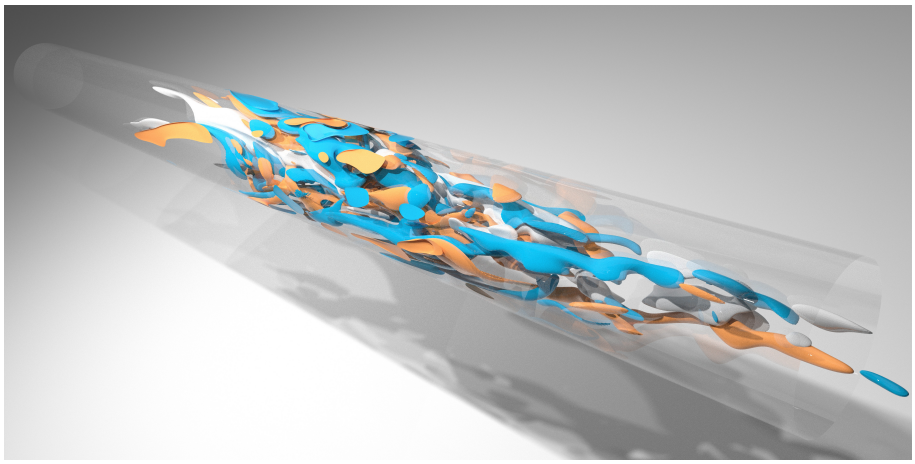
nsPipe-CUDA
Documentation and User Guide

Daniel Morón Montesdeoca



Acknowledgments

The original **nsPipe** code is ancillary to the nsCouette code initially developed by J. M. López *et al.* [Ló+20], who are all here acknowledged. The formulation and many of the methods used in the code were inspired by the openpipeflow code of Dr. Ashley Willis, who is here acknowledged. nsPipe-CUDA is based on the C-CUDA version of the nsCouette code, that was developed by A. Vela-Martín, who is here acknowledged. The project is coordinated by M. Avila who is here gratefully acknowledged. The benchmark of the code was performed by M. Rampp, who is here also acknowledged. This User Guide has been improved after the use and suggestions of P. Keuchel and F. Kranz, who are here also acknowledged.

Contents

1 Introduction

This document includes a detailed description of the `nsPipe-CUDA` code, that is a version of the `CPU` **`nsPipe`** code developed by López *et al.* [Ló+20]. The code numerically integrates the incompressible Navier-Stokes equations (NSE) using a pseudo-spectral method. The equations are written as:

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\,\boldsymbol{u} = -\nabla p + \frac{1}{Re}\boldsymbol{\nabla^2 u} + \boldsymbol{f}_p + \boldsymbol{f}_b, \text{ and } \nabla \cdot \boldsymbol{u} = 0, \tag{1.1}$$

where Re is the Reynolds number, $\boldsymbol{f}_p$ is a force driving the flow, and $\boldsymbol{f}_b$ is any body force acting on the fluid. Bold quantities correspond to vectors.

1.1 How to get the code?

Run the following command in a `Linux` terminal:

```
$ git clone https://github.com/Mordered/nsPipe-GPU.git
```

1.2 Required software to run the code

In order to run the `nsPipe-CUDA` code, a `CUDA`-capable `GPU` device with compute capability 2.0 (or higher), support for double-precision arithmetic and `NVIDIA`'s `CUDA` toolkit are required. The `GPU` code runs in a massively parallel setup with thousands of `GPU` threads and highly efficient memory management. It relies on `cuBLAS` kernels for linear algebra (see documentation here) and Fast-Fourier Transforms using `cuFFT` (see documentation here).

1.2.1 Check if your `GPU` has `CUDA` capabilities

Not all `NVIDIA`'s `GPUs` have `CUDA` capabilities. You can check if your `GPU` has `CUDA` capabilities here.

1.2.2 Install the `CUDA` toolkit

You can download the needed `CUDA` toolkit to use `CUDA` here. It should include the `cuFFT` and `cuBLAS` libraries needed to run the code. Otherwise you can find them here (cuFFT) and here (cuBLAS).

You are also recommended to use `NVIDIA's Nsight Compute` software to profile your `CUDA` codes. Find it here.

In order to compile your code you will need an NVIDIA compiler nvcc that is typically included in the CUDA toolkit, C compilers and make.

1.2.3 On-going development

This GPU version runs on single GPU devices, although there are plans to further develop it for hybrid MPI-GPU applications. Currently the maximum size of the case to be run is limited by the amount of memory of the GPU.

Another limitation of the nsPipe-CUDA code is its dependency on NVIDIA GPU architectures. As part of the development of this code, some functionalities to port the CUDA code to other GPUs have been tested, yielding similar performance capabilities in AMD GPUs.

1.3 How to use this document

If you want to quickly start using the code, and run the default case, you can go to the last chapter in this document, Appendix B, and ignore the rest of the information in this document. In the Appendix B you will find a quick tutorial on how to run and what to expect after running the default case of the nsPipe-CUDA code.

The rest of this document is structured as follows.

- There is a short introduction to pseudo-spectral methods in Chapter 2.
- Find in Chapter 3 a description of the numerical methods used by the nsPipe-CUDA code.
- Chapter 4 includes some basic notions of CUDA, and its architecture.
- In Chapter 5 the code is described in detail, including its modules, and some important features. At the end of the chapter, find a description on how to compile and run a code in a Linux environment.
- In Appendix A find a method to estimate the required grid resolution.

Most of the information included in this document has been extracted from the PhD thesis of Daniel Morón.

2 Basics of pseudo-spectral methods

The code uses a pseudo-spectral method, where two (homogeneous) spatial directions are discretized with a Fourier-Galerkin method, and the third (in-homogeneous) spatial direction is discretized with high-order finite differences. Find in the rest of this section a series of issues one needs to consider when using pseudo-spectral methods. The description is done for a one dimensional variable $f(x)$, defined in the periodic domain $0 \leq x \leq 2\pi$, but can be extended to the two dimensional case used in the code (see eq.(3.2)).

In the Fourier-Galerkin method, variables are treated as a truncated Fourier Series. The one-dimensional variable $f(x)$ is discretized as:

$$f(x) \approx \sum_{k=-N/2}^{N/2-1} \hat{f}_k e^{ikx}, \tag{2.1}$$

being $\hat{f}_k$ the Fourier spectral coefficients and e^{ikx} the Fourier harmonics. The former can be obtained with a Discrete Fourier Transform (DFT):

$$\hat{f}_k(x) = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ikx_j}, \tag{2.2}$$

where $x_j = j\,2\pi/N$. Find more details about Fourier-Galerkin methods in classical text books. The book by Trefethen [Tre00] is here recommended, for a nice introduction to spectral methods. For more detailed explanations you are suggested to read the books by Boyd [Boy01] and Peyret [Pey02].

2.1 Convergence of the Fourier Transform

Invoking Parseval identity:

$$\int_0^{2\pi} f^2 \mathrm{d}x = 2\pi \sum_k^{\infty} \hat{f}_k^2, \tag{2.3}$$

the error ϵ_N of approximating f using a truncated Fourier series can be shown to be:

$$\epsilon_N^2 = \int_0^{2\pi} f^2 \mathrm{d}x - 2\pi \sum_{|k|<N/2} \hat{f}_k^2 = 2\pi \sum_{|k|\geq N/2} \hat{f}_k^2. \tag{2.4}$$

This error is dominated by the spectral coefficient of $|k| \equiv N/2$. As long as f is continuous, this coefficient can be computed as:

$$2\pi \hat{f}_{N/2} = \int_0^{2\pi} f e^{-iNx/2} \mathrm{d}x = \frac{-2}{iN} \left[f\left(2\pi\right) - f\left(0\right) \right] + \frac{2}{iN} \int_0^{2\pi} \frac{\mathrm{d}f}{\mathrm{d}x} e^{-iNx/2} \mathrm{d}x. \tag{2.5}$$

If f is periodic, $f\left(2\pi\right) = f\left(0\right)$. One can apply this formula recursively as long as the derivative $\frac{\mathrm{d}^h f}{\mathrm{d}x^h}$ is continuous and periodic. At the end, the error of the discretization can be shown to be:

$$\epsilon_N \propto \left(N/2\right)^{-H} \left| \frac{\mathrm{d}^H f}{\mathrm{d}x^H} \right|, \tag{2.6}$$

where H is the last continuous and periodic derivative of f. The error thus, depends on how smooth f is [Pey02]. In case f is infinitely differentiable, the convergence of the method is better than any exponent, what is known as infinite convergence or spectral.

2.2 Fast-Fourier transform (FFT)

The Fast Fourier transform (FFT) computes the DFT (and corresponding inverse Fourier transform) in an outstandingly fast and efficient way. The algorithm takes advantage of the symmetries of the Fourier Transform to speed up the computations. Note that, to compute the coefficients $\hat{f}_k$, a simple DFT performs of the order of $\mathcal{O}\left(N^2\right)$ operations, while the FFT needs only $\mathcal{O}\left(2\log_2\left(N\right)N\right)$ operations. Find here the main ideas behind this outstanding speed-up of the calculations.

Let the variable f be discretized in $x_j = \frac{2\pi j}{N}$ discrete points, for $j = 0, 1, ...N-1$, in physical space. The spectral coefficient $\hat{f}_k$ can be computed as:

$$\hat{f}_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j \exp\left(-i\frac{2\pi}{N}kj\right). \tag{2.7}$$

Firstly, the FFT takes advantage of the odd/even symmetry between the points j:

$$\hat{f}_k = \frac{1}{N} \left\{ \sum_{m=0}^{N/2-1} f_{2m} \exp\left[-i\frac{2\pi}{N/2}km\right] + \sum_{m=0}^{N/2-1} f_{2m+1} \exp\left[-i\frac{2\pi}{N/2}k\left(m+\frac{1}{2}\right)\right] \right\}, \tag{2.8}$$

where

$$\exp\left[-i\frac{2\pi}{N/2}k\left(m+\frac{1}{2}\right)\right] = \exp\left[-i\frac{2\pi}{N}k\right] \exp\left[-i\frac{2\pi}{N/2}km\right]. \tag{2.9}$$

Let

$$C_k = \exp\left[-i\frac{2\pi}{N}k\right], \tag{2.10}$$

then

$$\hat{f}_k = \frac{1}{N} \left\{ \sum_{m=0}^{N/2-1} \left(f_{2m} + C_k f_{2m+1}\right) \exp\left[-i\frac{2\pi}{N/2}km\right] \right\}. \tag{2.11}$$

Secondly, the FFT takes advantage of the odd/even symmetry between the modes k. Note that if $k \geq N/2$, one can write $k = N/2 + r$, and find:

$$\exp\left[-i\frac{2\pi}{N/2}m\left(\frac{N}{2}+r\right)\right] = \exp\left[-i2\pi m\right]\exp\left[-i\frac{2\pi}{N/2}mr\right]. \tag{2.12}$$

Note that $\exp\left[-i2\pi m\right] = 1$, which means that for $k \geq N/2$: $\hat{f}_k = \hat{f}_{k-N/2}$.

The FFT takes advantage of this two ideas to construct a recursive algorithm, that goes from computing the first pair of $N = 2$ modes, to $N > 2$. There are more advanced versions of the FFT, that speed up the calculations even when N is not a multiple of 2.

2.3 Pseudo-spectral method

Spectral methods are really convenient for linear problems. The orthogonal properties of the Fourier decomposition allow for a fast parallelization of the problem. However, in the case of non-linear problems one must take care of a particular issue. Find below a one dimensional example.

Let

$$f\left(x\right) = g\left(x\right)h\left(x\right), \tag{2.13}$$

being g and h:

$$g\left(x\right) \approx \sum_{m=-N/2}^{N/2-1} \hat{g}_m e^{imx} \quad \text{and} \quad h\left(x\right) \approx \sum_{n=-N/2}^{N/2-1} \hat{h}_n e^{inx}, \tag{2.14}$$

in spectral space one finds:

$$\hat{f}_k = \sum_{m+n=k} \hat{g}_m \hat{h}_n. \tag{2.15}$$

In order to evaluate the product in equation (2.15) for every k, one needs to perform a total of $\mathcal{O}\left(N^2\right)$ operations. If instead, one performs the product 2.13 in N physical discrete points, the total number of operations is $\mathcal{O}\left(N\right)$. Thus, in the case of spectral methods, the computation of non-linear terms in spectral space, is much more expensive than for methods that consider the variables in physical space.

In order to avoid this problem one can use a hybrid pseudo-spectral algorithm [Boy01]. In this algorithm all the linear operations are performed in spectral space, while the non-linear terms are evaluated in physical space. Every time one needs to compute a nonlinear term, the algorithm performs an IFFT to go from spectral to physical space $\hat{f} \rightarrow f$. Then, it evaluates the nonlinear product in N physical points. After the product is computed, the algorithm performs a FFT to go back from physical to spectral space $f \rightarrow \hat{f}$. If one performs the inverse and direct Fourier transforms using a Fast Fourier transform algorithm, the total number of operations to compute the nonlinear terms is of order $\mathcal{O}\left(\left(1 + 2\log_2\left(N\right)\right)N\right)$, and not the $\mathcal{O}\left(N^2\right)$ operations required for a purely spectral method.

The code uses this pseudo-spectral approach to compute the non-linear terms in the NSE. There is however one last problem one must address when using pseudo-spectral methods, the problem of aliasing.

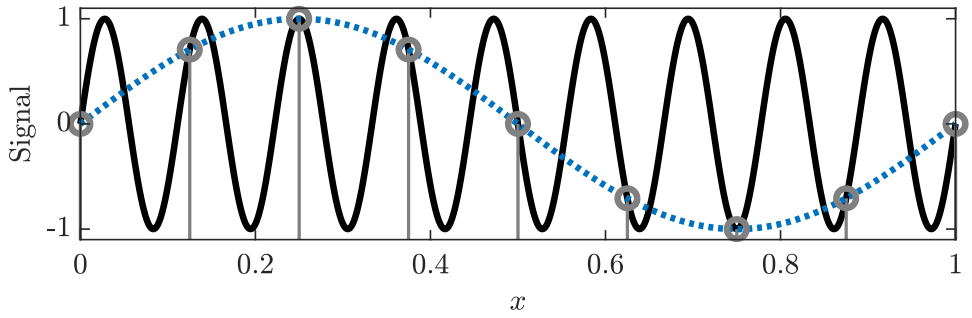


Fig. 2.1.: Example of aliasing error. The solid black line is the signal to be sampled. The vertical grey lines and open circles represent the discrete points at which the original signal is sampled. The dotted blue line represents the resultant sampled signal.

2.3.1 Aliasing

Aliasing is a type of error that takes place when one uses a sampling frequency smaller than the frequencies in the signal being processed. This results in day-to-day phenomena like the eye-perception of the blades of an helicopter rotating in the opposite direction to their actual rotating sense. See an example of aliasing in figure 2.1. In the figure, the signal of angular frequency ω is being sampled with a frequency $\omega/9$. The resultant sampled frequency is then of $\omega/9$, even though it does not exist in the actual data. In general, two trigonometric functions e^{ik_1x} and e^{ik_2x}, sampled at the same discrete points $x_j = \frac{2\pi j}{N}$, appear to be equal when $k_2 - k_1 = m_c N$ for $m_c = 0, \pm 1, \pm 2, ...$

This error is also found in pseudo-spectral methods, as described below for the one dimensional example. Say one wants to compute the product (2.13), but the variables g and h are discretized with an spectral method. One can perform an inverse Fourier transform and obtain the physical value of g and h at N discrete x_j points. Then the product is computed as:

$$f(x_j) = g(x_j)\, h(x_j), \tag{2.16}$$

at N discrete locations x_j.

Let:

$$\hat{f}_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x)\, e^{-ikx_j} \tag{2.17}$$

$$= \frac{1}{N} \sum_{j=0}^{N-1} \left[g(x)\, h(x) \right] e^{-ikx_j} \tag{2.18}$$

$$= \frac{1}{N} \sum_{j=0}^{N-1} \left[\left(\sum_{m=-N/2}^{N/2-1} \hat{g}_m e^{imx_j} \right) \left(\sum_{n=-N/2}^{N/2-1} \hat{h}_n e^{inx_j} \right) \right] e^{-ikx_j}, \tag{2.19}$$

this returns:

$$\hat{f}_k = \sum_{m+n=k} \hat{g}_m \hat{h}_n + \sum_{m+n=k\pm N} \hat{g}_m \hat{h}_n, \tag{2.20}$$

where the second sum is the aliasing error of sampling the original signal with N discrete points. This error will always appear whenever one tries to sample a signal with a discrete number of physical points/sampling frequencies. Pseudo-spectral methods either ignore this error, or avoid its effects by using a technique called padding.

2.3.2 Padding

Before performing the product (2.13), the variables g and h are computed as:

$$g(x) \approx \sum_{m=-J/2}^{J/2-1} \tilde{g}_m e^{imx} \quad \text{and} \quad h(x) \approx \sum_{n=-J/2}^{J/2-1} \tilde{h}_n e^{inx}, \tag{2.21}$$

where $J > N$ and:

$$\tilde{g}_m = \begin{cases} \hat{g}_m & |m| \leq N/2 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \tilde{h}_n = \begin{cases} \hat{h}_n & |n| \leq N/2 \\ 0 & \text{otherwise.} \end{cases} \tag{2.22}$$

The variables g and h are then interpolated to J physical points, and the product (2.13) computed in this J points. The spectral coefficients of f are then computed as:

$$\tilde{f}_k = \frac{1}{J} \sum_{j=0}^{J-1} f_j \exp\left(-i \frac{2\pi}{N} kj\right), \tag{2.23}$$

which can be shown to be equivalent to:

$$\tilde{f}_k = \sum_{m+n=k} \tilde{g}_m \tilde{h}_n + \sum_{m+n=k\pm J} \tilde{g}_m \tilde{h}_n. \tag{2.24}$$

Since the spectral coefficients of f will be padded back so the code only retains all $|k| \leq N/2$, the number of points J must be chosen so, for all $|k| \leq N/2$:

$$|J| > |m + n - k|. \tag{2.25}$$

According to the formulation, the worst case scenario corresponds to $m = n = -N/2$ and $k = N/2 - 1$. Therefore:

$$J \geq \frac{3N}{2}. \tag{2.26}$$

This means that, if everytime one goes from spectral to physical space, one interpolates in J and not on $2N$ physical points, one will push the aliasing error to modes $|k| > N/2$. These modes are in any case ignored by the code, effectively removing the aliasing error from the modes of interest $|k| \leq N/2$. This strategy is usually referred to as the 2/3 dealiasing rule.

2.4 Symmetry of the Fourier transform

The Fourier transform of a real signal results in symmetric Fourier modes. Let $k \in [-N/2, N/2 - 1]$, and $\hat{f}_k$ be the Fourier coefficients that result from the one dimensional Fourier transform of the real signal f. It can be shown that $\hat{f}_k = \hat{f}^{\dagger}_{-k}$ where the dagger stands for complex number conjugate.

In the case of a two-dimensional FT like equation (3.2), one can choose which direction to transform first. This way one can reduce the total number of Fourier modes considered in the discretization to one half, by reducing by one half the modes considered in one of the two directions.

3 Numerical methods in nsPipe-CUDA

In the case of `nsPipe-CUDA` the NSE in eq. (1.1) are integrated in an infinitely long pipe, that is periodic with respect to an axial length L_z. All the variables in the code are non-dimensionalized with respect to the pipe radius R (being the diameter $D = 2R$), with respect to the center-line velocity of the corresponding Hagen-Poiseuille profile U_c (that is equal to two-times the bulk velocity $U_c = 2\,U$) and the fluid density ρ_f. The driving force $\boldsymbol{f}_p = f_p\,(t)\,\boldsymbol{e_z}$, where $\boldsymbol{e_z}$ is the unitary vector in the axial direction, is adjusted at each time step so the bulk velocity complies with a desired bulk velocity U. Here the Reynolds number is defined as $Re = \frac{UD}{\nu} = \frac{U_c R}{\nu}$. The rest of body forces acting on the flow are set equal to zero $\boldsymbol{f}_b = \boldsymbol{0}$ unless stated otherwise. Note that, due to the choice of characteristic magnitudes in the code, the time scale in the code is given in R/U_c units. This means that a time step size of $t = 0.1R/U_c$ corresponds to $t = 0.025D/U$.

3.1 Cylindrical coordinates

The problem here is considered in cylindrical coordinates: (r, θ, z), where r is the radial and at the same time wall normal direction, z is the axial and at the same time stream-wise direction, and θ the azimuthal direction.

In cylindrical coordinates, the NSE take the following form for each velocity component:

$$\begin{aligned}\frac{\partial u_r}{\partial t} + u_r\frac{\partial u_r}{\partial r} + \frac{u_\theta}{r}\frac{\partial u_r}{\partial \theta} - \frac{u_\theta^2}{r} + u_z\frac{\partial u_r}{\partial z} &= -\frac{\partial p}{\partial r} \\ &+ \frac{1}{Re}\left[\frac{\partial}{\partial r}\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r u_r\right)\right) + \frac{1}{r^2}\frac{\partial^2 u_r}{\partial \theta^2} + \frac{\partial^2 u_r}{\partial z^2} - \frac{2}{r^2}\frac{\partial u_\theta}{\partial \theta}\right],\end{aligned}$$

$$\begin{aligned}\frac{\partial u_\theta}{\partial t} + u_r\frac{\partial u_\theta}{\partial r} + \frac{u_\theta}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{u_\theta u_r}{r} + u_z\frac{\partial u_\theta}{\partial z} &= -\frac{1}{r}\frac{\partial p}{\partial \theta} \\ &+ \frac{1}{Re}\left[\frac{\partial}{\partial r}\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r u_\theta\right)\right) + \frac{1}{r^2}\frac{\partial^2 u_\theta}{\partial \theta^2} + \frac{\partial^2 u_\theta}{\partial z^2} + \frac{2}{r^2}\frac{\partial u_r}{\partial \theta}\right], \text{ and}\end{aligned}$$

$$\frac{\partial u_z}{\partial t} + u_r\frac{\partial u_z}{\partial r} + \frac{u_\theta}{r}\frac{\partial u_z}{\partial \theta} + u_z\frac{\partial u_z}{\partial z} = -\frac{\partial p}{\partial z} + \frac{1}{Re}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u_z}{\partial \theta^2} + \frac{\partial^2 u_z}{\partial z^2}\right] + f_p.$$

The continuity equation takes the form:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r u_r\right) + \frac{1}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{\partial u_z}{\partial z} = 0. \tag{3.1}$$

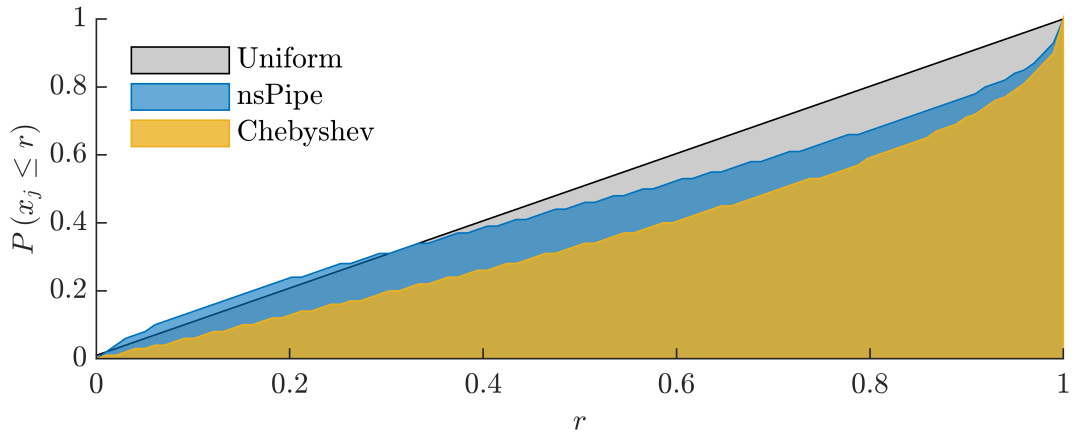


Fig. 3.1.: Cumulative distribution of radial grid points r_j for three different discretizations methods. The vertical axis represents the percentage of the total number of points clustered at locations $r_j \leq r$. In black, the case of uniformly distributed radial points. In yellow a grid with Chebyshev collocation points. In blue the discretization in the code.

3.2 Spatial discretization

The axial and azimuthal directions z and θ are treated as periodic, and are discretized using a Fourier-Galerkin method. Thus, any variable f (being f any of the three velocity components, or the pressure), are discretized as:

$$f(r,\theta,z,t) \approx \sum_{k=-K/2}^{K/2-1} \sum_{m=-M/2}^{M/2-1} \hat{f}_{k,m}(r,t)\, e^{ikk_0 z + imm_0\theta}, \tag{3.2}$$

where, $k = 1, 2, \ldots$ and $m = 1, 2, \ldots$ are the axial and azimuthal wavenumbers, k_0 and m_0 the axial and azimuthal first harmonics, and $\hat{f}_{k,m}$ the spectral coefficients. In the code, the default values are: $m_0 = 1$ and $k_0 = \frac{2\pi}{L_z}$, where L_z is the length of the pipe in radius. Also, the axial and azimuthal number of modes are denoted as $N_z = K/2$ and $N_\theta = M/2$. In the code, the variables are firstly Fourier transformed in z. This means that the number of modes considered is of $m_f = (N_z + 1) \times 2N_\theta$.

The code discretizes the inhomogeneous radial direction r using N_r discrete radial points. The discrete points are not uniformly distributed in the radial direction. They are clustered close the wall as shown in figure 3.1. The code does this by initially distributing the points as Chebyshev collocation points:

$$r_j = \cos\left(\frac{\pi j}{2N_r - 1}\right), \tag{3.3}$$

and later employing a relaxation to define more points away from the wall.

The radial derivatives are computed using high order finite differences. In order to compute the coefficients for the finite difference derivatives, also known as the weights, the strategy described in Fornberg [For88] is used. The method uses a recursive approach based on Lagrange polynomials, and computes the weights for stencils of arbitrary sizes and non-homogeneous grids. Here, only the final recursive expression is given. For a more detailed explanation see Fornberg [For88].

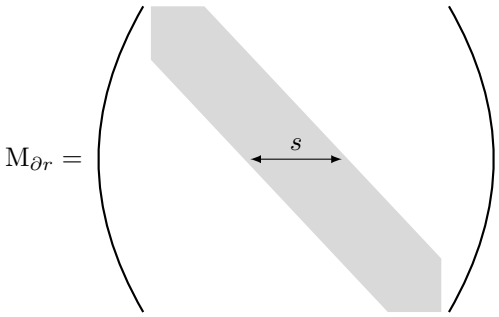


Fig. 3.2.: Banded structure of the radial derivative matrix. Kindly provided by Dr. Plana [PT22].

The coefficient $c_{i,j}^{k}$ corresponds to the weight of the i grid point, in a j stencil, to approximate the derivative of order k. Starting with $c_{0,0}^{0} = 1$, and assuming that any undefined coefficient is 0, any arbitrary coefficient can be computed as:

$$c_{i,j}^{k} = \begin{cases} \left[\frac{\prod_{l=0}^{j-2}(r_{j-1}-r_l)}{\prod_{l=0}^{j-1}(r_j-r_l)}\right]\left(kc_{i-1,j-1}^{k-1} - r_{j-1}c_{i-1,j-1}^{k}\right) & \text{for } i = j \\ \frac{1}{r_j-r_i}\left(r_j c_{i,j-1}^{k} - kc_{i,j-1}^{k-1}\right) & \text{otherwise.} \end{cases} \tag{3.4}$$

The user can select the length of the stencil in the code, being the default one $s = 7$. This results in diagonal banded matrices as shown in figure 3.2 which are computed at the beginning of the simulation, and efficiently stored as sparse. To compute the radial derivatives one only needs to perform a matrix vector multiplication. Say f is a one dimensional variable discretized at j radial locations, and vector $\boldsymbol{f}$ has all these values stored. The vector with the value of the radial derivative of f evaluated at each radial location is then:

$$\frac{\partial \boldsymbol{f}}{\partial r} = \mathrm{M}_{\partial r}\boldsymbol{f}. \tag{3.5}$$

3.3 Numerical integration

3.3.1 Time discretization of the momentum equation

Let the superscript n denote the time step, Δt the time step size, and c be a constant $c \in [0, 1]$; in the code, the NSE are discretized using a Crank-Nicolson method:

$$\begin{aligned} &\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^{n}}{\Delta t} = -c\left(\boldsymbol{u}^{n+1}\cdot\nabla\right)\boldsymbol{u}^{n+1} - (1-c)\left(\boldsymbol{u}^{n}\cdot\nabla\right)\boldsymbol{u}^{n} - \nabla p^{n+1} \\ &+ \frac{1}{Re}\left[c\boldsymbol{\nabla^2}\boldsymbol{u}^{n+1} + (1-c)\,\boldsymbol{\nabla^2}\boldsymbol{u}^{n}\right] + c\boldsymbol{f_p}^{n+1} + (1-c)\,\boldsymbol{f_p}^{n} + c\boldsymbol{f_b}^{n+1} + (1-c)\,\boldsymbol{f_b}^{n}, \end{aligned}$$

where $\boldsymbol{f}_b$ is any body force acting on the flow, $\boldsymbol{f_p} = f_p(t)\,\boldsymbol{e}_z$ the driving pressure gradient of the flow, and

$$\nabla\cdot\boldsymbol{u}^{n} = \nabla\cdot\boldsymbol{u}^{n+1} = 0. \tag{3.6}$$

Due to the presence of the non-linear term $(\boldsymbol{u}^{n+1} \cdot \nabla)\,\boldsymbol{u}^{n+1}$ and pressure ∇p^{n+1} the above equation needs to be iteratively solved in order to obtain the new velocity field $\boldsymbol{u}^{n+1}$. The code uses a predictor-corrector algorithm to do so in a numerically efficient way. In what follows the index k stands for a step of the iteration, or complementary of intermediate steps of the integration.

In what follows:

$$\mathbf{N_u}^* \equiv -(\boldsymbol{u}^* \cdot \nabla)\,\boldsymbol{u}^* + \boldsymbol{f_p}^* + \boldsymbol{f_b}^*, \tag{3.7}$$

$$\mathbf{rhs}^* \equiv \mathbf{N_u}^* + \left[\frac{1}{\Delta t} + \frac{1-c}{Re}\boldsymbol{\nabla^2}\right]\boldsymbol{u}^n, \text{ and} \tag{3.8}$$

$$\mathrm{X} \equiv \left[\frac{1}{\Delta t} - \frac{c}{Re}\boldsymbol{\nabla^2}\right]. \tag{3.9}$$

The asterisk $*$ can either correspond to n, $n+1$ or an intermediate step of the integration k.

3.3.2 Predictor-corrector time-stepping algorithm

The predictor-corrector algorithm integrates the NSE in two steps. In the first step, during the predictor, it computes an intermediate velocity field $\boldsymbol{u}^k$, for $k = 1$, as a guess to the actual velocity field $\boldsymbol{u}^{n+1}$. Then, during the corrector step, it iterates and obtains new $\boldsymbol{u}^{k+1}$ velocity fields to improve the original guess $\boldsymbol{u}^k$. At each iteration step it checks the error between the current $\boldsymbol{u}^{k+1}$ and past $\boldsymbol{u}^k$ guess. If the error is small enough it stops the iteration and sets $\boldsymbol{u}^{n+1} = \boldsymbol{u}^{k+1}$.

Predictor step

The nonlinear term $\mathbf{N_u}^n$ in equation (3.7) is computed using $\boldsymbol{u}^n$. With the non-linear term, the term $\mathbf{rhs}^n$ in equation (3.8) is then computed by setting $* \equiv n$.

With $\mathbf{rhs}^n$, an intermediate pressure p^k is obtained by solving the Poisson equation:

$$\nabla^2 p^k = \nabla \cdot \mathbf{rhs}^n, \tag{3.10}$$

with a Neumann boundary condition at the wall:

$$\left.\frac{\partial p^k}{\partial r}\right|_{r=R} = 0. \tag{3.11}$$

The first guess (prediction) of the new velocity field $\boldsymbol{u}^k$ is then computed by solving the Helmholtz problem:

$$\mathrm{X}\boldsymbol{u}^k = \mathbf{rhs}^n - \nabla p^k, \tag{3.12}$$

for each velocity component. The velocity field is solved so it is zero at the pipe wall. Finally, as shown later in §3.4.2 and 3.4.3 the boundary condition of the pressure at the wall are corrected, and the mass flux imposed (if needed).

The code then iterates on the guessed velocities $\boldsymbol{u}^k$ in the corrector step.

Corrector step

At each iteration of the corrector step, a new nonlinear term ${\mathbf{N_u}}^k$ is computed, eq. (3.7), using the velocity field $\boldsymbol{u}^k$. This nonlinear term is subsequently used to compute:

$$ {\mathbf{N_u}}^{k+1} = c{\mathbf{N_u}}^k + (1-c)\,{\mathbf{N_u}}^n. \tag{3.13} $$

With the new guess on ${\mathbf{N_u}}^{k+1}$ the term $\mathbf{rhs}^{k+1}$ in equation (3.8) is obtained after setting $* \equiv k+1$.

With $\mathbf{rhs}^{k+1}$, a new intermediate pressure p^{k+1} is obtained by solving again the Poisson equation:

$$ \nabla^2 p^{k+1} = \nabla \cdot \mathbf{rhs}^{k+1}, \tag{3.14} $$

with again, the Neumann boundary condition at the wall:

$$ \left.\frac{\partial p^{k+1}}{\partial r}\right|_{r=R} = 0. \tag{3.15} $$

A new velocity field $\boldsymbol{u}^{k+1}$ is then computed by solving the Helmholtz problem:

$$ \mathrm{X}\boldsymbol{u}^{k+1} = \mathbf{rhs}^{k+1} - \nabla p^{k+1}, \tag{3.16} $$

for each velocity component. The velocity field is solved so it is zero at the pipe wall. Afterwards, the boundary condition at the wall and the mass flux are adjusted.

The algorithm then computes the error between the new guess $\boldsymbol{u}^{k+1}$ and the previous one $\boldsymbol{u}^k$ as:

$$ err = \frac{d_2}{d_1}, \tag{3.17} $$

where

$$ d_1 = \max\left\{\max\left[\left(u_z^k\right)^2\right], \max\left[\left(u_\theta^k\right)^2\right], \max\left[\left(u_r^k\right)^2\right]\right\}, \text{ and} \tag{3.18} $$

$$ d_2 = \max\left\{\max\left[\left(u_z^{k+1}-u_z^k\right)^2\right], \max\left[\left(u_\theta^{k+1}-u_\theta^k\right)^2\right], \max\left[\left(u_r^{k+1}-u_r^k\right)^2\right]\right\}. \tag{3.19} $$

If the error is higher or equal to a certain tolerance, $err \geq tol$, the code sets $\boldsymbol{u}^k = \boldsymbol{u}^{k+1}$ and continues iterating. If the error is smaller, then the code sets $\boldsymbol{u}^{n+1} = \boldsymbol{u}^{k+1}$, finishes the iteration and the integration of the current time step $n+1$.

Integration parameters

The user can set the integration parameters c and tol, being the default ones $c = 0.51$ and $tol = 5e-5$.

3.4 Building blocks of the algorithm

In this section, some steps/aspects performed by the code during the integration of the NSE are described in detail. Find a work-flow of the integration algorithm in figure 3.4.

3.4.1 Change of variables

In cylindrical coordinates, the radial and azimuthal velocities are coupled through the vector Laplacian operator:

$$\boldsymbol{\nabla^2 u} \cdot \boldsymbol{e}_r = \nabla^2 u_r - \frac{2}{r^2}\frac{\partial u_\theta}{\partial \theta} - \frac{u_r}{r^2}, \tag{3.20}$$

$$\boldsymbol{\nabla^2 u} \cdot \boldsymbol{e}_\theta = \nabla^2 u_\theta + \frac{2}{r^2}\frac{\partial u_r}{\partial \theta} - \frac{u_\theta}{r^2}, \tag{3.21}$$

$$\boldsymbol{\nabla^2 u} \cdot \boldsymbol{e}_z = \nabla^2 u_z, \tag{3.22}$$

where ∇^2 is the Laplacian operator in cylindrical coordinates defined as:

$$\nabla^2 f = \frac{1}{r}\frac{\partial f}{\partial r} + \frac{\partial^2 f}{\partial r^2} + \frac{1}{r^2}\frac{\partial^2 f}{\partial \theta^2} + \frac{\partial^2 f}{\partial z^2} \tag{3.23}$$

The vector Laplacian can be decoupled in spectral space [Bis+83]:

$$u_\pm = u_r \pm i u_\theta. \tag{3.24}$$

With the use of this transformation, the vector Laplacian operator of $u_\pm$ is:

$$\nabla^2_\pm u_\pm = \nabla^2 u_\pm - \frac{u_\pm}{r^2} \pm \frac{2i}{r^2}\frac{\partial u_\pm}{\partial \theta}. \tag{3.25}$$

The original variables can be computed as:

$$u_r = \frac{1}{2}\left(u_+ + u_-\right), \quad \text{and} \quad u_\theta = -\frac{i}{2}\left(u_+ - u_-\right). \tag{3.26}$$

By performing this variable change, the equations to compute the cross-section velocities can be decoupled, speeding up the numerical integration.

3.4.2 Boundary conditions

In the code formulation the axial and azimuthal directions are treated as periodic directions. In the radial direction boundary conditions must be set at the pipe center-line and pipe wall.

Pipe center-line: parity condition

In order to impose boundary conditions at the pipe center-line, the code uses the method proposed by Trefthen [Tre00]. The idea is to expand the radial coordinate from $r \in (0, 1]$ to $r \in [-1, 1]$. Note that in this new domain there exists an equivalence between data points in the domain since the coordinate:

$$\boldsymbol{r}\left(r, \theta, z\right) \equiv \boldsymbol{r}\left(-r, \pi + \theta, z\right). \tag{3.27}$$

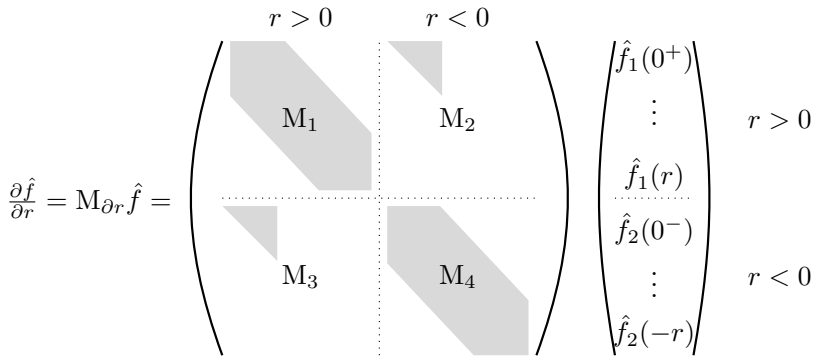


Fig. 3.3.: Decomposition of the radial derivation matrix according to the parity condition, kindly provided by Dr. Plana [PT22].

All the variables in the problem must have a unique value at each physical coordinate, independently of the sense of r. This imposes a condition to the fields also in spectral space as:

$$\begin{array}{ll} \hat{f}_m(r,m) = -\hat{f}_m(-r,m) & \text{if } m \text{ is odd} \\ \hat{f}_m(r,m) = \hat{f}_m(-r,m) & \text{if } m \text{ is even.} \end{array} \tag{3.28}$$

The code takes advantage of this parity conditions, to calculate the radial derivatives close to the pipe center-line.

Let $\hat{\boldsymbol{f}}_m(r)$ be a vector that contains all the discrete spectral coefficients of azimuthal wavenumber m in the $2N_r$ grid, where $r \in [-1,1]$. The derivative of $\hat{\boldsymbol{f}}_m(r)$ can be computed through a matrix-vector multiplication, as shown in figure 3.3, where $\mathrm{M}_{\partial r}$ is now a $2N_r \times 2N_r$ matrix. One can split the vector $\hat{\boldsymbol{f}}_m(r)$ in two parts: $\hat{\boldsymbol{f}}_{m,1}$ for $r > 0$ and $\hat{\boldsymbol{f}}_{m,2}$ for $r < 0$. Conversely the matrix can be splitted in 4 sub-matrices M_1, M_2, M_3 and M_4. From (3.28), $\hat{\boldsymbol{f}}_{m,2} = \pm\hat{\boldsymbol{f}}_{m,1}$, and from the symmetry of the problem $\mathrm{M}_1 + \mathrm{M}_2 = \mathrm{M}_3 + \mathrm{M}_4$. The calculation of the derivative can then be simplified to:

$$\begin{array}{ll} \frac{\partial \hat{f}_m}{\partial r} = (\mathrm{M}_1 - a\mathrm{M}_2)\,\hat{f}_m & \text{if } m \text{ is odd} \\ \frac{\partial \hat{f}_m}{\partial r} = (\mathrm{M}_1 + a\mathrm{M}_2)\,\hat{f}_m & \text{if } m \text{ is even.} \end{array} \tag{3.29}$$

where the constant a depends on the variable f and can be $a = \pm 1$. The axial velocity and pressure fields are even functions in r, as for instance $u_z(r,\theta,z) = u_z(-r,\pi+\theta,z)$. In this case $a = 1$. The radial and azimuthal velocity fields are odd functions in r, as for instance $u_r(r,\theta,z) = -u_r(-r,\pi+\theta,z)$. In this case $a = -1$.

In order to compute higher order derivatives, note that the derivative of an odd function in r is an even function in r, and conversely, the derivative of an even function in r is an odd function in r.

Pipe wall: influence matrix

At the pipe wall the code imposes a zero-velocity boundary condition. Regarding the pressure, at each predictor step, and corrector iteration, a Poisson problem is solved in order to compute p, see eq. (3.10) and (3.14). As mentioned earlier the Poisson equation is solved with Neumann boundary conditions at the wall:

$$\left.\frac{\partial p}{\partial r}\right|_{r=R} = 0. \tag{3.30}$$

But this boundary condition is unrealistic, and the resultant pressure field does not correspond to the actual pressure field of an incompressible flow, specially close to the wall [Rem06].

In order to avoid this error, the code implements an influence matrix method [Wil17]. The idea is to perform an additional step during the time integration, after solving the Helmholtz problem, see eq. (3.12) and (3.16). In this new step a better boundary condition to the pressure field is imposed, and additionally, the incompressible condition in the flow close to the wall improved. Find below a quick description of the method.

Let $\boldsymbol{u}^*$ be the resultant velocity field after solving the Helmohltz problem in eq. (3.12) or eq. (3.16), and p^* the corresponding pressure field, the one obtained in a previous step by solving eq. (3.10) or eq. (3.14) with the unrealistic Neumann boundary condition in eq. (3.30). The velocity field satisfies zero velocity at the wall, but, due to the unrealistic pressure field p^*, it is not guaranteed that it satisfies the incompressibility condition $\nabla \cdot \boldsymbol{u} = 0$ at all points in the flow, and in particular close to the pipe wall. The velocity and pressure fields are corrected with:

$$\boldsymbol{u}^\ddagger = \boldsymbol{u}^* + a_4 \boldsymbol{u}' + \sum_{j=1}^{3} a_j \boldsymbol{u}''_j, \text{ and } p^\ddagger = p^* + \frac{a_4}{\Delta t} p', \tag{3.31}$$

where a_j are tuning parameters that must be determined, so the improved velocity field $\boldsymbol{u}^\ddagger$ complies with the incompressible and zero velocity conditions at the wall:

$$\nabla \cdot \boldsymbol{u}^\ddagger\Big|_{r=R} = 0, \text{ and } \boldsymbol{u}^\ddagger\Big|_{r=R} = 0. \tag{3.32}$$

The auxiliary velocity fields $\boldsymbol{u}''_j$ are computed by solving:

$$\mathrm{X}\,\boldsymbol{u}''_j = 0, \tag{3.33}$$

with different boundary conditions:

$$\begin{array}{l} (u''_+, u''_-, u''_z)\big|_{r=R} = (1,0,0)\,, \quad \text{if } j = 1, \\ (u''_+, u''_-, u''_z)\big|_{r=R} = (0,1,0)\,, \quad \text{if } j = 2, \\ (u''_+, u''_-, u''_z)\big|_{r=R} = (0,0,i)\,, \quad \text{if } j = 3. \end{array} \tag{3.34}$$

The pseudo-pressure field p' is computed by solving the Poisson equation:

$$\nabla^2 p' = 0, \tag{3.35}$$

with the boundary condition:

$$\frac{\partial p'}{\partial r}\Big|_{r=R} = 1. \tag{3.36}$$

And the corresponding velocity as:

$$\boldsymbol{u}' = -\nabla p'. \tag{3.37}$$

As a side note, if one multiplies the above equation by X, one can obtain:

$$\mathrm{X}\,\boldsymbol{u}' = -\mathrm{X}\,(\nabla p') = c\frac{\nabla^2}{Re}(\nabla p') - \frac{\nabla p'}{\Delta t} = c\frac{\nabla}{Re}\left(\nabla^2 p'\right) - \frac{\nabla p'}{\Delta t}, \tag{3.38}$$

and, since $\nabla^2 p' = 0$, then:

$$\mathrm{X}\boldsymbol{u}' = -\frac{\nabla p'}{\Delta t}. \tag{3.39}$$

By imposing the boundary condition in eq. (3.32), to the improved velocity in eq. (3.31), one finds:

$$0 = \boldsymbol{u}^*|_{r=R} + a_4\, \boldsymbol{u}'|_{r=R} + \sum_{j=1}^{3} a_j\, \boldsymbol{u}_j''\Big|_{r=R}, \text{ and} \tag{3.40}$$

$$0 = (\nabla \cdot \boldsymbol{u}^*)_{r=R} + a_4\, (\nabla \cdot \boldsymbol{u}')_{r=R} + \sum_{j=1}^{3} a_j \left(\nabla \cdot \boldsymbol{u}_j''\right)_{r=R}, \tag{3.41}$$

that corresponds to a system of 4 equations with 4 unknowns, a_j for $j = 1, 2, 3, 4$. This system can be written as:

$$A\boldsymbol{a} = \boldsymbol{g} \rightarrow \begin{bmatrix} 1 & 0 & 0 & u'_+ \\ 0 & 1 & 0 & u'_- \\ 0 & 0 & i & u'_z \\ \nabla \cdot \boldsymbol{u}_1'' & \nabla \cdot \boldsymbol{u}_2'' & \nabla \cdot \boldsymbol{u}_3'' & \nabla \cdot \boldsymbol{u}_4'' \end{bmatrix}_{r=R} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} -u_+^* \\ -u_-^* \\ -u_z^* \\ -\nabla \cdot \boldsymbol{u}^* \end{bmatrix}_{r=R} \tag{3.42}$$

where A is the so-called influence matrix. By inverting the influence matrix, one can obtain the coefficients a_j, that allow $\boldsymbol{u}^\ddagger$ to satisfy the zero velocity and incompressible conditions at the wall at each time step. Note that at points different to the wall, the velocity $\boldsymbol{u}^\ddagger$ can still have a certain divergence $\left(\nabla \cdot \boldsymbol{u}^\ddagger\right)\Big|_{r<R} \neq 0$.

As a final note, if one multiplies equation (3.31) by X one obtains:

$$\mathrm{X}\boldsymbol{u}^\ddagger = \mathrm{X}\boldsymbol{u}^* + a_4 \mathrm{X}\boldsymbol{u}' + \sum_{j=1}^{3} a_j \mathrm{X}\boldsymbol{u}_j'' = \mathrm{X}\boldsymbol{u}^* + a_4 \mathrm{X}\boldsymbol{u}'. \tag{3.43}$$

If $* = k + 1$, and invoking eq. (3.39) one finds an evolution equation of $\boldsymbol{u}^\ddagger$ as:

$$\mathrm{X}\boldsymbol{u}^\ddagger = \mathbf{rhs}^{k+1} - \nabla p^{k+1} - a_4 \frac{\nabla p'}{\Delta t} = \mathbf{rhs}^{k+1} - \nabla p^\ddagger, \tag{3.44}$$

which represents a complementary equation to eq. (3.16), and where $p^\ddagger$ is the actual pressure field in the flow.

3.4.3 Driving of the flow

In the code the flow is driven with a constant bulk velocity constraint. In order to impose a predefined bulk velocity, the code takes advantage of the iterations in the predictor and the corrector steps to adjust the bulk velocity to machine precision. It does so by following several steps.

1. In the initialization phase of the code, the auxiliary axial velocity profile $u_{aux}\,(r)$ is computed as the resultant velocity from an unit axial impulse:

$$\mathrm{X}\,[0, 0, u_{aux}] = [0, 0, 1]\,. \tag{3.45}$$

Together with this velocity profile, an auxiliary bulk velocity u_a is computed as:

$$u_a = \langle u_{aux} \rangle_V = \frac{1}{L_z \pi R^2} \int_0^{L_z} \int_0^{2\pi} \int_0^R u_{aux} r \mathrm{d}r \mathrm{d}\theta \mathrm{d}z. \tag{3.46}$$

Note that here, and in the rest of the document, angled brackets denote averaging over one or more spatial directions, over the whole pipe volume V or time t.

2. Before solving the Helmholtz problems in equations (3.12) and (3.16), the code computes a guess of the driving pressure gradient as the balance to the mean viscous stresses at the wall:

$$\tilde{f}_p(t) = \frac{-1}{L_z \pi R^2} \int_0^{L_z} \int_0^{2\pi} \int_0^R \frac{1}{Re} \frac{\partial u_z^*}{\partial r}\bigg|_{r=R} r \mathrm{d}r \mathrm{d}\theta \mathrm{d}z, \tag{3.47}$$

where $* = k$ or $* = k+1$. It plugs this guess of the driving force in the term $\mathbf{N_u}^*$ computed in eq. (3.7).

3. After solving the Helmholtz problem, and adjusting the boundary conditions of $\boldsymbol{u}^*$, the current bulk velocity u_c is computed as:

$$u_c = \frac{1}{L_z \pi R^2} \int_0^{L_z} \int_0^{2\pi} \int_0^R u_z^\ddagger r \mathrm{d}r \mathrm{d}\theta \mathrm{d}z. \tag{3.48}$$

Note that at this stage, probably $u_c \neq U$.

4. A correction to the bulk velocity is then computed as:

$$\beta^* = \frac{U - u_c}{u_a}. \tag{3.49}$$

And the axial velocity at $* = k$ or $* = k+1$ is corrected as:

$$u_z^* = u_z^\ddagger + \beta^* \cdot u_{aux}(r). \tag{3.50}$$

As a final note, if one multiplies equation (3.50) by X, and by setting $* = k+1$, one arrives at:

$$\mathrm{X}\boldsymbol{u}^{k+1} = \mathrm{X}\boldsymbol{u}^\ddagger + \beta^{k+1}\boldsymbol{e}_z. \tag{3.51}$$

By plugging equation (3.44) one finds

$$\mathrm{X}\boldsymbol{u}^{k+1} = \mathbf{rhs}^{k+1} - \nabla p^\ddagger + \beta^{k+1}\boldsymbol{e}_z, \tag{3.52}$$

which represents a complementary equation to eq. (3.16). After invoking the definition of **rhs** in equation (3.8), it is easy to check that the actual pressure gradient driving the flow is equal to:

$$f_p(t) = \tilde{f}_p(t) + \beta(t), \tag{3.53}$$

and it is indirectly computed by the code at each discrete time step.

3.4.4 Solving the Poisson and Helmholtz problems

It can be shown that, for each Fourier mode, the solution of the discretized Poisson and Helmholtz problem reduces to a one-dimensional (radial) problem, which is solved by inverting a diagonal matrix, similar to the matrix shown in fig. 3.2. For the case of the Helmholtz problem, by invoking the

N_r	M_θ	M_z	Memory GPU	GPU	2x36	4x36	8x36	16x36
48	96	768	2GB	8.65	40.15	22.37	14.16	8.923
96	240	5760	30GB	397.45	2302.66	1230.55	649.68	372.28

Tab. 3.1.: Performance of the GPU code compared with the performance of the CPU code. The code has been tested for two different grids, described in the three first columns, with the number of physical radial N_r, azimuthal M_θ and axial M_z points. The third column denotes the total memory required to run the code in the GPU, for each grid size. The rest of the columns denote the computing time of performing a time step in ms, averaged over 1000 time steps. The GPU used is an A100 80GB (fifth column). The CPU code (sixth to ninth columns) is run with different numbers of Xeon 8360Y processors. A single processor has 36 cores.

change of variables explained in §3.4.1, the three velocity components can be decoupled. Therefore, instead of one, three Helmholtz problems can be solved, one for each velocity component, to obtain the velocity field of the corresponding Fourier mode. In the code, the Poisson and Helmholtz problems are solved with an in-house programmed LU decomposition.

3.5 Parallelization in CUDA

Every time a linear operation is performed by the code, that does not correspond to a solution of the Poisson or Helmholtz problem, each thread of the GPU card is assigned to one Fourier mode and one radial position. That thread performs the linear operation on the mode and the radial location, independently of the rest.

The same is done for the computation of the nonlinear term $\mathbf{N_u}^*$ in eq. (3.7). Each thread is assigned a discrete point in the domain, and computes $\mathbf{N_u}^*$ at that point and independently to the rest.

In order to solve the Poisson and Helmholtz problems, one thread is assigned to each Fourier mode. The thread then reads all the radial points of that particular mode, and inverts the corresponding matrix.

See an introduction to threads and blocks in CUDA in chapter 4.

3.6 Performance and validation of the code

The performance of the code was tested by comparing the computing time per time step of the GPU code with the computing time of the original **nsPipe** CPU code for different grids and number of CPUs. Note that all the cases have the same stencil s of radial points. See the main results of this comparison in table 3.1. There is an outstanding speed-up of the GPU code compared with the original code run in 72, 144 or 288 cores. Only by running the CPU code in 566 or more cores, both codes reach the same level of performance. Note that, this will require at least 16 CPUs or more, while the CUDA code needs a single GPU. In sum, the GPU code results in a much faster code than the CPU code.

3.6.1 Code validation

In order to validate the code, lifetimes statistics of puffs in SSPF at $Re = 1850$ are computed with the GPU code.

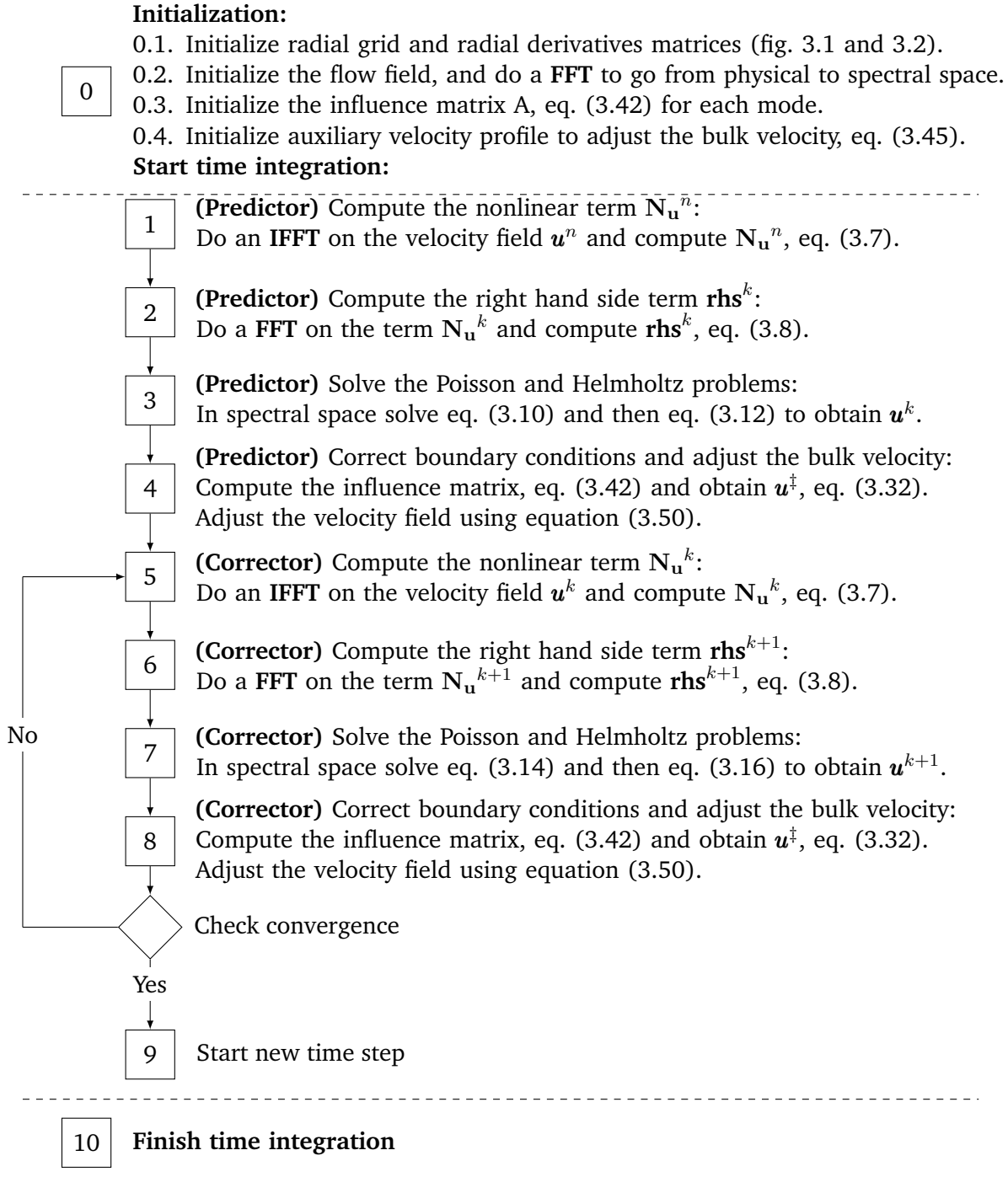


Fig. 3.4.: Description of the code and the time-stepping algorithm.

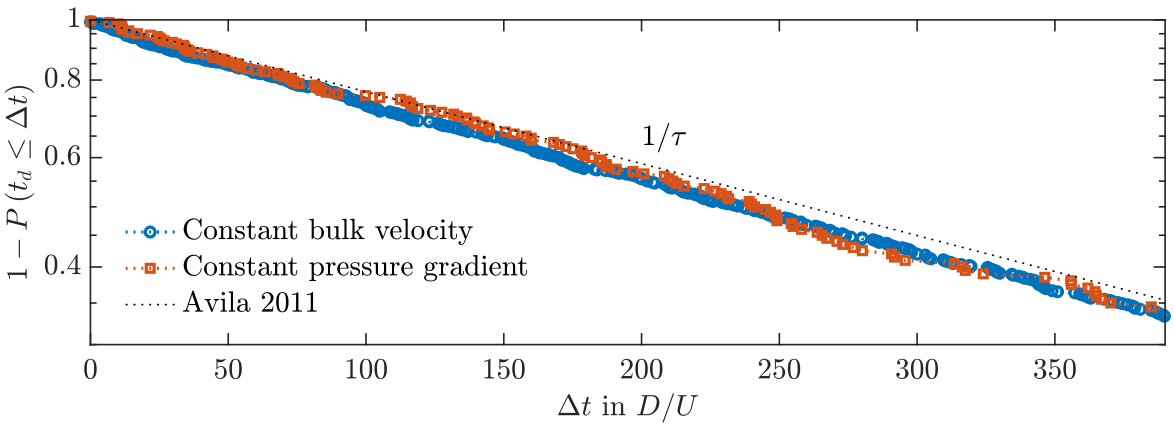


Fig. 3.5.: Lifetimes statistics of puffs in DNS using the new `C-CUDA` code in $L_z = 50D$ long pipes at $Re = 1850$. The vertical axis represents one minus the cumulative distribution function of puff decay after a time step size Δt. In blue (red) and circles (squares) cases with a constant bulk velocity (constant pressure gradient). The dotted line corresponds to the value of the exponential distribution proposed by Avila et al. [Avi+11], $1 - P_q \approx \exp(-\Delta t/\tau)$, where $\tau = \exp(\exp(0.005556 \cdot Re - 8.499))$, that fits the experimental data of Hof *et al.* [Hof+08].

See in figure 3.5 the lifetime distribution of puff decay. One line corresponds to simulations with a bulk velocity constraint (blue and circles), and the other (red and squares) with a constant pressure gradient constraint. In this second case, the pressure gradient is set so, the bulk velocity is always close to U. The simulations are capped to a maximum run time of $t \leq 400D/U$. The two distributions are compared with the exponential distribution derived by Avila et al. [Avi+11], $1 - P_q \approx \exp(-\Delta t/\tau)$, where $\tau = \exp(\exp(0.005556 \cdot Re - 8.499))$, that fits the experimental data of Hof *et al.* [Hof+08]. As seen in the figure, the distributions produced by the code matches the experimental fit, independently of the method used to drive the flow.

4 Basics of CUDA

While CPUs are designed, and optimized for reducing the latency of operations, GPUs are optimized for the parallel execution of operations. CUDA (Compute Unified Device Architecture) is a software layer, that allows for the development of codes to be run concurrently in CPUs and GPUs. The codes perform some tasks in a CPU and some tasks in a GPU, taking advantage of the strengths of each architecture when needed. Currently, CUDA allows for the use of C, C++ or FORTRAN languages. In the case of the nsPipe-CUDA, the language used is C. This means that, except for some functions of CUDA, the majority of the code is written in C.

4.1 Host and device

Perhaps the most important idea in CUDA is that the code is run both in: a CPU (what is called the host) and a GPU (what is called the device). In CUDA programming, during the execution of the code, the majority of operations are performed by the CPU. However, the GPU is used whenever a special type of function known as kernel is called (more about this below).

In CUDA the code is executed in the CPU unless a kernel is used. In the case of nsPipe-CUDA, the code in the CPU is in charge of performing all the serial operations. This includes, the execution of the main work-flow of the code and all the input/output operations, among others. The CPU is also in charge of exchanging memory and information with the GPU. During the execution of the code, memory is allocated in the host (CPU) (from an input file for instance) and from the CPU it is copied to the device (GPU). The GPU can only work with data that has been allocated in it (more on this later).

In CUDA, the GPU only performs operations when a special type of function, a kernel, is called (or when the user uses CUDA built-in functions). The smallest executing unit in a GPU is a thread. Everytime a kernel is called, the code spawns a number of threads in the GPU, that, in parallel, execute the same instructions written in the kernel. This philosophy is called SIMT (Single Instruction Multiple Thread). The user can select the number of threads to spawn during each kernel operation. Threads are organized in blocks, and blocks in grids (more on this below).

When compiling the code with nvcc, the compiler assigns the sections of the code corresponding to the CPU to the C compiler of the CPU, and the rest to the GPU. Find more detailed information in this short video.

4.2 Kernels

Find here a description of the syntax of kernel functions, and the way to call them.

4.2.1 Launching a kernel

The syntax to launch a kernel is almost equal to the syntax to call a C function. See an example below:

```
name_of_kernel<<<grid_size,block_size>>>(input1,input2);
```

Like in any C function the first element `name_of_kernel` is the name of the kernel to be called. Note that the kernel must be declared before using it, like any other C function. The last part of the call `(input1,input2)` are the inputs. Note that, if `input1` and `input2` are pointers, the data linked to the pointer must be allocated in the device. The inputs `«<grid_size,block_size»>` set the grid and block dimensions of threads.

4.2.2 Blocks and grids

Every time a kernel is called, the GPU spawns a given number of threads that will perform the same instructions, the ones declared in the kernel. The threads are grouped first in blocks and blocks are then grouped in grids. When calling a kernel the user must set the size of these blocks and grids.

Blocks and grids are three dimensional entities. That means that a block structures the threads inside it in three dimensions (x,y,z) like a 3D tensor would. Conversely grids do the same with blocks. The user is free to distribute the threads inside a block as one sees fit, as long as the total number of threads inside the block is below the limit of the corresponding GPU. Every GPU has a limit of total number of threads per block and of blocks per grid. The user can easily check these limitations having a look at the specs of his/her GPU.

Following the example shown above, the variables `grid_size,block_size` are declared and initialized as:

```
// Declare the variables with the dimensions of the kernel
dim3 block_size,grid_size;
// Initialize the dimensions of the blocks and grids.
block_size.x=3; block_size.y=3; block_size.z=3;
grid_size.x =3; grid_size.y =3; grid_size.z =3;
```

Note that the dimensions must be integer numbers. In this example, the user has set-up three-dimensional blocks of $3 \times 3 \times 3$ threads, in grids of $3 \times 3 \times 3$ blocks. In total, the kernel will be executed in 3^6 different threads.

4.2.3 Syntax of a kernel

Kernels must be defined before calling them. In order to tell the compiler that a given function must be treated as a kernel, the user must write the prefix `__global__` in front of the rest of the declaration of the kernel. The rest of the declaration follows exactly the same syntax a C function would.

Following the above example, at the beginning of the CUDA file `*.cu`, one would write:

```
// Actual kernel
__global__ void name_of_kernel(double* input1,double*input2){
// Do the operations
...
// Finish and exit the kernel
}
```

4.2.4 Inputs to a kernel

The kernel accepts all the types of inputs a normal `C` function would. Note however, that in the case of pointers, the kernel only accepts pointers allocated in the device (`GPU`).

4.2.5 Indexing of threads

The above kernel will be executed in all the threads defined by `grid_size,block_size`. The threads have information of their current position in the block and the grid. For that, `CUDA` includes a set of built-in variables that can be cast inside the kernel, and allow the user to know the position of a given thread among all blocks and grids. These are:

```
// Built in variables to get the dimensions of the grid in each x,y,z direction:
dim3 gridDim; int gridDim.x, gridDim.y, gridDim.z;

// Built in variables to get the index of the blocks in each x,y,z direction:
dim3 blockIdx; int blockIdx.x, blockIdx.y, blockIdx.z;

// Built in variables to get the dimensions of a block in each x,y,z direction:
dim3 blockDim; int blockDim.x, blockDim.y, blockDim.z;

// Built in variables to get the index of the blocks in each x,y,z direction:
dim3 threadIdx; int threadIdx.x, threadIdx.y, threadIdx.z;
```

4.2.6 Example of a kernel call

See here an example of a `CUDA` kernel that sums the $N = 8$ elements of two vectors of doubles `a` and `b`, and saves it to another vector `c`. Note that, before calling the kernel, the data of `a` and `b` must be in the device, and the memory required for vector `c` allocated there.

For the sake of simplicity, we call the kernel using blocks and grids only on the x direction. Somewhere in the main code one would write:

```
...
// Declare the variables with the dimensions of the kernel
dim3 block_size,grid_size;
// Initialize the dimensions of the blocks and grids.
block_size.x=4;
grid_size.x =2;
// Call the kernel
sum_vectors_k<<<grid_size,block_size>>>(c,a,b,N);
...
```

By doing this, exactly 8 threads will be spawned when executing the kernel. Since the rest of the blocks and grids dimensions (y,z) are not specified, they are set to a default value of 1. As inputs, the kernel needs the three vectors, and the dimensions of the vectors N:

```
// Actual kernel
__global__ void sum_vectors_k(double* c,double* a,double *b, int N){
// First identify the index of the thread executing this kernel
// The index of the thread in the whole grid is the sum of:
// + The index inside the block
// + The index of the corresponding block
int i=threadIdx.x + blockIdx.x*blockDim.x;
// Stop the execution of the thread if i>N-1
if(i>=N){return;}
// Otherwise declare auxiliary variables
double at,bt,ct;
// Read the value of a and b and save it to at, bt
at=a[i]; bt=b[i];
// Compute the sum
```

```
ct=at+bt;
// Save the sum to the new vector c
c[i]=ct;
}
```

When executed, each thread in the kernel will read one element of vectors `a` and `b`, save those values to the variables `at` and `bt` that are unique for each thread. It will then compute the sum of just one element and finally save it in the vector `c`. It is a good practice to check that all the threads executing the kernel are actually needed. In this example, additional threads from the N elements of the vector, are not required to perform the operation. In the example above, this is enforced by finishing the execution of any thread whose index is bigger than $N \geq 8$.

More on kernels, threads, blocks and grids in this short video.

4.3 Memory model in CUDA

Both the CPU and GPU in CUDA have separated memories during the execution of the code. Additionally the memory in the GPU can be further partitioned when a kernel is called. See a brief description of the memory management in a CUDA code in what follows. For more detailed information please check this video.

4.3.1 Global memory

The memory in the device is usually referred to as Global memory. The user can allocate memory in the device using the built-in CUDA function:

```
// Allocate memory on the device
cudaMalloc(&u,N*sizeof(double));
```

In this example the variable `u`,which is a vector of doubles of length N, is allocated in the device.

The global memory can be accessed by all the threads in a kernel. However it tends to be slow, and there can be conflicts between threads when different threads try to access the same spaces in the global memory.

The memory in the device can be de-allocated with:

```
cudaFree(u);
```

4.3.2 Copy memory between host and device

During some parts of the execution, the user may want to copy memory from the host to the device, from the device to the host or even from the device to the device itself. CUDA provides the user with a built-in function that allows to do this:

```
// Copy from host to device
cudaMemcpy(u,u_h,N*sizeof(double),cudaMemcpyHostToDevice)
// Copy from device to host
cudaMemcpy(u_h,u,N*sizeof(double),cudaMemcpyDeviceToHost)
// Copy from device to device
cudaMemcpy(uu,u,N*sizeof(double),cudaMemcpyDeviceToDevice)
```

Here `u` and `uu` are variables allocated in the device, and `u_h` a variable in the host. They all represent vectors of doubles with N elements.

4.3.3 Shared memory and memory in the thread

Each thread in a kernel, when spawned, has a certain amount of memory, where variables can be allocated. The total amount of memory depends on the GPU card. Although it is limited, this memory is extremely fast.

Between threads in the same block, the user can also define a shared memory. This memory is initialized inside the kernel during execution, and is a method by which threads, that for the rest of the time execute instructions independently, can communicate among themselves. The shared memory is faster than the global one.

5 Architecture and how to use the nsPipe-CUDA code

Find in this chapter a description of the modules of the code, followed by a short description of some additional capabilities of the code. Finally an example is included on how to set-up and run a nsPipe-CUDA simulation. The reader is expected to have some experience with C programming language.

5.1 Files in the nsPipe-CUDA code

In the folder Main_code/src/, one finds several CUDA files *.cu and a header file head.h. The main file is main.cu. The rest of CUDA files are modules to this main file, and are named as mod_*.cu. Note that all the CUDA files use the same header head.h.

All the module files share the same internal structure. First, the kernels used in the module are declared. Second, the internal variables/functions the module may have. Note that the information of the variables/functions declared in a module is only accessible by the module itself. Third, the module has its "Main Functions". These are functions that are called by other modules or the main main.cu CUDA file, and not by the module itself. Finally the module has its "Auxiliary functions" or functions that are internal to the module.

Find below a list and description of the most important files in the code.

5.1.1 head.h

The header file is splitted in three sections:

head.h: libraries

The code needs several C and CUDA libraries to run:

- Standard C libraries for math, input/output operations and operating on the time of the system: <math.h>,<stdio.h>,<stdlib.h>,<time.h>,<sys/time.h>.
- Libraries for H5 output <hdf5.h>,<hdf5_hl.h>.
- CUDA libraries. <cublas_v2.h> has functions to perform algebraic operations in CUDA. <cufft.h> has functions to set and perform FFT in CUDA.

`head.h`: Define parameters

In the second section the user can select the constant parameters of the simulation.

- Grid parameters:
  - `NR` is the total number of radial points.
  - `NT` the total number of azimuthal Fourier modes.
  - `NZ` half (plus 1) axial Fourier modes.
  - `LZ` is the length of the pipe in radius R.
  - `LT` is the periodic domain of the azimuthal direction. Normally it is set to `LT=`2π, so the azimuthal direction is periodic in the whole circumference.
  - `NTP` is the total number of azimuthal points in physical space.
  - `NZP` is half the total number of axial points in physical space.
- Flow parameters: `Re`, Reynolds number; `A_P` the magnitude of the initial perturbation to the flow field in $2U$.
- Input and output parameters (more on this below).
- Integrator parameters:
  - `sten`: sets the stencil of the radial finite differences.
  - `dt`: size of the time step in $R/(2U)$ units.
  - `d_im`: parameter to set how implicit/explicit the integrator is. If set equal to `1` the integrator is fully implicit, and if set to `0` fully explicit.
  - `nsteps`: number of total time steps to perform
  - `maxit`: maximum number of iterations of the predictor/corrector step.
  - `tol`: maximum allowed tolerance of the predictor/corrector step.
- `block_size` and `grid_size` denote the maximum size of blocks and grids. Every `GPU` has a maximum allowable block and grid sizes. Please adjust these variables if needed.
- `CHECK_CUDART` is a specialized function for CUDA, that allows the user to check if the call of a given `CUDA` function returned an error.

`head.h`: Headers of functions

The header file also includes the header for all the "Main Functions" in the modules, the ones called by other modules or the main file. It also defines the structures: `vfield`, that defines three dimensional `double2` pointers, and `size_p` that is needed for the FFT.

5.1.2 main.cu

The main file sets the flow field variables u, and rhs. Currently it also sets the device where to run the code. The variable dev must be set by the user to the desired GPU where the user wants to run the code.

In the main file, the setters of the modules are called. These functions initialize the variables of the modules. It then calls the function initField(u) where the flow field is initialized and then the function integrate(u,rhs), that integrates the flow field. After the time integration, it writes the final flow field in binary files and clears the memory in the CPU and GPU.

5.1.3 Modules of the code

- mod_boundary.cu has functions to impose boundary conditions at the wall.
- mod_cublasflux.cu has mainly functions to impose the bulk velocity.
- mod_deriv.cu has functions to compute the azimuthal and axial derivatives in spectral space.
- mod_fft.cu has the functions to initialize and perform the FFTs and the corresponding inverse.
- mod_integrate.cu has the functions to perform the time stepping and predictor/corrector operations.
- mod_io.cu has functions to perform input/output operations, and initialize the flow field.
- mod_linear.cu has functions to perform linear operations on the flow field, including the vector-laplacian operation, and the solution of the systems of equations.
- mod_nonlinear.cu has functions to perform non-linear operations in the flow field in physical space.
- mod_radialFD.cu has functions that initialize the radial grid and the matrices needed for the radial derivatives. It also has functions to compute the radial derivatives in spectral space.
- mod_utils.cu has auxiliary functions to, for instance, set a vector equal to 0 in the GPU.

5.2 Features of the code

5.2.1 Initialization

The velocity field u in spectral space is initialized in main.cu as a vfield structure, and directly saved in the GPU. The memory in the GPU needed for the each velocity component is allocated with the following commands:

```
// Allocate memory buffers
vfield u;
cudaMalloc(&u.r,NR*NT*NZ*sizeof(double2));
cudaMalloc(&u.t,NR*NT*NZ*sizeof(double2));
cudaMalloc(&u.z,NR*NT*NZ*sizeof(double2));
```

This defines three vectors of double2 with NR*NT*NZ elements each, i.e. the spectral coefficients of the velocity at each radial location and for each azimuthal/axial Fourier mode combination. Note that a double2 variable aux has two components aux.x and aux.y. The first component corresponds to the real part and the latter to the imaginary part of the complex spectral coefficient.

5.2.2 Sorting of spectral coefficients

The variable `u` is a structure in three dimensions, made by three vectors each in r, t, and z, that stand for radial, azimuthal and axial directions respectively. Each vector has all the spectral coefficients of each velocity field component (r, t and z). The coefficients are ordered according to their Fourier mode and radial position. The fastest running index in the vector, is the axial mode, then the azimuthal mode and finally the radial position. In the case of the azimuthal modes, note that the modes considered lay in the range $[-NT/2, NT/2-1]$. In the code, the azimuthal modes are ordered in increasing sense, first from $[0, NT/2-1]$, and then from $[-NT/2, -1]$.

Assume one sets `NT=4`, `NZ=3` and `NR=2`. In the code, the radial indexes are defined in the range $i_r \in [0, NR-1]$, the azimuthal modes $i_\theta \in [-NT/2, 0, NT/2-1]$ and axial modes $i_z \in [0, NZ-1]$. Thus, in this example, one will have a total of 24 different coefficients for each velocity component, in the range $i_r \in [0,1]$, $i_\theta \in [-2,-1,0,1]$, $i_z \in [0,1,2]$. Let $\hat{u}_r^{i_r,i_\theta,i_z}$ denote the coefficient of the radial velocity at the corresponding radial position, and for the corresponding azimuthal and axial modes, in this example the elements in the vector `u.r` would be ordered as:

$$\hat{u}_r^{0,0,0}, \hat{u}_r^{0,0,1}, \hat{u}_r^{0,0,2}, \hat{u}_r^{0,1,0}, \hat{u}_r^{0,1,1}, \hat{u}_r^{0,1,2}, \hat{u}_r^{0,-2,0}, \hat{u}_r^{0,-2,1}, \hat{u}_r^{0,-2,2}, \hat{u}_r^{0,-1,0}, \hat{u}_r^{0,-1,1}, \hat{u}_r^{0,-1,2}, \tag{5.1}$$

$$\hat{u}_r^{1,0,0}, \hat{u}_r^{1,0,1}, \hat{u}_r^{1,0,2}, \hat{u}_r^{1,1,0}, \hat{u}_r^{1,1,1}, \hat{u}_r^{1,1,2}, \hat{u}_r^{1,-2,0}, \hat{u}_r^{1,-2,1}, \hat{u}_r^{1,-2,2}, \hat{u}_r^{1,-1,0}, \hat{u}_r^{1,-1,1}, \hat{u}_r^{1,-1,2} \tag{5.2}$$

The coefficients of each velocity component are saved as `double2` vectors. That means that the real part of the first coefficient of the radial velocity, at the radial position closest to the center-line of the pipe, is denoted as `u[0].r.x`, and the imaginary part `u[0].r.y`. In the memory architecture of the code, the real part of each coefficient of the variable `u`, is immediately followed by its imaginary part.

5.2.3 The padded fields, and sorting of physical points

As described in Chapter 2, the code uses a pseudo-spectral method and a 3/2-aliasing rule. That means that, the non-linear terms in the Navier-Stokes equations are computed in physical space, and, in order to avoid the aliasing error, the velocity field is computed in a larger number of physical points than the number of spectral coefficients. In the code, both the azimuthal and axial points are increased by $3/2$ with respect to the number of spectral coefficients.

As described in section §2.3.2, before transforming the velocity field from spectral to physical space, it must be padded. The padded fields are allocated as vectors with length:

```
// Allocate memory buffers for the padded fields
vfield u_pad;
cudaMalloc(&u_pad.r,NR*NTP*NZP*sizeof(double2));
cudaMalloc(&u_pad.t,NR*NTP*NZP*sizeof(double2));
cudaMalloc(&u_pad.z,NR*NTP*NZP*sizeof(double2));
```

After performing the inverse Fourier Transform, and transform the velocity from spectral to physical space, the velocity in physical space is also saved in this vectors.

Note that, even though the velocity field in physical space has a real value at each physical location, it is saved using a `double2` vector. The code takes advantage of features of `C` language to amend this. Every time the velocity is required in physical space, the vectors are called as `double` and not as `double2` vectors. In the `double2` `x` vector, the element `x.x` is intermediately followed in memory by the element `x.y`.

When called as a `double` vector, the vectors with the padded velocity follow the same sorting as the spectral coefficients.

The fastest running index is the axial position that goes from index `0` to index `2*NZP-1`. Note that the total number of physical points is twice `NZP` since `NZP=3NZ/2` and `NZ` is only half the axial modes. **IMPORTANT!** the axial position in index `0` corresponds to axial position $z = 0$. However the last axial position is actually saved in the index `2*NZP-3` and not in the last index `2*NZP-1`. The values saved in the axial indexes `2*NZP-2` and `2*NZP-1` are non-physical, and act as auxiliary memory locations for the computations of FFT and inverse Fourier Transforms using `cuFFT` functions. Thus, the last axial location saved in index `2*NZP-3` corresponds to the axial location $z = L_z - \frac{L_z}{2NZP-2}$. The rest of axial locations are linearly and evenly distributed between indexes `0` and `2*NZP-3`.

The second fastest running index in the padded field vectors is the azimuthal position, that goes from index `0` to index `NTP-1`, being `NTP=3NT/2`. The azimuthal position of index `0` corresponds to $\theta = 0$, while the azimuthal position of index `NTP-1` corresponds to $\theta = 2\pi - \frac{2\pi}{NTP}$. The rest of azimuthal positions are linearly and evenly distributed between the two indexes/positions. The slowest running index is the radial position, that goes from the radial position closest to the center-line to the radial-most position.

5.2.4 Initial velocity field

In `nsPipe-CUDA` the velocity field is initialized with the laminar Hagen-Poiseuille profile in the whole domain:

$$u_z = \left(1 - r^2\right). \tag{5.3}$$

On top of the laminar profile, a localized perturbation is introduced, with the form:

$$u'_r = \left(1 - r^2\right)^2 \exp\left[-10 \sin^2\left(\pi x / L_x\right)\right] \sin\left(\theta\right), \tag{5.4}$$

$$u'_\theta = \left[\left(1 - r^2\right)^2 - 4r^2\left(1 - r^2\right)\right] \exp\left[-10 \sin^2\left(\pi x / L_x\right)\right] \cos\left(\theta\right), \tag{5.5}$$

where L_x is the length of the pipe. The parameter `A_P`, that can be set in the `head.h` file, multiplies the perturbation to set its initial magnitude.

5.3 Input/output in the `nsPipe-CUDA` code

The current version of the `nsPipe-CUDA` code, is able to produce up to 8 different output files, and allows the use of three input files to start/restart the simulations.

It must be noted that, before writing to a file, the information must be copied from the device to the host, as only the host can write directly to a file (although new `GPUs` allow to write directly from the device to file). The operation of writing from device to host can take a lot of time, and it is thus recommended to keep the number of them to a minimum.

The code works with three different types of files.

5.3.1 Binary files

The binary files can be used as input and outputs of the code. They have the information of the velocity field in spectral space, i.e. the variable `u`.

Output binary files

At the end of the simulation, the code produces three binary files, one per each velocity component: `ur.bin, ut.bin, uz.bin`. In each file, all the current coefficients of each velocity component are saved. Note that the coefficients in the files are ordered with the same strategy as in the code, see 5.2. The fastest running index is the axial coefficient, then the azimuthal one and finally the radial position. Also note that the amount of data saved in each `*.bin` file is equal to two times the number of spectral coefficients, `2*NR*NT*NZ`, as each coefficient has a real and an imaginary part. In the code the coefficients are ordered in pairs, with the real part directly followed by the imaginary part for each spectral coefficient.

Input binary files and restart of the code

In the `head.h` file, the user can set the value of the `restart` variable. If the variable is set to `restart = 0`, the code will initialize the velocity field with the method described above, with the laminar profile and on top of it a velocity perturbation. If the variable instead is set to `restart = 1`, the code will initialize the velocity field using the spectral coefficients saved in some binary files. For that, the code will try to read the binary files `ur_s.bin, ut_s.bin, uz_s.bin`, that must be located in the same folder where the executable of the code is.

5.3.2 HDF5 files

The user can also save the whole velocity field in physical space at different times steps during the integration in HDF5 files. By default, this output option is commented out in the code.

5.3.3 `*.txt` files

The code produces a series of `*.txt` files during the integration of the code. The user can select the frequency at which information is written to these files, using certain parameters in the `head.h` file.

The code allows the user to minimize the amount of times memory is copied from device to host by using a storing strategy in the device. At time steps where the code should write to one of the `*.txt` files, instead of directly copying the information to the host, and writing to the file, the code stores the information in a vector in the device. This vector continues storing data in the device until a limit, set by the user, is reached. Only then, the information is dumped in the host and written to the file. With this strategy the latency time is reduced, when writing to the files.

Find below a description of each `*.txt` file.

`io_friction.txt`

It saves three variables in three columns. The first column is time, the second the azimuthal and axial averaged axial velocity at the center-line of the pipe $\langle u_z\left(r=0\right)\rangle_{\theta,z}$:

$$\langle u_z\left(r=0\right)\rangle_{\theta,z} = \frac{1}{2\pi L_z}\int_0^{2\pi}\int_0^{L_z} u_z\left(r=0\right)\mathrm{d}\theta\mathrm{d}z, \tag{5.6}$$

and the third column the azimuthal and axial averaged friction velocity at the wall $\langle u_\tau\rangle_{\theta,z}$:

$$\langle u_\tau\left(r=0\right)\rangle_{\theta,z} = \frac{1}{2\pi L_z}\int_0^{2\pi}\int_0^{L_z} u_\tau\mathrm{d}\theta\mathrm{d}z, \tag{5.7}$$

being

$$u_\tau = \sqrt{\frac{1}{Re}\left|\frac{\partial u_z}{\partial r}\right|_{r=R}}. \tag{5.8}$$

The user can select the time (in $R/(2U)$ units) between data points in the `head.h` file with the variable `dt_frc`. The amount of rows saved before copying the data from device to host (and thus from device to file) can be set with the parameter `dc_frc`, also in the `head.h` file.

`io_meanpr.txt`

The first column corresponds to time. The first row of the file (excluding the first column) corresponds to the radial grid of the pipe. The following rows (excluding the first column) correspond to the mean profile in the pipe $\langle u_z\rangle_{\theta,z}$:

$$\langle u_z\rangle_{\theta,z} = \frac{1}{2\pi L_z}\int_0^{2\pi}\int_0^{L_z} u_z\mathrm{d}\theta\mathrm{d}z. \tag{5.9}$$

The user can select the time (in $R/(2U)$ units) between data points in the `head.h` file with the variable `dt_mnp`. The amount of rows saved before copying the data from device to host (and thus from device to file) can be set with the parameter `dc_mnp`, also in the `head.h` file.

`io_qcrosss.txt`

The first column corresponds to time. The first row of the file (excluding the first column) corresponds to the axial grid of the pipe. The following rows (excluding the first column) correspond to the cross-section averaged, cross section kinetic energy $\langle u_r^2+u_\theta^2\rangle_{r,\theta}$:

$$\left\langle u_r^2+u_\theta^2\right\rangle_{r,\theta} = \frac{1}{\pi R^2}\int_0^{2\pi}\int_0^{R}\left(u_r^2+u_\theta^2\right)r\mathrm{d}r\mathrm{d}\theta. \tag{5.10}$$

The user can select the time (in $R/(2U)$ units) between data points in the `head.h` file with the variable `dt_qcr`. The amount of rows saved before copying the data from device to host (and thus from device to file) can be set with the parameter `dc_qcr`, also in the `head.h` file.

io_ucrosss.txt

Same as the io_qcross.txt file, but this time it saves the cross section averaged axial kinetic energy $\langle u_z^2 \rangle_{r,\theta}$:

$$\left\langle u_z^2 \right\rangle_{r,\theta} = \frac{1}{\pi R^2} \int_0^{2\pi} \int_0^R u_z^2 r \mathrm{d}r \mathrm{d}\theta. \tag{5.11}$$

The user can select the time (in $R/(2U)$ units) between data points in the head.h file with the variable dt_qcr. The amount of rows saved before copying the data from device to host (and thus from device to file) can be set with the parameter dc_qcr, also in the head.h file.

5.4 How to set-up, compile and run a nsPipe-CUDA simulation

Find here a short description on how to compile and run the code in a Linux environment.

5.4.1 Before compiling

Before compiling, the user must select the desired parameters in the head.h file, and the desired device in the main.cu file (variable dev). The user can run nvidia-smi to check the available GPUs in the cluster of use. Then set dev=ID, where ID is the identification number of the desired GPU.

5.4.2 Compile the code

In one directory above the folder src, find the Makefile file whose content is:

```
NVCC = nvcc
LIBS = -lcufft -lcublas -lcurand -lcusparse
DEBUG = -g
GPU_SOURCES = $(wildcard src/*.cu)
GPU_OBJECTS = $(GPU_SOURCES:.cu=.o)

all: $(GPU_OBJECTS)
	$(NVCC) -o Pipe $(GPU_OBJECTS) $(LIBS)

$(GPU_OBJECTS): src/%.o: src/%.cu
	$(NVCC) -c $< -o $@

clean:
	rm src/*.o Pipe
```

In the case you want to have h5 capabilities, change the Makefile file to add two new libraries:

```
NVCC = nvcc
LIBS = -lcufft -lcublas -lhdf5 -lhdf5_hl -lcurand -lcusparse
DEBUG = -g
GPU_SOURCES = $(wildcard src/*.cu)
GPU_OBJECTS = $(GPU_SOURCES:.cu=.o)

all: $(GPU_OBJECTS)
	$(NVCC) -o Pipe $(GPU_OBJECTS) $(LIBS)

$(GPU_OBJECTS): src/%.o: src/%.cu
	$(NVCC) -c $< -o $@
```

```
clean:
	rm src/*.o Pipe
```

To compile the code the user only needs to have installed CUDA and have a C compiler, together with the nvcc compiler.

Run make to make the code. If the compiler process is successful, a executable Pipe will be created.

Note that, to clean the src directory, the user can run make clean to delete all files created during the compilation. Run make clean whenever the user wants to re-compile the code and avoid conflicts with previous compilations.

5.4.3 Running the code restart=0

Run the command

```
$ nohup ./Pipe &
```

to start the simulation. By including & the code will continue running even if the user exits the device. Apart from the aforementioned output files, the code will produce a nohup.out file. There, the user will be notified if an error occurs. Otherwise, the user can see here the information about the device selected, and periodically, the final error of the last predictor/corrector iteration and the current time step. At the end of the calculations the total computational time will be printed also here. See an example below:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++ Welcome to nsPipe in CUDA +++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Setting device 0
Preparing initial condition

 err=1.238904e-12
 Step 40 of 400000

 err=7.139728e-13
 Step 80 of 400000

 err=3.695701e-13
 Step 120 of 400000
...

Total Time = 3535.000s
```

Note that if one runs the command

```
$ nohup ./Pipe
```

The information will be printed directly to the console. However, if the user exits the device, the execution will stop. Note also that, sometimes, the print of information to the input/output files and the nohup.out file, can have a certain delay. One has to wait a couple of minutes to see the information that is on the other hand being printed there.

5.4.4 Running the code restart=1

In case the restart parameter is activated, the user must save in the folder where the executable Pipe is run the binary files ur_s.bin, ut_s.bin, uz_s.bin.

5.4.5 *Useful commands in a GPU cluster*

In order to see the available GPUs in your cluster, and the simulations running in them you can use the command:

```
$ nvidia-smi
```

In order to kill a CUDA process, one must identify the process identification number of the case of interest, the column PID after running nvidia-smi. Then one can run

```
$ kill ...
```

where the user must write the PID process ID instead of the three points.

Bibliography

[Avi+11] Kerstin Avila, David Moxey, Alberto de Lozar, et al. “The onset of turbulence in pipe flow”. In: *Science* 333.6039 (2011), pp. 192–196 (cit. on p. 21).

[Avi+23] Marc Avila, Dwight Barkley, and Björn Hof. “Transition to Turbulence in Pipe Flow”. In: *Annual Review of Fluid Mechanics* 55.1 (2023), null. eprint: `https://doi.org/10.1146/annurev-fluid-120720-025957` (cit. on p. 42).

[Bis+83] A Bishop, D Campbell, B Nicolaenko, and P Holmes. “Nonlinear Problems: Present and Future”. In: (1983) (cit. on p. 14).

[Boy01] John P Boyd. *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001 (cit. on pp. 3, 5).

[For88] Bengt Fornberg. “Generation of Finite Difference Formulas on Arbitrarily Spaced Grids”. In: *Mathematics of Computation* 51.184 (1988), pp. 699–706 (cit. on p. 10).

[Hof+08] Björn Hof, Alberto De Lozar, Dirk Jan Kuik, and Jerry Westerweel. “Repeller or attractor? Selecting the dynamical model for the onset of turbulence in pipe flow”. In: *Physical review letters* 101.21 (2008), p. 214501 (cit. on p. 21).

[Ló+20] Jose Manuel López, Daniel Feldmann, Markus Rampp, et al. “nsCouette – A high-performance code for direct numerical simulations of turbulent Taylor–Couette flow”. In: *SoftwareX* 11 (2020), p. 100395 (cit. on pp. ii, 1).

[Pey02] Roger Peyret. *Spectral methods for incompressible viscous flow*. Vol. 148. Springer, 2002 (cit. on pp. 3, 4).

[PT22] Carlos Plana Turmo. “Phase-field simulations of two-phase pipe flow”. PhD thesis. University of Bremen, 2022 (cit. on pp. 11, 15).

[Pop00] Stephen B Pope. *Turbulent flows*. Cambridge university press, 2000 (cit. on p. 39).

[Rem06] Dietmar Rempfer. “On boundary conditions for incompressible Navier-Stokes problems”. In: (2006) (cit. on p. 16).

[Tre00] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000 (cit. on pp. 3, 14).

[Wil17] Ashley P Willis. “The Openpipeflow Navier–Stokes solver”. In: *SoftwareX* 6 (2017), pp. 124–127 (cit. on p. 16).

A Grid estimation in `nsPipe-CUDA`

Find in this appendix a description of the way to estimate the grid in `nsPipe-CUDA` a priori, before running the simulation. The estimation is used as a guideline to define the final grid resolution of the simulations. This strategy was first proposed by D. Feldmann who is here greatly acknowledged.

A.1 Strategy to choose the grid resolution

In order to perform DNS of turbulent flows, the grid resolution must be fine enough so all the turbulent scales are well represented. There is a caveat however, one can only guess the size of the smallest scales before performing the simulation. Moreover it is of the best interest to keep the grid spacing as coarse as possible, in order to reduce the computation time. Thus, the goal is to make a good enough a priori estimation, that returns the minimal resolution possible that best represents the turbulent flow.

There are different methods to a priori estimate the size of the smallest scales and therefore of the minimum grid spacing beforehand. In this section a method proposed by Dr. Daniel Feldmann is explained. Note that the shear thickness is defined as $\delta_\nu = \nu/u_\tau = D/\left(2Re_\tau\right)$, where $u_\tau = \sqrt{\tau_w/\rho}$ is the shear velocity, τ_w is the magnitude of the average shear at the wall and $Re_\tau = u_\tau D/\left(2\nu\right)$ the shear Reynolds number.

A.1.1 Estimate the dissipation

First assume that the dissipation ε is equal to the volume (V) and time (t) averaged dissipation:

$$\varepsilon \approx \frac{1}{tV}\int_0^t \iiint_V \varepsilon\left(\boldsymbol{z}, t\right) \mathrm{d}V, \tag{A.1}$$

at each point of the flow.

Secondly, assume that all the energy injected to the flow is dissipated by the dissipation. The energy injected by the flow is assumed to be equal, on average, to the axial pressure gradient $\left\langle \frac{\partial p}{\partial z} \right\rangle_t$ that drives the flow. Then:

$$\left\langle \frac{\partial p}{\partial z} \right\rangle_t \cdot U \approx \rho\varepsilon. \tag{A.2}$$

From the Reynolds Average Navier Stokes equations, one can obtain an equation for the pressure gradient as:

$$\left\langle \frac{\partial p}{\partial z} \right\rangle_t = \left\langle \frac{4\tau_w}{D} \right\rangle_t, \tag{A.3}$$

which in turn leads to:

$$\varepsilon \approx \frac{4u_\tau^2 U}{D}. \tag{A.4}$$

A.1.2 Estimate the shear at the wall

Here, fully turbulent flow is assumed. According to Blasius empirical resistance formula [Pop00], the shear stress is given as:

$$\frac{\tau_w}{\rho U^2} = \frac{1}{8}\frac{0.3164}{Re^{1/4}}. \tag{A.5}$$

This means that the shear velocity, u_τ can be computed as:

$$\frac{u_\tau}{U} = \sqrt{\frac{\tau_w}{\rho U^2}} = \sqrt{\frac{1}{8}\frac{0.3164}{Re^{1/4}}} \tag{A.6}$$

and the shear Reynolds number can then be written as a function of Re:

$$Re_\tau = \frac{u_\tau}{2U} Re = 0.099373 Re^{7/8}. \tag{A.7}$$

A.1.3 Estimate the Kolmogorov scale

An approximate Kolmogorov scale can be computed as:

$$\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{1/4} \approx \left(\frac{D\nu^3}{4u_\tau^2 U}\right)^{1/4}. \tag{A.8}$$

In plus units, the Kolmogorov scale is then:

$$\eta^+ = \frac{\eta}{\delta_\nu} = \frac{\eta u_\tau}{\nu} \approx \left(\frac{D\nu^3 u_\tau^4}{4u_\tau^2 U\nu^4}\right)^{1/4} = \left(\frac{Du_\tau^2}{4U\nu}\right)^{1/4} = \left(\frac{u_\tau}{2U} Re_\tau\right)^{1/4}. \tag{A.9}$$

Invoking equation. A.7, one finds:

$$\eta^+ \approx \left(\frac{Re_\tau^2}{Re}\right)^{1/4}. \tag{A.10}$$

With this definition, one can estimate the number of grid points needed for each coordinate.

A.1.4 Radial points

Here a uniform radial grid is assumed. Thus, the grid spacing in r is constant and equal to $\Delta r = \frac{D}{2N_r}$, where N_r are the radial points.

The radial grid spacing is chosen so, in the worst case scenario $\Delta r^+ \leq \eta^+$. This is a typical requirement in the simulation of shear flows.

One can obtain an expression for the required number of radial grid points as:

$$\Delta r^+ \leq \eta^+ \rightarrow \frac{Re_\tau}{N_r} \leq \eta^+ \rightarrow N_r \geq \frac{Re_\tau}{\eta^+}. \tag{A.11}$$

By invoking eq. (A.10) and eq. (A.7), one can write the number of radial points as a function of Re:

$$N_r \geq Re^{1/4} Re_\tau^{1/2} = 0.31532 Re^{7/16} Re^{1/4}, \text{ and find:} \tag{A.12}$$

$$N_r \geq 0.31523 Re^{11/16}. \tag{A.13}$$

A.1.5 Azimuthal points

Here a uniform azimuthal grid is assumed. The larger grid spacing of an azimuthal grid is found at the wall. There, the grid spacing is equal to $\Delta\theta = \frac{\pi D}{2N_\theta}$, where N_θ is half the total number of azimuthal Fourier modes.

The number of azimuthal grid points is chosen so, at the wall $\Delta\theta^+ \leq 4\eta^+$, which is the common spacing used in the span-wise direction of the simulations of shear flows.

With this requirement one can obtain an expression of the total number of azimuthal grid points as:

$$\Delta\theta^+ \leq 4\eta^+ \rightarrow \frac{2\pi Re_\tau}{2N_\theta} \leq 4\eta^+ \rightarrow 2N_\theta \geq \frac{\pi Re_\tau}{2\eta^+}. \tag{A.14}$$

By invoking equation (A.11) one finds:

$$2N_\theta \geq \frac{\pi}{2} N_r. \tag{A.15}$$

A.1.6 Axial points

A uniform axial grid is assumed. For a pipe length $L_z^* = L_z D$, the axial grid spacing is $\Delta z = \frac{L_z D}{2N_z}$, where N_z is half the total number of axial Fourier modes.

The number of axial grid points is chosen so: $\Delta z^+ \leq 8\eta^+$, which is the common spacing used in the stream-wise direction of simulations of shear flows.

With this requirement, one can obtain an expression of the total number of axial grid points as:

$$\Delta z^+ \leq 8\eta^+ \rightarrow \frac{2L_z Re_\tau}{2N_z} \leq 8\eta^+ \rightarrow 2N_z \geq \frac{L_z Re_\tau}{4\eta^+}. \tag{A.16}$$

By invoking the last equality in equation (A.11) one finds:

$$2N_z \geq \frac{L_z}{4} N_r. \tag{A.17}$$

B Quickly run the default `nsPipe-CUDA` case

Find in this appendix a description and tutorial of the default case of `nsPipe-CUDA` found in the `Main_Code/` folder. It is here described what it computes, how to run it and how to post-process the results.

B.1 Description of the case

The default case considers the flow driven at a steady bulk velocity U in a $L_z = 100R$ long pipe at $Re = 1850$. The flow is initialized with the laminar Hagen-Poiseuille profile in the whole domain, see eq. (5.3), and on top of it, the perturbation in equation (5.5). The perturbation is scaled in magnitude with the parameter `A_P`, that is set to `A_P=0.3` in the `head.h` file.

At this Re the perturbation is expected to grow and trigger an axially localized turbulent puff, similar to the one shown in the cover of this document. The turbulent puff will remain localized, and move at a given axial velocity in the pipe. At this Re the mean speed of the puff will be slightly larger than the bulk velocity U. The code will integrate the evolution of the puff for `nsteps=1000000` time steps, with a step size of `dt=0.01`. This means that, after the computation is completed, the final time of the simulation will be at $t = 10000R/U_c = 2500D/U$.

Note that at this Re puffs decay at random times [Avi+23]. This time of decay will be different depending on the device you use, and can happen before $t = 10000R/U_c = 2500D/U$. You are welcomed to send your time of decay to us at to help us compile the times of decay at this Re. See below how to identify this time of decay.

The beginning of the default `head.h` file should read:

```
/* INTEGRATE NAVIER-STOKES EQUATIONS IN CYLINDRICAL COORDINATES USING A PSEUDO-SPECTRAL METHOD */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cublas_v2.h>
#include <cufft.h>
//#include <hdf5.h>
//#include <hdf5_hl.h>
#include <time.h>
#include <sys/time.h>
/*++++++++++++++++++++++++++++++ CONSTANTS ++++++++++++++++++++++++++++++*/
#define PI2 6.283185307179586 // two times pi

/*++++++++++++++++++++++++ SIMULATION PARAMETERS ++++++++++++++++++++++*/
// Grid points and modes
#define NR 48 // Radial grid points
```

```
#define NT 64 // 2*m_theta theta modes (must be even!)
#define NZ 257 // m_z+1 axial modes (must be odd! )

// Length of the pipe
#define LT PI2 // Theta length
#define LZ 100.0 // Length of the pipe in R

// Physical parameters
#define Re 1850 // Reynolds number
#define A_P 0.3 // Initial perturbation amplitude

/*++++++++++++++++++++++++++ IN&OUT PARAMETERS ++++++++++++++++++++++*/
// Restart
#define restart 0 // >0 will try to read *_s.bin files

// Friction file
#define dt_frc 0.5 // t in code units to write to io_friction
#define dc_frc 100 // Number of rows to store before writing
// Q&U CrossS file
#define dt_qcr 0.5 // t in code units to write to io_q&ucross
#define dc_qcr 100 // Number of rows to store before writing
// Mean file (mean profile file)
#define dt_mnp 0.5 // t in code units to write to io_meanpr
#define dc_mnp 100 // Number of rows to store before writing

/*++++++++++++++++++++++++ INTEGRATOR PARAMETERS ++++++++++++++++++++++*/
// Finite differences
#define sten 7 // Stencil of radial derivative
#define iw sten/2 // half stencil

// Time-Step size
#define dt 0.01 // time step size (constant)
#define d_im 0.51 // Factor multiplying the implicit term
#define nsteps 1000000 // Number of time steps
#define maxit 10 // Number of max iterations to converge
#define tol 5e-8 // Tolerance of convergence analysis

// Points after aliasing
#define NTP (3*NT/2) // Total of physical azimuthal points
#define NZP (3*(NZ-1)/2+1) // half of physical axial points

/*++++++++++++++++++++++ ARCHITECTURE PARAMETERS ++++++++++++++++++++++*/
// Prepare block dimensions
#define block_size 256 // Number of threads per block
#define grid_size 65535 // Number of blocks in direction y and z

...
```

B.2 Things to (maybe) change before running

Things you may consider to change before running the the default case are:

- **The grid size**. The main limitation of `nsPipe-CUDA` is the `GPU` memory. The default case fits in any `GPU` with `2GB` or more memory. Otherwise, you should reduce either `NR`, `NT` or `NZ`.
- **The maximum allowable size of blocks and grids**. Each `GPU` has a maximum allowable size of block and grids. This can be easily obtained from the `GPU CUDA` specs. The default `block_size` and `grid_size` are conservative enough for almost all `CUDA` capable `GPUs`, but you may need to change them.
- **The variable `dev` in `main.cu`**. This variable sets the `GPU` where your code will run. You may want to first run `nvidia-smi` to see the `GPUs` available, and change this variable accord-

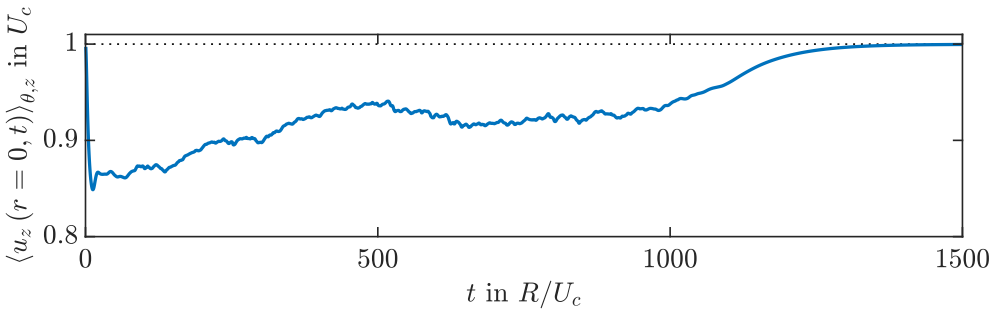


Fig. B.1.: Resultant axial and azimuthal averaged center-line velocity saved in the `io_friction.txt` after running the default case of `nsPipe-CUDA`. Note that the horizontal axis is limited to $t \leq 1500R/U_c$, while the simulation was run until $t \leq 10000R/U_c$.

ingly. Note that, as long as the maximum `GPU` memory is not reached, one can run multiple `nsPipe-CUDA` simulations in the same `GPU`, with an obvious decrease in the code speed.

B.3 Run the code

To run the code, in the `Main_code/` folder where the `src/` folder and the `Makefile` files are found, first run

```
$ make clean
```

to eliminate any previously compiled case. Then run:

```
$ make
```

to compile the code. This will generate the executable `Pipe`. Then, to run the code, simply run:

```
$ nohup ./Pipe &
```

B.4 Results and how to post-process them

The code produces different output files, as described in §5.3. To check the status of the simulation you can periodically see the information printed in the `nohup.out` file. To quickly see the results you can plot the data saved in the `io_friction.txt` file.

See in figure B.1 a plot of the center-line velocity (second column in the file) with respect to time (first column in the file). At the beginning of the simulation the flow is laminar and the center-line velocity is maximum and equal to U_c and thus equal to 1. As the turbulent puff is triggered, and evolves, the center-line velocity decreases and reaches a chaotic state, due to turbulence. At some point the puff decays and the center-line velocity returns to the laminar case, as the flow slowly re-laminarizes.

Independently on how fast your puff re-laminarizes, the evolution of the initial perturbation is always the same. That is why you can quickly asses your results compared with ours in figure B.2.

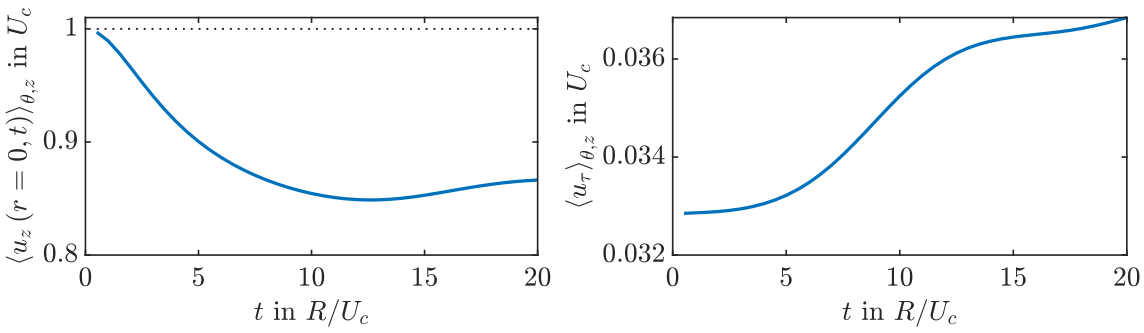


Fig. B.2.: Resultant axial and azimuthal averaged center-line velocity (left) and friction velocity u_τ (right) saved in the `io_friction.txt` after running the default case of `nsPipe-CUDA`. Note that the horizontal axis is limited to $t \leq 1500R/U_c$, while the simulation was run until $t \leq 10000R/U_c$.

In the figure, to the left you can see the initial center-line velocity evolution, and to the right, the initial friction velocity (third column of the `io_friction.txt` file) evolution.

Find the `MATLAB` code used to post-process this file and generate figures B.1 and B.2 below. You can use similar codes to load and post-process the rest of output `.txt` files of the code.

```
clc;clear;close all;
% This script reads the io_friction.txt file and plots the center-line
% velocity and friction velocity in the pipe.
% Variables used to adjust the plot
FntSz=8.5; ncen=13.5; LnWd=1; Co1=lines; Co1=Co1(1,:);
% Variables used to set the size of the subplot
x0=0.05; y0=0.15; dx=0.12;  DX=0.4; DY=0.8;  tf=20;
%% INITIALIZE
frc=load('io_friction.txt');

%% CREATE FIGURE B.1 OF THE USER GUIDE
figure(1);
% Line with the centerline velocity of the laminar case
plot(frc(:,1),1+0.*frc(:,1),':k','LineWidth',LnWd/2); hold on
% Line with the centerline velocity of the DNS
plot(frc(:,1),frc(:,2),'LineWidth',LnWd,'Color',Co1)
% Axes
ax=gca; ax.FontSize=FntSz; set(gca,'TickLabelInterpreter','latex')
% Limits of the plot
xlim([0 1500]); ylim([0.8 1.01]); yticks(0.8:0.1:1)
% Axes labels
ylabel("$\left \langle u_{z}\left(r=0,t\right) \right \rangle_{\theta,z}$ in $U_{c
    }$",...
    'Interpreter','latex','FontSize',FntSz)
xlabel("$t$ in $R/U_{c}$",'Interpreter','latex','FontSize',FntSz);
% Layer: top
set(gca,'layer','top');


%% CREATE FIGURE B.2 OF THE USER GUIDE
figure(2);
% subplot 1: centerline velocity
subplot(1,2,1)
% Line with the centerline velocity of the laminar case
plot(frc(:,1),1+0.*frc(:,1),':k','LineWidth',LnWd/2); hold on
```

```
% Line with the centerline velocity of the DNS
plot(frc(:,1),frc(:,2),'LineWidth',LnWd,'Color',Co1)
% Axes
ax=gca; ax.FontSize=FntSz; set(gca,'TickLabelInterpreter','latex')
% Limits of the plot
xlim([0 tf]); ylim([0.8 1.01]); yticks(0.8:0.1:1)
% Axes labels
ylabel("$\left \langle u_{z}\left(r=0,t\right) \right \rangle_{\theta,z}$ in $U_{c
    }$",...
    'Interpreter','latex','FontSize',FntSz)
xlabel("$t$ in $R/U_{c}$",'Interpreter','latex','FontSize',FntSz);
% Layer: top and position of subplot
set(gca,'layer','top');   set(gca,'Position',[x0 y0 DX DY]);
% subplot 1: centerline velocity
subplot(1,2,2)
% Line with the friction velocity of the DNS
plot(frc(:,1),frc(:,3),'LineWidth',LnWd,'Color',Co1)
% Axes
ax=gca; ax.FontSize=FntSz; set(gca,'TickLabelInterpreter','latex')
% Limits of the plot
xlim([0 tf]); %ylim([0.8 1.01]); yticks(0.8:0.1:1)
% Axes labels
ylabel("$\left \langle u_{\tau} \right \rangle_{\theta,z}$ in $U_{c}$",...
    'Interpreter','latex','FontSize',FntSz)
xlabel("$t$ in $R/U_{c}$",'Interpreter','latex','FontSize',FntSz);
% Layer: top and position of subplot
set(gca,'layer','top');   set(gca,'Position',[x0+dx+DX y0 DX DY]);
```

B.4.1 Postprocess the `.bin` files

At the end of the simulation, if the code did not abort due to errors, it will produce three binary files `ur.bin, ut.bin, uz.bin`, see §5.3.1. In each file, the corresponding spectral coefficients of each velocity component at the last time step are saved. You may want to transform this information to the velocity in physical space. Find below a MATLAB code that does just that:

```
clc;clear;close all;tic;
% This script loads one od the output binary files of the code. It reads
% the spectral coefficients, and computes the velocity in physical space.

%% INPUTS (must include them as in the CUDA code)
NR   =48;              % Radial points
NT   =64;              % 2*mth modes (must be even)
NZ   =257;             % mz+1 modes (must be odd)
sten=7;                % Stencil
LZ   =100;             % Length of the pipe in R
LT   =2*pi;            % Circunference in rad

%% INITIALIZE THE GRID AS IN THE CODE
N_=NR+floor(sqrt(NR));   r=zeros(NR,1);
% Grid and auxiliary grid
for i=N_-NR+1:N_; r(i-N_+NR)=0.5*(1+cos(pi*(N_-i)/N_)); end
for i=1:10; dr=1.5*r(1)-0.5*r(2); r=r.*(1+dr)-dr; end
% Define grid in 2D
thi=linspace(0,LT,3*NT/2); zi=linspace(0,LZ,3*(NZ-1));

%% READ THE BIN FILE AND PREPARE GRID IN 3D
file=fopen('ur.bin','r');  A=fread(file,'double');   A=A';  fclose(file);
% Radial velocity coefficients
ur_hat=A(1:2:end-1)+1i.*A(2:2:end);
```

```
ur_hat=reshape(ur_hat,[NT*NZ,NR]); ur_hat=ur_hat';
% Initialize the velocity field in physical space
ur=zeros(NR,3*(NZ-1),3*NT/2); ut=ur; uz=ur;
% Now compute the grid in 3D
% Note that you need to compute z as negative to get the correct direction
% of the flow
z=ur-zi;
% Azimuthal direction
th=ur; for i=1:3*NT/2; th(:,:,i)=th(:,:,i)+thi(i); end
disp('Loaded and ready to rumble'); toc

%% RECONSTRUCT THE VELOCITY
% Loop on wavenumbers
for k=1:NT*NZ
    % Azimuthal index and wavenumber
    it=floor(k/NZ); if it>=NT/2; it=it-NT; end; kth=it*2*pi/LT;
    % Axial index and wavenumber
    l=mod(k-1,NZ); kz=l*2*pi/LZ;
  % CUT-OFF
  if(abs(it)<(NT/2) && abs(l)<(NZ-1))
    % Prepare the field
    aux=exp(1i*kth.*th+1i*kz.*z);
    % Prepare the modes of the velocity
    umr=ur_hat(:,k);
    % Symmetry condition
    if l>0; umr=2.*umr;  end
    % Add to velocity
    ur=ur+real(umr.*aux);
  end% end if of cut-off
end
disp('U computed the velocity in physical space!');
disp('U can now continue plotting it or save it to file.'); toc
```