Tenth International Conference on
Computational Fluid Dynamics (ICCFD10)
Barcelona, Spain, July 9-13, 2018

ICCFD10-2018-145-044

Unified Geometries for Dynamic HPC Modeling

Graham J Orr, Xplicit Computing Inc
info@xplicitcomputing.com

Abstract

A common programming interface for all types of computational geometries is introduced, allowing type specialization, starting with distinct HPC optimizations separating *structured* and *unstructured* geometry branches. For structured types, element data is generated procedurally, while for unstructured types, data is both explicit and declarative. Further specialization down the tree defines G*rid*, *Tree*, *Mesh*, and *Network* types, each with their own characteristic uses, optimizations, and further sub-classifications. Unified access patterns can be implemented in heterogeneous (e.g. CPU, GPU) compute environments, providing a foundation to develop robust multi-domain spatial algorithms that utilize any geometry types.

Keywords: Spatial, Geometry, Grid, Mesh, Heterogeneous, Computing, Polymorphism

1 Introduction

Presented with a spectrum of real-world engineering challenges, it is evident no one geometry type is sufficient to cover all realms of numerical analysis. For a specific application, there always appears to be an appropriate grid, mesh, etc. A taxonomy helps catalog common underlying attributes and functions across known functionalities, resulting in a class hierarchy (related by inheritance) and common interface for all geometry types. This allows numerical algorithms (such as those used in physical models) to call through a common set of functions and dispatch to the appropriate optimization for a given type. Such characteristics are essential to modern HPC, scientific computing, and machine design.

As a first example of geometry specialization, consider a function that returns the position of a node in an optimized geometry: Unstructured meshes should read previously-written memory, whereas structured grids should calculate values at runtime using a constant-time indexed lookup (mitigating cache-misses and page-faults). Varied underlying machinery access geometries using the same function signature, redirecting runtime behavior while enforcing unified access patterns.

Benefits should be apparent to teams performing multi-disciplinary analyses. One team may be solving a structured fluid grid while another team may be optimizing an unstructured mechanical mesh. A common geometric language enables:

- effective and accurate communication across domains
- optimization of computing resources
- reuse and expansion of numerical algorithm assets
- project success for complex system work-flows

This paper provides technical foundations of a software geometry kernel enabling these functionalities within discrete computational domains, yielding a *50-fold* span in memory and runtime performance as demonstrated in *Xplicit Computing Inc*'s *XCompute*™ products and prototypes from 2012 through 2018. The breadth of numerical computing and its diverse future is presented anecdotally to support lacuna and allude to characteristic applications.

2 Class Inheritance – Hierarchy

An inheritance model's utility is to create a more modular and intuitive program – so that member attributes and functions can be reused or redefined as desired. One corollary is that developers and users are exposed to convenient standard structures and access patterns, further facilitating design cohesion, efficiency, and testability. Focus and clarity further enhances high-level optimizations and innovations when developing new algorithms.

Necessity for inheritance may not be immediately apparent, but stark contrasts exist between grid and mesh types: meshes have memory buffers for element positions and topology, while grids can calculate positions and elements procedurally. Yet, all must store scalar field data (but not necessarily topology) and conform to strict interface patterns established in the geometry base. Domains form a larger spatial family with profound commonalities and specializations:

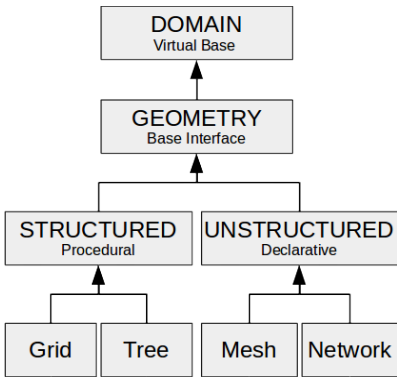


Illustration 1: Geometry Type Taxonomy. It is posited such inheritance and resulting polymorphism can support a standard common programming interface and optimizations for any type.

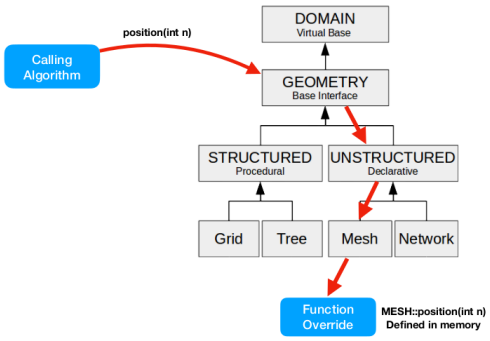


Illustration 2: Example mesh position function override specialization, redirecting generic call to return values statically stored in memory. This may minimize processing overhead, but also increases IO traffic.

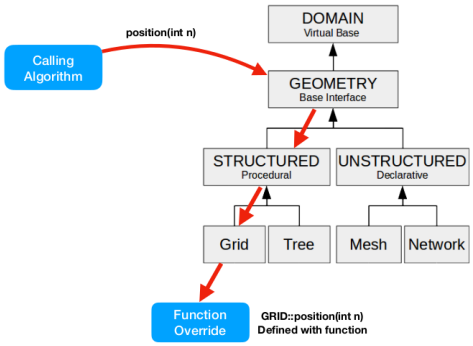


Illustration 3: Example grid position function override specialization, redirecting generic call to return values dynamically generated by local function. This may increase processing overhead, but minimize IO traffic.

2.1 Runtime Polymorphism – Modularity

C++ is statically-typed so class structures and functions must be defined at compile-time, but are not bound until runtime when a call through a virtual function table (*vtable*) invokes the appropriate overridden function in the derived class associated with the object – thus, redirecting behavior. Virtual functions imply those overridden with a specialized derived version containing identical function signature. This provides flexibility to general purpose programming that can make use of generic function dispatch, which can be useful when developing generic calling algorithms. Underlying common functions can be found in *Appendix A*.

Domain is a concrete virtual base class that can only call functions defined in its column on the table, providing functionalities for generic functions like: checking whether one domain intersects another or ray cast through space – yielding key optimizations (and internal faculties) per type. Functions like position containment and delta between extrema are less specialized but find wide application (spanning server and client applications).

Geometry inherits its attributes and functions from Domain, though itself is an abstract base containing both pure virtual and function overrides. Geometric topology, scalar data, and construction introduces new utilities such as element access, region creation, and value sampling. Further specialization in procedural and declarative branches uncover major differences in function management: Procedural leaves almost all functions to be defined later in derived types. Declarative populates its table with overrides that are utilized by derived types. Build functions for Grid, Tree, Network, and Mesh types have unique and highly-specialized implementations.

Polymorphism provides another crucial technique to define and re-define *OpenCL* to mimic *C++* function dispatching (on co-processor devices such as GPUs). As a function or a member override, string literals containing source code fragments are compiled into a larger CL program. Similarly, file paths can be changed to include external headers that geometry *CL* code may need in its *Just-In-Time* (JIT) compilation prior to kernel launch. Unlike *C++*, this redefinition can be done during runtime (and run on any device) – a breakthrough for dynamic heterogeneous programming.

2.2 Application Examples

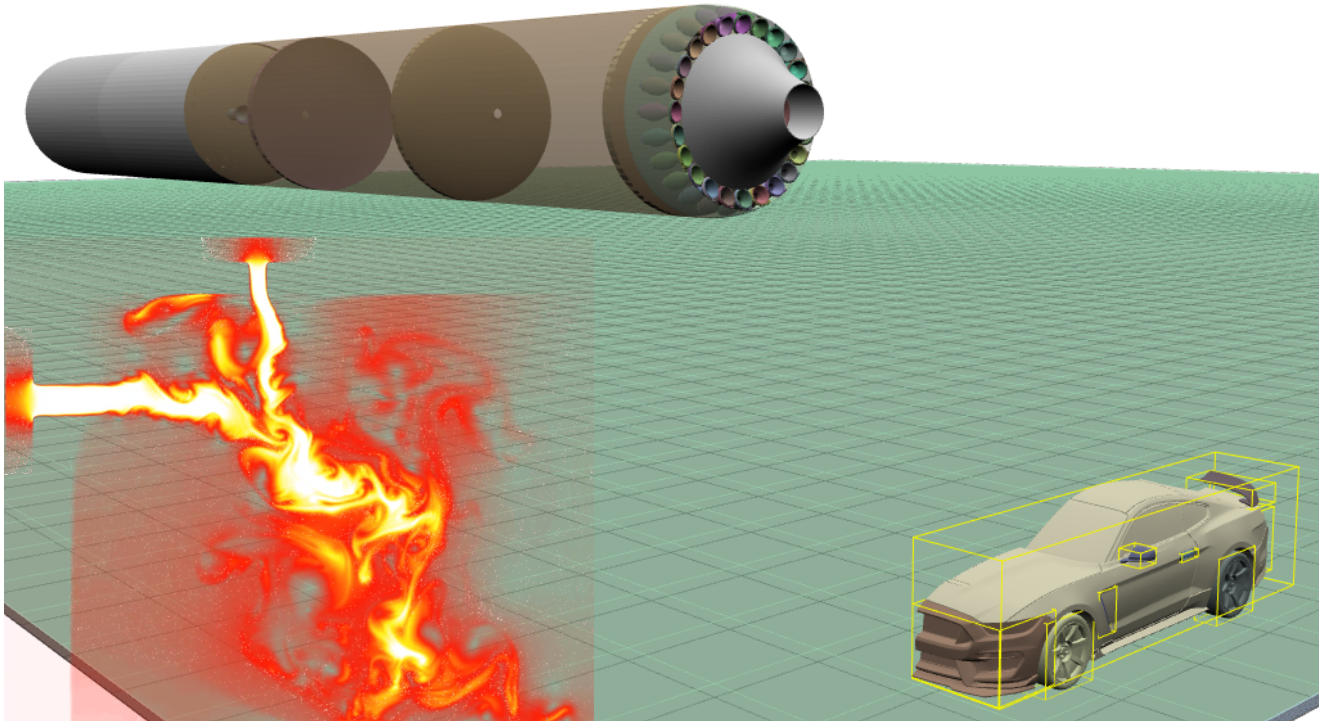


Illustration 4: Large computable geometries on consumer hardware, with structured & unstructured types in global space spanning wide length-scales. Half of Ford Mustang Shelby GT350R and aerospike rocket "Armstrong" on a structured brick road. Many of the domains used in automotive and aerospace analysis can leverage procedural geometries, while complex manifolds such as the Mustang body remain declared in memory. Procedural components and routing can be generated with specialized structured algorithms and others can be designed in CAD or other program. Adaptive resolution and numerical efficiency across all types enable both larger and faster simulations on equivalent resources. Regions are colored individually while regions of car are bound in yellow boxes.

Modern analyses require systems interact over wide space-time scales. Single global discretization approach is an extremely inefficient and non-scalable approach to multi-physics. Therefore, abstractions must provide a way to discretize and solve each problem individually and integrate as part of the whole. All permutations should be supported robustly with unified data coupling algorithms.

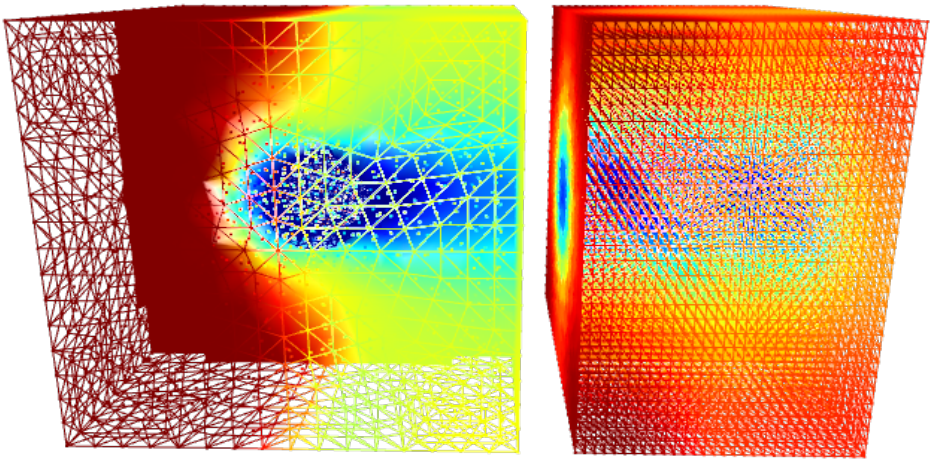


Illustration 5: Inter-domain mutual coupling between disparate geometry types. Unstructured mesh on left, structured grid on right, running two independent numerical methods coupled in time and space. In this test, a cylinder is placed inside a gas flow, with wake clearly visible through domains. Inter-operable algorithms enable higher-level coupling and global accuracy.

3. Geometry Base Interface

Currently, most algorithms developed by engineers are one-off scripts with little structure or standardization. This inhibits numerical R&D efforts from being reused in later projects as there is no universal way to adapt old code into new contexts. Through a universal interface, object-oriented programing encapsulates scientific computing concepts. Complex, integrated environments require more robust machine protocols in varying scenarios, as homogeneous approaches do not scale favorably against non-linear complexity. Humans and computers need elegant and inclusive spatial abstractions to yield cross-cutting functionalities, not more siloed and inefficient 'codes' without a lineage or future.

Algorithms encapsulate processes and can be connected to define work-flow across an organization, spanning numerical transport to state processes to high-level systems optimization. Ideally, generic algorithms are developed only once, and subsequently applied and adapted as an asset. Modularity facilitates code reuse and inspection; all the while establishing stylistic cues that further support heterogeneous computing machinery. To improve ease of use but not limit numerical capability, a finite set of standard functions are defined in conformance with a Turing-complete language (C++14) to support wide programming functionalities. Each geometry type may have optimal differentiation and integration strategies creating overlapping compatibility families to facilitate inter-domain processes and fall-back (or redundant) algorithms.

For instance, many numerical methods require evaluation of a gradient, yet there are many ways to achieve this depending on the application. If a continuum is assumed in the domain, then sampling can apply Reynolds transport and divergence theorem to yield a robust solution. In morphed grids and unstructured meshes, generic operators such as *Green-Gauss Gradient* can be the backbone of ubiquitous finite difference and viscid layer computations. Yet, where there is no topological divergence (orthonormal) this generality comes at some unnecessary numerical cost. Assumptions per application dictate the appropriateness of an algorithm (verb) bound to a geometry or other argument (noun).

Example: Generic algorithm compatible with grids, meshes, and trees – *Green-Gauss Gradient* (16 lines)

Gradients of scalar field $\phi(\vec{x})$ can be sampled over *K* duals with interface *dA* and control volume *V:*

$$\nabla\phi(\vec{x}) = \frac{1}{V(\vec{x})}\oint_A \phi_{dA}\, d\vec{A} = \frac{1}{V(\vec{x})}\sum_{k=0}^{K}\left[(1-\alpha)\phi(\vec{x}) + \alpha\phi(\vec{x}_k)\right]\Delta\vec{A}_k + \epsilon\left(O(\vec{x})^2\right) \tag{1}$$

Where $\alpha = 0.5$ for median sampling. With our API, the equivalent C++ computes gradients for any scalar parameter:

```
void GreenGaussGradient(DATA& data, GEOMETRY& geometry, PropertyKey val_pk){
    VECTOR& values = data(val_pk);                              // get data vector for scalar property
    auto grad_pk = PropertyKey(val_pk, Gradient);               // create a new pk by appending gradient modifier
    VECTOR& gradient = data(grad_pk);                           // get data vector for gradient property
    #pragma omp parallel for
    for (int n = 0; n < geometry.nNode(); n++){                 // iterate through all nodes (parallel)
      double volume = geometry.volume(n);                             // get volume at target dual node
      if (approxEqual(volume,0.))                                     // skip if volume is nearly zero
              continue;
      gradient.row(n) = VECTOR::Zero(1, geometry.D);            // before first dual zero out gradient
      auto K = geometry.nNeighbor(n);                           // get number of neighbors for node n
      for (int k = 0; k < K; k++){                                    // iterate through interior neighbors of n
           FACE face = geometry.getInteriorFace(n, k);                      // get the interior dual node face n-k
           if(face.empty())                                             // skip fragment if face is empty
               continue;
           int& ns = face.indices[1];                                   // flux source node index (opposing)
           double interfaceValue = .5 * (values(ns) + values(n));       // compute interface value as average
           for (auto d=0; d<geometry.D; d++)                            // for each spatial dimension
               gradient(n,d) += interfaceValue*face.area[d]/volume;     // sum gradient using Green's theorem
      }
    }
}
```

GEOMETRY is an abstract base class that derives from DOMAIN serving as a common programming interface for all types. An algorithm can call through a generic GEOMETRY pointer (address) and its functions will redirect to variations available in the *vtable*. Derived types can also call static base versions of function overrides. Generic algorithms operate on any geometry type through a unified base, greatly-improving code reuse and algorithm modularity. At its core, a name, heterogeneous DATA record and revision:

```
std::string name;                          // name of the geometry
DATA data;                                 // scalar attributes, such as Position or SDF vectors
REVISION revision;                         // major and minor change counter
STRUCTURED* background{nullptr};           // ptr to background geometry (optional, see § 3.3)
```

REGIONS may be defined by the user or algorithms zero through *D* dimensions for manifolds $\vec{x} \in \mathbb{R}^D$, including:

```
REGIONS<GROUP> groups;                     // 0d node groups of nodes
REGIONS<LOOP> loops;                       // 1d loops of edges
REGIONS<SURFACE> surfaces;                 // 2d surfaces of faces
REGIONS<VOLUME> volumes;                   // 3d volumes of cells
```

GEOMETRY is never instantiated itself, but its base functions (in *Appendix A)* are called from algorithms and behavior is redirected to take advantage of characteristic optimizations in derived types. Unified interface promotes high-level algorithm modularity and unlock optimizations that provide a more than an order-magnitude benefit to analysis resolution and/or execution speed. Assumptions about the topology can reduce traffic and memory access – where possible functions that may have otherwise accessed memory calculate values “on the fly” given a set of encoded inputs. Such procedural approach yields major benefits but limits flexibility that is apparent in explicit declarative types. Still, remarkable spectrum of unique adaptability and suitability exist for each, suggesting diverse future adaptations and applications on the horizon.

3.1 Structured Procedural Branch

STRUCTURED geometry is an abstract base with the defining attribute that topology is sufficiently-ordered to enable return of elements through a generative procedure. That is, given an integer-based element location *ijk*, we can call a function to return position and neighbors. Topology information is not explicitly stored, as elements and attributes are computed deterministically from a set of analytic functions rather than member attributes defined in memory. Beyond base geometry interface, few attribute or functional commonalities can be described in this virtual class, reflected by sparsity in its *vTable*. Efficiency in memory and computation is a primary advantage, enabling increase in capacity by more than an order of magnitude due to alleviated interconnect traffic and processor cache faults. Benefits are realized in numerical methods that present a more physical interpretation, and less apparent in schemes that reconstruct associativity as required by the processing format (incurring larger solver memory penalties). While some geometries may have optimal methods, the aim is to take advantage of efficiency where possible, and rely on compatibility when required.

3.1.1 Structured Regular Grids

Grids provide the simplest spatial scaffolding to approximate PDEs. *Finite Difference Method* (FDM) and *Lattice-Boltzmann Method* (LBM) demonstrate continued demand as applied to *Large Eddy Simulation* (LES) and *Direct Numerical Simulation* (DNS) - the evolution of the fluid field is assumed statistically isotropic (a priori), yet resolved in time and space with *Courant numbers* ideally on order of unity to limit numerical diffusion and phase error.

GRID inherits from STRUCTURED geometry, but adds the following member attributes:

```
glm::ivec size;        // node count in each dimension
int basis;             // coordinate transform (Cartesian, cylindrical, spherical, …)
```

Grid indexing samples field regularly through domain with integer location $\vec{k}$

$$\phi(\vec{x}(\vec{k})) \quad \text{for} \quad \vec{k}=(i,j,k,l)\in\mathbb{Z}^D \tag{2}$$

Spatial vector entries are ordered left to right, map similarly to non-dimensional, Cartesian, and other spaces:

$$\vec{k}(i,j,k,l)\sim\vec{\xi}(\xi,\eta,\zeta,\tau)\sim\vec{x}(x,y,z,t)\sim\vec{\theta}(r,\theta,z,t)\sim\ldots \tag{3}$$

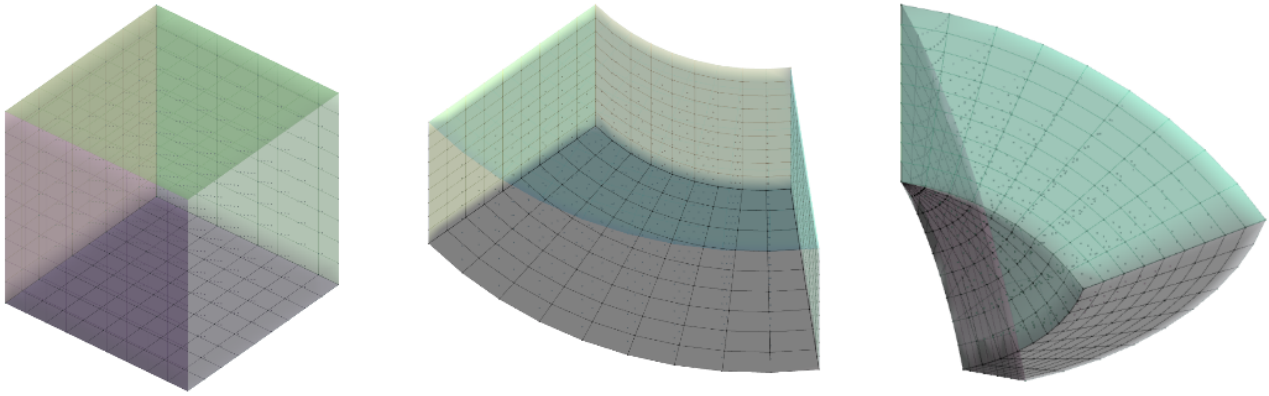


Illustration 6: Simple orthogonal grids of size IJK[10,10,10], from left to right: Cartesian, cylindrical, spherical isometries with min[1,0,0] & max[2,1,1]. This particular example demonstrates a regional degeneracy on the spherical surface k+.

Orthogonal basis transformations project grids to other coordinate bases, including non-dimensional, cylindrical, and spherical isometries. Associativity remains untouched (with exception to degenerate boundaries) although accessors for position, face normal, and volume functions are transformed for local scaling and skewing. Tensor Q_n is locally-varying (per-node, varying as a function of position):

$$\vec{x}\,'=Q_n\vec{x} \quad \text{for} \quad Q_n^{-1}=Q_n^T \qquad \forall n\in\vec{x} \tag{4}$$

Regular sampling presents an opportunity to incorporate digital signal processing techniques, well-suited to problems where spectral resolution and characterization are important, such as in LES [15]. The goal is to preserve spectral integrity of conserved scalar $\phi(\vec{x})$ for each degree of freedom (energy, momentum, mass, charge, species...) using strong form of substantive derivative along velocity transport $\vec{\psi}=\phi\vec{u}$ on orthogonal spatial basis:

$$\dot{q}\overset{\text{def}}{=}\frac{D\phi}{Dt}=\sum_{d=0}^{D}\frac{\partial\psi_d}{\partial x_d} \quad \text{or in terms of scalar rate} \quad \dot{\phi}=\dot{q}-\nabla\cdot\vec{\psi} \tag{5}$$

$\dot{q}$ is the source-generation term (zero for locally conservative or free), with non-conserved quantities (pressure, temperature, velocity…) derived in state equations. *Taylor Series* project continuous function derivatives about $\vec{x}+d\vec{x}$:

$$\phi(\vec{x}+d\vec{x})=\sum_{k=0}^{K}\frac{(d\vec{x})^k}{k!}\cdot\nabla^k\phi(\vec{x})+\epsilon(O(\vec{x})^K) \tag{6}$$

FDM achieves *K*-order accuracy by enlarging differencing stencil (expanding and factoring Eq 6) or tuning coefficients per numerical characteristics. Balanced differencing procedures (e.g. mirrored permutations outlined by *MacCormack* in 1971 [33]) limit numerical bias and improve robustness. Numeric and geometric singularities require dynamic low-pass filtering and variable timing to permit Courant numbers near or above unity. Feedback control *f* adjusts timing using temporal residual estimates from difference in last two temporal integration stages targeting residual ϵ_o :

$$dC_o(\vec{x})=C_o(\vec{x})\cdot f\,(1-\epsilon(\vec{x})/\epsilon_o)\,dt \tag{7}$$

Frequency response characterization is efficient and accurate within structured domains as *Z-transform* convert a regularly-sampled discrete filter $h(k)$ from index *k-space* to complex frequency *z-space* $H(z)=Z(h(\vec{k}))$, in 1d [25]:

$$H(z)=\sum_{k=-K}^{K} h(k)\,z^{-k} \tag{8}$$

Transfer function $H(z)$ can be directly applied to spectral field $\Phi(z)=Z(\phi(\vec{k}))$:

$$\Phi'(z)=H(z)\Phi(z) \tag{9}$$

Finite Impulse Response (FIR) filter $h(\vec{k})$ of size *K* and rank *D is* convolved with $\phi(\vec{k})$ to modify signal:

$$\phi'(\vec{k})=h(\vec{k})*\phi(\vec{k})+\epsilon(O(\vec{k})^K) \tag{10}$$

Inverse $H^{-1}(z)$ is analytically determined from $H(z)$, and equivalent filter $h^{-1}(k)=Z^{-1}(H^{-1}(z))$ can deconvolve processed signals to estimate original, as used in *Smagorinski SGS* turbulence models [15]:

$$\phi(\vec{k})=h(\vec{k})^{-1}*\phi'(\vec{k})+\epsilon(O(\vec{k})^K) \tag{11}$$

Gradients can be obtained by convolving differencing filters $\vec{g}(\vec{k})$ and scalar field $\phi(\vec{k})$ in each dimension:

$$\nabla\phi(\vec{k})=\vec{g}(\vec{k})*\phi(\vec{k})+\epsilon(O(\vec{k})^K) \tag{12}$$

Finite Difference Time Domain (FDTD) is a similar technique used for time-varying elliptic/parabolic electro-magnetics, usually *Maxwell's Equations* as function of curl (e.g. vorticity potential methods). FDM and FDTD are bound by analogous timing and stability with respective mediating local wave-speed $a(\vec{x})$ for sound and light. Index of refraction λ dictates local light speed $c(\vec{x})=c_\infty/\lambda(\vec{x})$, resulting in high electric field gradients near dielectric interfaces.

Tensor transformations extend grids to non-orthogonal curvilinear bases such as axisymmetric, *C*, *H*, and *O* topologies, ideal for analyses along analytic or piece-wise curves. Differencing schemes must correct for varying $d\vec{x}/d\vec{k}$ or revert to a *Green-Gauss* or *Least Squares* Gradient. Topology remains unchanged, requiring semi-smooth matrix Q_n on $\vec{x}$:

$$Q_n\vec{x}<(Q_n-dQ_n)(\vec{x}+d\vec{x}) \quad \text{equivalent to local change constraint:} \quad dQ_n(\vec{x}+d\vec{x})<Q_n(d\vec{x}) \tag{13}$$

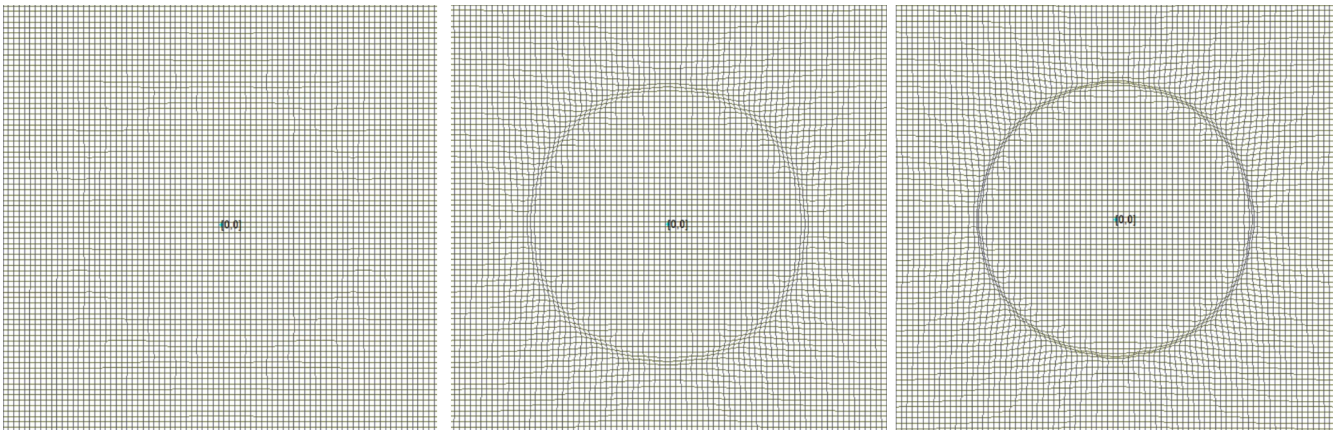


Illustration 7: Tensor morphing to resolve circle in structured grid. Left to right: 1x, 2x, 4x refinement

3.1.2 Structured Bifurcating Trees

Binary trees (such as *quadtrees*, *octrees*) provide the most efficient encoding of volumetric data, yielding constant access time and dynamic refinement with lightweight underlying representation.[14] The domain is divided in each dimension into locations that intrinsically preserve hierarchy information in an integer code, shifting load onto compute device (rather than memory or IO bottlenecks). Arithmetic properties of binary encoding allow direct bit-wise manipulation to resolve parent, child, and neighboring locations without memory access or relational tree traversal. Important for the foreseeable future, trees have promise for far-field physics (coupled to boundary meshes), complex topologies, and multiphase flows.

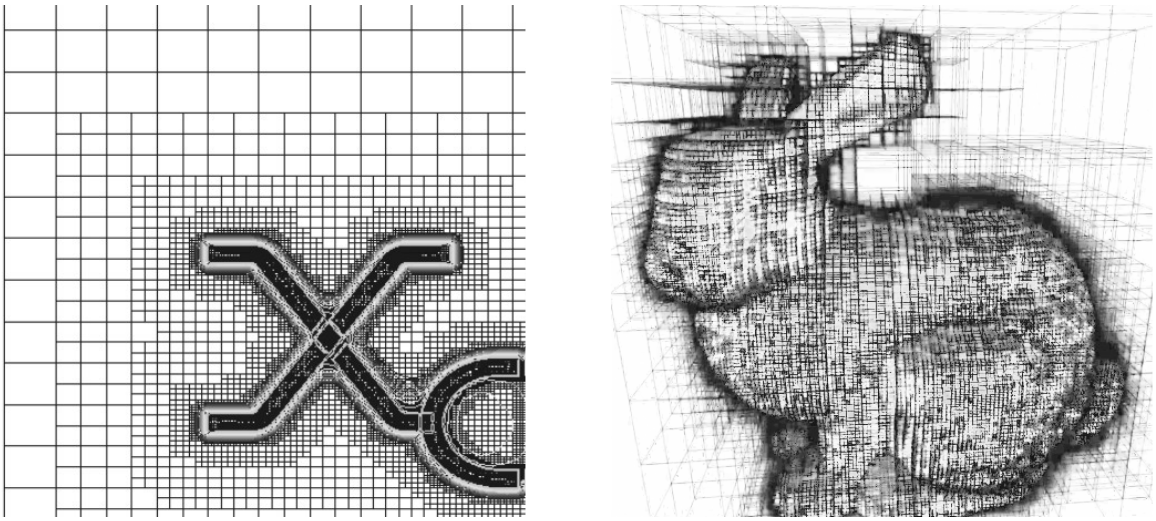


Illustration 8: Examples of trees in 2d and 3d. Left, prototype quadtree demonstrate growth layers around discrete shape. Right, octree (container) improves search space during mesh winding-number calculation.

TREE inherits from STRUCTURED and defines integer location codes for each dimension:

```
std::vector<glm::ivec> locations;       //ijk integer location codes for valid branches, l=level
```

Locations within $\vec{K}$ divide domain in powers of two up to max *L* levels deep (starting at root location $k_o = 2^{L-1}$, computing all attributes from an encoded large integer for each spatial dimension. Location $\vec{k}$ has center position:

$$\vec{x}(\vec{k}) = \vec{x_{min}} + (\vec{x_{max}} - \vec{x_{min}})\vec{k}/2^L \tag{14}$$

Quadrature points are cast into domain and positions are quantized into locations $\vec{x} \to \vec{k}$ to the desired level for initial tree leaves. Remaining tree topology is generated from these leaves using location operators, limiting memory access. Neighbor accessors require knowledge of local tree hierarchy, extracted from any valid location code. Child locations are computed from previous locations at level *l* with $\vec{k_{l+1}} = \vec{k_l} \pm \vec{k_l}/2$. Parent location is found by shifting the least significant true bit (at *L-l-1*) up by one, equivalent to $\vec{k_{l-1}}[L-l] = 1$ and $\vec{k_l}[L-l-1] = 0$. Location code machinery dynamically construct and return elements, starting with overrides for utilities such as nElement:

```
size_t TREE::nNode() const {return (1<<D)*locations.size();};          // 2^D nodes per location
size_t TREE::nEdge() const {return D*(1<<(D-1))*locations.size();};    // D*2^(D-1) edges per location
size_t TREE::nFace() const {return D*(D-1)*locations.size();};         // D*(D-1) faces per location
size_t TREE::nCell() const {return locations.size();}                  // 1 cell per location
```

Procedural overrides used in FDM and FVM redirect to location-based calculations, such as volume:

$$V(\vec{k}) = DOMAIN::volume()/2^{D \cdot l} \tag{15}$$

3.2 Unstructured Declarative Branch

UNSTRUCTURED geometry is an abstract base with explicitly-defined associative topology, requiring additional members to support declarative ELEMENTS (stored in memory):

```
NODES nodes;                // 0d nodes
EDGES edges;                // 1d boundary edges
EDGES connections;          // 1d interior edges
```

3.2.1 Unstructured Closed Meshes

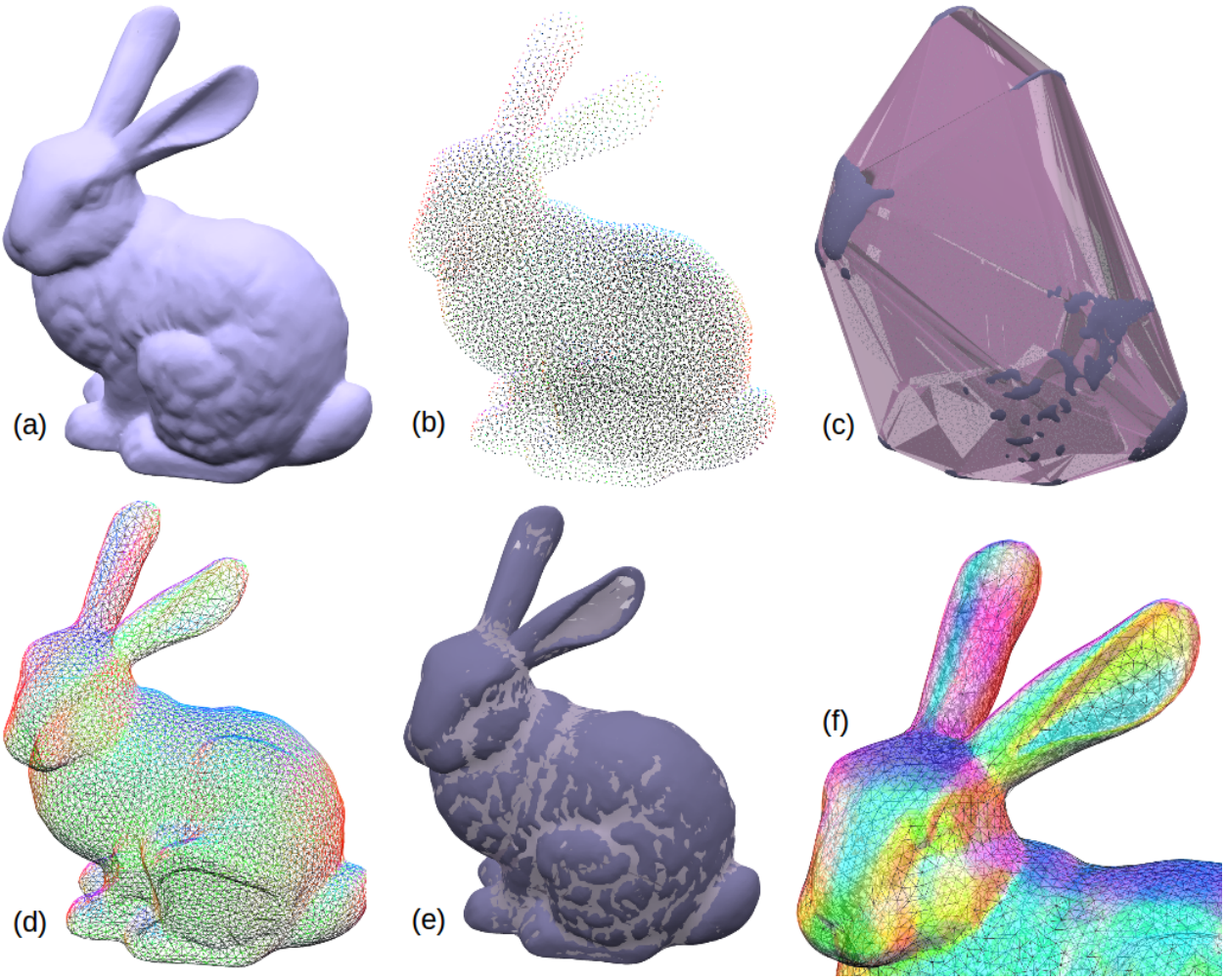


Illustration 9: Stochastic meshing process with 20K nodes, unrefined: a) Shape is defined by reference surfaces used to solve SDF background. b) Nodes are injected into valid SDF locations and equalized. c) Triangulation updates topology with over-wrapped mesh (comprised of convex hulls). d) Elements are evaluated and cleaned against SDF while surfaces are unwrapped. e) Original (darker) mesh superimposed over new mesh to visualize spatial deviation. f) Close-up showing error around high-curvature features.

MESH are UNSTRUCTURED, defined by a closed cellular topology on manifolds $\vec{x} \in \mathbb{R}^D$:

```
FACES faces;                // 2d boundary faces
FACES interfaces;           // 2d interior faces
CELLS cells;                // 3d interior cells
```

Polygonal meshes are common in feature-driven continuum mechanics – a marked improvement in flexibility and capacity to resolve details at varying length scales. *Finite Element Method* (FEM) and *Finite Volume Method* (FVM) are designed for such topologies (yet compatible with other types such as grid).

Generating a computationally-effective mesh remains a big pain point for practicing engineers [2,3]. Not only can geometric definition be tricky, but the resulting domain must be conducive to numerics that follow. Variable sampling may meet feature detail requirements, meanwhile establishing baseline limitations on the Courant number – constraining numerical performance or choice in solver. Ironically, small elements have least significant physical impact on the solution and should be discarded if contributions are below machine precision. Conformance and resolution must find balance with numerical conditioning as penalties amplify in downstream processes.

Meshing tends to fall into two categories: *feature-based* and *stochastic*. Feature-based solutions typically construct the domain by progressively-resolving regions from low-to-high fidelity. Control nodes are defined along parametric curves. Loops of edges and nodes are created along the curve between control nodes. Surface patches and corresponding faces, edges, and nodes are filled between loops. Volumes consisting of cells, faces, edges, and nodes are progressively resolved, providing high feature-control from feature-based representation. Large portions can be dominated by optimal element types (e.g. Hex8 in 3d), yet near complex interfaces, poor elements limit local CFL conditioning or matrix stiffness.[32] This is quantified as a bimodal element quality distribution based on cell circumscribe or potential energy test around each node.

Stochastic mesh generation based on *Per-Olof Persson's thesis* has been implemented – yielding high-quality unimodal distributions to improve computability.[4] A background geometry solves the Signed Distance Field (SDF) around the input reference surfaces to define an implicit shape function $\phi(\vec{x})$ with reference surfaces at *S*.[19] Negative values define interior and positive exterior (though robust sign determination can require expensive *winding-number* computation):

$$|\nabla \phi(\vec{x})| \stackrel{\text{def}}{=} 1 \quad \text{and} \quad \phi(S) \stackrel{\text{def}}{=} 0 \tag{16}$$

Initial nodes are distributed across valid (positive or negative) SDF locations, followed by improvement taking an edge-wise spring-truss analogy. Displacement is constrained normal to boundaries, while nodes near SDF-defined surfaces iteratively project to progressively conform:

$$\Delta \vec{x} = \frac{\phi(\vec{x}) \nabla \phi(x)}{|\nabla \phi(x)|^2} + \epsilon\left(O(\vec{x})^2\right) \tag{17}$$

Upon completion, concave hulls and invalid elements are evaluated at quadrature points against the SDF and cleaned (with a surface unwrapping procedure) yielding naturally-fitting simplexes, to be later interpreted as median duals if desired. Enhancements to feature size and curvature refinement can be implemented by appending algorithms (that process upon SDF or other properties) to the meshing script. Stochastic equalization can be applied after feature-based mesh generation to yield more isotropic CFL characteristics by annealing neighbors *k*, forcing toward local radius goal r_o:

$$||\vec{x} - \vec{x_k}|| = r_o(\vec{x}) + \epsilon\left(O(\vec{x})^2\right) \tag{18}$$

Triangulation size and speed present the primary challenge of stochastic closed-cell techniques. Semi-frequent Delaunay triangulations are required to update cell associativity as inter-node distances equalize and rearrange while annealing. Interval is controlled by tracking maximum relative node displacement since last triangulation, where values $r/r_o \sim 0.5$ demonstrate good balance between triangulation and equalization expense. As an ideal mesh is resolved, triangulations decrease in frequency and the meshing process accelerates. While many other heavy algorithms prove naturally parallel, rapid triangulation of millions of nodes require special adaptation of a native parallel *Delaunay* triangulator based on *GFlip-3d* [6,7] for *OpenCL*-compatible devices.

Linear elements can achieve higher spatial accuracy by reconstructing and storing secondary element neighborhoods locally or using tree-like search spaces, though remain poorly-conditioned beyond second-order accuracy (due to proportional memory cost with diminishing returns). High-speed memory and interconnects remain relatively expensive, so mesh generation and resulting topology footprint keep unstructured meshes from dominating numerical analysis, especially where

processes are resolved (instead of modeled). *Galerkin* approaches alleviate mesh limitations by constructing the solution as the sum of piecewise linearly-independent basis functions, demonstrating forward directions for unstructured domains.

FVM applies *Advection Upstream Splitting Method* (*AUSM*) or Roe flux to integrate median dual node-cells in strong form with *Monotone Upwind Schemes for Scalar Conservation Laws* (*MUSCL*) reconstruction yielding second-order accuracy for each degree of freedom ϕ .[18] A face-volume flux approach results in a robust scheme when convection drives transport. *Venkatakrishnan* and *Barth-Jesperson* limiters help suppress numerical overshoot.[1] Scalar transport around dual node at $\vec{x}$ is the sum of flux density $\vec{\psi}=\phi\vec{u}$ flowing through surrounding *K* median dual fragments with source *q*, area normal *dA*, and volume *dV*, and $\alpha=0.5$ for median sampling: [9]

$$\dot{q}(\vec{x})-\dot{\phi}(\vec{x})=\nabla\cdot\vec{\psi}(\vec{x})=\frac{1}{\Delta V(\vec{x})}\sum_{k=0}^{K}\left[(1-\alpha)\vec{\psi}(\vec{x})+\alpha\vec{\psi}(\vec{x}_k)\right]\cdot\Delta\vec{A}_k+\epsilon\left(O(\vec{x})^2\right) \tag{19}$$

A coefficient table enables arbitrary-order explicit Runge-Kutta time integration, while CFL timing can be adapted from FDM temporal integration strategies (*eq 7)*. Although a robust formulation for continua, where local convection is much less than the mediating wave speed $u(\vec{x})\ll a(\vec{x})$, FVM becomes poorly-conditioned and dominated by numerical diffusion. Elements in viscid flow become numerically inefficient within the boundary layer so a wall function or local time-stepping emolliates restrictions. In cases such as the incompressible limit, a different technique should be applied.

Continuous Galerkin FEM does not rely on velocity transport and can also achieve arbitrary-accuracy with weak form of the PDE by dividing the domain into non-overlapping elements connected by a sum of continuous weighted interpolants. Values are sampled at position $\vec{x}$ by averaging across *N* local element nodal values ϕ_n with sum: [37]

$$\phi(\vec{x})=\sum_{n=0}^{N}L_n(\vec{x})\phi_n+\epsilon\left(O(\vec{x})^K\right) \quad \text{requiring} \quad N\geq\frac{1}{D!}\prod_{d=1}^{D}(K+d) \quad \text{to form a basis.} \tag{20}$$

Lagrange polynomial shape function L_n is evaluated within element $\xi_n\in[-\vec{1},\vec{1}]$, *f*or tensor product elements (3d hex) :

$$L_n(\vec{x})=\prod_{i\neq n}\frac{x-x_i}{x_n-x_i}\prod_{j\neq n}\frac{y-y_j}{y_n-y_j}\prod_{k\neq n}\frac{z-z_k}{z_n-z_k} \tag{21}$$

Other simplex shapes require more involved shape functions and quadrature rules, detailed in references [19] & [20].

Gauss-Legendre quadrature rules exactly integrate shape functions, while gradients are computed in terms of analytic derivatives.[16] For steady-state problems the spatial discretization results in system of nonlinear equations $R_i v=0$, where $v=\left(\phi_0,\phi_1,\ldots,\phi_{NDOF}\right)^T$ is the interleaved assembly of scalar states, with one equation for each unconstrained degree of freedom. This system *R* is solved using *Newton-Raphson* iteration:

$$dv=-R_i v_\tau\left(\frac{\partial R_i v_\tau}{\partial v_j}\right)^{-1} \quad \text{updating state vector} \quad v_{\tau+1}=v_\tau+dv \tag{22}$$

Jacobian $\partial R_i v_\tau/\partial v_j$ is symmetric if the PDE is self-adjoint, and the linear system may be efficiently solved using a preconditioned conjugate gradient iterative solver, such as *OpenCL*-based *ViennaCL*. Temporal steady-state solution can be converged with *Newton-Raphson* iteration. Time-varying domains can use an appropriate ODE solver such as implicit-explicit *Crank-Nicolson* (e.g. Heat Equation) or *Newmark-Beta* (e.g. Linear Elastodynamics). Explicit *Runge-Kutta* integration techniques can also be used for transient analyses, though as §4 indicates, memory constraints can limit scale of space-time resolved analyses (in favor for structured types).

3.2.2 Unstructured Open Networks

NETWORK are UNSTRUCTURED, distinguished by an open *Markov-chain* structure to constitute the most casual topology associativity. If associativity information can construct a graph-like sparse matrix, stability may not be guaranteed, so an implicit technique may be required to converge upon steady-state or harmonic response using eigenvalue analysis.[17] Markov matrix $\bar{A}$ can be processed to yield non-linear $\vec{v}_{n+1} = \bar{A}_n \vec{v}_n$ or harmonic response, though other method can prove more numerically efficient by removing the need for topology altogether.

Particle-based methods or Lagrangian networks (from *N-body* down to *zero-body*) are often used when gradients are very large or when a statistical approach (non-continuum) is desired such as rarefied and plasma flows. Lagrangian domains displace $\vec{x}$ as $d\vec{x} = \dot{\vec{x}}\, dt$ from local data or background geometry, naturally refining node concentration to track physical density (as proxy for particle groups).[23,26] Dynamic associativity makes particle methods well-suited for non-linear interactions such as fracture mechanics and multi-physics.[24] However, the initial node distribution and correction for boundary effects can be precarious. *Particle In Cell* (PIC) utilizes a background geometry or associative map to track individuals or groups against background fields to accumulate rarefied gas or plasma field data.

Radial Basis Functions (RBF) use a standard potential function around each particle to reduce *N-body* complexity using a differentiable compact stencil. *Smoothed Particle Hydrodynamics* (SPH) also localizes the strong form of the PDE through convolved integration.[13] Mesh-free methods eliminate the requirement for a formal associative topology, facilitating open networking and new opportunities for hybrid techniques such as *Direct Simulation Monte-Carlo* (DSMC). RBF can apply Lie groups for direct integration, as well as indirect integration with *Reproducing Kernel Particle Method* (RKPM).[35]

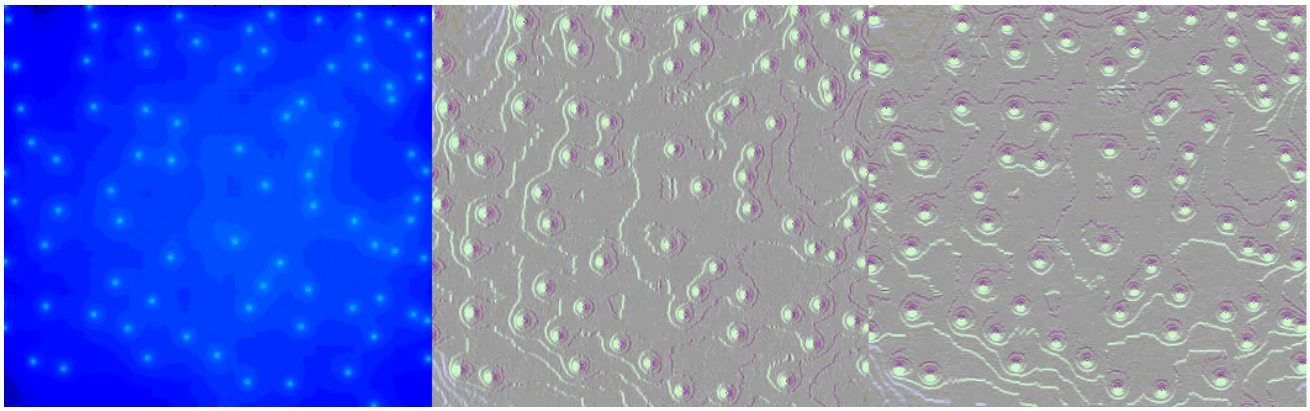


Illustration 10: Experimental RBF method applied to electrostatics, from left to right: Electrostatic potentials around each RBF node center, force field in the x-direction, force field in the y-direction. At equilibrium potential gradients at each node are zero, and when not, gradients evaluate positive or negative from neighboring contributions resulting in local acceleration. Iso-potential contours are naturally apparent between nodes, this pattern resulting shortly after release of a random initial distribution into a vacuum.

Convolved integration evaluates inter-particle forces as the gradient of local net potential field, computed as the superposition of displaced standard field potential $\phi_o(\vec{r})$ around each node. Extensions in quantum mechanics and vorticity methods may utilize a complex basis potential. Applied to mesh equalization, the standard potential function could be selected as the integral of *Per-Olof Persson's* linear radial spring-truss analogy $f(r) = -max(1 - r/r_o, 0)$ [4]:

Defining standard potential as the integral $\phi_o(r) \stackrel{\text{def}}{=} -\int_{\infty}^{r} f(r)\, dr$ from a forcing function. (23)

Applied to electrostatics, standard potential function $\phi_o(\vec{r})$ is the integral of the electric force:

$$\phi_o(r) = -\int_{\infty}^{r} \frac{r_o^2}{r^2} dr = \frac{r_o^2}{r} \quad \text{using} \quad f(r) = -\left(\frac{r_o}{r}\right)^2 \quad \text{for this example.} \tag{24}$$

Node positions are projected and superposed onto the background grid using the standard basis potential $\phi_o(\vec{x})$ around each center (discretely expressed as an interpolated Dirac delta δ or radially symmetric function with sum of 1.0), resulting in extensive net scalar potential field $\phi(\vec{x})$ with weights (or charge) λ and compact support for accuracy *K-1:*

$$\phi(\vec{x}) = \sum_{n=0}^{N} \lambda_n \phi_o(|\vec{x} - \vec{x}_n|) = \sum_{n=0}^{N} \lambda_n \delta_n^K * \phi_o^K + \epsilon(O(\vec{x})^K) \tag{25}$$

Local force field $\vec{f}(\vec{x})$ is determined from convolution-based gradient of the potential function as $\delta_{\nabla}^{\dot{K}} * \phi(\vec{x})$. Local forces are sampled at each node and pseudo-physical mass is applied to impose a timescale:

$$\vec{f}(\vec{x}) = -\nabla \phi(\vec{x}) = \frac{d(m\dot{\vec{x}})}{d\tau} \tag{26}$$

As an optimization, sum and gradients can be evaluated only locally around nodes. Non-linear second-order equation emerges, harmonic in $\vec{x}$:

$$m\ddot{\vec{x}} + \dot{m}\dot{\vec{x}} + \nabla \phi(\vec{x}) = \vec{0} \tag{27}$$

Steady-state is reached when $\ddot{\vec{x}} \to \vec{0} \wedge \dot{\vec{x}} \to \vec{0} \wedge \nabla \phi(r=0) \to \vec{0}$. While many applications utilize constant mass, the $\dot{m}$ term can be augmented with artificial damping ζ_o to facilitate convergence (at possible expense of global accuracy). Light damping is often appropriate – its effect on internal energy $e(\vec{x}) = \phi(\vec{x})/\rho(\vec{x}) - (\dot{\vec{x}} \cdot \dot{\vec{x}})/2$ can be quantified and correlated to traditional quality metrics.

3.3 Background Geometries

Any geometry can assign a utility "background" for one-way interpolation, often STRUCTURED for sampling efficiency. Background geometries provide a computational canvas for spatial algorithms such as unstructured mesh generation. Before meshing, the background geometry solves the SDF to provide the essential implicit shape representation required for node equalization [3,5]. Foreground can then sample against background geometry field for SDF (or other) interpolation. In similar manner (and/or with trees), multi-grids can be implemented. The resolution of the background ϕ^* dictates the spatial detail level resolved in the foreground ϕ at $\vec{x}$, using a local sampling strategy of the arbitrary form:

$$\phi(\vec{x}) = \frac{1}{k_\Sigma} \sum_{n=0}^{N} k_n \phi_n^* \quad \text{, where} \quad k_\Sigma = \sum_{n=0}^{N} k_n \tag{28}$$

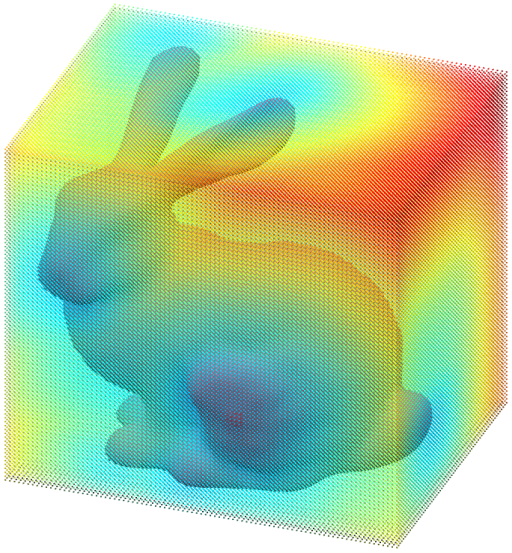


Illustration 11 SDF background field around reference surfaces defined by discrete triangles or analytic functions.

Structured Sampling

Type	Sample Count N	Weighting Coefficient k_n	Description
Grid	2^D	$\prod_{d=0}^{D} \left\| \frac{\Delta\vec{x} - \left\|\vec{x} - \vec{x}_n\right\|}{\Delta\vec{x}} \right\|_d$ (29)	linear volumetric fraction, node-centered
Tree	$K \cdot 2^D$	$\sum_{k=0}^{K} b_k \prod_{d=0}^{D} \left\| \frac{\Delta\vec{x}_k - \left\|\vec{x} - \vec{x}_n\right\|}{\Delta\vec{x}_k} \right\|_d$ (30)	recursive volumetric sum, cell-centered level weights b_k of accuracy K

4 Numerics Summary

Type	Characteristic	Supported Methods	Spatial Resolution (per dimension)Ω	Spatial Accuracy	Temporal Accuracy	IO Size (B/element)	RAM (B/element)Ξ	Speed-Up (GPU/CPU)Ψ
Mesh	closed simplex	FEM, FVM, RBF**	$1/\epsilon \approx 10^{15}$	1, 2	1, 2, +	240-360	800-2400	11-17x
Network**	open graph	FEM, PIC**, RBF**	$1/\epsilon \approx 10^{15}$	1	1	240-360 Γ	800-2400 Γ	TBD
Grid	regular indexed	FEM, FVM, LBM*, FDM**	$\sqrt[D]{2^{31}} \approx 10^3$	1, 2, +	1, 2, +	4	16-32	9-28x
Tree**	branch locations	FVM, FDM**	$2^{30} \approx 10^9$	1, 2	1, 2, +	16	16-32 Γ	TBD

Ω assumes 64-bit-float, 32-bit integer index (~2B element max per geometry)
Ψ estimate based on single $1000 desktop with approx 1M elements, see *Appendix B* [36]
Γ estimate inferred on similar branch attributes and functions
Ξ includes only geometry kernel CPU memory, not total application memory, assumes 3d

4.1 Native Formats

File	Open	Save	Context	Note
xcs	Yes	Yes	system	system setup file with parameterized setup and references
xcg	Yes	Yes	geometry	geometry file containing spatial data and topology
xco	Yes	Yes	data object	object output file for each property-key for each system
xcm	*	*	metaobject	media profile for user graphics
xcl	**	**	algorithm	compute language based on OpenCL with markups

4.2 Exchange Formats

File	Import	Export	Context	Note
stl	Yes	Yes	surfaces	binary and ascii import, binary-only export
obj	**	**	surfaces	Wavefront polygon format
ply	**	**	surfaces	Stanford polygon file format
msh	Yes	Yes	geometry	gmsh open format
vol	Yes	**	geometry	neutral mesh format
su2	**	**	geometry	Stanford open-source mesh
csv	**	Yes	system	comma separated value
vtu	**	Yes	system	Paraview dataset
xlsx	**	**	system	Excel spreadsheet based on XML
stp	**	**	system	standard exchange of product data

* beta, **scheduled

5 Declarations & Definitions

5.1 Heterogeneous Data – Regularity

DATA defines a map-like container that manages multiple VECTOR entries by Property Key. VECTOR consists of a master host array, analogous device buffer, and revision integers for each. Property Key *pk* is a unique key composed of a property attribute followed by any number of modifiers (to facilitate lexicographical look-up in DATA). A heterogeneous (CPU-GPU) algorithm may have the following programming pattern:

Contiguous VECTOR representing element values within field $\phi(\vec{x})$ and positions $\vec{x}$ can be accessed from DATA:

```
if (!data.contains(Positions))          // check to see if data contains an entry for property key "Positions"
     return false;                      // if not, do something else
VECTOR& positions = data(Positions);    // get a reference to positions vector (interleaved xyzt)
```

Operate concurrently on host VECTOR using *OpenMP C++*:

```
dmat M = Constants(Transform);          // get global data constant for "transform"
#pragma omp parallel for                // perform next for loop in parallel
for (int n=0; n<positions.rows(); n++)  // loop through all nodal positions, arranged in rows
     positions[n] = M * positions[n];   // perform computation without write collision
```

Synchronize VECTOR to device in preparation for heterogeneous processing:

```
positions.revision++;                   // increment revision to indicate host vector has changed
positions.sync();                       // if device vector revision is less than host, update device vector
```

Algorithms and bound arguments consolidate code fragments into OpenCL kernel sources. A builder manages and resolves bindings based on string literal substitution. The parallel portion of the above loop may generate CL equivalent:

```
const int n = get_global_id(0);                  // get this node (or element) index
positions_arr[n] = M_const * positions_arr[n];   // perform some computation
```

Instructions are invoked from a script able to execute C++ or OpenCL versions of the algorithm. After kernel launch and processing, device values can be buffered back to CPU memory space using revision and synchronization:

```
positions.device.revision++;            // increment device revision to indicate buffer has changed
positions.sync();                       // if device revision is greater than host, update host vector
```

5.2 Local & Global Coordinates

Interacting systems require a common global frame. Positions (and spatial vectors) within GEOMETRY are by convention described in untransformed local coordinates. One *3x3* (or *4x4)* homogeneous model matrix per domain provides the absolute transformation between local and global space $\vec{x}_{global} = Q_{global}\vec{x}_{local}$. Global coordinates for a domain are a product of local position with a global model matrix, constructed from the recursive product of local transforms in hierarchy:

$$Q_{global} = \prod_{local=root}^{domain} Q_{local} \tag{31}$$

If transformation Q_{local} is altered, children are recursively refreshed and a new Q_{global} is calculated for each domain to keep position accessors synchronized. This enables precise compound coordinate transformations and large-scale reuse while maintaining a common global state and sub-states (that can be reset to defaults within each child). Local model matrix can be individually controlled in static or dynamic routines (i.e. multi-DOF robot arm or moving chimera mesh).

5.3 Spatial Domains

DOMAIN is a base class that describes the spatial extrema with broad utility in numeric and other contexts. All spatial types inherit its convenient functionalities. DOMAIN defines common member attributes:

```
glm::dvec min;                  // minimum spatial extrema
glm::dvec max;                  // maximum spatial extrema
int D;                          // spatial dimensionality (1d, 2d, 3d)
bool global;                    // positions in global coordinates?
bool valid;                     // domain has been set and is usable?
```

5.4 Computable Elements

ELEMENT is defined as a discrete polyhedral simplex consisting of one or more node vertexes and quadrature points, often generated through an automated construction process (e.g. mesh generator). In memory they can be represented in contiguous form, enabling concurrent functions to return a single element in constant time while processing. Access patterns remain universal for querying number of elements with *nElement*, or returning an actual element with *getElement* functions, while underlying machinery is polymorphic.

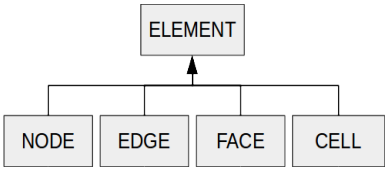


Illustration 12: Element Inheritance

ELEMENT defines a virtual simplex with base member attributes with up to *E* indices:

```
std::array<int, E> indices;     // indices of the nodes that constituent this element
int type;                       // simplex sub-type (Node1, Tri3, Tet4)
int id;                         // region identifier to map surfaces, volumes, edges, etc.
glm::dvec center;               // element barycenter position
```

NODE, EDGE, FACE, CELL derive from ELEMENT and append additional member attributes:

```
double NODE::volume;            // dual-cell control volume around node
glm::dvec EDGE::length;         // distance vector between two nodes
glm::dvec FACE::area;           // area normal vector from barycenter
double CELL::volume;            // scalar volume
```

ELEMENTS allow many simplexes to be stored and buffered as contiguous arrays (and device buffers). Each element may have different storage characteristics (and variable stride), so memory offsets enable constant-time and concurrent access with heterogeneous storage. This machinery is not exposed in the API, but are implemented under unstructured geometry types as declared topology:

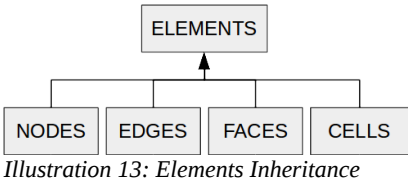


Illustration 13: Elements Inheritance

```
std::vector<int> types;         // enumerated types of each element stored
std::vector<int> indices;       // node indices with a variable strides
std::vector<int> offsets;       // node offset of each element as stride accumulates
std::vector<int> ids;           // identifiers to group elements for surfaces, volumes, edges, etc
std::vector<glm::dvec> centers; // element barycenter position
```

NODES, EDGES, FACES, CELLS derive from ELEMENTS and append member attributes:

```
std::vector<double> NODES::volumes;     // dual-cell control volumes around nodes
std::vector<glm::dvec> EDGES::lengths;  // distance vectors between nodes
std::vector<glm::dvec> FACES::areas;    // area normal vectors from barycenter
std::vector<double> CELLS::volumes;     // scalar volumes
```

Example: Accessing an Element – Two Approaches

A. Generic access using base ELEMENT (discarding RTTI):

```
ELEMENT element = geometry.getElement(elementIndex, 1);       // get element of linear dimensionality
double len = Length(geometry.position(element.indices[0]) - geometry.position(element.indices[1])); // yuck
```

B. Specialized access using derived EDGE (preserving RTTI):

```
EDGE edge = geometry.getEdge(INDEX edgeIndex);     // get edge using an index
double len = Length(edge.length);                  // look up specialized attribute, rather than calculating it
```

5.5 Element Traits

Implemented elements define vertexes and associativity, but some basic attributes are universal for a specific element type and should not be patterned. These globals are defined statically to map elements to attributes and functions. For instance, a Tet4 always has four nodes, so it is superfluous to include this information along with each defined element. Instead, we can call static function ElementTraits::nNode(ElementType type) to determine the number of nodes within an element from its type. Additional properties such as number of quadrature points and topological dimensionality can be queried. Specialized functions specify how to decompose into sub-elements and visualize each type.

5.6 Regions of Interest

Indexing multiple elements into regions allows users and algorithms to define and select specific spaces (as a subset of the larger topology), such as required when applying boundary conditions to surfaces or immersive conditions within volumes, or simply for export. Uniform access patterns across elements and regions enables efficiency in programming, computation, and user interaction. Inherited region types specialize definition of new attributes and functions specific to dimensionality *d*.

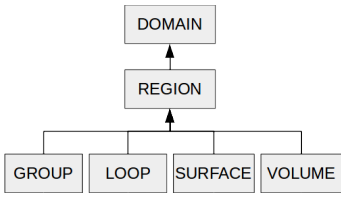


Illustration 14: Region Inheritance

REGION inherits from DOMAIN, specializing as a collection of elements of similar dimensionality (e.g. group of nodes, loop of edges, surface of faces, volume of cells):

```
GEOMETRY* geometry;              // ptr to parent that owns this region
int id;                          // identifier linked to elements with same id
string name;                     // assigned literal name
std::set<int> nodes;             // unique node indices that comprise this region
int nElement, offset;            // regions don't store elements, but jump to entry via geometry
```

Region-Element-Simplex Compatibility

d	Region	Element	Simplex
0	GROUP	NODE	Node1
1	LOOP	EDGE	Line2, Line3**, Line4**
2	SURFACE	FACE	Tri3, Tri6*, Tri10**, Quad4, Quad8**, Quad9**
3	VOLUME	CELL	Tet4, Tet10*, Prism6*, Hex8, HeX20**, HeX27**

** beta, **scheduled*

6 Conclusion

A set of qualitatively-orthogonal geometric constructs aims to facilitate spatial unification of common methods such as FDM, FEM, and FVM, as well as rapid expansion into novel numerics including LBM, PIC, and RBF methods. Each type possesses characteristic strengths and weaknesses, whereby the union of all types and subtypes form a complete basis for numerical computing (in different scenarios and combinations). Inheritance demonstrates clean program structure and enables major polymorphic optimizations within structured and unstructured computable domains. At least five numerics families were implemented in over one hundred algorithms to test and iterate on generalities and data structures resulting in a geometry kernel of approximately six-thousand lines of production C++14 and OpenCL, approximately one-quarter of server-side application. Experimental techniques appear to also fall within these classifications and planned expansions. 100M element geometries were tested on workstations with 32 GB of RAM, confirming 50-fold less memory required in structured types compared to unstructured types, implying on identical hardware that resolution can theoretically be further resolved by this factor if such types can be implemented. In continua, this may result in widespread adoption of tree geometries for far-fields and adapted grids or (least favorably) meshes for near-fields, with boundary algorithm mediating independent solvers optimized for the physics within each domain.

6.1 Future Work

- Deployment of LBM, FDM, and review impact on geometry kernel
- Implementation of TREE to improve meshing resolution and support advanced FVM
- Consolidation of FEM machinery to complete unification of the geometry kernel
- Expansion in exchange formats, including parametric data

6.2 Related Work

- A Scalable Convention for Data Properties in Numerical Computing
- Method of Automated Kernel Generation for Heterogeneous Processing
- Transmission Protocols for Numerical Data and Distribution

6.3 References

[1] Jiri Blazek. “ Computational Fluid Dynamics: Principles and Applications.” Elsevier, 2001
[2] WN Dawes, WP Kellar, SA Harvey. “Using Level Sets as the basis for a scalable, parallel, geometry engine and mesh generation system.” AIAA-2009-0372.
[3] WN Dawes, WP Kellar, SA Harvey, S Fellows, D Jaeggi. “A practical demonstration of scalable, parallel mesh generation.” AIAA-2009-0981.
[4] Per-Olof Persson. “Mesh Generation for Implicit Geometries.” MIT Department of Mathematics, 2005.
[5] Marc Schwalbach. “Application of Chimera Grids in Rotational Flow.” RWTH Aachen University, 2014.
[6] Jonathan Richard Shewchuk. “Theoretically Guaranteed Delaunay Mesh Generation – In Practice.” UC Berkeley, 2004.
[7] Ashwin Nanjappa. “Delaunay Triangulation in R3 on the GPU”. Department of Computer Science, National University of Singapore, 2012.
[8] C Praveen. “Finite Volume Method on Unstructured Grids.” Tata Institute of Fundamental Research, Center for Applied Mathematics, 2013.
[9] Anthony Corso. “Outline of FVM Scheme.” Technical Memorandum, Xplicit Computing, 2015.
[10] Thomas Economon, Chao Chen, Pravan Bharadwaj, Eric Darve, Juan Alonso. “Developing Solvers in the Legion Programming System.” ASC PSAAP II, Stanford University, 2014.
[11] Thomas Economon, Francisco Palacios, Juan Alonso. “Towards High-Performance Optimizations of the Unstructured Open-Source SU2 Suite.” 53rd AIAA Sciences Meeting. Stanford University, 2015.
[12] Lucy T Zhang, Gregory J Wagner, Win K Liu. “A Parallelized Meshfree Method with Boundary Enrichment for Large-Scale CFD. “ Journal of Computational Physics 176, 483-506, 2002.
[13] Shaofan Li, Wing Kam Liu. “Meshfree and particle methods and their applications.” Applied Mechanics Review, vol 55, no 1, Jan 2002. American Society of Mechanical Engineers.
[14] Sarah F Frisken, Ronald N Perry. “Simple and Efficient Traversal Methods for Quadtrees and trees.” Mitsubishi Electric Research Laboratories, November 2002. www.merl.com
[15] Stephen B Pope. “Turbulent Flows” Cornell University. Cambridge Press, 2000
[16] Hughes, T. J. (2012). The finite element method: linear static and dynamic finite element analysis. Courier Corporation.
[17] Klaus-Jugen Bathe, Edward L Wilson. “Solution Methods for Eigenvalue Problems in Structural Mechanics.” UC Berkeley, 1973.
[18] Harvard Lomax, Thomad Pulliam. “Fundamentals of Computational Fluid Dynamics.” NASA Ames Research Center, 1999
[19] Tyler Olsen. “Mathematical Representation of FEM Module Operations”. Xplicit Computing, July 2016.
[20] Tyler Olsen. “Yet Another Finite Element Library.” MIT, June 2016.
[21] C Dapogny, P Frey. “Computation of the signed distance function to a discrete contour on adapted triangulation.” Centre de Math, Ecole Polytech Palaiseasu, Renault Guyancourt, UPMC Univer Paris
[22] J C Carr, R K Beatson, J B Cherrie, TJ Mitchell, WR Fright, BC McCallum, T R Evans. “Reconstruction and Representation of 3D objects with Radial Basis Functions”. University of Canterbury
[23] Saurabh S Sawant. “Development of an AMR Octree DSMC Approach for Shock Dominated Flows” University of Illinois at Urbana-Champaign, 2015.
[24] Shaofan Li, Wing Kam Liu. “Meshfree and particle methods and their applications”. UC Berkeley, Northwest University, ASME 2002.
[25] John T Batina. A Gridless Euler/Navier-Stokes Solution Algorithm for Complex-Aircraft Applications. LRC NASA, Feb 1993.
[26] Thomas E Schwartzentruber, Leonardo C Scalabrin, Iain D Boyd. “Modular Implementation of a Hybrid DSMC-NS Algorithm for Hypersonic Non-Equilibrium Flows” University of Michigan, 2007.
[28] Steven W Smith “The Scientist and Engineer’s Guide To Digital Signal Processing” 606-630, 1997
[29] Burak Korkut, Deborah A Levin, Ozgur Tumuklu.“Simulations of Ion Thruster Plumes in Ground Facilities using Adaptive Mesh Refinement” Penn State University, UIUC
[30] Jeremy Aaron Kolker Horwitz. “Lattice Boltzmann Simulations of Multiphase Flows.” UIUC, 2013.
[31] J R Dormand, P J Prince. “A family of embedded Runge-Kutta formulae”. Journal of Computational and Applied Mathematics, vol 6, no 1, 1980.
[32] S N Atluri, T Zhu. “A new Meshless Local Petrov-Galerkin approach in computational mechanics”. Computational Mechanics 22, 117-127, UCLA, 1998.
[33] Pletcher, Tannehill, Anderson. “Computational Fluid Mechanics and Heat Transfer” CRC Press, Third Edition.
[34] Franklin, Powell, Emami-Naeini “Feedback Control of Dynamic Systems” Pearson Press, Fifth Edition.
[35] Wing Kam Liu, Sukky Jun, Yi Fei Zhang. “Reproducing Kernel Particle Methods” Numerical Methods for Fluids, Volume 20, issue 8-9, 1995.
[36] Anthony Corso. “X2 Benchmarking Tool Demonstration” Xplicit Computing Jan 27, 2017.
[37] J S Hesthaven, T Warburton. “Nodal High-Order Methods on Unstructured Grids”. Journal of Computational Physics, 2002.

Appendix A – Function Tables

Spatial Common Functions

Function Name	Description	Arguments	Domain	Geometry	Str'd	Grid	Tree	Unstr'd	Network	Mesh
build	construction sequence			V		O	O		O	O
setKernelSource	define CL kernel source	incl, args, body, needed		V		O	O	O		
setKernelArgs	set CL kernel arguments	args, kernel, prefix		V		O	O	O		
clear	clear contents		V	O		O	O	O		
envelope	envelope nodes (w/wo mask)	spatial, group, reset	T							
setMinMax	set min and max	position, reset	F							
getGlobal	get global domain	glm::mat	F							
delta	difference between max and min		F							
center	center position calculation		F							
contains	contains position array	positions	T							
contains	contains position	position	V	O		O				
volume	total volume calculation		V	O		O				
intersects	intersects ray test	ray	V	O		O				
intersects	intersects another domain	domain	V	O		O				
position	get position from node index	index		V		O	O	O		
getElement*	get element from element index	index, type		V		O	O	O		
nElement*	get element count	type		V		O	O	O		
nNeighbor	get node neighbor count	index		V		O	O	O		
getAdjaceny	get element adjacency	index		V				O		
sample	sample values at position	position		V		O	O	O		
makeRegions	make regions from elements	elements		T						

Element Common Functions

Function Name	Description	Arguments	Element	Node	Edge	Face	Cell
empty	check for valid information		F				
contains	check whether element uses node index	index	F				
sort	reorder elements based on element id		F				
decompose	decompose element into more primitive elements		F				
visualize	represent the element by points, lines, triangles		F				
isNeighbor	search for common topology	element	F				
getDomain	get local domain around element	geometry	F				

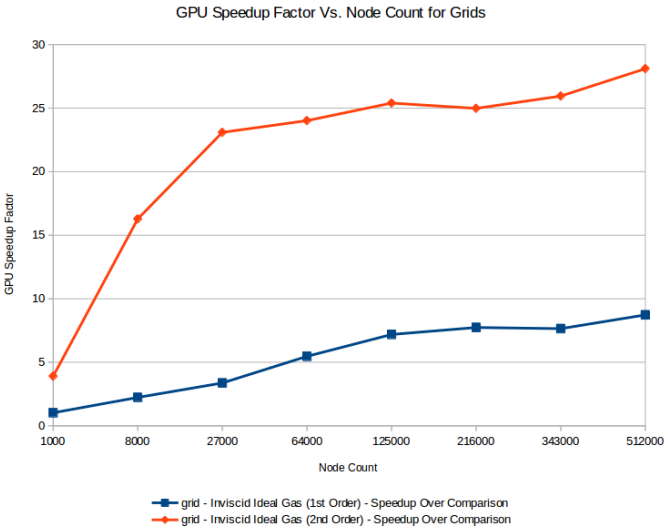
Region Common Functions

Function Name	Description	Arguments	Region	Group	Loop	Surface	Volume
nElement*	number of elements in region		V	O	O	O	O
getElement*	get an element by index in region	index	V	O	O	O	O
position	get element position at index	index	F				
volume	get element volume at index	index	F				
createGroup	create unique nodes from elements		F				

* specialized for derived elements (getNode, getEdge, getFace, getCell; and nNode, nEdge, nFace, nCell; and interior versions)

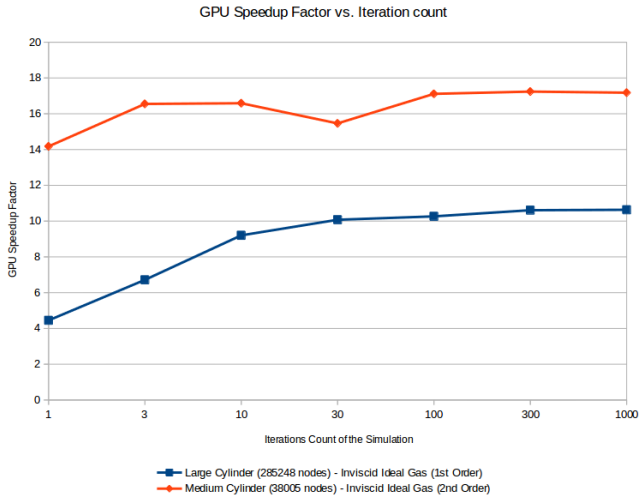
F = member function V = virtual base function O = function override T = function template

Appendix B – Preliminary FVM benchmarking, January 2017 [36]



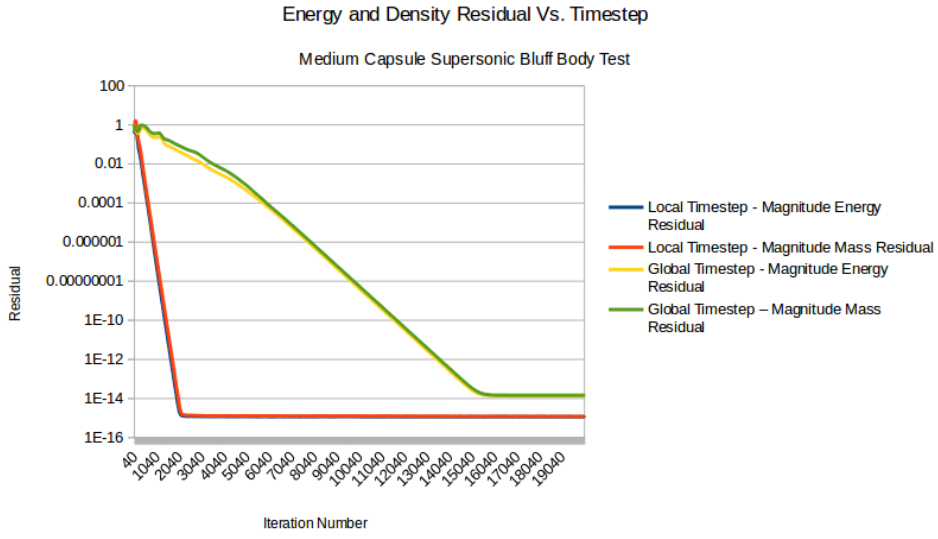
CPU : Intel(R) Core(TM) i5-4670K @ 3.40GHz
GPU : GeForce GTX 780

GPU delivered sizable speed-up over CPU for both structured and unstructured geometries. Benefits were most apparent in larger geometries and higher-order solvers. FVM tests reveal more than 15x speed-up on low-cost consumer GTX 780GPU as compared to roughly equivalent cost quad-core i5-4670K CPU. For sufficiently-large resolution, second-order grid performance plateaus around 25x, and continues to climb. However, performance ratios increase more gradually for first-order schemes, likely due to increased proportion of overhead to computation.



Different setups will yield varying results, though underlying characteristics should be apparent. CPU with more threads will improve performance in parallel-intensive algorithms, perhaps preferred for smaller coupled geometries. Very large simulations beyond a billion elements may require running on CPUs (as there may not be enough co-processor memory). Distributed approaches are also viable.

A small relative speed increase is measured after a number of iterations as the application compiles the compute program dynamically upon launch resulting in an initial timing impediment that is more prominent at fewer iteration counts.



Convergence rate is also important and varies depending on solver, geometry, and setup. Time-accurate simulation (aka global time-stepping) convergence remains fairly slow with FVM meshes if there is poor local CFL conditioning (leading to more numerical diffusion). Local time-stepping accelerates convergence by about an order magnitude (resulting in phase error in space-time solution).

Other techniques such as FDM are suited to time-resolved analyses, while FEM for direct or implicit-explicit integration – with their own convergence characteristics.