



# Partial Differential Equation Application Builder For Extreme Scalable Simulations

---

The documentation for Exasim 0.1  
Designed by N. C. Nguyen and J. Peraire  
Developed and written by N. C. Nguyen  
September 7, 2020

# 1 Overview

Exasim is an open-source software for generating discontinuous Galerkin codes and building executable applications 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 create *parametrized* PDE models and automatically produce high-performance applications. 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.

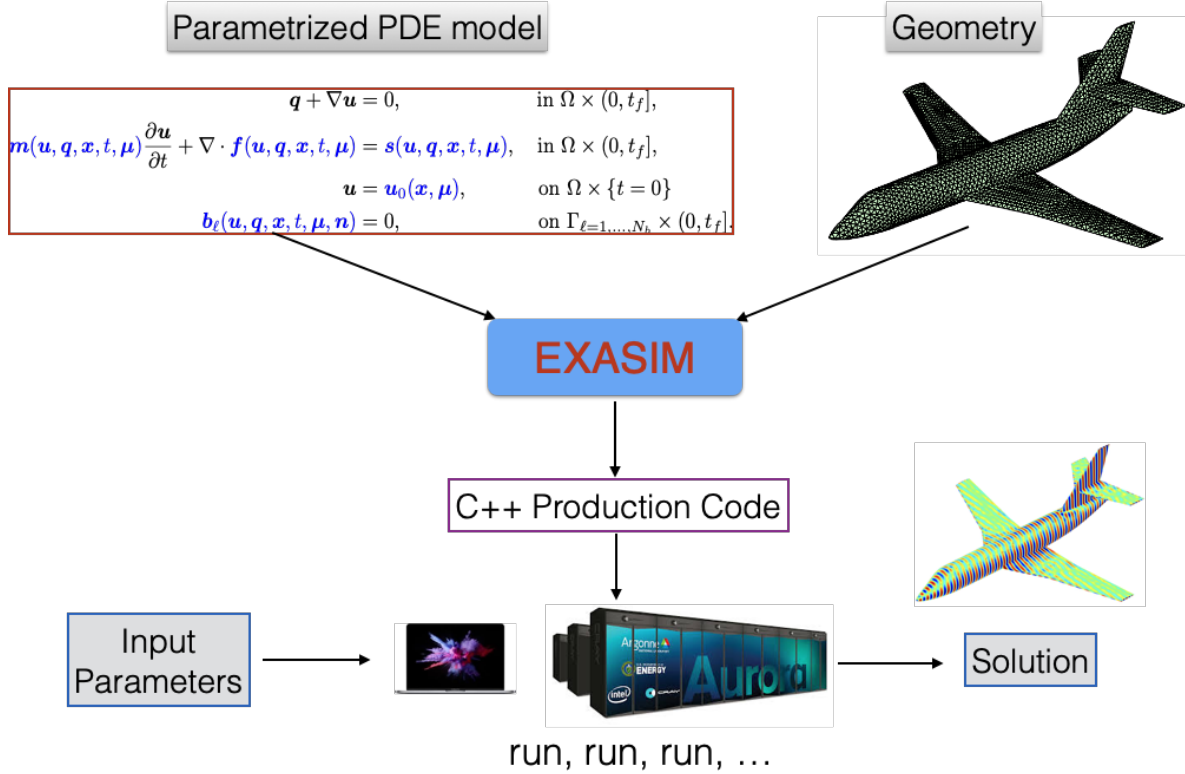


Figure 1: A parametrized PDE model is input into Exasim as functions (written in blue colors) of the field variables, spatial coordinates, time, and physical parameters. A geometry model or a finite element mesh is needed to describe the physical domain. Exasim generates C++ code to solve the parametrized 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 GNU Lesser General Public License, version 2.1. Please see the [license details](#) for terms and conditions. Exasim does not require installation.

## 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 build serial applications. An MPI library such as [Open-MPI](#) is required to build parallel applications. [CUDA Toolkit](#) is required to build applications on Nvidia GPUs. [Gmsh](#) is used for mesh generation. [METIS](#) is needed for mesh partition. And [Paraview](#) is needed for visualization.

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
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
< /dev/null 2> /dev/null ; brew install caskroom/cask/brew-cask 2> /dev/null
$ brew cask install paraview
$ brew cask install julia
$ brew install python
```

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

```
$ sudo apt-get install gcc
$ sudo apt-get install libblas-dev liblapack-dev
$ sudo apt-get install openmpi
$ sudo apt-get install metis
$ sudo apt-get install gmsh
```

```
$ sudo apt-get install paraview
$ sudo apt-get install julia
$ sudo apt-get 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, you can download and install [Julia](https://julialang.org/downloads/) 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 of PDE Applications

Exasim provides many examples that illustrate how to build executable applications 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

```
>> run("pdeapp.m")
```

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 applications can be built by making use of the provided examples. The process of building a new application 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

A finite element mesh of a PDE model is a tessellation of its physical domain by simple geometrical elements of various shapes including triangles, quadrangles, tetrahedra, and hexahedra. A conformal linear 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  $t \in \mathbb{R}^{n_v \times n_e}$  is a two-dimensional integer array storing mesh elements. Exasim provides a Mesh module to generate meshes (namely,  $p$  and  $t$ ) for simple geometries. Practitioners can use any open-source mesh generators such as [CUBIT](#), [CGAL](#), [DistMesh](#), [TetGen](#), [Mmg](#), [Gmsh](#), [MeshLab](#), [SALOME](#) for complex geometries. Because a high-order mesh is needed for the DG discretization of a PDE model, Exasim produces the high-order mesh from a given linear 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 generates VTU files to visualize the numerical solution with [Paraview](#).

## 1.9 Reporting Issues and Suggesting Improvements

Please click [here](#) to report any issues you encounter using Exasim. Please provide a detailed description of the issue as you can. We would like to have your feedback as well. If you have ideas for improvement, please let us know by leaving your comments [here](#).

# 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:

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

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

The vector-valued function  $\mathbf{m} = [m_i(\mathbf{u}, \mathbf{x}, t, \boldsymbol{\mu}), 1 \leq i \leq n_{cu}]$  is called *mass* function, the matrix-valued function  $\mathbf{f} = [f_{ij}(\mathbf{u}, \mathbf{x}, t, \boldsymbol{\mu}), 1 \leq i \leq n_{cu}, 1 \leq j \leq n_d]$  is called *flux* function, The vector-valued function  $\mathbf{s} = [s_i(\mathbf{u}, \mathbf{x}, t, \boldsymbol{\mu}), 1 \leq i \leq n_{cu}]$  is called *source* function. These functions are specified by practitioners 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. In particular, the steady-state version of the Model C can be obtained by setting  $\mathbf{m}$  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:

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

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.  $-\nabla \mathbf{u}$ ) in the *mass*, *flux*, and *source* functions.

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

$$\mathbf{q} + \nabla \mathbf{u} = \mathbf{0}, \quad \text{in } \Omega \times (0, t_f], \quad (3a)$$

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

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

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:

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

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  $\mathbf{u} = \frac{\partial \mathbf{w}}{\partial t}$ ,  $\mathbf{q} = -\nabla \mathbf{w}$ , and rewrite the Model D as follows

$$\frac{\partial \mathbf{w}}{\partial t} - \mathbf{u} = \mathbf{0}, \quad \text{in } \Omega \times (0, t_f], \quad (5a)$$

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \mathbf{u} = \mathbf{0}, \quad \text{in } \Omega \times (0, t_f], \quad (5b)$$

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

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

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

$$\Delta^2 u = f(x), \quad (6)$$

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

$$-\Delta v = f, \quad (7a)$$

$$-\Delta u = v. \quad (7b)$$

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

$$\mathbf{q} + \nabla \mathbf{u} = 0, \quad (8a)$$

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

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  $(\mathbf{u}, \mathbf{q}, \mathbf{w})$ , spatial variables  $\mathbf{x}$ , time variable  $t$ , and physical parameters  $\mu$ , to define governing equations and boundary conditions. We extend these functions with two new variables,  $\mathbf{v}$  and  $\eta$ , 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  $\mathbf{v}$  and external parameters  $\eta$ .

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

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

in a physical domain  $\Omega$  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

```

Since this particular equation has zero mass function, i.e.,  $m = 0$ , we do not need to define the *mass* function in the PDE model file. This also applies to the *source* function, that is, if  $s = 0$ , 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

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

with the initial solution  $u(\mathbf{x}, t = 0) = 0$ , 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

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

with two boundary conditions  $u = 0$  on  $\Gamma_1$  and  $(\mu_1 \mathbf{q} + c u) \cdot \mathbf{n} = -1$  on  $\Gamma_2$  and the initial solution  $u(\mathbf{x}, t = 0) = \mu_3$ . Here  $c = (-x_2, x_1)$  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[1]*u[1];
    f[2] = mu[1]*q[2] - x[2]*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

$$\frac{\partial u}{\partial t} + \nabla \cdot \mu \mathbf{q} = 0, \quad \frac{\partial \mathbf{q}}{\partial t} + \nabla u = 0, \quad \frac{\partial w}{\partial t} - u = 0, \quad (12)$$

with boundary condition  $u = 0$  on  $\partial\Omega$  and initial conditions  $u(x, t = 0) = \sin(\pi x_1) \sin(\pi x_2)$ ,  $\mathbf{q}(x, t = 0) = \mathbf{0}$ , and  $w(x, t = 0) = 0$ . 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};
```

### 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 uses Gmsh to generate meshes from a geometry model file and mesh size parameters specified in the geometry model file. Figure 2 shows a finite element mesh generated by Gmsh for the L-shaped domain defined in [lshape.geo](#).

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, 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 the format of Exasim's text mesh file

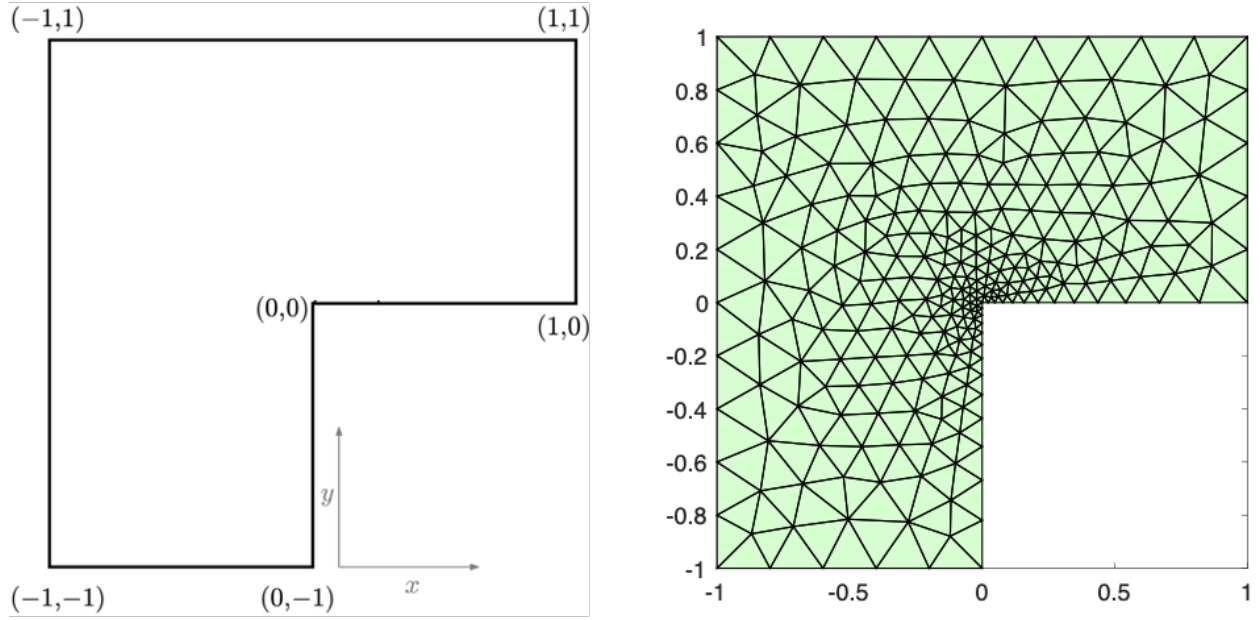


Figure 2: The L-shaped domain and its finite element mesh generated by Gmsh from the geometry model file [lshape.geo](#).

```
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
periodicboundary::Array{Int64,2}; # a 2D list of periodic boundaries
periodicboundaryexpr;        # expressions to determine 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  $\text{boundaryexpr} = [b_1(\mathbf{x}), b_2(\mathbf{x}), \dots, b_{n_{bc}}(\mathbf{x})]$  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  $\overline{\partial\Omega} = \bigcup_{j=1}^{n_{bc}} \overline{\Gamma_j}$ . In particular, if any point  $\mathbf{x} \in \partial\Omega$  satisfies  $b_1(\mathbf{x}) == \text{True}$ , then it belongs to  $\Gamma_1$ . Next, if  $\mathbf{x} \in \partial\Omega$  satisfies  $b_2(\mathbf{x}) == \text{True}$ , then it belongs to  $\Gamma_2$ . So on, until if  $\mathbf{x} \in \partial\Omega$  satisfies  $b_{n_{bc}}(\mathbf{x}) == \text{True}$ , then it belongs to  $\Gamma_{n_{bc}}$ . Note that if  $\mathbf{x} \in \partial\Omega$  satisfies  $b_2(\mathbf{x}) == \text{True}$ ,  $b_4(\mathbf{x}) == \text{True}$ , and  $b_5(\mathbf{x}) == \text{True}$ , then it belongs to  $\Gamma_2$ . 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,  $\text{boundarycondition} = [c_1, c_2, \dots, c_{n_{bc}}]$  is an integer array of  $n_{bc}$  entries which determine a boundary condition for each disjoint boundary. Specifically, disjoint boundary  $\Gamma_j$  accepts boundary condition  $c_j$ . Assume that we divide the domain boundary  $\partial\Omega$  into 4 disjoint boundaries  $\Gamma_1, \dots, \Gamma_4$  and that  $\text{boundarycondition} = [2, 3, 1, 2]$ . In this case,  $\Gamma_1$  accepts boundary condition 2,  $\Gamma_2$  accepts boundary condition 3,  $\Gamma_3$  accepts boundary condition 1, and  $\Gamma_4$  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,  $\text{boundarycondition}$  is tied to the implementation of boundary conditions in *ubou* and *fbou* functions.

Next,  $\text{curvedboundary} = [d_1, d_2, \dots, d_{n_{bc}}]$  is a boolean array of  $n_{bc}$  entries which determine whether a disjoint boundary is curved or straight. Specifically, disjoint boundary  $\Gamma_j$  is curved if  $d_j = 1$  or straight if  $d_j = 0$ . And  $\text{curvedboundaryexpr} = [s_1(\mathbf{x}), s_2(\mathbf{x}), \dots, s_{n_{bc}}(\mathbf{x})]$  is a list of  $n_{bc}$  functions that express the curved equation  $s_j(\mathbf{x}) = 0$  for the disjoint boundary  $\Gamma_j$ . If  $\Gamma_j$  is straight then both  $d_j$  and  $s_j(\mathbf{x})$  are set to 0. Otherwise, if  $\Gamma_j$  is curved then  $d_j$  is set to 1 and the function  $s_j(\mathbf{x})$  must be specified. For example, if  $\Gamma_j$  is a unit circle, then  $s_j(\mathbf{x}) = x_1^2 + x_2^2 - 1$ . Exasim uses  $\text{curvedboundary}$  and  $\text{curvedboundaryexpr}$  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,  $\text{periodicboundary} = [[e_{11}, e_{12}], [e_{21}, e_{22}], \dots, [e_{n_{\text{periodic}}1}, e_{n_{\text{periodic}}2}]]$  is a two-dimensional integer array of  $n_{\text{periodic}} \times 2$  entries which determine a set of  $n_{\text{periodic}}$  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  $\text{periodicboundary} = [[1, 3], [2, 4]]$ , then  $\Gamma_1$  (respectively,  $\Gamma_2$ ) is periodic to  $\Gamma_3$  (respectively,  $\Gamma_4$ ) and any mesh point on  $\Gamma_1$  (respectively,  $\Gamma_2$ ) can be mapped to a mesh

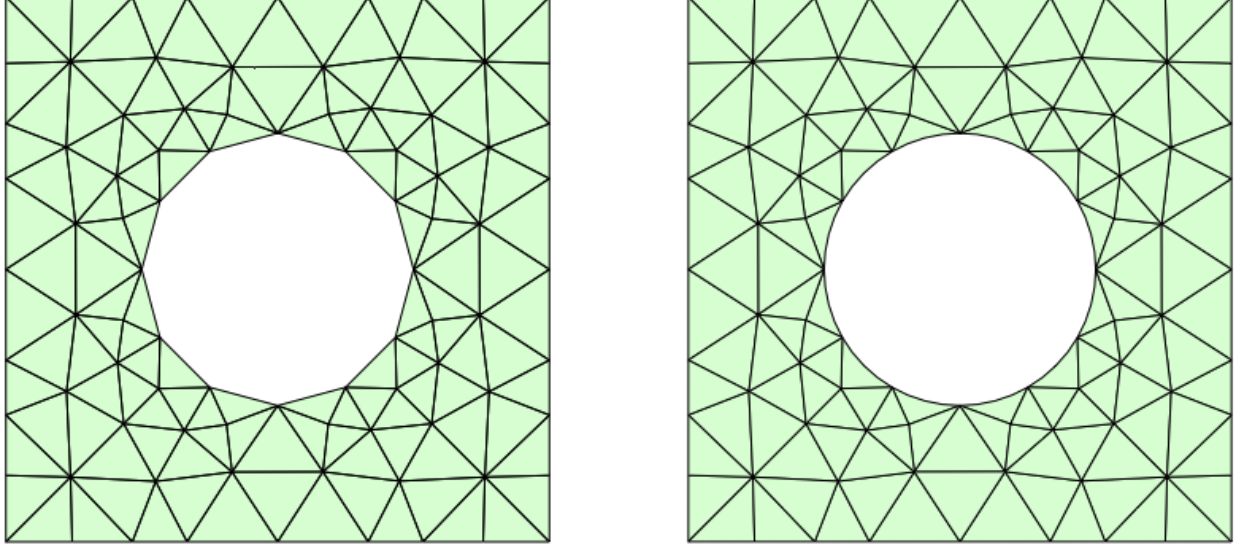


Figure 3: Standard finite element mesh (left) and high-order mesh (right) for a domain bounded by a unit circle and a square.

point on  $\Gamma_3$  (respectively,  $\Gamma_4$ ) by a mapping function, vice versa. And `periodicboundaryexpr = [[ $p_{11}(\mathbf{x})$ ,  $p_{12}(\mathbf{x})$ ], ..., [ $p_{n_{\text{periodic}}1}(\mathbf{x})$ ,  $p_{n_{\text{periodic}}2}(\mathbf{x})$ ]]` is a list of  $n_{\text{periodic}} \times 2$  mapping functions that determine how two disjoint boundaries are periodic each other. For example, in two dimensions, if  $p_{11}(\mathbf{x}) = x_2$  and  $p_{12}(\mathbf{x}) = x_2$  then every mesh point on  $\Gamma_{e_{11}}$  must have its second coordinate equal to the second coordinate of one and only one mesh point on  $\Gamma_{e_{12}}$ . In three dimensions, if  $p_{11}(\mathbf{x}) = [x_2, x_3]$  and  $p_{12}(\mathbf{x}) = [x_2, x_3]$  then  $\Gamma_{e_{11}}$  and  $\Gamma_{e_{12}}$  are periodic with respect to  $(x_2, x_3)$  coordinates.

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 Building PDE Applications

In this section, we describe how `Exasim` build executable applications 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);
versiondir = cdir[1:ii[end]] * "/" * version * "/Julia";
include(versiondir * "/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.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.mpi_procs = 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 application **mpiapp** built by Exasim can be found in the folder `app`. After building **mpiapp**, Exasim runs it 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 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 something like this

As mentioned earlier, Exasim provides many examples that illustrate how to build executable applications 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`.

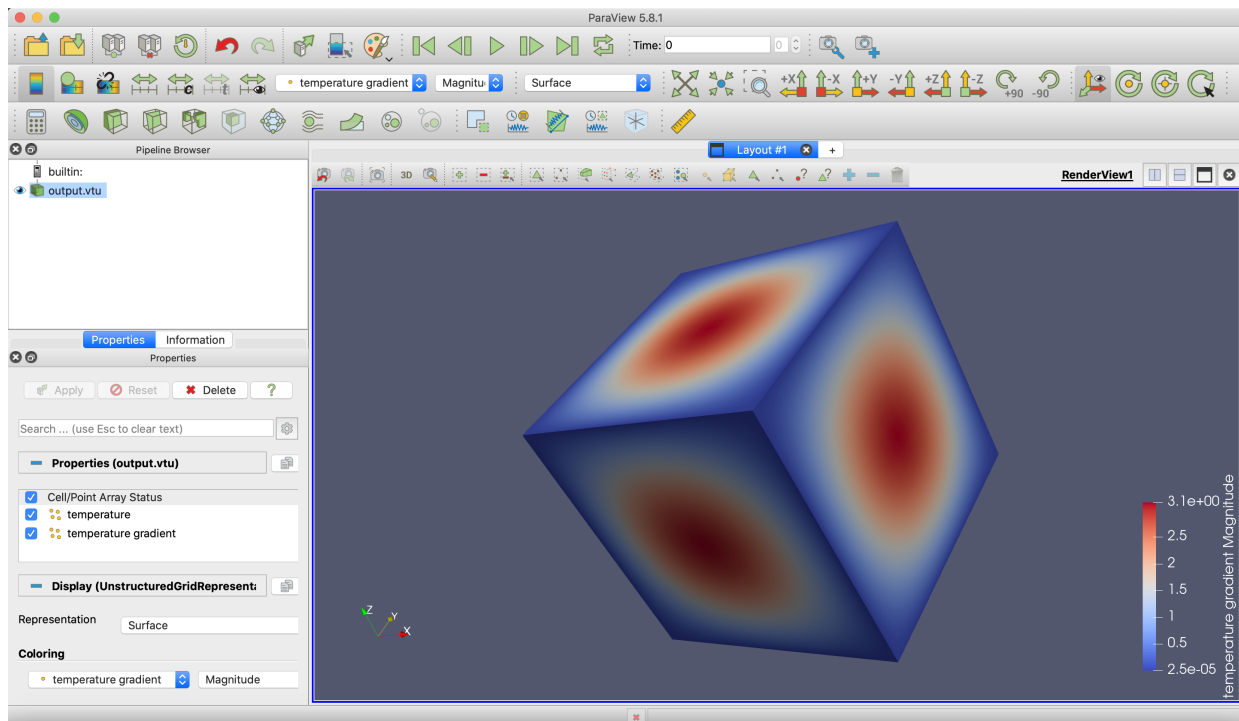


Figure 4: Visualize the numerical solution using Paraview.

## 4.2 Backward Compatibility

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

```
# specify an Exasim version to build application
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.

## 4.3 Exasim Modules

Next, the directories of Exasim modules are added to search path so that they can be found and called to build applications.

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

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

# Exasim modules
```

```
using Preprocessing, Mesh, Gencode, Postprocessing
```

## 4.4 PDE and Mesh Objects

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 `initializepde`. 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 builder 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 builder 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  $\Delta t_1, \Delta t_2, \dots, \Delta t_{n_{\text{steps}}}$ . 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  $\mu$  in the PDE model. Here `pde.tau` is the stabilization parameter  $\tau$  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.mpi_procs` 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.mpi_procs = 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.NLt看ol`), the maximum number of GMRES iterations (`pde.linearsolveriter`), GMRES tolerance (`pde.linearsolvvertol`), 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.gmsh, "filename.geo", nd);
```

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 Exasim

Exasim takes the `pde` object and the `mesh` object 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 build 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 library 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 library 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 libraries are NOT in the system search path
pde.gpuflags = "-Wl,-rpath, /path/to/CUDALibs -L/path/to/CUDALibs -lcudart_
↳lcublas";
```

## 5 Using PDE Applications

### 5.1 Producing C++ 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 build an executable file and store it in app folder
compilerstr = Gencode.compilecode(pde);

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

## 5.2 Deploying The Application

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. 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
```

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 mpirprocs ./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 mpirprocs ./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  $\Omega$ , 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  $\psi_K^k$  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:

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

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

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

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

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

$$\langle \mathbf{w}, \mathbf{v} \rangle_{\partial\mathcal{T}_h} = \sum_{K \in \mathcal{T}_h} \langle \mathbf{w}, \mathbf{v} \rangle_{\partial K} = \sum_{K \in \mathcal{T}_h} \int_{\partial K} \mathbf{w} \cdot \mathbf{v}, \quad (13c)$$



for  $w, v \in \mathcal{V}_h^k$ ,  $W, V \in \mathcal{Q}_h^k$ , 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  $(q_h(t), u_h(t)) \in \mathcal{Q}_h^k \times \mathcal{V}_h^k$  such that

$$(q_h, r)_{\mathcal{T}_h} + (u_h, \nabla \cdot r)_{\mathcal{T}_h} - \langle \hat{u}_h, r \cdot n \rangle_{\partial \mathcal{T}_h} = 0, \quad (14a)$$

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

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

$$(u_h|_{t=0} - u_0, w)_{\mathcal{T}_h} = 0, \quad (14c)$$

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

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

### 6.3 Examples of DG Methods

For the local DG (LDG) method [?], the numerical trace and flux are defined on the interior faces as follows

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

where  $\boldsymbol{\beta}$  is a single-valued vector and  $\boldsymbol{\sigma}$  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

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

where  $(\mathbf{u}_h, \mathbf{q}_h)$  are the numerical solution on the element  $K$ , while  $(\mathbf{u}_h^-, \mathbf{q}_h^-)$  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 builder. Note that Exasim's default implementation sets  $\boldsymbol{\beta} = \mathbf{0}$  and  $\boldsymbol{\sigma} = \tau \mathbf{I}$ , where  $\tau$  is specified by practitioners in the application builder script.

The hybridizable DG (HDG) method [refs] is a DG method with the following form of the numerical trace and flux:

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

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

### 6.4 Implementing Other DG Methods

Exasim allows practitioners to implement other DG methods by explicitly defining the numerical trace and flux in the application builder script. This is done by writing *uhat* function to define  $\widehat{\mathbf{u}}_h$  and *fhat* function to define  $\widehat{\mathbf{f}}_h \cdot \mathbf{n}$ . These functions have the following format

```
# define the numerical trace: uhat
uhat(xdg, udg, udgm, odg, odgm, wdg, wdgm, uhg, nlg, tau, uinf, param, time);
# define the normal component of the numerical flux: fhat dot n
fhat(xdg, udg, udgm, odg, odgm, wdg, wdgm, uhg, nlg, tau, uinf, param, time);
```

Here *udgm* represents  $(\mathbf{u}_h^-, \mathbf{q}_h^-)$ , while *wdgm* represents  $\mathbf{w}_h^-$ . In addition, the following flags need to be set

```
pde.extUhat = 1; # force Exasim to use the numerical trace defined in the
↳application script
pde.extFhat = 1; # force Exasim to use the numerical flux defined in the
↳application script
```

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  $\hat{u}_h$  and *fbou* function to define  $\hat{f}_h \cdot \mathbf{n}$ . The definition of these quantities on the boundary faces depends on the boundary conditions.

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

$$\hat{f}_h \cdot \mathbf{n} = \mathbf{f}(u_h, q_h) \cdot \mathbf{n} + \sigma \cdot (u_h - g_D). \quad (18)$$

This should be implemented in the *fbou* function.

To impose a Neumann boundary condition  $\mathbf{f} \cdot \mathbf{n} = g_N$  on  $\partial\Omega$ , we need to set  $\hat{f}_h \cdot \mathbf{n}$  to the Neumann data  $g_N$  on the boundary. Therefore, the *fbou* function returns  $g_N$ . The normal component of the numerical flux on the boundary is given by

$$\hat{u}_h = u_h + \sigma^{-1}(\mathbf{f}(u_h, q_h) \cdot \mathbf{n} - g_N). \quad (19)$$

This should be implemented in the *ubou* function.

When both Dirichlet and Neumann boundary conditions are imposed on the boundary, we set appropriate components of  $\hat{u}_h$  to the Dirichlet data and determine the same components of  $\hat{f}_h \cdot \mathbf{n}$  by (18). Then we set the other components of  $\hat{f}_h \cdot \mathbf{n}$  to the Neumann data and determine the other components of  $\hat{u}_h$  by (19).

To impose a Robin boundary condition  $\mathbf{f} \cdot \mathbf{n} + \gamma u = g_R$  on  $\partial\Omega$ , we first obtain  $\hat{u}_h$  by solving

$$\mathbf{f}(u_h, q_h) \cdot \mathbf{n} + \sigma \cdot (u_h - \hat{u}_h) + \gamma u_h = g_R.$$

which gives

$$\hat{u}_h = u_h + \sigma^{-1}(\mathbf{f}(u_h, q_h) \cdot \mathbf{n} + \gamma u_h - g_R) \quad (20)$$

Then the normal component of the numerical flux on the boundary is given by

$$\hat{f}_h \cdot \mathbf{n} = \mathbf{f}(u_h, q_h) \cdot \mathbf{n} + \sigma \cdot (u_h - \hat{u}_h) = g_R - \gamma u_h. \quad (21)$$

Equations (20) and (21) should be implemented in *ubou* and *fbou*, respectively.

In summary, a simple approach to implement boundary conditions is carried out in two steps. In the first step, we determine  $\hat{u}_h$  from the boundary conditions. In the second step, we set

$$\hat{f}_h \cdot \mathbf{n} = \mathbf{f}(u_h, q_h) \cdot \mathbf{n} + \sigma \cdot (u_h - \hat{u}_h). \quad (22)$$

Many boundary conditions can be easily implemented by using this approach.