Exceptional service in the national interest

Sandia National Laboratories

WCCM: July 31 - August 5, 2022

Hyperdimensional, Adaptive Finite Elements Using Camellia and Intrepid2

Nathan V. Roberts
nvrober@sandia.gov
Sandia National Laboratories

boratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LI
Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE

Outline

Structure Preservation

Typical high-level FEM codes ignore or discard structure in order to maintain generality.

Structure Preservation

Typical high-level FEM codes ignore or discard structure in order to maintain generality.

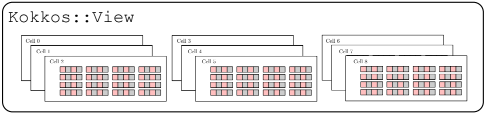
Example: using the standard Intrepid2 interface, if you want Jacobians on an affine grid, you compute and store these at each quadrature point, in a multi-dimensional array (a Kokkos View) with shape (C,P,D,D). This is wasteful, and waste grows with polynomial order and number of spatial dimensions.

By contrast, a custom implementation could store the same Jacobians in a (C,D,D) array. For a uniform grid, this reduces to an array of length (D).

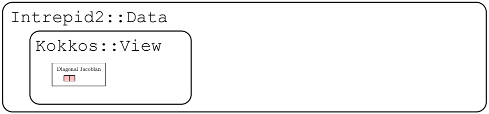
Structure Preservation

The new Intrepid2 Data class is a starting point for addressing this. It stores just the unique data, but presents the same functor interface as the standard View.

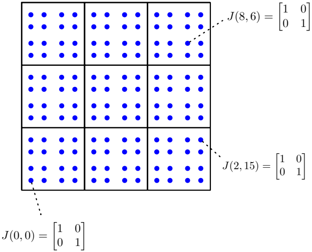
Old way: 4 doubles per Jacobian per point per cell.



New way: 2 doubles.



Same access pattern for both old and new:



Our interest is not primarily in reducing storage costs, but in enabling structure-aware algorithms, such as sum factorization.

Motivation: Sum Factorization

Assembly/Evaluation Costs[1]

	Storage	Assembly	Evaluation
Full Assembly + matvec	$O(p^{2d})$	$O(p^{3d})$	$O(p^{2d})$
Sum-Factorized Full Assembly + matvec	$O(p^{2d})$	$O(p^{2d+1})$	$O(p^{2d})$
Partial Assembly + matrix-free action	$O(p^{d})$	$O(p^{d})$	$O(p^{d+1})$

For hexahedral elements in 3D:

- standard assembly: $O(p^9)$ flops
- sum factorization: $O(p^7)$ flops in general; $O(p^6)$ flops in special cases.
- partial assembly: $O(p^4)$ flops (but need matrix-free solver)

Savings increase for higher dimensions...
Basic idea: save flops by factoring sums.

	Adds	Multiplies	Total Ops
$\sum_{i=1}^{N} \sum_{j=1}^{N} a_i b_j$	$N^2 - 1$	N^2	$2N^2 - 1$
$\sum_{i=1}^{N} a_i \sum_{j=1}^{N} b_j$	$2N - 2$	N	$3N - 2$

[1] Table 1 in Anderson et al, MFEM: A modular finite element methods library. doi: 10.1016/j.camwa.2020.06.009.

Intrepid2's Basis Class

- Principal method: `getValues()` — arguments: points, operator, Kokkos View for values
- Fills View with shape (P) or (P,D) with basis values at each ref. space quadrature point.

Structure has been lost:

- points: flat container discards tensor structure of points.
- values: each basis value is the product of tensorial component bases; we lose that by storing the value of the product.

Both points and values will generally require (a lot) more storage than a structure-preserving data structure would allow.

But our major interest is in supporting algorithms that take advantage of structure: we add a `getValues()` variant that accepts a `BasisValues` object (see next slide).

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.
- `TransformedBasisValues`: `BasisValues` object alongside a transformation matrix, stored in a `Data` object, that maps it to physical space.

Two Sum Factorization Approaches

In N-dimensional hypercube integration, we can have $N + 2$ nested summations; we want to compute and store these in an efficient manner.

We implement two sum factorization algorithms:

1. **Basis-indexed**:
   - standard approach (see e.g. Mora & Demkowicz)
   - loop nesting structure: point loops contain basis loops
   - intermediates are indexed by basis ordinals, with implicit reference to quadrature indices
2. **Point-indexed**:
   - our design, based on Intrepid2 data layout: we attempt to improve data locality.
   - loop nesting: basis loops contain point loops
   - intermediates are indexed by point ordinals, with implicit reference to basis ordinals

Estimated Flops for Each Algorithm

Sandia National Laboratories

We use Poisson assembly on a 16^3 grid, with elementwise integrals of the form

$$K_{ij} = \int_K \nabla\phi_i \cdot \nabla\phi_j \, \partial K,$$

as our test problem. We implement a flop estimator (counting each add or multiply as one flop), with results:

p	Standard	Basis-Indexed	Speedup	Point-Indexed	Speedup
1	1.6e+07	2.7e+07	0.60x	2.9e+07	0.55x
2	5.3e+08	3.6e+08	1.5x	3.8e+08	1.4x
3	6.7e+09	2.4e+09	2.8x	2.5e+09	2.7x
4	4.9e+10	1.1e+10	4.5x	1.1e+10	4.5x
5	2.5e+11	3.7e+10	6.8x	3.9e+10	6.4x
6	1.0e+12	1.1e+11	9.1x	1.1e+11	9.1x
7	3.3e+12	2.7e+11	12x	2.7e+11	12x
8	9.6e+12	6.0e+11	16x	6.1e+11	16x

(Speedup values here are theoretical, based only on flop counts.)

Poisson Results: Serial

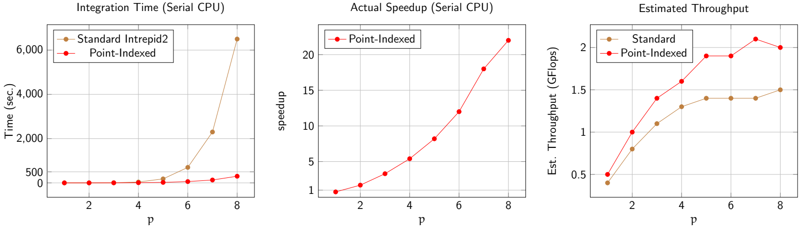


Figure: Serial (Intel Xeon W, 2.3 GHz) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Poisson Results: OpenMP

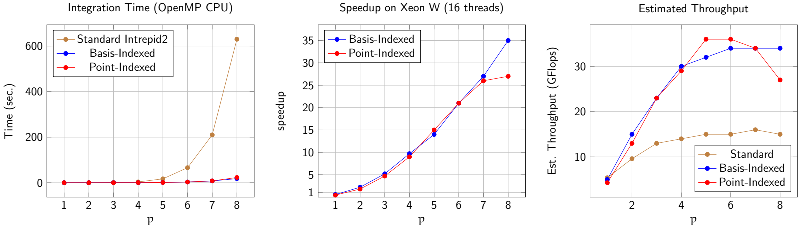


Figure: OpenMP (Intel Xeon W, 2.3 GHz, 16 threads) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Poisson Results: CUDA P100

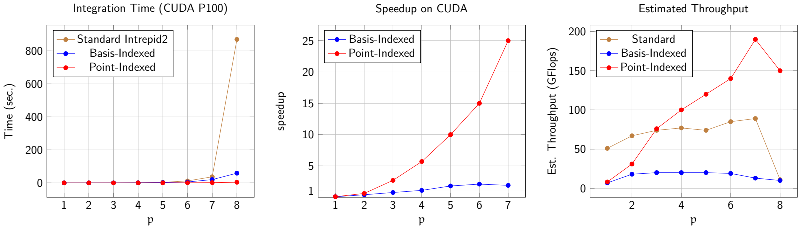


Figure: CUDA (P100) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Note: The $p = 8$ case has a dramatic slowdown for standard (for this case, the only workset size that ran to completion was 1); we exclude it from the speedup plot so as to not to throw off the scaling.

The 3D3V (3 space dimensions + 3 velocity dimensions) Vlasov-Poisson equations take the form:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{q}{m}\mathbf{E} \cdot \frac{\partial f}{\partial \mathbf{v}} = 0 \qquad (1)$$

$$\nabla \cdot \mathbf{E} = \frac{q}{\epsilon_0} \int f d^3 v \qquad (2)$$

$$\mathbf{E} + \nabla \phi = 0 \qquad (3)$$

Here, we have introduced a potential ϕ such that $\mathbf{E} = -\nabla \phi$ (convenient for BCs). We can simplify further by restricting to 1D1V:

$$\frac{\partial f}{\partial t} + v_x \frac{\partial f}{\partial x} + \frac{q}{m} E \cdot \frac{\partial f}{\partial v_x} = 0 \qquad (4)$$

$$\frac{\partial E}{\partial x} = \frac{q}{\epsilon_0} \int f d v_x \qquad (5)$$

$$E + \frac{\partial \phi}{\partial x} = 0 \qquad (6)$$

Vision for DPG Vlasov Solver

The goal: **flexible**, **robust**, **accurate** plasma physics solver for regimes that PIC does not address well.

Our approach: DPG for Vlasov.

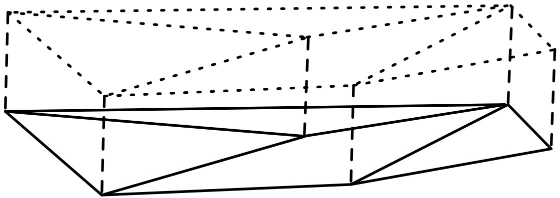
DPG has many attractive features:

- discrete stability is automatic
- almost total flexibility in solution basis (can go high-order)
- "minimum-residual method": solution error is minimized in an energy norm
- comes with a built-in error indicator: AMR is natural and robust

Vlasov in Camellia

Camellia is my Trilinos-based FEM library, with support for DPG + AMR.

- For Vlasov, we need hyper-dimensional meshes, up to 7D total.
- Key feature: allow **orthogonal extrusion** of any mesh in new dimensions.
  - Assume orthogonal: simplifies Jacobian computations, etc.
  - Do not assume uniform divisions: allow AMR in the new dimensions.



Camellia: Support for Structured Data

Sandia National Laboratories

- Camellia aims to be quite general, with support for arbitrary PDEs on unstructured grids.
- Working to add mechanisms to preserve structure for improved performance.
- A work in progress: foundation laid for e.g. using Intrepid2's sum factorization, but not yet implemented.
- Two examples: `Function` and `ExtrudedMeshTopology` classes.

Function Class and Structured Data

The `Function` class represents an arbitrary function, which may be mesh-dependent; subclasses include:

- `ConstantScalarFunction` - a constant scalar value.
- `SimpleSolutionFunction` - mesh-based solution for a specified variable.
- `Sin_ax` - sine of ax, where a is a constant.

`values()` method: accepts an object representing the computational/geometric context (e.g., which cells and points to compute values for), and outputs a multi-dimensional array with shape (C,P) (for scalar-valued functions).

Two key additions for structure preservation:

Function Class and Structured Data

The `Function` class represents an arbitrary function, which may be mesh-dependent; subclasses include:

- `ConstantScalarFunction` - a constant scalar value.
- `SimpleSolutionFunction` - mesh-based solution for a specified variable.
- `Sin_ax` - sine of ax, where a is a constant.

`values()` method: accepts an object representing the computational/geometric context (e.g., which cells and points to compute values for), and outputs a multi-dimensional array with shape (C,P) (for scalar-valued functions).

Two key additions for structure preservation:

- a version of `values()` that outputs to an `Intrepid2::Data` object (alongside methods that allow the subclass to specify the structure of the data)

Function Class and Structured Data

The `Function` class represents an arbitrary function, which may be mesh-dependent; subclasses include:

- `ConstantScalarFunction` - a constant scalar value.
- `SimpleSolutionFunction` - mesh-based solution for a specified variable.
- `Sin_ax` - sine of ax, where a is a constant.

`values()` method: accepts an object representing the computational/geometric context (e.g., which cells and points to compute values for), and outputs a multi-dimensional array with shape (C,P) (for scalar-valued functions).

Two key additions for structure preservation:

- a version of `values()` that outputs to an `Intrepid2::Data` object (alongside methods that allow the subclass to specify the structure of the data)
- a bit-packed member variable `_variesInDimension` that allows subclasses to specify in which spatial dimensions the `Function` varies

ExtrudedMeshTopology

Camellia's `MeshTopology` maintains the geometry of the mesh, including neighbor and parent-child relationships. (Contrast with `Mesh`, which additionally includes degrees of freedom for each cell.)

ExtrudedMeshTopology

Camellia's `MeshTopology` maintains the geometry of the mesh, including neighbor and parent-child relationships. (Contrast with `Mesh`, which additionally includes degrees of freedom for each cell.)

`ExtrudedMeshTopology` is a subclass of `MeshTopology` that supports orthogonal extrusion of a lower-dimensional `MeshTopology` in arbitrary dimensions.

ExtrudedMeshTopology

Camellia's `MeshTopology` maintains the geometry of the mesh, including neighbor and parent-child relationships. (Contrast with `Mesh`, which additionally includes degrees of freedom for each cell.)

`ExtrudedMeshTopology` is a subclass of `MeshTopology` that supports orthogonal extrusion of a lower-dimensional `MeshTopology` in arbitrary dimensions.

- constructor takes a root-level/unrefined `MeshTopology` and a set of coordinates in each extruded dimension.

ExtrudedMeshTopology

Camellia's `MeshTopology` maintains the geometry of the mesh, including neighbor and parent-child relationships. (Contrast with `Mesh`, which additionally includes degrees of freedom for each cell.)

`ExtrudedMeshTopology` is a subclass of `MeshTopology` that supports orthogonal extrusion of a lower-dimensional `MeshTopology` in arbitrary dimensions.

- constructor takes a root-level/unrefined `MeshTopology` and a set of coordinates in each extruded dimension.
- maintains a 1D `MeshTopology` object for each extrusion dimension, with the rule that this is at least as fine as any corresponding phase-space cell in that dimension.

ExtrudedMeshTopology

Camellia's `MeshTopology` maintains the geometry of the mesh, including neighbor and parent-child relationships. (Contrast with `Mesh`, which additionally includes degrees of freedom for each cell.)

`ExtrudedMeshTopology` is a subclass of `MeshTopology` that supports orthogonal extrusion of a lower-dimensional `MeshTopology` in arbitrary dimensions.

- constructor takes a root-level/unrefined `MeshTopology` and a set of coordinates in each extruded dimension.
- maintains a 1D `MeshTopology` object for each extrusion dimension, with the rule that this is at least as fine as any corresponding phase-space cell in that dimension.
- overrides `addCell()` method (a bottleneck for refinements), and maintains maps from phase-space cells to cells in each extrusion dimension (and back).

Challenges: Computational Cost and the Curse

The curse of dimensionality looms. We have three key mitigations:

1. Adaptive Mesh Refinement
   - Full support for isotropic h-adaptivity.
   - Anisotropic adaptivity: necessary for performance in high dimensions.
2. Underway: Hyperdimensional Serendipity bases[2]
3. Smart Assembly
   - Structure of Vlasov allows most terms to be integrated in lower dimensions, and multiplied by a pre-computed integral corresponding to remaining dimensions.
   - Not yet implemented.

[1] Serendipity basis support in Intrepid2; Trilinos master SHA1 22d0482, 7/7/22.

Space-Time Formulation: Vlasov

We may write the 1D1V Vlasov equation as:

$$\nabla_{xtv} \cdot \begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix} = 0.$$

Multiplying by test $w \in H^1$ and integrating by parts:

$$\langle \hat{t}_n, w \rangle - \left(\begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix}, \nabla_{xtv} w \right) = 0,$$

where formally

$$\hat{t}_n = \mathrm{tr} \left(\begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_t \\ n_v \end{bmatrix} \right).$$

We use the graph norm on the test space.

Space-Time Formulation: Poisson

Our space-time Poisson Formulation:

$$\begin{aligned}\langle \hat{V}_E, \tau\, n_x \rangle - (V_E, \partial_x \tau) + (E_x, \tau) &= 0 \\ \langle \hat{E}_x, q\, n_x \rangle - (E_x, \partial_x q) &= \left(\frac{\rho}{\epsilon_0}, q \right).\end{aligned}$$

Note that the traces $\hat{V}_E, \hat{E}_x$ are only defined at the spatial interfaces (those for which $n_x \neq 0$). Note also that ρ is two-dimensional: it varies in time as well as space. The usual situation is that BCs are imposed on $\hat{V}_E$ at the left and right boundaries; for the cold diode, we impose $\hat{V}_E = 0$ at each.

We use the graph norm on the test space.

Solution Strategy: Fixed Point Iteration

We use a fixed-point iteration with a set maximum number of iterations:

- up to 15 fixed-point iterations per solve, with early exit if the relative norm of the update falls below a tolerance (10^{-6}).
- Linear solves performed with Geometric-Multigrid-preconditioned conjugate gradient solver, tolerance between 10^{-7} and 10^{-9}.

The Cold Diode Problem

In the cold diode problem, a beam of electrons is emitted across a 1D anode-cathode gap, with an applied voltage across the gap.

- We have an exact solution due to Jaffé.
- EMPIRE-PIC has very accurate results for this problem.
- Tom Smith provided me the Python scripts used in EMPIRE's analysis; I've adapted these.

The Cold Diode Problem and Vlasov

Sandia National Laboratories

Some notes on our approach:

- We nondimensionalize for computations, such that $v^*_{\text{beam}} = 1$ and $t^*_{\text{final}} = 1$.
- We rescale on output for comparison to exact solution.
- Inflow BC: approximated with a Maxwellian with thermal velocity $\sigma = 0.025\, v_{\text{beam}}$.
- $\sigma > 0 \implies$ solving a slightly different problem; can expect some error due to that difference.
- Important to **resolve** the BC; we perform initial refinements to resolve to a given tolerance.
- We also introduce a linear temporal "ramp", phasing in the injection BC between $t = 0$ and $t = 0.25$.

Space-Time Results: Uniform Refinement Studies

Table: Relative L^2 errors

f order	Mesh Size	E err.	ϕ err.	n_e err.	v_x err.
0	$4 \times 40 \times 40$	2.458E-01	2.228E-01	2.276E-02	2.386E-02
0	$8 \times 80 \times 80$	1.228E-01	1.133E-01	1.130E-02	1.198E-02
0	$16 \times 160 \times 160$	6.137E-02	5.690E-02	5.630E-03	5.998E-03
1	$4 \times 20 \times 40$	2.481E-03	2.505E-02	2.446E-03	2.200E-03
1	$8 \times 40 \times 80$	7.065E-04	6.266E-03	6.660E-04	6.212E-04
1	$16 \times 80 \times 160$	3.924E-04	1.605E-03	3.641E-04	3.399E-04
2	$4 \times 10 \times 40$	5.021E-04	4.206E-04	2.586E-03	6.109E-04
2	$8 \times 20 \times 80$	3.660E-04	3.673E-04	4.753E-04	3.365E-04
2	$16 \times 40 \times 160$	3.618E-04	3.635E-04	4.016E-04	3.138E-04
3	$4 \times 5 \times 40$	6.151E-03	2.189E-03	2.614E-02	3.178E-03
3	$8 \times 10 \times 80$	3.624E-04	3.632E-04	4.126E-04	3.133E-04
3	$16 \times 20 \times 160$	3.619E-04	3.637E-04	3.353E-04	3.126E-04

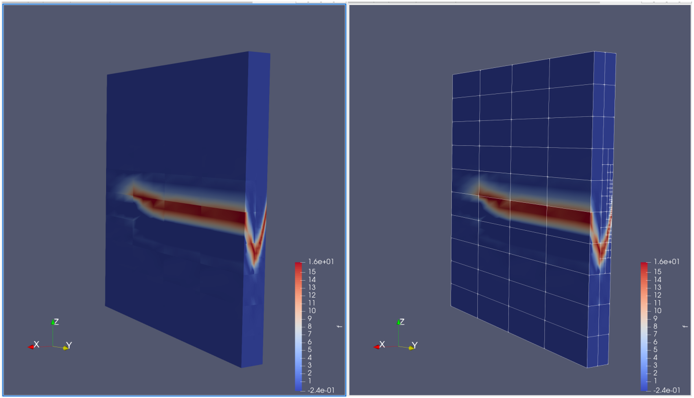
Uniform refinement study for space-time, for poly orders from 0 to 3. As with our finest time-marching solves, we see error of roughly 3×10^{-4} in each variable, due to the nonzero value for σ. Note that the second dimension is time; we use coarser discretizations in time for higher polynomial orders so that we have roughly the same number of temporal nodes as in the time-marching scheme.

Adaptive Space-Time Results

For this AMR run, we perform a set of initial refinements, driven by the error in the boundary condition, until that error is less than a specified tolerance in the relative L^2 norm on the boundary. In this run, we use the following setup:

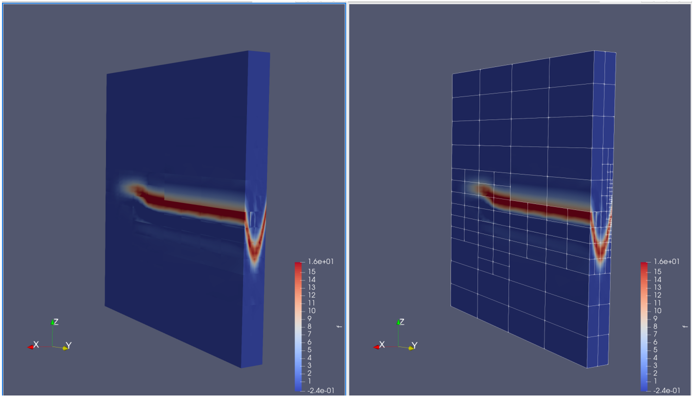
- coarse mesh: $2 \times 4 \times 10$ elements
- $\sigma = 0.025$
- BC tol: 10^{-5}
- quadratic field variables
- test space enrichment $\Delta p = 4$
- greedy refinement parameter $\theta = 0.2$

Adaptive Space-Time Results: Vlasov



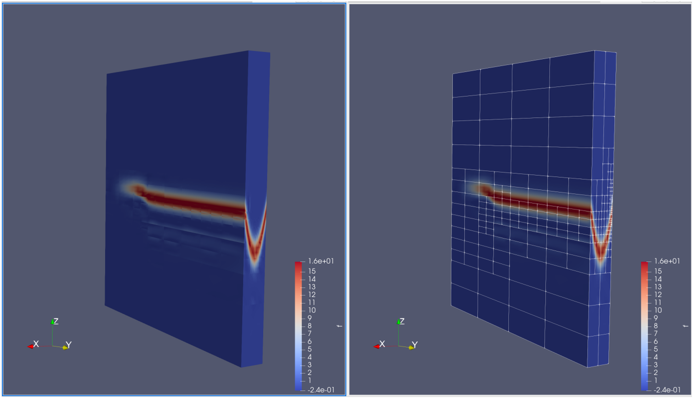
Vlasov solution for the cold diode problem, after 0 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



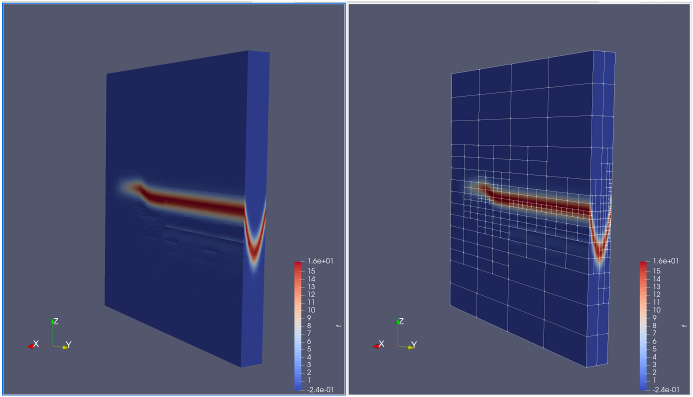
Vlasov solution for the the cold diode problem, after 1 energy-error refinement. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



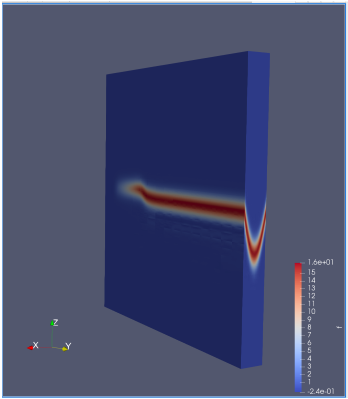
Vlasov solution for the cold diode problem, after 2 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

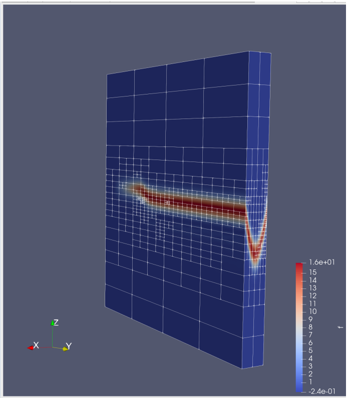
Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 3 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

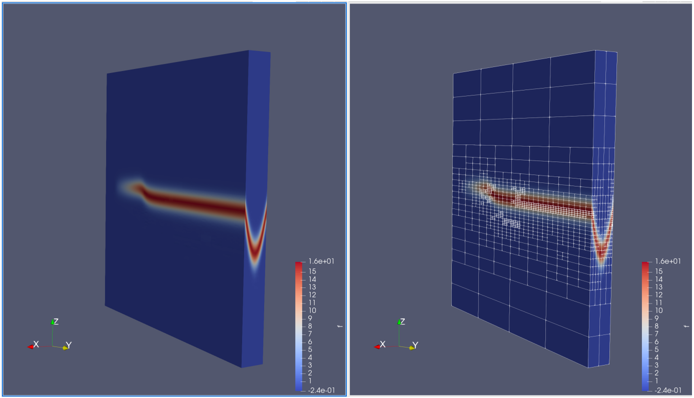
Adaptive Space-Time Results: Vlasov





Vlasov solution for the cold diode problem, after 4 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

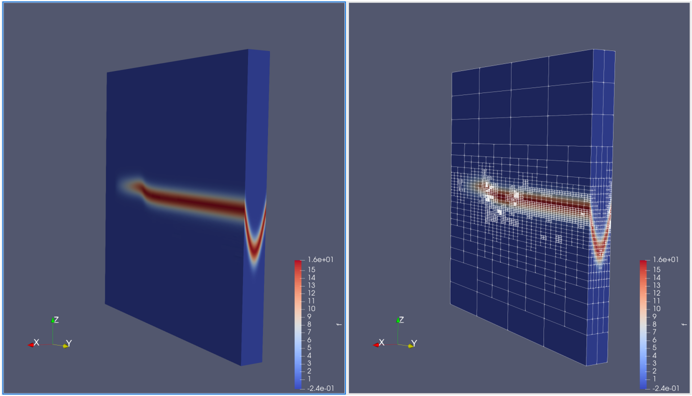
Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 5 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

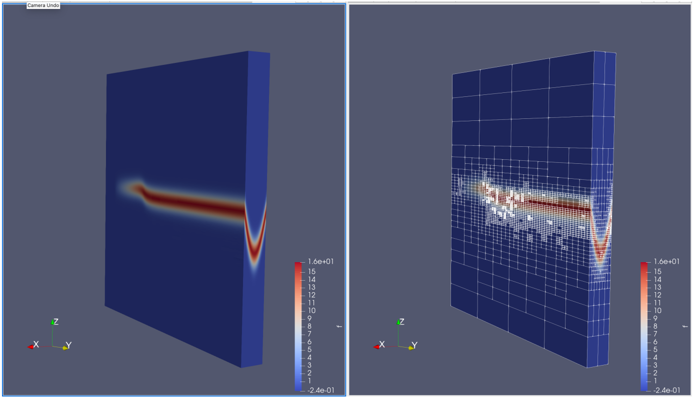
Adaptive Space-Time Results: Vlasov

Sandia National Laboratories



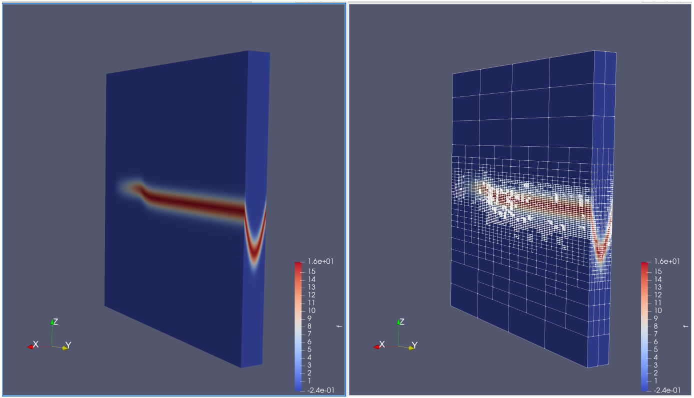
Vlasov solution for the cold diode problem, after 6 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 7 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 8 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Conclusion

- The more structured the problem, the greater gains we can achieve by taking advantage of that structure.
- Intrepid2 has a new, rich set of foundational classes for preserving structure through FEM computations.
- We have begun to take advantage of these in Camellia, particularly in the context of the Vlasov problem.

Thanks for your attention!