# MATH 934 – BURGERS EQUATION PROJECT

INSTRUCTOR: DR. ADAM LARIOS

Consider the viscous Burgers equation in 1D:

$$\begin{cases} u_t + uu_x = \nu u_{xx} + f, \\ u(x,t_0) = u_0(x). \end{cases} \tag{1}$$

with periodic boundary conditions on an interval of length $L$. Here, $\nu > 0$ is the viscosity, $f = f(x,t)$ is a *force* or *source-term*. The function $u_0$ is the initial data, which we will assume to be a periodic, square-integrate function, i.e., $\int_0^L |u_0(x)|^2\,dx < \infty$ (normally, this won't be a big issue in our computations).

As with the heat equation, if we formally express the solution and the forcing via its Fourier series, we have:

$$u(x,t) = \sum_k \widehat{u}_k(t) e^{ik\frac{2\pi x}{L}}, \qquad f(x,t) = \sum_k \widehat{f}_k(t) e^{ik\frac{2\pi x}{L}}$$

Formally substituting this into equation (1), we obtain the following relationship between the coefficients:

$$\frac{d}{dt}\widehat{u}_k = -\nu k^2 \widehat{u}_k + \widehat{f}_k(t) - \widehat{(uu_x)}_k, \qquad \text{for each } k.$$

Now we see a major difference with the heat equation: Due to the nonlinearity, the ODE for $u_k$ is now coupled to every other Fourier mode. Using the Cauchy product formula, one may find the Fourier coefficients of $uu_x$. The above ODE can that be expressed as

$$\frac{d}{dt}\widehat{u}_k = -\nu k^2 \widehat{u}_k + \widehat{f}_k(t) - \sum_m im\widehat{u}_m \widehat{u}_{k-m}, \qquad \text{for each } k.$$

The sum in the last term is often called a *convolution* (see "Notes on Convolution" at the end of this document. It poses two major problems

- Where should we truncate the sum, so that we don't end up with modes out of our domain?
- For each $k$ (assuming we have $N$ modes), we have to compute $N$ multiplications, for a total of $N^2$ multiplications.

The first problem is already taken care of by the 2/3's dealiasing rule, discussed in class. The second problem is worse: it means that our simulation is going to be very expensive due to having to compute $N^2$ multiplications each time step (actually, even each Runge-Kutta stage!). However, there is a simple solution to it: note that computing $uu_x$ is physical space only requires $N$ multiplications, one at each grid point. This gives us the follow scheme to compute $uu_x$ efficiently.

Scheme to compute $uu_x$

1. Given: `u_hat`, a vector of the Fourier coefficients of $u$.
2. Dealias `u_hat` by zeroing out high wave modes using 2/3's dealiasing rule:
   `u_hat(dealias_modes) = 0;`
3. Compute the Fourier coefficients of (dealiased) $u_x$ by multipling by $ik$ in Fourier space.
4. Compute the inverse Fourier transform (`ifft`) of both vectors to have vectors `u` and `u_x` in physical space.
5. Multiply these vectors to obtain $uu_x$ in physical space.
6. Bring $uu_x$ back to spectral space using the `fft`.

That's it! The above algorithm can be applied as part of each Runge-Kutta stage. It may be helpful to put the computation of the Runge-Kutta stages as separate function. **See my code `heat_rk4.m` for an example of how to make the Runge-Kutta stages into a function.**

**Task 0:** Modify your heat equation code to be able to handle a function $f(x,t)$ on the right-hand side. Alternatively, you can just use my code, `heat_rk4.m` if you like. We will need this forcing in Task 2.

**Task 1:** Write a MATLAB code to solve the 1D Burgers equation using spectral (i.e. Fourier) methods for the spatial component, and Runge-Kutta-4 for the time stepping. You are encouraged to modify your heat equation code (or you can modify my heat equation code if you like). Use the following parameters:

- Initial condition: $u_0(x) = \sin(x)$
- Final time: 5.0.
- Interval length: $L = 2\pi$
- Viscosity: $\nu = 0.102$
- Number of physical gridpoints: $N = 256$
- Forcing: $f(x,t) = 0$
- Use a $\Delta t$ which respects the same CFL as in the heat equation.

Plot the solution in real-time by putting a plot statement in the loop.

Congratulations! You just solved a nonlinear PDE using a computer!

**Task 2:**

Exploration

Use your code, and the same parameters above, except for use $\nu = 0.005$. What do you see? What is going on here is that larger viscosity prevents Burgers equation from developing smaller scales. With higher resolution, the simulation will run just fine.

With no viscosity, Burgers equation will develop infinite small scales until it forms a singularity (it "blows up"). This isn't a numerical artifact; the PDE itself blows up in finite time (for this initial data, at time $T = 1$). Try to run with $\nu = 0.0$. You will have to use a different time-step. Try *advective CFL condition* $\Delta t = \Delta x/\texttt{max}(\texttt{abs}(u_0))$ instead of the usual viscous CFL condition (since $(\Delta x)^2/\nu$ is infinite when $\nu = 0$).

We are left with a problem: Given a viscosity $\nu$, how do we know if our simulation has enough resolution? Conversely, if we are only going to run at a given resolution $N$, what is the smallest viscosity for which we can still trust the simulation?

Experimentation

The answer is that we should be resolving all Fourier modes, say, within machine precision for example. To see the Fourier modes, try something like this in your time loop (comment out any other plots for now):

```
loglog(abs(u_hat(1:(N/2)))/N);
axis([1,N/2,1e-30,1]);
title(sprintf('u(x,%1.3f)',t));
drawnow;
```

The modes corresponding to the highest frequencies (fastest oscillations) will be on the right-side of the plot. If you see the active modes (those above machine precision) touch the right-hand side, you should consider the plot to be under-resolved. By running several simulations, try to find the smallest viscosity for which the simulation is resolved at $N = 64$ grid points.

## Task 3:

Even though our code now seems well-resolved, how do we know if we are getting the correct solution? To test this, we will use the method of manufactured solutions. The idea is, instead of trying to solve the PDE, pick a solution, and find the PDE that it solves. For this exercise, choose a function $w(x,t)$ which is $2\pi$-periodic in $x$. Don't make it especially complex, but not too trivial either. Then, by computing space and time derivatives of your chosen function $w$, set $f$ and $u_0$ as follows:

$$f(x,t) = w_t + w w_x - \nu w_{xx}$$
$$u_0(x) = w(x,0)$$

Then, for these choice of $f$ and $u_0$, $w$ solves Burger's equation (1). You now have the exact solution $w(x,t)$, and you can use your code with these $f$ and $u_0$ choices to try to capture $w$. To see how well you are doing, you can compute the error at each time step by computing the $L^2$ norm. The $L^2$ norm can be computed in either physical space or spectral space, so long as you adjust by the correct factor. This fact is known as Parseval's identity. For an interval of length `L` using `N` grid points, with `dx=L/N`, the $L^2$ norm of the errors can be computed by:

Physical space $L^2$ norm:

```
error = norm(u_exact - u_approx)*sqrt(dx);
```

or, if `u_exact_hat = fft(u_exact)` and `u_approx_hat = fft(u_approx)`,

Spectral space $L^2$ norm:

```
error = norm(u_exact_hat - u_approx_hat)*sqrt(L)/N;
```

This is the error at a fixed time. If you want to assign a single number to the error, you could try looking at the maximum of the errors in time, or the $L^2$ norm of the errors in time, for example.

See if you can make the error "small" in some norm, and smaller still as the resolution increases. Keep in mind that you have two resolutions now: space ($\Delta x$) and time ($\Delta t$). To see how the error depends on $\Delta x$, you can keep $\Delta t$ very small as you vary $\Delta x$. To see how the error depends on $\Delta t$ you have to be a little more careful, since you have to respect the CFL. Try playing around with things to try to see if you can understand how the error depends on resolution.

Have fun, and happy coding!

Notes on Convolution

When multiplying sums, we can organize the terms by the sum of their indices. For example, the term $a_2b_3$ has indices that sum to $2+3=5$. Doing this for a finite sum of $N=8$ terms results in the following computation, with each row being a "convolution" of the coefficients.

$$
\begin{aligned}
& (a_0+a_1+a_2+a_3+a_4+a_5+a_6+a_7)\times(b_0+b_1+b_2+b_3+b_4+b_5+b_6+b_7) \\
= \quad & \ a_0b_0 \\
& + a_0b_1 + a_1b_0 \\
& + a_0b_2 + a_1b_1 + a_2b_0 \\
& + a_0b_3 + a_1b_2 + a_2b_1 + a_3b_2 \\
& + a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 \\
& + a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0 \\
& + a_0b_6 + a_1b_5 + a_2b_4 + a_3b_3 + a_4b_2 + a_5b_1 + a_6b_0 \\
& + a_0b_7 + a_1b_6 + a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2 + a_6b_1 + a_7b_0 \\
& + a_1b_7 + a_2b_6 + a_3b_5 + a_4b_4 + a_5b_3 + a_6b_2 + a_7b_1 \\
& + a_2b_7 + a_3b_6 + a_4b_5 + a_5b_4 + a_6b_3 + a_7b_2 \\
& + a_3b_7 + a_4b_6 + a_5b_5 + a_6b_4 + a_7b_3 \\
& + a_4b_7 + a_5b_6 + a_6b_5 + a_7b_4 \\
& + a_5b_7 + a_6b_6 + a_7b_5 \\
& + a_6b_7 + a_7b_6 \\
& + a_7b_7
\end{aligned}
$$

Note that this is exactly $N^2 = 64$ multiplications.

More generally, for sums of a general size $N$, we can write this as

$$\left(\sum_{k=0}^{N-1} a_k\right)\left(\sum_{k=0}^{N-1} b_k\right) = \sum_{k=0}^{2N-2} c_k$$

where

$$c_k = \sum_{m+n=k} a_n b_m$$

or, more explicitly,

$$c_k = \begin{cases} \displaystyle\sum_{m=0}^{k-1} a_k b_{m-k} & \text{for } k \le N-1, \\ \displaystyle\sum_{m=k-N+1}^{k-1} a_m b_{k-m} & \text{for } k \ge N. \end{cases}$$