PhD Thesis, University of Washington

A Domain-Hybridized Plasma Model Using Discontinuous Galerkin Finite Elements

Chapter 4: WARPXM Structure

Iman Datta

2021

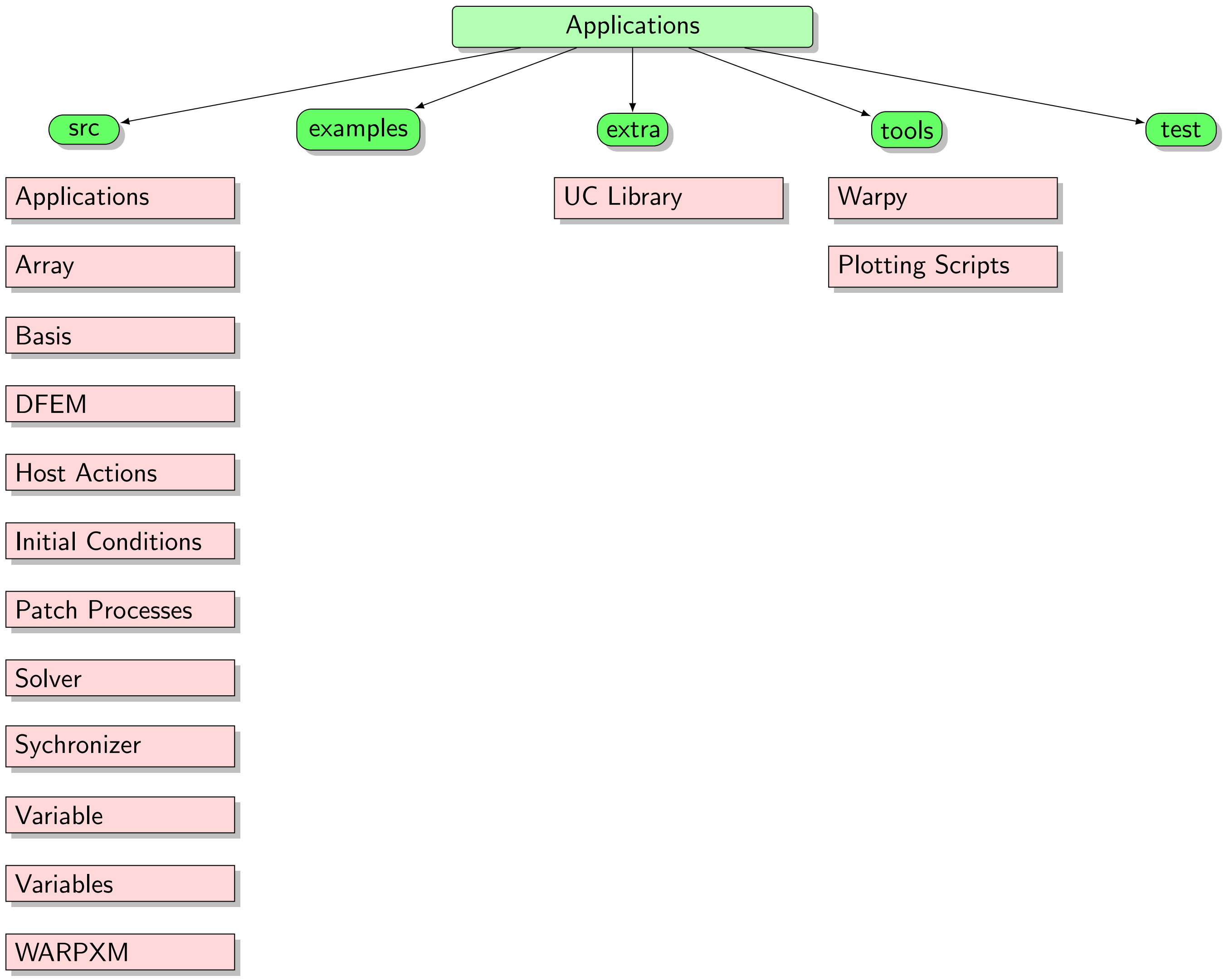
Note: The following chapter has been copied in whole from the original thesis [1]. Links are broken where they refer to other chapters. Please see the original dissertation for a complete version of the chapter.

Contents

The models and numerical methods described in Chapters **??** and **??**, respectively are implemented in the WARPXM (Washington Approximate Riemann Plasma eXtended modeling platform - Many-core version) code [5]. This is an unstructured framework designed to solve the plasma models using the discontinuous Galerkin method. The code was initially built by Sean Miller [6] which extended previous work on the structured WARPX and WARPM codes built by previous students ([9, 10, 2, 4, 8]) to an unstructured framework. This allows for simulation of more complex geometries for which the discontinuous Galerkin method is amenable. The code is also parallelized into subdomains and patches, upon which a problem can be broken up into multiple MPI processes across multiple host machines. In broad terms, the code consists of an unstructured library, a set of host actions, and a set of patch processes. The unstructured library handles incorporation of an external mesh file into usable information as well as handling geometric information associated with the mesh that is required by solvers. Hostactions are calculations over the the entire domain, such as time integration, while patch processes are calculations taking place within the patch level, such as the discontinuous Galerkin solver. A synchronizer is also used to transfer information between neighboring patches when required.

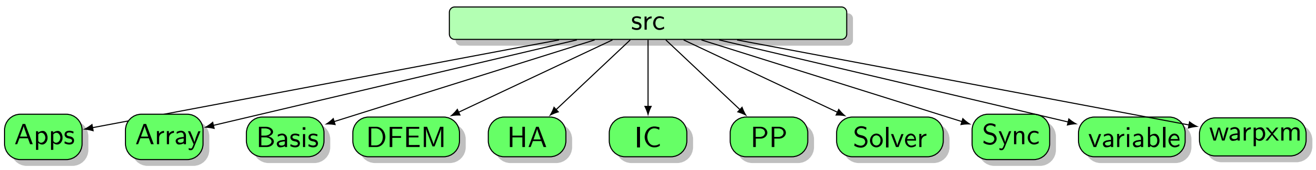
The rest of this chapter gives an overview of the general solver algorithm followed by a high level description of these main sections of the code.

1 High-Level Structure



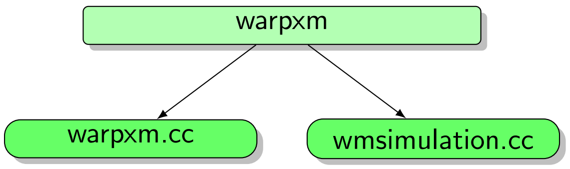
At the highest level, the code has a source directory upon which the meat of the code sits. There is an examples directory with example input files. The “extra” directory holds the Unstructured Converter (UC) library, which translates meshes into information that WARPXM can read and also performs patch and subdomain decomposition. The tools directory holds Warpy, which is a python suite developed to generate and run input files in WARPXM.

2 Source Directory



In the source directory are the integral pieces of the code. Physics applications, solvers, finite element bases, and variable arrays are all developed here.

2.1 warpxm



In this section, the basic workflow of WARPXM is overviewed. The entry point of the code is in the file `src/warpxm/warpxm.cc`. Here all of the required components are setup and run. A general outline of `src/warpxm/warpxm.cc` is as follows

- `main()`
  - `warpxm_init()`
    - initialize MPI
    - initialize petsc
  - `warpxm_main()`
    - initialize simulation
      - read input file
      - create cryptset
      - setup simulation
      - run simulation
  - `warpxm_finalize()`
    - finalize petsc
    - finalize MPI

It can be seen that MPI and petsc are integrated at the highest level. Then the code reads in the input file using a cryptset object before setting up and running the simulation. The cryptset class is the module that translates the input file into the simulation parameters. The simulation is itself an object written in its own file `wmsimulation.cc`, detailed next.

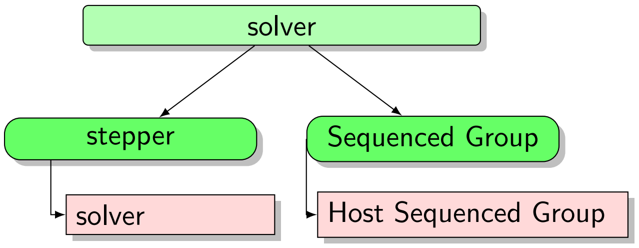
- `setup()`
  - determines a run name and sets up various logging streams for output messaging to the user

- creates a solver based on cryptset parameters
- sets up solver

- `simulate()`
  - runs `solver.solve()`

Overall, this shows that the simulation sets up and runs the solver of the actual problem and relays information between the user and the simulation. An overview of `wmsolver.cc` is described next.

2.2 Solver

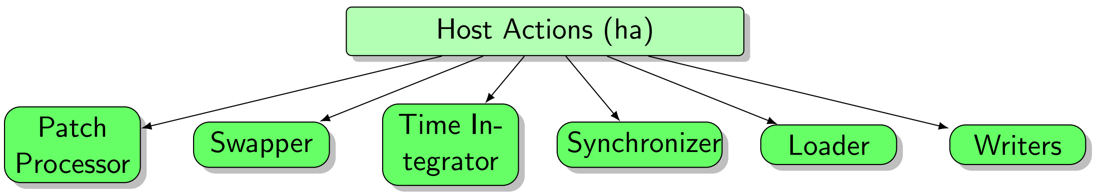


- `setup()`
  - setup `WmDomain` object
    - Sets up mesh object, which uses the UC Domain object to set up unstructured patches, which are then used to generate unstructured geometry objects to develop relevant mesh information
  - Read in variables
    - initialize variables
    - setup variables
  - Initialize host actions
    - for all host actions
      - initialize host action
      - setup host action
  - compile sequence groups
    - `startOnly`
    - `endOnly`
    - `perStep`
- `solve()`
  - `presolve()`
    - initialize hostactions
    - initialize startOnly or restart sequence

  - for each writeout frame to final time
    * `advance()` solution to next frame
      · while time less than time of next frame, run `step_dt()`, which runs `step()` function of the time integrator host action.

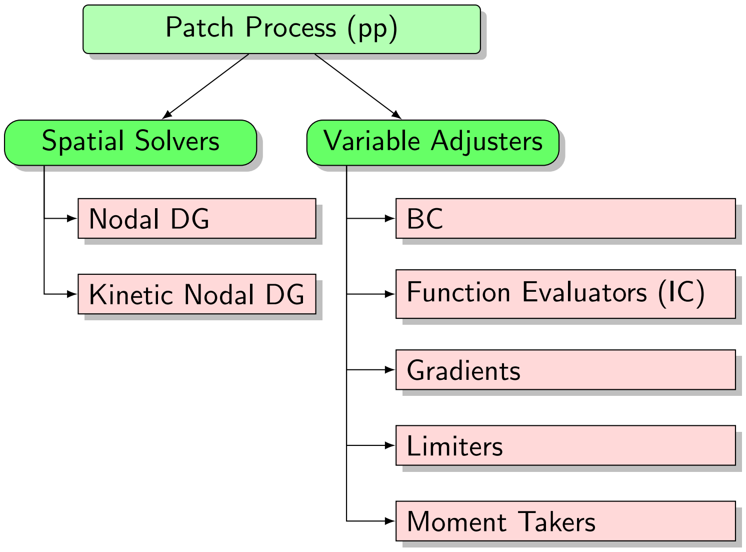
This shows that the solver is responsible for running the simulation as a whole. In the `setup()` function a `WmDomain` object is initialized, which in turn uses a `mesh` object to interface with the unstructured converter library, translating the mesh into usable information for WARPXM. It then reads in variables and sets up host actions, such as the time integrator. Finally it arranges tasks into various groups, such as `startOnly` occurring at the beginning of the simulation, `endOnly` occurring only at the end and `perStep` occurring at each timestep. The solution is advanced in the `solve()` function which runs the `advance()` function to call the time integrator from frame to frame. The time integrator itself is responsible for continuously advancing the solution within these frames. It also handles the spatial solvers and variable adjusters to update the right hand side of the calculation, occurring as patch processes. These components used by the solver (unstructured converter, host actions, and patch processes) are discussed further in following sections.

2.3 Host Actions



Host actions are procedures occurring across the entire domain of the calculation. Important examples include the time integrator (also known as the temporal solver), synchronizer which copies variables at interface elements between patches on different MPI processes, variable loader which loads variables from input files, swapper which swaps like variables between time integrator stages, writers which write variables to output files, and the patchprocessor, which coordinates various patchprocesses within the domain. They should all have a `step()` function delineating procedures to be performed at each time step. At this level various time integration methods could be written. In this work explicit Runge-Kutta methods as described in Sec. **??** are employed.

2.4 Patch Processes



These are processes that occur at the patch level and can be called upon by a host action. In the case of the time integrator host action for example, the spatial solver and variable adjusters are called. The spatial solver effectively calculates the right hand side of the discretized partial differential equation in question. In WARPXM, this is the DG method given in Eq. (**??**). Variable adjusters, as their name suggests, adjusts the variable being solved for in some way before the time integration step occurs. This is the effect of boundary conditions applying a given value to a “ghost boundary node”, gradient solvers (e.g. the application of Eq. (**??**)), and limiters. Initial conditions are calculated in a similar manner, but which are only applied as a startOnly step at the beginning of a simulation. Some more detail of these variables adjusters are given below.

- Boundary Conditions

  Boundary conditions specific to equation sets can be written by the end user. If non-periodic boundary conditions are required, WARPXM creates and extra layer of “ghost” elements around the domain, upon which boundary conditions can be set on nodes just exterior to the domain boundary. The user can call these node locations and apply equation-set dependent boundary conditions. Additionally, “virtual” boundary conditions between subdomains can be applied. This is useful for simulations solving different equation sets on different subdomains of the domain, but need interface conditions to stitch them together on the subdomain interfaces, as is needed by the domain-decomposed hybrid method. In these situations, the “ghost” layer is simply the first layer on the adjacent subdomain.

- Initial Conditions

  This module uses applications developed by the end user to create a set of variable adjusters known as function evaluators on the appropriate subdomains managed by a host action (called variable adjuster runner) that applies these function evaluators only at the beginning of the simulation. The user can write applications just like other physics applications for this purpose, where the initial conditions are set on variables through the `evaluate_function()` method that exists in the application class.

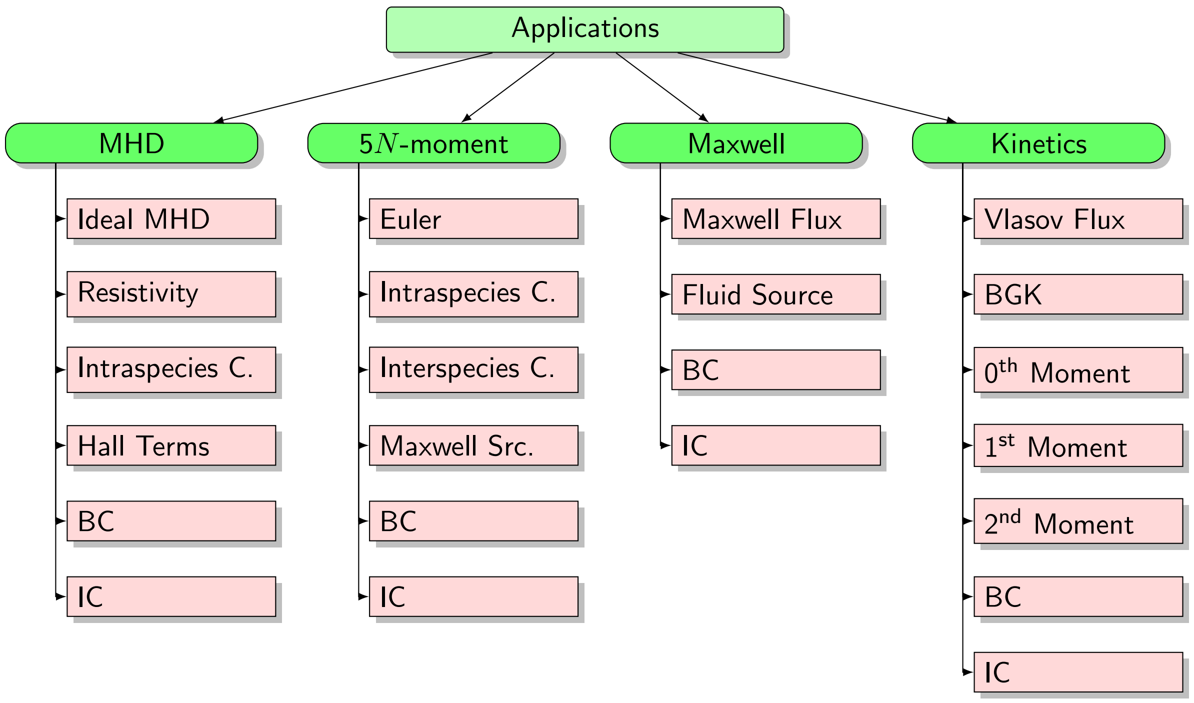
- Gradients

  This module is an application of Eq. (**??**). In a problem involving higher order derivatives, for each timestep one applies this gradient calculation, followed by a relevant boundary condition on them, before applying the spatial solver equation. In this work, the local discontinuous Galerkin (LDG) formulation has been implemented through Eqs. (**??**) and (**??**) as well as the interior penalty (IP) method in Eqs. (**??**), (**??**), and (**??**).

- Limiters

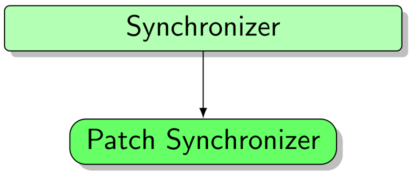
  A few limiter implementations of the slope-moment type limiters mentioned in Sec. **??** have been implemented as variable adjusters. These include limiters described by Moe et al. [7] and Tu et al. [11]. An artificial viscosity limiter has also been implemented, though as an application for the spatial solver instead of a variable adjuster, as it effectively adds a term to the equation set.

2.5 Applications



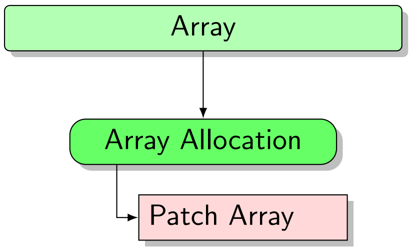
The actual implementation of the physics models as discussed in Chapter **??** happens at this level. Specifically, for the DG solver this happens when applying Eq. (**??**) where $\hat{\mathcal{F}}^\lambda$, $\tilde{\mathcal{F}}^\lambda$, and $\hat{\mathcal{S}}^\lambda$ are determined by the physics model. These are implemented through a set of applications which the end user writes.

2.6 Synchronizer



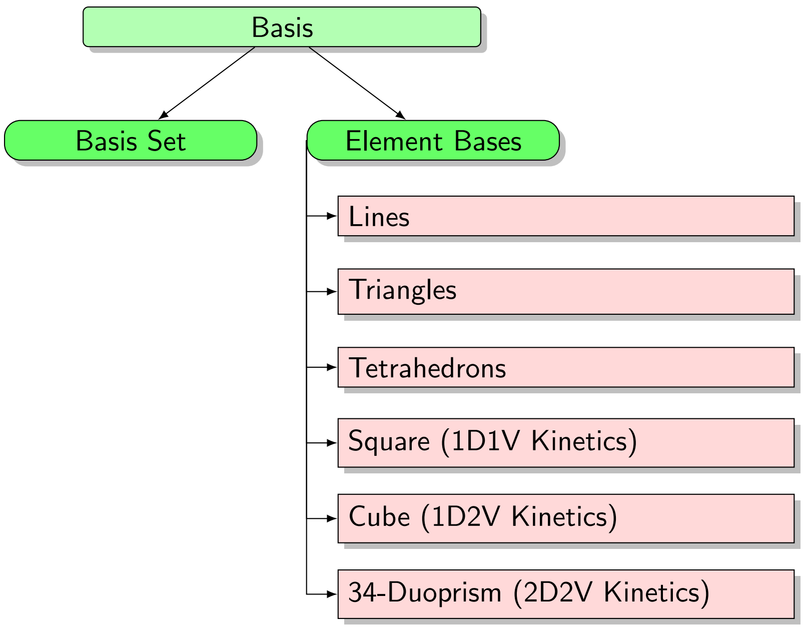
The synchronizer manages the information passing between patches within the domain. Normally, running WARPXM with multiple processes results in the subdomains subdividing into separate patches, one for each MPI process. The synchronizer holds the element numbers between patches that need to be synced at particular points in the simulation. It is held by a host action, such as a time integrator, which uses it to sync data along patch boundaries between timesteps or variable adjuster applications. Asynchronous sends and receives are used, where data is sent between adjacent patches, and then each adjacent patch waits to receive its data from the other patch before moving on in the calculation.

2.7 Array



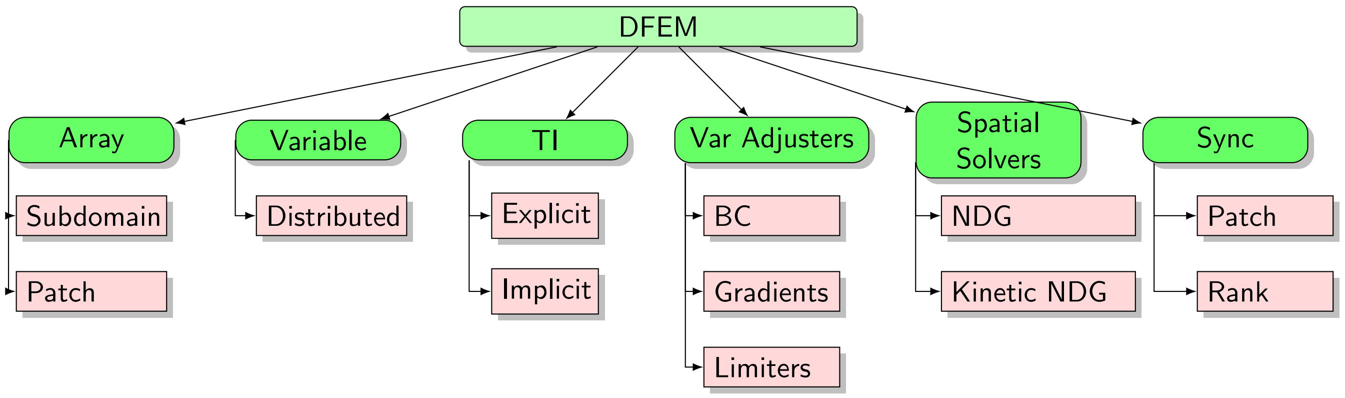
The patch array class here sets up a C++ vector in which data is to be stored. Full implementation is done in the DFEM directory to write this specifically for DG. This array is called by the DG method to operate on variables at node locations. It also gets written to the hdf5 output on writeout steps.

2.8 Basis



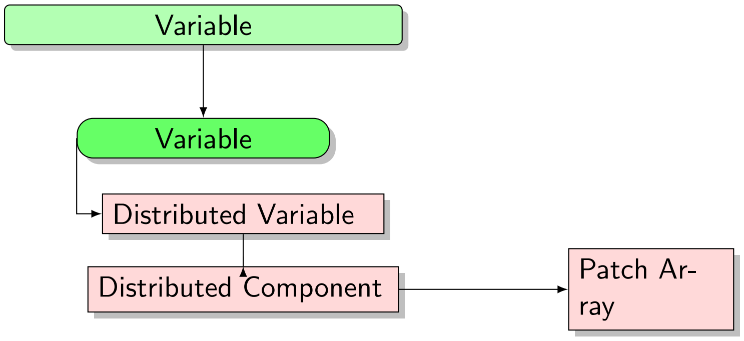
This holds element basis information in the form of text files for various element types with various order. These files are precomputed using mathematica scripts and yield information such as node locations, basis functions, inverse mass matrices, advection matrices, etc. For kinetics, python scripts are used to compute bases using tensor products of lower dimensional elements as described in Sec. **??**. The role of the basis set is to translate these text files into variables/arrays/matrices for use in WARPXM.

2.9 DFEM



The DFEM (Discontinuous Finite Elements) module is where the actual implementations of various host actions, patch processes, variables, arrays, etc. for the DG formulation are written. From the perspective of object-oriented programming, the base (parent) classes of these modules are implemented elsewhere but the children classes specific to DFEM implementation are defined here.

2.10 Variable



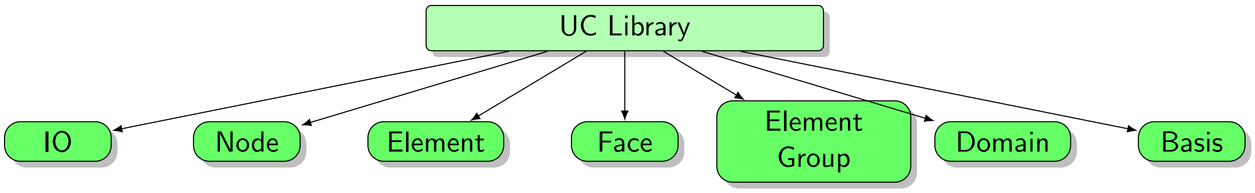
The variable class holds actual patch arrays for various model components and is used when writing to and reading from output files. A distributed variable may be of an "MHD Fluid" which then would have 8 distributed components ($[\rho,\ \rho v_x,\ \rho v_y,\ \rho v_z,\ e,\ B_x,\ B_y,\ B_z]$). Each of these distributed components holds the patch arrays with their values at various nodes in the patch.

3 Unstructured Framework

The unstructured framework in WARPXM can be thought of as having two primary components, consisting of

1. Unstructured Converter (UC) Library
2. Geometric Calculations (Unstructured Geometry Object)

3.1 Unstructured Converter Library



The UC library handles incorporation of the mesh file into geometric information that WARPXM can use and handles subdomain and patch decomposition. It does this through a series of objects that handle different aspects of the domain decomposition and geometric calculations.

At the highest level, the domain object is the interface class responsible for reading, partitioning, accessing and writing unstructured meshes. It holds an element group which is an object that represents various groups of elements, including full domains, subdomains, and patches. Information from the meshfile is converted into these structures. The Metis graph partitioning library [3] is then invoked by the highest-level element group representing the entire domain to perform patch partitioning. Further details of how this library is used is in Sec. 3.2.

Contained by the element groups are nodes, elements, and faces, which are also determined from the meshfile. Nodes are the actual points with coordinates given by the meshfile while elements are collections of nodes connected by a connectivity mapping. They also hold face objects, also determined by connections between nodes. The faces are used to give elements local neighborhoods, mapping them to neighboring elements through these faces. They also have an orientation, which determine how a neighboring face is oriented with respect to it, which determines the opposite node numbers used in numerical flux calculations.

Additionally, a layering calculation is also performed on each element, setting a distance between an element and the edge of the element group. This collects elements within the element groups into separate layers, which can be used by the numerical method to apply certain calculations on specific layers of the element group. External layers are also added, specifying the elements that must be synchronized with those from an adjacent patch, usually on a different MPI process, during various stages in simulations. Finally, periodic boundary conditions are enabled by a determination of conjoining nodes on the domain.

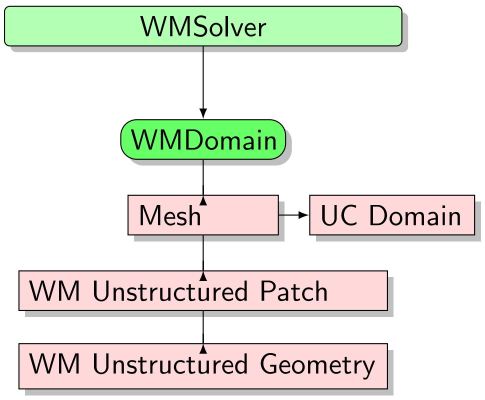
3.2 Metis Partitioning

The goal of the patch partition is to split the domain (or subdomain) up into roughly equivalent sizes, or weightings, between patches, often with each patch being worked on by a separate MPI process. Current usage involves a call to the `METIS_PartGraphRecursive()` function in the Metis library. This function requires the mesh to be organized into a graph, where “vertices” are the elements and the “edges” are the element neighbors. Normally, an input mesh file contains 3 sources of information, the node coordinates, the element connectivities which give nodes that make up each element, and nodesets which specify boundary nodes. The UC library uses this information to calculate element neighborhoods which give the neighboring elements along each element face, as well as the orientation between the faces. The neighborhood information is then called into the `METIS_PartGraphRecursive()` function. The information needed by the function needs to be put in a CSR format where the value array corresponds to locations of an element $\times$ element matrix (though not explicitly given to the function) where we have 1’s (or true’s) where the element in a row has a neighbor at a column. The column array (called `adjncy`) corresponds to the neighbors and the row pointer array (called `xadj`) allows for indexing into the neighbors for given elements. The function then partitions the mesh according to a specified number of partitions (the number of patches the domain is to be decomposed into) and weights for each partition. Current usage exhibits default behavior in which all patches (often one for each compute device available which are CPUs in the present system) are weighted evenly so that each patch should contain roughly the same number of nodes. However, weightings can be given to different processes for load balancing in case certain compute cores are faster than others, for example if GPUs are additionally used. Also, if different models are used in different patches, weights can also be assigned due the difference in computational cost between models.

Metis also provides the functionality to take mesh information directly in the form of connectivities instead of neighborhoods. In this case, the matrix that the CSR arrays holds can be thought of an element $\times$ node matrix where 1’s (or true’s) are where the element in a row has a node at a column. The function `METIS_PartMeshDual()` could be used for this where the column array becomes instead an array of connectivity nodes per element (denoted as `eind`) and the rowpointer array becomes the array to these connectivities per element (denoted as `eptr`). One specifies the number of connecting nodes that specify an edge between elements (1 for lines in 1D, 2 for triangles in 2D, or 3 for tetrahedrons in 3D), and the Metis library determines neighborhoods on its own before then performing the graph partition. The advantage of this method is that it can be used directly with a meshfile that only gives connectivity information and that neighborhoods do not have to be calculated (though it is calculated in WARPXM for use in the DG algorithm to find neighboring elements). However at this time, the graph function `METIS_PartGraphRecursive()` is used to partition domains and subdomains. Future work should consider adjustments of weights according to model usages in hybrid simulations for load balancing between partitions.

An example of usage of the Metis partitioning is shown in Appendix **??**.

3.3 Geometric Calculations



The output of the unstructured converter library is used to develop elemental geometry information, such as Jacobians, element centroid locations, area/volumes, etc. These are specific calculations for various element types, and as element types are added into WARPXM, these calculations must be made. It is created through the mesh which uses the UC Domain object to first develop an unstructured patch object, which then has the necessary information to populate the unstructured geometry object with the relevant information. This is then further expanded upon by the unstructured DG object which adds DG information on to this, such as basis information and other relevant information required by the DG solver. At the time of writing, the unstructured geometry object contains the geometric calculations for 1D line, 2D triangle, and 3D tetrahedron elements based on the information in the mesh file as interpreted by the UC library.

As an example of a Jacobian calculation, consider a 2D isoparametric triangle with three nodes at $(0,0)$, $(1,0)$ and $(1,1)$, respectively, in a 2D isoparametric space (ξ_1, ξ_2). Assuming basis expansion as given in Eq. (**??**), the position in real space for some element λ can be written as

$$\boldsymbol{x}^\lambda = \hat{\boldsymbol{x}}_1^\lambda \psi_1(\boldsymbol{\xi}(\boldsymbol{x})) + \hat{\boldsymbol{x}}_2^\lambda \psi_2(\boldsymbol{\xi}(\boldsymbol{x})) + \hat{\boldsymbol{x}}_3^\lambda \psi_3(\boldsymbol{\xi}(\boldsymbol{x})), \tag{3.1}$$

which has a three-node second-order basis. With a nodal basis this becomes

$$\boldsymbol{x}^\lambda = \boldsymbol{x}_1^\lambda (1 - \xi_1(\boldsymbol{x}) - \xi_2(\boldsymbol{x})) + \boldsymbol{x}_2^\lambda \xi_1(\boldsymbol{x}) + \boldsymbol{x}_3^\lambda \xi_2(\boldsymbol{x}) \tag{3.2}$$

assuming $\boldsymbol{\xi} \in (0,1)$. So for example, the Jacobian terms are

$$J^\lambda = \begin{pmatrix} \frac{\partial x_1^\lambda}{\partial \xi_1} = x_2^\lambda - x_1^\lambda & \frac{\partial x_1^\lambda}{\partial \xi_2} = x_3^\lambda - x_1^\lambda \\ \frac{\partial y_1^\lambda}{\partial \xi_1} = y_2^\lambda - y_1^\lambda & \frac{\partial y_1^\lambda}{\partial \xi_2} = y_3^\lambda - y_1^\lambda \end{pmatrix}. \tag{3.3}$$

Thus, given coordinates from the mesh of an element, λ, the Jacobian can be readily calculated when required.

4 Kinetic Framework

Kinetic calculations in WARPXM are performed by stacking a velocity space on top the physical space that already exists. Functionally, this means the distributed variable array, holding variable values on element nodes are extended into “super elements” that stack all nodes in the velocity space onto the element for kinetic variables. For example, in the simple case of second-order 1D1V, nominally there are two nodes per each line element that is allocated into the distributed array. However, for phase space variables, nodes are added according to the number of nodes in the corresponding phase space element and the extent of velocity space. In second-order 1D1V, the phase space element is square. So if velocity space extends 10 elements in v_x, then the phase space super element will consist of $4 \times 10 = 40$ nodes per super element. The kinetic DG implementation then must correctly calculate node numbers when in phase space. In this way, any arbitrary geometry in physical space can be used.

5 Domain-Decomposed Hybrid Method

The domain-decomposed hybrid method as described in Sec. **??** can be thought of as a boundary condition between models at a subdomain interface. The direct variable translation method as described in Sec. **??** directly translates variables at these subdomain interfaces to the appropriate model before calculating consistent numerical fluxes. This is performed through virtual boundary conditions, which set values of variables on internal edges of subdomains that are consistent with adjacent subdomains’ models. The numerical flux can then be calculated at each subdomain interface using the values consistent with each model. For the composite distribution function method as described in Sec. **??**, the boundary condition calculations can be sidestepped and the numerical fluxes can be directly calculated, such as by summation of the sided moments of the distribution functions on either side of a subdomain interface that construct the composite distribution function.

The domain-decomposed hybrid method allows for reduction in required memory and simulation time due to the fact that certain subdomains can use reduced models with smaller sets of distributed variables and less computationally-intensive calculations. Future work could make the domain decomposition dynamic or adaptive, in which the mesh is remapped at periodic intervals using the Metis partitioning tools as described in Sec. 3.2. Such dynamic remapping of subdomains can be useful in simulations in which the physical accuracy of models in different regions change over time, which can be determined using metrics such as χ in Eq. (**??**).

References

[1] Iman Datta. *A Domain-Hybridized Plasma Model Using Discontinuous Galerkin Finite Elements.* PhD thesis, University of Washington, 2021.

[2] Ammar H. Hakim. *High Resolution Wave Propagation Schemes for Two-Fluid Plasma Simulations.* PhD thesis, University of Washington, Seattle, WA, 2006.

[3] George Karypis and Vipin Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359—392, 1999.

[4] John Loverich. *A Discontinuous Galerkin Method for the Two-Fluid Plasma System and Its Application to the Z-Pinch.* PhD thesis, University of Washington, Seattle, WA, 2005.

[5] John Loverich, Ammar Hakim, and Uri Shumlak. A Discontinuous Galerkin Method for Ideal Two-Fluid Plasma Equations. *Communications in Computational Physics*, 9(2):240–268, 2011.

[6] Sean Miller. *Modeling collisional processes in plasmas using discontinuous numerical methods.* PhD thesis, University of Washington, Seattle, WA, 2016.

[7] Scott Moe, James Rossmanith, and David Seal. A Simple and Effective High-Order Shock-Capturing Limiter for Discontinuous Galerkin Methods. *arXiv*, arXiv:1507.03024, 07 2015.

[8] Noah Reddell. *A Kinetic Vlasov Model for Plasma Simulation Using Discontinuous Galerkin Method on Many-Core Architectures.* PhD thesis, University of Washington, Seattle, WA, 2016.

[9] Eder Marinho Sousa. *A Blended Finite Element Method for Multi-Fluid Plasma Modeling.* PhD thesis, University of Washington, Seattle, WA, 2014.

[10] Bhuvana Srinivasan. *Numerical Methods for 3-dimensional Magnetic Confinement Configurations using Two-Fluid Plasma Equations.* PhD thesis, University of Washington, Seattle, WA, 2010.

[11] Shuangzhang Tu and Shahrouz Aliabadi. A Slope Limiting Procedure in Discontinuous Galerkin Finite Element Method for Gasdynamics Applications. *International Journal of Numerical Analysis and Modeling*, 2(2):163–178, 2005.