TDLI Summer School 2018 – Computational Astrophysics

1. Advection Equation

Write a program that solves the advection equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0,$$

using periodic boundary conditions, a grid of 100 zones, a grid length of 1 ($x = 0 \ldots 1$), and the following initial conditions:

$$u = \begin{cases} 1, x < 1/2 \\ 0, x \geq 1/2 \end{cases}$$

Evolve this square wave to $t = 0.5$ using *three* of the following methods:

(a) Lax-Friedrichs

(b) Lax-Wendroff

(c) 1st order upwind differences

(d) 2nd order RSA (reconstruct-solve average) with the minmod limiter.

(e) 2nd order RSA (reconstruct-solve average) with the MC limiter.

Plot the results for $t = 0.5$.

2. Writing a Hydro Code

(a) Write a 1D Godunov-based hydro code with the following features:

- HLLE and HLLC Riemann solver
- 2nd order reconstruction using the van Leer or MC limiter
- A 2nd order time-integrator

Solve the shock tube problem with a discontinuity at $x = 0.5$ on a grid of 256 points covering the range $0 \ldots 1$ for the following left and right states using an ideal gas equation of state with $\gamma = 5/3$:

- $\rho_L = 1, \rho_R = 0.125, P_L = 1, P_R = 0.1, u_L = u_R = 0$
- $\rho_L = \rho_R = 1, P_L = P_R = 0.01, u_L = -0.2, u_R = 0.2$ (What boundary conditions do you need to use?)
- $\rho_L = \rho_R = 1, P_L = 0.4, P_R = 0.02, u_L = 0.4, u_R = -1.4$

Plot your results for ρ, P, and u at $t = 0.2$ and compare them with the exact solution.

I recommend implementing the HLLE solver first, and then switching to HLLC after doing some tests.

(b) Generalise your hydro code to 2D, preferrably using an unsplit approach. I will make an output routine available to you to write HDF5 files that you can read in VISIT. If you're keen you can also try to write your own output routine for 2D data.

(c) Use the 2D code to simulate the Kelvin-Helmholtz instability. You will need periodic boundary conditions in the x-direction and periodic boundary conditions in the y-direction. Choose $\rho = P = 1$ in non-dimensional units and $v_x = \pm 1/2$. To track the two fluids on both side of the interface, you can, e.g., set $v_z = \pm -0.01$ and use v_z as a tracer. Define appropriate initial conditions $y = y_0 + \delta \cos 2\pi x/\lambda$. for the shear interface. The amplitude of the deformation should be a few cells initially. Choose the extent of your domain as $0 \le x \le 1$ and $0 \le y \le 1$.

(d) Implement a constant gravitational acceleration and simulate the Rayleigh-Taylor instability in 2D using two fluids with a density ratio of $\rho_0/\rho_1 = 1/2$.

(e) Simulate a shock tube problem with $\rho_{\rm L} = 1$, $\rho_{\rm R} = 1$, $P_{\rm L} = P_{\rm R} = 1$, $v_{x,{\rm L}} = 5$, and $v_{x,{\rm L}} = 0$ in 2D. For this problem, you need fixed boundary conditions in the x-direction and periodic boundary conditions in the y-direction. Then insert a deformed contact discontinuity ahead of the shock:

$$\rho = 2 \quad \text{if} \quad x > 0.4 + 0.15\cos(20\pi y)\cos(3.224\pi y).$$

What happens when the shock runs over this contact discontinuity?

Programming advice:

I recommend that you structure you code roughly as described below. I also recommend that you explicitly pass all the required input data to a given subroutine when you call it instead of storing it globally in a module variable. This is not always feasible in a really large code, and may not allow you or the compiler to optimise the code to get the last GFLOP out of your CPU. It will, however, help you debug your code and avoid indexing errors (which are among the nastiest bugs you will ever come across).

I also recommend that you write your code in Fortran. If there's one thing that Fortran is good at it's handling arrays, and that is extremely useful for grid-based hydro codes.

(a) Subroutine **boundary**:

Task: Set ghost zones for directional sweep.
Input/Output: A quantity defined in cells $1, 2, \ldots, n$, to which ghost zones will be added (zones $-1, 0$ and $n+1, n+2$).
Input: n, and if you want to do more complicated test problems also a switch for boundary conditions (e.g. 1 for periodic, 2 for reflecting boundary conditions, 3 for pre-specified values, which would have to be passed as an additional argument).

(b) Subroutine **conserved_var**:

Task: Convert primitive variables to conserved variables.
Input: An array containing the primitive variables, c_s and P may still be undefined.
Output: An array containing all the conserved variables.

(c) Subroutine **eos**: Task: Determine pressure and sound speed using an ideal gas, gamma-law equation of state $P = (\gamma - 1)\rho\epsilon$
Input: ρ, ϵ as arrays of length m (specifiable as argument).
Output: P and c_s as arrays of length m.

(d) Subroutine **primitives**: Task: Convert conserved variables to primitive variables.
Input: A $m \times 5$ array containing the conserved variables ρ, ρu, ρv, ρw, $\rho(\epsilon + v^2/2)$. m should be arbitrary and specifiable as an argument.
Output: A $m \times 7$ array containing the primitives ρ, u, v, w, ϵ, P and c_s.

(e) Subroutine **riemann**: Solve the Riemann problem at cell interfaces.
Input: ρ, u, v, w, ϵ, P, c_s on the left and right boundary of cells $0, 1, \ldots, n, n+1$ and the number of zones n.
Output: The fluxes ρu, $\rho u^2 + P$, $(e + P)u$ at cell interfaces (index $0 \ldots n$).

(f) Subroutine **reconstruct**: Task: Using a piecewise-linear interpolant, determine the state variables at cell interfaces. Uniform grid spacing can be assumed.
Input: A cell-centred quantity defined in zones $-1, 0, 1, \ldots, n, n+1$.
Output: The reconstructed value at the left and right interface of zones $0, 1, 2, \ldots, n, n+1$.

(g) Subroutine **rk_step**: Task:

- Determine the global minimum for the allowed time-step using by calling **timestep**. Be careful in 2D!
- Pass 1D arrays containing the primitive variables to **sweep** to determine the time derivatives of the conserved variables.
- Advance the conserved variables by one time step using your favourite time integrator (e.g. Heun's method) and update the simulation time.

Input/Output: Arrays containing the conserved (dimension $n_x \times n_y \times 5$) and primitives quantities (dimension $n_x \times n_y \times 7$).
Input: The grid dimensions n_x and n_y, the grid spacing along each direction, and the current simulation time.
Optional Input:

- Value of the gravitational acceleration (in 2D: for each direction!)
- Switch for boundary conditions and an array of values for ghost cells to be passed on to **boundary** (in 2D: for each direction!).

(h) Subroutine **setup_grid**: Task: Set up the computational grid.
Input: Grid length and number of zones in each direction.
Output: Grid spacing and cell coordinates.

(i) Subroutine **setup_sod**: Task: Set initial conditions.
Input: ρ, u and P for the Sod shock tube problem on both sides of the discontinuity.
Output: An array containing the primitive variables (you don't need to set c_s yet.

(j) Subroutine **sweep**: Task:

- Call **boundary** to set ghost zones for 1D sweep.
- Call **reconstruct** to interpolate onto cell interfaces.
- Call **riemann** to obtain the fluxes at cell interfaces.
- Compute time derivatives of the conserved variables for a 1D slice of the hydro grid.

Input:

- An $n \times 7$ array containing the primitive variables:
- The grid spacing δx.

Output: An $n \times 5$ array containing the time derivative of the conserved variables.
Optional Input:

- Value of the gravitational acceleration.
- Switch for boundary conditions and an array of values for ghost cells to be passed on to **boundary**.
- In multi-D: The sweep direction.

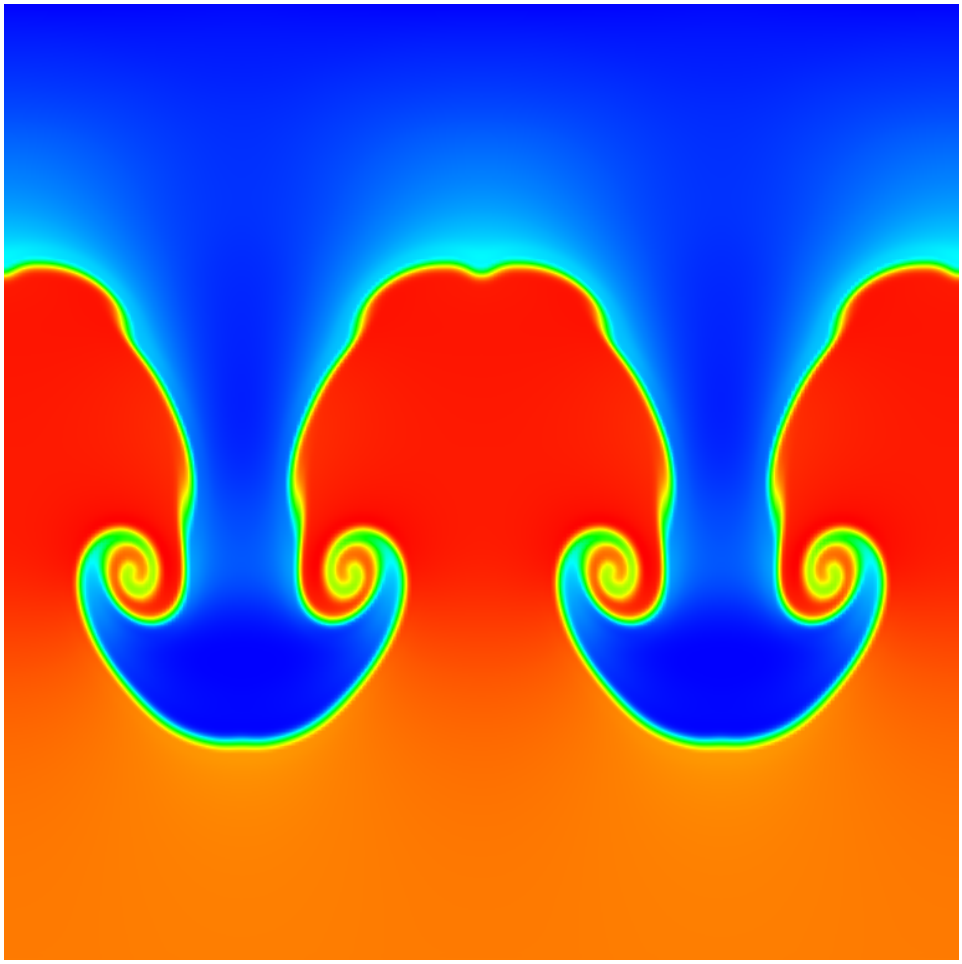
(k) Subroutine **timestep**:

Task: Determine the CFL time step.
Input: A $m \times 7$ array containing the primitives (with specifiable m), and the grid spacing δx.
Output: The maximum time step δt allow by the CFL criterion reduced by a CFL factor that you have to choose and that may depend on your setup.

(l) Program **hydro**:

Task:

- Call routine to set up the grid and the hydro data.
- Call **rk_step** until the final simulation time is reached.
- Output the data at the end and (recommended) after specified intervals during the simulation.



3. Radiation Hydrodynamics

(a) **Stiff Source Terms in the Collision Integral**: The contributions of absorption/emission reactions like $n + \nu_e \leftrightharpoons p + e^-$ in the collision integral can be written in the following form,

$$\left(\frac{\partial f}{\partial t}\right)_{n\ \nu_e \leftrightharpoons p\ e^-} = -\kappa c(f - f_{\mathrm{eq}}).$$

From the previous problem, you know that the equilibration time-scale $1/(\kappa c)$ can be extremely short. This causes problems if we want to take reasonable time steps in a simulation ($\Delta t = 10^{-7} \ldots 10^{-6}$ s).

Consider the simple differential equation,

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\lambda(u - \sin t).$$

In the Python programme `stiff_ode_explicit.py`, we try to solve this with a simple forward Euler method, where we advance to the next time step according to

$$u_{i+1} = u_i - \delta t(u_i - \sin t_i).$$

As initial condition, we choose $u(0) = 1$. Run the programme by executing:

`python3.5 stiff_ode_explicit.py`

What happens at very small t? Next, edit the programme to set the time step to $\Delta t = 0.018$, then to $\Delta t = 0.02$ and to $\Delta t = 0.025$. Is this still an acceptable solution?

Apparently, we face a severe time step constraint. Going to higher-order integration methods (Runge-Kutta) does not cure this limitation. To avoid the problem, we can compute the right-hand side at the next time step instead:

$$u_{i+1} = u_i - \delta t(u_{i+1} - \sin t_{i+1}).$$

This is the backward Euler method. For this simple example, you can easily modify the algorithm to implement it. Verify that the solution remains stable for large time steps.

The backward Euler method is stable but not very accurate. One could try to achieve higher-order accuracy by solving

$$u_{i+1} = u_i - \delta t\left(\frac{u_{i+1} + u_i}{2} - \sin t_{i+1/2}\right).$$

instead (implicit midpoint method). Try whether you can implement this method and find out whether it is stable for arbitrary Δt. If so does it also give a "nice solution"?

Note: Stiff source terms also occur in many other contexts (e.g. nuclear reaction networks, cooling in radiative shock). For some of these applications, implicit higher-order methods (Kaps-Rentrop, Bader-Deuflhard) are the method of choice, which we do not treat here.

(b) **The Diffusion Equation:** Verify that the diffusion approximation we discussed it in the lectures boils down to a single evolution equation for J:

$$\frac{\partial J}{\partial t} - \frac{1}{3}\nabla \cdot \left(\frac{1}{\kappa_s + \kappa_a}\nabla J\right) = \kappa_a(J_{\rm eq} - J).$$

This is very similar to the prototypical 1D equation,

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0.$$

For this equation, a simple finite-difference scheme to advance a solution on from time step n to time step $n + 1$ might be:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \frac{1}{\Delta x}\left(\frac{u_{i+1}^n - u_i^n}{\Delta x} - \frac{u_i^n - u_{i-1}^n}{\Delta x}\right) = 0.$$

Try to implement this in Python on a domain $0 \le x \le 1$ with initial conditions

$$u(x, 0) = e^{-\frac{(x-0.5)^2}{0.2}}.$$

Run the programme using 100 zones and the maximum time step $\Delta t = \Delta x^2/2$ and compare to the analytical solution

$$u(x, t) = e^{-\frac{(x-0.5)^2}{2\times(0.1^2+2t^2)}}\sqrt{\frac{0.1^2}{0.1^2 + 2t^2}}$$

Check what happens for $\Delta t = \Delta x^2$ or larger Δt.

Again, the way to avoid the severe time step constraint is to use an implicit method. Implement the backward Euler method,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \frac{1}{\Delta x}\left(\frac{u_{i+1}^{n+1} - u_i^{n+1}}{\Delta x} - \frac{u_i^{n+1} - u_{i-1}^{n+1}}{\Delta x}\right) = 0.$$

Since this method couples the new solution values at all grid points with each other, we need to solve a linear system with a tridiagonal coefficient matrix. The SciPy function `scipy.linalg.solve_banded()` can be used for this purpose. Run the code with various time steps far greater than the explicit stability limit (e.g. $\Delta t = (100 \ldots 200)\Delta x^2$) and verify that the method remains stable. Also run with $\Delta t = \Delta x^2/2$. Can you see a difference in the numerical solution? If you are brave, you can try to adjust the code to get second-order convergence in time (how?). Does this make the numerical solution more accurate for large Δt?

(c) **Two-Moment Approximation:** Let us now solve a two-moment system in 1D assuming a constant scattering opacity κ and no source term in the equation for J:

$$\begin{aligned} \frac{\partial J}{\partial t} + \frac{\partial H}{\partial x} &= 0 \\ \frac{\partial H}{\partial t} + \frac{1}{3}\frac{\partial K}{\partial x} &= -\kappa H. \end{aligned}$$

Solve this equation by operator splitting, i.e. first advancie the hyperbolic part of the system (without source terms) for one time-step, and then update H using an implicit method.

Use the same Gaussian as in the previous problem as initial conditions for J. Run with $\kappa = 100$ and convince yourself that the numerical solution remains close to the analytic solution of a diffusion equation with the same κ (Why?).

Now set $\kappa = 10$. Do you see changes?

Next, set $\kappa = 10^5$ and re-run. Nothing happens because diffusion is now extremely slow. Then change the width of the Gaussian (variable `width`); set it to 0.01 instead of 0.1. What happens now? Is this physical, and if not what is the reason for this phenomenon?

If you have some background in computational astrophysics, you can try to fix this: For the solution of the hyperbolic part, we use the Kurganov-Tadmor central scheme with signal velocity $c_s = 1/\sqrt{3}$. See what happens if you switch of the diffusive term in the flux formula if the optical depth per zone $\tau = \kappa \delta x$ is larger than unity, or suppress this term by multiplying it with $\max(1 - \tau, 0)$.