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.

Implementation

Our solver takes advantage of C++ templates to get (potentially) good performance while keeping a clean abstraction between the solver code and the details of the physics. The `Shallow2D` class specifies the precision of the comptutation (single precision), the data type used to represent vectors of unknowns and fluxes (the C++ `std::array`). We are really only using the class as name space; we never create an instance of type `Shallow2D`, and the `flux` and `wave_speed` functions needed by the solver are declared as static (and inline, in the hopes of getting the compiler to optimize for us).

```
struct Shallow2D {

    // Type parameters for solver
    typedef float real;
    typedef std::array<real,3> vec;

    // Gravitational force (compile time constant)
    static constexpr real g = 9.8;

    // Compute shallow water fluxes F(U), G(U)
    static void flux(vec& FU, vec& GU, const vec& U) {
        real h = U[0], hu = U[1], hv = U[2];

        FU[0] = hu;
        FU[1] = hu*hu/h + (0.5*g)*h*h;
        FU[2] = hu*hv/h;

        GU[0] = hv;
        GU[1] = hu*hv/h;
        GU[2] = hv*hv/h + (0.5*g)*h*h;
    }

    // Compute shallow water wave speed
    static void wave_speed(real& cx, real& cy, const vec& U) {
        using namespace std;
        real h = U[0], hu = U[1], hv = U[2];
        real root_gh = sqrt(g * h);  // NB: Don't let h go negative!
        cx = abs(hu/h) + root_gh;
        cy = abs(hv/h) + root_gh;
    }
};
```

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 $\theta = 2.0$.

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.

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.

```
template <class real>
struct MinMod {
    static constexpr real theta = 2.0;

    // Branch-free computation of minmod of two numbers
    static real xmin(real a, real b) {
        return ((std::copysign((real) 0.5, a) +
                 std::copysign((real) 0.5, b)) *
                std::min( abs(a), abs(b) ));
    }

    // Limited combined slope estimate
    static real limdiff(real um, real u0, real up) {
        real du1 = u0-um;         // Difference to left
        real du2 = up-u0;         // Difference to right
        real duc = 0.5*(du1+du2); // Centered difference
        return xmin( theta*xmin(du1, du2), duc );
    }
};
```

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.

Interface

We want a clean separation between the physics, the solver, and the auxiliary limiter methods used by the solver. At the same time, we don't want to pay the overhead (mostly in terms of lost optimization opportunities) for calling across an abstraction barrier in the inner loops of our solver. We can get around this in C++ by providing the solver with *template arguments*, resolved at compile time, that describe separate classes to implement the physics and the limiter.

The `Central2D` solver class takes two template arguments: `Physics` and `Limiter`. For `Physics`, we expect the name of a class that defines:

- A type for numerical data (`real`)
- A type for solution and flux vectors in each cell (`vec`)
- A flux computation function (`flux(vec& F, vec& G, const vec& U)`)
- A wave speed computation function (`wave_speed(real& cx, real& cy, const vec& U)`).

The `Limiter` argument is a type with a static function `limdiff` with the signature

```
limdiff(fm, f0, fp)
```

The semantics are that `fm`, `f0`, and `fp` are three successive grid points in some direction, and the function returns an approximate (scaled) derivative value from these points.

The solver keeps arrays for the solution, flux values, derivatives of the solution and the fluxes, and the solution at the next time point. We use the C++ `vector` class to manage storage for these arrays; but since we want to think of them as 2D arrays, we also provide convenience functions to access them with multiple indices (though we maintain C-style 0-based indexing). The internal arrays are padded with ghost cells; the ghost cell in the lower left corner of the domain has index (0,0).

```
template <class Physics, class Limiter>
class Central2D {
public:
    typedef typename Physics::real real;
    typedef typename Physics::vec  vec;

    // If you haven't seen this syntac before, this "constructor" is an C++ Initializer Lis
    Central2D(real w, real h,     // Domain width / height
              int nx, int ny,     // Number of cells in x/y (without ghosts)
              real cfl = 0.45) :  // Max allowed CFL number
        nx(nx), ny(ny),
        nx_all(nx + 2*nghost),
        ny_all(ny + 2*nghost),
        dx(w/nx), dy(h/ny),
        cfl(cfl),
        u_ (nx_all * ny_all),
        f_ (nx_all * ny_all),
        g_ (nx_all * ny_all),
        ux_(nx_all * ny_all),
        uy_(nx_all * ny_all),
        fx_(nx_all * ny_all),
        gy_(nx_all * ny_all),
        v_ (nx_all * ny_all) {}

    // Advance from time 0 to time tfinal
    int run(real tfinal);

    // Call f(Uxy, x, y) at each cell center to set initial conditions
    template <typename F>
    void init(F f);

    // Diagnostics
    void solution_check();

    // Array size accessors
    int xsize() const { return nx; }
    int ysize() const { return ny; }
    // Read / write elements of simulation state
    vec&       operator()(int i, int j) {
        return u_[offset(i+nghost,j+nghost)];
    }

    const vec& operator()(int i, int j) const {
        return u_[offset(i+nghost,j+nghost)];
    }

private:
    static constexpr int nghost = 3;   // Number of ghost cells

    const int nx, ny;          // Number of (non-ghost) cells in x/y
    const int nx_all, ny_all;  // Total cells in x/y (including ghost)
    const real dx, dy;         // Cell size in x/y
    const real cfl;            // Allowed CFL number

    std::vector<vec> u_;            // Solution values
    std::vector<vec> f_;            // Fluxes in x
    std::vector<vec> g_;            // Fluxes in y
    std::vector<vec> ux_;           // x differences of u
    std::vector<vec> uy_;           // y differences of u
    std::vector<vec> fx_;           // x differences of f
    std::vector<vec> gy_;           // y differences of g
    std::vector<vec> v_;            // Solution values at next step

    // Array accessor functions

    int offset(int ix, int iy) const { return iy*nx_all+ix; }

    vec& u(int ix, int iy)    { return u_[offset(ix,iy)]; }
    vec& v(int ix, int iy)    { return v_[offset(ix,iy)]; }
    vec& f(int ix, int iy)    { return f_[offset(ix,iy)]; }
    vec& g(int ix, int iy)    { return g_[offset(ix,iy)]; }

    vec& ux(int ix, int iy)   { return ux_[offset(ix,iy)]; }
    vec& uy(int ix, int iy)   { return uy_[offset(ix,iy)]; }
    vec& fx(int ix, int iy)   { return fx_[offset(ix,iy)]; }
    vec& gy(int ix, int iy)   { return gy_[offset(ix,iy)]; }

    // Wrapped accessor (periodic BC)
    int ioffset(int ix, int iy) {
        return offset( (ix+nx-nghost) % nx + nghost,
                       (iy+ny-nghost) % ny + nghost );
    }
    vec& uwrap(int ix, int iy)  { return u_[ioffset(ix,iy)]; }

    // Apply limiter to all components in a vector
    static void limdiff(vec& du, const vec& um, const vec& u0, const vec& up) {
        for (int m = 0; m < du.size(); ++m)
            du[m] = Limiter::limdiff(um[m], u0[m], up[m]);
    }

    // Stages of the main algorithm
    void apply_periodic();
    void compute_fg_speeds(real& cx, real& cy);
    void limited_derivs();
    void compute_step(int io, real dt);

};
```

Initialization

Before starting the simulation, we need to be able to set the initial conditions. The `init` function does exactly this by running a callback function at the center of each cell in order to initialize the cell U value. For the purposes of this function, cell (i, j) is the subdomain $[i\Delta x, (i+1)\Delta x] \times [j\Delta y, (j+1)\Delta y]$.

```
template <class Physics, class Limiter>
template <typename F>
void Central2D<Physics, Limiter>::init(F f)
{
    for (int iy = 0; iy < ny; ++iy)
        for (int ix = 0; ix < nx; ++ix)
            f(u(nghost+ix,nghost+iy), (ix+0.5)*dx, (iy+0.5)*dy);
}
```

Time stepper implementation

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`.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::apply_periodic()
{
    // Copy data between right and left boundaries
    for (int iy = 0; iy < ny_all; ++iy)
        for (int ix = 0; ix < nghost; ++ix) {
            u(ix,          iy) = uwrap(ix,          iy);
            u(nx+nghost+ix,iy) = uwrap(nx+nghost+ix,iy);
        }

    // Copy data between top and bottom boundaries
    for (int ix = 0; ix < nx_all; ++ix)
        for (int iy = 0; iy < nghost; ++iy) {
            u(ix,          iy) = uwrap(ix,          iy);
            u(ix,ny+nghost+iy) = uwrap(ix,ny+nghost+iy);
        }
}
```

Initial flux and speed computations

At the start of each time step, we need the flux values at cell centers (to advance the numerical method) and a bound on the wave speeds in the x and y directions (so that we can choose a time step that respects the specified upper bound on the CFL number).

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_fg_speeds(real& cx_, real& cy_)
{
    using namespace std;
    real cx = 1.0e-15;
    real cy = 1.0e-15;
    for (int iy = 0; iy < ny_all; ++iy)
        for (int ix = 0; ix < nx_all; ++ix) {
            real cell_cx, cell_cy;
            Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));
            Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));
            cx = max(cx, cell_cx);
            cy = max(cy, cell_cy);
        }
    cx_ = cx;
    cy_ = cy;
}
```

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.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::limited_derivs()
{
    for (int iy = 1; iy < ny_all-1; ++iy)
        for (int ix = 1; ix < nx_all-1; ++ix) {

            // x derivs
            limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
            limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );

            // y derivs
            limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
            limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
        }
}
```

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.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_step(int io, real dt)
{
    real dtcdx2 = 0.5 * dt / dx;
    real dtcdy2 = 0.5 * dt / dy;
    // Predictor (flux values of f and g at half step)
    for (int iy = 1; iy < ny_all-1; ++iy)
        for (int ix = 1; ix < nx_all-1; ++ix) {
            vec uh = u(ix,iy);
            for (int m = 0; m < uh.size(); ++m) {
                uh[m] -= dtcdx2 * fx(ix,iy)[m];
                uh[m] -= dtcdy2 * gy(ix,iy)[m];
            }
            Physics::flux(f(ix,iy), g(ix,iy), uh);
        }

    // Corrector (finish the step)
    for (int iy = nghost-io; iy < ny+nghost-io; ++iy)
        for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
            for (int m = 0; m < v(ix,iy).size(); ++m) {
                v(ix,iy)[m] =
                    0.2500 * ( u(ix,  iy)[m] + u(ix+1,iy  )[m] +
                               u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -
                    0.0625 * ( ux(ix+1,iy  )[m] - ux(ix,iy  )[m] +
                               ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
                               uy(ix,  iy+1)[m] - uy(ix,  iy)[m] +
                               uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m] ) -
                    dtcdx2 * ( f(ix+1,iy  )[m] - f(ix,iy  )[m] +
                               f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -
                    dtcdy2 * ( g(ix,  iy+1)[m] - g(ix,  iy)[m] +
                               g(ix+1,iy+1)[m] - g(ix+1,iy)[m] );
            }
        }

    // Copy from v storage back to main grid
    for (int j = nghost; j < ny+nghost; ++j){
        for (int i = nghost; i < nx+nghost; ++i){
            u(i,j) = v(i-io,j-io);
        }
    }
}
```

Advance 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.

```
template <class Physics, class Limiter>
int Central2D<Physics, Limiter>::run(real tfinal)
{
    bool done = false;
    real t = 0;
    int nstep = 0;
    while (!done) {
        real dt;
        for (int io = 0; io < 2; ++io) {
            real cx, cy;
            apply_periodic();
            compute_fg_speeds(cx, cy);
            limited_derivs();
            if (io == 0) {
                dt = cfl / std::max(cx/dx, cy/dy);
                if (t + 2*dt >= tfinal) {
                    dt = (tfinal-t)/2;
                    done = true;
                }
            }
            compute_step(io, dt);
            t += dt;
            ++nstep;
        }
    }
    return nstep;
}
```

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).

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::solution_check()
{
    using namespace std;
    real h_sum = 0, hu_sum = 0, hv_sum = 0;
    real hmin = u(nghost,nghost)[0];
    real hmax = hmin;
    for (int j = nghost; j < ny+nghost; ++j)
        for (int i = nghost; i < nx+nghost; ++i) {
            vec& uij = u(i,j);
            real h = uij[0];
            h_sum += h;
            hu_sum += uij[1];
            hv_sum += uij[2];
            hmax = max(h, hmax);
            hmin = min(h, hmin);
            assert( h > 0) ;
        }
    real cell_area = dx*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);
}
```

I/O

After finishing a run (or every several steps), we might want to write out a data file for post processing. One simple approach is to draw a gray scale or color picture showing some scalar quantity at each point. The Portable Gray Map (PGM) format is one of the few graphics formats that can be dumped out in a handful of lines of code without any library calls. The files can be converted to something more modern and snazzy (like a PNG or GIF) later on. Note that we don't actually dump out the state vector for each cell – we need to produce something that is an integer in the range [0,255]. That's what the function `f` is for!

```
template <class Sim, typename F>
void write_pgm(const char* fname, const Sim& u, F f)
{
    using namespace std;
    FILE* fp = fopen(fname, "wb");
    fprintf(fp, "P5\n");
    fprintf(fp, "%d %d 255\n", u.xsize(), u.ysize());
    for (int iy = u.ysize()-1; iy >= 0; --iy)
        for (int ix = 0; ix < u.xsize(); ++ix)
            fputc(min(255, max(0, f(u(ix,iy)))), fp);
    fclose(fp);
}
```

An alternative to writing an image file is to write 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.

```
template <class Sim>
class SimViz {
public:

    SimViz(const char* fname, const Sim& sim) : sim(sim) {
        fp = fopen(fname, "w");
        if (fp) {
            float xy[2];
            xy[0] = sim.xsize();
            xy[1] = sim.ysize();
            fwrite(xy, sizeof(float), 2, fp);
        }
    }

    void write_frame() {
        if (fp)
            for (int j = 0; j < sim.ysize(); ++j)
                for (int i = 0; i < sim.xsize(); ++i) {
                    float uij = sim(i,j)[0];
                    fwrite(&uij, sizeof(float), 1, fp);
                }
    }

    ~SimViz() {
        if (fp)
            fclose(fp);
    }

private:
    const Sim& sim;
    FILE* fp;
};
```

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.

For the driver, we need to put everything together: we're running a `Central2D` solver for the `Shallow2D` physics with a `MinMod` limiter:

```
typedef Central2D< Shallow2D, MinMod<Shallow2D::real> > Sim;
```

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 helpers

We want to be able to get numbers and strings with a default value when nothing is specified. Lua 5.3 has this as a built-in, I think, but the following codes are taken from earlier versions of Lua.

```
double lget_number(lua_State* L, const char* name, double x)
{
    lua_getfield(L, 1, name);
    if (lua_type(L, -1) != LUA_TNIL) {
        if (lua_type(L, -1) != LUA_TNUMBER)
            luaL_error(L, "Expected %s to be a number", name);
        x = lua_tonumber(L, -1);
    }
    lua_pop(L, 1);
    return x;
}


int lget_int(lua_State* L, const char* name, int x)
{
    lua_getfield(L, 1, name);
    if (lua_type(L, -1) != LUA_TNIL) {
        if (lua_type(L, -1) != LUA_TNUMBER)
            luaL_error(L, "Expected %s to be a number", name);
        x = lua_tointeger(L, -1);
    }
    lua_pop(L, 1);
    return x;
}


const char* lget_string(lua_State* L, const char* name, const char* x)
{
    lua_getfield(L, 1, name);
    if (lua_type(L, -1) != LUA_TNIL) {
        if (lua_type(L, -1) != LUA_TSTRING)
            luaL_error(L, "Expected %s to be a string", name);
        x = lua_tostring(L, -1);
    }
    lua_pop(L, 1);
    return x;
}
```

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.

```
class LuaInit {
public:
    LuaInit(lua_State* L) : L(L) {}

    void push_init() {
        lua_getfield(L, 1, "init");
        if (lua_type(L, -1) != LUA_TFUNCTION)
            luaL_error(L, "Expected init to be a string");
    }

    void operator()(Sim::vec& u, double x, double y) {
        lua_pushvalue(L, -1);
        lua_pushnumber(L, x);
        lua_pushnumber(L, y);
        int nfield = u.size();
        lua_call(L, 2, nfield);
        for (int k = 0; k < nfield; ++k)
            u[k] = lua_tonumber(L, k-nfield);
        lua_pop(L, nfield);
    }

private:
    lua_State* L;
};
```

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");

    double w = lget_number(L, "w", 2.0);
    double h = lget_number(L, "h", w);
    double cfl = lget_number(L, "cfl", 0.45);
    double ftime = lget_number(L, "ftime", 0.01);
    int nx = lget_int(L, "nx", 200);
    int ny = lget_int(L, "ny", nx);
    int frames = lget_int(L, "frames", 50);
    const char* fname = lget_string(L, "out", "sim.out");

    Sim sim(w, h, nx, ny);

    LuaInit initf(L);
    initf.push_init();
    sim.init(initf);
    lua_pop(L,1);

    printf("%g %g %d %d %g %d %g\n", w, h, nx, ny, cfl, frames, ftime);
    SimViz<Sim> viz(fname, sim);
    sim.solution_check();
    viz.write_frame();

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

    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;
}
```