Contents

Jiang-Tadmor central difference scheme

Jiang and Tadmor proposed a high-resolution finite difference scheme for solving hyperbolic PDE systems in two space dimensions. The method is particularly attractive because, unlike many other methods in this space, it does not require that we write any solvers for problems with special initial data (so-called Riemann problems), nor even that we compute Jacobians of the flux functions.

While this code is based loosely on the Fortran code at the end of Jiang and Tadmor's paper, we've written the current code to be physics-agnostic (rather than hardwiring it to the shallow water equations – or the Euler equations in the Jiang-Tadmor paper). If you're interested in the Euler equations, feel free to add your own physics class to support them!

Staggered grids

The Jiang-Tadmor scheme works by alternating between a main grid and a staggered grid offset by half a step in each direction. Understanding this is important, particularly if you want to apply a domain decomposition method and batch time steps between synchronization barriers in your parallel code!

In even-numbered steps, the entry `u(i,j)` in the array of solution values represents the average value of a cell centered at a point (x_i, y_j). At the following odd-numbered step, the same entry represents values for a cell centered at $(x_i + \Delta x/2, y_j + \Delta y/2)$. However, whenever we run a simulation, we always take an even number of steps, so that outside the solver we can just think about values on the main grid. If `uold` and `unew` represent the information at two successive *even* time steps (i.e. they represent data on the same grid), then `unew(i,j)` depends indirectly on `u(p,q)` for $i - 3 \leq p \leq i + 3$ and $j - 3 \leq q \leq j + 3$.

We currently manage this implicitly: the arrays at even time steps represent cell values on the main grid, and arrays at odd steps represent cell values on the staggered grid. Our main `run` function always takes an even number of time steps to ensure we end up on the primary grid.

MinMod limiter

Numerical methods for solving nonlinear wave equations are complicated by the fact that even with smooth initial data, a nonlinear wave can develop discontinuities (shocks) in finite time.

This makes for interesting analysis, since a "strong" solution that satisfies the differential equation no longer makes sense at a shock – instead, we have to come up with some mathematically and physically reasonable definition of a "weak" solution that satisfies the PDE away from the shock and satisfies some other condition (an entropy condition) at the shock.

The presence of shocks also makes for interesting *numerical* analysis, because we need to be careful about employing numerical differentiation formulas that sample a discontinuous function at points on different sides of a shock. Using such formulas naively usually causes the numerical method to become unstable. A better method – better even in the absence of shocks! – is to consider multiple numerical differentiation formulas and use the highest order one that "looks reasonable" in the sense that it doesn't predict wildly larger slopes than the others. Because these combined formulas *limit* the wild behavior of derivative estimates across a shock, we call them *limiters*. With an appropriate limiter, we can construct methods that have high-order accuracy away from shocks and are at least first-order accurate close to a shock. These are sometimes called *high-resolution* methods.

The MinMod (minimum modulus) limiter is one example of a limiter. The MinMod limiter estimates the slope through points f_-, f_0, f_+ (with the step h

scaled to 1) by

$$f' = \text{minmod}((f_+ - f_-)/2, \theta(f_+ - f_0), \theta(f_0 - f_-))$$

where the minmod function returns the argument with smallest absolute value if all arguments have the same sign, and zero otherwise. Common choices of θ are $\theta = 1.0$ and $\theta = 2.0$.

There are many other potential choices of limiters as well. We'll stick with this one for the code, but you should feel free to experiment with others if you know what you're doing and think it will improve performance or accuracy.

Shallow water equations

Physics picture

The shallow water equations treat water as incompressible and inviscid, and assume that the horizontal velocity remains constant in any vertical column of water. The unknowns at each point are the water height and the total horizontal momentum in a water column; the equations describe conservation of mass (fluid is neither created nor destroyed) and conservation of linear momentum. We will solve these equations with a numerical method that also exactly conserves mass and momentum (up to rounding error), though it only approximately conserves energy.

The basic variables are water height (h), and the velocity components (u, v). We write the governing equations in the form

$$U_t = F(U)_x + G(U)_y$$

where

$$U = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, F = \begin{bmatrix} hu \\ h^2u + gh^2/2 \\ huv \end{bmatrix} G = \begin{bmatrix} hv \\ huv \\ h^2v + gh^2/2 \end{bmatrix}$$

The functions F and G are called *fluxes*, and describe how the conserved quantities (volume and momentum) enter and exit a region of space.

Note that we also need a bound on the characteristic wave speeds for the problem in order to ensure that our method doesn't explode; we use this to control the Courant-Friedrichs-Levy (CFL) number relating wave speeds, time steps, and space steps. For the shallow water equations, the characteristic wave speed is $\sqrt{gh}$ where g is the gravitational constant and h is the height of the water; in addition, we have to take into account the velocity of the underlying flow.

Interface

To provide a general interface, we make the flux and speed functions take arrays that consist of the `h`, `hu`, and `hv` components in sequential arrays separated by

`field_stride`: for example, the start of the height field data is at `U`, the start of the x momentum is at `U+field_stride`, and the start of the y momentum is at `U+2*field_stride`.

```
void shallow2d_flux(float* FU, float* GU, const float* U,
                    int ncell, int field_stride);
void shallow2d_speed(float* cxy, const float* U,
                     int ncell, int field_stride);
```

Implementation

The actually work of computing the fluxes and speeds is done by local (`static`) helper functions that take as arguments pointers to all the individual fields. This is helpful to the compilers, since by specifying the `restrict` keyword, we are promising that we will not access the field data through the wrong pointer. This lets the compiler do a better job with vectorization.

```
static const float g = 9.8;


static
void shallow2dv_flux(float* restrict fh,
                     float* restrict fhu,
                     float* restrict fhv,
                     float* restrict gh,
                     float* restrict ghu,
                     float* restrict ghv,
                     const float* restrict h,
                     const float* restrict hu,
                     const float* restrict hv,
                     float g,
                     int ncell)
{
    memcpy(fh, hu, ncell * sizeof(float));
    memcpy(gh, hv, ncell * sizeof(float));
    for (int i = 0; i < ncell; ++i) {
        float hi = h[i], hui = hu[i], hvi = hv[i];
        float inv_h = 1.0f/hi;
        fhu[i] = hui*hui*inv_h + (0.5f*g)*hi*hi;
        fhv[i] = hui*hvi*inv_h;
        ghu[i] = hui*hvi*inv_h;
        ghv[i] = hvi*hvi*inv_h + (0.5f*g)*hi*hi;
    }
}


static
```

```
void shallow2dv_speed(float* restrict cxy,
                      const float* restrict h,
                      const float* restrict hu,
                      const float* restrict hv,
                      float g,
                      int ncell)
{
    float cx = cxy[0];
    float cy = cxy[1];
    for (int i = 0; i < ncell; ++i) {
        float hi = h[i];
        float inv_hi = 1.0f/h[i];
        float root_gh = sqrtf(g * hi);
        float cxi = fabsf(hu[i] * inv_hi) + root_gh;
        float cyi = fabsf(hv[i] * inv_hi) + root_gh;
        if (cx < cxi) cx = cxi;
        if (cy < cyi) cy = cyi;
    }
    cxy[0] = cx;
    cxy[1] = cy;
}


void shallow2d_flux(float* FU, float* GU, const float* U,
                    int ncell, int field_stride)
{
    shallow2dv_flux(FU, FU+field_stride, FU+2*field_stride,
                    GU, GU+field_stride, GU+2*field_stride,
                    U,  U +field_stride, U +2*field_stride,
                    g, ncell);
}


void shallow2d_speed(float* cxy, const float* U,
                     int ncell, int field_stride)
{
    shallow2dv_speed(cxy, U, U+field_stride, U+2*field_stride, g, ncell);
}
```

Finite volume solver

Interface

Physics function types

From the perspective of the solver, the physics can be characterized by two functions: the flux function and the max wave speed function (used to control the time step). We define callback types for these two functions, with the assumption that the different components of the solution and fluxes are separated by `field_stride`.

```
typedef void (*flux_t)(float* FU, float* GU, const float* U,
                       int ncell, int field_stride);
typedef void (*speed_t)(float* cxy, const float* U,
                        int ncell, int field_stride);
```

Solver data structure

The solver has a number of parameters that define the physics (the `flux` and `speed` functions mentioned above as well as the number of fields `nfield`); the spatial discretization (`nx`, `ny`, `ng`, `dx`, `dy`); and the time discretization (`cfl`). In addition, we have storage for the current solution and for various quantities that are internally important (fluxes and numerical derivatives). These latter should probably be hidden, but I haven't done so yet.

```
typedef struct central2d_t {

    int nfield;   // Number of components in system
    int nx, ny;   // Grid resolution in x/y (without ghost cells)
    int ng;       // Number of ghost cells
    float dx, dy; // Cell width in x/y
    float cfl;    // Max allowed CFL number

    // Flux and speed functions
    flux_t flux;
    speed_t speed;

    // Storage
    float* u;
    float* v;
    float* f;
    float* g;
    float* scratch;

} central2d_t;
```

For the most part, we treat the `central2d_t` as a read-only structure. The

exceptions are the constructor and destructor functions.

```
central2d_t* central2d_init(float w, float h, int nx, int ny,
                            int nfield, flux_t flux, speed_t speed,
                            float cfl);
void central2d_free(central2d_t* sim);
```

For initialization and for reporting on the solution, it's helpful to expose how indexing is done. We manage this with an offset function. Here `k` is the field index, and `(ix,iy)` are the (zero-based) cell index, where cell `(0,0)` is a corner real (non-ghost) cell.

```
int  central2d_offset(central2d_t* sim, int k, int ix, int iy);
```

Running the simulation

The `central2d_run` function advances the simulation from the current state by time `tfinal`. It returns the number of steps taken, determined by the CFL restriction and by the requirement that we always take steps in multiples of two so that we end at the reference grid.

```
int central2d_run(central2d_t* sim, float tfinal);
```

Applying boundary conditions

Ideally, we would not be applying boundary conditions inside the solver, but through an external call (as we do with the other cases). This is because, apart from periodic boundary conditions, the way that we manipulate ghost cell data in order to enforce BCs usually depends a bit on the BCs that are appropriate to the physics (which may vary from field to field). For this exercise, we're always going to use periodic BCs. But I want to leave the interface function public in the eventuality that I might swap in a function pointer for applying the BCs.

```
void central2d_periodic(float* u, int nx, int ny, int ng, int nfield);
```

Implementation

Structure allocation

```
central2d_t* central2d_init(float w, float h, int nx, int ny,
                            int nfield, flux_t flux, speed_t speed,
                            float cfl)
{
    // We extend to a four cell buffer to avoid BC comm on odd time steps
    int ng = 4;

    central2d_t* sim = (central2d_t*) malloc(sizeof(central2d_t));
    sim->nx = nx;
    sim->ny = ny;
```

```
    sim->ng = ng;
    sim->nfield = nfield;
    sim->dx = w/nx;
    sim->dy = h/ny;
    sim->flux = flux;
    sim->speed = speed;
    sim->cfl = cfl;

    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
    int nc = nx_all * ny_all;
    int N  = nfield * nc;
    sim->u  = (float*) malloc((4*N + 6*nx_all)* sizeof(float));
    sim->v  = sim->u +   N;
    sim->f  = sim->u + 2*N;
    sim->g  = sim->u + 3*N;
    sim->scratch = sim->u + 4*N;

    return sim;
}


void central2d_free(central2d_t* sim)
{
    free(sim->u);
    free(sim);
}


int central2d_offset(central2d_t* sim, int k, int ix, int iy)
{
    int nx = sim->nx, ny = sim->ny, ng = sim->ng;
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
    return (k*ny_all+(ng+iy))*nx_all+(ng+ix);
}
```

Boundary conditions

In finite volume methods, boundary conditions are typically applied by setting appropriate values in ghost cells. For our framework, we will apply periodic boundary conditions; that is, waves that exit one side of the domain will enter from the other side.

We apply the conditions by assuming that the cells with coordinates `nghost <= ix <= nx+nghost` and `nghost <= iy <= ny+nghost` are "canonical", and

setting the values for all other cells `(ix,iy)` to the corresponding canonical values `(ix+p*nx,iy+q*ny)` for some integers `p` and `q`.

```
static inline
void copy_subgrid(float* restrict dst,
                  const float* restrict src,
                  int nx, int ny, int stride)
{
    for (int iy = 0; iy < ny; ++iy)
        for (int ix = 0; ix < nx; ++ix)
            dst[iy*stride+ix] = src[iy*stride+ix];
}

void central2d_periodic(float* restrict u,
                        int nx, int ny, int ng, int nfield)
{
    // Stride and number per field
    int s = nx + 2*ng;
    int field_stride = (ny+2*ng)*s;

    // Offsets of left, right, top, and bottom data blocks and ghost blocks
    int l = nx,   lg = 0;
    int r = ng,   rg = nx+ng;
    int b = ny*s, bg = 0;
    int t = ng*s, tg = (nx+ng)*s;

    // Copy data into ghost cells on each side
    for (int k = 0; k < nfield; ++k) {
        float* uk = u + k*field_stride;
        copy_subgrid(uk+lg, uk+l, ng, ny+2*ng, s);
        copy_subgrid(uk+rg, uk+r, ng, ny+2*ng, s);
        copy_subgrid(uk+tg, uk+t, nx+2*ng, ng, s);
        copy_subgrid(uk+bg, uk+b, nx+2*ng, ng, s);
    }
}
```

Derivatives with limiters

In order to advance the time step, we also need to estimate derivatives of the fluxes and the solution values at each cell. In order to maintain stability, we apply a limiter here.

The minmod limiter *looks* like it should be expensive to computer, since superficially it seems to require a number of branches. We do something a little tricky, getting rid of the condition on the sign of the arguments using the `copysign` instruction. If the compiler does the "right" thing with `max` and `min` for floating point arguments (translating them to branch-free intrinsic operations), this

implementation should be relatively fast.

```
// Branch-free computation of minmod of two numbers times 2s
static inline
float xmin2s(float s, float a, float b) {
    float sa = copysignf(s, a);
    float sb = copysignf(s, b);
    float abs_a = fabsf(a);
    float abs_b = fabsf(b);
    float min_abs = (abs_a < abs_b ? abs_a : abs_b);
    return (sa+sb) * min_abs;
}


// Limited combined slope estimate
static inline
float limdiff(float um, float u0, float up) {
    const float theta = 2.0;
    const float quarter = 0.25;
    float du1 = u0-um;   // Difference to left
    float du2 = up-u0;   // Difference to right
    float duc = up-um;   // Twice centered difference
    return xmin2s( quarter, xmin2s(theta, du1, du2), duc );
}


// Compute limited derivs
static inline
void limited_deriv1(float* restrict du,
                    const float* restrict u,
                    int ncell)
{
    for (int i = 0; i < ncell; ++i)
        du[i] = limdiff(u[i-1], u[i], u[i+1]);
}


// Compute limited derivs across stride
static inline
void limited_derivk(float* restrict du,
                    const float* restrict u,
                    int ncell, int stride)
{
    assert(stride > 0);
    for (int i = 0; i < ncell; ++i)
        du[i] = limdiff(u[i-stride], u[i], u[i+stride]);
```

```
}
```

Advancing a time step

Take one step of the numerical scheme. This consists of two pieces: a first-order corrector computed at a half time step, which is used to obtain new F and G values; and a corrector step that computes the solution at the full step. For full details, we refer to the Jiang and Tadmor paper.

The `compute_step` function takes two arguments: the `io` flag which is the time step modulo 2 (0 if even, 1 if odd); and the `dt` flag, which actually determines the time step length. We need to know the even-vs-odd distinction because the Jiang-Tadmor scheme alternates between a primary grid (on even steps) and a staggered grid (on odd steps). This means that the data at (i, j) in an even step and the data at (i, j) in an odd step represent values at different locations in space, offset by half a space step in each direction. Every other step, we shift things back by one mesh cell in each direction, essentially resetting to the primary indexing scheme.

We're slightly tricky in the corrector in that we write

$$v(i,j) = (s(i+1,j) + s(i,j)) - (d(i+1,j) - d(i,j))$$

where $s(i, j)$ comprises the u and x-derivative terms in the update formula, and $d(i, j)$ the y-derivative terms. This cuts the arithmetic cost a little (not that it's that big to start). It also makes it more obvious that we only need four rows worth of scratch space.

```
// Predictor half-step
static
void central2d_predict(float* restrict v,
                       float* restrict scratch,
                       const float* restrict u,
                       const float* restrict f,
                       const float* restrict g,
                       float dtcdx2, float dtcdy2,
                       int nx, int ny, int nfield)
{
    float* restrict fx = scratch;
    float* restrict gy = scratch+nx;
    for (int k = 0; k < nfield; ++k) {
        for (int iy = 1; iy < ny-1; ++iy) {
            int offset = (k*ny+iy)*nx+1;
            limited_deriv1(fx+1, f+offset, nx-2);
            limited_derivk(gy+1, g+offset, nx-2, nx);
            for (int ix = 1; ix < nx-1; ++ix) {
                int offset = (k*ny+iy)*nx+ix;
                v[offset] = u[offset] - dtcdx2 * fx[ix] - dtcdy2 * gy[ix];
```

```
            }
        }
    }
}


// Corrector
static
void central2d_correct_sd(float* restrict s,
                          float* restrict d,
                          const float* restrict ux,
                          const float* restrict uy,
                          const float* restrict u,
                          const float* restrict f,
                          const float* restrict g,
                          float dtcdx2, float dtcdy2,
                          int xlo, int xhi)
{
    for (int ix = xlo; ix < xhi; ++ix)
        s[ix] =
            0.2500f * (u [ix] + u [ix+1]) +
            0.0625f * (ux[ix] - ux[ix+1]) +
            dtcdx2  * (f [ix] - f [ix+1]);
    for (int ix = xlo; ix < xhi; ++ix)
        d[ix] =
            0.0625f * (uy[ix] + uy[ix+1]) +
            dtcdy2  * (g [ix] + g [ix+1]);
}


// Corrector
static
void central2d_correct(float* restrict v,
                       float* restrict scratch,
                       const float* restrict u,
                       const float* restrict f,
                       const float* restrict g,
                       float dtcdx2, float dtcdy2,
                       int xlo, int xhi, int ylo, int yhi,
                       int nx, int ny, int nfield)
{
    assert(0 <= xlo && xlo < xhi && xhi <= nx);
    assert(0 <= ylo && ylo < yhi && yhi <= ny);

    float* restrict ux = scratch;
    float* restrict uy = scratch +   nx;
```

```
    float* restrict s0 = scratch + 2*nx;
    float* restrict d0 = scratch + 3*nx;
    float* restrict s1 = scratch + 4*nx;
    float* restrict d1 = scratch + 5*nx;

    for (int k = 0; k < nfield; ++k) {

        float*       restrict vk = v + k*ny*nx;
        const float* restrict uk = u + k*ny*nx;
        const float* restrict fk = f + k*ny*nx;
        const float* restrict gk = g + k*ny*nx;

        limited_deriv1(ux+1, uk+ylo*nx+1, nx-2);
        limited_derivk(uy+1, uk+ylo*nx+1, nx-2, nx);
        central2d_correct_sd(s1, d1, ux, uy,
                             uk + ylo*nx, fk + ylo*nx, gk + ylo*nx,
                             dtcdx2, dtcdy2, xlo, xhi);

        for (int iy = ylo; iy < yhi; ++iy) {

            float* tmp;
            tmp = s0; s0 = s1; s1 = tmp;
            tmp = d0; d0 = d1; d1 = tmp;

            limited_deriv1(ux+1, uk+(iy+1)*nx+1, nx-2);
            limited_derivk(uy+1, uk+(iy+1)*nx+1, nx-2, nx);
            central2d_correct_sd(s1, d1, ux, uy,
                                 uk + (iy+1)*nx, fk + (iy+1)*nx, gk + (iy+1)*nx,
                                 dtcdx2, dtcdy2, xlo, xhi);

            for (int ix = xlo; ix < xhi; ++ix)
                vk[iy*nx+ix] = (s1[ix]+s0[ix])-(d1[ix]-d0[ix]);
        }
    }
}


static
void central2d_step(float* restrict u, float* restrict v,
                    float* restrict scratch,
                    float* restrict f,
                    float* restrict g,
                    int io, int nx, int ny, int ng,
                    int nfield, flux_t flux, speed_t speed,
                    float dt, float dx, float dy)
{
```

```
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;

    float dtcdx2 = 0.5 * dt / dx;
    float dtcdy2 = 0.5 * dt / dy;

    flux(f, g, u, nx_all * ny_all, nx_all * ny_all);

    central2d_predict(v, scratch, u, f, g, dtcdx2, dtcdy2,
                      nx_all, ny_all, nfield);

    // Flux values of f and g at half step
    for (int iy = 1; iy < ny_all-1; ++iy) {
        int jj = iy*nx_all+1;
        flux(f+jj, g+jj, v+jj, nx_all-2, nx_all * ny_all);
    }

    central2d_correct(v+io*(nx_all+1), scratch, u, f, g, dtcdx2, dtcdy2,
                      ng-io, nx+ng-io,
                      ng-io, ny+ng-io,
                      nx_all, ny_all, nfield);
}
```

Advance a fixed time

The `run` method advances from time 0 (initial conditions) to time `tfinal`. Note that `run` can be called repeatedly; for example, we might want to advance for a period of time, write out a picture, advance more, and write another picture. In this sense, `tfinal` should be interpreted as an offset from the time represented by the simulator at the start of the call, rather than as an absolute time.

We always take an even number of steps so that the solution at the end lives on the main grid instead of the staggered grid.

```
static
int central2d_xrun(float* restrict u, float* restrict v,
                   float* restrict scratch,
                   float* restrict f,
                   float* restrict g,
                   int nx, int ny, int ng,
                   int nfield, flux_t flux, speed_t speed,
                   float tfinal, float dx, float dy, float cfl)
{
    int nstep = 0;
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
    bool done = false;
```

```
    float t = 0;
    while (!done) {
        float cxy[2] = {1.0e-15f, 1.0e-15f};
        central2d_periodic(u, nx, ny, ng, nfield);
        speed(cxy, u, nx_all * ny_all, nx_all * ny_all);
        float dt = cfl / fmaxf(cxy[0]/dx, cxy[1]/dy);
        if (t + 2*dt >= tfinal) {
            dt = (tfinal-t)/2;
            done = true;
        }
        central2d_step(u, v, scratch, f, g,
                       0, nx+4, ny+4, ng-2,
                       nfield, flux, speed,
                       dt, dx, dy);
        central2d_step(v, u, scratch, f, g,
                       1, nx, ny, ng,
                       nfield, flux, speed,
                       dt, dx, dy);
        t += 2*dt;
        nstep += 2;
    }
    return nstep;
}


int central2d_run(central2d_t* sim, float tfinal)
{
    return central2d_xrun(sim->u, sim->v, sim->scratch,
                          sim->f, sim->g,
                          sim->nx, sim->ny, sim->ng,
                          sim->nfield, sim->flux, sim->speed,
                          tfinal, sim->dx, sim->dy, sim->cfl);
}
```

Driver code

The driver code is where we put together the time stepper and the physics routines to actually solve the equations and make pretty pictures of the solutions.

Diagnostics

The numerical method is supposed to preserve (up to rounding errors) the total volume of water in the domain and the total momentum. Ideally, we should also not see negative water heights, since that will cause the system of equations to blow up. For debugging convenience, we'll plan to periodically print diagnostic

information about these conserved quantities (and about the range of water heights).

```
void solution_check(central2d_t* sim)
{
    int nx = sim->nx, ny = sim->ny;
    float* u = sim->u;
    float h_sum = 0, hu_sum = 0, hv_sum = 0;
    float hmin = u[central2d_offset(sim,0,0,0)];
    float hmax = hmin;
    for (int j = 0; j < ny; ++j)
        for (int i = 0; i < nx; ++i) {
            float h = u[central2d_offset(sim,0,i,j)];
            h_sum += h;
            hu_sum += u[central2d_offset(sim,1,i,j)];
            hv_sum += u[central2d_offset(sim,2,i,j)];
            hmax = fmaxf(h, hmax);
            hmin = fminf(h, hmin);
        }
    float cell_area = sim->dx * sim->dy;
    h_sum *= cell_area;
    hu_sum *= cell_area;
    hv_sum *= cell_area;
    printf("-\n  Volume: %g\n  Momentum: (%g, %g)\n  Range: [%g, %g]\n",
           h_sum, hu_sum, hv_sum, hmin, hmax);
    assert(hmin > 0);
}
```

I/O

After finishing a run (or every several steps), we might want to write out a data file for further processing by some other program – in this case, a Python visualizer. The visualizer takes the number of pixels in x and y in the first two entries, then raw single-precision raster pictures.

```
FILE* viz_open(const char* fname, central2d_t* sim, int vskip)
{
    FILE* fp = fopen(fname, "w");
    if (fp) {
        float xy[2] = {sim->nx/vskip, sim->ny/vskip};
        fwrite(xy, sizeof(float), 2, fp);
    }
    return fp;
}

void viz_close(FILE* fp)
{
```

```
    fclose(fp);
}

void viz_frame(FILE* fp, central2d_t* sim, int vskip)
{
    if (!fp)
        return;
    for (int iy = 0; iy < sim->ny; iy += vskip)
        for (int ix = 0; ix < sim->nx; ix += vskip)
            fwrite(sim->u + central2d_offset(sim,0,ix,iy),
                   sizeof(float), 1, fp);
}
```

Lua driver routines

A better way to manage simulation parameters is by a scripting language. Python is a popular choice, but I prefer Lua for many things (not least because it is an easy build). It's also quite cheap to call a Lua function for every point in a mesh (less so for Python, though it probably won't make much difference).

Lua callback functions

We specify the initial conditions by providing the simulator with a callback function to be called at each cell center. The callback function is assumed to be the `init` field of a table at index 1.

```
void lua_init_sim(lua_State* L, central2d_t* sim)
{
    lua_getfield(L, 1, "init");
    if (lua_type(L, -1) != LUA_TFUNCTION)
        luaL_error(L, "Expected init to be a string");

    int nx = sim->nx, ny = sim->ny, nfield = sim->nfield;
    float dx = sim->dx, dy = sim->dy;
    float* u = sim->u;

    for (int ix = 0; ix < nx; ++ix) {
        float x = (ix + 0.5) * dx;
        for (int iy = 0; iy < ny; ++iy) {
            float y = (iy + 0.5) * dy;
            lua_pushvalue(L, -1);
            lua_pushnumber(L, x);
            lua_pushnumber(L, y);
            lua_call(L, 2, nfield);
            for (int k = 0; k < nfield; ++k)
                u[central2d_offset(sim,k,ix,iy)] = lua_tonumber(L, k-nfield);
            lua_pop(L, nfield);
```

```
        }
    }

    lua_pop(L,1);
}
```

Running the simulation

The `run_sim` function looks a lot like the main routine of the "ordinary" command line driver. We specify the initial conditions by providing the simulator with a callback function to be called at each cell center. Note that we have two different options for timing the steps – we can use the OpenMP timing routines (preferable if OpenMP is available) or the POSIX `gettimeofday` if the `SYSTIME` macro is defined. If there's no OpenMP and `SYSTIME` is undefined, we fall back to just printing the number of steps without timing information.

```
int run_sim(lua_State* L)
{
    int n = lua_gettop(L);
    if (n != 1 || !lua_istable(L, 1))
        luaL_error(L, "Argument must be a table");

    lua_getfield(L, 1, "w");
    lua_getfield(L, 1, "h");
    lua_getfield(L, 1, "cfl");
    lua_getfield(L, 1, "ftime");
    lua_getfield(L, 1, "nx");
    lua_getfield(L, 1, "ny");
    lua_getfield(L, 1, "vskip");
    lua_getfield(L, 1, "frames");
    lua_getfield(L, 1, "out");

    double w     = luaL_optnumber(L, 2, 2.0);
    double h     = luaL_optnumber(L, 3, w);
    double cfl   = luaL_optnumber(L, 4, 0.45);
    double ftime = luaL_optnumber(L, 5, 0.01);
    int nx       = luaL_optinteger(L, 6, 200);
    int ny       = luaL_optinteger(L, 7, nx);
    int vskip    = luaL_optinteger(L, 8, 1);
    int frames   = luaL_optinteger(L, 9, 50);
    const char* fname = luaL_optstring(L, 10, "sim.out");
    lua_pop(L, 9);

    central2d_t* sim = central2d_init(w,h, nx,ny,
                                      3, shallow2d_flux, shallow2d_speed, cfl);
    lua_init_sim(L,sim);
```

```
    printf("%g %g %d %d %g %d %g\n", w, h, nx, ny, cfl, frames, ftime);
    FILE* viz = viz_open(fname, sim, vskip);
    solution_check(sim);
    viz_frame(viz, sim, vskip);

    double tcompute = 0;
    for (int i = 0; i < frames; ++i) {
#ifdef _OPENMP
        double t0 = omp_get_wtime();
        int nstep = central2d_run(sim, ftime);
        double t1 = omp_get_wtime();
        double elapsed = t1-t0;
#elif defined SYSTIME
        struct timeval t0, t1;
        gettimeofday(&t0, NULL);
        int nstep = central2d_run(sim, ftime);
        gettimeofday(&t1, NULL);
        double elapsed = (t1.tv_sec-t0.tv_sec) + (t1.tv_usec-t0.tv_usec)*1e-6;
#else
        int nstep = central2d_run(sim, ftime);
        double elapsed = 0;
#endif
        solution_check(sim);
        tcompute += elapsed;
        printf("  Time: %e (%e for %d steps)\n", elapsed, elapsed/nstep, nstep);
        viz_frame(viz, sim, vskip);
    }
    printf("Total compute time: %e\n", tcompute);

    viz_close(viz);
    central2d_free(sim);
    return 0;
}
```

Main

The main routine has the usage pattern

```
lshallow tests.lua args
```

where `tests.lua` has a call to the `simulate` function to run the simulation. The arguments after the Lua file name are passed into the Lua script via a global array called `args`.

```
int main(int argc, char** argv)
{
    if (argc < 2) {
```

```
        fprintf(stderr, "Usage: %s fname args\n", argv[0]);
        return -1;
    }

    lua_State* L = luaL_newstate();
    luaL_openlibs(L);
    lua_register(L, "simulate", run_sim);

    lua_newtable(L);
    for (int i = 2; i < argc; ++i) {
        lua_pushstring(L, argv[i]);
        lua_rawseti(L, 1, i-1);
    }
    lua_setglobal(L, "args");

    if (luaL_dofile(L, argv[1]))
        printf("%s\n", lua_tostring(L,-1));
    lua_close(L);
    return 0;
}
```