

Les Houches Summer School on Gravitational Waves 2018

Exercises Numerical Relativity

Harald Pfeiffer, Albert-Einstein-Institute

This morning, we discussed various algorithms to solve hyperbolic partial differential equations. Let's get our hands dirty and actually code one example to completion.

Your goal is to program a numerical PDE solver for the simplest of hyperbolic equations, the **1-D advection equation**,

$$\frac{\partial u}{\partial t}(x, t) + \frac{\partial u}{\partial x}(x, t) = 0, \quad x \in [0, 1], t \geq 0. \quad (1)$$

As indicated, we will use the interval $x \in [0, 1]$, and to keep things simple, we will use **periodic boundary conditions**.

We aim to solve Eq. (1) with three different numerical methods: (1) finite-differences, (2) pseudo-spectral methods, and (3) discontinuous Galerkin.

If you have never coded before, this is a challenging assignment. In that case, even “just” getting the finite-differences code to work will be an impressive accomplishment. On the other hand, if you have a lot of numerical experience, you might find the early tasks quite easy, and arrive quickly at the more advanced methods. Work at your own pace. We recommend you work in groups, however, it is educational if everybody in a group tries to get her/his own code working.

We will use the initial guess

$$u(x, 0) = e^{-2 \cos(2\pi x)}. \quad (2)$$

This function is periodic, and has more structure than a simple sine-wave¹

We recommend that you use **Python**, as it is very convenient to use, and fast enough for our 1-dimensional problems. For parts (1) and (2) you can also use another language of your choice. For part (3), we supply a Python sub-routine to deal with Legendre polynomials. Without python, you'd be on your own here.

¹In fact, a sine-wave $\sin(2\pi x)$ is one of the basis-functions of the pseudo-spectral method as developed in (2) below. Therefore the seemingly obvious initial guess of $u(x, 0) = \sin(2\pi x)$ would be represented *exactly* by the pseudo-spectral method. The exponential in Eq. (2) levels the playing field, somewhat.

1 Finite Differences

Use a uniform grid with N grid-points

$$x_i = \frac{i}{N}, \quad i = 0, \dots, N - 1, \quad (3)$$

i.e. with grid-spacing $h = (x_{\max} - x_{\min})/N = 1/N$. The solution is represented by the values at the grid-points:

$$u(x, t) \approx u(x_i, t), \quad i = 0, \dots, N - 1. \quad (4)$$

At fixed time, this will be represented by an array of doubles (in Python, a `numpy array`) Discretizing the spatial derivative with central differences, we get

$$\frac{\partial u}{\partial x}(x_i, t) = \frac{u(x_{i+1}, t) - u(x_{i-1}, t)}{2h} + \mathcal{O}(h^2). \quad (5)$$

Ignoring the higher-order corrections $\mathcal{O}(h^2)$, this is a formula to compute $\partial u/\partial x$. For the boundary points ($i = 0$ and $i = N - 1$), one has to wrap around and use points from the opposite end of the interval in Eq. (5). This way, you obtain a set of ordinary differential equations for the values of the solutions at the grid-points:

$$\frac{du(x_i, t)}{dt} = F[u(x_i, t)], \quad (6)$$

where the right-hand-side $F[u(x_i, t)]$ couples the different spatial grid-points with terms like Eq. (5).

Task FD-1: Write a function that computes the right-hand-side of Eq. (6), i.e. it takes an `array` of doubles of length N (representing the grid-point values $u(x_i)$ at time t , and returns an `array` of length N representing F at the grid-points. This function will internally index the `array` `u` according to Eq. (5). It will also internally need to accommodate the periodic boundary conditions. Test this function by feeding it $\sin(2\pi x)$, and check that the result is (approximately) $2\pi \cos(2\pi x)$. Also apply this function to Eq. (2), plot, and ensure by visual inspection that the result seems right.

Writing the array of variables $u(x_i)$ as \mathbf{u} , Eq. (6) becomes a vector equation:

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}[\mathbf{u}]. \quad (7)$$

This is now a set of *ordinary* differential equations for the variables \mathbf{u} . With this viewpoint, called *Method of Lines (MOL)* we can now employ any method to solve ordinary differential equations. Let us now develop a few time-steppers so we have building blocks when we get to the later methods.

1.1 Forward Euler

We begin with the simplest possible time-stepper, the Forward-Euler method. We discretize time,

$$t \rightarrow t^k \equiv k \Delta t, \quad k = 0, 1, \dots \quad (8)$$

We also write the vector of variables at time t^k as

$$\mathbf{u}^k \equiv (u(x_i, t^k))_{i=0, \dots, N-1}. \quad (9)$$

The Forward-Euler method is now

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \Delta t \mathbf{F}[\mathbf{u}^k] \quad (10)$$

Task FD-2: Write a function, called `FE_Step` that performs one step of the Forward-Euler method. I.e. this function takes an `array` `u` representing \mathbf{u}^k , a time-step Δt and a pointer to the function `F[u]` you coded in Task FD-1. It then returns an `array` `u` that represents the variables one time-step later, i.e. \mathbf{u}^{k+1} .

All time-steppers we encounter today are *explicit*, i.e. the spatial derivatives are only computed on already known data. Explicit methods are only stable if the time-step is sufficiently small, roughly,

$$\Delta t \lesssim \text{const } h, \quad (11)$$

where h is the spatial grid-spacing, and the constant is of order unity (its precise value depends on the time-stepper and the spatial discretisation method).

Task FD-3: Write a function, called `Evolve` that calls the single-time-step function as often as needed, to evolve up to a desired final time T_{final} . This function should take a 'Courant Factor' `CF`, and then automatically choose a step-size satisfying $\Delta t < \text{CF} \Delta x_{\text{min}}$. (This will make convergence tests a lot more convenient). A possible calling sequence for this function is given just below. Check `Evolve` with `CF = 1/2`, and by evolving to $t = 1/10$, $t = 1/5$, etc. Observe whether the solution looks as expected (i.e. translated by 1/10, 1/5, etc.)

```
def Evolve(t, T_final, u, F, Tstepper, CF, info):
    """Evolve the evolution equations represented by right-hand-side 'F'
    with time-stepper Tstepper until final time 'T_final'.
        t          - current time
        T_final    - final time
        u          - solution at current time 't'
        F          - right-hand-side of evolution equations
                   calling sequence F(t, u, info)
        Tstepper  - routine that performs one time-step.
```

```

        calling sequence Tstepper(t, u, F, dt, info)
CF      - courant-factor; dt will be chosen s.t. dt < CF*dxmin
info    - named tuple with extra needed information.
        It is assumed that 'info.dxmin' returns the minimal
        grid-spacing. 'info' is also passed into 'TStepper',
        and 'F', should those need extra information.

returns
    t_final, u_final"""

... work here

return t_final, u_final

```

info can be conveniently represented as a `namedtuple`. For example

```

FD_Info_t=collections.namedtuple('FD_Info_t', 'dxmin, x')
x = np.linspace(0., 1., N, endpoint=False)
info=FD_Info_t(dxmin=1./N, x=x)
print("dxmin={}".format(info.dxmin))

```

Just plotting data and looking at it is of course not good enough to ensure the code is correct. The primary means to test for correctness is via a **convergence test**. As the resolution is increased (i.e. $h \rightarrow 0$, and $\Delta t \rightarrow 0$), the solution should approach a limiting solution, and it should approach this solution at the correct rate, given the choices of discretization. The spatial discretization in Eq. (5) is second order accurate.

Task 4: Perform simulations up to $t = 1$ with Forward-Euler, and confirm that the error decays as $\propto N^{-2}$ and $\propto \Delta t$. Because the time-convergence of Forward Euler is so abysmally slow you will have to go to very small time-steps, say Courant factors $CF \sim 2^{-1} \dots 2^{-8}$. Plot the error vs. N and vs. Δt

1.2 Better time-steppers

Clearly, Forward Euler is the limitation, so let's use more quickly converging time-steppers. **Runge-Kutta 2** uses two right-hand-side evaluations, and achieves a time-step error of $O(\Delta t^2)$

$$\mathbf{w}_1 = F[t, \mathbf{u}] \tag{12}$$

$$\mathbf{w}_2 = F[t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t\mathbf{w}_1] \tag{13}$$

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \Delta t\mathbf{w}_2 \tag{14}$$

Runge-Kutta 4 uses four right-hand-side evaluations, and achieves a time-step error of $\mathcal{O}(\Delta t^4)$

$$\mathbf{w}_1 = F(t, \mathbf{u}^k) \tag{15}$$

$$\mathbf{w}_2 = F(t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t\mathbf{w}_1) \tag{16}$$

$$\mathbf{w}_3 = F(t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t\mathbf{w}_2) \tag{17}$$

$$\mathbf{w}_4 = F(t + \Delta t, \mathbf{u}^k + \Delta t\mathbf{w}_3) \tag{18}$$

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \frac{\Delta t}{6} (\mathbf{w}_1 + 2\mathbf{w}_2 + 2\mathbf{w}_3 + \mathbf{w}_4) \tag{19}$$

Task FD-5: Perform simulations up to $t = 1$ with Runge-Kutta 2 and Runge-Kutta 4. Plot the error at $t = 1$ (compared to the analytical solution) vs. time-step for different choices of N . Confirm that the spatial discretization error decays $\propto N^{-2}$. You will find that for any time-step Δt for which the methods are stable, the time-discretization error is already smaller than the spatial discretization error. Therefore, it is difficult to verify that the time-stepping errors decay as $\propto \Delta t^2$ and $\propto \Delta t^4$, respectively. You can postpone this to the next section.

Optional Task FD-6: Implement Eq. (5) with higher-order spatial stencils. For instance, a 5-point stencil that represents $\partial u/\partial x$ with an error $\mathcal{O}(h^4)$.

2 Pseudo-spectral collocation methods

The difficulty with finite-differences is the low order accurate spatial differencing stencil. We could increase the order, but let's go a different route.

Let's expand the solution in basis-functions. Because of periodicity, we will use a Fourier series

$$u(x, t) \approx \sum_{k=0}^{\tilde{N}-1} \tilde{a}_k(t) \cos(2\pi kx) + \tilde{b}_k(t) \sin(2\pi kx) = \operatorname{Re} \sum_{k=0}^{\tilde{N}-1} \tilde{c}_k(t) e^{-2\pi i kx}, \quad (20)$$

where the complex coefficients $\tilde{c}_k = \tilde{a}_k + i\tilde{b}_k$.

As discussed this morning, for smooth, periodic functions a Fourier series converges exponentially in the number of modes, \tilde{N} . Therefore, the derivative

$$\frac{\partial u}{\partial x} = \sum_{k=0}^{\tilde{N}-1} -2\pi k \tilde{a}_k \sin(2\pi kx) + 2\pi k \tilde{b}_k \cos(2\pi kx) \quad (21)$$

will also be exponentially accurate. From Eq. (21), we can read off the spectral coefficients of the Fourier series of the derivative $\partial u / \partial x$:

$$\tilde{a}'_k = 2\pi k \tilde{b}_k, \quad \tilde{b}'_k = -2\pi k \tilde{a}_k, \quad \text{or} \quad \tilde{c}'_k = -2\pi i k \tilde{c}_k. \quad (22)$$

If we can use the expansion Eq. (20), then we know our solution everywhere with high accuracy (i.e. we can interpolate). We can then also use Eq. (22) to compute derivatives with high accuracy. If we can use this to evaluate $F[u]$ in Eq. (7), we will have our spatial discretization error vastly reduced.

It turns out that for a Fourier-series the associated real-space collocation points are equally spaced:

$$x_j = \frac{j}{N}, \quad j = 0, \dots, N-1. \quad (23)$$

This is identical to the finite-difference example above (fundamentally, a periodic problem is translation invariant, and so equal-spacing must be the right choice).

The grid-points x_j are also the corresponding grid-points for Gaussian quadrature, and each grid-point carries the same weight. That means, we can compute the Fourier coefficients as a sum:

$$\tilde{a}_k = \frac{1}{2} \int_0^1 u(x) \cos(2\pi kx) dx \approx \frac{1}{2N} \sum_j u(x_j) \cos(2\pi kx_j) \quad (24)$$

$$\tilde{b}_k = \frac{1}{2} \int_0^1 u(x) \sin(2\pi kx) dx \approx \frac{1}{2N} \sum_j u(x_j) \sin(2\pi kx_j) \quad (25)$$

$$\tilde{c}_k = \frac{1}{2} \int_0^1 u(x) e^{2\pi i kx} dx \approx \frac{1}{2N} \sum_j u(x_j) e^{2\pi i kx_j} = \frac{1}{2N} \sum_j u(x_j) e^{2\pi i jk/N} \quad (26)$$

The factor $1/2$ arises, because the average of $\sin^2(x)$ over a full period is $1/2$. Equation (26) is a discrete Fourier transform, and can be evaluated with built-in functions in python.

Task PS-1: Read the **Python documentation** on fast Fourier transforms, and figure out how precisely you need to call `numpy.fft`-routines to implement Eq. (26). This may involve overall scaling, and it may involve complex conjugation to get the sign-conventions of Eq. (26). Implement a function that takes \mathbf{u} and returns the spectral coefficients \tilde{c}_k . Test by transforming $\sin(4\pi x) - 1/6 \cos(8\pi x)$ to ensure you obtain $\tilde{c}_2 = i$ and $\tilde{c}_4 = -1/6$, with the other terms vanishing. Transform also a constant function, to explore the conventions the FFT-routines use for the $k = 0$ coefficients (they often differ by a factor of 2, which doesn't matter, as long as Task PS-2 works.)

Task PS-2: Evaluating Eq. (20) at the grid-points $u(x_j)$ is the inverse transformation (from spectral to physical space), $\tilde{c}_k \rightarrow u(x_j)$. Show that this can also be written as a fast Fourier transform. Work out the convention, and implement as a function complementing the one of Task PS-1. Test by ensuring that PS-1 followed by PS-2 returns the original data.

Now we're almost done computing derivatives with pseudo-spectral methods. Let's finish this with

Task PS-3: Implement a function that evaluates the right-hand-side of Eq. (7) as follows:

- (1) transform to spectral coefficients \tilde{c}_k ;
- (2) compute the spectral coefficients of the first derivative by Eq. (22)
- (3) transform back to real space values (via task PS-2)

Test by computing the right-hand-side for $\sin(2\pi x)$, checking that you obtain $-2\pi \cos(2\pi x)$. (the minus sign arises because $\partial u / \partial t = -\partial u / \partial x$).

Now we've got everything we need to apply method-of-lines using a pseudo-spectral expansion.

Task PS-4: Perform simulations using the pseudo-spectral right-hand-side from Task PS-4. After initial tests, evolve up to $T_{\text{final}} = 1.02$ with Runge-Kutta 4. (do not use $T_{\text{final}} = 1!$). Compute the difference of $u(x, T_{\text{final}})$ with the analytical solution; plot its L2-norm vs. time-step for different choices of N . Confirm that the time-stepping error decays $\propto \Delta t^4$. Confirm that the spatial discretization error decays exponentially. You will need quite small N to make spatial discretization errors large enough to be noticeable. You will need very small Courant factors to push the time-discretization error small enough to compete with the spatial discretization errors. Use N in the 10's, use CF down to 2^{-8} .

Optional Task PS-5: Why emphasizes Task PS-4 to avoid $T_{\text{final}} = 1$?

3 Discontinuous Galerkin

Continuing with the interval $[x_{\min}, x_{\max}] = [0, 1]$, let's do a discontinuous Galerkin solution on K elements, each with width $h \equiv 1/K$. In each element D^k we will employ a polynomial expansion with the same polynomial expansion order N , for a number of grid-points per element of $N_p = N + 1$. The total number of grid-points, therefore, is $N = K N_p = K(N + 1)$.

3.1 Lagrange Nodal Basis

In each element, we will use an underlying expansion in Lagrange interpolating polynomials $l_j(r)$ based on Legendre-collocation points

$$r_i, \quad i = 0, \dots, N, \quad (27)$$

with $r_0 = -1$ and $r_N = 1$. Given a set of collocation points r_j , the Lagrange interpolating polynomials are defined to be the unique N -th order polynomials which satisfy

$$l_j(r_i) = \delta_{ij}, \quad i, j = 0, \dots, N, \quad (28)$$

i.e. each $l_j(r)$ vanishes at all collocation points, except r_j .

Furthermore, as derived this morning, we define the mass-matrix on the reference element

$$\mathcal{M}_{ij}^{\text{ref}} \equiv \int_{-1}^1 l_i(r) l_j(r) dr, \quad (29)$$

and the stiffness-matrix

$$\mathcal{S}_{ij}^{\text{ref}} \equiv \int_{-1}^1 l_i(r) \frac{dl_j(r)}{dr} dr. \quad (30)$$

It turns out that Eqs. (27), (29) and (30) can be computed reasonably easily by exploiting various relations between Legendre polynomials. Details can be found in Hesthaven & Warburton, Chapters 2 and 3. For expediency, we provide a Python function to compute these quantities, which can be called as

```
r, Minv_ref, MinvS_ref = ReferenceElement(N)
```

This function returns r_j , the inverse $\mathcal{M}^{\text{ref}-1}$, and the matrix $\mathcal{M}^{\text{ref}-1} \mathcal{S}^{\text{ref}}$. It returns the inverses of \mathcal{M}^{ref} , because those are easier to compute, and because those are the ones actually needed below.

3.2 Actual implementation

Task DG-1: The k -th element D^k covers $x \in [x^k, x^{k+1}]$. Derive x^k , assuming each element has the same width. We need a linear mapping from the reference interval $r \in [-1, 1]$ to $[x^k, x^{k+1}]$ (where $k = 0, \dots, K - 1$ numbers the element). Write down this mapping. Map the reference points r_j to the collocation points of the element, x_j^k . Then construct a vector of length $K(N + 1)$ containing all grid-points.

Task DG-2: The mass-matrix in element D^k is given by

$$\mathcal{M}_{ij}^k = \int_{x^k}^{x^{k+1}} l_i^k(x) l_j^k(x) dx, \quad (31)$$

where the Lagrange interpolating polynomials on the k -th domain $l_j^k(x)$ are obtained from the reference $l_j(r)$ by the linear transformation $x \rightarrow r$ you worked out in task DG-1. Express \mathcal{M}^k in terms of \mathcal{M}^{ref} . Similarly, express the stiffness matrix on the k -th element \mathcal{S}^k in terms of \mathcal{S}^{ref} .

Now we are in the position to implement the final evolutionary equation for discontinuous Galerkin, as very briefly sketched in class this morning:

$$\begin{aligned} \frac{d\mathbf{u}_h^k}{dt} = & -\mathcal{M}^{k-1} \mathcal{S}^k \mathbf{f}_h^k \\ & + \left(\mathcal{M}^{k-1}\right)_{iN} \left(f_h^k(x^{k+1}) - f^*(x^{k+1})\right) - \left(\mathcal{M}^{k-1}\right)_{i0} \left(f_h^k(x^k) - f^*(x^k)\right) \end{aligned} \quad (32)$$

with flux on the common boundary at x^k given by

$$f^*(x^k) = a\{\{u\}\} + |a| \frac{1-\alpha}{2} [[u]], \quad (33)$$

$$\{\{u\}\} = \frac{u_h^k(x^k) + u_h^{k+1}(x^k)}{2}, \quad (34)$$

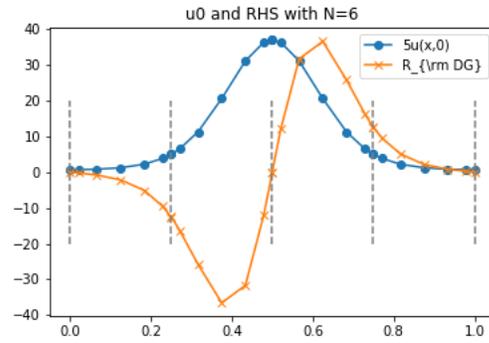
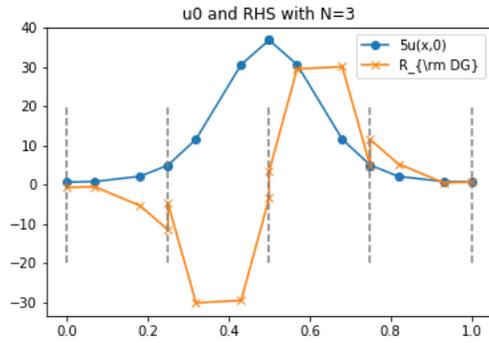
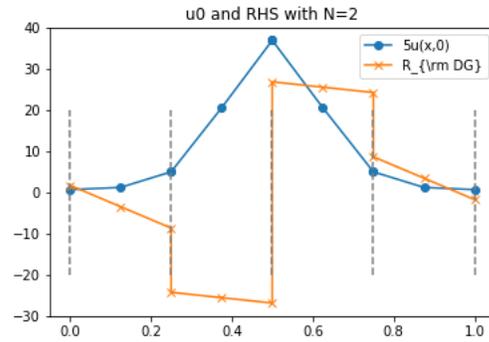
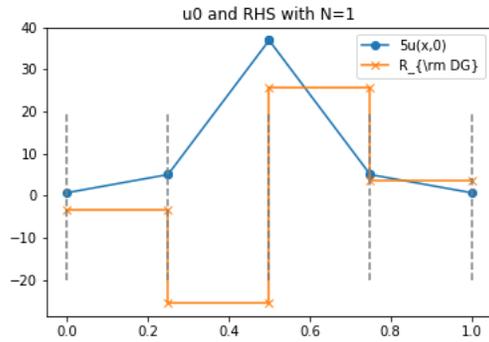
$$[[u]] = u_h^k(x^k) - u_h^{k+1}(x^k), \quad (35)$$

$$(36)$$

where furthermore the advection speed $a = 1$. The parameter α modifies properties of the flux f^* , just set $\alpha = 1/2$.

Task DG-3: in task DG-1, you worked out the vector of all grid-points. This combines all “local” vectors \mathbf{u}_h^k , $k = 0, \dots, K-1$, which appear in Eq. (32). Implement Eq. (32) operating on the vector of all grid-points, $(u^k(x_j^k, t), k = 0, \dots, K-1, j = 0, \dots, N-1)$. The first line of Eq. (32) operates on each element separately. The second line couples neighboring elements. To implement periodic boundary conditions, you will have to connect the left side of the first element with the right side of the last element.

For guidance, for our initial profile Eq. (2), using $K = 4$, and $N = 1, 2, 3, 6$, respectively, Harald obtains the following plots.



Task DG-4: Use the function from Task DG-3 as right-hand-side in Runge-Kutta 4, and thus solve the advection equation with discontinuous Galerkin. Specifically, evolve for one light-crossing time ($T_{\text{final}} = 1$) and compute the difference to the analytic solution. For fixed N , plot the L2-norm of this difference vs the number K of elements. Repeat for different order N , say up to $N = 10$, and check whether you achieve the expected convergence rate $err \sim K^{-N-1}$. Because the convergence order is so high, you will need quite small time-steps to make the Runge-Kutta4 time-discretization error small enough so that you can observe the spatial discretization error. Therefore, experiment with different choices for the Courant factor.