vec &b)

A wrappper for implementing a sparse solve using Eigen from SuperLU.

> **Note**
>
> This function requires the EIGEN to be used when Armadillo is built

Parameters

- **A** – a sparse matrix LHS of Ax=b
- **b** – a vector for the RHS of Ax=b

static double **trapz**(const vec &x, const vec &y)

Implements the trapezoidal rule for 1D numerical integration.

Computes the area under a curve defined by vectors x and y using: A 0.5 * (x_1 - x) * (y + y_1)

Parameters

- **x** – Vector of x-coordinates
- **y** – Vector of y-values at corresponding x

Returns

Estimated area under the curve

Usage Examples

Here's an example using utility functions in a parabolic equation:

Listing 6: Parabolic 1D Example (examples/cpp/parabolic1D.cpp)

```
#include <iostream>
#include <math.h>
#include "mole.h"
/**
 * This example uses MOLE to solve the heat equation u_t-alpha*u_xx=0[with alpha=1] over␣
```

(continues on next page)

(continued from previous page)

```
↪[0,1]^2,
 * u(x,0)=0 for x in (-1,1), u(1,t)=u(-1,t)=100 for t in [0,1].
 */
int main() {
    int k=2;// Operators' order of accuracy
    double t0=0; //initial time
    double tf=1; //final time
    double a=0; //left boundary
    double b=1; //right boundary
    int m=2*k+1; //num of cells
    double dx=(b-a)/m;
    double dt=tf/(ceil((3*tf)/(dx*dx))); //Von Neumann stability criterion for explicit␣
↪scheme, if k > 2 then dt/dx^2<0.5.
    /*Note that unlike the matlab example, dt isn't dx^2/3 because if a and b are changed␣
↪or the final time is changed
    *the final time tf may not be a multiple of dt. This value of dt guarantees that the␣
↪final time will be a mutiple of dt.
    * while still satisfying Von Neumann stablity criterion
    *Note that in this example dt=dx^2/3 like it is in the matlab example.
    */
    Laplacian L(k,m,dx);
    vec solution(m + 2);
    solution(0)=100;
    solution(m+1)=100;
    vec k1(m+2);
    double t=t0;
    while (t <= tf) { //time integration with euler method.
        k1=L*(solution);
        solution=solution+dt*k1;
        t=t+dt;
    }
    std::cout << solution;


    return 0;
}
```

Advanced Usage

For advanced usage patterns and performance optimization, check out the individual class documentation pages.

CHAPTER

FIVE

MATLAB/OCTAVE

This section documents the MATLAB/Octave implementation of the MOLE toolkit. The documentation is generated automatically from the docstrings in the MATLAB/Octave source files.

5.1 Function Categories

The MOLE MATLAB/Octave API consists of several main categories:

- **Differential Operators**: Core operators for gradient, divergence, curl, and Laplacian calculations
- **Interpolation Operators**: Functions for interpolating values between different grid locations
- **Boundary Conditions**: Operators for handling various boundary conditions (Dirichlet, Neumann, Robin, Mixed)
- **Grid Transformation**: Functions for grid generation and coordinate transformations
- **Weight Functions**: Functions for computing weights used in numerical schemes
- **Mimetic Operators**: Specialized operators preserving mathematical properties

The toolkit provides implementations across different dimensions (1D, 2D, 3D) with support for:

- Non-uniform grids
- Curvilinear coordinates
- Various boundary conditions
- Different interpolation schemes
- Mimetic discretization

5.2 MATLAB/Octave Function Index

This page provides an index of all MATLAB/Octave functions in the MOLE library.

5.2.1 Gradient Operators

- *grad()* - Returns a m+1 by m+2 one-dimensional mimetic gradient operator
- *grad2D()* - Returns a two-dimensional mimetic gradient operator
- *grad2DCurv()* - Returns a 2D curvilinear mimetic gradient
- *grad2DNonUniform()* - Returns a two-dimensional non-uniform mimetic gradient
- *grad3D()* - Returns a three-dimensional mimetic gradient operator

- *grad3DCurv()* - Returns a 3D curvilinear mimetic gradient
- *grad3DNonUniform()* - Returns a three-dimensional non-uniform mimetic gradient
- *gradNonUniform()* - Returns a m+1 by m+2 one-dimensional non-uniform mimetic gradient

5.2.2 Divergence Operators

- *div()* - Returns a m+2 by m+1 one-dimensional mimetic divergence operator
- *div2D()* - Returns a two-dimensional mimetic divergence operator
- *div2DCurv()* - Returns a 2D curvilinear mimetic divergence
- *div2DNonUniform()* - Returns a two-dimensional non-uniform mimetic divergence
- *div3D()* - Returns a three-dimensional mimetic divergence operator
- *div3DCurv()* - Returns a 3D curvilinear mimetic divergence
- *div3DNonUniform()* - Returns a three-dimensional non-uniform mimetic divergence
- *divNonUniform()* - Returns a m+2 by m+1 one-dimensional non-uniform mimetic divergence

5.2.3 Laplacian Operators

- *lap()* - Returns a m+2 by m+2 one-dimensional mimetic laplacian operator
- *lap2D()* - Returns a two-dimensional mimetic laplacian operator
- *lap3D()* - Returns a three-dimensional mimetic laplacian operator

5.2.4 Nodal Operators

- *nodal()* - Returns a one-dimensional operator that approximates the first-order
- *nodal2D()* - Returns a two-dimensional operator that approximates the first-order
- *nodal2DCurv()* - Returns a 2D curvilinear nodal operator
- *nodal3D()* - Returns a three-dimensional operator that approximates the first-order
- *nodal3DCurv()* - Returns a 3D curvilinear nodal operator
- *sidedNodal()* - Returns a one-dimensional nodal operator with one-sided stencils

5.2.5 Interpolation Functions

- *interpol()* - Returns a m+1 by m+2 one-dimensional interpolation operator
- *interpol2D()* - Returns a two-dimensional interpolation operator
- *interpol3D()* - Returns a three-dimensional interpolation operator
- *interpolD()* - Returns a m+1 by m+2 one-dimensional interpolation operator
- *interpolD2D()* - Returns a two-dimensional interpolation operator
- *interpolD3D()* - Returns a three-dimensional interpolation operator
- `interpolCentersToFacesD1D()` - Interpolates values from cell centers to faces
- `interpolCentersToFacesD2D()` - Interpolates values from cell centers to faces
- `interpolCentersToFacesD3D()` - Interpolates values from cell centers to faces
- *interpolCentersToNodes1D()* - Interpolates values from cell centers to nodes

- *interpolCentersToNodes2D()* - Interpolates values from cell centers to nodes
- *interpolCentersToNodes3D()* - Interpolates values from cell centers to nodes
- *interpolFacesToCentersG1D()* - Interpolates values from faces to cell centers
- *interpolFacesToCentersG2D()* - Interpolates values from faces to cell centers
- *interpolFacesToCentersG3D()* - Interpolates values from faces to cell centers
- *interpolNodesToCenters1D()* - Interpolates values from nodes to cell centers
- *interpolNodesToCenters2D()* - Interpolates values from nodes to cell centers
- *interpolNodesToCenters3D()* - Interpolates values from nodes to cell centers

5.2.6 Boundary Conditions

- *addScalarBC1D()* - Apply boundary conditions to a 1D system
- *addScalarBC1Dlhs()* - Create left-hand side matrix for 1D boundary conditions
- *addScalarBC1Drhs()* - Create right-hand side vector for 1D boundary conditions
- *addScalarBC2D()* - Apply boundary conditions to a 2D system
- *addScalarBC2Dlhs()* - Create left-hand side matrix for 2D boundary conditions
- *addScalarBC2Drhs()* - Create right-hand side vector for 2D boundary conditions
- *addScalarBC3D()* - Apply boundary conditions to a 3D system
- *addScalarBC3Dlhs()* - Create left-hand side matrix for 3D boundary conditions
- *addScalarBC3Drhs()* - Create right-hand side vector for 3D boundary conditions
- *boundaryIdx2D()* - Get boundary indices for a 2D domain
- *mixedBC()* - Constructs a 1D mimetic mixed boundary conditions operator
- *mixedBC2D()* - Constructs a 2D mimetic mixed boundary conditions operator
- *mixedBC3D()* - Constructs a 3D mimetic mixed boundary conditions operator
- *neumann2DCurv()* - Returns a 2D curvilinear Neumann BC operator
- *neumann3DCurv()* - Returns a 3D curvilinear Neumann BC operator
- *robinBC()* - Returns a m+2 by m+2 one-dimensional mimetic boundary operator that
- *robinBC2D()* - Returns a two-dimensional mimetic boundary operator that implements
- *robinBC3D()* - Returns a three-dimensional mimetic boundary operator that

5.2.7 Grid and Transformation Functions

- *gridGen()* - Generate a grid using transfinite interpolation
- *tfi()* - Transfinite interpolation for grid generation
- *ttm()* - Tensor-product transfinite mapping
- *jacobian2D()* - Calculate the Jacobian matrix for 2D grid transformations
- *jacobian3D()* - Calculate the Jacobian matrix for 3D grid transformations

5.2.8 Utility Functions

- *amean()* - Returns the arithmetic mean for every two pairs in a column vector
- *hmean()* - Returns the harmonic mean for every two pairs in a column vector
- *weightsP()* - Returns the m+1 weights of P
- *weightsP2D()* - Returns the two-dimensional weights of P
- *weightsQ()* - Returns the m+2 weights of Q
- *weightsQ2D()* - Returns the two-dimensional weights of Q
- *rk4()* - Explicit Runge-Kutta 4th-order method
- *curl2D()* - Returns a two-dimensional mimetic curl operator
- *DI2()* - Returns a 2D diagonal scaling matrix
- *DI3()* - Returns a 3D diagonal scaling matrix
- *GI1()* - Returns a 1D geomeric interpretation matrix
- *GI13()* - Returns a 3D geometric interpretation matrix
- *GI2()* - Returns a 2D geometric interpretation matrix
- *mimeticB()* - Returns a m+2 by m+1 one-dimensional mimetic boundary operator
- *tensorGrad2D()* - Calculate tensor gradient in 2D

5.3 MATLAB/Octave API

This page documents the API of the MOLE MATLAB/Octave module. Functions are organized by category.

5.3.1 Differential Operators

Gradient Operators

grad(*k*, *m*, *dx*, *dc*, *nc*)

PURPOSE

Returns a one-dimensional mimetic gradient operator depending on whether

SYNOPSIS

```
function grad
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (2x1 vector for left and right vertices, resp.)
(optional) nc : b0 (2x1 vector for left and right vertices, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
```

CROSS-REFERENCE INFORMATION

This function calls:

`gradPeriodic()` Returns a m by m+2 one-dimensional mimetic gradient operator

`gradNonPeriodic()` Returns a m+1 by m+2 one-dimensional mimetic gradient operator

This function is called by:

gradNonUniform() Returns a m+1 by m+2 one-dimensional non-uniform mimetic gradient

robinBC() Returns a m+2 by m+2 one-dimensional mimetic boundary operator that

mimeticB() Returns a m+2 by m+1 one-dimensional mimetic boundary operator

lap() Returns a one-dimensional mimetic Laplacian operator depending on whether

addScalarBC1Dlhs() This functions uses geometry and boundary type conditions to create

mixedBC() Constructs a 1D mimetic mixed boundary conditions operator

weightsP() Returns the m+1 weights of P

grad3DCurv() Returns a 3D curvilinear mimetic gradient

grad2D(*k*, *m*, *dx*, *n*, *dy*, *dc*, *nc*)

PURPOSE

Returns a two-dimensional mimetic gradient operator depending on whether

SYNOPSIS

```
function grad2D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (4x1 vector for left, right, bottom, top boundaries, resp.)
(optional) nc : b0 (4x1 vector for left, right, bottom, top boundaries, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
               n : Number of cells along y-axis
              dy : Step size along y-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

`gradPeriodic()` Returns a m by m+2 one-dimensional mimetic gradient operator

`gradNonPeriodic2D()` Returns a two-dimensional mimetic gradient operator

`gradNonPeriodic()` Returns a m+1 by m+2 one-dimensional mimetic gradient operator

This function is called by:

lap2D() Returns a two-dimensional mimetic Laplacian operator depending on whether

grad2DCurv() Returns a 2D curvilinear mimetic gradient

grad2DCurv(*k*, *X*, *Y*)

PURPOSE

Returns a 2D curvilinear mimetic gradient

SYNOPSIS

```
function grad2DCurv
```

DESCRIPTION

CROSS-REFERENCE INFORMATION

This function calls:

grad2D() Returns a two-dimensional mimetic gradient operator depending on whether

GI2()

jacobian2D() Returns:

grad2DNonUniform(*k*, *xticks*, *yticks*)

PURPOSE

Returns a two-dimensional non-uniform mimetic gradient operator

SYNOPSIS

```
function grad2DNonUniform
```

DESCRIPTION

```
(including the boundaries!)

               k : Order of accuracy
          xticks : Centers' ticks (x-axis)
          yticks : Centers' ticks (y-axis)
```

CROSS-REFERENCE INFORMATION

This function calls:

gradNonUniform() Returns a m+1 by m+2 one-dimensional non-uniform mimetic gradient

grad3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *dc*, *nc*)

PURPOSE

Returns a three-dimensional mimetic gradient operator depending on whether

SYNOPSIS

```
function grad3D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)
(optional) nc : b0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
               n : Number of cells along y-axis
              dy : Step size along y-axis
               o : Number of cells along z-axis
              dz : Step size along z-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

`gradNonPeriodic3D()` Returns a three-dimensional mimetic gradient operator

`gradPeriodic()` Returns a m by m+2 one-dimensional mimetic gradient operator

`gradNonPeriodic()` Returns a m+1 by m+2 one-dimensional mimetic gradient operator

This function is called by:

`lap3D()` Returns a three-dimensional mimetic Laplacian operator depending on whether

`grad3DCurv()` Returns a 3D curvilinear mimetic gradient

grad3DCurv(k, X, Y, Z)

PURPOSE

Returns a 3D curvilinear mimetic gradient

SYNOPSIS

```
function grad3DCurv
```

DESCRIPTION

CROSS-REFERENCE INFORMATION

This function calls:

grad3D() Returns a three-dimensional mimetic gradient operator depending on whether

grad() Returns a one-dimensional mimetic gradient operator depending on whether

GI13()

jacobian3D() Returns:

grad3DNonUniform(*k*, *xticks*, *yticks*, *zticks*)

PURPOSE

Returns a three-dimensional non-uniform mimetic gradient operator

SYNOPSIS

```
function grad3DNonUniform
```

DESCRIPTION

```
(including the boundaries!)

               k : Order of accuracy
          xticks : Centers' ticks (x-axis)
          yticks : Centers' ticks (y-axis)
          zticks : Centers' ticks (z-axis)
```

CROSS-REFERENCE INFORMATION

This function calls:

gradNonUniform() Returns a m+1 by m+2 one-dimensional non-uniform mimetic gradient

gradNonUniform(*k*, *ticks*)

PURPOSE

Returns a m+1 by m+2 one-dimensional non-uniform mimetic gradient

SYNOPSIS

```
function gradNonUniform
```

DESCRIPTION

```
operator
(including the boundaries!)
```

(continues on next page)

(continued from previous page)

```
              k : Order of accuracy
          ticks : Centers' ticks e.g. [0 0.5 1 3 5 7 8 9 9.5 10]
```

CROSS-REFERENCE INFORMATION

This function calls:

grad() Returns a one-dimensional mimetic gradient operator depending on whether

This function is called by:

grad2DNonUniform() Returns a two-dimensional non-uniform mimetic gradient operator

grad3DNonUniform() Returns a three-dimensional non-uniform mimetic gradient operator

Divergence Operators

div(*k*, *m*, *dx*, *dc*, *nc*)

PURPOSE

Returns a one-dimensional mimetic divergence operator depending on whether

SYNOPSIS

```
function div
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (2x1 vector for left and right vertices, resp.)
(optional) nc : b0 (2x1 vector for left and right vertices, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
```

CROSS-REFERENCE INFORMATION

This function calls:

`divNonPeriodic()` Returns a m+2 by m+1 one-dimensional mimetic divergence operator

`divPeriodic()` Returns a m+2 by m one-dimensional mimetic divergence operator

This function is called by:

mimeticB() Returns a m+2 by m+1 one-dimensional mimetic boundary operator

lap() Returns a one-dimensional mimetic Laplacian operator depending on whether

divNonUniform() Returns a m+2 by m+1 one-dimensional non-uniform mimetic divergence

weightsQ() Returns the m+2 weights of Q

div3DCurv()

div2D(*k*, *m*, *dx*, *n*, *dy*, *dc*, *nc*)

PURPOSE

Returns a two-dimensional mimetic divergence operator depending on whether

SYNOPSIS

```
function div2D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (4x1 vector for left, right, bottom, top boundaries, resp.)
(optional) nc : b0 (4x1 vector for left, right, bottom, top boundaries, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
               n : Number of cells along y-axis
              dy : Step size along y-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

`divNonPeriodic2D()` Returns a two-dimensional mimetic divergence operator

`divNonPeriodic()` Returns a m+2 by m+1 one-dimensional mimetic divergence operator

`divPeriodic()` Returns a m+2 by m one-dimensional mimetic divergence operator

This function is called by:

lap2D() Returns a two-dimensional mimetic Laplacian operator depending on whether

div2DCurv()

curl2D() Returns a two-dimensional mimetic curl operator

div2DCurv(*k*, *X*, *Y*)

div2DNonUniform(*k*, *xticks*, *yticks*)

PURPOSE

Returns a two-dimensional non-uniform mimetic divergence operator

SYNOPSIS

```
function div2DNonUniform
```

DESCRIPTION

```
    k : Order of accuracy
xticks : Edges' ticks (x-axis)
yticks : Edges' ticks (y-axis)
```

CROSS-REFERENCE INFORMATION

This function calls:

divNonUniform() Returns a m+2 by m+1 one-dimensional non-uniform mimetic divergence

div3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *dc*, *nc*)

PURPOSE

Returns a three-dimensional mimetic divergence operator depending on whether

SYNOPSIS

```
function div3D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)
(optional) nc : b0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)

               k : Order of accuracy
               m : Number of cells
              dx : Step size
               n : Number of cells along y-axis
              dy : Step size along y-axis
               o : Number of cells along z-axis
              dz : Step size along z-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

`divNonPeriodic()` Returns a m+2 by m+1 one-dimensional mimetic divergence operator

`divNonPeriodic3D()` Returns a three-dimensional mimetic divergence operator

`divPeriodic()` Returns a m+2 by m one-dimensional mimetic divergence operator

This function is called by:

div3DCurv()

lap3D() Returns a three-dimensional mimetic Laplacian operator depending on whether

div3DCurv(*k*, *X*, *Y*, *Z*)

div3DNonUniform(*k*, *xticks*, *yticks*, *zticks*)

PURPOSE

Returns a three-dimensional non-uniform mimetic divergence operator

SYNOPSIS

```
function div3DNonUniform
```

DESCRIPTION

```
     k : Order of accuracy
xticks : Edges' ticks (x-axis)
yticks : Edges' ticks (y-axis)
zticks : Edges' ticks (z-axis)
```

CROSS-REFERENCE INFORMATION

This function calls:

divNonUniform() Returns a m+2 by m+1 one-dimensional non-uniform mimetic divergence

divNonUniform(*k*, *ticks*)

PURPOSE

Returns a m+2 by m+1 one-dimensional non-uniform mimetic divergence

SYNOPSIS

```
function divNonUniform
```

DESCRIPTION

```
operator

              k : Order of accuracy
          ticks : Edges' ticks e.g. [0 0.1 0.15 0.2 0.3 0.4 0.45]
```

CROSS-REFERENCE INFORMATION

This function calls:

div() Returns a one-dimensional mimetic divergence operator depending on whether

This function is called by:

div3DNonUniform() Returns a three-dimensional non-uniform mimetic divergence operator

div2DNonUniform() Returns a two-dimensional non-uniform mimetic divergence operator

Curl Operators

curl2D(*k*, *m*, *dx*, *n*, *dy*, *west*, *east*, *south*, *north*, *U*, *V*)

PURPOSE

Returns a two-dimensional mimetic curl operator

SYNOPSIS

```
function curl2D
```

DESCRIPTION

```
west, east,
south, north : west, east, south, north limits
U(X,Y) must be defined as function handle
V(X,Y) must be defined as function handle

               k : Order of accuracy
               m : Number of cells along x-axis
              dx : Step size along x-axis
               n : Number of cells along y-axis
              dy : Step size along y-axis
               U : Vector space function acting on x-direction
               V : Vector space function acting on y-direction
```

CROSS-REFERENCE INFORMATION

This function calls:

div2D() Returns a two-dimensional mimetic divergence operator depending on whether

Laplacian Operators

lap(*k*, *m*, *dx*, *dc*, *nc*)

PURPOSE

Returns a one-dimensional mimetic Laplacian operator depending on whether

SYNOPSIS

```
function lap
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (2x1 vector for left and right vertices, resp.)
(optional) nc : b0 (2x1 vector for left and right vertices, resp.)
```

(continues on next page)

(continued from previous page)

```
             k : Order of accuracy
             m : Number of cells
            dx : Step size
```

CROSS-REFERENCE INFORMATION

This function calls:

div() Returns a one-dimensional mimetic divergence operator depending on whether

grad() Returns a one-dimensional mimetic gradient operator depending on whether

lapNonPeriodic() Returns a m+2 by m+2 one-dimensional mimetic laplacian operator

lap2D(*k*, *m*, *dx*, *n*, *dy*, *dc*, *nc*)

PURPOSE

Returns a two-dimensional mimetic Laplacian operator depending on whether

SYNOPSIS

```
function lap2D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (4x1 vector for left, right, bottom, top boundaries, resp.)
(optional) nc : b0 (4x1 vector for left, right, bottom, top boundaries, resp.)

              k : Order of accuracy
              m : Number of cells along x-axis
             dx : Step size along x-axis
              n : Number of cells along y-axis
             dy : Step size along y-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

grad2D() Returns a two-dimensional mimetic gradient operator depending on whether

div2D() Returns a two-dimensional mimetic divergence operator depending on whether

lapNonPeriodic2D() Returns a two-dimensional mimetic laplacian operator

lap3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *dc*, *nc*)

PURPOSE

Returns a three-dimensional mimetic Laplacian operator depending on whether

SYNOPSIS

```
function lap3D
```

DESCRIPTION

```
or not the operator will contain a periodic boundary condition type
a0 U + b0 dU/dn = g,
(optional) dc : a0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)
(optional) nc : b0 (6x1 vector for left, right, bottom, top, front, back boundary␣
↪types, resp.)

               k : Order of accuracy
               m : Number of cells along x-axis
              dx : Step size along x-axis
               n : Number of cells along y-axis
              dy : Step size along y-axis
               o : Number of cells along z-axis
              dz : Step size along z-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

div3D() Returns a three-dimensional mimetic divergence operator depending on whether

lapNonPeriodic3D() Returns a three-dimensional mimetic laplacian operator

grad3D() Returns a three-dimensional mimetic gradient operator depending on whether

5.3.2 Interpolation Functions

Node to Center Interpolation

interpolNodesToCenters1D(k, m)

PURPOSE

interpolation operator from nodal coordinates to staggered centers

SYNOPSIS

```
function interpolNodesToCenters1D
```

DESCRIPTION

```
m is the number of cells in the logical x-axis
nodal logical coordinates are [1:1:m]
centers logical coordinates [1,1.5:m-0.5,m]
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolFacesToCentersG1D() 1D interpolation from faces to centers

interpolNodesToCenters2D(*k*, *m*, *n*)

PURPOSE

interpolation operator from nodal coordinates to staggered centers

SYNOPSIS

```
function interpolNodesToCenters2D
```

DESCRIPTION

```
m, n, are the number of cells in the logical x-, y- axes
nodal logical coordinates are [1:1:m]x[1:1:n]
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolFacesToCentersG1D() 1D interpolation from faces to centers

interpolNodesToCenters3D(*k*, *m*, *n*, *o*)

PURPOSE

interpolation operator from nodal coordinates to staggered centers

SYNOPSIS

```
function interpolNodesToCenters3D
```

DESCRIPTION

```
m, n, o, are the number of cells in the logical x-, y-, z- axes
nodal logical coordinates are [1:1:m]x[1:1:n]x[1:1:o]
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]x[1,1.5:o-0.5,o]
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolFacesToCentersG1D() 1D interpolation from faces to centers

Center to Node Interpolation

interpolCentersToNodes1D(k, m)

PURPOSE

interpolation operator from nodal coordinates to staggered centers

SYNOPSIS

```
function interpolCentersToNodes1D
```

DESCRIPTION

```
m is the number of cells in the logical x-axis
nodal logical coordinates are [1:1:m]
centers logical coordinates [1,1.5:m-0.5,m]
```

CROSS-REFERENCE INFORMATION

This function calls:

`interpolCentersToFacesD1D()` 1D interpolation from centers to faces.

interpolCentersToNodes2D(k, m, n)

PURPOSE

interpolation operator from staggered to nodes

SYNOPSIS

```
function interpolCentersToNodes2D
```

DESCRIPTION

```
m, n, are the number of cells in the logical x-, y- axes
nodal logical coordinates are [1:1:m]x[1:1:n]
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]
```

CROSS-REFERENCE INFORMATION

This function calls:

`interpolCentersToFacesD1D()` 1D interpolation from centers to faces.

interpolCentersToNodes3D(k, m, n, o)

PURPOSE

interpolation operator from staggered to nodes

SYNOPSIS

```
function interpolCentersToNodes3D
```

DESCRIPTION

```
m, n, o, are the number of cells in the logical x-, y-, z- axes
nodal logical coordinates are [1:1:m]x[1:1:n]x[1:1:o]
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]x[1,1.5:o-0.5,o]
```

CROSS-REFERENCE INFORMATION

This function calls:

`interpolCentersToFacesD1D()` 1D interpolation from centers to faces.

Face Interpolation

interpolFacesToCentersG1D(k, m)

PURPOSE

1D interpolation from faces to centers

SYNOPSIS

```
function interpolFacesToCentersG1D
```

DESCRIPTION

```
centers logical coordinates [1,1.5:m-0.5,m]
m is the number of cells in the logical x-axis
```

CROSS-REFERENCE INFORMATION

This function is called by:

interpolFacesToCentersG3D() 3D interpolation from faces to centers

interpolFacesToCentersG2D() 2D interpolation from faces to centers

interpolNodesToCenters3D() interpolation operator from nodal coordinates to staggered centers

interpolNodesToCenters2D() interpolation operator from nodal coordinates to staggered centers

interpolNodesToCenters1D() interpolation operator from nodal coordinates to staggered centers

interpolFacesToCentersG2D(k, m, n)

PURPOSE

2D interpolation from faces to centers

SYNOPSIS

```
function interpolFacesToCentersG2D
```

DESCRIPTION

```
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]
m, n, are the number of cells in the logical x- and y- axes
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolFacesToCentersG1D() 1D interpolation from faces to centers

interpolFacesToCentersG3D(*k*, *m*, *n*, *o*)

PURPOSE

3D interpolation from faces to centers

SYNOPSIS

```
function interpolFacesToCentersG3D
```

DESCRIPTION

```
centers logical coordinates [1,1.5:m-0.5,m]x[1,1.5:n-0.5,n]x[1,1.5:o-0.5,o]
m, n, o, are the number of cells in the logical x-, y-, z- axes
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolFacesToCentersG1D() 1D interpolation from faces to centers

General Interpolation

interpol(*m*, *c*)

PURPOSE

Returns a m+1 by m+2 one-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpol
```

DESCRIPTION

```
m : Number of cells
c : Left interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function is called by:

interpol3D() Returns a three-dimensional interpolator of 2nd-order

interpol2D() Returns a two-dimensional interpolator of 2nd-order

interpol2D(*m*, *n*, *c1*, *c2*)

PURPOSE

Returns a two-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpol2D
```

DESCRIPTION

```
 m : Number of cells along x-axis
 n : Number of cells along y-axis
c1 : Left interpolation coeff.
c2 : Bottom interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function calls:

interpol() Returns a m+1 by m+2 one-dimensional interpolator of 2nd-order

interpol3D(*m*, *n*, *o*, *c1*, *c2*, *c3*)

PURPOSE

Returns a three-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpol3D
```

DESCRIPTION

```
 m : Number of cells along x-axis
 n : Number of cells along y-axis
 o : Number of cells along z-axis
c1 : Left interpolation coeff.
c2 : Bottom interpolation coeff.
c3 : Front interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function calls:

interpol() Returns a m+1 by m+2 one-dimensional interpolator of 2nd-order

interpolD(*m*, *c*)

PURPOSE

Returns a m+2 by m+1 one-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpolD
```

DESCRIPTION

```
m : Number of cells
c : Left interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function is called by:

interpolD2D() Returns a two-dimensional interpolator of 2nd-order

interpolD3D() Returns a three-dimensional interpolator of 2nd-order

interpolD2D(*m*, *n*, *c1*, *c2*)

PURPOSE

Returns a two-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpolD2D
```

DESCRIPTION

```
 m : Number of cells along x-axis
 n : Number of cells along y-axis
c1 : Left interpolation coeff.
c2 : Bottom interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolD() Returns a m+2 by m+1 one-dimensional interpolator of 2nd-order

interpolD3D(*m*, *n*, *o*, *c1*, *c2*, *c3*)

PURPOSE

Returns a three-dimensional interpolator of 2nd-order

SYNOPSIS

```
function interpolD3D
```

DESCRIPTION

```
 m : Number of cells along x-axis
 n : Number of cells along y-axis
 o : Number of cells along z-axis
c1 : Left interpolation coeff.
c2 : Bottom interpolation coeff.
c3 : Front interpolation coeff.
```

CROSS-REFERENCE INFORMATION

This function calls:

interpolD() Returns a m+2 by m+1 one-dimensional interpolator of 2nd-order

5.3.3 Boundary Conditions

General Boundary Conditions

addScalarBC1D(*A*, *b*, *k*, *m*, *dx*, *dc*, *nc*, *v*)

PURPOSE

Separates cases non-periodic and periodic for dealing with boundary data

SYNOPSIS

```
function addScalarBC1D
```

DESCRIPTION

```
output
input

              A0 : Linear operator with boundary conditions added
              b0 : Right hand side with boundary conditions added
               A : Linear operator without boundary conditions added
               b : Right hand side without boundary conditions added
               k : Order of accuracy
               m : Number of cells
              dx : Step size
              dc : a0 (2x1 vector for left and right vertices, resp.)
              nc : b0 (2x1 vector for left and right vertices, resp.)
               v : g (2x1 vector for left and right vertices, resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

addScalarBC1Dlhs() This functions uses geometry and boundary type conditions to create

addScalarBC1Drhs() This function uses the non-periodic boundary condition type of each vertex

addScalarBC1Dlhs(*k*, *m*, *dx*, *dc*, *nc*)

PURPOSE

This functions uses geometry and boundary type conditions to create

SYNOPSIS

```
function addScalarBC1Dlhs
```

DESCRIPTION

```
modifications of matrix A associated to each of the boundary faces.
output
input

              Al : modification of matrix A due to left boundary condition
              Ar : modification of matrix A due to right boundary condition
               k : Order of accuracy
               m : Number of cells
              dx : Step size
              dc : Dirichlet coefficient (2x1 vector for left and right vertices,␣
↪resp.)
              nc : Neumann coefficient (2x1 vector for left and right vertices,␣
↪resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

grad() Returns a one-dimensional mimetic gradient operator depending on whether

This function is called by:

addScalarBC1D() Separates cases non-periodic and periodic for dealing with boundary data

addScalarBC2Dlhs() This functions uses geometry and boundary type conditions to create

addScalarBC3Dlhs() This functions uses geometry and boundary type conditions to create

addScalarBC1Drhs(*b*, *v*, *vec*)

PURPOSE

This function uses the non-periodic boundary condition type of each vertex

SYNOPSIS

```
function addScalarBC1Drhs
```

DESCRIPTION

```
and the rhs b values associated to left, and right vertices to modify the rhs␣
↪vector b.
output
input

              b : Right hand side with boundary conditions added
              b : Right hand side without boundary conditions added
              v : value (2x1 vector for left and right vertices, resp.)
            vec : vector with indices of rhs associated to bc
```

CROSS-REFERENCE INFORMATION

This function is called by:

addScalarBC1D() Separates cases non-periodic and periodic for dealing with boundary data

addScalarBC2D(*A*, *b*, *k*, *m*, *dx*, *n*, *dy*, *dc*, *nc*, *v*)

PURPOSE

This function assumes that the unknown u, which represents the discrete

SYNOPSIS

```
function addScalarBC2D
```

DESCRIPTION

```
solution the continuous second-order 2D PDE operator
L U = f,
with continuous boundary condition
a0 U + b0 dU/dn = g,
are given at the 2D cell centers and boundary face centers. Furthermore,
all discrete calculations are performed at the 2D cell centers and boundary
face centers.
The function receives as input quantities associated to the discrete
analog of the continuous problem given by the squared linear system
A u = b
where A is the discrete analog of L and b is the discrete analog of g,
both constructed by the user without boundary conditions.
The function output is the modified square linear system
A u = b
where both A and b include boundary condition information.
The boundary condition is always one of the following forms:
For Dirichlet set: a0 not equal zero and b0 = 0.
For Neumann set  : a0 = 0 and b0 not equal zero.
For Robin set    : both a0 and b0 not equal zero.
For Periodic set : both a0 = 0 and b0 = 0.
For periodic bc, it is assumed that not only u but also du/dn are the same
in both extremes of the domain since a second-order PDE is assumed.
Periodic boundary conditions can be applied along some axes and
non-periodic to some others.
For consistence with the way boundary operators are calculated to avoid
overwriting of the values v, the left and right boundary conditions are
assumed to be column vectors of (m+2)*n components, and the bottom and
top faces are assumed to be vectors of (m+2)*(n+2) components.
The order of these components is as follows:
For left and right edges, the ordering is the one given by columns vectors
where x increases. For bottom and top faces, the ordering is the one given
by columns vectors where y increases.
The code assumes the following assertions:
assert(k >= 2, 'k >= 2');
assert(mod(k, 2) == 0, 'k % 2 = 0');
assert(m >= 2*k+1, ['m >= ' num2str(2*k+1) ' for k = ' num2str(k)]);
output
input

               A : Linear operator with boundary conditions added
               b : Right hand side with boundary conditions added
               A : Linear operator without boundary conditions added
               b : Right hand side without boundary conditions added
               k : Order of accuracy
               m : Number of horizontal cells
              dx : Step size horizontal cells
               n : Number of vertical cells
              dy : Step size of vertical cells
              dc : a0 (4x1 vector for left, right, bottom, top boundaries, resp.)
              nc : b0 (4x1 vector for left, right, bottom, top boundaries, resp.)
               v : g (4x1 vector of arrays for left, right, bottom, top boundaries,␣
```

(continues on next page)

(continued from previous page)

```
↪resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

addScalarBC2Dlhs() This functions uses geometry and boundary type conditions to create

addScalarBC2Drhs() function b = addBC2Drhs(b, m, n, dc, nc, v, vec)

addScalarBC2Dlhs(*k*, *m*, *dx*, *n*, *dy*, *dc*, *nc*)

PURPOSE

This functions uses geometry and boundary type conditions to create

SYNOPSIS

```
function addScalarBC2Dlhs
```

DESCRIPTION

```
modifications of matrix A associated to each of the boundary edges.
output
input

            Abcl : Matrix coefficients associated to boundary conditions for left␣
↪edge
            Abcr : Matrix coefficients associated to boundary conditions for right␣
↪edge
            Abcb : Matrix coefficients associated to boundary conditions for bottom␣
↪edge
            Abct : Matrix coefficients associated to boundary conditions for top␣
↪edge
               k : Order of accuracy
               m : Number of the horizontal cells
              dx : Step size
               n : Number of the vertical cells
              dy : Horizontal cell size
              dc : a0 (4x1 vector for left, right, bottom, top boundaries, resp.)
              nc : b0 (4x1 vector for left, right, bottom, top boundaries resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

addScalarBC1Dlhs() This functions uses geometry and boundary type conditions to create

This function is called by:

addScalarBC2D() This function assumes that the unknown u, which represents the discrete

addScalarBC2Drhs(*b*, *dc*, *nc*, *v*, *rl*, *rr*, *rb*, *rt*)

PURPOSE

function b = addBC2Drhs(b, m, n, dc, nc, v, vec)

SYNOPSIS

```
function b = addBC2Drhs(b, m, n, dc, nc, v, vec)
```

DESCRIPTION

```
This function uses the boundary condition type of each face and the rhs b
indices and values associated to left, right, bottom, top, front, back
faces to modify the rhs vector b.
output
input

               b : Right hand side with boundary conditions added
               b : Right hand side without boundary conditions added
              dc : a0 (4x1 vector for left, right, bottom, top boundary types, resp.
↪)
              nc : b0 (4x1 vector for left, right, bottom, top boundary types, resp.
↪)
               v : g (4x1 vector of arrays for left, right, bottom, top boundaries,␣
↪resp.)
              rl : indices of rhs left indices
              rr : indices of rhs right indices
              rb : indices of rhs bottom indices
              rt : indices of rhs top indices
             vec : vector with indices of rhs associated to bc
```

CROSS-REFERENCE INFORMATION

This function is called by:

addScalarBC2D() This function assumes that the unknown u, which represents the discrete

addScalarBC3D(*A*, *b*, *k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *dc*, *nc*, *v*)

PURPOSE

This function assumes that the unknown u, which represents the discrete

SYNOPSIS

```
function addScalarBC3D
```

DESCRIPTION

```
solution the continuous second-order 3D PDE operator
L U = f,
with continuous boundary condition
a0 U + b0 dU/dn = g,
```

(continues on next page)

(continued from previous page)

```
are given at the 3D cell centers and boundary face centers. Furthermore,
all discrete calculations are performed at the 3D cell centers and boundary
face centers.
The function receives as input quantities associated to the discrete
analog of the continuous problem given by the squared linear system
A u = b
where A is the discrete analog of L and b is the discrete analog of g,
both constructed by the user without boundary conditions.
The function output is the modified square linear system
A u = b
where both A and b include boundary condition information.
The boundary condition is always one of the following forms:
For Dirichlet set: a0 not equal zero and b0 = 0.
For Neumann set  : a0 = 0 and b0 not equal zero.
For Robin set    : both a0 and b0 not equal zero.
For Periodic set : both a0 = 0 and b0 = 0.
For periodic bc, it is assumed that not only u but also du/dn are the same
in both extremes of the domain since a second-order PDE is assumed.
Periodic boundary conditions can be applied along some axes and
non-periodic to some others.
For consistence with the way boundary operators are calculated to avoid
overwriting of the values v, the left and right boundary conditions are
assumed to be column vectors of n*o components, the bottom and top
boundary conditions are assumed to be column vectors of (m+2)*o
components, and the front and back faces are assumed to be vectors of
(m+2)*(n+2) components.
The order of these components is as follows:
For left and right faces, the ordering is the one by columns of the
matrix where y increase along rows, and z increase along columns.
For bottom and top faces, the ordering is the one by columns of the
matrix where x increase along rows, and z increase along columns.
For front and back faces, the ordering is the one by columns of the
matrix where x increase along rows, and y increase along columns.
The code assumes the following assertions:
assert(k >= 2, 'k >= 2');
assert(mod(k, 2) == 0, 'k % 2 = 0');
assert(m >= 2*k+1, ['m >= ' num2str(2*k+1) ' for k = ' num2str(k)]);
output
input

               A : Linear operator with boundary conditions added
               b : Right hand side with boundary conditions added
               A : Linear operator without boundary conditions added
               b : Right hand side without boundary conditions added
               k : Order of accuracy
               m : Number of horizontal cells
              dx : Step size of horizontal
               n : Number of vertical cells
              dy : Step size of vertical cells
               o : Number of depth cells
              dz : Step size of depth cells
              dc : a0 (6x1 vector for left, right, bottom, top, front, back␣
```

(continues on next page)

(continued from previous page)

```
↪boundary types, resp.)
             nc : b0 (6x1 vector for left, right, bottom, top, front, back␣
↪boundary types, resp.)
              v : g (6x1 vector of arrays for left, right, bottom, top, front,␣
↪back boundaries, resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

addScalarBC3Dlhs() This functions uses geometry and boundary type conditions to create

addScalarBC3Drhs() This function uses the boundary condition type of each face and the rhs b

addScalarBC3Dlhs(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *dc*, *nc*)

PURPOSE

This functions uses geometry and boundary type conditions to create

SYNOPSIS

```
function addScalarBC3Dlhs
```

DESCRIPTION

```
modifications of matrix A associated to each of the boundary faces.
output
input

            Abcl : Matrix coefficients associated to boundary conditions for left␣
↪face
            Abcr : Matrix coefficients associated to boundary conditions for right␣
↪face
            Abcb : Matrix coefficients associated to boundary conditions for bottom␣
↪face
            Abct : Matrix coefficients associated to boundary conditions for top␣
↪face
            Abcf : Matrix coefficients associated to boundary conditions for front␣
↪face
            Abcz : Matrix coefficients associated to boundary conditions for back␣
↪face
               k : Order of accuracy
               m : Number of the horizontal cells
              dx : Step size
               n : Number of the vertical cells
              dy : Horizonttal cell size
               o : Number of the depth cells
              dz : Depth cell size
              dc : a0 (6x1 vector for left, right, bottom, top, front, back␣
↪boundary types, resp.)
```

(continues on next page)

(continued from previous page)

```
              nc : b0 (6x1 vector for left, right, bottom, top, front, back␣
↪boundary types, resp.)
```

CROSS-REFERENCE INFORMATION

This function calls:

addScalarBC1Dlhs() This functions uses geometry and boundary type conditions to create

This function is called by:

addScalarBC3D() This function assumes that the unknown u, which represents the discrete

addScalarBC3Drhs(*b*, *dc*, *nc*, *v*, *rl*, *rr*, *rb*, *rt*, *rf*, *rz*)

PURPOSE

This function uses the boundary condition type of each face and the rhs b

SYNOPSIS

```
function addScalarBC3Drhs
```

DESCRIPTION

```
indices and values associated to left, right, bottom, top, front, back
faces to modify the rhs vector b.
output
input

               b : Right hand side with boundary conditions added
               b : Right hand side without boundary conditions added
              dc : a0 (6x1 vector for left, right, bottom, top, front, back␣
↪boundary types, resp.)
              nc : b0 (6x1 vector for left, right, bottom, top, front, back␣
↪boundary types, resp.)
               v : g (6x1 vector of arrays for left, right, bottom, top, front,␣
↪back boundaries, resp.)
              rl : indices of rhs left indices
              rr : indices of rhs right indices
              rb : indices of rhs bottom indices
              rt : indices of rhs top indices
              rf : indices of rhs front indices
              rz : indices of rhs back indices
```

CROSS-REFERENCE INFORMATION

This function is called by:

addScalarBC3D() This function assumes that the unknown u, which represents the discrete

Neumann Boundary Conditions

neumann2DCurv(G, m, n, b)

PURPOSE

Returns a 2D curvilinear Neumann BC operator

SYNOPSIS

```
function neumann2DCurv
```

DESCRIPTION

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

neumann3DCurv(G, m, n, o, b)

PURPOSE

Returns a 3D curvilinear Neumann BC operator

SYNOPSIS

```
function neumann3DCurv
```

DESCRIPTION

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

Robin Boundary Conditions

robinBC(k, m, dx, a, b)

PURPOSE

Returns a m+2 by m+2 one-dimensional mimetic boundary operator that

SYNOPSIS

```
function robinBC
```

DESCRIPTION

```
imposes a boundary condition of Robin's type

               k : Order of accuracy
               m : Number of cells
              dx : Step size
               a : Dirichlet Coefficient
               b : Neumann Coefficient
```

CROSS-REFERENCE INFORMATION

This function calls:

grad() Returns a one-dimensional mimetic gradient operator depending on whether

This function is called by:

robinBC2D() Returns a two-dimensional mimetic boundary operator that

robinBC3D() Returns a three-dimensional mimetic boundary operator that

robinBC2D(*k*, *m*, *dx*, *n*, *dy*, *a*, *b*)

PURPOSE

Returns a two-dimensional mimetic boundary operator that

SYNOPSIS

```
function robinBC2D
```

DESCRIPTION

```
imposes a boundary condition of Robin's type

               k : Order of accuracy
               m : Number of cells along x-axis
              dx : Step size along x-axis
               n : Number of cells along y-axis
              dy : Step size along y-axis
               a : Dirichlet Coefficient
               b : Neumann Coefficient
```

CROSS-REFERENCE INFORMATION

This function calls:

robinBC() Returns a m+2 by m+2 one-dimensional mimetic boundary operator that

robinBC3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *a*, *b*)

PURPOSE

Returns a three-dimensional mimetic boundary operator that

SYNOPSIS

```
function robinBC3D
```

DESCRIPTION

```
imposes a boundary condition of Robin's type

               k : Order of accuracy
               m : Number of cells along x-axis
              dx : Step size along x-axis
               n : Number of cells along y-axis
              dy : Step size along y-axis
               o : Number of cells along z-axis
              dz : Step size along z-axis
               a : Dirichlet Coefficient
               b : Neumann Coefficient
```

CROSS-REFERENCE INFORMATION

This function calls:

robinBC() Returns a m+2 by m+2 one-dimensional mimetic boundary operator that

Mixed Boundary Conditions

mixedBC(*k*, *m*, *dx*, *left*, *coeffs_left*, *right*, *coeffs_right*)

PURPOSE

Constructs a 1D mimetic mixed boundary conditions operator

SYNOPSIS

```
function mixedBC
```

DESCRIPTION

```
          k : Order of accuracy
          m : Number of cells
         dx : Step size
       left : Type of boundary condition at the left boundary ('Dirichlet',
↪'Neumann', 'Robin')
 coeffs_left : Coefficients for the left boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
      right : Type of boundary condition at the right boundary ('Dirichlet',
↪'Neumann', 'Robin')
```

(continues on next page)

(continued from previous page)

```
coeffs_right : Coefficients for the right boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
```

CROSS-REFERENCE INFORMATION

This function calls:

grad() Returns a one-dimensional mimetic gradient operator depending on whether

This function is called by:

mixedBC2D() Constructs a 2D mimetic mixed boundary conditions operator

mixedBC3D() Constructs a 3D mimetic mixed boundary conditions operator

mixedBC2D(*k*, *m*, *dx*, *n*, *dy*, *left*, *coeffs_left*, *right*, *coeffs_right*, *bottom*, *coeffs_bottom*, *top*, *coeffs_top*)

PURPOSE

Constructs a 2D mimetic mixed boundary conditions operator

SYNOPSIS

```
function mixedBC2D
```

DESCRIPTION

```
           k : Order of accuracy
           m : Number of cells in x-direction
          dx : Step size in x-direction
           n : Number of cells in y-direction
          dy : Step size in y-direction
        left : Type of boundary condition at the left boundary ('Dirichlet',
↪'Neumann', 'Robin')
  coeffs_left : Coefficients for the left boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
       right : Type of boundary condition at the right boundary ('Dirichlet',
↪'Neumann', 'Robin')
 coeffs_right : Coefficients for the right boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
      bottom : Type of boundary condition at the bottom boundary ('Dirichlet',
↪'Neumann', 'Robin')
coeffs_bottom : Coefficients for the bottom boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
         top : Type of boundary condition at the top boundary ('Dirichlet',
↪'Neumann', 'Robin')
   coeffs_top : Coefficients for the top boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
```

CROSS-REFERENCE INFORMATION

This function calls:

mixedBC() Constructs a 1D mimetic mixed boundary conditions operator

mixedBC3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*, *left*, *coeffs_left*, *right*, *coeffs_right*, *bottom*, *coeffs_bottom*, *top*, *coeffs_top*, *front*, *coeffs_front*, *back*, *coeffs_back*)

PURPOSE

Constructs a 3D mimetic mixed boundary conditions operator

SYNOPSIS

```
function mixedBC3D
```

DESCRIPTION

```
           k : Order of accuracy
           m : Number of cells in x-direction
          dx : Step size in x-direction
           n : Number of cells in y-direction
          dy : Step size in y-direction
           o : Number of cells in z-direction
          dz : Step size in z-direction
        left : Type of boundary condition at the left boundary ('Dirichlet',
↪'Neumann', 'Robin')
  coeffs_left : Coefficients for the left boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
       right : Type of boundary condition at the right boundary ('Dirichlet',
↪'Neumann', 'Robin')
 coeffs_right : Coefficients for the right boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
      bottom : Type of boundary condition at the bottom boundary ('Dirichlet',
↪'Neumann', 'Robin')
coeffs_bottom : Coefficients for the bottom boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
         top : Type of boundary condition at the top boundary ('Dirichlet',
↪'Neumann', 'Robin')
   coeffs_top : Coefficients for the top boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
       front : Type of boundary condition at the front boundary ('Dirichlet',
↪'Neumann', 'Robin')
 coeffs_front : Coefficients for the front boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
        back : Type of boundary condition at the back boundary ('Dirichlet',
↪'Neumann', 'Robin')
  coeffs_back : Coefficients for the back boundary condition (a, b for Robin,␣
↪otherwise coeff. for Dirichlet or Neumann)
```

CROSS-REFERENCE INFORMATION

This function calls:

mixedBC() Constructs a 1D mimetic mixed boundary conditions operator

5.3.4 Grid Generation and Transformation

Grid Generation

gridGen(*method*, *grid_name*, *m*, *n*, *plot_grid*, *varargin*)

PURPOSE

Returns X and Y which are both m by n matrices that contains the physical

SYNOPSIS

```
function gridGen
```

DESCRIPTION

```
coordinates

      grid_name : String with the name of the grid folder
              m : Number of nodes along the horizontal axis
              n : Number of nodes along the vertical axis
      plot_grid : If true -> plot the grid
       varargin : Maximum number of iterations (Required for TTM)
```

CROSS-REFERENCE INFORMATION

This function calls:

ttm() https://www.sciencedirect.com/science/article/pii/0022247X78902172?via%3Dihub

tfi() https://en.wikipedia.org/wiki/Transfinite_interpolation

tfi(*grid_name*, *m*, *n*, *plot_grid*)

PURPOSE

Returns X and Y which are both m by n matrices that contains the physical

SYNOPSIS

```
function tfi
```

DESCRIPTION

```
coordinates

      grid_name : String with the name of the grid folder
```

(continues on next page)

(continued from previous page)

```
            m : Number of nodes along the horizontal axis
            n : Number of nodes along the vertical axis
    plot_grid : If defined -> grid will be plotted
```

CROSS-REFERENCE INFORMATION

This function is called by:

gridGen() Returns X and Y which are both m by n matrices that contains the physical

Jacobian Calculation

jacobian2D(k, X, Y)

PURPOSE

returns
Determinant of the Jacobian (XeYn - XnYe)

SYNOPSIS

```
function jacobian2D
```

DESCRIPTION

```
Xe : dx/de metric
Xn : dx/dn metric
Ye : dy/de metric
Yn : dy/dn metric
 k : Order of accuracy
 X : x-coordinates (physical) of meshgrid
 Y : y-coordinates (physical) of meshgrid
```

CROSS-REFERENCE INFORMATION

This function calls:

nodal2D() Returns a two-dimensional operator that approximates the first-order

This function is called by:

div2DCurv()

grad2DCurv() Returns a 2D curvilinear mimetic gradient

nodal2DCurv()

jacobian3D(k, X, Y, Z)

PURPOSE

returns
Determinant of the Jacobian

SYNOPSIS

```
function jacobian3D
```

DESCRIPTION

```
Xe : dx/de metric
Xn : dx/dn metric
Xc : dx/dc metric
Ye : dy/de metric
Yn : dy/dn metric
Yc : dy/dc metric
Ze : dz/de metric
Zn : dz/dn metric
Zc : dz/dc metric
 k : Order of accuracy
 X : x-coordinates (physical) of meshgrid
 Y : y-coordinates (physical) of meshgrid
 Z : z-coordinates (physical) of meshgrid
```

CROSS-REFERENCE INFORMATION

This function calls:

nodal3D() Returns a three-dimensional operator that approximates the first-order

This function is called by:

nodal3DCurv()

div3DCurv()

grad3DCurv() Returns a 3D curvilinear mimetic gradient

Nodal Operators

nodal(*k*, *m*, *dx*)

PURPOSE

Returns a m+1 by m+1 one-dimensional operator that approximates the

SYNOPSIS

```
function nodal
```

DESCRIPTION

```
first-order derivatives on a uniform nodal grid

               k : Order of accuracy
               m : Number of nodes
              dx : Step size
```

CROSS-REFERENCE INFORMATION

This function is called by:

nodal3D() Returns a three-dimensional operator that approximates the first-order

nodal2D() Returns a two-dimensional operator that approximates the first-order

nodal2D(*k*, *m*, *dx*, *n*, *dy*)

PURPOSE

Returns a two-dimensional operator that approximates the first-order

SYNOPSIS

```
function nodal2D
```

DESCRIPTION

```
derivatives on a uniform nodal grid

               k : Order of accuracy
               m : Number of nodes along x-axis
              dx : Step size along x-axis
               n : Number of nodes along y-axis
              dy : Step size along y-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

nodal() Returns a m+1 by m+1 one-dimensional operator that approximates the

This function is called by:

jacobian2D() Returns:

nodal2DCurv()

nodal2DCurv(*k*, *X*, *Y*)

nodal3D(*k*, *m*, *dx*, *n*, *dy*, *o*, *dz*)

PURPOSE

Returns a three-dimensional operator that approximates the first-order

SYNOPSIS

```
function nodal3D
```

DESCRIPTION

```
derivatives on a uniform nodal grid

               k : Order of accuracy
               m : Number of nodes along x-axis
              dx : Step size along x-axis
               n : Number of nodes along y-axis
              dy : Step size along y-axis
               o : Number of nodes along z-axis
              dz : Step size along z-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

nodal() Returns a m+1 by m+1 one-dimensional operator that approximates the

This function is called by:

nodal3DCurv()

jacobian3D() Returns:

nodal3DCurv(*k*, *X*, *Y*, *Z*)

sidedNodal(*m*, *dx*, *type*)

PURPOSE

Returns a m+1 by m+1 one-dimensional sided approximation for uniformly

SYNOPSIS

```
function sidedNodal
```

DESCRIPTION

```
spaced data points. This function is handy for advective terms.

               m : Number of cells
              dx : Step size
            type : 'backward', 'forward' or 'centered'
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

5.3.5 Mimetic Weights

weightsP(*k*, *m*, *dx*)

PURPOSE

Returns the m+1 weights of P

SYNOPSIS

```
function weightsP
```

DESCRIPTION

```
 k : Order of accuracy
 m : Number of cells
dx : Step size
```

CROSS-REFERENCE INFORMATION

This function calls:

grad() Returns a one-dimensional mimetic gradient operator depending on whether

This function is called by:

weightsP2D() Returns the 2mn+m+n weights of P in 2-D

mimeticB() Returns a m+2 by m+1 one-dimensional mimetic boundary operator

`generateWeights()` Generates a comma-delimited file for weights used by MOLE libray

weightsP2D(*k*, *m*, *dx*, *n*, *dy*)

PURPOSE

Returns the 2mn+m+n weights of P in 2-D

SYNOPSIS

```
function weightsP2D
```

DESCRIPTION

```
 k : Order of accuracy
 m : Number of cells along x-axis
dx : Step size along x-axis
 n : Number of cells along y-axis
dy : Step size along y-axis
```

CROSS-REFERENCE INFORMATION

This function calls:

weightsP() Returns the m+1 weights of P

weightsQ(*k*, *m*, *dx*)

PURPOSE

Returns the m+2 weights of Q

SYNOPSIS

```
function weightsQ
```

DESCRIPTION

```
 k : Order of accuracy
 m : Number of cells
dx : Step size
```

CROSS-REFERENCE INFORMATION

This function calls:

div() Returns a one-dimensional mimetic divergence operator depending on whether

This function is called by:

mimeticB() Returns a m+2 by m+1 one-dimensional mimetic boundary operator

`generateWeights()` Generates a comma-delimited file for weights used by MOLE libray

weightsQ2D(*m*, *n*, *d*)

PURPOSE

Returns the (m+2)(n+2) weights of Q in 2-D

SYNOPSIS

```
function weightsQ2D
```

DESCRIPTION

```
Only works for 2nd-order 2-D Mimetic divergence operator

               m : Number of cells along x-axis
               n : Number of cells along y-axis
               d : Step size (assuming d = dx = dy)
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

5.3.6 Utility Functions

amean(*X*)

PURPOSE

Returns the arithmetic mean for every two pairs in a column vector

SYNOPSIS

```
function amean
```

DESCRIPTION

```
And, Y(1) = X(1), Y(end) = X(end)

               X : Column vector
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

hmean(*X*)

PURPOSE

Returns the harmonic mean for every two pairs in a column vector

SYNOPSIS

```
function hmean
```

DESCRIPTION

```
And, Y(1) = X(1), Y(end) = X(end)

               X : Column vector
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

rk4(*func*, *tspan*, *dt*, *y0*)

PURPOSE

Explicit Runge-Kutta 4th-order method

SYNOPSIS

```
function rk4
```

DESCRIPTION

```
Returns : t (evaluation points) and y (solutions) of the specified ODE
   func : Function handler
  tspan : [t0 tf]
     dt : Step size
     y0 : Initial conditions
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

ttm(*grid_name*, *m*, *n*, *iters*, *plot_grid*)

PURPOSE

Returns X and Y which are both m by n matrices that contains the physical

SYNOPSIS

```
function ttm
```

DESCRIPTION

```
coordinates

       grid_name : String with the name of the grid folder
               m : Number of nodes along the horizontal axis
               n : Number of nodes along the vertical axis
       plot_grid : If defined -> grid will be plotted
```

CROSS-REFERENCE INFORMATION

This function is called by:

gridGen() Returns X and Y which are both m by n matrices that contains the physical

boundaryIdx2D(*m*, *n*)

PURPOSE

Returns the indices of the nodes that lie on the boundary of a 2D nodal

SYNOPSIS

```
function boundaryIdx2D
```

DESCRIPTION

```
grid

               m : Number of nodes along x-axis
               n : Number of nodes along y-axis
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

DI2(*m*, *n*, *type*)

DI3(*m*, *n*, *o*, *type*)

GI1(*M*, *m*, *n*, *type*)

GI13(*M*, *m*, *n*, *o*, *type*)

GI2(*M*, *m*, *n*, *type*)

mimeticB(*k*, *m*)

PURPOSE

Returns a m+2 by m+1 one-dimensional mimetic boundary operator

SYNOPSIS

```
function mimeticB
```

DESCRIPTION

```
k : Order of accuracy
m : Number of cells
```

CROSS-REFERENCE INFORMATION

This function calls:

div() Returns a one-dimensional mimetic divergence operator depending on whether

weightsP() Returns the m+1 weights of P

grad() Returns a one-dimensional mimetic gradient operator depending on whether

weightsQ() Returns the m+2 weights of Q

tensorGrad2D(*K*, *G*)

PURPOSE

Returns a two-dimensional flux operator

SYNOPSIS

```
function tensorGrad2D
```

DESCRIPTION

```
K : Tensor (e.g. diffusion tensor)
G : 2D mimetic gradient operator
```

CROSS-REFERENCE INFORMATION

No cross-reference information found. This typically means this function neither calls nor is called by other functions in the codebase.

CHAPTER

SIX

MATHEMATICAL FUNCTIONS REFERENCE

This section provides detailed mathematical descriptions of the operators implemented in MOLE.

6.1 CSRC Report on MOLE Library

6.1.1 Introduction

Physical phenomena are typically modeled as a set of differential equations subject to conservation laws. Numerical methods used to solve these equations are of vital importance in the paradigm of computational science. In this document, we talk about MOLE, an open-source library that implements mimetic discretization methods (MDM) to intuitively solve partial differential equations (PDE).

Mimetic operators are derived by constructing discrete analogs of the continuum differential operators $\nabla, \nabla\cdot, \nabla\times$, and ∇^2. Since most continuum models are described in terms of these operators, the MDM approach has recently gained space in the context of numerical PDEs.

Qualities of a mimetic operator:

- It is a discrete analog of the continuum operator
- It satisfies identities from vector calculus
- It satisfies global and local conservation laws
- It provides uniform order of accuracy
- It is easy to use (and reusable)

In 2003, Castillo and Grone came up with a matrix analysis approach to construct high-order approximations of divergence and gradient operators *[5]*. However, in their approach, the 4^{th}-order operators have three free-parameters. The mimetic operators implemented in MOLE are based on the work of *[6]* which are a substantial improvement of the operators introduced in *[5]*. These new operators have no free-parameters, have optimal bandwidth, are more accurate, and in the worst case they deliver the same accuracy as the ones from 2003.

There are many applications of MDM solving continuum problems, including in the geosciences (porous media) [*1*, *11*]; fluid dynamics (Navier-Stokes) [*2*, *3*]; image processing [*9*]; general relativity [*8*]; and electromagnetism [*10*].

6.1.2 On the Mathematics

MDM not only provide uniform order of accuracy (all the way to the boundary), but they also satisfy fundamental identities from vector calculus,

- Gradient of a constant $G\,f = 0$
- Free stream preservation $D\,v = 0$
- Curl of the gradient $C\,G\,f = 0$

- Divergence of the curl $D\,C\,v = 0$
- Divergence of the gradient $D\,G\,f = L\,f$

In addition, the discrete version of the extended Gauss' divergence theorem is also satisfied with high-order accuracy:

$$\langle D\,v, f\rangle_Q + \langle v, G\,f\rangle_P = \langle B\,v, f\rangle \; - - - \; (1)$$

The deduction of equation (1) can be found in *[4]*.

When using MDM we are not discretizing the equations (as it is done with standard finite-difference methods (FDM)), but instead we construct a discrete analog to the differential operator,

$$\begin{array}{ccc} \frac{\partial^2 f}{\partial x^2} & & \frac{\partial^2}{\partial x^2} f \\ \downarrow & vs & \downarrow \\ \frac{f_{j+1} - 2f_j + f_{j-1}}{\Delta\, x^2} & & D\,G\,f \end{array}$$

6.2 Staggered Grids

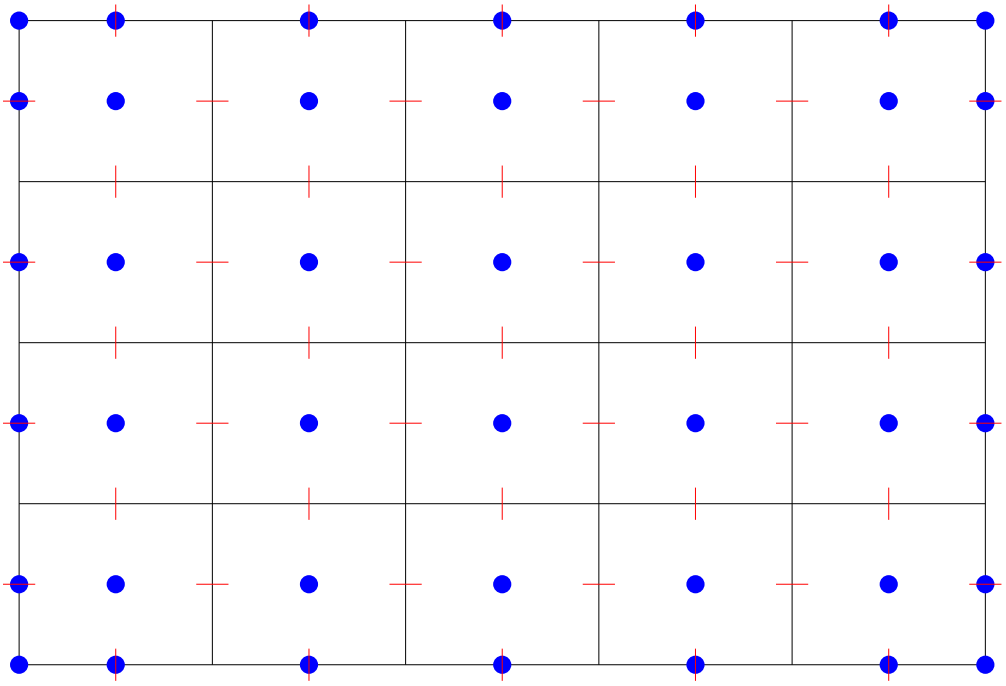
Mimetic operators are defined over staggered grids.

In 1D:

Staggered grid representation in 1D showing scalar and vector field locations

Scalar fields live in blue circles and vector fields in vertical red segments.

In 2D:



Staggered grid representation in 2D showing scalar and vector field component locations

Scalar fields are defined at blue circles, and horizontal and vertical vector field components are defined at horizontal and vertical red segments, respectively.

In 3D:

The continuum domain is a parallelepiped. Scalar fields are defined at cell centers and at the cell middle edges that are part of the edges of the parallelepiped. Vector fields for the horizontal, vertical, and in depth vector field components are defined at the cell face centers that are perpendicular to the horizontal, vertical and in depth directions, respectively.

There are several ways to generate uniform structured staggered grids (in MATLAB/Octave). What is important is to remember that we need to store the coordinates for two different quantities.

Suppose

$$\text{west} = 0; \quad \text{east} = 1; \quad m = 10; \quad dx = (\text{east} - \text{west})/m;$$

$$\text{xgridSca} = [\text{west} \quad \text{west} + dx/2 : dx : \text{east} - dx/2 \quad \text{east}];$$

This grid holds the coordinates of all scalar quantities (cell center + boundaries). Vector quantities are just:

$$\text{xgridVec} = \text{west} : dx : \text{east};$$

The previous commands create a couple of one-dimensional arrays with the coordinates of each field (scalar and vectorial). Now, we need a couple of arrays to hold the actual values of these fields.

NOTE: Depending on the problem, you may need or not to explicitly create such arrays. It is often a good practice to preallocate the memory for better performance:

$$\text{scalarField} = \text{zeros}(\text{numel}(\text{xgridSca}), 1);$$

and

$$\text{vectorField} = \text{zeros}(\text{numel}(\text{xgridVec}), 1);$$

6.2.1 Using the Operators

Inside the "**examples/matlab**" folder you will find several MATLAB/Octave scripts that use MOLE to solve well known partial differential equations.

Our selection includes steady-state and time-dependent problems:

- Burger's equation
- Richards' equation (highly nonlinear, mixed form)
- Wave equation (with symplectic schemes)
- Heat equation (explicitly and implicitly)
- Etc.

Each script in the "**examples/matlab**" folder is adequately commented.

You may notice that all programs have the same taxonomy:

1. Definition of variables and initialization (initial and boundary conditions). Grid generation and obtainment of Mimetic Operators.
2. Apply operators. Solve system of equations (if applicable). Time integration (if applicable). Update fields.
3. Plot or process results. Check for conservation of energy, mass, etc. (optional).

6.2.2 References

1. Aarnes, J., Krogstad, S., and Lie, K-A, Multiscale mixed/mimetic methods on corner-point grids. Computational Geoscience. 2008.

2. Abba, A., and Bonaventura, L., A mimetic finite-difference method for large eddy simulation of incompressible flow. Technical Report. Politecnico di Milano, Milano, Italy, August 2010.

3. Barbosa, E., and Daube, O., A finite-difference method for 3D incompressible flows in cylindrical coordinates. Computational Fluids. 2005.

4. Castillo, J.E., and Miranda, G.F., Mimetic Discretization Methods, CRC Press, Boca Raton, Florida, USA, 2013.

5. Castillo, J.E., and Grone, R.D., A matrix analysis approach to higher-order approximations for divergence and gradients satisfying a global conservative law, SIAM J. Matrix Anal. Appl., Vol. 25, No. 1, pp. 128-142, 2003.

6. Corbino, J., and Castillo, J.E., High-order mimetic difference operators satisfying the extended Gauss divergence theorem, Journal of Computational and Applied Mathematics, 364 (2020).

7. Corbino, J., Dumett, M.A, and Castillo, J.E., MOLE: Mimetic Operators Library Enhanced: The Open-Source Library for Solving Partial Differential Equations using Mimetic Methods, 2024. JOSS.

8. Di Bartolo, C., Gambini, R., and Pullin, J., Consistent and mimetic discretizations in general relativity. Journal of mathematical physics. 2005.

9. Haber, E., and Modersitzki, J., A multilevel method for image registration. SIAM journal on scientific computing. 2006.

10. Hyman, J.M., and Shashkov, M., Mimetic discretizations for Maxwell's equations and equations of magnetic diffusion. Fourth International Conference on Mathematical and Numerical Aspects of Wave Propagation, Golden, Colorado. SIAM. 1998.

11. Hyman, J.M., Morel, J., Shashkov, M., and Steinberg, S., Mimetic finite-difference methods for diffusion equations. Computational Geoscience. 2002.

6.3 Why Staggered Grids?

Authors: Johnny Corbino, Miguel A. Dumett
Abstract: This document justifies the usage of staggered grids in mimetic differences.

6.3.1 Introduction

The general purpose for using staggered grids is to differentiate between (physical) quantities that utilize exterior and interior boundary orientations. Naturally, this splitting triggers the usage of a primal and a dual grids. Nevertheless, the current implementation of MOLE does not considers orientations, therefore in what follows an argument in favor of using staggered grids is given.

6.3.2 Justification

Mimetic difference operators are constructed for staggered grids [*1*].

This allows:

- More accurate solutions.
- Expected physical behavior (overcomes checkerboard pressure fields).

6.3.3 Checkerboard pressure fields

Suppose a two-dimensional incompressible flow without a body force. The governing equations are:

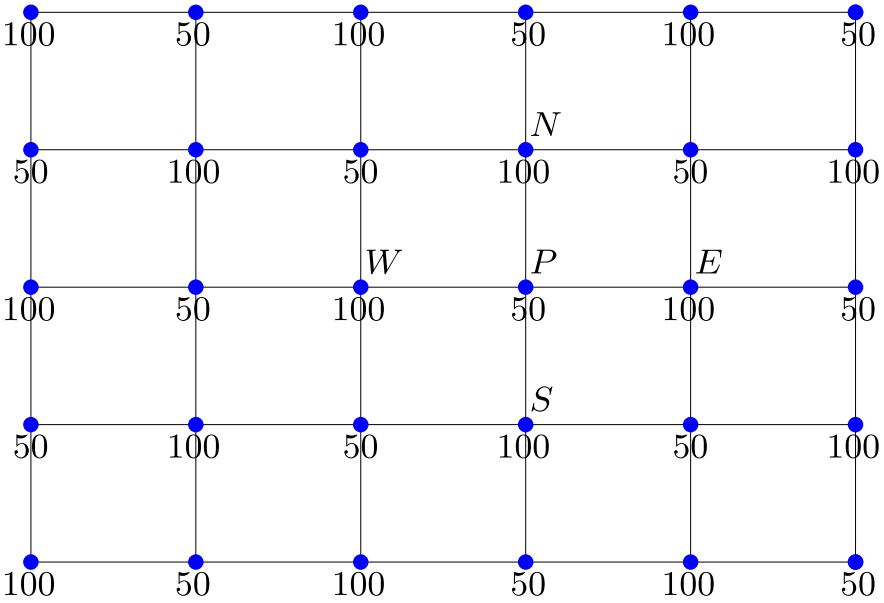
$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

$$\rho\, u \frac{\partial u}{\partial x} + \rho\, v \frac{\partial u}{\partial y} = \frac{\partial}{\partial x}\left(\mu \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\mu \frac{\partial u}{\partial y}\right) - \frac{\partial p}{\partial x}$$

$$\rho\, u \frac{\partial v}{\partial x} + \rho\, v \frac{\partial v}{\partial y} = \frac{\partial}{\partial x}\left(\mu \frac{\partial v}{\partial x}\right) + \frac{\partial}{\partial y}\left(\mu \frac{\partial v}{\partial y}\right) - \frac{\partial p}{\partial y}$$

Equations (1)-(3) are the continuity equation, and the momentum equations in the x- and y-direction, respectively.

The checkerboard pressure field is shown in Figure 1.



Checkerboard pressure field showing the pressure oscillations

The figure represents a checkerboard pressure field.

The source term for the momentum equations in the x- and y-direction can be expressed as

$$S_u = -\frac{\partial p}{\partial x} \quad \text{and} \quad S_v = -\frac{\partial p}{\partial y}$$

respectively. If central difference is employed, the pressure gradient becomes,

$$\left(\frac{\partial p}{\partial x}\right)_P = \frac{p_E - p_W}{\Delta x} = \frac{\frac{p_E+p_P}{2} - \frac{p_P+p_W}{2}}{\Delta x} = \frac{p_E - p_W}{2\,\Delta x}$$

and similarly,

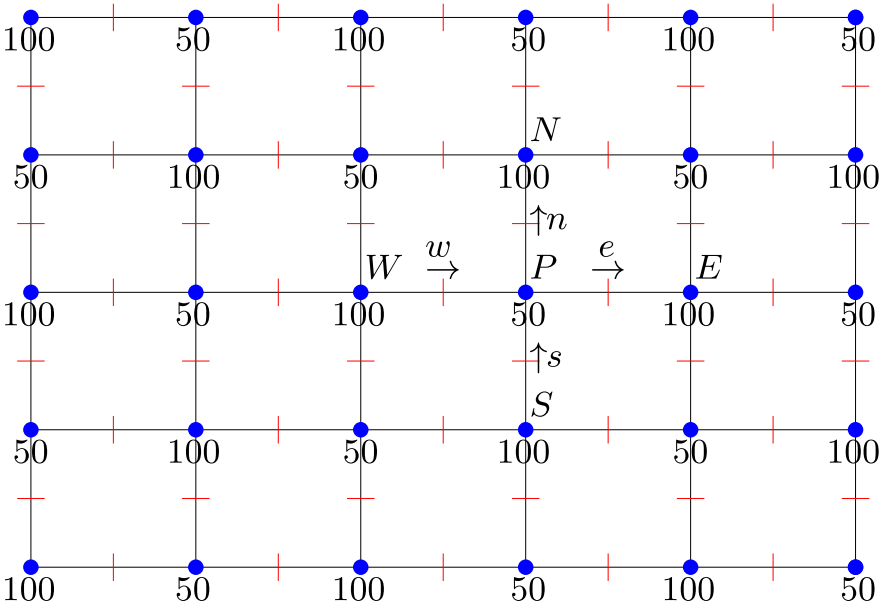
$$\left(\frac{\partial p}{\partial y}\right)_P = \frac{p_N - p_S}{2\,\Delta y}$$

From equations (4)-(5), we can see that **the pressure gradient at point P is related to the pressures of the neighbor grid points and not to its own pressure**. Therefore, if we have a pressure distribution as shown in the figure above, the discretization scheme represented by equations (4)-(5) will obtain $\frac{\partial p}{\partial x} = 0$ and $\frac{\partial p}{\partial y} = 0$ throughout the computational domain. This means that **it will not recognize the difference between a checkerboard pressure field and a uniform pressure field**. This behavior is obviously non-physical.

6.3.4 Staggered grids

A remedy for the aforementioned problem is to use staggered grids. The idea is to store scalar variables at the ordinary nodal points, and vector variables at the cell faces in between the nodal points.

The staggered grid arrangement is shown in Figure 2.



Staggered grid arrangement showing the location of different variables

The figure shows the new arrangement.

If we consider the checkerboard pressure field again, substitution of the appropriate nodal pressure values into equations (4)-(5) now yields very significant non-zero pressure gradient terms. The staggering of the velocity avoids the unrealistic behavior for spatially oscillating pressure fields. In addition, this new arrangement does not require interpolation to calculate velocities at the cell faces (where they are needed for the scalar transport-convection-diffusion computations.

6.3.5 References

1. J. Corbino, and J.E. Castillo, “High-order mimetic finite-difference operators satisfying the extended Gauss divergence theorem,” J. Comput. Appl. Math., vol. 364, 2020, 112326.

6.4 Jacobian

Author: Miguel A. Dumett
Abstract: This document presents mimetic differences gradient and divergence operators in structured curvilinear geometries. It uses mimetic interpolation operators to move quantities among the staggered grids of the mesh.

6.4.1 Introduction

Suppose a PDE is given on a physical spatial domain $\mathcal{P}$ in three-dimensions (3D), with coordinates x, y, z. Suppose $\mathcal{P}$ is the result of a bijective smooth map $\mathcal{X}$ given by

$$\begin{aligned} x = &\quad x(\xi,\eta,\kappa) \\ y = &\quad y(\xi,\eta,\kappa) \\ z = &\quad z(\xi,\eta,\kappa), \end{aligned}$$

and that the inverse map of $\mathcal{X}$ is Θ which is given by

$$\begin{aligned} \xi = &\quad \xi(x,y,z) \\ \eta = &\quad \eta(x,y,z) \\ \kappa = &\quad \kappa(x,y,z), \end{aligned}$$

and it maps $\mathcal{P}$ onto a 3D logical Cartesian domain $\mathcal{L}$. If one defines a staggered grid on $\mathcal{L}$, composed of faces F and centers/boundaries C, then $\mathcal{X}(C \cup F)$ is an structured grid on $\mathcal{P}$, with centers/boundaries $\mathcal{C} = \mathcal{X}(C)$ and faces $\mathcal{F} = \mathcal{X}(F)$.

The Jacobian of the transformation $\mathcal{X}$ is given by

$$J = \frac{\partial(x,y,z)}{\partial(\xi,\eta,\kappa)} = \begin{bmatrix} x_\xi & x_\eta & x_\kappa \\ y_\xi & y_\eta & y_\kappa \\ z_\xi & z_\eta & z_\kappa \end{bmatrix}.$$

For $u : \mathcal{X} \to \mathbb{R}$, with $u = u(x,y,z) = u(x(\xi,\eta,\kappa), y(\xi,\eta,\kappa), z(\xi,\eta,\kappa))$ and hence $u = u(\xi,\theta,\kappa)$, the chain rule implies

$$\begin{aligned} u_\xi &= u_x x_\xi + u_y y_\xi + u_z z_\xi \\ u_\eta &= u_x x_\eta + u_y y_\xi + u_z z_\eta \\ u_\kappa &= u_x x_\kappa + u_y y_\xi + u_z z_\kappa \end{aligned}$$

or equivalently,

$$\begin{bmatrix} u_\xi \\ u_\eta \\ u_\kappa \end{bmatrix} = \begin{bmatrix} x_\xi & y_\xi & z_\xi \\ x_\eta & y_\eta & z_\eta \\ x_\kappa & y_\kappa & z_\kappa \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = J^T \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$$

Hence

$$\begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = (J^T)^{-1} \begin{bmatrix} u_\xi \\ u_\eta \\ u_\kappa \end{bmatrix}.$$

Since

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} = \frac{1}{\Delta} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ac - bd \end{bmatrix},$$

where

$\Delta = a(ei - fh) - b(di - fg) + c(dh - eg).$

If one denotes

$$J^T = \begin{pmatrix} (1) = x_\xi & (2) = y_\xi & (3) = z_\xi \\ (4) = x_\eta & (5) = y_\eta & (6) = z_\eta \\ (7) = x_\kappa & (8) = y_\kappa & (9) = z_\kappa \end{pmatrix}$$

then

$$(J^T)^{-1} = \frac{1}{\Delta} \begin{pmatrix} (5)(9) - (6)(8) & (3)(8) - (2)(9) & (2)(6) - (3)(5) \\ (6)(7) - (4)(9) & (1)(9) - (3)(7) & (3)(4) - (1)(6) \\ (4)(8) - (5)(7) & (2)(7) - (1)(8) & (1)(5) - (2)(4) \end{pmatrix}$$

with

$\Delta = (1)((5)(9) - (6)(8)) - (2)((4)(9) - (6)(7)) + (3)((4)(8) - (5)(7)).$

If one uses the gradient to approximate the partial derivatives of the Jacobian, then

$$J_G^T = I_{xyz}^{F \to C} \tilde{G}_{\xi\eta\kappa}$$

where $\tilde{G}_{xyz}$ is the same as G_{xyz} with $\hat{I}_p$ replaced by I_{p+2}, the identity matrix of order $p + 2$. If one computes the Jacobian at the centers then the physical gradient is given by

$$G_{xyz} = I_{xyz}^{C \to F} (J_G^T)^{-1} I_{\xi\eta\kappa}^{F \to C} G_{\xi\eta\kappa}.$$

Similarly, one can construct the Jacobian based on the divergence operator.

6.5 Non-Uniform Gradient and Divergence Operators

Author: Miguel A. Dumett
Date: 2025-03-23
Abstract: This document provides formulas for the mimetic difference gradient and divergence operators for non-uniform one-dimensional meshes.

6.5.1 Introduction

On an interval $[a, b]$, consider n equal size subintervals, each of length $h = \frac{b-a}{n}$.

Then

- the uniform node grid (with $n + 1$ points) is given by

$$X_N^u = \{a, a + h, \cdots, b - h, b\},$$

- the uniform center grid (with $n + 2$ points) is given by

$$X_C^u = \{a, a + \frac{h}{2}, a + \frac{3h}{2}, \cdots, b - \frac{3h}{2}, b - \frac{h}{2}, b\}$$

Suppose a non-uniform grid on interval $[a, b]$, with n non-equal subintervals, is given by

- the set of $n + 1$ nodes of the non-uniform grid,

$$X_N = \{x_0 = a < x_1 < x_2 < \cdots < x_{n-1} < x_n = b\},$$

- and the corresponding $n + 2$ non-uniform centers,

$$X_C = \{y_0 = a < y_1 < y_2 < \cdots < y_n < y_{n+1} = b\}, \qquad y_i = (x_i + x_{i-1})/2, \quad i = 1, \cdots, n.$$

Then

1. In 1D, the non-uniform gradient G_{nu} in terms of the uniform gradient G_u is given by

$$G_{nu} = \text{diag}((G_u X_C)^{-1})\, G_u,$$

2. and the 1D non-uniform divergence D_{nu} in terms of the uniform divergence D_u is given by

$$D_{nu} = \text{diag}((D_u X_N)^{-1})\, D_u.$$

   Since the first and last rows of D_u are zero then the vector $D_u X_N$ will have zeros in its first and last component and hence it will not be possible to compute the inverses of both components. To avoid these infinity values, one substitutes the first and last components of $D_u X_N$ by ones.

6.6 Mimetic Discretization of the Integration by Parts Formula

The divergence theorem states that

$$\int_\Omega (\nabla \cdot \mathbf{u})\, q\, dV = \int_{\partial\Omega} q\, (\mathbf{u} \cdot \mathbf{n})\, ds - \int_\Omega \mathbf{u} \cdot \nabla q\, dV \qquad (1)$$

where the boundary integral represents the flux through $\partial\Omega$.

In one dimension, this reduces to the familiar integration by parts (IBP) formula:

$$\int_a^b u'(x)\, q(x)\, dx = \big[u(x)\, q(x)\big]_a^b - \int_a^b u(x)\, q'(x)\, dx \qquad (2)$$

where the boundary term is $\big[u(x)\, q(x)\big]_a^b = u(b)q(b) - u(a)q(a)$.

If the boundary term vanishes (e.g., homogeneous Dirichlet or periodic boundary conditions), we obtain the following

$$\int_a^b u'(x)\, q(x)\, dx = -\int_a^b u(x)\, q'(x)\, dx \qquad (3)$$

and let

$$u_h \in \mathcal{F}_h, \qquad q_h \in \mathcal{C}_h$$

be discrete fields, where $\mathcal{F}_h$ denotes the discrete space associated with face-centered (vector) quantities, and $\mathcal{C}_h$ the space associated with cell-centered (scalar) quantities.

Define the one-dimensional mimetic operators ***divergence*** and ***gradient***:

$$D : \mathcal{F}_h \to \mathcal{C}_h, \qquad G : \mathcal{C}_h \to \mathcal{F}_h$$

The weighted inner products on $\mathcal{F}_h$ and $\mathcal{C}_h$ are induced by diagonal, positive-definite matrices P and Q, respectively.

Then, the discrete analog of (3) is given by

$$\langle Du_h, q_h \rangle_Q = -\langle u_h, Gq_h \rangle_P, \qquad \forall\, u_h, q_h$$

or, in matrix form,

$$(Du_h)^T Q\, q_h = -\, u_h^T P\, (Gq_h)$$

The example integration1D.m illustrates how the weight matrix Q can be used to approximate the integral of a Ricker wavelet.

Additional operators will be documented in the future.

CHAPTER

SEVEN

EXAMPLES

The MOLE library contains many examples written in OCTAVE/MATLAB and C++. These examples span a broad range of partial differential equations (PDEs). Below are more technical explanations of the examples included in the library.

NOTE: The name for both OCTAVE/MATLAB and C++ will be the same. The files `elliptic1D.m` and `elliptic1D.cpp` solve the same differential equation explained here under Elliptic1D. There are many more OCTAVE/MATLAB examples, so if you cannot find a C++ example, it is only in the OCTAVE/MATLAB folder.

7.1 Elliptic Problems

Elliptic partial differential equations are used to model steady-state phenomena with no time dependence. Common examples include the Poisson equation and Laplace equation, which describe electrostatic potentials, steady-state heat distribution, and equilibrium fluid flow.

7.1.1 1D Elliptic Problems

These examples demonstrate the solution of elliptic equations in one dimension, including boundary value problems and various solution methods.

Elliptic1D

Solves the 1D Poisson equation with Robin boundary conditions.

$$\nabla^2 u(x) = f(x)$$

with $x \in [0, 1]$, and $f(x) = e^x$. The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with $a = 1$, $b = 1$, and $g = 0$, and

$$au(0) + b\frac{du(0)}{dx} = 0$$

$$au(1) + b\frac{du(1)}{dx} = 2e$$

This corresponds to the call to robinBC1D of `robinBC1D(k, m, dx, a, b)`.

This example is implemented in:

- MATLAB/ OCTAVE

- C++

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Add Scalar Boundary Conditions

Solves the 1D Poisson equation with Robin boundary conditions. This is the exact same problem as elliptic1D.m, with `addScalarBC1D` used instead of `addRobinBC`. The equation to solve is

$$-\nabla^2 u(x) = e^x$$

with $x \in [0, 1]$. The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with

$$1u(0) + 1\frac{du(0)}{dx} = 0$$

$$1u(1) + 1\frac{du(1)}{dx} = 2e$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [1, 1]$, $b = [1, 1]$ and $g = [0, 2e]$. Substituting these values in gives:

$$u(0) + \frac{du(0)}{dx} = 0$$

$$u(1) + \frac{du(1)}{dx} = 2e$$

The key difference is the implementation of the boundary condition operators. In elliptic1D, the RHS of the Robin operator is included on lines 26-28, yet in this example, the boundary conditions are set via the `addScalarBC1D` operator.

The true solution is

$$u(x) = e^x$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Homogenous Dirichlet Boundary Conditions

Solves the 1D Poisson equation with homogeneous Dirichlet boundary conditions.

$$-\nabla^2 u(x) = 1$$

with $x \in [0, 1]$. The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with

$$1u(0) + b\frac{du(0)}{dx} = 0$$

$$1u(1) + b\frac{du(1)}{dx} = 0$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [1, 1]$, $b = [0, 0]$ and $g = [0, 0]$. Substituting these values in gives:

$$u(0) = 0$$

$$u(1) = 0$$

The true solution is

$$u(x) = \frac{x(1-x)}{2}$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann

- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Left Dirichlet and Right Neumann Boundary Conditions

Solves the 1D Poisson equation with left Dirichlet and right Neumann boundary conditions.

$$-\nabla^2 u(x) = 1$$

with $x \in [0, 1]$. The boundary conditions are given by

$$a_n u + b_n \frac{du}{dx} = g_n$$

with the left hand side boundary condition (Neumann) satisfying

$$0u(0) + 1\frac{du(0)}{dx} = 0$$

and the right hand boundary condition (Dirichlet) satisfying

$$1u(1) + 0\frac{du(1)}{dx} = 0$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [0, 1]$, $b = [1, 0]$ and $g = [0, 0]$. Substituting these values in gives:

$$\frac{du(0)}{dx} = 0$$

$$u(1) = 0$$

The true solution is:

$$u(x) = \frac{1 - x^2}{2}$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Left Dirichlet and Right Robin Boundary Conditions

Solves the 1D Poisson equation with left Dirichlet and right Robin boundary conditions.

$$-\nabla^2 u(x) = \pi^2 \sin(\pi x)$$

with $x \in [0, 1]$.

The boundary conditions are given by

$$a_n u + b_n \frac{du}{dx} = g_n$$

with the left hand side boundary condition (Dirichlet) satisfying

$$1u(0) + 0\frac{du(0)}{dx} = 10$$

and the right hand boundary condition (Robin) satisfying

$$400u(1) + 1\frac{du(1)}{dx} = 15$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [1, 400]$, $b = [0, 1]$ and $g = [10, 15]$. Substituting these values in gives:

$$u(0) = 10$$

$$400u(1) + \frac{du(1)}{dx} = 15$$

The true solution is:

$$u(x) = \sin(\pi x) + \frac{\pi - 3985}{401}x + 10$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Pure Neumann Boundary Conditions

Solves the 1D Poisson boundary value problem with Neumann boundary conditions.

$$-\nabla^2 u(x) = x - \frac{1}{2}$$

with $x \in [0, 1]$.

The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with the left hand side boundary condition (Neumann) satisfying

$$0u(0) + 1\frac{du(0)}{dx} = 0$$

and the right hand boundary condition (Neumann) satisfying

$$0u(1) + 1\frac{du(1)}{dx} = 0$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [0, 0]$, $b = [1, 1]$ and $g = [0, 0]$. Substituting these values in gives:

$$\frac{du(0)}{dx} = 0$$

$$\frac{du(1)}{dx} = 0$$

The exact solution (with constant C) is:

$$u(x) = C + \frac{x^2}{4} - \frac{x^3}{6}$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Left Neumann Right Robin Boundary Conditions

Solves the 1D Poisson boundary value problem with left Neumann and right Robin boundary conditions.

$$-\nabla^2 u(x) = \pi^2 \sin(\pi x)$$

with $x \in [0, 1]$.

The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with the left hand side boundary condition (Neumann) satisfying

$$0u(0) + 1\frac{du(0)}{dx} = 10$$

and the right hand boundary condition (Robin) satisfying

$$400u(1) + 1\frac{du(1)}{dx} = 15$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [0, 400]$, $b = [1, 1]$ and $g = [10, 15]$. Substituting these values in gives:

$$\frac{du(0)}{dx} = 10$$

$$400u(1) + \frac{du(1)}{dx} = 15$$

The exact solution is:

$$u(x) = \sin(\pi x) + -(10 + \pi)x + \frac{402\pi + 4025}{400}$$

The example is taken from this paper

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Pure Robin Boundary Conditions

Solves the 1D Poisson boundary value problem with pure Robin boundary conditions.

$$-\nabla^2 u(x) = \pi^2 \sin(\pi x)$$

with $x \in [0, 1]$.

The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with the left hand side boundary condition (Robin) satisfying

$$-200u(0) + 1\frac{du(0)}{dx} = 10$$

and the right hand boundary condition (Robin) satisfying $\$400u(1) + 1\frac{du(1)}{dx} = 15\$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [-200, 400]$, $b = [1, 1]$ and $g = [10, 15]$. Substituting these values in gives:

$$-200u(0) + \frac{du(0)}{dx} = 10$$

$$400u(1) + \frac{du(1)}{dx} = 15$$

The exact solution is:

$$u(x) = \sin(\pi x) + \frac{35 - \pi}{403}x + \frac{402\pi - 3995}{80600}$$

The example is taken from this paper

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Non-Homogenous Dirichlet Boundary Conditions

Solves the 1D Poisson equation with non-homogeneous Dirichlet boundary conditions.

$$-\nabla^2 u(x) = 1$$

with $x \in [0, 1]$. The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with

$$1u(0) + 0\frac{du(0)}{dx} = \frac{1}{2}$$

$$1u(1) + 0\frac{du(1)}{dx} = \frac{1}{2}$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [1, 1]$, $b = [0, 0]$ and $g = [1/2, 1/2]$. Substituting these values in gives:

$$u(0) = \frac{1}{2}$$

$$u(1) = \frac{1}{2}$$

The true solution is

$$u(x) = \frac{-x^2 + x + 1}{2}$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions
- Non-Periodic Boundary Conditions

Elliptic1D Non-Periodic Dirichlet Boundary Conditions

Solves the 1D Poisson equation with homogeneous non-periodic Dirichlet boundary conditions.

(This is the same as Homogeneous Dirichlet example)

$$-\nabla^2 u(x) = 1$$

with $x \in [0, 1]$. The boundary conditions are given by

$$au + b\frac{du}{dx} = g$$

with

$$1u(0) + b\frac{du(0)}{dx} = 0$$

$$1u(1) + b\frac{du(1)}{dx} = 0$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. $a = [1, 1]$, $b = [0, 0]$ and $g = [0, 0]$. Substituting these values in gives:

$$u(0) = 0$$

$$u(1) = 0$$

The true solution is

$$u(x) = \frac{x(1-x)}{2}$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Periodic Boundary Conditions

Elliptic1D Periodic Dirichlet Boundary Conditions

Solves the 1D Poisson equation with periodic boundary conditions.

$$-\nabla^2 u(x) = 4\pi^2 \sin(2\pi x)$$

with $x \in [0, 1]$. The boundary conditions here are a special case, and periodicity is all that is required. Mathematically,

$$u(0) = u(1)$$

and

$$\frac{du(0)}{dx} = \frac{du(1)}{dx}$$

This corresponds to the call to addScalarBC1D of `addScalarBC1D(A,b,k,m,dx,dc,nc,v)`, where `dc`, `nc`, and `vc` are vectors which hold the coefficients for a, b, and g in the above system of equations. To request periodicity, the values must be all zeros. $a = [0, 0]$, $b = [0, 0]$ and $g = [0, 0]$.

This tells the MOLE library to build a 1D periodic boundary operator. This same logic is extended to 2 and 3 dimensions. A periodic boundary operator is returned if ALL of the values for the appropriate boundary vector values `a,b,g` are zero.

The true solution (where C is a constant) is

$$u(x) = \sin(2\pi x) + C$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants of this example with different boundary conditions:

- Homogeneous Dirichlet
- Non-Homogeneous Dirichlet
- Left Dirichlet, Right Neumann
- Left Dirichlet, Right Robin
- Left Neumann, Right Neumann
- Left Neumann, Right Robin
- Left Robin, Right Robin
- Non-Periodic Boundary Conditions

7.1.2 2D Elliptic Problems

These examples cover two-dimensional elliptic PDEs across various geometries, including examples on curved domains and specialized cases of the Poisson equation.

Elliptic2D

Solves the 2D Poisson equation with Robin boundary conditions on a nonuniform grid.

$$\nabla^2 u(x, y) = f(x, y)$$

with $x \in [0, 10], y \in [0, 10]$, and

$$f(x, y) = \begin{cases} (x - 0.5)^2 + (y - 0.5)^2 & \text{along boundaries} \\ 4 & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

with $a = 1$, $b = 0$, and $g = 0$, which is equivalent to Dirichlet conditions along each boundary. This corresponds to the call to robinBC2D of `robinBC2D(k, m, 1, n, 1, 1, 0)`.

Elliptic2D Case 2

Solves the 2D Poisson equation with Robin boundary conditions on a nonuniform sinusoidal grid.

$$\nabla^2 u(x, y) = f(x, y)$$

with $x \in [-\pi, 2\pi], y \in [-\pi, \pi]$, and

$$f(x, y) = \begin{cases} \sin(x)\sin(y) & \text{along boundaries} \\ -2\sin(x)\sin(y) & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

with $a = 1$, $b = 0$, and $g = 0$, which is equivalent to Dirichlet conditions along each boundary. This corresponds to the call to robinBC2D of `robinBC2D(k, m, 1, n, 1, 1, 0)`.

Elliptic2D Nodal Curv

Solves the 2D Poisson equation with Robin boundary conditions on a curvilinear grid using the nodal mimetic operator. This requires manually setting the boundary condition in the Laplacian, as there is no boundary condition operator for the nodal curvilinear operators.

$$\nabla^2 u(x, y) = f(x, y)$$

with $x \in [0, 50], y \in [0, 50]$, and

$$f(x, y) = \begin{cases} (x - 0.5)^2 + (y - 0.5)^2 & \text{along boundaries} \\ 4 & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

The MATLAB/ OCTAVE code uses the function `boundaryIdx2D` to find the correct locations for boundary condition weights in the nodal Laplacian. The code then sets the appropriate values to 0 or 1.

Elliptic2D Nodal Curv Sinusoidal

Solves the 2D Poisson equation with Robin boundary conditions on a curvilinear sinusoidal grid using the nodal mimetic operator. This requires manually setting the boundary condition in the Laplacian, as there is no boundary condition operator for the nodal curvilinear operators.

$$\nabla^2 u(x, y) = f(x, y)$$

with $x \in [-\pi, 2\pi], y \in [-\pi, \pi]$, and

$$f(x,y) = \begin{cases} \sin(x) + \cos(y) & \text{along boundaries} \\ -\sin(x) - \cos(y) & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

with $a = 1$, $b = 0$, and $g = 0$, which is equivalent to Dirichlet conditions along each boundary. The MATLAB/OCTAVE code uses the function `boundaryIdx2D` to find the correct locations for the weights of the boundary condition in the Laplacian node. The code then sets the appropriate values to 0 or 1.

Minimal Poisson 2D

Solves the 2D Poisson equation with Robin boundary conditions on a uniform grid where $\Delta x = \Delta y = 1$.

$$\nabla^2 u(x,y) = f(x,y)$$

with $x \in [0, 5], y \in [0, 5]$, and

$$f(x,y) = \begin{cases} 100 & \text{if y = 0} \\ 0 & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

with $a = 1$, $b = 0$, and $g = 0$. This corresponds to the call to robinBC2D of `robinBC2D(k, m, 1, n, 1, 1, 0)`.

7.1.3 3D Elliptic Problems

This section demonstrates three-dimensional elliptic partial differential equations, showing how to set up and solve problems in 3D space.

Elliptic3D

Similar to `minimal_poisson2D`, this solves a 3D problem using mimetic Laplacian, where one side of the domain is set to 100, and allowed to diffuse.

$$\nabla^2 u(x,y,z) = f(x,y,z)$$

with $x \in [0, 5], y \in [0, 6], z \in [0, 7]$, and

$$f(x,y,z) = \begin{cases} 100 & \text{if z = 0} \\ 0 & \text{otherwise} \end{cases}$$

The boundary conditions are given by

$$au + b\nabla u = g$$

with $a = 1$, $b = 0$, and $g = 0$. This corresponds to the call to robinBC3D of `robinBC3D(k, m, 1, n, 1, o, 1, 1, 0)`.

7.2 Parabolic Problems

Parabolic PDEs describe diffusion processes where initial discontinuities are immediately smoothed out. The heat equation is a classic example, modeling how temperature distributes over time in a medium.

7.2.1 1D Parabolic Problems

These examples demonstrate one-dimensional parabolic equations, focusing on heat/diffusion problems and their numerical solutions.

1D Heat Equation

This example solves the one-dimensional heat equation with Dirichlet boundary conditions, which is a classic parabolic PDE:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

where T is the temperature and α is the thermal diffusivity.

Domain and Boundary Conditions

The domain is $x \in [0, 1]$ with Dirichlet boundary conditions:

- $T(0, t) = 100$
- $T(1, t) = 100$

Discretization

The spatial discretization uses the mimetic laplacian operator with a specified order of accuracy k. The temporal discretization can be either:

1. Explicit (forward Euler): $T^{n+1} = T^n + \alpha \Delta t L T^n$
2. Implicit (backward Euler): $T^{n+1} = (I - \alpha \Delta t L)^{-1} T^n$

where L is the mimetic discrete Laplacian operator.

The time step is constrained by the stability condition for the explicit scheme: $\$\Delta t \leq \frac{\Delta x^2}{3\alpha}\$$

This example is implemented in:

- MATLAB/OCTAVE
- C++

Results

The solution shows the heat diffusing through the domain, with the temperature at the boundaries held constant at 100. The explicit scheme is conditionally stable, requiring a small time step, while the implicit scheme is unconditionally stable but requires solving a linear system at each time step.

Terzaghi One-Dimensional Consolidation

Simulates **Terzaghi's 1D consolidation** using mimetic finite difference operators from the MOLE library. The system models **transient flow and deformation** in a saturated soil column under a **constant compressive load** at one end with **drainage permitted only at the loaded boundary**.

The governing pressure equation is:

$$\frac{\partial p}{\partial t} = c_f \nabla^2 p$$

with $x \in [0, 25]$ meters. Displacement and strain are derived from the pressure, and Darcy's law is used to compute fluid flux.

Boundary conditions:

- **Dirichlet** at $x = 0$: $p(0, t) = 0$
- **Neumann** at $x = L$: $\dfrac{dp}{dx}(L, t) = 0$ This setup models **open drainage** at the loaded face and **no flow** at the fixed base.

This corresponds to a domain with **impermeable backing** and **open drainage at the loaded end**.

Numerical Strategy

- Pressure is initialized to a uniform value $P_0 = 10\ \mathrm{MPa}$
- Integration is performed using **Forward Euler**
- Mimetic MOLE operators:
  - `lap()` for pressure diffusion
  - `grad()` for Darcy flux
  - `div()` for residual calculations
- Spatial discretization uses a **staggered grid** with ghost cells to enforce boundary conditions

Analytical Benchmark

An analytical solution is computed using a **Fourier series expansion**:

$$p(x,t) = \sum_{n=0}^{\infty} \left(\frac{4P_0}{n\pi} \sin\left(\frac{n\pi x}{2L}\right) e^{-\frac{n^2\pi^2 c_f t}{4L^2}} \right), \quad n = 2k+1$$

The benchmark solution includes:

- Pressure field
- Flux via Darcy's law
- Strain and displacement
- Mass conservation residual

Outputs

At selected time snapshots (1, 10, 40, 70 hours), the following are printed and plotted:

- **Numerical and analytical pressure** profiles
- **Darcy flux** from numerical and analytical solutions
- **Displacement fields**
- **Mass balance residuals**
- **Relative L2 error** tables
- **3D surface plots** for pressure, displacement, and residual evolution

Physical Parameters

Parameter	Value	Description
P_0	10 MPa	Face load
c_f	1×10^{-4}	Diffusivity
K	$1 \times 10^{-12}\,\mathrm{m}^2$	Permeability
μ	$1 \times 10^{-3}\,\mathrm{Pa} \cdot \mathrm{s}$	Dynamic viscosity
K_s	$1 \times 10^{8}\,\mathrm{Pa}$	Bulk modulus
α	1.0	Biot coefficient
S_s	$1 \times 10^{-5}\,\mathrm{Pa}^{-1}$	Specific storage coefficient
ρ	$1000\,\mathrm{kg/m}^3$	Fluid density
g	$9.81\,\mathrm{m/s}^2$	Gravitational acceleration

Code Location

This example is implemented in:

- MATLAB/ OCTAVE (terzaghi1D.m)

7.3 Hyperbolic Problems

Hyperbolic PDEs model wave propagation and transport phenomena where information travels along characteristic curves. Examples include the wave equation, transport equation, and conservation laws such as Burgers' equation.

7.3.1 1D Hyperbolic Problems

These examples demonstrate one-dimensional hyperbolic equations, including wave propagation, transport problems, and numerical methods like upwind schemes and Lax-Friedrichs.

Hyperbolic1D

Solves the 1D advection equation with periodic boundary conditions.

$$\frac{\partial U}{\partial t} + a\frac{\partial U}{\partial x} = 0$$

where $U = u(x,t)$ and $a = 1$ is the advection velocity. The domain $x \in [0,1]$ and $t \in [0,1]$ with initial condition

$$u(x,0) = \sin(2\pi x)$$

Periodic boundary conditions are used

$$u(0,t) = u(1,t)$$

Using finite differences for the time derivative

$$\frac{\partial U}{\partial t} = \frac{U_i^{n+1} - U_i^n}{\Delta t}$$

where U_i^n is $u(x_i, t_n)$, and the mimetic operator $\mathbf{D}$ for the space derivative.

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} + a\mathbf{D}U_i^n = 0$$

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = -a\mathbf{D}U_i^n$$

$$U_i^{n+1} = U_i^n - a\Delta t \mathbf{D} U_i^n$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants:

- Upwind Scheme
- Lax-Friedrichs Scheme

Hyperbolic1D upwind

Solves the 1D advection equation with periodic boundary conditions using a sided nodal mimetic operator. The main feature of this code is the lack of an interpolator.

$$\frac{\partial U}{\partial t} + a\frac{\partial U}{\partial x} = 0$$

where $U = u(x,t)$ and $a = 1$ is the advection velocity. The domain $x \in [0,1]$ and $t \in [0,1]$ with initial condition

$$u(x,0) = \sin(2\pi x)$$

Periodic boundary conditions are used

$$u(0,t) = u(1,t)$$

Using finite differences for the time derivative

$$\frac{\partial U}{\partial t} = \frac{U_i^{n+1} - U_i^n}{\Delta t}$$

where U_i^n is $u(x_i, t_n)$, and the mimetic operator $\mathbf{D}$ for the space derivative.

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} + a\mathbf{D}U_i^n = 0$$

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = -a\mathbf{D}U_i^n$$

$$U_i^{n+1} = U_i^n - a\Delta t \mathbf{D} U_i^n$$

$$U_i^{n+1} = U_i^n (I - S)$$

The last line $I - S$ is the identity matrix I minus the premultiplied $a\Delta t\mathbf{D}$.

Hyperbolic1D Lax Friedrichs

Solves the 1D advection equation with periodic boundary conditions using a sided nodal mimetic operator and the Lax-Friedrichs scheme for time integration. The main feature of this code is the lack of an interpolator. This is first order in both time and space, even though a second order mimetic operator is used.

$$\frac{\partial U}{\partial t} + a\frac{\partial U}{\partial x} = 0$$

where $U = u(x,t)$ and $a = 1$ is the advection velocity. The domain $x \in [0,1]$ and $t \in [0,1]$ with initial condition

$$u(x,0) = \sin(2\pi x)$$

Periodic boundary conditions are used

$$u(0,t) = u(1,t)$$

Using finite differences for the time derivative

$$\frac{\partial U}{\partial t} = \frac{U_i^{n+1} - U_i^n}{\Delta t}$$

where U_i^n is $u(x_i, t_n)$, and the mimetic operator $\mathbf{D}$ for the space derivative.

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} + a\mathbf{D}U_i^n = 0$$

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = -a\mathbf{D}U_i^n$$

$$U_i^{n+1} = \mathbf{U}_i^n - a\Delta t\mathbf{D}U_i^n$$

The last line $\mathbf{U}$ is the derived (average) value of the U from the U^n timestep.

Wave 1D

Solves the one-way wave equation using the position Verlet or Forest-Ruth algorithms.

$$\frac{\partial^2 U}{\partial t^2} - c^2 \frac{\partial^2 U}{\partial x^2} = 0$$

where $U = u(x,t)$ defined on the domains $x \in [0,1]$ and $t \in [0,1]$, and wave speed $c = 2$. Initial position and velocity are given as

$$u(x,0) = \sin(\pi x)$$

$$u'(x,0) = 0$$

This example is implemented in:

- MATLAB/ OCTAVE

Additional MATLAB/ OCTAVE variants:

- Wave 1D Case 2
- Wave 1D with Time-Varying BC

Wave 1D Case 2

Solves the one-way wave equation using the position Verlet or Forest-Ruth algorithms with higher wave speed.

$$\frac{\partial^2 U}{\partial t^2} - c^2 \frac{\partial^2 U}{\partial x^2} = 0$$

where $U = u(x,t)$ defined on the domains $x \in [0,1]$ and $t \in [0,0.06]$, and wave speed $c = 100$. Initial position and velocity are given as

$$u(x,0) = \begin{cases} \sin(\pi x) & 2 < x < 3 \\ 0 & \text{otherwise} \end{cases}$$

$$u'(x,0) = 0$$

Burgers1D

This example deals with the conservative form of the inviscid Burgers equation in 1D.

$$\frac{\partial U}{\partial t} + \frac{\partial}{\partial x}\left(\frac{U^2}{2}\right) = 0$$

with $U = u(x,t)$ defined on the domain $x \in [-15, 15]$, from time $t \in [0, 10]$ and initial conditions

$$u(x,0) = e^{\frac{-x^2}{50}}$$

The wave is allowed to propagate across the domain while the area under the curve is calculated.

This example is implemented in:

- MATLAB/ OCTAVE
- C++

7.3.2 2D Hyperbolic Problems

These examples illustrate two-dimensional wave propagation and related hyperbolic phenomena.

Wave 2D

Solves the two-dimensional wave equation using the position Verlet algorithm.

$$\frac{\partial^2 U}{\partial t^2} - c^2 \frac{\partial^2 U}{\partial x^2} = 0$$

where $U = u(x,y,t)$ defined on the domains $x \in [0,1], y \in [0,1]$ and $t \in [0,1]$, and wave speed $c = 1$. The boundaries are Dirichlet. Initial position and velocity are given as

$$u(x,y,0) = \sin(\pi x)\sin(\pi y)$$

$$u'(x,y,0) = 0$$

Wave 2D Case 2

Solves the two-dimensional wave equation using the position Verlet algorithm. This example uses the fourth order mimetic Laplacian, and extends the domain

$$\frac{\partial^2 U}{\partial t^2} - c^2 \frac{\partial^2 U}{\partial x^2} = 0$$

where $U = u(x,y,t)$ defined on the domains $x \in [-5, 10], y \in [-5, 10]$ and $t \in [0, 0.3]$, and wave speed $c = 100$. The boundaries are Dirichlet. Initial position and velocity are given as

$$u(x,y,0) = \begin{cases} \sin(\pi x)\sin(\pi y) & 2 < x < 3,\ 2 < y < 3 \\ 0 & \text{otherwise} \end{cases}$$

$$u'(x,y,0) = 0$$

7.4 Schrödinger Problems

The Schrödinger equation is a fundamental equation in quantum mechanics that describes how the quantum state of a physical system changes over time. These examples demonstrate numerical approaches to solving this important equation.

7.4.1 1D Schrödinger Problems

These examples demonstrate one-dimensional quantum systems modeled by the Schrödinger equation, including time-dependent and time-independent cases.

Schrodinger1D

Solves the 1D time-independent Schrödinger equation.

$$H\psi = E\psi$$

where H is the Hamiltonian operator, ψ is the wave function, and E is the energy. The Hamiltonian includes the kinetic energy term represented by the Laplacian and a potential energy term.

This example is implemented in:

- C++

7.5 Sturm-Liouville Problems

Sturm-Liouville theory deals with a class of second-order linear differential equations with important applications in physics and engineering. These problems yield eigenvalues and eigenfunctions that can be used to solve more complex PDEs through spectral methods.

7.5.1 Chebyshev Sturm-Liouville Problem

This example solves the Chebyshev differential equation, which is a classic Sturm-Liouville problem:

$$(1 - x^2)u'' - xu' + n^2u = 0, \quad -1 < x < 1$$

with Dirichlet boundary conditions: $$u(-1) = 1, \quad u(1) = 1$$

The exact solution to this problem is the Chebyshev polynomial of the first kind of degree n, denoted as $T_n(x)$. For $n = 2$, the solution is $T_2(x) = 2x^2 - 1$.

Mathematical Background

Chebyshev's differential equation is a special case of the Sturm-Liouville problem, which has the general form:

$$\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u + \lambda r(x)u = 0$$

For Chebyshev's equation, we have:

- $p(x) = 1 - x^2$
- $q(x) = 0$
- $r(x) = 1$
- $\lambda = n^2$

Discretization

The equation is discretized using mimetic finite difference operators. The spatial derivative operators are constructed with a specified order of accuracy k.

The discrete system is:

$$Au = b$$

where:

- $A = (1 - x^2)L - xIG + n^2 I$
- L is the mimetic Laplacian
- G is the mimetic gradient
- I is the interpolation operator from faces to centers

Boundary conditions are applied using the `addScalarBC1D` function.

This example is implemented in:

- MATLAB/OCTAVE

Results

The numerical solution closely matches the exact solution, which is the Chebyshev polynomial $T_2(x) = 2x^2 - 1$.

Chebyshev polynomials are important in numerical analysis and approximation theory because they:

1. Minimize the maximum error in polynomial approximation
2. Have roots that are optimal interpolation points (Chebyshev nodes)
3. Are closely related to the Fourier cosine series

7.6 Navier-Stokes Problems

The Navier-Stokes equations describe the motion of viscous fluid substances, forming the foundation of fluid dynamics. These examples demonstrate numerical approaches to solving these complex, nonlinear PDEs.

7.6.1 Lock Exchange Problem

This example simulates a 2D lock exchange problem using mimetic methods. The lock exchange is a classic fluid dynamics problem where two fluids of different densities initially separated by a vertical barrier (lock) are allowed to flow into each other when the barrier is removed. This creates complex flow patterns including gravity currents and Kelvin-Helmholtz instabilities.

Governing Equations

The simulation uses the Boussinesq approximation, which accounts for density variations only in the buoyancy term:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho_0}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g}\alpha(T - T_0)$$

$$\nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \kappa \nabla^2 T$$

where:

- $\mathbf{u}$ is the velocity field
- p is the pressure
- T is the temperature
- ρ_0 is the reference density
- ν is the kinematic viscosity
- α is the thermal expansion coefficient
- $\mathbf{g}$ is the gravitational acceleration
- T_0 is the reference temperature

Domain and Initial Conditions

The simulation is conducted on a rectangular domain $[0, 100] \times [0, 20]$ meters. Initially, fluids of different densities (or temperatures) are separated at the mid-point of the domain with a narrow transition region.

Numerical Method

The equations are solved using a fractional step method (projection method):

1. **Predictor Step**: Compute an intermediate velocity field $\mathbf{u}^*$ without enforcing incompressibility
2. **Pressure Solution**: Solve a Poisson equation for pressure to enforce incompressibility
3. **Corrector Step**: Project the velocity field to be divergence-free
4. **Temperature Advection**: Update temperature using upwind/downwind differencing

Spatial discretization uses mimetic operators:

- Divergence operator (D)
- Gradient operator (G)
- Laplacian operator ($L = DG$)

This example is implemented in:

- MATLAB/OCTAVE
- C++

Results

The simulation captures the gravity currents that form as the denser fluid flows beneath the lighter fluid. Depending on the Reynolds number, Kelvin-Helmholtz billows may develop along the interface. Due to the relatively coarse grid, numerical diffusion limits the formation of sharp billows, but increasing the resolution would allow for more detailed structures to emerge.

7.7 Mixed Problems

Mixed problems combine characteristics of different PDE types (elliptic, parabolic, hyperbolic) or couple multiple physical phenomena. Convection-diffusion equations are a classic example, combining transport (hyperbolic) with diffusion (parabolic).

7.7.1 1D Mixed Problems

Transport1D

Solves the 1D advection-reaction-dispersion equation:

$$\frac{\partial C}{\partial t} + v\frac{\partial C}{\partial x} = D\frac{\partial^2 C}{\partial x^2}$$

where C is the concentration, v is the pore-water flow velocity, and D is the dispersion coefficient.

This example is implemented in:

- C++

7.7.2 2D Mixed Problems

Work in Progress

Check back soon for specific implementations and examples.

7.7.3 3D Mixed Problems

3D Convection-Diffusion Equation

This example solves the three-dimensional convection-diffusion equation using mimetic methods. The convection-diffusion equation is a parabolic partial differential equation that describes physical phenomena where particles, energy, or other physical quantities are transferred inside a physical system due to two processes: diffusion and convection.

Mathematical Model

The convection-diffusion equation has the form:

$$\frac{\partial C}{\partial t} + \nabla \cdot (\mathbf{v}C) = \nabla \cdot (D\nabla C)$$

where:

- C is the concentration (or density)
- $\mathbf{v}$ is the velocity field
- D is the diffusion coefficient

This equation combines:

- The diffusion term: $\nabla \cdot (D\nabla C)$
- The convection term: $\nabla \cdot (\mathbf{v}C)$

Application Context

This example simulates CO_2 transport in a geological formation with impermeable shale layers. The simulation represents CO_2 injection through a well, and its subsequent movement through porous media with varying permeability. This type of simulation is important for:

- Carbon capture and storage (CCS) studies
- Underground contaminant transport
- Enhanced oil recovery analysis

Numerical Method

The equation is solved using operator splitting with:

1. A Forward-Time Central-Space (FTCS) scheme for the diffusion term
2. An upwind scheme for the convection term

Mimetic operators are used for spatial discretization:

- Divergence operator (D)
- Gradient operator (G)
- Interpolation operator (I)

Time step constraints include:

- von Neumann stability criterion for diffusion: $\Delta t \leq \frac{\Delta x^2}{3D}$
- CFL condition for convection: $\Delta t \leq \frac{\Delta x}{\max(|\mathbf{v}|)}$

This example is implemented in:

- MATLAB/OCTAVE
- C++

Results

The simulation shows how CO_2 spreads through the domain, with the shale layers acting as barriers to flow. The concentration profile evolves over time due to:

1. Molecular diffusion (spreading in all directions)
2. Advective transport (preferential movement in the direction of flow)
3. Reduced transport through low-permeability layers

This type of simulation is valuable for understanding subsurface fluid dynamics and designing effective carbon storage strategies.

7.8 Integration Examples

While mimetic operators are generally used for solving PDEs, their properties extend to all parts of calculus, including integration.

7.8.1 Integration1D.m

Ricker Wavelet Propagation using Mimetic Quadrature Weights

This program solves the 1D acoustic wave equation using *mimetic operators*. The governing equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} + f(t)$$

where:

- $u(x,t)$ = wavefield (displacement or pressure)
- c = wave propagation speed (can be constant or spatially varying)

- $f(t)$ = source term (Ricker wavelet in time)

The Ricker wavelet source is defined as:

$$f(t) = (1 - 2\pi^2 f_0^2 (t - t_0)^2) * exp(-\pi^2 f_0^2 (t - t_0)^2)$$

where f_0 is the dominant frequency and t_0 is the source time delay.

For this particular problem we use the function

$$f(t) = (1 - x^2) exp(\frac{-x^2}{2})$$

MIMETIC DISCRETIZATION

In the mimetic framework, spatial derivatives are represented by discrete gradient (G) and divergence (D) operators that satisfy a discrete analogue of the integration-by-parts identity:

$$\langle v, Du \rangle_Q + \langle Gv, u \rangle_P = boundary\, terms$$

where $\langle a, b \rangle_Q = a^T Q b$ defines an inner product weighted by Q.

Here:

- Q : Diagonal matrix of quadrature weights at *cell centers*
- P : Diagonal matrix of quadrature weights at *cell faces*

Both Q and P are positive definite diagonal matrices. Their dimensions are chosen so that the following operations are valid:

```
Q * D        (divergence operator in cell-centered space)
P * G        (gradient operator in face-centered space)
G' * P       (adjoint of P*G, equivalent to -D for closed boundaries)
```

The *mimetic boundary operator* is defined as:

```
B = Q * D + G' * P
```

This ensures that the discrete operators exactly satisfy conservation laws and reproduce the divergence theorem on the computational grid.

NUMERICAL INTEGRATION

The second spatial derivative $\partial^2 u / \partial x^2$ is obtained through the mimetic Laplacian:

```
L = D * (P¹ * G)
```

so that the discrete form of the wave equation becomes:

```
Q * (d²u/dt²) = c² * Q * L * u + Q * f
```

Since Q is diagonal, it acts as a discrete mass matrix that defines the quadrature weights for integration over the computational domain.

In this implementation, we use the weights from Q explicitly for the numerical integration step. We have boundary conditions such that the only term of interest is the Q * f term. P and G are still conceptually part of the mimetic framework but are not directly required for this reduced problem.

ALGORITHM OVERVIEW

1. Define grid spacing and boundary conditions.
2. Define the function to integrate
3. Approximate the integral weights Q
4. Set the boundary conditions at the ends
5. Multiply weights * f to get estimate of the integral
6. Compare to MATLAB trapz and integral functions

This example is implemented in:

- MATLAB/OCTAVE

7.9 Time Integrators

Time integration methods are essential for solving time-dependent PDEs, converting them to systems of ODEs that are solved numerically. These examples demonstrate various time-stepping techniques and their properties.

7.9.1 Second-Order Runge-Kutta Method (RK2)

This example demonstrates the second-order Runge-Kutta (RK2) method for solving ordinary differential equations (ODEs). The method is implemented to solve a first-order ODE of the form:

$$\frac{dy}{dt} = f(t, y) = \sin^2(t) \cdot y$$

with initial condition $y(0) = 2.0$ over the time interval $[0, 5]$.

Method Description

The RK2 method (also known as the midpoint method) uses two stages to compute each time step:

1. First stage (slope at the beginning):

$$k_1 = f(t_i, y_i)$$

2. Second stage (slope at midpoint):

$$k_2 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1)$$

3. Solution update:

$$y_{i+1} = y_i + h \cdot k_2$$

where:

- h is the step size (set to 0.1 in the examples)

- t_i is the current time
- y_i is the solution at time t_i

This example is implemented in:

- MATLAB/OCTAVE
- C++

Both implementations include visualization of the solution using plotting tools (MATLAB's built-in plot function and GNUplot for C++).

7.9.2 4th-Order Runge-Kutta Method (RK4)

This example demonstrates the classic 4th-order Runge-Kutta method (RK4) for solving ordinary differential equations (ODEs). The RK4 method is a widely used numerical integration technique that provides a good balance of accuracy, stability, and computational efficiency.

Mathematical Background

The RK4 method solves initial value problems of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

The method advances the solution from t_n to $t_{n+1} = t_n + h$ using the formula:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where:

$$\begin{aligned} k_1 &= f(t_n, y_n) && (7.1)\\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) && (7.2)\\ k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) && (7.3)\\ k_4 &= f(t_n + h, y_n + hk_3) && (7.4) \end{aligned}$$

The method is 4th-order accurate, meaning the local truncation error is $O(h^5)$, and the global error is $O(h^4)$.

Example Problem

In this example, we solve the ODE:

$$\frac{dy}{dt} = \sin^2(t) \cdot y, \quad y(0) = 2$$

This is a linear ODE with a time-varying coefficient. The exact solution can be found by separation of variables:

$$y(t) = y_0 \exp\left(\int_0^t \sin^2(s)ds\right) = y_0 \exp\left(\frac{t}{2} - \frac{\sin(2t)}{4}\right)$$

This example is implemented in:

- MATLAB/OCTAVE
- C++

Properties of RK4

The RK4 method has several important properties:

1. **Accuracy**: 4th-order convergence means that halving the step size reduces the error by a factor of approximately 16.
2. **Stability**: The RK4 method has a relatively large stability region, making it suitable for many non-stiff problems.
3. **Self-starting**: Unlike multi-step methods, RK4 doesn't require special starting procedures or values from previous steps.
4. **Function Evaluations**: RK4 requires four function evaluations per step, which is more expensive than simpler methods like Euler's method but often justified by the improved accuracy.

Applications in PDE Solving

While this example demonstrates RK4 for an ODE, the method is often used in the time integration of PDEs after spatial discretization (method of lines). In the MOLE library, RK4 can be combined with mimetic operators for spatial discretization to create high-order accurate PDE solvers.

7.10 Tutorials

Tutorials for using the mimetic operators including comparison with other standard ways of solving systems of PDEs. These examples are heavily commented, and show how standard workflows could be modified to use mimetic operators.

7.10.1 Mimetic Difference Operators Vs Finite Differences

Solving the 1D Two-Way Wave Equation

This tutorial demonstrates how to solve the **1D two-way wave equation** using both **standard finite differences (FD)** and **mimetic finite differences (MD)**.
We will use three main functions:

1. *finite_diff_two_way_wave_eq*
2. *mimetic_diff_two_way_wave_eq*
3. *comparison_two_way_wave_md_vs_fd*

The goal is to understand the **differences in accuracy, stability, and numerical properties** between traditional finite and mimetic difference operators. The two numerical functions have been created to be as similar as possible, so that only the changes necessary for mimetic difference operators are visible.

While each code can be run independanlty, the easiest way to compare the two methods is with `comparison_two_way_wave_md_vs_fd`

Overview

The **1D two-way wave equation** is given by:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in [-2, 2] \tag{1}$$

Both solvers use a **centered-in-time, centered-in-space (CTCS)** scheme, which is **second-order accurate in both time and space**.

We assume no boundary effects (large domain), allowing a clean comparison of the spatial discretization methods. Here is a little bit more information about the two seperate functions.

`finite diff two way wave eq`

Purpose: This function solves the the wave equation using **standard finite differences (FD)**.

Time-stepping scheme (CTCS): If we discretize the equation with standard centered second order finite differences, Equation(1) turns into

$$\frac{U_i^{k-1} - 2U_i^k + U_i^{k+1}}{\Delta t^2} = c^2 \frac{U_{i+1}^k - 2U_i^k + U_{i+1}^k}{\Delta x^2}$$

Rearranging the equation for the unknown value U_i^{k+1} we get:

$$U_i^{k+1} = 2U_i^k - U_i^{k-1} + \frac{c^2 \Delta t^2}{\Delta x^2} \left(U_{i-1}^k - 2U_i^k + U_{i+1}^k \right)$$

Which can be expressed as a matrix D_{fd} times the vector U^k

$$U^{k+1} = 2U^k - U^{k-1} + D_{fd} U^k$$

The matrix is written out in the code using sparse matrix operations and is equivalent to:

$$D_{fd} = \frac{c^2 \Delta t^2}{\Delta x^2} \begin{bmatrix} -2 & 1 & 0 & \cdots \\ 1 & -2 & 1 & \cdots \\ 0 & 1 & -2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

The new values are updated each time step with values from the previous two time steps (k, k-1).

$$U^{k+1} = 2U^k - U^{k-1} + D_{fd} U^k$$

Function Outputs

Variable	Description
`U2_fd`	Final solution at the last time step
`error_fd`	Norm of error vs. analytic/reference solution
`walltime_fd`	Wall-clock time (seconds)
`flops_fd`	Estimated floating-point operation count

Notes

We are using an explicit two-step scheme (leapfrog). The CFL condition must be satisfied: $c\Delta t^2/\Delta x^2 \leq 1$. If you change the stepping, or increase the number of cells, be sure to change this value. No boundary conditions are applied, the domain is large enough to avoid reflection effects. This is to test just the spatial scheme without any boundary considerations.

`mimetic diff two way wave eq`

Purpose: Solve the wave equation using **mimetic difference operators (MD)**.

The mimetic Laplacian operator `L` replaces the finite-difference stencil, automatically enforcing discrete analogs of conservation and symmetry properties.

Time-stepping scheme: The mimetic operator directly takes the place fo the second derivative (Laplacian), therefore we only need to discretize the time derivative. Doing so to Equation (1) yields:

$$\frac{U_i^{k-1} - 2U_i^k + U_i^{k+1}}{\Delta t^2} = c^2 L\left(U^k\right)$$

Here, `L` is constructed from mimetic gradient (`G`) and divergence (`D`) operators such that:

$$L = D\,G$$

where

$$\mathbf{D} = \frac{1}{\Delta x}\begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} & 0 & \cdots & 0 \\ 1 & -1 & 0 & \cdots & 0 \\ 0 & 1 & -1 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

and

$$\mathbf{G} = \frac{1}{\Delta x}\begin{bmatrix} -\frac{1}{2} & 1 & 0 & 0 & \cdots & 0 & 0 \\ -\frac{1}{2} & -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & -1 & 1 & 0 & 0 \\ 0 & \cdots & 0 & 0 & -1 & -1 & 1 \\ 0 & \cdots & 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{2} \end{bmatrix}.$$

ensuring energy and flux consistency at the discrete level. The mimetic library Laplacian operator is just the composition DG under the hood.

Rearranging our equation for the unknown value U^{k+1} leads us to:

$$U_i^{k+1} = 2U_i^k - U_i^{k-1} + c^2\Delta t^2 L(U^k)$$

To save computations in the code, `L` is premultiplied by $C^2\Delta t^2$ before the computation loop.

Outputs

Variable	Description
`U2_md`	Final solution at the last time step
`error_md`	Norm of error vs. analytic/reference solution
`walltime_md`	Wall-clock time (seconds)
`flops_md`	Estimated floating-point operation count

Notes

This uses the same CTCS time integration scheme as FD. The **order of accuracy** can be increased by adjusting the `k` parameter (e.g. `k=2`, `k=4`, etc.). Mimetic operators preserve **discrete conservation laws**, often improving numerical stability and physical fidelity. If you change the number of cells, be sure to change the time step, to conform to the CFL condition.

comparison two way wave md vs fd

Purpose: Compare **mimetic** and **finite difference** results for the same PDE setup.

This script runs both solvers and measures differences in:

- Numerical accuracy
- Computational cost
- Wall-clock time
- Error convergence rate

Assorted plots are generated showing differences between the two methods.

Grid Comparison

The key difference between FD and MD methods lies in the spatial grid. Here is a slightly more involved explanation to help the user understand the difference in grids.

Finite Difference Grid (Uniform)

Each cell is the same width x between **A** and **B**, with points at cell boundaries (0,1,2,...N-1,N,N+1):

```
A                                                                   B
 <-----dx-----> <-----dx----->  ...  <-----dx-----> <-----dx----->
|----cell 1----|----cell 2----| ... |----cell N-1--|----cell N----|
0              1              2 ... N-1            N             N+1
```

For example, from 0 to 1 with 5 cells has six values (x0,x1,x2,x3,x4,x5) each 0.2 away from each other:

```
0.0   0.2   0.4   0.6   0.8   1.0
 o-----o-----o-----o-----o-----o
 x0    x1    x2    x3    x4    x5
```

Mimetic Difference Grid (Staggered)

Mimetic operators use a staggered grid—half-step offsets at boundaries improve conservation and symmetry. Here, note that **cell 1** and **cell N** are not the sam size as the internal cells, and that there are now **N+2** points.

```
A                                                       B
 <--dx/2--> <-----dx----->  ...  <-----dx-----> <--dx/2-->
|--cell 1--|----cell 2----| ... |----cell N-1--|--cell N--|
0          1              2 ... N             N+1        N+2
```

For the same interval (0–1) with 5 cells, a staggered grid will have smaller dt/s at the beginning and end, and also one mroe point (x0,x1,x2,x3,x4,x5,x6):

```
0.0   0.1   0.3   0.5   0.7   0.9   1.0
 o-----o-----o-----o-----o-----o-----o
 x0    x1    x2    x3    x4    x5    x6
```

This staggered layout provides **better discrete analogs** of differential operators, leading to improved conservation and often reduced numerical dispersion.

Convergence and Accuracy

Both FD and MD methods use a **second-order accurate** CTCS scheme in time.

Scheme	Spatial Order	Time Order	Conservation	Grid Type
Finite Difference	2	2	Approximate	Uniform
Mimetic Difference	2 (or higher)	2	Exact (discrete)	Staggered

---

Practical Notes

- This setup isolates **spatial discretization effects**—no boundary reflections or external forcing.
- Increasing `N` refines the grid and reduces spatial error.
- For fair comparison, both solvers should use the same `dt`, `dx`, and runtime length.
- The mimetic operator's flexibility allows easy order-of-accuracy experiments by simply changing `k` in `mimetic_diff_two_way_wave_eq.m`.

Summary

Feature	Finite Difference	Mimetic Difference
Grid	Uniform	Staggered
Operator	Explicit stencil	Discrete mimetic operator
Conservation	Approximate	Exact (discrete)
Flexibility	Fixed 2nd order	Adjustable order (k)
Accuracy	2nd order	2nd order
Ease of implementation	Simple	Slightly more complex
Best use case	Quick prototyping	Physically consistent PDE solvers

---

References

- Castillo, J. E., *Mimetic Methods for Partial Differential Equations*. Cambridge University Press, 2008.
- LeVeque, R. J., *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007.

More examples will be added in the future.

CHAPTER

EIGHT

MOLE OPEN-SOURCE ECOSYSTEM COMMUNITY ROLES

The MOLE Open-Source Ecosystem (OSE) organization places great value in its community, and community engagement is one of the pillars of the organization. The ***MOLE OSE Community*** comprises all MOLE stakeholders from the MOLE leadership team to all individuals interested in the MOLE functionalities and applications that use the MOLE library. Here, we define the different roles that members in the MOLE community play.

8.1 Table of Contents

8.2 MOLE Leadership Team

This group is composed of the library development founders, current lead developers, and software maintainers and managers of the MOLE OSE. The leadership team is responsible for maintaining the cohesiveness, robustness, security, reliability, succession plans, integrity, and quality of the software library.

The current members of the MOLE Leadership team are:

- Prof. Jose Castillo (founder),
- Dr. Johnny Delgado Corbino (founder),
- Prof. Valeria Barra,
- Dr. Jared Brzenski,
- Dr. Tony Drummond,
- Prof. Miguel Dumett,
- Dr. Giulia Pagallo,
- Prof. Chris Paolini,

In addition to the MOLE Leadership Team, the MOLE OSE organization counts with a **MOLE Steering Council** to facilitate the decision-making processes based on community consensus. The council members are formed by the leadership team members, chairs from the MOLE governing circles, MOLE founders, and strategic partners who financially support the MOLE OSE. For more details consult the MOLE OSE Organization page.

8.3 MOLE OSE Contributors

A contributor is anyone who has made contributions to the MOLE Library or the MOLE Open-Source Ecosystem. A full list of contributors is available in the Our Contributors Section in the MOLE repository. In addition to technical and conceptual contributions to the open-source software, the MOLE community includes contributions from industry, academic and professional connections that can also support the community through financial or managerial resources. Contributors help sustain the MOLE ecosystem and are invited to participate in the evaluation of the MOLE organizational pillars.

8.4 MOLE OSE Collaborators

Collaborators are a multidisciplinary group of members from industry and academia with whom the MOLE leadership team collaborates in the development of new computational applications using the MOLE library, or developers from other open-source ecosystems that interoperate with the MOLE library. Collaborators are also MOLE Contributors, and together with the leadership team work on the sustainability of the MOLE open-source ecosystem.

8.5 MOLE Library Users

Users are all members from industry and academia who are using MOLE to implement PDE Solvers. Users are a vital part of the MOLE Community, and their feedback is collected by the *MOLE Leadership team*, and used in strategic planning. MOLE Users are encouraged to become MOLE Contributors.

8.6 MOLE OSE Curious

Anyone with an interest in ***Mimetic Discretization*** and the fast prototyping of numerical solutions using MOLE from Matlab, C++ or Python. Individuals interested in knowing more about MOLE should sign up for our mailing lists, check the MOLE website and GitHub repository, where anyone can check MOLE forums.

8.7 Job Opportunities

Currently, there are no job opportunities. We will use this space to advertise employment and collaboration opportunities in the MOLE OSE.

CHAPTER

NINE

CONTRIBUTING TO MOLE: COMPREHENSIVE GUIDE

Thank you for considering contributing to MOLE (Mimetic Operators Library Enhanced)! This guide provides detailed instructions for contributing to the MOLE project, whether you're adding core functionality, examples, or documentation.

9.1 Table of Contents

9.2 Getting Started

9.2.1 Prerequisites

Before contributing, ensure you have:

- **For MATLAB/Octave**: MATLAB R2016b+ or GNU Octave 4.0+
- **For C++**: CMake 3.10+, OpenBLAS, Eigen3, Armadillo
- **For Documentation**: Python 3.7+, Sphinx, Doxygen

9.2.2 Setting Up Development Environment

1. **Fork the Repository**: Fork the MOLE repository to your GitHub account
2. **Clone Your Fork**:

```
git clone https://github.com/YOUR_USERNAME/mole.git
cd mole
```

3. **Create a Development Branch**:

```
git checkout -b feature/your-feature-name
```

4. **Create an Issue or start from an existing one**:

- Tackle an existing issue. We keep a list of good first issues that are self-contained and suitable for a newcomer to try and work on.
- If there is no existing Issue describing what you want to report/discuss/propose to the community, please create a new Issue
- Once you are ready with your contribution (whether it is for core functionality, examples, or documentation), follow the guidelines below and remember to link your open Pull Request (PR) with the Issue describing what you propose to contribute. It is very important for the community project management to keep organized PR-Issue pairs. If you are not sure how to link a PR with an Issue, please review these Linking a Pull Request to an Issue guidelines from GitHub.

9.3 Contributing to Core Functionality

Core functionality includes mimetic operators, boundary conditions, and utility functions that form the foundation of the MOLE library.

9.3.1 Core API Structure

The MOLE library follows a consistent structure across MATLAB and C++ implementations:

MATLAB/Octave Core Functions

Core functions are located in `src/matlab_octave/` and follow this pattern:

```
function OUTPUT = functionName(k, m, dx, ...)
% BRIEF_DESCRIPTION
%
% Parameters:
%                k : Order of accuracy
%                m : Number of cells (along x-axis for multidimensional)
%               dx : Step size (along x-axis for multidimensional)
%    (additional parameters as needed)
%
% Returns:
%          OUTPUT : Sparse matrix representing the operator
%
% ------------------------------------------------------------------------
% SPDX-License-Identifier: GPL-3.0-or-later
% © 2008-2024 San Diego State University Research Foundation (SDSURF).
% See LICENSE file or https://www.gnu.org/licenses/gpl-3.0.html for details.
% ------------------------------------------------------------------------

    % Input validation
    assert(k >= 2, 'Order of accuracy k must be >= 2');
    assert(mod(k, 2) == 0, 'Order of accuracy k must be even');
    assert(m >= 2*k+1, ['Number of cells m must be >= ' num2str(2*k+1) ' for k = '␣
```

(continues on next page)

(continued from previous page)

```
↪num2str(k)]);

    % Implementation
    % ...

end
```

C++ Core Classes

C++ implementations are in `src/cpp/` and follow this pattern:

```
/*
 * SPDX-License-Identifier: GPL-3.0-or-later
 * © 2008-2024 San Diego State University Research Foundation (SDSURF).
 * See LICENSE file or https://www.gnu.org/licenses/gpl-3.0.html for details.
 */

/**
 * @file classname.h
 * @brief Brief description
 * @date Creation date
 */

#ifndef CLASSNAME_H
#define CLASSNAME_H

#include "required_headers.h"

/**
 * @brief Brief class description
 */
class ClassName : public sp_mat {
public:
    using sp_mat::operator=;

    /**
     * @brief Constructor description
     * @param k Order of accuracy
     * @param m Number of cells
     * @param dx Step size
     */
    ClassName(u16 k, u32 m, Real dx);

    // Additional constructors for 2D, 3D, etc.
};

#endif // CLASSNAME_H
```

9.3.2 Core Function Categories

When contributing core functionality, identify which category your contribution fits:

1. **Differential Operators**: `grad`, `div`, `lap`, `curl`
2. **Interpolation Functions**: `interpol`, `interpol2D`, `interpol3D`
3. **Boundary Conditions**: `robinBC`, `mixedBC`, `addScalarBC`
4. **Utility Functions**: `jacobian2D`, `jacobian3D`, `weights`
5. **Grid Functions**: Curvilinear and non-uniform grid support

9.3.3 Core Function Requirements

1. **Consistent Interface**: Follow the parameter ordering convention: `(k, m, dx, n, dy, o, dz, ...)`
2. **Input Validation**: Validate all input parameters with clear error messages
3. **Sparse Matrix Output**: Return sparse matrices for efficiency
4. **Boundary Condition Support**: Consider how your operator interacts with boundary conditions
5. **Multidimensional Support**: Provide 1D, 2D, and 3D versions where applicable
6. **Documentation**: Include comprehensive function documentation

9.3.4 Example: Adding a New Operator

If adding a new operator, follow this checklist:

1. **MATLAB Implementation** (`src/matlab_octave/newoperator.m`):

```
function OP = newoperator(k, m, dx)
% Returns a new mimetic operator
%
% Parameters:
%                k : Order of accuracy
%                m : Number of cells
%               dx : Step size

    % Validation
    assert(k >= 2 && mod(k, 2) == 0, 'k must be even and >= 2');

    % Implementation using existing operators
    G = grad(k, m, dx);
    D = div(k, m, dx);

    % Combine operators as needed
    OP = someOperation(G, D);
end
```

2. **C++ Implementation** (`src/cpp/newoperator.h` and `src/cpp/newoperator.cpp`)
3. **Add to API Documentation** (`doc/sphinx/source/api/`)
4. **Create Test Examples** (see Examples section)

9.4 Contributing Examples

Examples demonstrate how to use MOLE to solve specific PDEs and are crucial for user education.

9.4.1 Example Structure and Organization

Examples are organized by PDE type in the `examples/` directory:

```
examples/
├── matlab_octave/          # MATLAB/Octave examples
│   ├── elliptic1D.m        # Basic examples
│   ├── parabolic2D.m       # 2D examples
│   └── compact_operators/ # Specialized examples
└── cpp/                    # C++ examples
    ├── elliptic1D.cpp
    └── transport1D.cpp
```

9.4.2 Example Categories

Organize your examples by PDE type:

1. **Elliptic**: Steady-state problems (Poisson, Laplace)
2. **Parabolic**: Time-dependent diffusion (heat equation, reaction-diffusion)
3. **Hyperbolic**: Wave-like phenomena (advection, wave equation)
4. **Mixed**: Problems involving multiple PDE types
5. **Specialized**: Navier-Stokes, Schrödinger, etc.

9.4.3 MATLAB/Octave Example Template

```
% Solves the [EQUATION NAME] with [BOUNDARY CONDITIONS]
% [Brief description of the physics and mathematical formulation]

clc
close all

addpath('../../src/matlab_octave')  % REQUIRED: Add path to MOLE library

%% Problem Parameters
% [Describe each parameter with physical meaning]
k = 2;              % Order of accuracy
m = 50;             % Number of cells
west = 0;           % Domain limits
east = 1;
dx = (east-west)/m; % Grid spacing

%% Physical Parameters
% [Define problem-specific parameters]
alpha = 1;          % Thermal diffusivity (example)
t_final = 1;        % Simulation time

%% Grid Setup
```

(continues on next page)

(continued from previous page)

```
% 1D Staggered grid
xgrid = [west west+dx/2 : dx : east-dx/2 east];

%% Operator Construction
L = lap(k, m, dx);                    % Laplacian operator
L = L + robinBC(k, m, dx, a, b);      % Add boundary conditions

%% Initial and Boundary Conditions
U = initial_condition(xgrid);         % Define initial condition
U(1) = boundary_value_west;           % West boundary
U(end) = boundary_value_east;         % East boundary

%% Time Integration (if applicable)
dt = dx^2/(4*alpha);                  % CFL condition
L_time = alpha*dt*L + speye(size(L)); % Time-stepping operator

for t = 0:dt:t_final
    % Plotting
    plot(xgrid, U, 'LineWidth', 2)
    title(sprintf('Time = %.3f', t))
    xlabel('x')
    ylabel('u(x,t)')
    drawnow

    % Time step
    U = L_time * U;
end

%% Analytical Solution Comparison (if available)
U_analytical = analytical_solution(xgrid, t_final);
plot(xgrid, U, 'o-', xgrid, U_analytical, '--')
legend('Numerical', 'Analytical')
```

9.4.4 C++ Example Template

```
/**
 * @file example_name.cpp
 * @brief Solves the [EQUATION NAME] with [BOUNDARY CONDITIONS]
 *
 * [Detailed description of the physics and mathematical formulation]
 *
 * Equation: [Mathematical equation in LaTeX-style comments]
 * Domain: [Domain description]
 * Boundary Conditions: [BC description]
 */

#include "mole.h"
#include <iostream>
#include <iomanip>

using namespace arma;
```

(continues on next page)

(continued from previous page)

```
int main() {
    // Problem parameters
    constexpr u16 k = 2;        // Order of accuracy
    constexpr u32 m = 50;       // Number of cells
    constexpr Real dx = 1.0/m;  // Grid spacing

    // Physical parameters
    constexpr Real alpha = 1.0; // Thermal diffusivity
    constexpr Real t_final = 1.0;

    // Construct operators
    Laplacian L(k, m, dx);
    RobinBC BC(k, m, dx, a, b);
    L = L + BC;

    // Initial conditions
    vec U = initial_condition();

    // Time integration
    Real dt = dx*dx/(4*alpha);
    sp_mat L_time = alpha*dt*L + speye(size(L));

    for (Real t = 0; t <= t_final; t += dt) {
        // Output current solution
        std::cout << "Time: " << std::fixed << std::setprecision(3) << t << std::endl;

        // Time step
        U = L_time * U;
    }

    // Output final solution
    std::cout << "Final solution:" << std::endl;
    U.print();

    return 0;
}
```

9.4.5 Example Requirements

1. **Self-contained**: Each example should run independently
2. **Well-commented**: Explain the physics, mathematics, and implementation
3. **Parameter Documentation**: Describe all parameters and their physical meaning
4. **Clear Output**: Include appropriate visualization or numerical output
5. **Validation**: Compare with analytical solutions when possible
6. **Consistent Naming**: Use descriptive variable names following MOLE conventions

9.4.6 Mathematical Documentation Requirements

Each example should include:

1. **Mathematical Formulation**: Clear statement of the PDE being solved
2. **Domain Description**: Spatial and temporal domains
3. **Boundary Conditions**: Precise specification of BCs
4. **Initial Conditions**: For time-dependent problems
5. **Analytical Solution**: If available, for validation

9.5 Contributing to Documentation

Documentation contributions help users understand and effectively use MOLE.

9.5.1 Documentation Structure

MOLE uses Sphinx for user documentation and Doxygen for API reference:

```
doc/
├── sphinx/                 # User documentation
│   └── source/
│       ├── examples/       # Example documentation
│       ├── api/           # API references
│       └── math_functions/ # Mathematical background
└── doxygen/               # API documentation
```

9.5.2 Example Documentation Template

Create documentation for examples in `doc/sphinx/source/examples/[Category]/[Dimension]/`:

```
### ExampleName

Brief description of what this example solves.

$$
\text{Mathematical equation in LaTeX}
$$

with domain $x \in [a,b]$ and boundary conditions:

$$
\text{Boundary condition equations}
$$

#### Mathematical Background

[Detailed explanation of the physics and mathematics]

#### Implementation Details
```

(continues on next page)

(continued from previous page)

```
[Key implementation considerations, numerical methods used]

#### Results

[Description of expected results, plots, validation]

---

This example is implemented in:
- [MATLAB/Octave](https://github.com/csrc-sdsu/mole/blob/main/examples/matlab_octave/
↪example_name.m)
- [C++](https://github.com/csrc-sdsu/mole/blob/main/examples/cpp/example_name.cpp) *(if␣
↪available)*

#### Variants

Additional variants with different boundary conditions or parameters:
- [Variant 1](link_to_variant)
- [Variant 2](link_to_variant)
```

9.5.3 Documentation Guidelines

1. **Mathematical Notation**: Use LaTeX for equations
2. **Code References**: Link to actual implementation files
3. **Cross-references**: Link related examples and API functions
4. **Images**: Include plots and diagrams where helpful
5. **Consistent Structure**: Follow the established template

9.6 Code Standards and Guidelines

9.6.1 MATLAB/Octave Standards

1. **Function Names**: Use descriptive names following the existing convention
2. **Variable Names**: Use clear, descriptive variable names
3. **Comments**: Document complex algorithms and physics
4. **Error Handling**: Use `assert` for input validation
5. **Performance**: Use sparse matrices, avoid loops when possible

9.6.2 C++ Standards

1. **Naming Convention**:
   - Classes: `PascalCase`
   - Functions: `camelCase`
   - Variables: `snake_case` for local, `camelCase` for members

2. **Headers**: Include proper copyright and license headers
3. **Documentation**: Use Doxygen-style comments
4. **Memory Management**: Use smart pointers when appropriate
5. **Performance**: Leverage Armadillo's optimizations

9.6.3 General Guidelines

1. **Consistency**: Follow existing code patterns
2. **Testing**: Ensure your contributions work with provided examples
3. **Documentation**: Document all public interfaces
4. **License**: Include appropriate license headers
5. **Dependencies**: Minimize external dependencies

9.7 Testing and Validation

9.7.1 Testing Your Contributions

1. **Unit Testing**: Test individual functions with known inputs/outputs
2. **Integration Testing**: Test how your contribution works with existing code
3. **Convergence Testing**: Verify order of accuracy for new operators
4. **Example Testing**: Ensure examples run and produce expected results

9.7.2 Validation Methods

1. **Analytical Solutions**: Compare with known exact solutions
2. **Convergence Studies**: Verify theoretical order of accuracy
3. **Conservation Laws**: Check that operators preserve conservation properties
4. **Cross-platform Testing**: Test on both MATLAB and Octave (for MATLAB code)

9.7.3 Performance Considerations

1. **Sparse Matrix Efficiency**: Ensure operators are sparse
2. **Memory Usage**: Monitor memory consumption for large problems
3. **Computational Complexity**: Document algorithmic complexity
4. **Scalability**: Test with various problem sizes

9.8 Submission Process

9.8.1 Before Submitting

1. **Code Review**: Review your code for style and functionality
2. **Testing**: Run all relevant tests and examples
3. **Documentation**: Ensure documentation is complete and accurate
4. **Commit Messages**: Write clear, descriptive commit messages
5. **Atomic commits**: Please make your commits well-organized and atomic, using `git rebase --interactive` as needed

9.8.2 Pull Request Guidelines

1. **Title**: Use descriptive title indicating what was added/changed
2. **Description**: Provide detailed description of changes
3. **Testing**: Describe testing performed
4. **Examples**: Include or reference relevant examples
5. **Documentation**: Link to any new documentation
6. **PR/Issue pair**: Link your PR to an existing Issue that describes your intended contribution

9.8.3 Pull Request Template

```
## Description
Brief description of changes made.

## Type of Change
- [ ] New core functionality
- [ ] New example
- [ ] Documentation update
- [ ] Bug fix
- [ ] Performance improvement

## Mathematical Details
[For new operators] Mathematical formulation and properties.

## Testing
- [ ] Unit tests pass
- [ ] Examples run successfully
- [ ] Convergence studies completed (if applicable)
- [ ] Cross-platform compatibility verified

## Documentation
- [ ] Code is well-commented
- [ ] API documentation updated
- [ ] Example documentation added (if applicable)
- [ ] Mathematical background provided

## Related Issues
Fixes #(issue number)
```

(continues on next page)

(continued from previous page)

```
## Additional Notes
Any additional information or context.
```

9.8.4 Review Process

1. **Automated Checks**: CI/CD will run automated tests
2. **Code Review**: Maintainers will review code quality and consistency
3. **Mathematical Review**: Mathematical correctness will be verified
4. **Documentation Review**: Documentation completeness will be checked
5. **Performance Review**: Performance impact will be assessed

---

9.9 Getting Help

9.9.1 Resources

1. **Documentation**: Read the online documentation
2. **Examples**: Study existing examples for patterns and best practices
3. **Issues**: Check GitHub issues for known problems
4. **Discussions**: Use GitHub Discussions for questions and ideas

9.9.2 Contact

For questions, support, or contributions, contact the MOLE maintainers at:

- jcastillo@sdsu.edu
- mdumett@sdsu.edu
- paolini@engineering.sdsu.edu
- jjbrzenski@sdsu.edu
- vbarra@sdsu.edu

For specific types of support:

- **GitHub Issues**: For bug reports and feature requests
- **GitHub Discussions**: For general questions and discussions

9.9.3 Contributing License Agreement

By contributing to MOLE, you agree that your contributions will be licensed under the GNU General Public License v3.0 or later. Ensure you have the right to license your contributions under this license.

9.10 Authorship and Recognition

MOLE contains components authored by many individuals from the computational science community. We believe it is essential that contributors receive appropriate recognition through both informal acknowledgment and academically-recognized credit systems such as publications and citations.

9.10.1 Authorship Criteria

Status as a named author in MOLE publications, the user manual, and DOI-bearing archives will be granted to those who:

1. **Make significant contributions to MOLE** in any of the following areas:
   - Implementation of core functionality (operators, boundary conditions, utilities)
   - Creation of comprehensive examples and tutorials
   - Documentation and mathematical framework development
   - Conceptualization of new features or mathematical approaches
   - Code review, testing, and validation
   - Community building and user support
2. **Maintain and support their contributions** over time, including:
   - Responding to issues related to their contributions
   - Updating code to maintain compatibility
   - Providing ongoing documentation and support

9.10.2 Recognition Process

- **Automatic Recognition**: Maintainers will monitor contributions and add qualifying contributors to the `AUTHORS` file
- **Self-Nomination**: If you believe your contributions meet the authorship criteria but haven't been acknowledged, please:
  - Email the maintainers (see *Contact* section)
  - Create a GitHub issue describing your contributions
- **Periodic Review**: The maintainer team regularly reviews the contributor list to ensure proper recognition

9.10.3 Publication Guidelines

MOLE Software Publications

Authors of publications about MOLE as a whole, including:

- Software papers and technical reports
- DOI-bearing software archives
- Major release announcements
- Comprehensive method descriptions

shall offer co-authorship to all individuals listed in the `AUTHORS` file at the time of submission.

Publications Using MOLE Features

Authors of publications that describe **specific MOLE contributions or new features** shall:

1. Review the `AUTHORS` file to identify relevant contributors
2. Evaluate the intellectual contributions of listed authors to the specific work
3. Offer co-authorship to those who made significant intellectual contributions to the featured work
4. At minimum, acknowledge MOLE and cite appropriate references

Publications Using MOLE for Research

No co-authorship expectation exists for those publishing research that uses MOLE as a computational tool (versus creating new features in MOLE). However:

- **Citation**: Please cite MOLE appropriately (see CITATION.cff)
- **Acknowledgment**: Consider acknowledging significant support or advice received from MOLE developers
- **Judgment**: Use your best judgment regarding the significance of support received in developing your use case and interpreting results

9.10.4 Citing MOLE

When using MOLE in your research, please cite:

```
@misc{mole,
  title = {{MOLE}: Mimetic Operators Library Enhanced},
  author = {[See AUTHORS file for complete list]},
  url = {https://github.com/csrc-sdsu/mole},
  note = {Version X.X.X},
  year = {2024}
}
```

For specific mathematical methods or operators, also cite the relevant publications listed in our documentation.

9.10.5 Examples of Significant Contributions

To clarify what constitutes "significant contributions," here are examples:

Core Development:

- Implementing new differential operators or boundary conditions
- Adding support for new grid types or geometries
- Developing new mathematical formulations
- Creating fundamental algorithmic improvements

Documentation and Education:

- Writing comprehensive tutorials or guides
- Creating educational examples with mathematical background
- Developing API documentation for major components
- Contributing to the mathematical framework documentation

Testing and Validation:

- Developing comprehensive test suites

- Conducting convergence studies and validation
- Cross-platform compatibility work
- Performance optimization and benchmarking

Community Building:

- Mentoring new contributors
- Organizing workshops or educational events
- Significant bug reporting and issue management
- Long-term maintenance and support

Thank you for contributing to MOLE! Your contributions help advance computational science and benefit the entire research community.

Thank you for considering contributing to MOLE!

9.11 Our Contributors

We're grateful to all our contributors who have helped make MOLE better:

CHAPTER

TEN

MOLE OPEN-SOURCE ECOSYSTEM ORGANIZATION

10.1 Table of Contents

10.2 Statement of Purpose

First, it is important to state the purpose and scope of the MOLE OSE organization to provide context for the MOLE OSE structural organization depicted in Figure 1. The MOLE Open-Source Ecosystem organization was established to support the MOLE library's ecosystem as a sustainable, robust, secure, and more readily available library of high-order mimetic differential operators. The MOLE OSE serves academic and industrial communities that require high-performance solvers for PDEs.

10.3 MOLE OSE Organization Pillars

The MOLE organization is strategically supported by four working pillars;

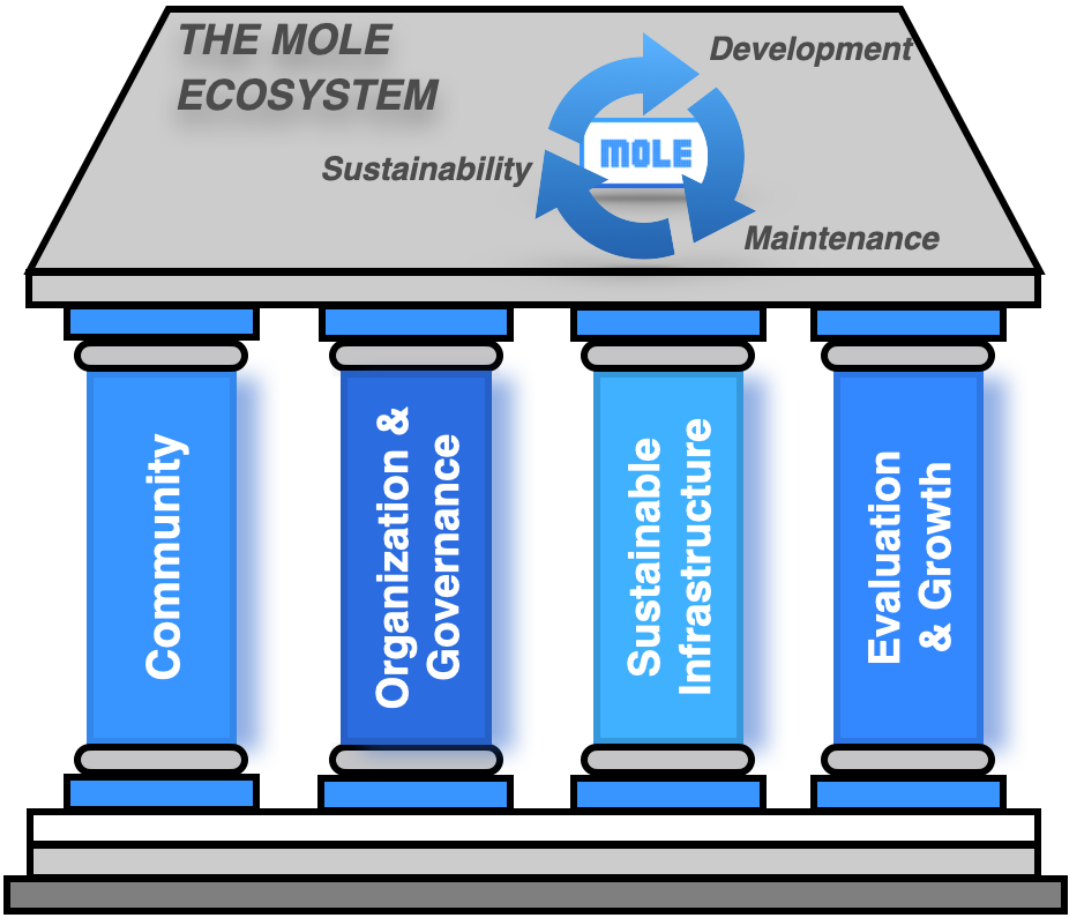


Figure 1. The four pillars of the MOLE Ecosystem Organization.

10.4 Community Engagement Pillar

This pillar hosts all activities and actions related to the management and support of a leadership team of experts in the fields of computational sciences, software engineering, and high-performance computing working cohesively with external collaborators, contributors and users of the MOLE library. Further, the leadership team, collaborators, contributors and users form ***the MOLE community***.

In order to engage and recruit new members, this pillar fosters the organization of the Annual MOLE Users meeting and in person community engagements like national and international workshops, active participation in conferences, and integration of MOLE into courses taught at SDSU. The production of the quarterly newsletters and targetted communications are managed also under this pillar.

10.5 Organization and Governance Pillar

This pillar results from a well-defined governance model, the pillar fosters and promotes work interactions between the leadership team and all the MOLE OSE stakeholders. For instance, the creation and update the Governance and

Code of Conduct documents, the prompt resolution of conflicts, the design and implementation of the MOLE OSE strategic plan. Under this pillar, the leadership team ensures that all the pillars and organizational circles are relevant and effective to the MOLE OSE operations.

10.6 Sustainable Infrastructure Pillar

This pillar supports the work of an interdisciplinary, multilocation, and cohesive MOLE community, including hands-on tutorials, documentation, and online tools to promote active and timely communications between members of the MOLE Community.

It adopts agile and distributed open-source development practices, ensuring regular releases, responsive maintenance, and integration of community contributions. The OSE sustainable infrastructure is built on a well-maintained GitHub repository to support active collaborations between contributors and other sustainable OSEs.

10.7 Evaluation and Growth Pillar

This pillar hosts the implementation and management of a set of well-defined metrics and performance indicators that guarantee the sustainability and relevancy of the MOLE library and the OSE organization, their growth, their evolution and the library's adoption within third-party academic and commercial software projects.

We have established proactive policies for software quality assurance, security audits, and mitigation of technical and governance risks within the MOLE OSE Evaluation and Growth pillar, where we have feedback loops to produce measurable performance indicators of the effectiveness of our 4-pillars OSE.

10.8 MOLE OSE Governance Model

The four MOLE OSE pillars have interconnected and synergetic activities that sustain the MOLE OSE Organization which actively relies on input from the MOLE community. Our governance model has well-defined processes for contributions, library operations and management. Thus, the management of the MOLE organization is led by a steering council, and four governing circles.

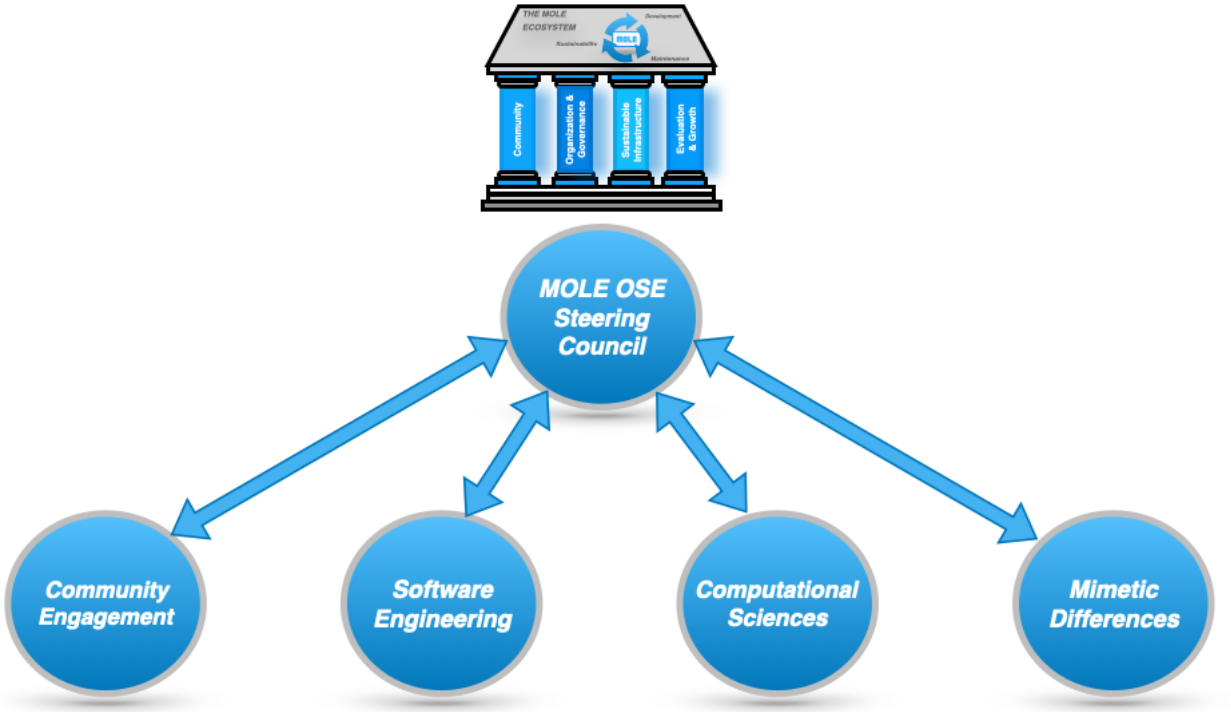


Figure 2. The MOLE OSE governance and leadership is organized in four strategic areas reflecting the domains of expertise and competence of the members in the Leadership team and founders of MOLE library. These domains are; software engineering, computational sciences, mimetic differences, and community engagement.

See the MOLE OSE Governance document for more information on how the steering council or any of the governance circles onboard or offboard their members.

10.8.1 MOLE Steering Council

The members of council are responsible for facilitating the establishment of community consensus, for stewarding project resources, and in some situations they may need to make project decisions whenever the normal decision-making process fails at reaching consensus by the community. The council members are formed by the leadership team members, chairs from the MOLE governing circles, MOLE founders, and strategic partners who financially support the MOLE OSE.

All council members possess the required expertise in the software engineering and numerical areas relevant to the purpose and scope of the MOLE organization.

10.8.2 MOLE Community Engagement Circle

The members of this circle oversee some of the activities in the *MOLE Community Engagement* and *MOLE Organization and Governance* pillars pertaining to outreach and education events, promotion of the MOLE library functionality, and onboarding of new members to the MOLE community.

The circle is chaired by members of the MOLE Leadership team, and it is open to any member of the MOLE community, as long as they agree to comply with the MOLE OSE Code of Conduct, and there are no objections.

10.8.3 MOLE Software Engineering Circle

The members of this circle oversee most of the activities in the *MOLE OSE Sustainable Infrastructure* pillar and some in the *MOLE OSE Organization and Governance* pillar. This includes: security of computer programs, library design and implementations, MOLE software ecosystem, and ecosystems of MOLE-required software, build systems, documentation systems, test harness, design and implementation of the MOLE APIs in the targeted languages, and compliance up-to-date software engineering standards, MOLE library licensing and periodic MOLE library releases.

Membership of this circle will require demonstrated software engineering competency, in particular within the scope of the MOLE Ecosystem.

10.8.4 MOLE Computational Sciences Circle

The members of this circle safeguard the relevancy of the MOLE library by fostering strategic collaborations to solve PDE's using mimetic differences in emerging computational applications and challenges in sciences and engineering. Members of this circle support activities in the *MOLE Community Engagement* and *MOLE OSE Evaluation and Growth* pillars. In particular, they produce MiniApps to showcase and demonstrate the applicability of the MOLE library, and also make decisions on library extensions to support needs from the community.

Membership to this circle is extended to anyone who possesses expertise in computational areas where the MOLE library can be used.

10.8.5 MOLE Mimetic Differences Circle

The members of this circle are responsible for maintaining and advancing the implementation of Mimetic Differences methods (MD) and for promoting and recruiting applications that can benefit from the use of MD as new users of MD, or members from industry and academia who already use MD but are in need of robust, reliable and scalable MD implementations. This circle supports the sustainability of key algorithmic and MOLE library functionalities through activities under the *OSE Evaluation and Growth* pillar.

Membership to this circle is extended to anyone who possesses expertise in mathematics and scientific computing, in particular in numerical solutions to partial differential equations.

Participants can be removed from any of the governance circles whenever a proposal to do so passes the consensus process as described in the MOLE OSE Governance document.

CHAPTER

ELEVEN

MOLE OPEN-SOURCE ECOSYSTEM GOVERNANCE

11.1 Table of Contents

11.2 Statement of Purpose

This document formalizes the description of the governing processes used by the MOLE Open-source Software Ecosystem (OSE) organization in the decision-making process involving ordinary and extraordinary operations in the sustainable development and maintenance of the MOLE Library, its software infrastructure and MOLE OSE organization. These processes and provisions also apply to all work with external collaborators from academia and industry. It also describes the procedures the MOLE OSE steering council uses for conflict resolution.

The ultimate goal of the governance definitions and processes is to promote a work environment that is collaborative, promotes openness and transparency, encourages software contributions to the MOLE library, and supports the advancement of multiple fields in computational sciences. This document is intended to be used in conjunction with the MOLE OSE Code of Conduct document, which outlines the rules of engagement between a multidisciplinary and diverse group of contributors and members of the MOLE Community.

11.3 MOLE OSE Governance

First, we are obliged to state the scope of the software ecosystem for the MOLE Library to provide the context for all the kinds of operations, activities and interactions expected within and around the MOLE OSE organization, and this well-defined purpose guides MOLE's rules of collaborative engagement (see the MOLE OSE Code of Conduct) and the decision-making governing processes.

MOLE is a high-order mimetic differential operators library for solving PDEs. The library is developed by a distributed group of contributors led by the MOLE leadership team. The MOLE leadership team is responsible for maintaining the cohesiveness, robustness, security, reliability, succession plans, integrity, and quality of the software library, and

contributors are individuals who contribute algorithms, code design, code implementations, documentation, or user support to the MOLE library or the MOLE OSE organization. The role definitions for all the stakeholders in this organization are defined in the Community Roles page.

Anyone with an interest in mimetic methods and their use in the formulation of numerical solutions to computational problems can become a member of the MOLE community. Additionally, anyone with an interest in learning more about the MOLE library functionalities or mimetic discretizations and their wide-range of applicability is invited to join the MOLE community. Lastly, as an open-source software library, the MOLE community also welcomes participants interested in its financial sustainability.

11.4 Consensus-Based Decision-Making

A thriving community is essential to the success and sustainability of the MOLE OSE organization. In fact, MOLE users, contributors and collaborators help make the library more high-quality, robust, relevant, and ready to tackle more problems in computational sciences. In turn, the MOLE OSE organization prioritizes the establishment of trust in the longevity and maintenance of the library. Users need assurance that the software ecosystem will not become obsolete or unsupported. Therefore, most decisions in the organization are made by consensus of the MOLE Community, allowing the MOLE OSE to leverage on the multidisciplinary expertise in the community.

Anyone in the MOLE community can provide feedback, participate in forums and software reviews, report issues, make suggestions and voice opinions on decisions.

The MOLE governance is organized in four functional circles and a steering council, as described in more detail in the MOLE OSE Organization page. The four governance circles are: *community engagement, software engineering, computational sciences* and *mimetic methods*. The ***steering council*** is responsible for stewarding project resources, and in some situations they may need to make project decisions whenever the normal decision-making process fails at reaching consensus by the community.

Most MOLE OSE organization decisions are made on consensus, and whenever a decision cannot be reached by consensus, a formal vote is called by the chairs of the corresponding governance circle. In the context of open-source software development, we use the definition of consensus from Chapter 4: Social and Political Infrastructure by Karl Fogel, and extracted from the book Producing Open Source Software, and we quote:

> *Consensus simply means an agreement that everyone is willing to live with. It is not an ambiguous state: a group has reached consensus on a given question when someone proposes that consensus has been reached and no one contradicts the assertion.*

11.5 The Voting System

Whenever a vote is called to resolve a conflict, there will be a deadline for responding to the vote. Note that not all members of the community are expected to vote on all decisions, and the voting time will not be delayed for anyone. In the best interest of timely and effectively resolving a conflict, the governance circle may choose to escalate a conflict to the MOLE Steering Council.

For full disclosure, the MOLE OSE organization also has these decisions that are not open to a vote by the MOLE community. These decisions are handled within a governance circle or the MOLE steering council:

- Decisions that cannot be made by consensus will resolve with a binding decision through a simple majority vote of the active steering council members,
- The onboarding and offboarding of members of the steering council will require a two-thirds majority vote of the active steering council members,

- The onboarding and offboarding of members in a governance circle will require a simple majority vote of the active members in the circle,
- The election of chairs and co-chairs of a governance circle are elected by a majority vote of active members of the steering council. All governance circle chairs and co-chairs are members of the steering council,
- The removal of MOLE community members will require a two-thirds majority vote of the active steering council members,
- Any changes to the MOLE OSE organization, its structure, governance composition will require a two-third majority of the active steering council members,

The outcome of any decision will be announced to the MOLE community. The MOLE community is encouraged to provide feedback and opinions on decisions made by any of the above voting scenarios.

11.6 Removal of Members from the Community

In order to promote a welcoming, intellectually stimulating and productive work environment, the MOLE OSE organization has a set of guiding rules of engagement listed in the MOLE Code of Conduct document. If a member is in direct violation of any of these rules, a formal request for their removal from the MOLE community will be made to the steering committee. This also applies to any community members that enters in conflict of interest with the MOLE OSE organization.

11.7 Removal of Members from the Steering Council or A Governance Circle

When members of the steering council or a governance circle become inactive, or are unable to participate in discussions and meetings, they will be removed from the steering council or a governance circle, requiring a two-thirds majority vote of the active steering council members or active members in a governance circle. Members can also be removed if there are any conflicts of interests or violations of the MOLE OSE Code of Conduct.

11.8 Conflicts of Interest

Members of the steering council and governing circles will enter in conflict of interests with the MOLE OSE organization in one of the following scenarios;

- Personal financial interests not directly connected with the MOLE OSE financial sustainability, that may influence their work on MOLE OSE organization.
- Sell, partially or in full, any of the products in the MOLE OSE organization for personal gain.

All members of the steering council and governance circles must disclose to the rest of the steering council any conflict of interest they may have. Members with a conflict of interest in a particular issue may participate in council or governance circle discussions on that issue, but must recuse themselves from voting on the issue.

Conflict of interest forms will be signed on the yearly bases by all members of the steering council and governance circles.

11.9 Acknowledgement

The rich community of Open Source Software has provided the knowledge and materials that we used while researching creating the MOLE OSE Governance model and its governing processes. Besides, Karl Fogel's book, we would like to acknowledge the following OSEs:

- PETSc,
- NumPy,
- TensorFlow,
- and other OSE's in the high performance E4S community.

CHAPTER

TWELVE

CONTRIBUTOR COVENANT CODE OF CONDUCT

12.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

12.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

12.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

12.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

12.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

12.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

CHAPTER

THIRTEEN

MOLE LIBRARY PUBLICATIONS

The documents and publications in this page highlight some of the early successes of the MOLE Library, as well as accomplishments from using mimetic differences in the solution of Partial Differential Equations.

13.1 Introduction to Mimetic Methods and MOLE

- J.E. Castillo and R.D. Grone, A matrix analysis approach to higher-order approximations for divergence and gradients satisfying a global conservation law, SIAM J. Matrix Anal. Appl., Vol. 25, No. 1, pp. 128-142, 2003. doi.org/10.1137/S089547980139802.
- M. Dumett and J.E. Castillo, General Framework for Mimetic Differences Publication Number: *CSRCR2024-07*, June 2024.
- J. Corbino, M. Dumett and J.E. Castillo, MOLE: Mimetic Operators Library Enhanced, The Journal of Open Source Software, 9(99), 6288 (2024), https://doi.org/10.21105/joss.06288.
- J. Corbino, and J.E. Castillo, High-order mimetic finite-difference operators satisfying the extended Gauss divergence theorem, J. Comput. Appl. Math., v. 364, 2020, 112326.https://doi.org/10.1016/j.cam.2019.06.042.
- J. E. Castillo and M. Yasuda, Linear Systems Arising for Second-Order Mimetic Divergence and Gradient Discretizations, Journal of Mathematical Modelling and Algorithms, 4(1), 67–82. https://doi.org/10.1007/s10852-004-3523-1.

13.2 Advantages of Mimetic Methods Over Other Methods

- M.A. Dumett, An Initial Comparison Between Mimetic Differences and Other Discretization Methods,DOI:10.13140/RG.2.2.14762.89289
- J. E. Castillo and M. Yasuda, A Comparison of Two Matrix Operator Formulations for Mimetic Divergence and Gradient Discretizations, International Conference on Parallel and Distributed Processing Techniques and Applications, Volume: III, (2003).

13.3 Scientific Applications Using MOLE

- M. Ferrer, J. De La Puente, A. Farrés, and J. E. Castillo, 3D Viscoelastic Anisotropic Seismic Modeling with High-Order Mimetic Finite Differences, ICOSAHOM 2014, Lecture Notes in Computational Science and Engineering 106, DOI 10.1007/978-3-319-19800-2_18.