Exasim

Generating Discontinuous Galerkin Codes For Extreme Scalable Simulations

The documentation for Exasim 0.1

Developed and written by N. C. Nguyen

Designed by N. C. Nguyen and J. Peraire

February 3, 2021

1 Overview

Exasim is an open-source software for generating discontinuous Galerkin codes to numerically solve *parametrized* partial differential equations (PDEs) on different computing platforms with distributed memory. It combines high-level languages and low-level languages to easily construct *parametrized* PDE models and automatically produce high-performance C++ codes. The construction of *parametrized* PDE models and the generation of the stand-alone C++ production code are handled by high-level languages, while the production code itself can run on various machines, from laptops to the largest supercomputers, with both CPU and GPU processors. Figure 1 illustrates the intended use of Exasim for solving PDEs.

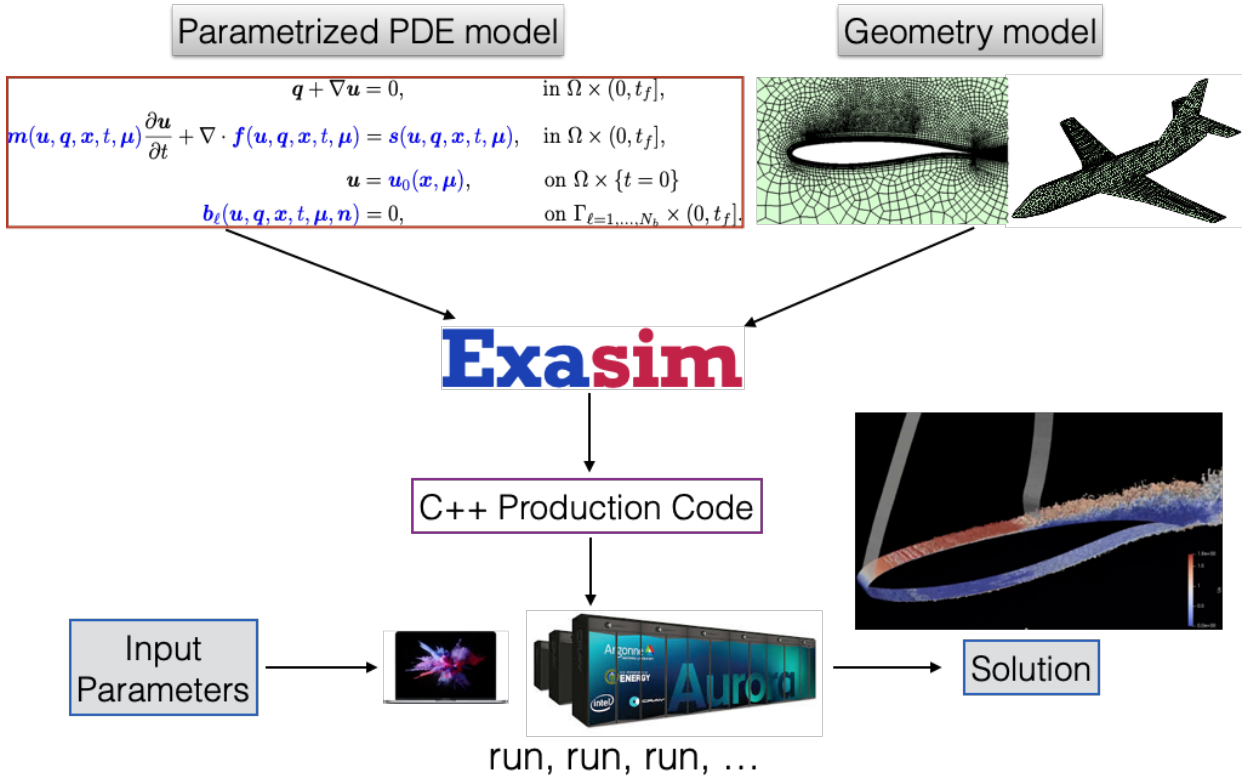

Figure 1: A parametrized PDE model is input into Exasim as functions (written in blue colors) of the field variables, spatial coordinates, time, and physical parameters. A geometry model or a finite element mesh is needed to describe the physical domain. Exasim generates C++ code to solve the parametrized PDE model for many input parameters across different computing platforms.

What make Exasim unique are the following distinctive features:

- Exasim intuitively simplifies modeling and simulation
- generates stand-alone C++ production code and gives practitioners freedom to modify the code and execute it as desired
- provides implicit high-order DG solution of parametrized PDE models
- provides full GPU functionality, meaning that all code components from discretization schemes to iterative solvers are deployed fully on GPUs
- and is available in Julia, Python, and Matlab.

1.1 Obtaining Exasim

Click here to download Exasim's source code. Alternatively, you can clone the repository directly on the command line via

```
git clone https://github.com/exapde/Exasim.git
```

Exasim is freely available under the MIT License. Please see the license details for terms and conditions.

After downloading the source code, please make sure that the name of the folder is Exasim. If it has a different name, please rename it to Exasim.

1.2 Installing External Packages

Exasim automatically generates and compiles stand-alone C++ code on the fly. To do that, Exasim requires a C++ compiler and Blas/Lapack libraries to generate serial codes. An MPI library such as Open-MPI is required to generate parallel codes. CUDA Tookit is required to generate CUDA codes on Nvidia GPUs. Gmsh is used for mesh generation. METIS is needed for mesh partition. And Paraview is needed for visualization.

To install the required packages, please open Julia, Python, or Matlab and go to the folder Exasim/Installation and run the install script.

If you use Exasim with Julia, type the following line and hit return

```
julia> include("install.jl")
```

If you use Exasim with Python, type the following line and hit return

```
>>> exec(open("install.py").read())
```

If you use Exasim with Matlab, type the following line and hit return

```
>> install
```

After installation, the executable files (or their symbolic links) of these packages are usually found in the directory /usr/local/bin or /usr/bin. To know the directory of an executable file, open the terminal and type "which executable". For example, if you type "which gmsh" and see /usr/local/bin/gmsh or /usr/bin/gmsh on the terminal screen, it means that Gmsh was installed in the directory /usr/local/bin or /usr/bin, respectively. On the other hand, if you see "gmsh not found" on the terminal screen, then either Gmsh is not installed or it is installed in a directory which is not included in the PATH environment variable. If Gmsh is not installed, you should install it. If Gmsh was already installed and you know the directory containing Gmsh's executable file, you must add that directory to the PATH environment variable. Please see the setpath script in the the folder Exasim/Installation for adding directories to the PATH environment variable. Doing so will enable Exasim to find the external softwares and run them. For example, when Paraview is installed on MacOS systems, it is usually installed in the directory /Applications. So, when you type "which paraview" in the terminal, you may see "paraview not found" even though it is already installed. This is because Paraview's executable file is located at the directory /Applications/ParaView-5.8.1.app/Contents/MacOS which is not included on the PATH environment variable. In order for Exasim to find paraview, you must include /Applications/ParaView-5.8.1.app/Contents/MacOS in the PATH environment variable. To do that, please modify the setpath script in the the folder Exasim/Installation.

You can try `Exasim` without installing the required packages since `Exasim` automatically searches the required packages on your computer system. If `Exasim` could not find any required package, then follow the below steps to install that package.

MacOS systems: Most required external packages can be conveniently installed by using the package manager Homebrew. After installing Homebrew, open the terminal and run the following commands:

```
$ brew install gcc
$ brew install openblas
$ brew install lapack
$ brew install openmpi
$ brew install metis
$ brew install gmsh
$ brew cask install paraview
$ brew cask install julia
$ brew install python
```

Linux systems: Open the terminal and run the following commands:

```
$ sudo apt install gcc
$ sudo apt install libblas-dev liblapack-dev
$ sudo apt install openmpi
$ sudo apt install metis
$ sudo apt install gmsh
$ sudo apt install paraview
$ sudo apt install julia
$ sudo apt install python
```

Windows systems: It is highly recommended to install the above packages via Windows Subsystem for Linux and Ubuntu. The installation on Windows Subsystem for Linux is the same as on Linux system.

As `Exasim` uses high-level languages to generate C++ code, you need one of the three languages Julia, Python, or Matlab to run `Exasim`. You can install Julia and Python as described above. Alternatively, wou can download and install Julia at the website https://julialang.org/downloads/. Optionally, we recommend you to install Atom editor. Atom allows you to write C, C++, Julia, Python, Matlab codes and run your codes interactively.

Depending on which language you use to run `Exasim`, you need to install a few more packages.

Julia: `Exasim` requires SymPy and Revise, which can be obtained by using the following commands in Julia's REPL session

```
julia> import Pkg; Pkg.add("SymPy"); Pkg.add("Revise");
```

It is important to note that Julia's SymPy calls Python's SymPy from Julia. In order for Julia's SymPy to work., you need to install both Python and Python's SymPy.

Python: `Exasim` requires numpy, scipy, and sympy, which can be obtained by using the following commands in the terminal

```
$ sudo pip3 install numpy
$ sudo pip3 install scipy
$ sudo pip3 install sympy
```

for all operating systems.

Matlab: `Exasim` requires Symbolic Math Toolbox.

1.3 Examples

Many examples are provided to illustrate how to generate DG codes for solving a wide variety of PDEs including Poisson equation, wave equation, heat equation, advection, convection-diffusion, elasticity, Euler equations, Navier-Stokes equations, and MHD equations. To try out any of the provided examples, practitioners go to a folder under Exasim/Applications and run pdeapp.jl in Julia REPL session, pdeapp.py in Python REPL session, or pdeapp.m in Matlab Command Window.

For Julia, go to any folder under Exasim/Applications, type the following line and hit return

```
julia> include("pdeapp.jl")
```

For Python, go to any folder under Exasim/Applications, type the following line and hit return

```
>>> exec(open("pdeapp.py").read())
```

For Matlab, go to any folder under Exasim/Applications, type the following line and hit return

```
>> pdeapp
```

If successful, `Exasim` produces three new folders. The app folder contains the source code and executable application, the datain folder contains input files for the executable application, and the dataout folder contains the output files produced by running the executable application, which stores the numerical solution of the PDE model defined in the pdeapp script. `Exasim` also opens Paraview to visualize the numerical solution. Because `Exasim` runs the executable application in the REPL session, the executable prints out the simulation progress in the REPL session window. Alternatively, you can run the executable from the terminal. The generated source code can also be transferred to another computer and compiled to run on that computer.

New PDE models/DG solvers can be generated by making use of the provided examples. The process of generating DG code for a particular PDE model is described Section 4.

1.4 PDE Models

`Exasim` supports a wide range of PDE models described in Section 2. Physical problems governed by these PDE models can be found in fluid mechanics, solid mechanics, and electromagnetism.

1.5 Finite Element Mesh

Exasim provides a Mesh module to generate meshes for simple geometries. Any open-source mesh generators such as CUBIT, CGAL, DistMesh, TetGen, Mmg, Gmsh, MeshLab, SALOME can be used for complex geometries. Exasim uses Gmsh to generate meshes from geometry model files. Because a high-order mesh is needed for the DG discretization of a PDE model, Exasim produces the high-order mesh from a standard finite element mesh.

1.6 Discretization Methods

Discretization methods refer to numerical methods used to discretize spatial derivatives and time derivatives of a PDE model. Discontinuous Galerkin (DG) methods are used for spatial discretization, while diagonally implicit Runge-kutta (DIRK) schemes are used for temporal discretization. These methods are implemented in C++ and the source codes can be found in the folder Exasim/Version0.1/Kernel. Exasim allows practitioners to implement a wide variety of DG methods (see Section 6 for details).

1.7 Matrix-Free Iterative Solvers

Solvers refer to solution methods used to solve nonlinear and linear systems arising from the discretization of a PDE model. Exasim implements a matrix-free Newton-GMRES solver. Newton method is used to solve nonlinear systems of equations arising from the DG/DIRK discretization of PDE models. For each Newton iteration, GMRES is used to solve the resulting linear systems of equations. The matrix-vector multiplications in GMRES are computed in matrix-free fashion using Taylor's series expansion of the residual vector to the first order or the second order. Reduced basis method is used to construct an approximation to the Jacobian matrix for preconditioning the linear systems. The C++ implementation of these methods can be found in the folder Exasim/Version0.1/Kernel.

1.8 Visualization

Exasim uses Paraview to visualize and analyze simulation results obtained by running the C++ production code. To do that, Exasim generates VTU files and opens Paraview to visualize the computed solution.

1.9 Reporting Issues and Suggesting Improvements

Please click here to report any issues you encounter using Exasim and provide a detailed description of the issue as you can. If you have ideas for improvement, we would love to hear them by emailing us at exapde@gmail.com.

2 Parametrized PDE Models

Exasim produces executable applications to solve a wide variety of PDE models. The underlying PDE system must be written as a set of first-order PDEs. In this section, we describe how to input a PDE model into Exasim.

2.1 Model C: Convection Model

The Model C consists of any set of PDEs that can be written in the following form:

$$\boldsymbol{m}(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu})\frac{\partial \boldsymbol{u}}{\partial t} + \nabla\cdot\boldsymbol{f}(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}) = \boldsymbol{s}(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}), \quad \text{in } \Omega\times(0,t_f], \tag{1}$$

with appropriate initial and boundary conditions. Here $\boldsymbol{u} = [u_1, u_2, \ldots, u_{n_{cu}}]$ is the vector of n_{cu} state variables, $\boldsymbol{x} = [x_1, \ldots, x_{n_d}]$ is the vector of coordinate variables in Ω, t represents time variable in $(0, t_f]$, and $\boldsymbol{\mu} = [\mu_1, \ldots, \mu_{n_{param}}]$ is a vector of n_{param} physical parameters. The state vector $\boldsymbol{u}$ is the exact solution of the Model C (1). `Exasim` produces codes to compute the approximate solution $\boldsymbol{u}_h$.

The vector-valued function $\boldsymbol{m} = [m_i(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}), 1 \le i \le n_{cu}]$ is called *mass* function, the matrix-valued function $\boldsymbol{f} = [f_{ij}(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}), 1 \le i \le n_{cu}, 1 \le j \le n_d]$ is called *flux* function, The vector-valued function $\boldsymbol{s} = [s_i(\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}), 1 \le i \le n_{cu}]$ is called *source* function. These functions are specified by writing functions in high-level languages (Julia, Python, or Matlab).

Examples of the Model C include linear convection equation, the Burgers equation, the Euler equations, and the shallow water equations. `Exasim` can solve both steady-state and unsteady problems. The steady-state version of the Model C can be obtained by setting $\boldsymbol{m}$ to zeros.

2.2 Model D: Convection-Diffusion Model

The Model D consists of any set of PDEs that can be written in the following form:

$$\boldsymbol{m}(\boldsymbol{u},-\nabla\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu})\frac{\partial \boldsymbol{u}}{\partial t} + \nabla\cdot\boldsymbol{f}(\boldsymbol{u},-\nabla\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}) = \boldsymbol{s}(\boldsymbol{u},-\nabla\boldsymbol{u},\boldsymbol{x},t,\boldsymbol{\mu}), \quad \text{in } \Omega\times(0,t_f], \tag{2}$$

with appropriate initial and boundary conditions. The Model D is a generalization of the Model C by including the negative gradient of the state vector (i.e. $-\nabla\boldsymbol{u}$) in the *mass*, *flux*, and *source* functions.

It is convenient to introduce additional state variables $\boldsymbol{q} = -\nabla\boldsymbol{u}$ and rewrite the Model D as follows

$$\boldsymbol{q} + \nabla\boldsymbol{u} = \boldsymbol{0}, \qquad \text{in } \Omega\times(0,t_f], \tag{3a}$$

$$\boldsymbol{m}(\boldsymbol{u},\boldsymbol{q},\boldsymbol{x},t,\boldsymbol{\mu})\frac{\partial \boldsymbol{u}}{\partial t} + \nabla\cdot\boldsymbol{f}(\boldsymbol{u},\boldsymbol{q},\boldsymbol{x},t,\boldsymbol{\mu}) = \boldsymbol{s}(\boldsymbol{u},\boldsymbol{q},\boldsymbol{x},t,\boldsymbol{\mu}), \quad \text{in } \Omega\times(0,t_f], \tag{3b}$$

with appropriate initial and boundary conditions. The set of state variables $(\boldsymbol{u},\boldsymbol{q})$ is the exact solution of the Model D (3). `Exasim` produces codes to compute the approximate solution $(\boldsymbol{u}_h,\boldsymbol{q}_h)$.

Examples of the Model D include the Poisson equation, convection-diffusion equations, linear elasticity equations, nonlinear elasticity equations, the incompressible Navier-Stokes equations, and the compressible Navier-Stokes equations.

2.3 Model W: Wave Model

The Model W consists of any set of PDEs that can be written in the following form:

$$\boldsymbol{m}\left(\frac{\partial \boldsymbol{w}}{\partial t},-\nabla\boldsymbol{w},\boldsymbol{w},\boldsymbol{x},t,\boldsymbol{\mu}\right)\frac{\partial^2 \boldsymbol{w}}{\partial t^2} + \nabla\cdot\boldsymbol{f}\left(\frac{\partial \boldsymbol{w}}{\partial t},-\nabla\boldsymbol{w},\boldsymbol{w},\boldsymbol{x},t,\boldsymbol{\mu}\right) = \boldsymbol{s}\left(\frac{\partial \boldsymbol{w}}{\partial t},-\nabla\boldsymbol{w},\boldsymbol{w},\boldsymbol{x},t,\boldsymbol{\mu}\right), \tag{4}$$

with appropriate initial and boundary conditions. The Model W is a generalization of the Model D by having the second-order time derivatives.

It is convenient to introduce additional state variables $\boldsymbol{u} = \frac{\partial \boldsymbol{w}}{\partial t}$, $\boldsymbol{q} = -\nabla \boldsymbol{w}$, and rewrite the Model D as follows

$$\frac{\partial \boldsymbol{w}}{\partial t} - \boldsymbol{u} = \boldsymbol{0}, \qquad \text{in } \Omega \times (0, t_f], \tag{5a}$$

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \boldsymbol{u} = \boldsymbol{0}, \qquad \text{in } \Omega \times (0, t_f], \tag{5b}$$

$$\boldsymbol{m}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{x}, t, \boldsymbol{\mu}) \frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{x}, t, \boldsymbol{\mu}) = \boldsymbol{s}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{x}, t, \boldsymbol{\mu}), \quad \text{in } \Omega \times (0, t_f]. \tag{5c}$$

The set of state variables $(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{w})$ is the exact solution of the Model W (5). `Exasim` produces codes to compute the approximate solution $(\boldsymbol{u}_h, \boldsymbol{q}_h, \boldsymbol{w}_h)$.

The Model W deals with wave propagation problems. Examples of the Model D include the wave equation, linear elastodynamics, nonlinear elastodynamics, and the Maxwell's equations.

2.4 High-Order PDE Models

Elliptic, parabolic, and hyperbolic PDEs of order two are widely used to model physical problems in engineering and science. However, there many other important types of PDE, including the Korteweg-de Vries (KdV) equation and the biharmonic equation. We will show that `Exasim` can deal with high-order PDEs. To this end, we consider the fourth-order biharmonic equation

$$\Delta^2 u = f(\boldsymbol{x}), \tag{6}$$

with appropriate boundary conditions. We introduce a new variable $v = -\Delta u$ and write the fourth-order biharmonic equation as a set of two coupled Poisson equations as follows

$$-\Delta v = f, \tag{7a}$$

$$-\Delta u = v. \tag{7b}$$

We next define $\boldsymbol{u} = [v, u]$, $\boldsymbol{s} = [f, v]$, $\boldsymbol{q} = -\nabla \boldsymbol{u}$, and rewrite the above system as follows

$$\boldsymbol{q} + \nabla \boldsymbol{u} = 0, \tag{8a}$$

$$\nabla \cdot \boldsymbol{q} = \boldsymbol{s}(\boldsymbol{u}, \boldsymbol{x}). \tag{8b}$$

This set of PDEs belongs to the Model D (3).

In general, high-order PDEs can be written as a set of first-order PDEs by introducing new state variables. Indeed, the Model C (1), Model D (3), and Model W (5) are nothing but a set of first-order PDEs.

2.5 PDE Model File

Practitioners write a PDE model file to define a parametrized PDE model to be solved. This involves writing *mass*, *flux*, *source*, *ubou*, *fbou*, *initu*, *initq*, *initw*, and *initv* functions in terms of the state variables $(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{w})$, spatial variables $\boldsymbol{x}$, time variable t, and physical parameters $\boldsymbol{\mu}$, to define governing equations and boundary conditions. We extend these functions with two new variables, $\boldsymbol{v}$ and $\boldsymbol{\eta}$, which represent the external fields and external parameters, respectively. This extension allows a mean to coupling `Exasim` with an external code through the external fields $\boldsymbol{v}$ and external parameters $\boldsymbol{\eta}$.

- The first three functions, *mass*, *flux*, and *source*, implement the mass $\boldsymbol{m}$, flux $\boldsymbol{f}$, and source $\boldsymbol{s}$, respectively.
- The next two functions, *ubou*, and *fbou*, implement the boundary values for the solution $\boldsymbol{u}$ and the normal component of the flux $\boldsymbol{f}_b = \boldsymbol{f} \cdot \boldsymbol{n}$, respectively. Both *ubou*, and *fbou* also depends on the trace variables $\widehat{\boldsymbol{u}}$, the normal vector $\boldsymbol{n}$, and the DG stabilization parameter τ (see Section 6 for details.)
- The last four functions, *initu*, *initq*, *initw*, and *initv*, implement the initial values for $\boldsymbol{u}$, $\boldsymbol{q}$, $\boldsymbol{w}$, and $\boldsymbol{v}$, respectively.

Below is a "blank" PDE model file which does not define these functions yet. It gives a sense of how these functions should be defined. Depending on a particular PDE model, you need to define these functions accordingly.

```
function mass(u, q, w, v, x, t, mu, eta)
    # define m below
    return m;
end
function flux(u, q, w, v, x, t, mu, eta)
    # define f below
    return f;
end
function source(u, q, w, v, x, t, mu, eta)
    # define s below
    return s;
end
function ubou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    # define ub below
    return ub;
end
function fbou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    # define fb below
    return fb;
end
function initu(x, mu, eta)
    # define u0 below
    return u0;
end
function initq(x, mu, eta)
    # define q0 below
    return q0;
end
function initw(x, mu, eta)
    # define w0 below
    return w0;
end
function initv(x, mu, eta)
    # define v0 below
```

```
    return v0;
end
```

Let's consider the Poisson equation $-\nabla \cdot \mu_1 \nabla u = 3\pi^2 \sin(\pi x_1)\sin(\pi x_2)\sin(\pi x_3)$ rewritten as

$$\nabla \cdot \mu_1 \boldsymbol{q} = 3\pi^2 \sin(\pi x_1)\sin(\pi x_2)\sin(\pi x_3), \qquad \boldsymbol{q} + \nabla u = 0, \tag{9}$$

in a physical domain Ω with $u = \mu_2$ on $\partial\Omega$. The PDE model file for the above Poisson equation is listed below

```
function flux(u, q, w, v, x, t, mu, eta)
    f = mu[1]*q;
    return f;
end
function source(u, q, w, v, x, t, mu, eta)
    s = (3*pi*pi)*sin(pi*x[1])*sin(pi*x[2])*sin(pi*x[3]);
    return s;
end
function ubou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    ub = mu[2];
    return ub;
end
function fbou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    f = flux(u, q, w, v, x, t, mu, eta);
    fb = f[1]*n[1] + f[2]*n[2] + f[3]*n[3] + tau[1]*(u[1]-uhat[1]);
    return fb;
end
function initu(x, mu, eta)
    u0 = 0.0;
    return u0;
end
```

```
function f = flux(u, q, w, v, x, t, mu, eta)
    f = mu[1]*q;
end
function s = source(u, q, w, v, x, t, mu, eta)
    s = (3*pi*pi)*sin(pi*x[1])*sin(pi*x[2])*sin(pi*x[3]);
end
function ub = ubou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    ub = mu[2];
end
function fb = fbou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    f = flux(u, q, w, v, x, t, mu, eta);
    fb = f[1]*n[1] + f[2]*n[2] + f[3]*n[3] + tau[1]*(u[1]-uhat[1]);
end
function u0 = initu(x, mu, eta)
    u0 = 0.0;
end
```

```
template <typename T>  __global__  void kernelgpuFlux(T *f, T *xdg, T *udg, T *odg,
T *wdg, T *uinf, T *param, T time, int ng, int nc, int ncu, int nd, int ncx, int nco, int ncw)
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   while (i<ng) {
        T param1 = param[0];
        T udg2 = udg[1*ng+i];
        T udg3 = udg[2*ng+i];
        T udg4 = udg[3*ng+i];
        f[0*ng+i] = param1*udg2;
        f[1*ng+i] = param1*udg3;
      f[2*ng+i] = param1*udg4;
   i += blockDim.x * gridDim.x;
   }
}
```

```
template <typename T>  __global__
void  kernelPairs(T *U, T *rij, T *param,  T *qi,
      T *qj,  int *ei, int *ej,  int npairs)
{
   int t = threadIdx.x + blockIdx.x * blockDim.x;
   while (t < npairs) {
      T r  = rij[t];
      T r2 = r*r;
      T r4 = r2*r2;
      T r6 = r2*r4;
      T r12 = r6*r6;
      U[t] = param[0]*(param[1]/r12 - param[2]/r6);
      t += blockDim.x * gridDim.x;
   }
}
```

Since this particular equation has zero mass function, i.e., $\boldsymbol{m} = 0$, we do not need to define the *mass* function in the PDE model file. This also applies to the *source* function, that is, if $\boldsymbol{s} = 0$, then we do not need to define the *source* function in the PDE model file. The initial guess for the solution u of the Poisson equation is implemented in the function *initu*.

Instead of the Poisson equation if we would like to solve the following heat equation

$$\mu_3 \frac{\partial u}{\partial t} + \nabla \cdot \mu_1 \boldsymbol{q} = 3\pi^2 \sin(\pi x_1)\sin(\pi x_2)\sin(\pi x_3), \qquad \boldsymbol{q} + \nabla u = 0, \tag{10}$$

with the initial solution $u(\boldsymbol{x}, t = 0) = 0$, then we need to add the following snippet to the above PDE model file

```
function mass(u, q, w, v, x, t, mu, eta)
    m = mu[3];
    return m;
end
```

Next, we assume that we want to solve the following 2D convection-diffusion equation

$$\mu_2 \frac{\partial u}{\partial t} + \nabla \cdot (\mu_1 \boldsymbol{q} + \boldsymbol{c} u) = 0, \qquad \boldsymbol{q} + \nabla u = 0, \tag{11}$$

with two boundary conditions $u = 0$ on Γ_1 and $(\mu_1 \boldsymbol{q} + cu) \cdot \boldsymbol{n} = -1$ on Γ_2 and the initial solution $u(\boldsymbol{x}, t = 0) = \mu_3$. Here $\boldsymbol{c} = (x_2, -x_1)$ is the convective velocity. The PDE model file for the above equation is listed below

```
function mass(u, q, w, v, x, t, mu, eta)
    m = mu[2];
    return m;
end
function flux(u, q, w, v, x, t, mu, eta)
    f = 0.0*q;
    f[1] = mu[1]*q[1] + x[2]*u[1];
    f[2] = mu[1]*q[2] - x[1]*u[1];
    return f;
end
function ubou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    ub = [0*u[1], 0*u[1]];
    ub[1] = 0;      # Dirichlet boundary condition
    ub[2] = u[1];   # Neumann boundary condition
    return ub;
end
function fbou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    fb = [0*u[1], 0*u[1]];
    f = flux(u, q, w, v, x, t, mu, eta);
    fb[1] = f[1]*n[1] + f[2]*n[2] + tau[1]*(u[1]-uhat[1]);   # Dirichlet
    fb[2] = -1;                                              # Neumann
    return fb;
end
function initu(x, mu, eta)
    u0 = mu[3];
    return u0;
end
```

Now assume that we want to solve the following 2D wave equation $\frac{\partial^2 w}{\partial t^2} - \mu \Delta w = 0$ rewritten as

$$\frac{\partial u}{\partial t} + \nabla \cdot \mu \boldsymbol{q} = 0, \qquad \frac{\partial \boldsymbol{q}}{\partial t} + \nabla u = 0, \qquad \frac{\partial w}{\partial t} - u = 0, \tag{12}$$

with boundary condition $u = 0$ on $\partial\Omega$ and initial conditions $u(\boldsymbol{x}, t = 0) = \sin(\pi x_1) \sin(\pi x_2)$, $\boldsymbol{q}(\boldsymbol{x}, t = 0) = \boldsymbol{0}$, and $w(\boldsymbol{x}, t = 0) = 0$. The PDE model file for the above equation is listed below

```
function mass(u, q, w, v, x, t, mu, eta)
    m = 1.0;
    return m;
end
function flux(u, q, w, v, x, t, mu, eta)
```

```
    f = mu[1]*q;
    return f;
end
function ubou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    ub = 0.0;
    return ub;
end
function fbou(u, q, w, v, x, t, mu, eta, uhat, n, tau)
    f = flux(u, q, w, v, x, t, mu, eta);
    fb = f[1]*n[1] + f[2]*n[2] + tau[1]*(u[1]-0.0);
    return fb;
end
function initu(x, mu, eta)
    u0 = sin(pi*x[1])*sin(pi*x[2]);
    return u0;
end
function initq(x, mu, eta)
    q0 = 0*x;
    return q0;
end
function initw(x, mu, eta)
    w0 = 0.0;
    return w0;
end
```

Thus far, we show how to write PDE model files for *scalar* PDEs. `Exasim` provides PDE model files for the elasticity equations, Euler equations, Navier-Stokes equations, and magnetohydrodynamics (MHD) equations. They can be found in the folder Exasim/Applications.

3 Geometry Model and Finite Element Mesh

3.1 Geometry Model

A physical domain of interest is represented by a geometry model known as boundary representation or BREP. BREP's geometry entities include points, curves, surfaces, and volumes: a volume is bounded by a set of surfaces, a surface is bounded by a set of curves, and a curve is bounded by two end points. `Exasim` uses Gmsh's geometry model to represent physical domains. Below is an example of Gmsh's geometry model file lshape.geo for an L-shaped domain shown in Figure 2:

```
mesh_size = 0.2;
Point(1) = {-1.0, -1.0, 0.0, mesh_size };
Point(2) = {0.0, -1.0, 0.0, mesh_size };
Point(3) = {0.0, 0.0, 0.0, mesh_size/10};
Point(4) = {1.0, 0.0, 0.0, mesh_size };
Point(5) = {1.0, 1.0, 0.0, mesh_size };
Point(6) = {-1.0, 1.0, 0.0, mesh_size };
```

```
Line(7) = {1,2};
Line(8) = {2,3};
Line(9) = {3,4};
Line(10) = {4,5};
Line(11) = {5,6};
Line(12) = {6,1};

Line Loop(13) = {7,8,9,10,11,12};

Plane Surface(14) = {13};
```

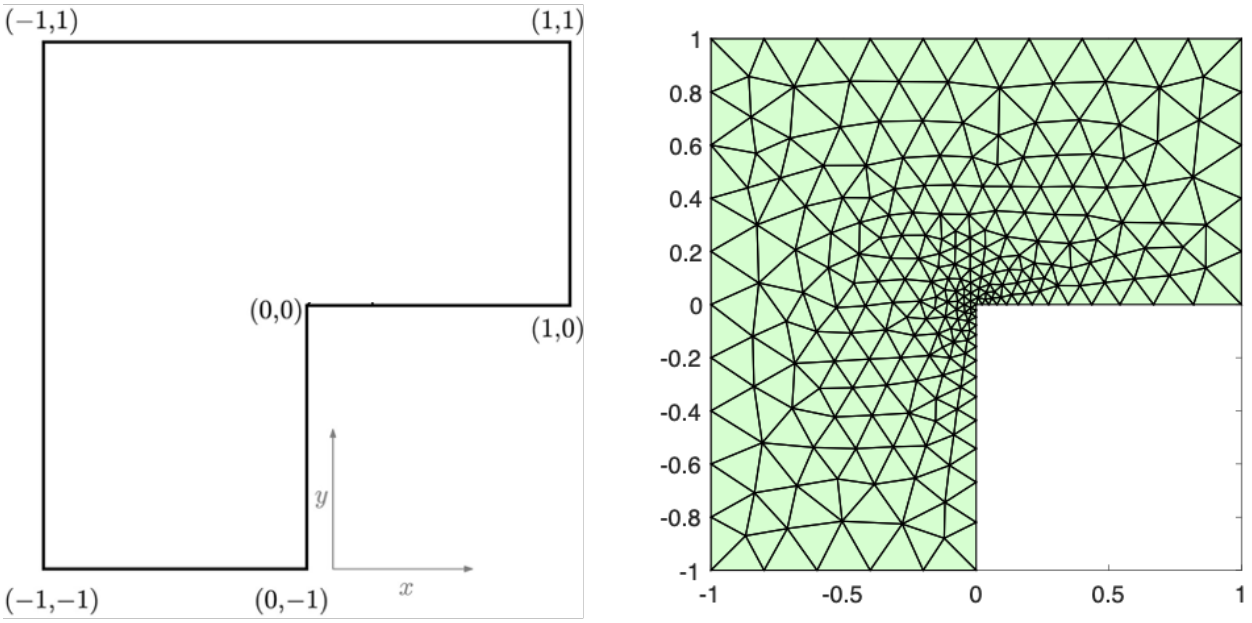

Figure 2: The L-shaped domain and its finite element mesh generated by Gmsh from the geometry model file lshape.geo.

3.2 Finite Element Mesh

A finite element mesh of a geometry model is a tessellation of its geometry by simple geometrical elements of various shapes such as lines, triangles, quadrangles, tetrahedra, prisms, hexahedra and pyramids. `Exasim` can handle conformal finite element meshes of triangular, quadrilateral, tetrahedra, and hexahedra elements. In `Exasim`, a finite element mesh is composed of p and t, where $p \in \mathbb{R}^{n_d \times n_p}$ is a two-dimensional float array storing mesh points and $t \in \mathbb{I}^{n_{ve} \times n_e}$ is a two-dimensional integer array storing mesh elements. Here n_p is the number of mesh points, n_{ve} is the number of vertices of an element, and n_e is the number of elements. There are three different ways to input a finite element mesh into `Exasim`. First, `Exasim` has a Mesh module to generate meshes for some simple geometries.

Second, `Exasim` uses Gmsh to generate meshes from a geometry model file and mesh size parameters specified in the geometry model file, says filename.geo, through the following function

```
# Call Gmsh to generate a mesh for a domain described in the file filename.geo
p, t = Mesh.gmshcall(pde, "filename", nd, elemtype);
```

where nd is the dimensionality of the physical domain and elemtype denotes the type of elements. In Exasim, elemtype = 0 means triangles in 2D and tetrahedra in 3D, and elemtype = 1 means quadrilaterals in 2D and hexahedra in 3D. Note that file extension ".geo" must be excluded. Figure 2 shows a finite element mesh generated by Gmsh for the L-shaped domain defined in lshape.geo.

And third, a finite element mesh can be imported into Exasim via either a text file or binary file as follows

```
# Read a mesh from an input file
p, t = Mesh.readmesh("filename.ext", mode);
```

Here mode can be either 0 (binary) or 1 (ascii). Both a filename and its extension must be provided. Below is the format of Exasim's text mesh file

```
nd np nve ne
p[1,1] p[1,2] ... p[1,nd]
p[2,1] p[2,2] ... p[2,nd]
...
p[np,1] p[np,2] ... p[np,nd]
t[1,1] t[1,2] ... t[1,nve]
t[2,1] t[2,2] ... t[2,nve]
...
t[ne,1] t[ne,2] ... t[ne,nve]
```

The first line of the mesh file consists of four integers which are n_d, n_p, n_{ve}, and n_e, respectively. The next n_p lines store the coordinates for each of the mesh points. The last n_e lines store the element connectivities for each of the mesh elements. The format of the binary mesh file follows that of the text mesh file. It is important to note that all the entries in the binary mesh file are treated as double (float64) type. Exasim will read the t array from the binary file as double array and convert it into integer array.

3.3 High-Order Finite Element Mesh

The array tuple (p,t) represents a standard finite element mesh. Since Exasim uses high-order DG methods for spatial discretization of PDEs, it creates a high-order mesh from a standard finite element mesh (p,t). In Exasim, a high-order mesh has the following data structure:

```
mutable struct MeshStruct
    p::Array{Float64,2};        # points of a linear mesh
    t::Array{Int64,2};          # elements of a linear mesh
    boundaryexpr;              # expressions to determine boundaries
    boundarycondition::Array{Int64,2}; # a list of boundary conditions
    curvedboundary::Array{Int64,2}; # boolean flags for curved boundaries
    curvedboundaryexpr;        # expressions to determine curved boundaries
    periodicexpr;               # expressions to map periodic boundaries
```

```
    f::Array{Int64,2};          # faces of a linear mesh
    tprd::Array{Int64,2};       # elements for periodic conditions
    dgnodes::Array{Float64,3}; # spatial nodes of a high-order mesh
end
```

Here boundaryexpr = $[b_1(\boldsymbol{x}), b_2(\boldsymbol{x}), \ldots, b_{n_{bc}}(\boldsymbol{x})]$ is a *priority* list of n_{bc} user-specified boolean functions to divide the whole physical boundary $\partial\Omega$ into n_{bc} disjoint boundaries $\Gamma_j, 1 \leq j \leq n_{bc}$, such that $\overline{\partial\Omega} = \cup_{j=1}^{n_{bc}} \overline{\Gamma}_j$. In particular, if any point $\boldsymbol{x} \in \partial\Omega$ satisfies $b_1(\boldsymbol{x})$ == True, then it belongs to Γ_1. Next, if $\boldsymbol{x} \in \partial\Omega$ satisfies $b_2(\boldsymbol{x})$ == True, then it belongs to Γ_2. So on, until if $\boldsymbol{x} \in \partial\Omega$ satisfies $b_{n_{bc}}(\boldsymbol{x})$ == True, then it belongs to $\Gamma_{n_{bc}}$. Note that if $\boldsymbol{x} \in \partial\Omega$ satisfies $b_2(\boldsymbol{x})$ == True, $b_4(\boldsymbol{x})$ == True, and $b_5(\boldsymbol{x})$ == True, then it belongs to Γ_2. In other words, the first boolean function gets the first priority and the last boolean function gets the last priority. This allows us to apply boundary conditions to disjoint boundaries $\Gamma_j, 1 \leq j \leq n_{bc}$. Note that each disjoint boundary accepts only one boundary condition.

Next, boundarycondition = $[c_1, c_2, \ldots, c_{n_{bc}}]$ is an integer array of n_{bc} entries which determine a boundary condition for each disjoint boundary. Specifically, disjoint boundary Γ_j accepts boundary condition c_j. Assume that we divide the domain boundary $\partial\Omega$ into 4 disjoint boundaries $\Gamma_1, \ldots, \Gamma_4$ and that boundarycondition = [2, 3, 1, 2]. In this case, Γ_1 accepts boundary condition 2, Γ_2 accepts boundary condition 3, Γ_3 accepts boundary condition 1, and Γ_4 accepts boundary condition 2. Note that boundary condition 1, boundary condition 2, and boundary condition 3 must be implemented in *ubou* and *fbou* functions as discussed in the previous section. In other words, boundarycondition is tied to the implementation of boundary conditions in *ubou* and *fbou* functions.

Next, curvedboundary = $[d_1, d_2, \ldots, d_{n_{bc}}]$ is a boolean array of n_{bc} entries which determine whether a disjoint boundary is curved or straight. Specifically, disjoint boundary Γ_j is curved if $d_j = 1$ or straight if $d_j = 0$. And curvedboundaryexpr = $[s_1(\boldsymbol{x}), s_2(\boldsymbol{x}), \ldots, s_{n_{bc}}(\boldsymbol{x})]$ is a list of n_{bc} functions that express the curved equation $s_j(\boldsymbol{x}) = 0$ for the disjoint boundary Γ_j. If Γ_j is straight then both d_j and $s_j(\boldsymbol{x})$ are set to 0. Otherwise, if Γ_j is curved then d_j is set to 1 and the function $s_j(\boldsymbol{x})$ must be specified. For example, if Γ_j is a unit circle, then $s_j(\boldsymbol{x}) = x_1^2 + x_2^2 - 1$. `Exasim` uses curvedboundary and curvedboundaryexpr to create high-order elements for the curved boundaries. Figure 3 shows an example of a standard finite element mesh and a high-order mesh generated by `Exasim`.

Next, periodicexpr = $[[e_{11}, p_{11}(\boldsymbol{x}), e_{12}, p_{12}(\boldsymbol{x})], \ldots, [e_{n_{periodic}1}, p_{n_{periodic}1}(\boldsymbol{x}), e_{n_{periodic}2}, p_{n_{periodic}2}(\boldsymbol{x})]]$ is a two-dimensional list of $n_{periodic} \times 4$ entries which determine a set of $n_{periodic}$ periodic boundary conditions. Each periodic boundary condition requires two disjoint boundaries. When two disjoint boundaries are periodic with each other, there must be a one-to-one relationship for every mesh point on these two boundaries. This requirement must be met by a standard finite element mesh (p, t). Hence, periodic boundary conditions place constraints on generating mesh points on periodic boundaries. For example, if periodicexpr = $[[1, p_{11}(\boldsymbol{x}), 3, p_{12}(\boldsymbol{x})], [2, p_{21}(\boldsymbol{x}), 4, p_{22}(\boldsymbol{x})]]$, then Γ_1 (respectively, Γ_2) is periodic to Γ_3 (respectively, Γ_4) and any mesh point on Γ_1 (respectively, Γ_2) can be mapped to a mesh point on Γ_3 (respectively, Γ_4) by a mapping function, vice versa. The mapping functions $p_{11}(\boldsymbol{x})$ and $p_{12}(\boldsymbol{x})$, (respectively, $p_{21}(\boldsymbol{x})$ and $p_{21}(\boldsymbol{x})$) determine how two disjoint boundaries Γ_1 and Γ_3 (respectively, Γ_2 and Γ_4) are periodic each other. For example, in two dimensions, if $p_{11}(\boldsymbol{x}) = x_2$ and $p_{12}(\boldsymbol{x}) = x_2$ then every mesh point on Γ_1 must have its second coordinate equal to the second coordinate of one and only one mesh point on Γ_3. In three dimensions, if $p_{11}(\boldsymbol{x}) = [x_2, x_3]$ and $p_{12}(\boldsymbol{x}) = [x_2, x_3]$ then Γ_1 and Γ_3 are periodic with respect to (x_2, x_3) coordinates.

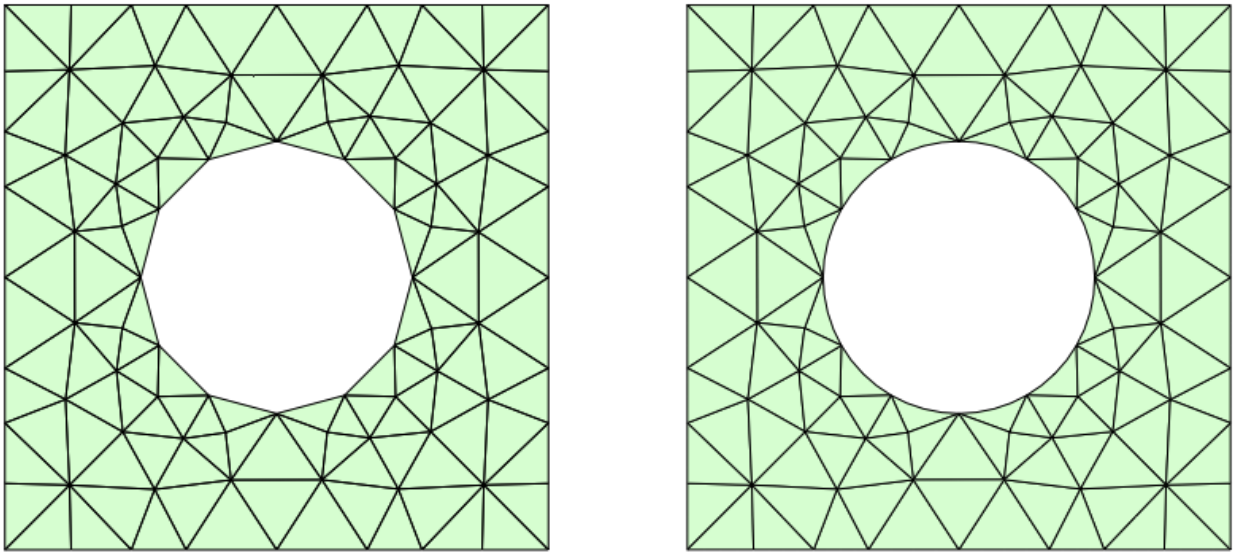

Figure 3: Standard finite element mesh (left) and high-order mesh (right) for a domain bounded by a unit circle and a square.

Next, f is an integer array of size $n_{fe} \times n_e$, where n_{fe} is the number of faces of an element. It indicates if a face is inside the physical domain or on disjoint boundaries. Note that f is determined by `Exasim` from mesh.boundaryexpr.

Next, tprd is an integer array of size $n_{ve} \times n_e$. So, tprd is the same size as t and contains element connectivities for handling periodic boundary conditions. When there are periodic boundaries, the element connectivities need to be updated by `Exasim`.

Finally, dgnodes is a float array of size $n_{pe} \times n_d \times n_e$ storing mesh points to represent high-order elements, where n_{pe} is the number of polynomials per element. Note that dgnodes depends on the polynomial degree and the element shape, and that it is computed by `Exasim`. The ordering of mesh points for the high-order master element is based on the following rule: mesh points along the first coordinate x_1 are listed before those along the second x_2 and third x_3 coordinates; and mesh points along the second coordinate x_2 are listed before those along the third x_3 coordinate.

4 Exasim

In this section, we describe how `Exasim` generate executable DG codes to solve parametrized PDE models. This is done by writing a script file in Julia, Python, or Matlab. Below is an example of a script file pdeapp.jl for solving the 3D Poisson equation (9).

4.1 3D Poisson Equation as "Hello, World"

```
# specify an Exasim version to run
version = "Version0.1";

# External modules
using Revise, DelimitedFiles, SymPy
```

```
# Add Exasim to Julia search path
cdir = pwd(); ii = findlast("Exasim", cdir);
include(cdir[1:ii[end]] * "/Installation/setpath.jl");

# Exasim modules
using Preprocessing, Mesh, Gencode, Postprocessing

# create pde structure and mesh structure
pde, mesh = Preprocessing.initializeexasim(version);

# Define PDE model: governing equations, initial and boundary conditions
pde.model = "ModelD";              # ModelC, ModelD, ModelW
include("pdemodel.jl");            # include the PDE model file

# Set discretization parameters, physical parameters, and solver parameters
pde.porder = 3;                    # polynomial degree
pde.physicsparam = [1.0 0.0];      # thermal conductivity and boundary value
pde.tau = [1.0];                   # DG stabilization parameter

# Choose computing platform and set number of processors
#pde.platform = "gpu";             # choose this option if running on Nvidia GPUs
pde.mpiprocs = 2;                  # number of MPI processors

# create a linear mesh of 8 by 8 by 8 hexes on a unit cube
mesh.p,mesh.t = Mesh.cubemesh(8,8,8,1);
# expressions for disjoint boundaries
mesh.boundaryexpr = [p->(p[2,:] .< 1e-3), p->(p[1,:] .> 1-1e-3), p->(p[2,:] .>␣
 ↪1-1e-3), p->(p[1,:] .< 1e-3), p->(p[3,:] .< 1e-3), p->(p[3,:] .> 1-1e-3)];
mesh.boundarycondition = [1 1 1 1 1 1]; # Set boundary conditions

# call Exasim to generate and run C++ code to solve the PDE model
sol, pde, mesh, ~,~,~,~  = Postprocessing.exasim(pde,mesh);

# visualize the numerical solution of the PDE model using Paraview
pde.visscalars = ["temperature", 1];  # list of scalar fields for visualization
pde.visvectors = ["temperature gradient", [2, 3, 4]]; # list of vector fields
mesh.dgnodes = Postprocessing.vis(sol,pde,mesh); # visualize the solution
x = mesh.dgnodes[:,1,:]; y = mesh.dgnodes[:,2,:]; z = mesh.dgnodes[:,3,:];
uexact = sin.(pi*x).*sin.(pi*y).*sin.(pi*z); # exact solution
uh = sol[:,1,:];                             # numerical solution
maxerr = maximum(abs.(uh[:]-uexact[:]));
print("Maximum absolute error: $maxerr\n");
print("Done!");
```

This script file can be found in the folder /Exasim/Applications/Poisson/Poisson3d. In Julia REPL environment, when you go to this directory and run the script file as follows

julia> include("pdeapp.jl")

The executable code **mpiapp** generated by Exasim can be found in the folder app. Exasim runs the code from Julia to solve the 3D Poisson equation (9). If all goes well, you should see the following lines in Julia REPL session.

```
generate code...
compile code...
run code...
Using 2 processors to solve the problem on CPU platform...
Old RHS Norm: 0.0930155,  New RHS Norm: 0.0930155
GMRES converges to the tolerance 0.001 within  46 iterations and 0 RB dimensions
PTC Iteration: 1,  Residual Norm: 9.14294e-05
Old RHS Norm: 9.14294e-05,  New RHS Norm: 8.56196e-05
GMRES converges to the tolerance 0.001 within  113 iterations and 1 RB dimensions
PTC Iteration: 2,  Residual Norm: 8.6313e-08
Maximum absolute error: 4.792391576413646e-5
Done!
```

In addition, Exasim will open Paraview and visualize the numerical solution if Paraview is already installed on your computer. If Paraview is not installed, please install it and and add the following line

```
# visualize the numerical solution of the PDE model using Paraview
pde.paraview = "/path/to/paraview/executable";
```

to the pdeapp script. Here "/path/to/paraview/executable" is the path to the Paraview executable file. For example, on MacOS systems, it can be "/Applications/ParaView-5.8.1.app/Contents/MacOS/paraview" for Paraview version 5.8.1. Alternatively, you open the file initializepde.jl in the folder /Exasim/Version0.1/Julia/Preprocessing and replace pde.paraview = "paraview" with the above line of code. If you use Matlab or Python, you can do likewise.

Once Paraview opens, click the Apply button and select "temperature gradient", you should see Figure 4.

As mentioned earlier, Exasim provides many examples that illustrate how to generate DG codes for solving a wide variety of PDEs including Poisson equation, wave equation, heat equation, advection, convection-diffusion, elasticity, Euler equations, Navier-Stokes equations, and MHD equations. These examples are placed in the folder Exasim/Applications.

4.2 Backward Compatibility

Exasim is purposefully designed to make the code generator compatible with all of its future versions. The following line is placed at the top of any application script.

```
# specify an Exasim version
version = "Version0.1";
```

If a new version of Exasim is released in the future, only this line will be modified to the new version, while the rest of the script will stay the same.

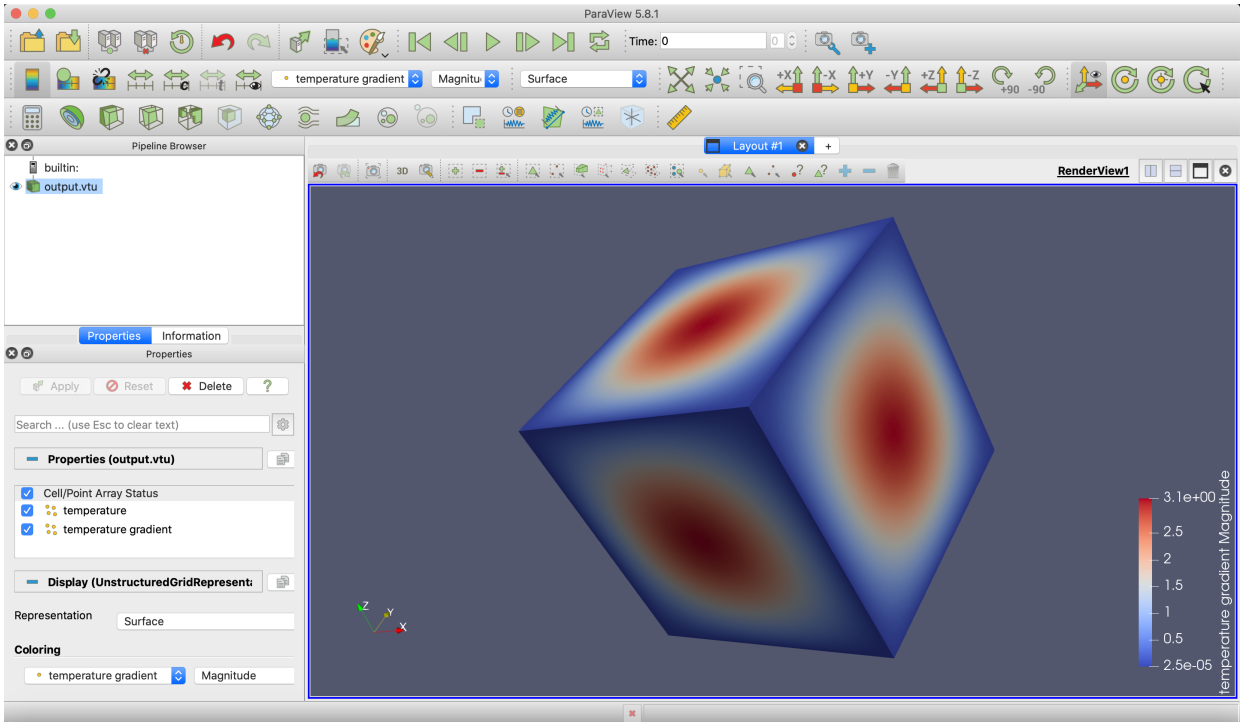

Figure 4: Visualize the numerical solution using Paraview.

4.3 Exasim Modules

Next, the directories of `Exasim` modules are added to search path so that they can be found.

```
# External modules
using Revise, DelimitedFiles, SymPy

# Add Exasim to Julia search path
cdir = pwd(); ii = findlast("Exasim", cdir);
include(cdir[1:ii[end]] * "/Installation/setpath.jl");

# Exasim modules
using Preprocessing, Mesh, Gencode, Postprocessing
```

4.4 PDE Object and Mesh Object

Next, `Exasim` creates an object of PDE structure and an object of Mesh structure. The Mesh Structure is described in Section 3.3, while the PDE structure can be found in the file intializepde. Essentially, `Exasim` sets many default values to the pde object, while the mesh object is empty.

```
# create pde structure and mesh structure
pde, mesh = Preprocessing.initializeexasim(version);
```

4.5 PDE Model File

All application scripts have the previous lines of code and do not require any user's inputs up to this point. Next, we need to define a PDE model by writing a model file as described in Section 2.5. Once a PDE model file is written to express the governing equations, boundary conditions, and initial solutions of a specific PDE model, it must be included in the application script. Furthermore, it is required to set pde.model to the correct PDE model type. As discussed in Section 2.5, `Exasim` supports three types of PDE models.

```
# Define PDE model: governing equations, initial and boundary conditions
pde.model = "ModelD";              # ModelC, ModelD, ModelW
include("pdemodel.jl");            # include the PDE model file
```

4.6 Setting Parameters

Next, we set physical parameters, discretization parameters, and solver parameters. The below parameters are always necessary. First, pde.porder is the degree of polynomials used to approximate the PDE solution on every element. For time-dependent PDEs, pde.dt is a float array consisting of the time steps $\Delta t_1, \Delta t_2, \ldots, \Delta t_{n_{steps}}$. For steady-state PDEs, pde.dt must be set to 0. All physical parameters of the problem can be assigned to pde.physicsparam, which is related to the vector $\boldsymbol{\mu}$ in the PDE model. Here pde.tau is the stabilization parameter τ of the DG scheme (see Section 6). By default, pde.platform is set to "cpu". If you have Nvidia GPUs on your computer, you can set pde.platform to "gpu" to enable the executable application running on GPUs. And pde.mpiprocs is the number of MPI processors used to compute the numerical solution of the PDE model in parallel.

```
# Set discretization parameters, physical parameters, and solver parameters
pde.porder = 3;                    # polynomial degree
pde.dt = [0.0];                    # steady-state problem
pde.physicsparam = [1.0 0.0];      # thermal conductivity and boundary value
pde.tau = [1.0];                   # DG stabilization parameter

# Choose computing platform and set number of processors
#pde.platform = "gpu";             # choose this option if running on Nvidia GPUs
pde.mpiprocs = 2;                  # number of MPI processors
```

For time-dependent PDE models, we need to set a value for pde.nstage and pde.torder, which are the number of stages and the order of accuracy for a diagonally implicit Runge-Kutta (DIRK) scheme, respectively. `Exasim` supports DIRK11, DIRK12, DIRK22, DIRK23, DIRK33, and DIRK34 schemes. There are a number of other parameters that are mostly related to solvers such as the maximum number of Newton iterations (pde.NLiter), Newton tolerance (pde.NLtol), the maximum number of GMRES iterations (pde.linearsolveriter), GMRES tolerance (pde.linearsolvertol), the number of GMRES restart (pde.GMRESrestart), and some other parameters.

4.7 Finite Element Mesh

As mentioned earlier, there are three different ways to bring a finite element mesh into `Exasim`. First, `Exasim` has a Mesh module to generate meshes for some simple geometries. Second, if a

Gmsh's geometry model file is provided, Exasim makes a call to Gmsh to generate a mesh and import that mesh into Exasim. And third, a finite element mesh can be imported into Exasim via either a text file or binary file.

Below is an example of calling a function in Mesh module to generate a mesh.

```
# create a linear mesh of 8 by 8 by 8 hexes on a unit cube
mesh.p,mesh.t = Mesh.cubemesh(8,8,8,1);
```

To use Gmsh to generate a mesh for a physical domain, you need to provide a Gmsh geometry file, says, filename.geo, that describes the domain of interest and sets various mesh sizes. Exasim makes a call to Gmsh to generate a mesh as follows. Note that nd is the dimensionality of the physical domain.

```
# Call Gmsh to generate a mesh for a domain described in the file filename.geo
mesh.p, mesh.t = Mesh.gmshcall(pde, "filename", nd, elemtype);
```

Note that file extension ".geo" must be excluded. Last but not least, you can use your favorite mesh generator to generate a mesh and write that mesh into a text file or a binary file according to the format discussed in Section 3.2. Then Exasim can read that mesh from the file as follows.

```
# Read a mesh from an input file
mesh.p, mesh.t = Mesh.readmesh("filename.ext", mode);
```

Here mode can be either 0 (binary) or 1 (ascii). Both a filename and its extension must be provided.

4.8 Code Generation

Exasim takes the pde object and the mesh object to as input. It then produces binary input files, generates a C++ code, compiles that code, runs the code on your computer, and returns the numerical solution of the PDE model.

```
function exasim(pde, mesh)

# search compilers and set options
pde = Gencode.setcompilers(pde);

# generate input files and store them in datain folder
pde, mesh, master, dmd = Preprocessing.preprocessing(pde,mesh);

# generate source codes and store them in app folder
Gencode.gencode(pde);

# compile source codes to produce an executable file and store it in app folder
compilerstr = Gencode.compilecode(pde);

# run executable file to compute solution and store it in dataout folder
runstr = Gencode.runcode(pde);
```

```
# get solution from output files in dataout folder
sol = Postprocessing.fetchsolution(pde,master,dmd);

return sol,pde,mesh,master,dmd,compilerstr,runstr

end
```

Here sol is a multi-dimensional float array containing the numerical solution of the PDE model. The master struct contains shape functions and quadratures for the master element and master face. The dmd struct contains a domain decomposition of the finite element mesh into subdomains. It is needed to obtain the numerical solution from the binary output files which are generated by running the code. Here compilerstr is a cell array of strings that shows how to compile the source code, while runstr is an array of strings that shows how to run the code.

4.9 Visualization

The following snippet shows how to visualize the numerical solution stored in sol. We assume here that sol contains pressure, velocity, and temperature, and the associated gradients in three dimensions. So, sol has 20 components. The first component of sol is the pressure, the next three components are the velocity fields, the fifth component is the temperature field. The 6th-10th components are their gradients with respect to x_1, 11th-15th components with respect to x_2, and 16th-20th components with respect to x_3.

```
# list of scalar fields for visualization
pde.visscalars = ["pressure", 1, "temperature", 5];
 # list of vector fields for visualization
pde.visvectors = ["velocity field", [2, 3, 4], "pressure gradient", [6, 11, 16],␣
 ↪"x-vel gradient", [7, 12, 17], "y-vel gradient", [8, 13, 18], "z-vel␣
 ↪gradient", [9, 14, 19], "temperature gradient", [10, 15, 20]];
# visualize the numerical solution using Paraview
Postprocessing.vis(sol,pde,mesh);
```

Visualization is carried out using Paraview. Paraview is the most popular software for scientific visualization. It has many useful functionalities for visualizing meshes, scalar fields, vector fields, and tensor fields. It also allows you to postprocess the numerical solution.

4.10 Compiling Options

The compiling options are set by pde.cpuflags and pde.gpuflags. Their default settings are

```
pde.cpuflags = "-O2 -ldl -lm -lblas -llapack";
pde.gpuflags = "-lcudart -lcublas";
```

It is assumed that Blas/Lapack and CUDA libraries are installed and recognized by the C++ compiler. If these libraries are not in your system search path or you may want to try different libraries, you can set pde.cpuflags and pde.gpuflags as follows

```
# use the below options if blas/lapack libary is NOT in the system search path
pde.cpuflags = "-O2 -ldl -lm -Wl,-rpath, /path/to/blaslapack -L/path/to/
 ↪blaslapack -lblas -llapack";
# use the below options if MKL libary is available on your system
pde.cpuflags = "-O2 -ldl -lm -Wl,-rpath, /path/to/MKL -L/path/to/MKL␣
 ↪-lmkl_intel_lp64 -lmkl_sequential -lmkl_core";
```

```
# use the below options if CUDA libaries are NOT in the system search path
pde.gpuflags = "-Wl,-rpath, /path/to/CUDAlibs -L/path/to/CUDAlibs -lcudart␣
 ↪-lcublas";
```

5 Using Exasim on Supercomputers

5.1 Producing C++ Source Code

Instead of running C++ code on your computer, you may want to produce the C++ code, modify and execute it on another computer, cluster, or supercomputer. This can be done by calling Postprocessing.producecode as follows

```
# generate and compile C++ code
compilerstr, pde, mesh, master, dmd = Postprocessing.producecode(pde, mesh);
```

```
function  producecode(pde,mesh)

# generate input files and store them in datain folder
pde, mesh, master, dmd = preprocessing(pde,mesh);

# generate source codes and store them in app folder
Gencode.gencode(pde);

# compile source codes to generate an executable file and store it in app folder
compilerstr = Gencode.compilecode(pde);

return compilerstr,pde,mesh,master,dmd
end
```

5.2 Deploying C++ Source Code

Next, you transfer generated C++ source codes in the app folder, pre-written C++ code in the folder Exasim/Version0.1/Kernel, the core libraries Exasim/Core folder, together with the input files in the datain folder to a remote computer. In the remote computer, go to the Exasim/Core folder and compile the core libraries as follows

```
g++ -fPIC -O3 -c commonCore.cpp
ar rvs commonCore.a commonCore.o
g++ -fPIC -O3 -c opuCore.cpp
```

```
ar rvs opuCore.a opuCore.o
nvcc -D_FORCE_INLINES -O3 -c --compiler-options '-fPIC' gpuCore.cu
ar -rvs gpuCore.a gpuCore.o
```

and copy the generated static libraries to the folder Exasim/Core/Linux. Then move to the app folder and compile the static libraries as follows

```
g++ -fPIC -O3 -c opuApp.cpp
ar -rvs opuApp.a opuApp.o
nvcc -D_FORCE_INLINES -O3 -c --compiler-options '-fPIC' gpuApp.cu
ar -rvs gpuApp.a gpuApp.o
```

To obtain an MPI application running on CPUs, you can compile the source code as follows

```
mpicxx -std=c++11 -D _MPI ../../../../Version0.1/Kernel/Main/main.cpp -o mpiapp .
 ↪./../../../Core/Linux/commonCore.a ../../../../Core/Linux/opuCore.a opuApp.a␣
 ↪-O2 -ldl -lm -lblas -llapack
```

You can now run the code as follows

```
mpirun -np mpiprocs ./mpiapp ../datain/ ../dataout/out
```

To obtain an MPI application running on GPUs, you can compile the source code as follows

```
mpicxx -std=c++11  -D _MPI -D _CUDA ../../../../Version0.1/Kernel/Main/main.cpp␣
 ↪-o gpumpiapp ../../../../Core/Linux/commonCore.a ../../../../Core/Linux/
 ↪gpuCore.a ../../../../Core/Linux/opuCore.a opuApp.a gpuApp.a -O2 -ldl -lm␣
 ↪-lblas -llapack -lcudart -lcublas
```

Depending on the remote computer, you may need to load CUDA and CUDA-aware MPI libraries to compile CUDA code. The MPI/CUDA code can be run as follows

```
mpirun -gpu -np mpiprocs ./gpumpiapp ../datain/ ../dataout/out
```

Instead of using mpirun command to run MPI applications, some supercomputers may have a special command for this purpose.

5.3 Performing Parametric Studies

Exasim makes it relatively easy to carry out parametric studies. The following snippet shows how to execute a parametric study once the executable application was already built.

```
# execute a parametric study by looping over a set of physical parameter vectors
for i = 1:size(paramset,1)
    # set app.physicsparam to a new parameter vector
    pde.physicsparam = paramset[i,:];
    # save the modified app struct into the binary file app.bin
    Preprocessing.writeapp(pde,"datain/app.bin");
```

```
    # run the executable to compute solution for the modified app struct
    Gencode.runcode(pde);
    # get solution from output files in dataout folder
    sol = Postprocessing.fetchsolution(pde,master,dmd);
    # do something with sol
end
```

6 Discontinuous Galerkin Methods

In this section, we describe discontinuous Galerkin methods which are used by `Exasim` to discretize PDEs. `Exasim` allows practitioners to devise DG methods through the implementation of the numerical trace and flux.

6.1 Approximation Spaces

Let $\Omega \subseteq \mathbb{R}^{n_d}$ be a physical domain with Lipschitz boundary $\partial\Omega$. We denote by $\mathcal{T}_h$ a collection of disjoint, regular, k-th degree curved elements K that partition Ω, and set $\partial\mathcal{T}_h := \{\partial K : K \in \mathcal{T}_h\}$ to be the collection of the boundaries of the elements in $\mathcal{T}_h$. We denote by $\mathcal{F}_h$ a collection of disjoint, regular, k-th degree curved faces F that results from $\mathcal{T}_h$. For any $F \in \mathcal{F}_h$, if it belongs to two different elements K^+ and K^- (namely, $F = \partial K^+ \cap \partial K^-$), then F is an *interior* face. If F belongs to only one element, then it is a *boundary* face. Let $\mathcal{P}^k(D)$ denote the space of complete polynomials of degree k on a domain D. Let $L^2(D)$ be the space of square-integrable functions on D, and let ψ_K^k denote the k-th degree parametric mapping from the reference element K_{ref} to some element $K \in \mathcal{T}_h$ in the physical domain. We then introduce the following discontinuous finite element spaces:

$$\boldsymbol{\mathcal{Q}}_h^k = \left\{\boldsymbol{r} \in [L^2(\mathcal{T}_h)]^{n_{cu}\times n_d} \ : \ (\boldsymbol{r}\circ\boldsymbol{\psi}^k)|_K \in [\mathcal{P}^k(K_{ref})]^{n_{cu}\times n_d} \ \ \forall K \in \mathcal{T}_h\right\},$$

$$\boldsymbol{\mathcal{V}}_h^k = \left\{\boldsymbol{w} \in [L^2(\mathcal{T}_h)]^{n_{cu}} \ : \ (\boldsymbol{w}\circ\boldsymbol{\psi}^k)|_K \in [\mathcal{P}^k(K_{ref})]^{n_{cu}} \ \ \forall K \in \mathcal{T}_h\right\}.$$

Next, we define several inner products associated with these finite element spaces as

$$(\boldsymbol{w},\boldsymbol{v})_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}(\boldsymbol{w},\boldsymbol{v})_K = \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{w}\cdot\boldsymbol{v}, \tag{13a}$$

$$(\boldsymbol{W},\boldsymbol{V})_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}(\boldsymbol{W},\boldsymbol{V})_K = \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{W}:\boldsymbol{V}, \tag{13b}$$

$$\langle\boldsymbol{w},\boldsymbol{v}\rangle_{\partial\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\langle\boldsymbol{w},\boldsymbol{v}\rangle_{\partial K} = \sum_{K\in\mathcal{T}_h}\int_{\partial K} \boldsymbol{w}\cdot\boldsymbol{v}, \tag{13c}$$

for $\boldsymbol{w},\boldsymbol{v} \in \boldsymbol{\mathcal{V}}_h^k$, $\boldsymbol{W},\boldsymbol{V} \in \boldsymbol{\mathcal{Q}}_h^k$, where $\cdot$ and $:$ denotes the scalar product and Frobenius inner product, respectively.

Note that the above spaces consist of functions that are continuous inside every element and yet discontinuous across the boundary of any two neighboring elements. In other words, the functions in these spaces are *double-valued* on the *interior* faces of the finite element mesh. Furthermore, they are *multiple-valued* at the vertices of the finite element mesh. The number of function values at any particular vertex is equal to the number of faces connected to that vertex. Therefore, DG methods often have many times more degrees of freedom than continuous finite element methods.

Fortunately, the higher number of degrees of freedom comes with some important beneficial features. DG methods provide a stable high-order discretization of linear convection operators and result in diagonal-block mass matrix. The latter feature makes explicit time integration efficient for (and popular with) DG methods.

6.2 Weak Formulation

We pay our attention to the Model D only, since the DG discretization of the Model C and the Model W follows the same procedure. The DG discretization of the Model D reads as follows: Find $(\boldsymbol{q}_h(t), \boldsymbol{u}_h(t)) \in \boldsymbol{\mathcal{Q}}_h^k \times \boldsymbol{\mathcal{V}}_h^k$ such that

$$\left(\boldsymbol{q}_h, \boldsymbol{r}\right)_{\mathcal{T}_h} + \left(\boldsymbol{u}_h, \nabla \cdot \boldsymbol{r}\right)_{\mathcal{T}_h} - \left\langle \widehat{\boldsymbol{u}}_h, \boldsymbol{r} \cdot \boldsymbol{n} \right\rangle_{\partial\mathcal{T}_h} = 0\,, \tag{14a}$$

$$\left(\boldsymbol{m}\frac{\partial\, \boldsymbol{u}_h}{\partial t}, \boldsymbol{w}\right)_{\mathcal{T}_h} - \left(\boldsymbol{f}, \nabla \boldsymbol{w}\right)_{\mathcal{T}_h} + \left\langle \widehat{\boldsymbol{f}}_h \cdot \boldsymbol{n}, \boldsymbol{w} \right\rangle_{\partial\mathcal{T}_h} - \left(\boldsymbol{s}, \boldsymbol{w}\right)_{\mathcal{T}_h} = 0\,, \tag{14b}$$

for all $(\boldsymbol{r}, \boldsymbol{w}) \in \boldsymbol{\mathcal{Q}}_h^k \times \boldsymbol{\mathcal{V}}_h^k$ and all $t \in [0, t_f)$, as well as

$$\left(\boldsymbol{u}_h|_{t=0} - \boldsymbol{u}_0, \boldsymbol{w}\right)_{\mathcal{T}_h} = 0\,, \tag{14c}$$

for all $\boldsymbol{w} \in \boldsymbol{\mathcal{V}}_h^k$. Here $\widehat{\boldsymbol{u}}_h$ is the numerical trace approximating the solution $\boldsymbol{u}$ on element boundaries, while $\widehat{\boldsymbol{f}}_h$ is the numerical flux approximating the flux $\boldsymbol{f}$ on the element boundaries. Generally, the numerical trace and flux can be *double-valued* on the interior faces.

Different choices of the numerical trace $\widehat{\boldsymbol{u}}_h$ and numerical flux $\widehat{\boldsymbol{f}}_h$ yield different DG methods. They play an important role in the stability and accuracy of the resulting DG method. The key requirement for the numerical trace $\widehat{\boldsymbol{u}}_h$ is that it must be continuous on the faces of the finite element mesh. This requirement means that $\widehat{\boldsymbol{u}}_h^+|_F = \widehat{\boldsymbol{u}}_h^-|_F$ for all $F \in \mathcal{F}_h$, where $\widehat{\boldsymbol{u}}_h^+|_F$ (respectively, $\widehat{\boldsymbol{u}}_h^-|_F$) is the value of $\widehat{\boldsymbol{u}}_h$ on F from K^+ (respectively, K^-). (Note however that $\widehat{\boldsymbol{u}}_h$ has multiple values at the vertices.) The key requirement for the numerical flux $\widehat{\boldsymbol{f}}_h$ is that its normal component must be continuous on the *interior* faces of the finite element mesh, which means that $\widehat{\boldsymbol{f}}_h^+ \cdot \boldsymbol{n}^+|_F + \widehat{\boldsymbol{f}}_h^- \cdot \boldsymbol{n}^-|_F = 0$ for all interior faces $F \in \mathcal{F}_h$. Here $\boldsymbol{n}^+|_F$ (respectively, $\boldsymbol{n}^-|_F$) is the normal vector pointing outward K^+ (respectively, K^-).

6.3 Examples of DG Methods

For the local DG (LDG) method [2], the numerical trace and flux are defined as follows

$$\begin{aligned}
\widehat{\boldsymbol{u}}_h^+ &= \frac{1}{2}(\boldsymbol{u}_h^+ + \boldsymbol{u}_h^-) + (\boldsymbol{u}_h^+ - \boldsymbol{u}_h^-)\boldsymbol{\beta} \cdot \boldsymbol{n}^+, \\
\widehat{\boldsymbol{u}}_h^- &= \frac{1}{2}(\boldsymbol{u}_h^+ + \boldsymbol{u}_h^-) + (\boldsymbol{u}_h^- - \boldsymbol{u}_h^+)\boldsymbol{\beta} \cdot \boldsymbol{n}^-, \\
\widehat{\boldsymbol{f}}_h^+ \cdot \boldsymbol{n}^+ &= \frac{1}{2}\left(\boldsymbol{f}(\boldsymbol{u}_h^+, \boldsymbol{q}_h^+) + \boldsymbol{f}(\boldsymbol{u}_h^-, \boldsymbol{q}_h^-)\right) \cdot \boldsymbol{n}^+ + \boldsymbol{\tau} \cdot (\boldsymbol{u}_h^+ - \boldsymbol{u}_h^-), \\
\widehat{\boldsymbol{f}}_h^- \cdot \boldsymbol{n}^- &= \frac{1}{2}\left(\boldsymbol{f}(\boldsymbol{u}_h^+, \boldsymbol{q}_h^+) + \boldsymbol{f}(\boldsymbol{u}_h^-, \boldsymbol{q}_h^-)\right) \cdot \boldsymbol{n}^- + \boldsymbol{\tau} \cdot (\boldsymbol{u}_h^- - \boldsymbol{u}_h^+),
\end{aligned} \tag{15}$$

where $\boldsymbol{\beta}$ is a single-valued vector and $\boldsymbol{\tau}$ is a single-valued matrix. It is obvious that the above numerical trace and flux satisfy the requirements mentioned earlier. The LDG method (15) can be

succinctly rewritten as follows

$$\begin{aligned}\widehat{\boldsymbol{u}}_h &= \frac{1}{2}(\boldsymbol{u}_h + \boldsymbol{u}_h^-) + (\boldsymbol{u}_h - \boldsymbol{u}_h^-)\boldsymbol{\beta}\cdot\boldsymbol{n},\\ \widehat{\boldsymbol{f}}_h\cdot\boldsymbol{n} &= \frac{1}{2}\left(\boldsymbol{f}(\boldsymbol{u}_h,\boldsymbol{q}_h) + \boldsymbol{f}(\boldsymbol{u}_h^-,\boldsymbol{q}_h^-)\right)\cdot\boldsymbol{n} + \boldsymbol{\tau}\cdot(\boldsymbol{u}_h - \boldsymbol{u}_h^-),\end{aligned} \tag{16}$$

where $(\boldsymbol{u}_h, \boldsymbol{q}_h)$ are the numerical solution on the element K, while $(\boldsymbol{u}_h^-, \boldsymbol{q}_h^-)$ are the numerical solution on the neighboring element K^- that shares a face with K. Because the LDG method (16) is implemented in `Exasim`, practitioners do not need to define it in the application script. Note that `Exasim`'s default implementation sets $\boldsymbol{\beta} = \boldsymbol{0}$ and $\boldsymbol{\tau} = \tau\boldsymbol{I}$, where τ is specified by practitioners in the application script.

The hybridized DG (HDG) method [1, 3, 4] is a DG method with the following form of the numerical trace and flux:

$$\begin{aligned}\widehat{\boldsymbol{u}}_h &= \frac{1}{2}(\boldsymbol{u}_h + \boldsymbol{u}_h^-) + (\boldsymbol{u}_h - \boldsymbol{u}_h^-)\boldsymbol{\beta}\cdot\boldsymbol{n} + \gamma(\boldsymbol{q}_h - \boldsymbol{q}_h^-)\cdot\boldsymbol{n},\\ \widehat{\boldsymbol{f}}_h\cdot\boldsymbol{n} &= \frac{1}{2}\left(\boldsymbol{f}(\widehat{\boldsymbol{u}}_h,\boldsymbol{q}_h) + \boldsymbol{f}(\widehat{\boldsymbol{u}}_h,\boldsymbol{q}_h^-)\right)\cdot\boldsymbol{n} + \boldsymbol{\tau}\cdot(\boldsymbol{u}_h - \boldsymbol{u}_h^-),\end{aligned} \tag{17}$$

where γ is a single-valued scalar function. The HDG method has a number of advantages over the LDG method including optimal convergence of $\boldsymbol{q}_h$ and super-convergence of $\boldsymbol{u}_h$.

6.4 Implementing Other DG Methods

`Exasim` allows practitioners to implement their own DG methods by explicitly defining the numerical trace and flux in the PDE model file. This is done by writing *uhat* function to define $\widehat{\boldsymbol{u}}_h$ and *fhat* function to define $\widehat{\boldsymbol{f}}_h\cdot\boldsymbol{n}$. These functions have the following format

```
# define the numerical trace: uhat
uhat(u, q, w, v, x, t, mu, eta, uhat, n, tau, um, qm, wm, vm);
# define the normal compoment of the numerical flux: fhat dot n
fhat(u, q, w, v, x, t, mu, eta, uhat, n, tau, um, qm, wm, vm);
```

Here (um, qm, wm, vm) represents $(\boldsymbol{u}_h^-, \boldsymbol{q}_h^-, \boldsymbol{w}_h^-, \boldsymbol{v}_h^-)$. In addition, the following flags need to be set

```
pde.extUhat = 1; # Exasim uses uhat defined in the PDE model file
pde.extFhat = 1; # Exasim uses fhat defined in the PDE model file
```

They will make `Exasim` use practitioners' implementation instead of the default implementation. This feature gives practitioners freedom to create new DG methods.

6.5 Implementing Boundary Conditions

DG methods define the numerical trace and flux on the *interior* faces. It remains to define them on the *boundary* faces to complete the DG discretization of the Model D. In `Exasim`, it is done by writing *ubou* function to define $\widehat{\boldsymbol{u}}_h$ and *fbou* function to define $\widehat{\boldsymbol{f}}_h\cdot\boldsymbol{n}$. The definition of these quantities on the boundary faces depends on the boundary conditions.

To impose a Dirichlet boundary condition $\boldsymbol{u} = \boldsymbol{g}_D$ on $\partial\Omega$, we need to set $\widehat{\boldsymbol{u}}_h$ to the Dirichlet data $\boldsymbol{g}_D$ on the boundary. Therefore, the *ubou* function returns $\boldsymbol{g}_D$. The normal component of the numerical flux on the boundary is given by

$$\widehat{\boldsymbol{f}}_h \cdot \boldsymbol{n} = \boldsymbol{f}(\boldsymbol{u}_h, \boldsymbol{q}_h) \cdot \boldsymbol{n} + \boldsymbol{\tau} \cdot (\boldsymbol{u}_h - \widehat{\boldsymbol{u}}_h). \tag{18}$$

This should be implemented in the *fbou* function.

To impose a Neumann boundary condition $\boldsymbol{f} \cdot \boldsymbol{n} = \boldsymbol{g}_N$ on $\partial\Omega$, we need to set $\widehat{\boldsymbol{f}}_h \cdot \boldsymbol{n}$ to the Neumann data $\boldsymbol{g}_N$ on the boundary. Therefore, the *fbou* function returns $\boldsymbol{g}_N$. Furthermore, we need to set $\widehat{\boldsymbol{u}}_h = \boldsymbol{u}_h$ in the *ubou* function.

When both Dirichlet and Neumann boundary conditions are imposed on a boundary, we set appropriate components of $\widehat{\boldsymbol{u}}_h$ to the Dirichlet data and determine the same components of $\widehat{\boldsymbol{f}}_h \cdot \boldsymbol{n}$ by (18). Then we set the other components of $\widehat{\boldsymbol{f}}_h \cdot \boldsymbol{n}$ to the Neumann data and set the other components of $\widehat{\boldsymbol{u}}_h$ to the corresponding components of $\boldsymbol{u}_h$.

References

[1] B. Cockburn, J. Gopalakrishnan, and R. Lazarov. Unified Hybridization of Discontinuous Galerkin, Mixed, and Continuous Galerkin Methods for Second Order Elliptic Problems. *SIAM Journal on Numerical Analysis*, 47(2):1319–1365, jan 2009.

[2] B. Cockburn and C.-W. Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, dec 1998.

[3] P. Fernandez, N. C. Nguyen, and J. Peraire. The hybridized discontinuous Galerkin method for implicit large-eddy simulation of transitional turbulent flows. *Journal of Computational Physics*, 336:308–329, 2017.

[4] N. C. Nguyen and J. Peraire. Hybridizable discontinuous Galerkin methods for partial differential equations in continuum mechanics. *Journal of Computational Physics*, 231(18):5955–5988, jul 2012.