Enjoy Unimportant Things

Yung-Yu Chen

2016/4/15

Jobspace

Why We Hold Jobspace

- A community of
  - Diversity. We've prepared frontend technology and knowledge discovery after this kick-off meetup. More topics are being conceived. Attendees can enjoy a broad range of topics.
  - Knowledge. We invite experienced speakers and work together for a quality talk in each meetup.
  - Interaction. We make the surroundings as comfortable as possible to bolster vibrant discussions. We use the talk to gather people of matching interests. We expect attendees to get as much from each other as from the speaker.
- We invest time and money for such a community. Attendees only need to come and enjoy.
  - We listen to you and make things happen. You don't need to be a volunteer.

About This Talk

- We will start with a little bit about Science
  - Etymology: Scientia (Latin)
    - Knowledge based on demonstrable and reproducible data (Merriam-Webster Dictionary)
  - John Ruskin, *The Eagle's Nest* (1872)
    - "In science you must not talk before you know. In art you must not talk before you do. In literature you must not talk before you think.
  - To know, is the ultimate goal.
- Reality: Human works barely work.
  - #gotofail

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
  goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
  goto fail;
```

- To reach the truth, defy illusion, we have to work hard to ensure we aren't getting things wrong.

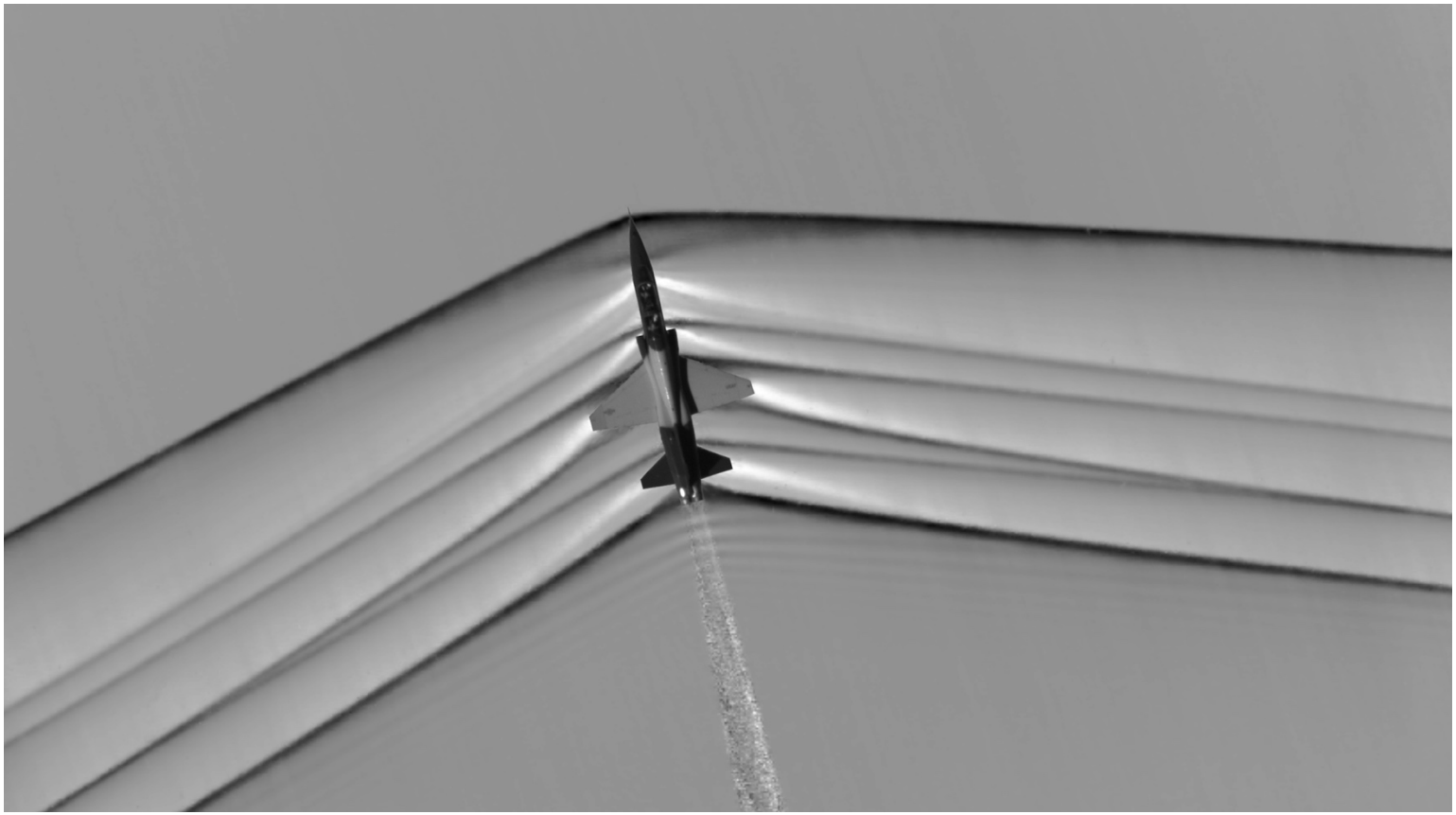
Computational Science for Fluid Mechanics

Computational Fluid Dynamics (CFD) in the Context of Aerospace Applications

To See before to Know

- Image of density gradient: Schlieren photography
  - Invented 150 years ago in 1864 by August Teopler
  - Recently developed to be applied outside laboratory.

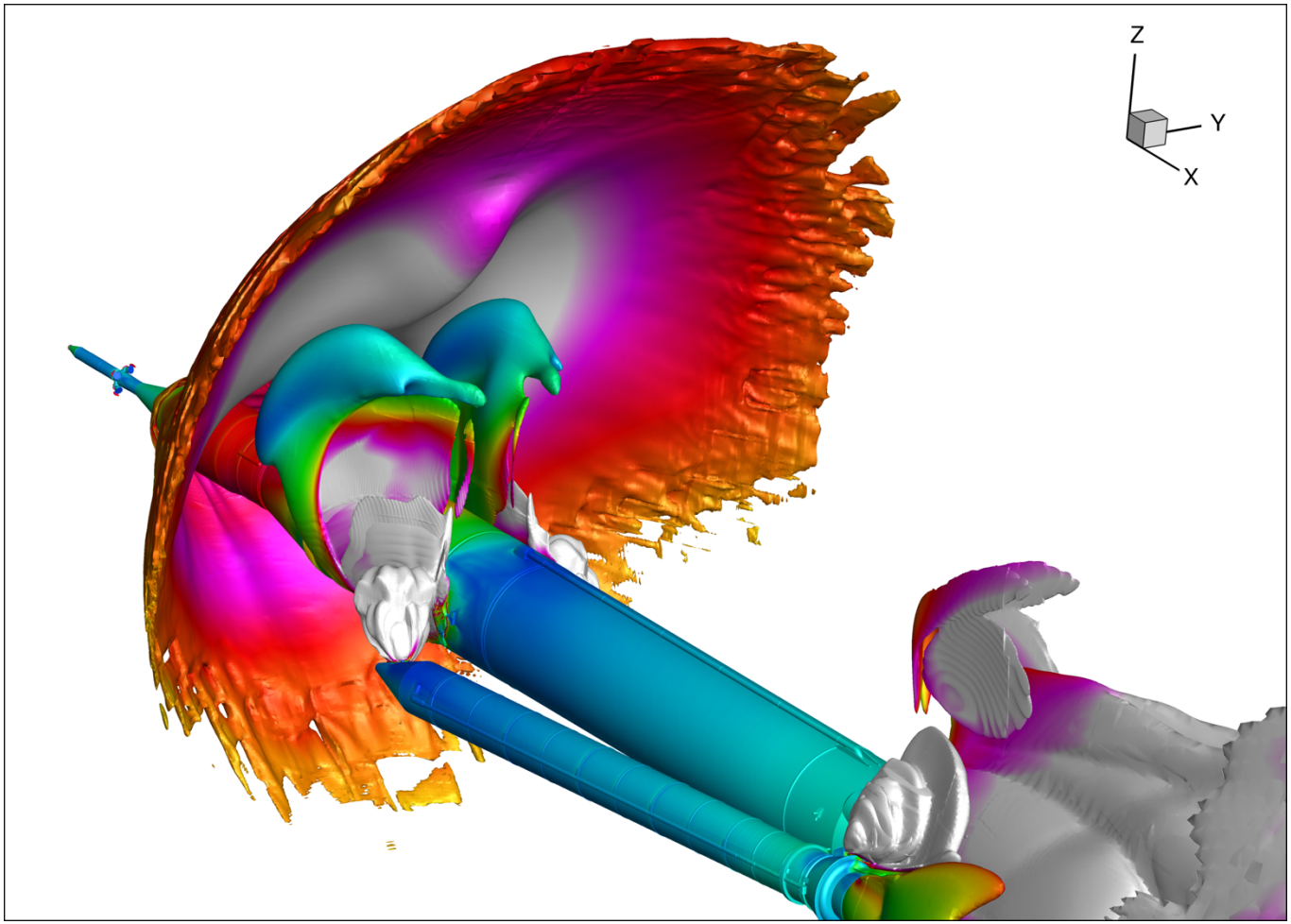
Shock waves imaged with a full-scale supersonic vehicle in flight.



http://go.nasa.gov/230fQOa

To See More to Know More

- CFD (computational fluid dynamics) gives you much more.

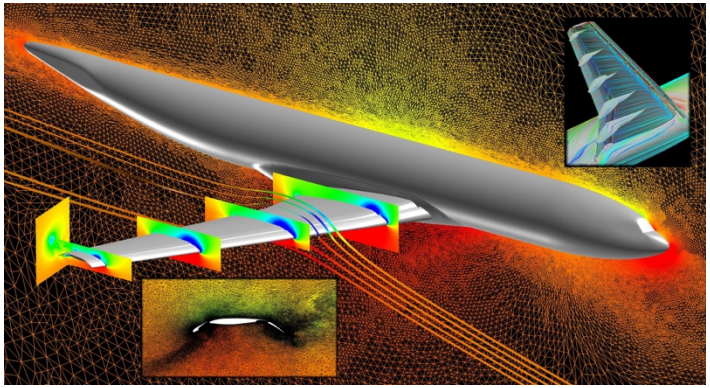


Pressure plots of shocks and the plumes for booster separation of NASA's Space Launch System (SLS).

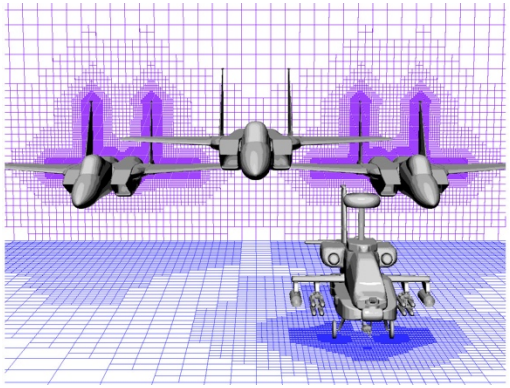
http://go.nasa.gov/1Vl9B1p

What a Solver Needs to Do

- Data input.
  - Spatial geometry (meshes): Unstructured mesh or structured mesh.
  - Parameters. Sometimes logic.
- Execute numerical method for the governing equations.
  - Conservation laws, constitutive models, and process models.
  - The distributed-memory parallel programming is a must for high-performance computing (HPC).
- Data output.
  - Big volume of storage: With the resolution of 50 million elements and a minimal of 5 variables on each element, each time step takes 1GB space.
  - A normal run of 10,000 steps can eat 10TB if we do it naively.



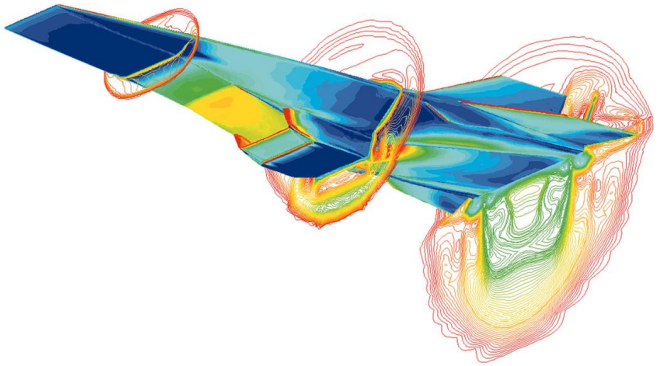
http://go.nasa.gov/1S36OaS



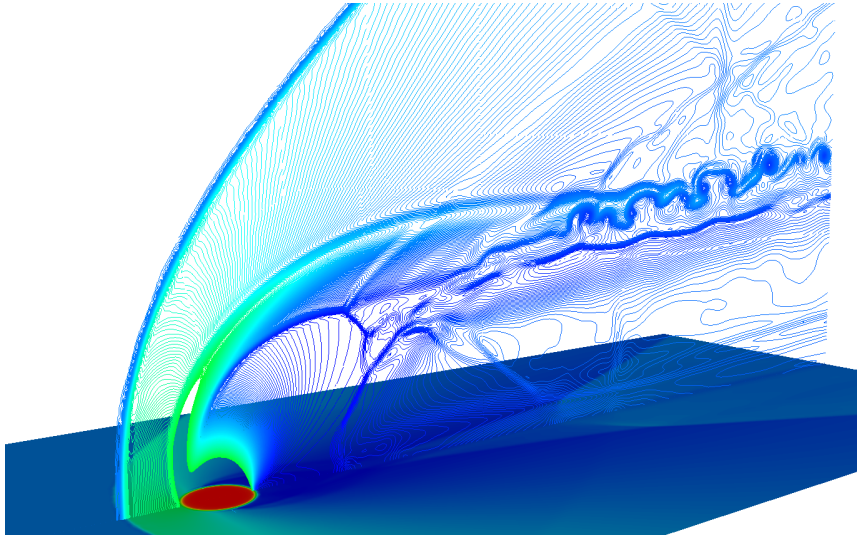
http://go.nasa.gov/23ldny1

What Else to be Done

- Visualization and analysis.
  - Conventionally scientists use a post-processer to visualize the simulations.
  - The huge amount of data pose difficulty to this approach.



http://go.nasa.gov/232db6E



- Scripting and automation.
- Portability and then reproducibility.
  - Using hundreds to thousands of slave nodes is normal.
  - Easy to get entangled with environment.

The Equations to Solve

- Consider the inviscid Euler equations for supersonic flows.
- 5 independent variables: $\rho, \rho v_1, \rho v_2, \rho v_3, \rho\left(e + \frac{v_k^2}{2}\right)$

Employ Einstein's summation convention for the Cartesian tensors.

- Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v_j}{\partial x_j} = 0$$

- Conservation of momentum:

$$\frac{\partial \rho v_i}{\partial t} + \frac{\partial \rho v_i v_j}{\partial x_j} = \frac{\partial p}{\partial x_j}$$

- Conservation of energy:

$$\frac{\partial}{\partial t}\left[\rho\left(e + \frac{v_k^2}{2}\right)\right] + \frac{\partial}{\partial x_j}\left[\rho\left(e + \frac{v_k^2}{2}\right) v_j\right] = -\frac{\partial p v_j}{\partial x_j}$$

- Equation of state:

$$p = \rho R T \quad \Rightarrow \quad e = \frac{1}{\gamma - 1}\frac{p}{\rho}$$

- We end up with a system of <u>hyperbolic partial differential equations</u> (PDEs), also known as <u>conservation laws</u>.

How to Solve

- The embedded nonlinearity results into shocks. Continuous solutions of the differential equations do not exist.
  - "The shocks manifest themselves mathematically as surfaces on which density ... have discontinuity ... the shock surfaces are in motion relative to the network of points in space-time used for the numerical work, and the differential equations and boundary conditions are non-linear." Von Neumann and Richtmyer (1950) *Journal of Applied Physics* 21(3):232–237
- The space-time conservation element and solution element (CESE) method solve the integral equations.

$$\frac{\partial u_m}{\partial t} + \sum_{k=1}^{3} \frac{\partial f_m^{(\kappa)}(\mathbf{u})}{\partial x_k} = 0 \;\Rightarrow\; \oint_{\partial V} \mathbf{h}_m(\mathbf{u}) \cdot \mathrm{d}\mathbf{a},\; m = 1, \ldots, 5$$

Space-time coordinate: $\begin{pmatrix} x_1 & x_2 & x_3 & t \end{pmatrix}^t$

Space-time flux: $\mathbf{h}_m = \begin{pmatrix} f_m^{(1)} & f_m^{(2)} & f_m^{(3)} & u_m \end{pmatrix}^t$

Chang (1995) *Journal of Computational Physics* 119(2):295–324

Code up Solvers

- We must hide the complexity of mathematics, or the code will be messy.
  - Luckily, the CESE method can wrap the physical meanings in the Jacobian matrices of the quasi-linear form of the differential equations.

$$\frac{\partial \mathbf{u}}{\partial t} + \sum_{k=1}^{3} \frac{\partial \mathbf{f}^{(k)}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial x_k} = 0$$

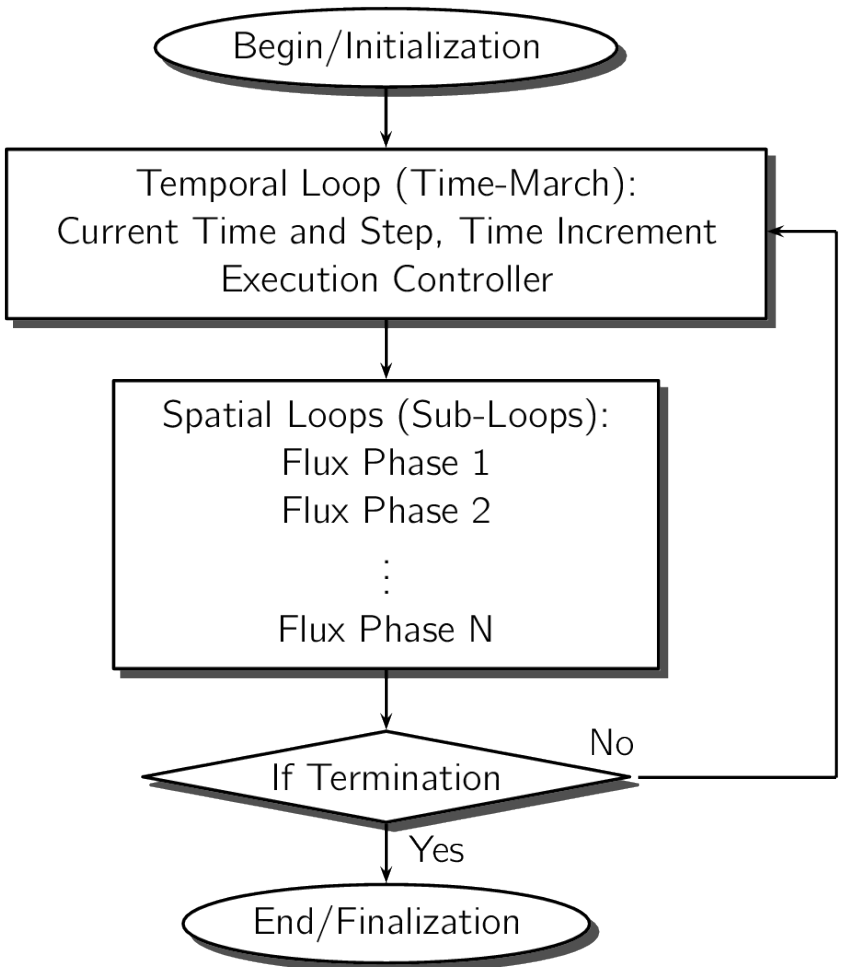
The Jacobian $\frac{\partial \mathbf{f}^{(k)}}{\partial \mathbf{u}} = \mathrm{A}^{(k)}(\mathbf{u})$

  - This can handle any physical process governed by hyperbolic PDEs (aka conservation lays).
- I have incorporated solvers of gas dynamics, stress waves, and acoustics into a system: SOLVCON.
- The back bone is a nested-loop structure, which should be employed by all such hyperbolic PDE solvers.
  - But in most codes this structure isn't resolved. It is very difficult to maintain the codes.

Build a Framework in Python

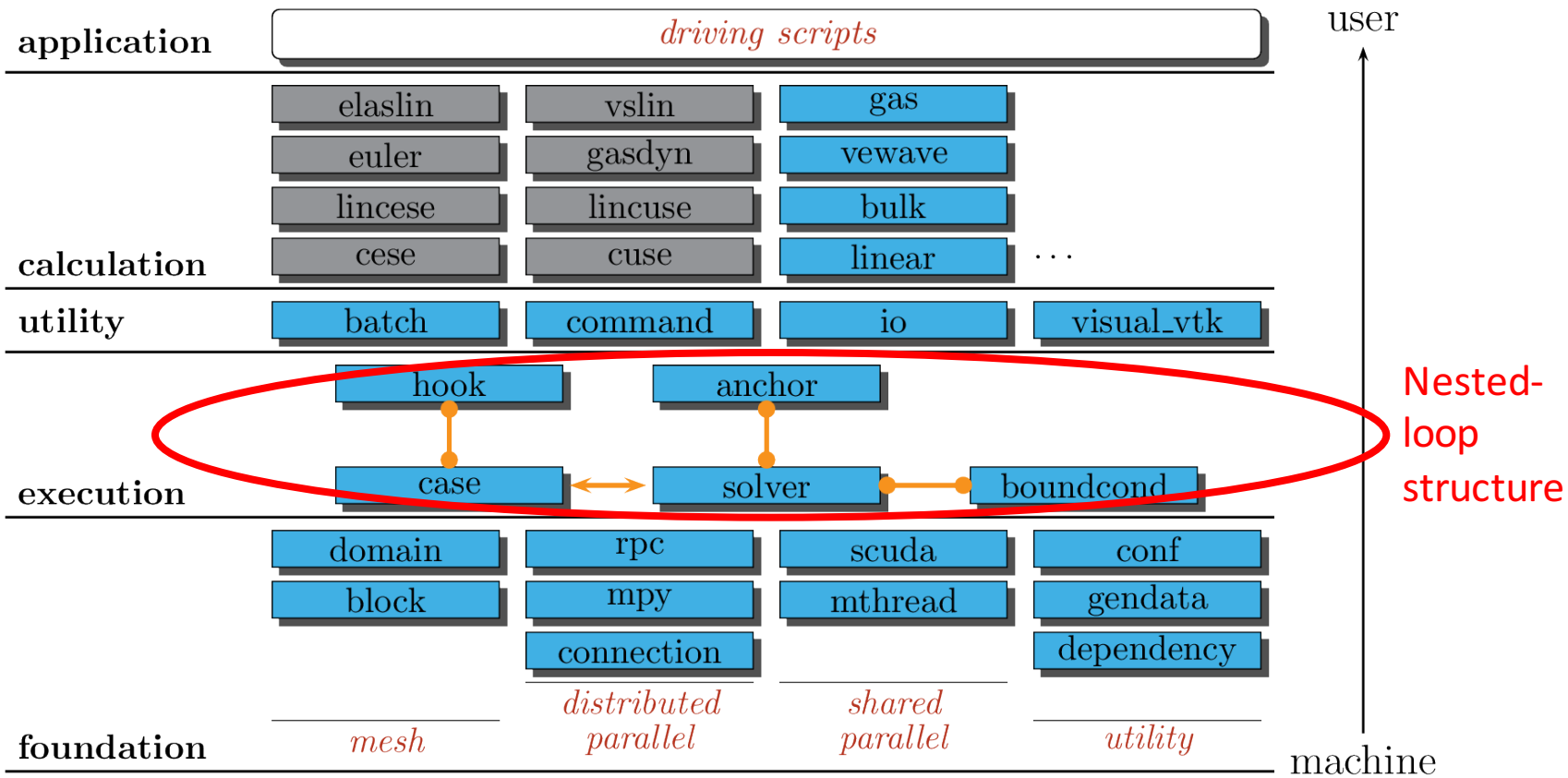
The Nested-Loop Structure

- An outer loop controls time-marching: how the simulation steps forward in time.
- Inside the temporal loop, the solver sweeps over the mesh data structure in multiple phases. I call each of the phases a spatial loop.
- This is incarnated in a package called SOLVCON.



This is How Python Comes into Play

- Designing a framework calls for high-level concepts, which are easily materialized in high-level languages.
- The current architecture →
  - Good structure.
  - Utmost performance.



Segregation, Segregation, Segregation

```
1  void
2  #if NDIM == 3
3  sc_gas_calc_dsoln_3d
4  #else
5  sc_gas_calc_dsoln_2d
6  #endif
```

- This is part of a number-crunching kernel
- It is well-organized
- Imagine 3000 lines like this with little convention

```
void
#if NDIM == 3
sc_gas_calc_dsoln_3d
#else
sc_gas_calc_dsoln_2d
#endif
(sc_mesh_t *msd, sc_gas_algorithm_t *alg) {
    int clnfc;

    int *pcltpn;
    int *pclfcs, *pfccls;
    double *pcecnd, *picecnd, *pjcecnd;
    double *pisoln, *pjsol, *pjsoln, *pdsol, *pdsoln;
    double *pjsolt;
    // scalars.
    double hdt;
    double tau, vob, voc, wgt, ofg1, sgm0;
    double grd0, grd1, grd2;
    // arrays.
    double xps[CLMFC][NDIM], dsp[CLMFC][NDIM];
    double crd[NDIM], cnd[NDIM], cndge[NDIM], sft[NDIM];
    double dst[NDIM][NDIM];
    double dnv[NDIM][NDIM];
    double udf[NEQ][NDIM];
    double gfd[MFGE][NEQ][NDIM];
    double dlt[MFGE][NEQ];
    double dla[NEQ];
    // interators.
    int icl, ifl, ifl1, ifc, jcl, ieq, ivx;
    int ig0, ig1, ig, ifg;
    hdt = alg->time_increment * 0.5;
    #pragma omp parallel for private(clnfc, pcltpn, pclfcs, \
    pfccls, pcecnd, picecnd, pjcecnd, \
    pisoln, pjsol, pjsoln, pdsol, pdsoln, pjsolt, \
    tau, vob, voc, wgt, ofg1, sgm0, \
    grd0, grd1, grd2, \
    xps, dsp, crd, cnd, cndge, sft, dst, dnv, udf, gfd, dlt, dla, \
    icl, ifl, ifl1, ifc, jcl, \
    ieq, ivx, ig0, ig1, ig, ifg) \
    firstprivate(hdt)
    for (icl=0; icl<msd->ncell; icl++) {
        pcltpn = msd->cltpn + icl;   // 1 flops.
        ig0 = ggerng[pcltpn[0]][0];
        ig1 = ggerng[pcltpn[0]][1];
        ofg1 = 1.0/(ig1-ig0);
        pclfcs = msd->clfcs + icl*(CLMFC+1);

        // determine sigma0 and tau.
        sgm0 = alg->sigma0 / fabs(alg->cfl[icl]);   // 3 flops.
        tau = alg->taumin + fabs(alg->cfl[icl]) * alg->tauscale;

        // calculate the vertices of GGE with the tau parameter.
        pclfcs = msd->clfcs + icl*(CLMFC+1);
        picecnd = alg->cecnd + icl*(CLMFC+1)*NDIM;
        pcecnd = picecnd;
        clnfc = pclfcs[0];
        for (ifl=1; ifl<=clnfc; ifl++) {    // clnfc*(16+8) flops.
            ifl1 = ifl - 1;
            ifc = pclfcs[ifl];
            pfccls = msd->fccls + ifc*FCREL;
            jcl = pfccls[0] + pfccls[1] - icl;
            pjcecnd = alg->cecnd + jcl*(CLMFC+1)*NDIM;
            pcecnd += NDIM;
            // location of P/M points and displacement to neighboring solution
            // points.
            sft[0] = (picecnd[0] + pjcecnd[0])/2;
            sft[0] += alg->cnbfac*(pcecnd[0] - sft[0]);
            xps[ifl1][0] = (pjcecnd[0] - sft[0])*tau + sft[0];
            dsp[ifl1][0] = xps[ifl1][0] - pjcecnd[0];
            sft[1] = (picecnd[1] + pjcecnd[1])/2;
            sft[1] += alg->cnbfac*(pcecnd[1] - sft[1]);
            xps[ifl1][1] = (pjcecnd[1] - sft[1])*tau + sft[1];
            dsp[ifl1][1] = xps[ifl1][1] - pjcecnd[1];
#if NDIM == 3
            sft[2] = (picecnd[2] + pjcecnd[2])/2;
            sft[2] += alg->cnbfac*(pcecnd[2] - sft[2]);
            xps[ifl1][2] = (pjcecnd[2] - sft[2])*tau + sft[2];
            dsp[ifl1][2] = xps[ifl1][2] - pjcecnd[2];
#endif
        };
```

```
        // calculate average point.
        crd[0] = crd[1] = 0.0;
#if NDIM == 3
        crd[2] = 0.0;
#endif
        for (ifl=0; ifl<clnfc; ifl++) { // clnfc*(2+1) flops.
            crd[0] += xps[ifl][0];
            crd[1] += xps[ifl][1];
#if NDIM == 3
            crd[2] += xps[ifl][2];
#endif
        };
        crd[0] /= clnfc;    // 2+1 flops.
        crd[1] /= clnfc;
#if NDIM == 3
        crd[2] /= clnfc;
#endif
        // calculate GGE centroid.
        voc = cndge[0] = cndge[1] = 0.0;
#if NDIM == 3
        cndge[2] = 0.0;
#endif
        for (ig=ig0; ig<ig1; ig++) {
            cnd[0] = crd[0];
            cnd[1] = crd[1];
#if NDIM == 3
            cnd[2] = crd[2];
#endif
            for (ivx=0; ivx<NDIM; ivx++) {  // MFGE*NDIM*(4+2) flops.
                ifl = ggefcs[ig][ivx]-1;
                cnd[0] += xps[ifl][0];
                cnd[1] += xps[ifl][1];
#if NDIM == 3
                cnd[2] += xps[ifl][2];
#endif
                dst[ivx][0] = xps[ifl][0] - crd[0];
                dst[ivx][1] = xps[ifl][1] - crd[1];
#if NDIM == 3
                dst[ivx][2] = xps[ifl][2] - crd[2];
#endif
            };
            cnd[0] /= NDIM+1;   //  MFGE*(2+(4,16)) flops.
            cnd[1] /= NDIM+1;
#if NDIM == 3
            cnd[2] /= NDIM+1;
            sft[0] = dst[0][1]*dst[1][2] - dst[0][2]*dst[1][1];
            sft[1] = dst[0][2]*dst[1][0] - dst[0][0]*dst[1][2];
            sft[2] = dst[0][0]*dst[1][1] - dst[0][1]*dst[1][0];
            vob = fabs(sft[0]*dst[2][0] + sft[1]*dst[2][1] + sft[2]*dst[2][2]);
            vob /= 6;
#else
            vob = fabs(dst[0][0]*dst[1][1] - dst[0][1]*dst[1][0]);
            vob /= 2;
#endif
            voc += vob; // MFGE*(5+2) flops.
            cndge[0] += cnd[0] * vob;
            cndge[1] += cnd[1] * vob;
#if NDIM == 3
            cndge[2] += cnd[2] * vob;
#endif
        };
        cndge[0] /= voc;    // 2+1 flops.
        cndge[1] /= voc;
#if NDIM == 3
        cndge[2] /= voc;
#endif
```

```
        // calculate GGE shift.
        pcecnd = alg->cecnd + icl*(CLMFC+1)*NDIM;
        sft[0] = alg->sftfac * (pcecnd[0] - cndge[0]);  // 4+2 flops.
        sft[1] = alg->sftfac * (pcecnd[1] - cndge[1]);
#if NDIM == 3
        sft[2] = alg->sftfac * (pcecnd[2] - cndge[2]);
#endif
        for (ifl=0; ifl<clnfc; ifl++) { // clnfc*(2+1) flops.
            dsp[ifl][0] += sft[0];
            dsp[ifl][1] += sft[1];
#if NDIM == 3
            dsp[ifl][2] += sft[2];
#endif
        };

        // calculate gradient and weighting delta.
        for (ieq=0; ieq<NEQ; ieq++) {
            dla[ieq] = 0;
        };
        pisoln = alg->soln + icl*NEQ;
        for (ig=ig0; ig<ig1; ig++) {
            ifg = ig-ig0;   // MFGE*1 flops
            for (ivx=0; ivx<NDIM; ivx++) {
                ifl = ggefcs[ig][ivx];
                ifc = pclfcs[ifl];
                ifl -= 1;
                pfccls = msd->fccls + ifc*FCREL;
                jcl = pfccls[0] + pfccls[1] - icl;
                // distance.
                dst[ivx][0] = xps[ifl][0] - cndge[0]; // MFGE*NDIM*(2+1) flops.
                dst[ivx][1] = xps[ifl][1] - cndge[1];
#if NDIM == 3
                dst[ivx][2] = xps[ifl][2] - cndge[2];
#endif
                // solution difference.
                pjsol = alg->sol + jcl*NEQ;
                pjsoln = alg->soln + jcl*NEQ;
                pjsolt = alg->soln + jcl*NEQ;
                pdsol = alg->dsol + jcl*NEQ*NDIM;
                for (ieq=0; ieq<NEQ; ieq++) {   // MFGE*NDIM*NEQ*(9+2) flops.
                    voc = pjsol[ieq] + hdt*pjsolt[ieq] - pjsoln[ieq];
                    voc *= alg->taylor;
                    udf[ieq][ivx] = pjsoln[ieq] + voc - pisoln[ieq];
                    udf[ieq][ivx] += dsp[ifl][0]*pdsol[0];
                    udf[ieq][ivx] += dsp[ifl][1]*pdsol[1];
#if NDIM == 3
                    udf[ieq][ivx] += dsp[ifl][2]*pdsol[2];
#endif
                    pdsol += NDIM;
                };
            };
            // prepare inverse matrix for gradient. MFGE*(3,32) flops.
#if NDIM == 3
            dnv[0][0] = dst[1][1]*dst[2][2] - dst[1][2]*dst[2][1];
            dnv[0][1] = dst[0][2]*dst[2][1] - dst[0][1]*dst[2][2];
            dnv[0][2] = dst[0][1]*dst[1][2] - dst[0][2]*dst[1][1];
            dnv[1][0] = dst[1][2]*dst[2][0] - dst[1][0]*dst[2][2];
            dnv[1][1] = dst[0][0]*dst[2][2] - dst[0][2]*dst[2][0];
            dnv[1][2] = dst[0][2]*dst[1][0] - dst[0][0]*dst[1][2];
            dnv[2][0] = dst[1][0]*dst[2][1] - dst[1][1]*dst[2][0];
            dnv[2][1] = dst[0][1]*dst[2][0] - dst[0][0]*dst[2][1];
            dnv[2][2] = dst[0][0]*dst[1][1] - dst[0][1]*dst[1][0];
            voc = dnv[0][2]*dst[2][0] + dnv[1][2]*dst[2][1]
                + dnv[2][2]*dst[2][2];
#else
            dnv[0][0]  =  dst[1][1]; dnv[0][1] = -dst[0][1];
            dnv[1][0] = -dst[1][0]; dnv[1][1] =  dst[0][0];
            voc = dst[0][0]*dst[1][1] - dst[0][1]*dst[1][0];
#endif
```

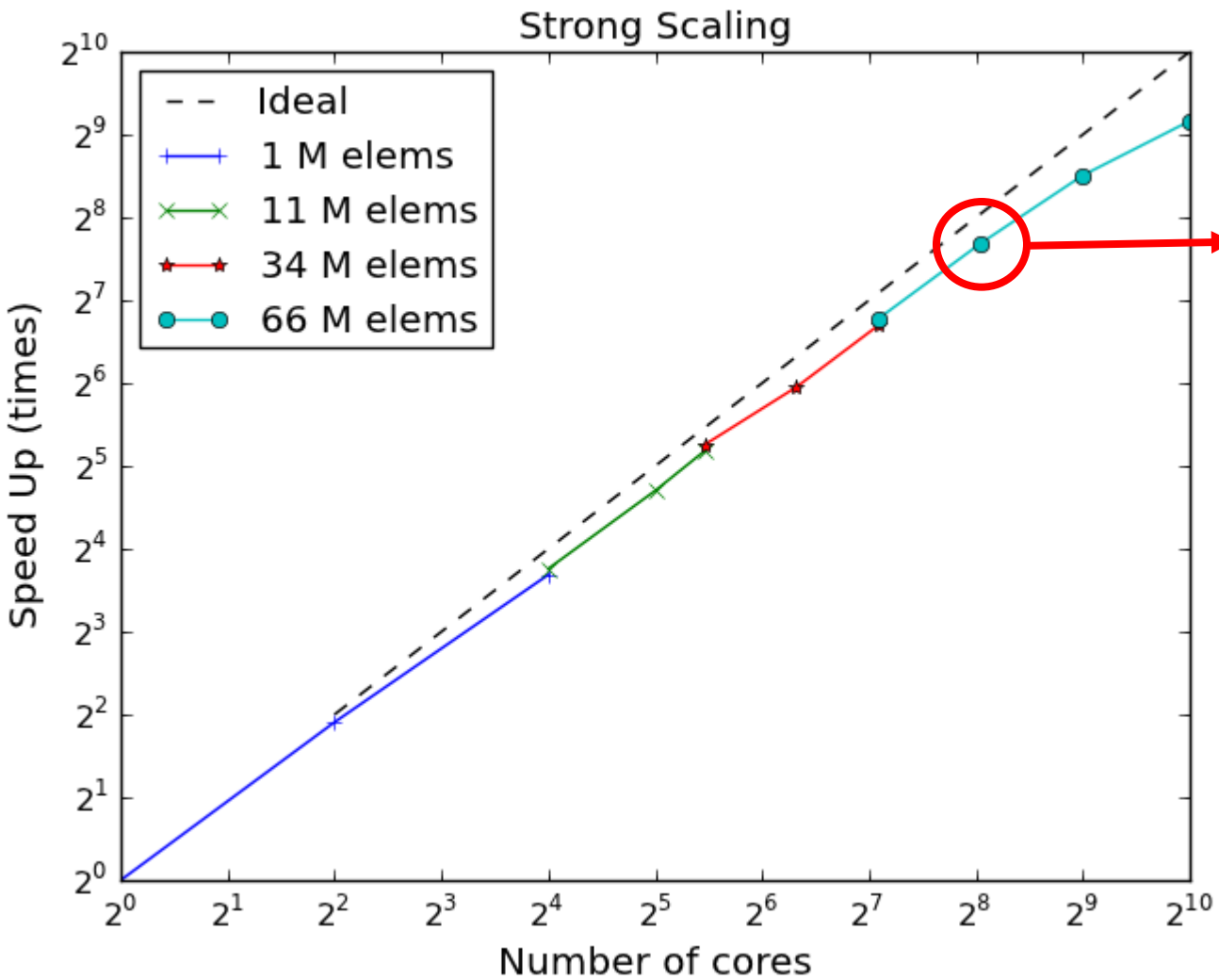
```
            // calculate grdient and weighting delta.
            for (ieq=0; ieq<NEQ; ieq++) {   // MFGE*NEQ*(17+12) flops.
                grd0 = dnv[0][0]*udf[ieq][0] + dnv[0][1]*udf[ieq][1];
#if NDIM == 3
                grd0 += dnv[0][2]*udf[ieq][2];
#endif
                grd0 /= voc;
                grd1 = dnv[1][0]*udf[ieq][0] + dnv[1][1]*udf[ieq][1];
#if NDIM == 3
                grd1 += dnv[1][2]*udf[ieq][2];
#endif
                grd1 /= voc;
#if NDIM == 3
                grd2 = dnv[2][0]*udf[ieq][0] + dnv[2][1]*udf[ieq][1];
                grd2 += dnv[2][2]*udf[ieq][2];
                grd2 /= voc;
#endif
                // store for later weight.
                gfd[ifg][ieq][0] = grd0;
                gfd[ifg][ieq][1] = grd1;
#if NDIM == 3
                gfd[ifg][ieq][2] = grd2;
#endif
                // W-1/2 weight function.
                wgt = grd0*grd0 + grd1*grd1;
#if NDIM == 3
                wgt += grd2*grd2;
#endif
                wgt = 1.0 / pow(sqrt(wgt+ALMOST_ZERO), alg->alpha);
                // store and accumulate weighting function.
                dla[ieq] += wgt;
                dlt[ifg][ieq] = wgt;
            };
        };

        // calculate W-3/4 delta and sigma_max.
        // NOTE: udf is reused here.
        for (ieq=0; ieq<NEQ; ieq++) {
            udf[ieq][0] = udf[ieq][1] = 0.0;
        };
        for (ig=ig0; ig<ig1; ig++) {
            ifg = ig-ig0;
            for (ieq=0; ieq<NEQ; ieq++) {   // MFGE*NEQ*2 flops.
                wgt = dlt[ifg][ieq] / dla[ieq] - ofg1;
                dlt[ifg][ieq] = wgt;
                udf[ieq][0] = fmax(udf[ieq][0], wgt);
                udf[ieq][1] = fmin(udf[ieq][1], wgt);
            };
        };
        // NOTE: dla is reused here for sigma_max.
        for (ieq=0; ieq<NEQ; ieq++) {   // NEQ*5 flops.
            dla[ieq] = fmin(
                (1.0-ofg1)/(udf[ieq][0]-ALMOST_ZERO),
                -ofg1/(udf[ieq][1]-ALMOST_ZERO)
            );
            dla[ieq] = fmin(dla[ieq], sgm0);
        };

        // weight and update gradient.
        pdsoln = alg->dsoln + icl*NEQ*NDIM;
        for (ieq=0; ieq<NEQ; ieq++) {
            pdsoln[0] = pdsoln[1] = 0.0;
#if NDIM == 3
            pdsoln[2] = 0.0;
#endif
            pdsoln += NDIM;
        };
        for (ig=ig0; ig<ig1; ig++) {    // MFGE*NEQ*(6+2) flops.
            ifg = ig-ig0;
            pdsoln = alg->dsoln + icl*NEQ*NDIM;
            for (ieq=0; ieq<NEQ; ieq++) {
                wgt = ofg1 + dla[ieq]*dlt[ifg][ieq];
                pdsoln[0] += wgt*gfd[ifg][ieq][0];
                pdsoln[1] += wgt*gfd[ifg][ieq][1];
#if NDIM == 3
                pdsoln[2] += wgt*gfd[ifg][ieq][2];
#endif
                pdsoln += NDIM;
            };
        };
    };
};
```

```
                pdsoln += NDIM;
            };
        };
    };
};
```

No Clock Should Be Wasted



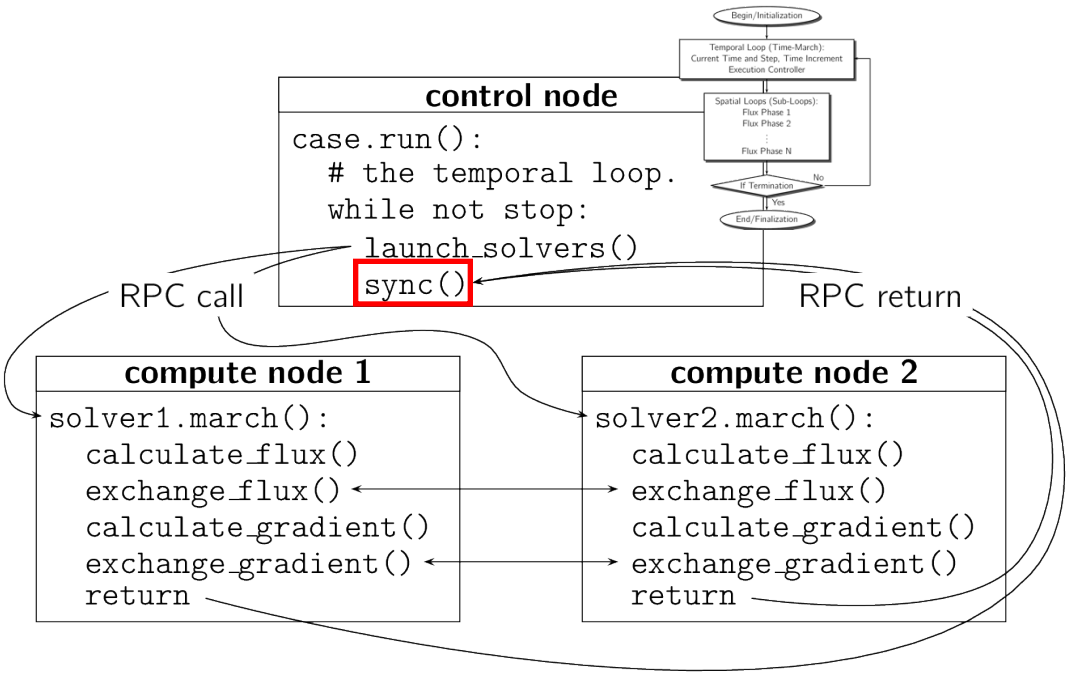
- 66 million elements
- 53 hours
- 66 slave nodes (quad-core)
- 264 cores
- 13992 core-hours

Run this on EC2 could cost USD 300 to 1,000.

When you need this much computing power, you want no waste.

Spatial Loops Executed on Remote Slaves

- Parallel computing is hard.
- Parallel computing on multiple servers is harder.
- SOLVCON defines a class MeshCase to orchestrate the simulation in the control node.
  - It mandates the basic nested-loop structure.



- In slave nodes SOLVCON defines a class MeshSolver to implement spatial loops.
  - It is the number-crunching work horse on the slave nodes.
- The case-solver distinction serves two purposes:
  1. Segregate the numerical code (in solver) from the control code (in case).
  2. Implement remote execution.

Remote Execution

- The de facto infrastructure for massive parallelism is MPI (Message Passing Interface).
  - MPICH, OpenMPI, MVAPICH (this supports RDMA), etc.
  - In Python, the de facto wrapper is mpi4py.
- Python standard library package provides a simple RPC framework:
  - In the slave node run the worker process:

```
from multiprocessing.connection import Listener
with Listener(("localhost", 8723), authkey=b"mypapaword") as listener:
    with listener.accept() as conn:
        func, data = conn.recv()
        print(func(data))
```

  - In the controller node run the commanding process:

```
from multiprocessing.connection import Client
with Client(("localhost", 8723), authkey=b"mypapaword") as conn:
    conn.send((sum, [1, 3, 5]))
```

Implement as a Hybrid System

Languages Involved

SOLVCON's Top-Down Strategy

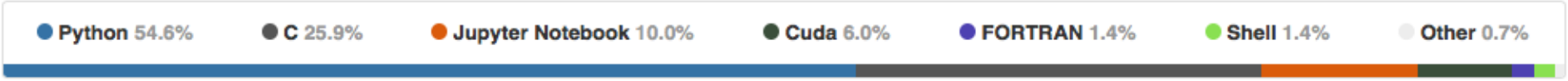
- Python: The framework
- C: The number cruncher
- Cython: Bridge Python and C
  - ctypes is useful too, but Cython is more robust and versatile.
- Data structures are defined in Python

A Bottom-Up Strategy

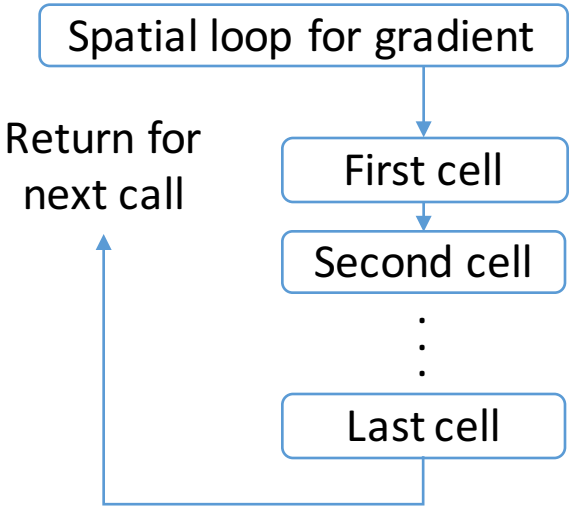
- C++: Fast low-level code
- Boost.Python or similar: Selective exposure of the low-level API
- Python: Wrapper for scripting
- Data structures are defined in C++

C Is the King for Speed

- System looks like Python, works like Python, but is C inside.



- C is used inside the spatial loops for performance.
- I try to minimize API (function calls), and define it at the spatial loop
- Inside a spatial loop, the C code iterates over all elements (cells) and calculates.
- Rule of thumb for HPC code.
  - No function call.
  - No branch statement ("if").
  - Minimal FPU usage.
- C++ may significantly simplify (shorten) the code, but
  - C++11 came just recently.
  - C is difficult enough for Fortran coders.



Python vs C

```c
void caction(double arr0[][1000], double arr1[][1000]) {
    for (int it=1; it<999; it++) {
        for (int jt=1; jt<999; jt++) {
            arr0[it][jt] += arr1[it-1][jt-1];
            arr0[it][jt] += arr1[it-1][jt  ];
            arr0[it][jt] += arr1[it-1][jt+1];
            arr0[it][jt] += arr1[it  ][jt+1];
            arr0[it][jt] += arr1[it+1][jt+1];
            arr0[it][jt] += arr1[it+1][jt  ];
            arr0[it][jt] += arr1[it+1][jt-1];
            arr0[it][jt] += arr1[it  ][jt-1];
        };
    };
};
```

C version runs 51.4 milliseconds

Python version runs 5.74 seconds.

```python
import numpy as np
def action():
    arr0 = np.empty([1000,1000], dtype='float64')
    arr0.fill(0)
    arr1 = np.empty([1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

- Python is 100x slower than C.
- Use Python if you don't care about number-crunching speed.

But We Want Minimal C

- C code isn't easy to maintain!
  - C++ code is even more difficult to maintain.
- C should be only used in spatial loops.
  - Those loops take more than 99% of the total computing time. No compromise of performance.
- Cython is the best tool to combine the two worlds.

```
import numpy as np
cimport numpy as cnp
cdef extern:
    void caction(double* arr0, double* arr1)
def action():
    cdef cnp.ndarray[double, ndim=2] arr0 = np.empty(
        [1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[double, ndim=2] arr1 = np.empty(
        [1000,1000], dtype='float64')
    arr1.fill(1)
    caction(&arr0[0,0], &arr1[0,0])
    assert 7968032 == arr0.sum()
```

This is our C function

Call our C function

Cython Performance Is Good

- For time-consuming code, Cython may be a good choice.
- Trick 1: type variables. Trick 2: direct indexing.

```
import numpy as np
cimport numpy as cnp
def action():
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr0 = np.empty([1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr1 = np.empty([1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

- 139 milliseconds (C 51.4 milliseconds): Just 2.7 times slower than C.

Define Classes in Cython

- Easier OO in the hybrid system.

Declare C fields

```
cdef class Table:
    cdef readonly intptr_t nghost
    cdef readonly intptr_t nbody
    cdef readonly char *_full
    cdef readonly char *_body
    cdef readonly object _nda
```

Define classy wrapper

```
cdef class Table:
    """
    Lookup table that allows ghost entity.
    """

    def __cinit__(self, *args, **kw):
        self.nghost = -1 # Sentinel
        self.nbody = -1 # Sentinel
        self._full = <char*>(NULL) # Sentinel
        self._body = <char*>(NULL) # Sentinel

    def __init__(self, nghost, nbody, *args, **kw):
        self.nghost = int(nghost)
        self.nbody = int(nbody)
        # Pop all custom keyword arguments.
        dtype = kw.pop("dtype", None)
        creator_name = kw.pop("creation", "empty")
        # Create the ndarray.
        create = getattr(np, creator_name)
        self._nda = create([nghost+nbody]+list(args), dtype=dtype)
        if not self._nda.flags.c_contiguous:
            raise ValueError("not C Contiguous")
        ndim = len(self._nda.shape)
        if ndim == 0:
            raise ValueError("zero dimension is not allowed")
        # Assign pointers.
        cdef cnp.ndarray cnda = self._nda
        self._full = self._body = <char*>(cnda.data)
        offset = self._calc_offset(self._nda.shape, self.nghost)
        self._body += self._nda.itemsize * offset
```

Use C struct

- Define C struct

```
cdef public:
    ctypedef struct sc_mesh_t:
        int ndim, nnode, nface, ncell, nbound, ngstnode, ngstface, ngstcell
        # geometry.
        double *ndcrd
        double *fccnd
        double *fcnml
        double *fcara
        double *clcnd
        double *clvol
        # meta.
        int *fctpn
        int *cltpn
        int *clgrp
        # connectivity.
        int *fcnds
        int *fccls
        int *clnds
        int *clfcs
```

- Use it in Cython/Python

```
cdef class Mesh:
    """
    Data set of unstructured meshes of mixed elements.
    """
    def __cinit__(self):
        self._msd = <sc_mesh_t *>malloc(sizeof(sc_mesh_t))

    def __dealloc__(self):
        if NULL != self._msd:
            free(<void*>self._msd)
            self._msd = NULL
```

Recap Cython

- For CFD we are obstinate at speed.
  - But we don't want 6000 lines of Fortran playing with GOTO. Bus factor at nightmare level.
- Python is very good for architecting. We use Cython to make it work with fast C kernels.
  - The hybrid system runs as fast as a pure Fortran reference. Sometimes faster.

- Vanilla Python is 100x slower than C.
- Cython when used correctly can deliver comparable speed to C.

- Cython helps a Python-based system to achieve C speed, and makes the C code look classy in Python.
- It significantly reduces the boilerplates, but there are still constraints.
  - C++ would remove the constraints, but bring other issues to a project.

Boost.Python

- A hello world

```cpp
#include <boost/python.hpp>
char const* greet() {
    return "hello, world";
}
BOOST_PYTHON_MODULE(hello_ext) {
    using namespace boost::python;
    def("greet", greet);
}
```

- Boost.Python use C++ templates to create the boilerplate needed by a Python module.
- It handles Python facilities well, e.g.,
  - Exception

```cpp
#include <boost/python.hpp>
#include <boost/python/errors.hpp>
void translate(const SomeException &err) {
    PyErr(SetString(PyExc_MemoryError, err.what());
}
BOOST_PYTHON_MODULE(error_ext) {
    using namespace boost::python;
    register_exception_translator<SomeException>(translate);
}
```

  - Iterator

```cpp
#include <boost/python.hpp>
#include <list>
struct Iterable {
  typedef std::list<int>::iterator iterator;
  iterator begin() { m_data.begin(); }
  iterator end() { m_data.end(); }
  std::list<int> m_data;
}
BOOST_PYTHON_MODULE(iter_ext) {
  using namespace boost::python;
  class_<Iterable>("Iterable", init<>))
  .add_property("iter", range(Iterable::begin, Iterable::end))
  .def("__iter__", range(Iterable::begin, Iterable::end))
  ;
}
```

- NumPy ndarrays needs special care.

NumPy and Arrays

NumPy

- NumPy package is a foundation of most scientific Python tools.
  - It defines ndarray: The data structure of typed memory buffers.
  - It influenced Python buffer protocol and memory view.
  - Cython provides its own typed memory view.
- Arrays (contiguous memory buffers) are the basics of HPC.
  - Typing is necessary. For CFD number-crunching floating-points are the key.
- Interesting things are the multi-dimensional arrays.
  - NumPy makes them very easy to use. ndarray means N-dimensional array.
  - Managing them in C is painful.
  - C++ smart pointers make managing them less painful, but far away from convenient.
- Basic capabilities:
  - Creation.
  - Shaping, typing, combining, splitting, etc.

Data in Look-Up Tables

- Look-up tables are low-level data structures optimized for reading.

```python
def _build_tables(self, ndim, nnode, nface, ncell,
                  ngstnode, ngstface, ngstcell):
    """
    Allocate memory by creating solvcon.mesh.Tabl

    This method alters content in self object.

    :param ndim: number of dimensionality.
    :param nnode: number of body nodes.
    :param nface: number of body faces.
    :param ncell: number of body cells.
    :param ngstnode: number of nodes for ghost ce
    :param ngstcell: number of faces for ghost ce
    :param ngstcell: number of ghost cells.
    :return: nothing.
    """
    # metrics.
    self.tbndcrd = mesh.Table(ngstnode, nnode, nd
    self.tbfccnd = mesh.Table(ngstface, nface, nd
    self.tbfcnml = mesh.Table(ngstface, nface, nd
    self.tbfcara = mesh.Table(ngstface, nface, dt
    self.tbclcnd = mesh.Table(ngstcell, ncell, nd
    self.tbclvol = mesh.Table(ngstcell, ncell, dt
    # meta/type.
    self.tbfctpn = mesh.Table(ngstface, nface, dt
    self.tbcltpn = mesh.Table(ngstcell, ncell, dt
    self.tbclgrp = mesh.Table(ngstcell, ncell, dt
    self.tbclgrp.F.fill(-1)
    # connectivities.
    self.tbfcnds = mesh.Table(ngstface, nface, se
    self.tbfccls = mesh.Table(ngstface, nface, 4,
    self.tbclnds = mesh.Table(ngstcell, ncell, se
    self.tbclfcs = mesh.Table(ngstcell, ncell, self.CLMFC+1, dtype='int32')
    for name in ('fcnds', 'fccls', 'clnds', 'clfcs'):
        getattr(self, 'tb'+name).F.fill(-1)
```

```cpp
cdef class Table:
    """
    Lookup table that allows ghost entity.
    """

    def __cinit__(self, *args, **kw):
        self.nghost = -1 # Sentinel
        self.nbody = -1 # Sentinel
        self._full = <char*>(NULL) # Sentinel
        self._body = <char*>(NULL) # Sentinel

    def __init__(self, nghost, nbody, *args, **kw):
        self.nghost = int(nghost)
        self.nbody = int(nbody)
        # Pop all custom keyword arguments.
        dtype = kw.pop("dtype", None)
        creator_name = kw.pop("creation", "empty")
        # Create the ndarray.
        create = getattr(np, creator_name)
        self._nda = create([nghost+nbody]+list(args), dtype=dtype)
        if not self._nda.flags.c_contiguous:
            raise ValueError("not C Contiguous")
        ndim = len(self._nda.shape)
        if ndim == 0:
            raise ValueError("zero dimension is not allowed")
        # Assign pointers.
        cdef cnp.ndarray cnda = self._nda
        self._full = self._body = <char*>(cnda.data)
        offset = self._calc_offset(self._nda.shape, self.nghost)
        self._body += self._nda.itemsize * offset
```

- ndarray works perfectly for such use cases.

Popular Creation API

- empty(): Create an untouched buffer.
- zeros(): Create a buffer using calloc.
- arange(): Fill the buffer like range().
- array(): Fill the buffer using the input container.

```
>>> import numpy as np
>>> np.empty(5, dtype='float64')
array([  0.00000000e+000,   0.00000000e+000,   2.13235710e-314,
         2.12311406e-314,   0.00000000e+000])
>>> np.zeros(5, dtype='float64')
array([ 0.,  0.,  0.,  0.,  0.])
>>> np.arange(5, dtype='float64')
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.array([3, 7, 5, -1, 0], dtype='float64')
array([ 3.,  7.,  5., -1.,  0.])
```

- It's a convention to write "`import numpy as np`".

Reshape Arrays

- reshape(): Change to any shape.

```
>>> arr = np.arange(6, dtype='int32')
>>> arr.reshape((2,3))
array([[0, 1, 2],
       [3, 4, 5]], dtype=int32)
>>> arr.reshape((2,3))[0,0] = 7
>>> arr
array([7, 1, 2, 3, 4, 5], dtype=int32)
>>> arr.reshape((2,4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

reshaped array is just a view

cannot change number of elements

- flatten(): Copy to a flat array.

```
>>> brr = arr.reshape((2,3))
>>> brr.flatten()
array([7, 1, 2, 3, 4, 5], dtype=int32)
>>> brr.flatten()[0] = 70
>>> brr
array([[7, 1, 2],
       [3, 4, 5]], dtype=int32)
```

doesn't change original

- ravel(): Create a flat view.

```
>>> brr.ravel()
array([7, 1, 2, 3, 4, 5], dtype=int32)
>>> brr.ravel()[0] = 70
>>> brr
array([[70,  1,  2],
       [ 3,  4,  5]], dtype=int32)
```

change original

Cast Array

- astype(): Cast array elements to a new array.

```
>>> arr
array([70,  1,  2,  3,  4,  5], dtype=int32)
>>> arr.astype('float64')
array([ 70.,   1.,   2.,   3.,   4.,   5.])
>>> arr.astype('float64')[5] = 100
>>> arr
array([70,  1,  2,  3,  4,  5], dtype=int32)
```

- tolist(): Shorthand to make a Python list. 80% faster than list().

```
$ python -m timeit -s "import numpy as np; arr=np.arange(100000)" "arr.tolist()"
100 loops, best of 3: 2.48 msec per loop
$ python -m timeit -s "import numpy as np; arr=np.arange(100000)" "list(arr)"
100 loops, best of 3: 4.39 msec per loop
```

- Data I/O.

```
>>> crr = np.fromstring("1, 3, 5, 7, 9, 2, 4, 6, 8, 10", sep=",")
>>> np.save("crr.npy", crr)
>>> np.load("crr.npy")
array([  1.,   3.,   5.,   7.,   9.,   2.,   4.,   6.,   8.,  10.])
```

Combine Arrays

- hstack(): Stack along row

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_{11} & b_{12} & b_{13} \\ a_{21} & a_{22} & a_{23} & b_{21} & b_{22} & b_{23} \end{pmatrix}$$

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 & b_3 & b_4 \end{pmatrix} \rightarrow \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & b_1 & b_2 & b_3 & b_4 \end{pmatrix}$$

- vstack(): Stack along column

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix}$$

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \end{pmatrix} \rightarrow \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \end{pmatrix}$$

```
>>> arr = np.arange(6, dtype='float64').reshape((2,3))
>>> brr = np.arange(6, 12, dtype='float64').reshape((2,3))
>>> arr
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> brr
array([[  6.,   7.,   8.],
       [  9.,  10.,  11.]])
>>> np.hstack((arr,brr))
array([[  0.,   1.,   2.,   6.,   7.,   8.],
       [  3.,   4.,   5.,   9.,  10.,  11.]])
>>> np.vstack((arr,brr))
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
```

horizontal stack

vertical stack

```
>>> a1rr = np.arange(4, dtype='float64')
>>> b1rr = np.arange(4, 8, dtype='float64')
>>> a1rr
array([ 0.,  1.,  2.,  3.])
>>> b1rr
array([ 4.,  5.,  6.,  7.])
>>> np.hstack((a1rr, b1rr))
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])
>>> np.vstack((a1rr, b1rr))
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]])
```

- concatenate()

```
>>> np.concatenate((arr,brr))
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
>>> np.concatenate((a1rr, b1rr))
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])
```

Slicing

- Make use of Python slice

```
>>> arr = np.arange(6, dtype='float64')
>>> arr
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> arr[0::2]
array([ 0.,  2.,  4.])
>>> arr[::-1]
array([ 5.,  4.,  3.,  2.,  1.,  0.])
```

- Select a whole dimension or dimensions

```
>>> brr = np.arange(15, dtype='float64').reshape((3,5))
>>> brr
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.]])
>>> brr[(0,2),:]
array([[  0.,   1.,   2.,   3.,   4.],
       [ 10.,  11.,  12.,  13.,  14.]])
```

Select only two rows

```
>>> crr = np.arange(24,
...     dtype='float64').reshape((2,3,4))
>>> crr
array([[[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]],
       [[ 12.,  13.,  14.,  15.],
        [ 16.,  17.,  18.,  19.],
        [ 20.,  21.,  22.,  23.]]])
>>> crr[...,(0,2)]
array([[[  0.,   2.],
        [  4.,   6.],
        [  8.,  10.]],
       [[ 12.,  14.],
        [ 16.,  18.],
        [ 20.,  22.]]])
```

Select two groups in the last dimension

Indexing Using Boolean Arrays

- Logical operators return a Boolean array as a “selector”.

```
>>> arr = np.arange(6, dtype='float64')
>>> arr
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> slct = arr < 3
>>> slct
array([ True,  True,  True, False, False, False], dtype=bool)
>>> arr[slct]
array([ 0.,  1.,  2.])

>>> slct = (arr < 3) | (arr > 4)
>>> slct
array([ True,  True,  True, False, False,  True], dtype=bool)
>>> arr[slct]
array([ 0.,  1.,  2.,  5.])
```

- The shapes must be identical

```
>>> slct = np.concatenate((arr < 3, arr > 4))
>>> slct
array([ True,  True,  True, False, False, False, False, False, False,
       False, False,  True], dtype=bool)
>>> arr[slct]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 11 is out of bounds for axis 1 with size 6
```

The Rest

Python Takes You Far

Documentation

www.solvcon.net/en/latest/

read doc database journal python clip

SOLVCON 0.1.4+ documentation » next | modules | index

SOLVCON

Table Of Contents

SOLVCON
- Install
- Documents
- Development
- References

Next topic

Unstructured Meshes

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

SOLVCON

SOLVCON is a collection of conservation-law solvers that use the space-time Conservation Element and Solution Element (CESE) method [Chang95]. The equations to be solved are formulated as:

$$\frac{\partial \mathbf{u}}{\partial t} + \sum_{k=1}^{3} \mathrm{A}^{(k)}(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x_k} = 0$$

where $\mathbf{u}$ is the unknown vector and $\mathrm{A}^{(1)}$, $\mathrm{A}^{(2)}$, and $\mathrm{A}^{(3)}$ are the Jacobian matrices.

Install

Clone from https://github.com/solvcon/solvcon:

```
$ git clone https://github.com/solvcon/solvcon
```

SOLVCON needs the following packages: gcc 4.3+ (clang on OSX works as well), Python 2.7/3.4, six 1.10.0, Cython 0.16+, Numpy 1.5+, LAPACK, NetCDF 4+, SCOTCH 6.0+, Nose 1.0+, Paramiko 1.14+, boto 2.29.1+, gmsh 2.5+, and VTK 5.6+.

A script at `contrib/conda.sh` is provided to install the dependency with Anaconda.

The following command builds and installs SOLVCON:

```
$ python setup.py install
```

After installed you can run the unit tests:

Sphinx documents are good

```
$ python -c 'import solvcon as sc; sc.test()'
```

Additional build and tests:

- Building document requires Sphinx 1.3.1+, pstake 0.3.4+, Sphinxcontrib issue tracker 0.11, and graphviz 2.28
  the binary of SOLVCON is built, the following commands can build the document:

v: latest

Python Speeds up Development

- The simulations are simply Python scripts with delayed evaluation.
  - Users can calculate anything in the “input file”.
- Simple route to in situ visualization.
  - No need to keep the 10TB data per run, and no need to reduce resolution in analysis.
- Abundant 3rdparty libraries. Straight-forward I/O code.
  - 77963 packages now.
- Fast prototyping.
  - Python is indeed slow. But a proof of concept is so easy to code.
- Jupyter notebook.
  - Rapidly prototype prototype code.
  - Wonderful analysis platform.

The Plan of Jobspace

What I Learned in 10 Years

- The unimportant things are critical, and difficult.
- "In order to be sustainable, dedicated resources must be allocated toward the formation of a streamlined and improved software development process that can be leveraged across various projects, lowering software development costs, and releasing researchers and developers to focus on scientific or algorithmic implementation aspects." The second of the six recommendations in NASA/CR-2014-218178.
  - The Richtmyer paper was published in 1950.
- Unimportant things lead to important things.
  - NumPy is made by an electrical engineer.
  - IPython is made by a physicist.
  - Pandas is made by a mathematician.

Have More Fun

- It's difficult to sort out complex things without writing it down.
  - I don't fully understood an equation before deriving it enough times to memorize it.
- Ideas sparkle in critical discussions.
  - Get together intelligent and enthusiastic people. Let people feel comfortable. Have fun.
  - Talk to each other analytically.
- I want to invite more interesting talks to Jobspace.
  - Let's enjoy the hard work of others.
  - Let's enjoy the discussions to each other.
- Thank you!