Implement High-Performance PDE Solvers for First-Principle Simulations by Using Python

Yung-Yu Chen
yyc@solvcon.net

Developer
SOLVCON Project

PyCon Japan 2012
September 16

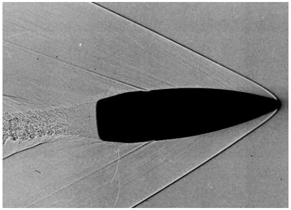
It's about SOLVCON

- A solver constructor.
  - Perform first-principle simulations for physical processes governed by partial differential equations (PDEs).
  - I focus on conservation laws at the time being.
  - Written in Python and with the performance hot-spot accelerated by C (or CUDA).
  - Address high-performance computing (HPC) by mesh-based, array-oriented programming.
- Slides: `http://j.mp/yyc_pyconjp2012`

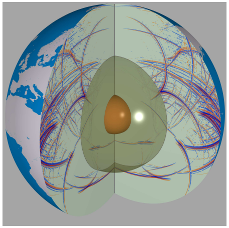
Outline

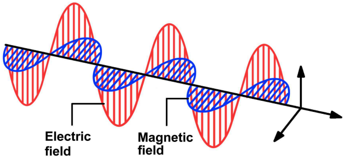
Conservation Laws Govern The World

Fluid mechanics, solid mechanics, electromagnetism, etc.


Supersonic flow.

Atmospheric flow.


Seismic waves.



Electromagnetic waves.

Experiments?

Hypersonic Wind Tunnel (Mach 5+)

Experiments, Seriously?

First-Order Hyperbolic PDEs

- The fore-mentioned problems share a common trait: Demanding time-accurate solutions of conservation laws.
- So this is what I want to solve:

$$\frac{\partial u_m}{\partial t} + \sum_{\mu=1}^{3} \frac{\partial f_m^{(\mu)}(\mathbf{u})}{\partial x_\mu} = s_m(\mathbf{u})$$

$$\Rightarrow \quad \boxed{\oint_{S(V)} \mathbf{h}_m(\mathbf{u}) \cdot d\mathbf{a} = \int_V s_m(\mathbf{u}) dv} \tag{1}$$

$$m = 1, \ldots, M.$$

Now a Programming Problem

Coding for first-principle simulators is difficult. Why?

1. Recall the math:

$$\frac{\partial u_m}{\partial t} + \sum_{\mu=1}^{3} \frac{\partial f_m^{(\mu)}(\mathbf{u})}{\partial x_\mu} = s_m(\mathbf{u}) \tag{1}$$

2. Various approaches to meshing and the associated data structures.
3. Parallel programming for HPC.
4. Data management and result analysis.

Why Write Code for Simulation?

Come on, there are many commercial codes. Is it really needed to develop yet another research code?

- Yes, because without the access to the source code, a computational scientist can never fully trust the code she uses.
- Commercial codes always implement the algorithms that can serve a larger crowd. We have to code the cutting-edge things by ourselves.

Open-Source Research Codes?

- Good codes like FEniCS and FiPy didn't focus on time-accurate solution of conservation laws with HPC.
    - FEniCS is based on finite-element method (FEM).
    - FiPy is based on projection method.
- Plenty of research codes are open-source, but
    - As the name suggests, many research codes lack the modern structure that allows reuse.
    - 10 incompatible versions of a code in a research group? Nobody likes it, but it's quite often.

And I Learned the CESE Method

- The space-time Conservation Element and Solution Element (CESE) method, developed by Chang at NASA Glenn.
    - Directly solves generic hyperbolic PDEs (Eq. (1)).
- Enable pluggable multi-physics in SOLVCON.
    - Compressible flows: $\mathbf{u} = (\rho, \rho v_1, \rho v_2, \rho v_3, \rho e)^t$.
    - Stress waves in solids: $\mathbf{u} = (v_1, v_2, v_3, \sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{23}, \sigma_{13}, \sigma_{12})^t$.
    - Electromagnetic waves: $\mathbf{u} = (E_1, E_2, E_3, B_1, B_2, B_3)^t$.
    - Acoustics, shallow-water, viscoelasticity, etc.

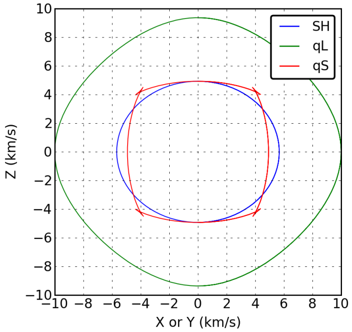
Chang (1995) Journal of Computational Physics 119(2):295–324
Chen (2011), Ph.D. Dissertation

Scope of SOLVCON

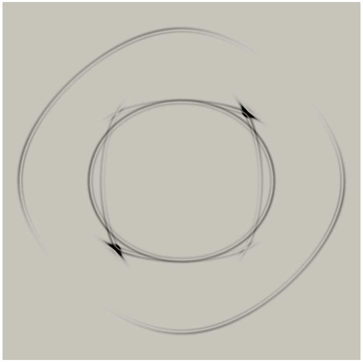
- A Python-based software framework for constructing time-accurate solvers of conservation laws for any physical processes.
- SOLVCON currently contains referential solvers that use the CESE method.
    - Unstructured meshes of mixed elements are used in two- or three-dimensional space.
    - Message-passing is built into the framework for parallel computing.
- A PDE-solving toolkit that can be embedded into other applications.

Application: Stress Wave in Solids

- Beryl: Anisotropic crystal of hexagonal symmetry.



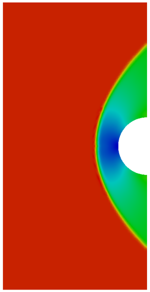
Exact solution of group velocity

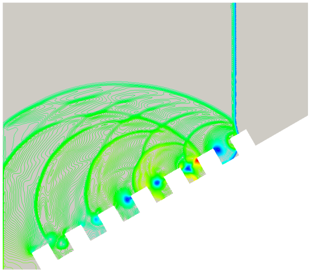


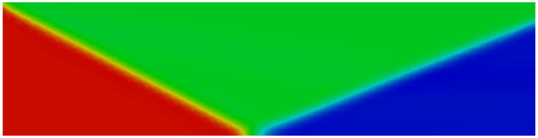
Simulated result

Yang et al. (2011) J. Vib. Acoust. 133(2): 021001

Application: Supersonic Flows



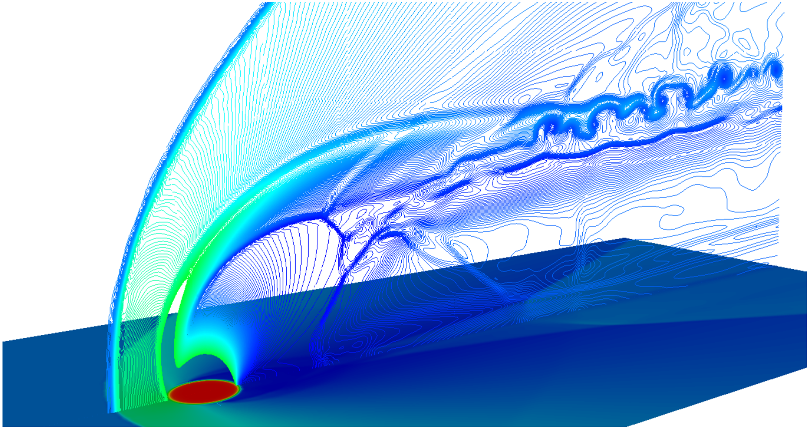




- 2D cases:
  - Flow over a cylinder.
  - Oblique shock by a ramp.
  - Moving shock climbing a ramp.
  - Moving shock diffraction by a step.
  - Moving shock past dust layer.
  - Reflection of oblique shock.
  - Implosion.
- 3D cases:
  - Sod's shock tube.
  - Flow over sphere.
  - Jet in cross flow.

Jet in Supersonic Cross Flow

66 million elements are used in the simulation.



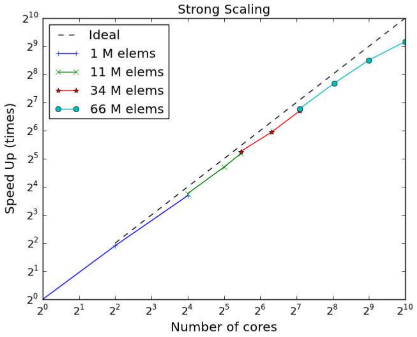
Density

Runtime Benchmark

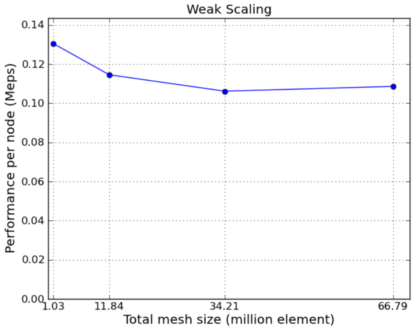
- Benchmark with hybrid parallel computing.
  - MPI across nodes; pthread within a node.
  - Run on Glenn@OSC: 4 cores/node with 10Gbps IB.
- Performance in million elements per second (Meps).

Number of cells (M)		1	11	34	66
Perf. (Meps)	1 core	0.035	–	–	–
	4 cores	0.13	–	–	–
	16 cores	0.45	0.47	–	–
	32 cores	–	0.91	–	–
	44 cores	–	1.26	1.33	–
	80 cores	–	–	2.16	–
	136 cores	–	–	3.61	3.82
	264 cores	–	–	–	7.17
	512 cores	–	–	–	12.7
	1024 cores	–	–	–	20.0

Scaling



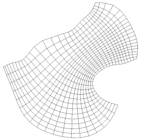
Fix Overall Mesh Size



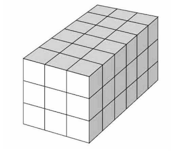
Fix Per-Node Mesh Size

Mesh-Based Programming

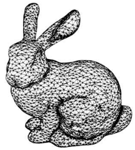
Curvilinear



Cartesian



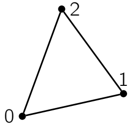
Unstructured



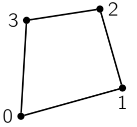
- Two ways to discretize space:
    - Structured mesh: The connectivity of elements is structured. Cartesian is a special case.
    - Unstructured mesh: The connectivity of elements is irregular.
- SOLVCON uses unstructured mesh:
    - The data structure defines **connectivity** and **geometry**.
    - Code is based on the data structure.

Unstructured Meshes of Mixed Elements

- Two-dimensional elements:

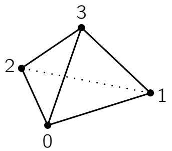


triangle

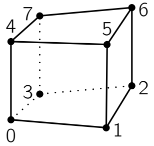


quadrilateral

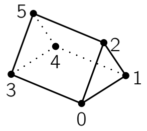
- Three-dimensional elements:



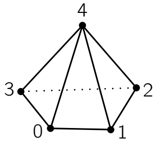
tetrahedron



hexahedron

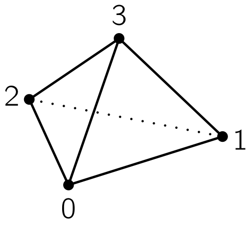


prism



pyramid

Look-Up Tables for Meshes

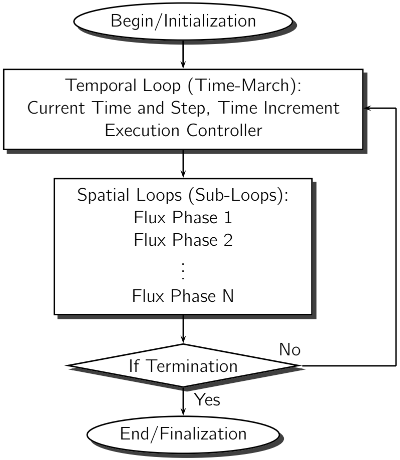


tetrahedron

- Consider cell-based solvers (value is stored at cell centers):
    - 4 nodes for the cell: 0, 1, 2, 3.
    - 4 faces for the cell: 012, 013, 123, 203.
- Tables to connect all cells:
    - Global index of nodes (`ndcrd`).
    - Nodes in each cell (`clnds`).
    - Faces in each cell (`clfcs`).
    - Nodes in each face (`fcnds`).
    - Cells connected by each face (`fccls`).

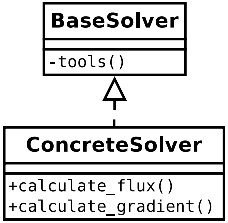
Two-Loop Structure of PDE Solvers

- The basic execution flow of SOLVCON:
  - Temporal loop for temporal (or pseudo-temporal) integration.
  - Spatial loops iterate over elements.
- The structure is general to all PDE solvers.



Solver Kernel for Spatial Loops

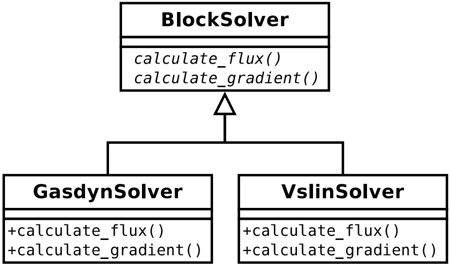
- A solver kernel is a Python class.
- The base class implements utility methods for spatial loops.
- The algorithms directly work with the mesh look-up tables.
- The concrete solver implements real algorithms, in C, or other fast languages.



Inheritance for Multi-Physics

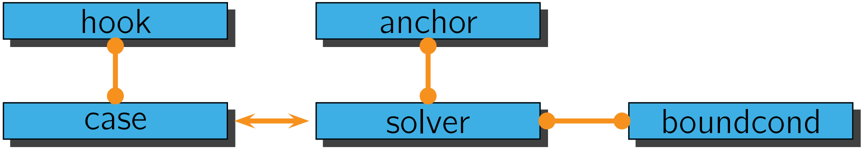
- For a multi-physics algorithm, like the CESE method, a class hierarchy can be designed to host multiple physical processes.

- The physical processes are segregated.



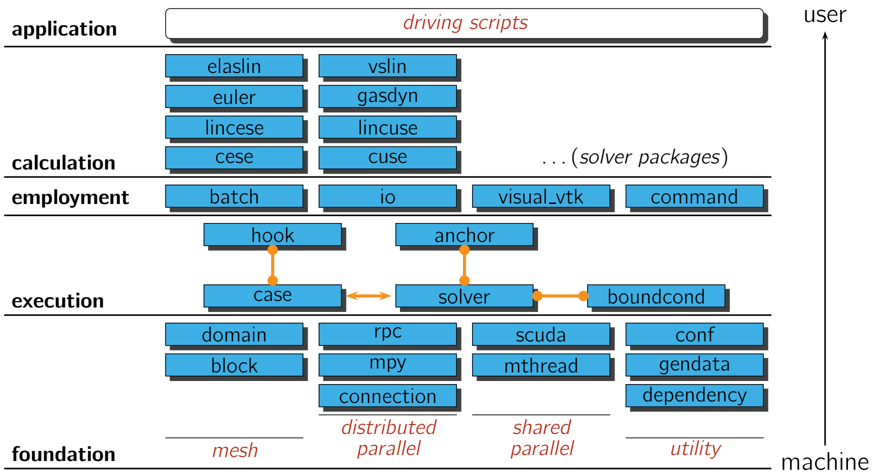
Temporal Loop and Call-Back

- A standalone class hierarchy (`Case`) is designed to host the temporal loop.

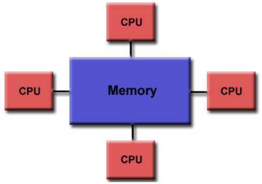


- `Hook` and `Anchor` are call-back objects for `Case` and `Solver`, respectively.
  - Supplement of main algorithms.
  - Lazy initialization.
  - Facilitating parallel computing and in-situ analysis.

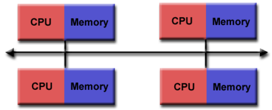
Overall Design of SOLVCON



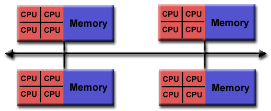
Two Types of Parallel Computing



+



=



distributed (cluster)

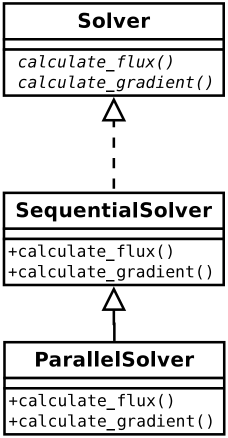
hybrid

shared (multi-/many-core)

- Simultaneously use shared-memory and distributed-memory parallel computing (DMPC & DMPC, respectively).
  - Main difference: Addressing space.
- Inter-process communication is needed.
  - DMPC is much more complex than SMPC.
  - DMPC determines the scalability.
  - MapReduce is unsuitable.

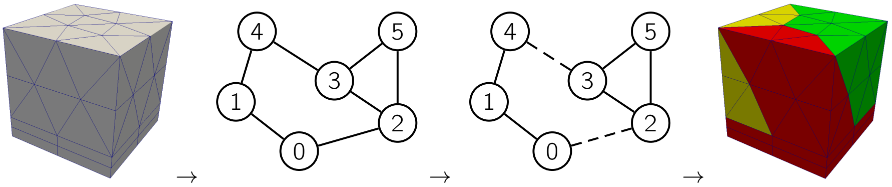
Extending Solver Kernel for SMPC

- A `Solver` class can be extended to use shared-memory parallel computing.
- Only the spatial loops are modified.
- Can use pthread, OpenMP, CUDA, OpenCL, etc.



Domain Decomposition for DMPC

- Before computation: Domain decomposition.
  - Use connectivity data to build the graph of cells.
  - Partition the graph by calling METIS or SCOTCH library.
  - Use the partitioned graph to decompose mesh data.

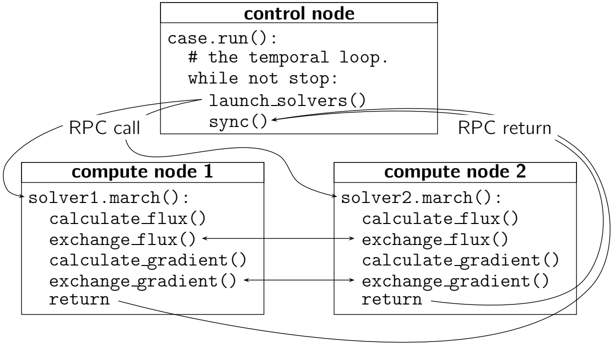


- During computation: Exchange data of the cells on the interface of different sub-domains.
  - Use MPI to communicate among sub-domains.

Solver Kernels Need Not Know DMPC

- DMPC is in SOLVCON framework.
- SMPC is in solver kernels.

DMPC Execution Flow in SOLVCON



control node

```
case.run():
  # the temporal loop.
  while not stop:
    launch_solvers()
    sync()
```

compute node 1

```
solver1.march():
  calculate_flux()
  exchange_flux()
  calculate_gradient()
  exchange_gradient()
  return
```

compute node 2

```
solver2.march():
  calculate_flux()
  exchange_flux()
  calculate_gradient()
  exchange_gradient()
  return
```

- When developing solver kernels, we do not need to worry about the complexity of DMPC.
- Hybrid parallelism is achieved by the segregation.

Post-Processing is Bottleneck

- High-resolution simulations generate a lot of data:
  - For 50 million element mesh, the data for one scalar (single-precision) are 200 MB.
  - A typical run has at least 10,000 time steps.
  - Transient analysis: $10,000 \times 200\,\text{MB} = 2\,\text{TB}$.
  - $\rho, p, T, \vec{v}, \vec{\omega}$ for CFD: $2\,\text{TB} \times 9 = 18\,\text{TB}$.
- Workaround: Reducing output frequency.
  - Every 100 time steps: 160 GB.
- Post-processing the solutions is painfully time-consuming:
  - The large data are usually processed by using a single workstation.
  - Turnaround time could be in months.

Solutions in SOLVCON

- Parallel I/O.
    - Each sub-domain outputs its own solutions.
    - It is used with parallel post-processing.
- In situ visualization.
    - Visualization is being done on the fly with the simulation.
    - Everything happens in memory.
    - Output only graphic files, which are much smaller than the full solution field.
- Parallel I/O and in situ visualization are complementary to each other.

Driving Scripts

- Driving scripts manage simulations for SOLVCON.
  - A driving script must create a `Case` object and call its (i) `init()`, (ii) `run()`, and (iii) `cleanup()` methods.
  - No input file is needed.
- Driving scripts can specify logic to the simulations in addition to parameters.
  - Anything higher than the foundation layer (the lowest layer) can be replaced by code written in driving scripts.
  - Including but not limited to `Case`, `Solver`, `BC` classes, `Hook`, and `Anchor` classes.

Three-Dimensional Sod's Shock Tube

- `$SCROOT/examples/gasdyn/tube`.
  - The one-dimensional version is a standard test case for gas-dynamics codes.
  - This version demonstrates the three-dimensional capability of SOLVCON.
  - You also need CUBIT to generate the mesh.
- Let's demo:
  - `cd $SCROOT/examples/gasdyn/tube`
  - `./go`

General Structure of a Driving Script

- The driving script for the shock tube problem:

```
from solvcon.kerpak import gasdyn
class DiaphragmIAnchor(gasdyn.GasdynIAnchor):
    ...
def tube_base(casename=None,
    gamma=None, rho1=None, p1=None, rho2=None, p2=None,
    psteps=None, ssteps=None, **kw
):
    ...
if __name__ == '__main__':
    cse = tube_base('tube_20', use_incenter=True,
        gamma=1.4, rho1=1.0, p1=1.0, rho2=0.125, p2=0.25,
        time_increment=1.8e-3, steps_run=400, ssteps=40, psteps=1)
    cse.init()
    cse.run()
    cse.cleanup()
```

- There's another "delayed" mode for driving scripts. See examples/gasdyn/blnt.

Use Anchor for Initial Condition

- The `provide()` method is invoked before the temporal loop:

```python
class DiaphragmIAnchor(gasdyn.GasdynIAnchor):
    ...
    def provide(self):
        super(DiaphragmIAnchor, self).provide()
        gamma = self.gamma
        svr = self.svr
        svr.soln[:,0].fill(self.rho1)
        svr.soln[:,1].fill(0.0)
        svr.soln[:,2].fill(0.0)
        if svr.ndim == 3:
            svr.soln[:,3].fill(0.0)
        svr.soln[:,svr.ndim+1].fill(self.p1/(gamma-1))
        # set.
        slct = svr.clcnd[:,0] > 0.0
        svr.soln[slct,0] = self.rho2
        svr.soln[slct,svr.ndim+1] = self.p2
        # update.
        svr.sol[:] = svr.soln[:]
```

Setup the Flow of the Simulation

- A Case object is created for the simulation:

```
def tube_base(...):
    ...
    # set up case.
    basedir = os.path.abspath(os.path.join(os.getcwd(), 'result'))
    cse = gasdyn.GasdynCase(basedir=basedir, rootdir=env.projdir,
        basefn=casename, mesher=mesher, bcmap=bcmap, **kw)
    # informative.
    cse.runhooks.append(hook.BlockInfoHook)
    ...
    # initializer.
    ...
    cse.runhooks.append(DiaphragmIAnchor,
        gamma=gamma, rho1=rho1, p1=p1, rho2=rho2, p2=p2)
    # post processing.
    ...
    cse.runhooks.append(gasdyn.GasdynOAnchor, rsteps=ssteps)
    ...
    return cse
```

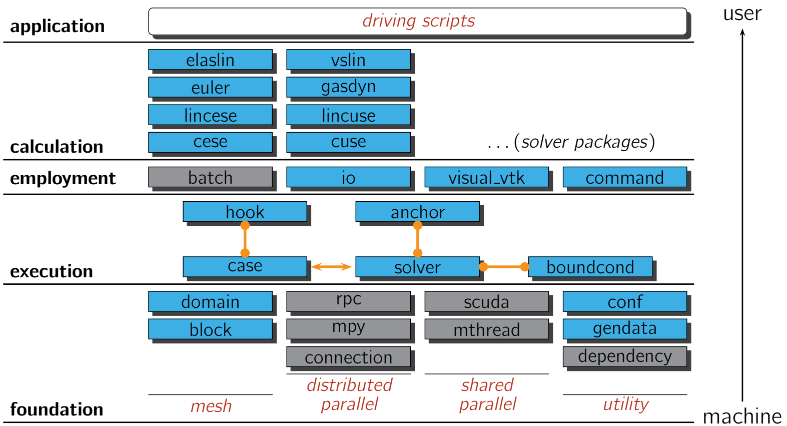
The Results

- Run the simulation case:

```
if __name__ == '__main__':
    cse = tube_base('tube_20', use_incenter=True,
        gamma=1.4, rho1=1.0, p1=1.0, rho2=0.125, p2=0.25,
        time_increment=1.8e-3, steps_run=400, ssteps=40, psteps=1)
    cse.init()
    cse.run()
    cse.cleanup()
```

- Let's launch ParaView to visualize and render the calculated results.

A Planned Renovation of SOLVCON



- Simply the architecture: Rely more on mpi4py, OpenMP, etc.
- Use Cython instead of ctypes for maintainability.

Conclusions

- We've used Python to construct a high-performance system for solving conservation laws or PDEs.
    - It's multi-physics: Fluid mechanics and solid mechanics (and electromagnetics in the future).
    - It has a clear architecture.
    - Parallel computing is made easy.
- It's very productive.
- And ...

Python Rocks

Yes, it is!

Thank You