IMPRS GW Astronomy – Computational Physics 2022

Problem Set 1

Tim Dietrich, Harald Pfeiffer

1 Floating Point Operations

As discussed in the lecture, round-off errors and other problems due to floating point operations can cause a lot of headache. Hence, it is of special importance to be careful. Please look at the python code below.

```python
Delta=[]
t=0.
dt=0.1
i=0
for i in range(1000000):
    t=t+dt
    i=i+1
    Delta.append(t-i*dt)
Delta=np.array(Delta)

fig,ax=plt.subplots(1,3,figsize=(12,5))
ax[0].plot(Delta)
ax[0].set_xlim(0,5000)
ax[0].set_ylim(-.5e-10,2e-10)
ax[1].plot(Delta)
ax[1].set_xlim(0,80000)
ax[1].set_ylim(-1e-8,1e-8);
ax[2].plot(Delta);
fig.suptitle('Delta (three different zooms)');
```

Task FP-1: Execute the code yourself, and explain the features in the output. In particular, explain the magnitude of the peaks and their spacing.

Task FP-2: Please change the timestep to 0.099853515625. Execute the code agian and explain your findings. What is did cause the issue in task Task FP-1?

Another potential problem occurs when you have equalities and inequalities. Please have a look at the following C-code.

```c
#include<stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main(int argc, char *argv[])
{
   double a, b;

   [...]


   if (a==b)
     printf("a is equal b \n");
   else
     printf("a is not equal b \n");

   if ((1./a)==(1./b))
     printf("1/a is equal 1/b \n");
   else
     printf("1/a is not equal 1/b \n");

   return 0;
}
```

Task FP-3: Please fill the part [...] in such a way that we obtain the output: 'a is equal b' and '1/a is not equal 1/b'. To compile the code you have to install a C-compiler, e.g., `gcc`, and compile this code.

Another problem discussed in the lecture, was the summation or subtraction of two numbers of very different size. For the purpose of investigating this problem, please have a look at the following code, which is written in yet another programming languange: Fortan.

```
program addNumbers

! This simple program adds two numbers
   implicit none

! Type declarations
   real (kind=4) :: a1, b1, x1, y1, z1
   real (kind=8):: a2, b2, x2, y2, z2
   real (kind=16):: a3, b3, x3, y3, z3


!  Computation with single precision
   a1 = 1.0
   b1 = 1.E-13
   x1 = a1 - sqrt(a1-b1)
   y1 = b1/(a1+sqrt(a1-b1))
   z1 = b1/(a1+sqrt(a1+b1))
   print *, 'With single precision, we obtain ', x1, y1, z1

   !  Computation with double precision
   a2 = 1.0
   b2 = 1.E-13
   x2 = a2 - sqrt(a2-b2)
   y2 = b2/(a2+sqrt(a2-b2))
   z2 = b2/(a2+sqrt(a2+b2))
   print *, 'With double precision, we obtain ', x2, y2, z2

```

```fortran
    !  Computation with quad precision
    a3 = 1.0
    b3 = 1.E-13
    x3 = a3 - sqrt(a3-b3)
    y3 = b3/(a3+sqrt(a3-b3))
    z3 = b3/(a3+sqrt(a3+b3))
    print *, 'With double precision, we obtain ', x3, y3, z3

end program addNumbers
```

To compile this code, you have to install a fortran compiler on your machine, e.g., `gfortran`.

Task FP-4: Please run the code and explain the output. What do you see and how could you explain this. Is it surprising to you that for double and quadrupole precision even a wrong equation provides a more precise answer to the real result?

2 Numerically solving the Advection Equation

In the fifth lecture we will discuss 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**,

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

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 two different numerical methods: (1) finite-differences, and (2) pseudo-spectral methods.

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

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

This function is periodic, and has more structure than a simple sine-wave[1]

You can use any programming language of your choice. We recommend **Python**, as it is convenient to use, provides the fft routines we use below, and is fast enough for our 1-dimensional problems.

[1]In fact, a sine-wave $\sin(2\pi x)$ is one of the basis-functions of the pseudo-spectral methods 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.

2.1 Finite Differences

Use a uniform grid with N grid-points

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

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

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

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

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

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:

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

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:

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}[\mathbf{u}]. \tag{7}$$

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.

2.2 Forward Euler

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

$$t \to t^k \equiv k\,\Delta t, \quad k = 0, 1, ... \tag{8}$$

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

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

The Forward-Euler method is now

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \Delta t\,\mathbf{F}[\mathbf{u}^k] \tag{10}$$

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 $\mathbf{F}[\mathbf{u}]$ 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,

$$\Delta t \lesssim \text{const}\, h, \tag{11}$$

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 $\Delta t < \texttt{CF}\Delta x_{\min}$. (This will make convergence tests a lot more convenient). A possible calling sequence for this function is given just below. Check `Evolve` with $\texttt{CF} = 1/2$, 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.)

Sooner or later, it is useful to standardize on function calling sequences that are general enough for the entire exercise. We recommend you do this now, while working on **FD-3**. Specifically, the instructor has found the following calling interface for `Evolve` useful, which you are recommended to duplicate:

```
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        - A function computing the right-hand-side of the
                  evolution equations. Calling sequence:  F(t, u, info)
       Tstepper - A function that performs one time-step.
                  Calling sequence:  Tstepper(t, u, F, dt, info)
       CF        - courant-factor; 'Evolve' will choose a timestep dt
                   that satisfies  dt < CF*dxmin
       info      - a namedtuple with any additional information needed by
                   'F' or 'Tstepper'.  Specifically, 'info.dxmin' must
                   return the minimal grid-spacing.  'info' is passed
                   into 'TStepper' and 'F'. These two functions can either
                   ignore 'info', or retrieve any information from there
                   that they need.
    returns
      t_final, u_final"""

    ... Your python code goes here ...

    return t_final, u_final
```

The extra variable `info` will become more useful later, when we have more additional information that needs to be passed around[2]. It is convenient to use a `namedtuple` for `info`. 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=", info.dxmin) # dxmin (required to compute dt from CF)
print("x    =", info.x)     # all grid-points (useful for plots)
```

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. $h \to 0$, and $\Delta t \to 0$), 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 FD-4: Perform simulations up to $T = 1$ with Forward-Euler.
Compute the root-mean-square difference to the analytic solution,

$$\texttt{err:}=\left[\frac{1}{N}\sum_{i=0}^{N}(u(x_i,T)-u_{\text{Analytic}}(x_i,T))^2\right]^{1/2}.$$

Plot the error vs. N and vs. Δt. Confirm that the error decays as expected: $\propto N^{-2}$ 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 $\sim 2^{-1} \dots 2^{-8}$.

[2]Do NOT use global variables to pass information around; this is much too error-prone.

2.3 Better time-steppers

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

$$\begin{aligned}
\mathbf{w}_1 &= F[t, \mathbf{u}] && (12)\\
\mathbf{w}_2 &= F[t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t \mathbf{w}_1] && (13)\\
\mathbf{u}^{k+1} &= \mathbf{u}^k + \Delta t\, \mathbf{w}_2 && (14)
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{w}_1 &= F(t, \mathbf{u}^k) && (15)\\
\mathbf{w}_2 &= F(t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t \mathbf{w}_1) && (16)\\
\mathbf{w}_3 &= F(t + 0.5\Delta t, \mathbf{u}^k + 0.5\Delta t \mathbf{w}_2) && (17)\\
\mathbf{w}_4 &= F(t + \Delta t, \mathbf{u}^k + \Delta t \mathbf{w}_3) && (18)\\
\mathbf{u}^{k+1} &= \mathbf{u}^k + \frac{\Delta t}{6}\left(\mathbf{w}_1 + 2\mathbf{w}_2 + 2\mathbf{w}_3 + \mathbf{w}_4\right) && (19)
\end{aligned}$$

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 $\mathcal{O}(h^4)$.

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

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

where the complex coefficients $\tilde{c}_k = \tilde{a}_k + i\tilde{b}_k$.
As discussed in the lecture, for smooth, periodic functions a Fourier series converges exponentially in the number of modes, $\tilde{N}$. Therefore, the derivative

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

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

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

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:

$$x_j = \frac{j}{N}, \quad j = 0, \ldots, N-1. \tag{23}$$

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:

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

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

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

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's `numpy.fft` module. Unfortunately, there are many different conventions for Fourier transforms, and therefore:

Task PS-1: Read the **Python documentation** on fast Fourier transforms, and figure out how precisely you need to call a `numpy.fft`–routine 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).

Task PS-2: The evaluation of Eq. (20) at the grid-points $u(x_j)$ is the inverse transformation (from spectral to physical space), $\tilde{c}_k \to u(x_j)$. Show that this can also be written as a fast Fourier transform. Work out the conventions, 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.

We're almost done computing derivatives with pseudo-spectral methods. To finish:

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 all pieces 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 noticable. 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 $T_{\text{final}} = 1$?