UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"
Corso di Laurea Magistrale
in Ingegneria del Veicolo (D.M. 270/04)

Implementation of a Synthetic Turbulent Inlet for turbulent flow simulation in CaNS

Implementazione di un Generatore di turbolenza sintetico per flussi turbolenti in CaNS

Supervisor:
Prof. Andrea Cimarelli

Candidate:
Emanuele Spadafora

Co-supervisor:
Dr.-Ing. Davide Gatti

ANNO ACCADEMICO 2025-2026

«Audentes fortuna iuvat.»

— Virgil

Abstract

Compared to time-evolving turbulence, Direct Numerical Simulations (DNS) require physically consistent velocity boundary conditions, and the computed solution depends strongly on the quality of the prescribed inflow. This work presents one of the available synthetic turbulent inlet generators, specifically Klein's digital filtering algorithm and its implementation in the open-source solver CaNS. The described method uses first- and second-order one-point statistics to generate a turbulent mean and fluctuation field to be imposed at the boundary. The statistics provided can be generated from other simulations or directly from measurements, thus providing a flexible method for inflow generation. The turbulent inlet is employed to perform direct numerical simulations of turbulent channels with inlet/outlet boundary conditions, in order to assess the quality of the artificial inlet in terms of the streamwise length required by turbulence to develop the well-known statistics of fully developed doubly periodic channels, such as those of Moser, Kim and Mansour (1999) at $Re_\tau = 180$. The method is first developed in Python as a proof of concept, where the residual discrepancies are analysed and traced back to the constant-length-scale approximation of the standard formulation, and subsequently ported to CaNS as a Fortran module.

Contents

Italian summary **1**

1 Introduction **7**
1.1 The Random and Chaotic Nature of Turbulence 8
1.2 Inflow Turbulence Generation Methods 11
1.2.1 Library-Based Methods 12
1.2.2 Recycling-Rescaling-Based Methods 12
1.2.3 Synthetic Turbulence Generators 13
1.2.4 Comparison and Method Selection 16
1.3 Objectives and Thesis Structure 17

2 Generation of the Turbulent Inlet Velocity Field **18**
2.1 Klein's Digital Filter Method 19
2.1.1 Problem Statement and Motivation 19
2.1.2 Generation of Correlated Random Fields 20
2.1.3 Choice of Filter Kernel and Length-Scale Recovery . . 21
2.1.4 Lund Transformation for Anisotropic Turbulence . . . 22
2.1.5 Summary of Klein's Algorithm 22
2.2 Kempf's Low-Storage Implementation 23
2.2.1 Separable Filtering 23
2.2.2 Low-Storage Rolling Buffer 24
2.2.3 Parallel Implementation Without Communication . . . 25
2.2.4 Computational Cost Comparison 25
2.3 Algorithm implementation 26

3 Python implementation and validation of the Digital Filter's algorithm **27**
3.1 Motivation for a Python prototype 27
3.2 Introduction to Python Prototype 28
3.3 Python code 29
3.3.1 Handling Non-Uniform Grids 29

3.3.2 Filter Coefficients . . . 29
3.3.3 Cholesky Decomposition . . . 30
3.3.4 Rolling Buffer and Optimized Streamwise Filtering . . 30
3.3.5 Time-Step Loop . . . 31
3.3.6 Input Parameters . . . 31
3.3.7 Determination of the Normalized Length Scales (n) . . 32
3.3.8 Uniform auxiliary grid . . . 34
3.4 Generated data and validation . . . 36
3.4.1 Results . . . 36
3.4.2 Limitations of the prototype . . . 40

4 Implementation and working principle of the algorithm in CaNS 43
4.1 From Python to Fortran: Explicit Loops and Derived Types . 44
4.1.1 Convolution Without Libraries . . . 44
4.1.2 Interpolation Without `SciPy` . . . 46
4.1.3 Storage Layout of the `turbulentInlet_type` . . . 47
4.1.4 Random Number Generation . . . 48
4.2 Ring Buffer with Moving Head . . . 50
4.2.1 Moving Head Index . . . 50
4.2.2 Filtering with the Moving Head . . . 51
4.2.3 Memory Layout . . . 51
4.3 Parallelization and Halo Cells . . . 52
4.3.1 Why Parallelization Is Mandatory . . . 52
4.3.2 Domain Decomposition in CaNS . . . 53
4.3.3 Slab and Pencil Decomposition . . . 53
4.3.4 Halo Cells . . . 55
4.3.5 OpenMP and OpenACC Parallelization . . . 57
4.4 Temporal Advancement and Taylor's Hypothesis . . . 57
4.4.1 The Frozen-Turbulence Approximation . . . 57
4.4.2 Sub-Cell Shift Accumulation . . . 58
4.4.3 Choosing the Bulk Velocity . . . 58
4.4.4 Retiring and Replenishing Planes . . . 59
4.5 Integration with the CaNS Solver . . . 60
4.5.1 Initialization . . . 60
4.5.2 The Time-Stepping Loop . . . 61
4.5.3 Application During the Runge-Kutta Stages . . . 62
4.5.4 The Boundary-Condition Routines . . . 63
4.6 Summary and Discussion . . . 64

5 Channel flow's setup and results **66**
5.1 Preliminary Simulations 66
5.1.1 First Doubly-periodic simulation 67
5.1.2 Results for the first Doubly-periodic simulation 69
5.1.3 Second Doubly-periodic simulation 72
5.1.4 Results for the second Doubly-periodic simulation . . . 73
5.2 Main simulation with Digital Filter 73
5.2.1 Setup for channel's simulation 74
5.2.2 Digital filter inlet configuration 78
5.2.3 Flow development and averaging window 79
5.2.4 Results 79

6 Conclusions and Future Work **86**
6.1 Summary of Results 86
6.2 Answers to the Objectives 87
6.3 Limitations and Future Work 88

List of Figures

1.1 Divergence of the x-component solution of the Lorenz system (1.2) for $t > 27$. 10
1.2 Schematic of the recycling–rescaling inflow generator. 13

2.1 Influence of conventional inflow boundary conditions on the simulation of a plane jet. Comparison between inflow with zero fluctuations, random fluctuations and fluctuations from an external channel flow DNS. Adapted from "A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations" by Klein, Sadiki, and Janicka [1]. 19
2.2 Separable filtering as three successive one-dimensional convolutions. The three-dimensional convolution (2.1) is decomposed into a sequence of 1D filters applied along the k, j and i directions, progressively reducing the working set until the final scalar $R^{\alpha}_{I,J,K}$ is obtained. 24

3.1 Profiles of terms in the balance equation for kinetic energy. . . 34
3.2 Mean streamwise velocity as a function of the channel height. 37
3.3 Mean streamwise velocity as a function of y^+ (logarithmic axis). 38
3.4 Reynolds stress tensor components. 38

4.1 Schematic of the Klein digital-filter inflow generation: the padded random field is successively filtered in the x, y and z directions to produce the correlated noise plane $\eta_c(i_y, i_z)$. . . 46
4.2 Empirical histogram of $N = 10^4$ samples drawn from `gen_normal`, compared with the analytic standard normal density $\mathcal{N}(0, 1)$. . 49
4.3 Slab decomposition of the $16 \times 16 \times 16$ inlet block into four x-aligned slabs. 53
4.4 Pencil decomposition of the $16 \times 16 \times 16$ inlet block into eight z-aligned towers. 54

4.5 Left: partition of the computational domain into four MPI ranks. Right: the subdomain of rank 1 surrounded by one layer of halo cells. 55
4.6 Staggered grid arrangement of a single computational cell. Pressure p stored at the cell centre; velocity components u, v and w stored on the faces normal to the x, y and z directions, respectively. 63

5.1 Adimensionalized streamwise velocity as a function of y^+ . . . 69
5.2 Reynolds stress components. 70
5.3 Viscous and Reynolds stress adimensionalized with τ. 71
5.4 Total stress profile adimensionalized with τ_w. 71
5.5 Adimensionalized streamwise velocity as a function of $\log(y^+)$ 74
5.6 Reynolds stress components. 75
5.7 Viscous and Reynolds stress adimensionalized with τ. 75
5.8 Total stress profile adimensionalized with τ_w. 76
5.9 Computational domain of the channel-flow DNS. The streamwise direction is periodic with length $L_x = 18\pi\delta$, the wall-normal direction spans the channel height $L_y = 2\delta$ between the two solid walls, and the spanwise direction is periodic with length $L_z = 2\pi\delta$. 76
5.10 Evolution of the local friction coefficient $c_f(x)$ in streamwise direction compared to the value of the global friction coefficient C_f obtained from the second doubly-periodic simulation. 80
5.11 Adimensionalized streamwise velocity compared to the reference one. 81
5.12 Components of Reynolds stress tensor. 82
5.13 Shear stresses in function of the adimensionalized half-channel height. 83
5.14 Fractional contribution of the viscous and Reynolds stresses to the total stress. 84
5.15 Streamwise development of the dimensionless time-averaged streamwise velocity variance. 85

List of Tables

1.1 Qualitative comparison of inflow turbulence generation method families. ✓ = satisfies, ∼ = partially satisfies, × = does not satisfy. 16

2.1 Comparison of computational effort and memory requirements. 25

3.1 Main parameters of the `InflowGenerator` class. 31
3.2 Main parameters for the generation of the velocity field. 36

4.1 Members of the derived type `turbulentInlet_type`. Array shapes are given in Fortran order. 48

5.1 Simulation parameters for the first doubly-periodic channel simulation at $Re_\tau = 180$ 67
5.2 Comparison of different quantities between the results from the first doubly-periodic simulation and reference ones at nominal $Re_\tau = 180$ 69
5.3 Comparison of different quantities between the results for the second doubly-periodic simulation and reference ones at nominal $Re_\tau = 180$ 73
5.4 Computational parameters for CaNS for the main spatially developing channel simulation with turbulent inlet. 77
5.5 Integral quantities for the turbulent channel at nominal $Re_\tau = 180$, comparing the main simulation with the digital-filter inlet (TIG) against the second doubly-periodic reference run. 85

Nomenclature

Acronyms & Abbreviations

CPU Central Processing Unit

DFSEM Divergence-Free Synthetic Eddy Method

GPU Graphics Processing Unit

HPC High-Performance Computing

MPI Message Passing Interface

NumPy Numerical Python library

.npz NumPy zip archive file format

OpenMP Open Multi-Processing

OpenACC Open Accelerators

RANS Reynolds-Averaged Navier-Stokes

RNG Random Number Generator

SciPy Scientific Python library

Greek Letters

α	Velocity-component index, $\alpha \in \{u, v, w\}$
β	Physical dimensions of the fluid layer (Lorenz system)
Δt	Time step
Δ	Grid mesh spacing
δ	Half channel height or boundary layer thickness

δ_{99} 99% boundary layer thickness

$\delta_{\mathbf{mn}}$ Kronecker delta

δ_{inlt} Boundary layer thickness at inlet plane

Δ_z^{uni} Uniform auxiliary grid spacing in the wall-normal direction

δ_{recy} Boundary layer thickness at recycle plane

δr_j Separation increment

$\Delta x^+, \Delta y^+, \Delta z^+$ Grid spacing in wall units in the streamwise, spanwise, and wall-normal directions

$\eta_c(i_y, i_z)$ Correlated noise plane

ν Kinematic viscosity

π Mathematical constant in the Gaussian autocorrelation model

ρ Fluid density / Rayleigh number in eq. (1.2)

σ Prandtl number (Lorenz system)

σ_{R^α} Standard deviation of the filtered random field R^α

τ Large eddy turnover time

τ_w Local wall shear stress

Latin Letters

$\mathbf{a}$ Lund lower-triangular Cholesky factor matrix

a_{ij} Components of the Lund transformation matrix

$B_{i,j,k}$ Discrete three-dimensional filter kernel

$\tilde{b}_s$ Unnormalised one-dimensional filter coefficient at index s

b_s Normalised one-dimensional filter coefficient at index s

$c_f(x)$ Local skin friction coefficient

C_f Global skin friction coefficient

u_τ Friction velocity

i_g, j_g, k_g Global logical grid coordinates in the streamwise, spanwise, and wall-normal directions

$\{IC\}$ Set of initial conditions

i, j, k Local grid indices in the streamwise, spanwise, and wall-normal directions

N_0 Index of the first zero-crossing

`ix_head` Ring-buffer head index

L Integral length-scale

n_x, n_y, n_z Normalised length scales in the streamwise, spanwise, and wall-normal directions

L_x, L_y, L_z Integral length scales in the streamwise, spanwise, and wall-normal directions

`lo(2), lo(3)` Global indices of the first interior point owned by the current MPI rank in the spanwise and wall-normal directions

$R_{ii}(r_k)$ Normalized two-point correlation of u_i' in direction x_k

N_x, N_y, N_z Filter half-widths in the streamwise, spanwise, and wall-normal directions

n_{shift} Number of planes shifted per time step

$n(1{:}3)$ Local grid dimensions on the current MPI rank

$n_g(1{:}3)$ Global grid dimensions in the streamwise, spanwise, and wall-normal directions

p Pressure

P Total number of MPI ranks

$r^{\alpha}_{i,j,k}$ Three-dimensional array of spatially uncorrelated normally distributed random numbers for velocity component α

Re_τ Friction Reynolds number

$R^{\alpha}_{I,J,K}$ Filtered random field for velocity component α

$\mathbf{R}$ Reynolds stress tensor

r_0	Separation at the first zero-crossing of the correlation
r	Separation distance
t	Time
R(r)	Transversal correlation function
$\mathbf{u}$	Velocity vector
u, v, w	Velocity vector components
U_∞	Free-stream velocity
U_{bulk}	Bulk velocity
U^α	Normalised filtered random field for velocity component α
x, y, z	Spatial coordinates / Lorenz system variables
x_0, y_0, z_0	Initial conditions for the Lorenz system
y^+	Wall-normal coordinate in wall units, literature convention
y/δ	Wall-normal coordinate normalized by half-channel height, literature convention

Mathematical Operators / Subscripts / Superscripts

$\langle \cdot \rangle$	Averaging operator
$\bar{\cdot}$	Averaging operator
$'$	Fluctuation superscript
$\int$	Integral operator
max	Maximum function
∇	Gradient / Divergence operator
∇^2	Laplacian operator
$\mathcal{O}$	Order of magnitude notation
$+$	Superscript denoting a dimensionless inner scaling quantity
$\sum$	Summation operator
$\top$	Transpose operator

Italian summary

Il problema dell'inlet turbolento.

Le simulazioni che riproducono con alta fedeltà i flussi turbolenti, come le Direct Numerical Simulation (DNS) ed i Large-Eddy Simulation (LES), richiedono condizioni al contorno di ingresso che siano tridimensionali, dipendenti dal tempo e statisticamente coerenti con il flusso che si intende riprodurre. A differenza dei solutori Reynolds-Averaged Navier-Stokes (RANS), nei quali è sufficiente prescrivere quantità medie e una scala di turbolenza, le simulazioni DNS e LES risolvono direttamente una porzione significativa dello spettro turbolento e necessitano quindi di un campo di velocità istantaneo all'ingresso. Poiché il moto dei fluidi viscosi è governato dalle equazioni di Navier-Stokes, ovvero un sistema di equazioni differenziali alle derivate parziali fortemente non lineari, la sensibilità alle condizioni iniziali ed al contorno è estrema: una perturbazione infinitesima può amplificarsi esponenzialmente nel tempo, come mostrato dal modello semplificato di Lorenz in Figura 1.1. Ne consegue che un errore anche piccolo nel campo di velocità prescritto al piano di ingresso si propaga all'interno del dominio e altera le statistiche turbolente a valle. Quanto più il campo imposto si avvicina a una soluzione delle equazioni di Navier-Stokes, tanto minore risulta la perturbazione introdotta nel campo di velocità all'interno del dominio. La generazione di condizioni di ingresso turbolente realistiche rappresenta quindi uno dei problemi centrali delle simulazioni spazialmente sviluppate, ed è l'argomento principale di questa tesi.

Le famiglie di metodi disponibili.

La letteratura propone tre grandi famiglie di approcci. I metodi *library-based* estraggono piani di velocità da un database pre-esistente, ottenuto da una simulazione precursore o da una simulazione concorrente, oppure da un database esterno riscalato sulla base di una soluzione RANS. I metodi *recycling-rescaling* campionano un piano di velocità a valle, lo riscalano tramite dei parametri di riferimento e lo reinseriscono all'ingresso, evitando la simulazione ausiliaria ma introducendo una periodicità spuria che può contaminare la fisica a bassa frequenza. I metodi *synthetic turbulence generators* -

generatori di turbolenza sintetica - costruiscono invece un campo artificiale che riproduce le proprietà statistiche della turbolenza voluta senza dipendere da dati preesistenti, se non, ad esempio, i profili medi di velocità e gli sforzi di Reynolds. Questi ultimi sono generalmente la scelta più pratica: consentono la generazione di campi di velocità in tempo reale, facilmente parallelizzabili e privi di periodicità spuria, al costo di una distanza di adattamento a valle dell'ingresso, necessaria perché il campo sintetico diventi una soluzione fisicamente consistente delle equazioni di Navier-Stokes.

Il metodo del filtro digitale.

L'algoritmo specifico adottato in questo lavoro è il filtro digitale proposto da Klein, Sadiki e Janicka [1]. L'idea centrale è quella di convolvere un insieme di numeri casuali Gaussiani, spazialmente incorrelati, con un filtro passa-basso la cui larghezza identifica la scala integrale di turbolenza. La forma del filtro è scelta in modo che la funzione di autocorrelazione del campo filtrato coincida con il modello Gaussiano derivato da Batchelor per turbolenza omogenea a stadi di decadimento avanzati. Una successiva trasformazione lineare, basata sulla decomposizione di Cholesky del tensore degli sforzi di Reynolds secondo la formulazione di Lund, Wu e Squires [2], mappa i tre campi filtrati indipendenti sulle fluttuazioni anisotrope desiderate. La versione originale presenta però due limiti rilevanti: la convoluzione tridimensionale ha un costo dell'ordine di $\mathcal{O}(L^3)$ e la memorizzazione dell'intero campo casuale per tutta la durata della simulazione esaurisce rapidamente la memoria disponibile, entrambi risolvibili.

La formulazione low-storage di Kempf.

Kempf, Wysocki e Pettit [3] hanno dimostrato che il kernel del filtro si separa nel prodotto tensoriale di tre filtri monodimensionali b_x, b_y e b_z , riducendo il costo computazionale da $\mathcal{O}(L^3)$ a $\mathcal{O}(3L)$ e permettendo di sostituire la convoluzione tridimensionale con tre passaggi monodimensionali successivi. Il processo è semplice: per ciascuna delle tre componenti di velocità del flusso - u, v, w - viene generato un campo di numeri casuali Gaussiani spazialmente incorrelati, che viene poi filtrato lungo le tre direzioni spaziali applicando in sequenza i filtri b_x, b_y e b_z. A questo si aggiunge:

- una strategia di rolling buffer, ovvero un buffer circolare che mantiene in memoria solo i $2N_x + 1$ piani casuali strettamente necessari al filtraggio in direzione streamwise: a ogni passo temporale il piano più vecchio viene sovrascritto da uno nuovo, senza che l'intero campo casuale debba essere conservato per tutta la durata della simulazione;

- uno schema di seeding deterministico del generatore di numeri casuali, che permette a ogni processo MPI di ricostruire localmente il valore corretto senza scambio di messaggi.

La combinazione di queste due ottimizzazioni rende il metodo efficiente in memoria e banalmente parallelizzabile: il seeding deterministico elimina la necessità di comunicazione tra processi, mentre la parallelizzazione su memoria condivisa e su GPU è affidata alle interfacce OpenMP e OpenACC. Questa è la formulazione adottata in questa tesi.

Il prototipo Python e la sua validazione.

L'algoritmo è stato implementato dapprima in Python, come classe chiamata `InflowGenerator`, con l'obiettivo di validare la formulazione matematica, la strategia di interpolazione, la trasformazione di Lund e la gestione del buffer in un ambiente flessibile prima di affrontare la portabilità in Fortran. Particolare attenzione è stata dedicata alla gestione di griglie non uniformi nella direzione normale alla parete: poiché la griglia fisica è tipicamente discretizzata in modo non uniforme, il filtraggio viene eseguito su una griglia ausiliaria uniforme, dove la larghezza del kernel è parametrizzata dalla scala integrale espressa in unità di celle, $n_i = L_i/\Delta_i^{\text{uni}}$, e il campo risultante viene successivamente interpolato sulla griglia di destinazione tramite spline cubiche. Il prototipo è stato validato rispetto i dati di riferimento di Moser, Kim e Mansour [4] per un canale turbolento a $Re_\tau = 180$, con $N_t = 1000$ passi temporali e 20 diverse realizzazioni con seed diverso per separare il rumore statistico dal comportamento sistematico dell'algoritmo. I risultati mostrano che il profilo di velocità media viene recuperato con un errore massimo inferiore all'1% nella media d'insieme, e che le componenti del tensore degli sforzi di Reynolds riproducono correttamente sia la forma sia la posizione dei picchi, con uno scostamento sistematico solo nella regione di buffer, attribuibile alla forma Gaussiana del kernel e all'uso di una scala di lunghezza costante lungo la direzione normale alla parete.

La portabilità in Fortran e l'integrazione in CaNS.

Il passaggio dal prototipo Python al modulo Fortran `mod_tig`, integrato nel solver DNS CaNS, ha richiesto la riscrittura esplicita di tutte le operazioni che in Python erano delegate alle librerie NumPy e SciPy: le tre convoluzioni, l'interpolazione a spline cubiche e la generazione dei numeri casuali. A questa traduzione si sono aggiunte tre ottimizzazioni sostanziali. La prima è il ring buffer con testa mobile: invece di traslare fisicamente l'intero array a ogni passo temporale, si mantiene un indice che punta al piano più vecchio e si sovrascrive quello, riducendo il costo di gestione del buffer da $\mathcal{O}(N_x L_y L_z)$ a $\mathcal{O}(L_y L_z)$ per passo, facendo avanzare l'indice mobile ogni volta che un nuovo piano viene generato. La seconda è l'avanzamento del buffer secondo l'ipotesi di Taylor, $n_{\text{shift}} = \lfloor U_{\text{bulk}} \Delta t/\Delta x \rfloor$, con accumulo della parte frazionaria tra passi successivi: questo permette di seguire il passo temporale adattivo di CaNS e di imporre il corretto tasso di avvezione fisica del piano di ingresso. Se il passo temporale non è abbastanza grande, ovvero se $U_{\text{bulk}} \Delta t/\Delta x < 1$,

allora $n_{\text{shift}} = 0$ e il buffer non viene avanzato: il generatore riutilizza il piano di ingresso del passo precedente, che è la scelta fisicamente corretta, perché per $\Delta t \ll \Delta x/U_{\text{bulk}}$ la turbolenza trasportata attraverso il piano di ingresso è praticamente immutata. La parte frazionaria non viene però scartata, ma resta accumulata in `x_shift_accum` e contribuisce all'avanzamento non appena supera l'unità, cosicché il tasso di convezione media nel tempo rimane esatta e non si accumula alcuna deriva. La terza è l'estrazione del sotto-blocco locale, incluse le celle *halo*, direttamente dal piano globale di ingresso tramite un semplice scorrimento di indici, che elimina completamente la comunicazione tra processi all'interno del generatore. Il modulo è parallelizzato con OpenMP e OpenACC e comunica con il solver attraverso il tipo derivato `turbulentInlet_type`. L'eliminazione della comunicazione tra processi deriva dal fatto che ogni processo genera autonomamente il campo di velocità completo di dimensione $M_y \times M_z$ e successivamente ne estrae soltanto la porzione di dati necessaria ai calcoli successivi. Questa scelta comporta un aumento voluto del costo computazionale, ma in cambio si risparmia interamente sul tempo di comunicazione tra processi.

La simulazione principale.

Prima della simulazione con inlet sintetico turbolento sono state eseguite due simulazioni preliminari di canale doppiamente periodico. La prima, con la stessa discretizzazione di MKM, ha confermato che CaNS riproduce le statistiche di riferimento entro l'1% sulle quantità integrali, pur confrontando dati generati con due tipologie di codici diversi, uno di tipo spettrale, quello di Kim, Moin e Moser [5], e uno ale differenze finite, ovvero CaNS [6]. La seconda, con la discretizzazione in direzione normale alla parete di Pirozzoli e Orlandi a 100 punti, ha verificato che lo stretching utilizzato nel setup di destinazione finale non degrada l'accuratezza della soluzione, riducendo al contempo il tempo per passo temporale di circa il 27%. I risultati di questa seconda simulazione sono stati esportati come file `.dat` e utilizzati come input per il generatore, in modo che le differenze osservate nella simulazione principale fossero attribuibili all'algoritmo e non alle differenze numeriche tra i codici. La simulazione principale è stata condotta in un dominio di dimensioni $L_x \times L_y \times L_z = 18\pi \times 2\pi \times 2$ con una griglia di $1272 \times 282 \times 100$ punti, con condizioni di periodicità nella direzione spanwise, no-slip alle pareti, Neumann all'uscita e inlet turbolento generato dal filtro digitale. Il numero di punti non è casuale: come prima cosa, utilizzando 64 cores si può effettuare una decompsizione a penna - *pencil decomposition* - di dimensione 8×8. Secondariamente, questi valori combinati al numero di Reynolds Re_τ ed alle dimensioni del dominio, servono per ottenere un valore target della spaziatura di griglia in unità di parete Δi^+ . La prima fase di 60000 passi temporali ha portato il campo inizialmente quiescente a uno stato statistica-

mente stazionario, e la seconda fase di 120000 passi ha fornito la finestra di media, per un totale di circa 70 flow-through times.

Risultati e limiti.

La simulazione conferma che l'inlet digitale riproduce il flusso target entro pochi punti percentuali su tutte le quantità integrali: $U_{bulk} = 15.68$ contro 15.67, $u_\tau = 0.998$ contro 0.994, e $Re_\tau = 179.77$ contro 177.18 della baseline doppiamente periodica. Il profilo di velocità media in coordinate di parete recupera correttamente sia il sublayer viscoso sia la regione logaritmica, e le quattro componenti del tensore degli sforzi di Reynolds ($\langle u'u' \rangle^+$, $\langle v'v' \rangle^+$, $\langle w'w' \rangle^+$ e $\langle u'v' \rangle^+$) riproducono forma e posizione dei picchi su tutta l'altezza del canale. L'aspetto negativo è la distanza di adattamento: il coefficiente di attrito locale $c_f(x)$ non raggiunge il plateau prima di $x \approx 30\delta$ a valle dell'ingresso, un valore più lungo del range tipico di 10δ-20δ riportato in letteratura. Il valore asintotico è però corretto, oscillando intorno a $C_f \approx 0.008$, il che dimostra che l'inlet sintetico non altera lo stato di equilibrio del flusso ma ne ritarda soltanto il raggiungimento. La causa principale di questo ritardo è l'uso di una scala di lunghezza costante in direzione normale alla parete, che rende il kernel Gaussiano troppo largo vicino alla parete e troppo stretto nella regione esterna.

Risposte agli obiettivi.

Gli obiettivi posti nell'introduzione sono stati tutti raggiunti. Il filtro digitale è stato implementato e validato come modulo Python autonomo, capace di gestire griglie non uniformi e tensori di Reynolds anisotropi. La validazione contro i dati di riferimento a $Re_\tau = 180$ ha confermato il recupero accurato del profilo di velocità media e delle quattro componenti degli sforzi di Reynolds. La portabilità in Fortran e l'integrazione in CaNS sono state completate senza perdita di accuratezza e senza sacrificare la scalabilità parallela. La simulazione DNS del canale turbolento con inlet digitale ha prodotto risultati in accordo con i dati di riferimento entro pochi punti percentuali su tutte le quantità integrali.

Sviluppi futuri.

La validazione dell'algoritmo è attualmente completa solo a un singolo numero di Reynolds. L'estensione naturale è ripetere l'analisi a $Re_\tau = 395$, dove la separazione di scale tra regione interna ed esterna è maggiore e l'approssimazione di scala di lunghezza costante è messa alla prova più severamente. Sul fronte dell'algoritmo, la modifica più promettente è l'introduzione di scale di lunghezza dipendenti dalla coordinata normale alla parete, $n_x(z)$, $n_y(z)$, $n_z(z)$, derivate dalle scale integrali locali dei dati di riferimento: il costo aggiuntivo è minimo, perché richiede soltanto uno stencil separato per ogni posizione, precalcolato una volta sola in fase di inizializzazione, e il guadagno atteso in termini di distanza di adattamento è sostanziale. Una

seconda direzione riguarda la forma del kernel: l'uso di filtri esponenziali o della correlazione trasversale di Laguarda e Hickel [7], che include valori negativi e meglio approssima la funzione di correlazione reale di un flusso di taglio, riprodurrebbe gli spettri target con maggiore fedeltà rispetto al kernel Gaussiano. Una terza possibilità è l'adozione di una formulazione *divergence-free* come il synthetic eddy method, che eliminerebbe la piccola divergenza residua introdotta al piano di ingresso dalla correzione di pressione sullo schema staggered di CaNS. Infine, la prescrizione di statistiche di ordine superiore al secondo migliorerebbe la fedeltà strutturale del campo generato, al costo di uno sforzo computazionale e implementativo aggiuntivo. Va inoltre ricordato che le statistiche presentate sono una singola realizzazione da un seed fisso: una vera media d'insieme richiede di ripetere l'intera simulazione con un seed globale diverso e di mediare i risultati, mentre un re-seeding del generatore a ogni passo temporale non produrrebbe lo stesso effetto, perché il campo a valle conserva la memoria dell'ingresso attraverso la convezione e piani di ingresso diversi all'interno della stessa simulazione non sono realizzazioni indipendenti dello stato di equilibrio. Oltre allo studio di canali turbolenti, uno degli scopi finali è quello di utilizzare questo algoritmo per la simulazione di strati limite turbolenti.

Considerazioni finali.

Nel complesso, i risultati di questa tesi mostrano che il filtro digitale low-storage di Kempf può essere integrato in un solver DNS alle differenze finite senza perdita di accuratezza e senza sacrificare la scalabilità parallela, e che il generatore risultante produce turbolenza fisicamente realistica a $Re_\tau = 180$ con una distanza di adattamento più lunga dei valori di letteratura ma comunque accettabile per domini della lunghezza qui utilizzata. I miglioramenti elencati affrontano i limiti rimanenti e indicano la strada verso un generatore applicabile a numeri di Reynolds più elevati e a configurazioni geometriche più complesse.

Chapter 1

Introduction

Advances in processor technology over the past three decades have made it increasingly feasible to apply high-fidelity simulation techniques - Direct Numerical Simulation (DNS) and Large-Eddy Simulation (LES) - to flows of practical engineering interest. Unlike Reynolds-Averaged Navier-Stokes (RANS) solvers, which model the entire turbulent energy cascade, DNS and LES resolve a significant portion of the unsteady, three-dimensional turbulent motion directly, and therefore offer richer physical insight and higher predictive accuracy. However, this added fidelity comes with demanding requirements not only on spatial and temporal resolution, but also on the quality of the prescribed boundary conditions. Among these, the inlet boundary condition is particularly critical: in spatially developing simulations of turbulent boundary layers, jets, or channel flows, the velocity field imposed at the inflow plane determines the quality of the turbulent field established inside the domain and therefore directly affects the reliability of the entire computation.

The motion of viscous fluids is governed by the Navier-Stokes equations, a system of highly non-linear partial differential equations. As with any system of differential equations, attempting to solve them without well-posed initial and boundary conditions yields an infinite family of general solutions containing arbitrary functions, rather than a unique physical state. Therefore, the specification of precise boundary conditions is mandatory. In spatially developing turbulent flows - such as channel flows or boundary layers - the required instantaneous velocity field at the inlet is inherently unknown *a priori*. Consequently, it is critical to impose an inflow boundary condition whose spatial structure and temporal evolution closely replicate physical reality.

Several strategies exist to address this challenge. One approach relies on periodic boundary conditions in the streamwise direction, though this often requires significantly longer simulation times for the turbulence to fully

develop. Another method involves mapping time-series data saved from a precursor simulation or measurements in the laboratory, but this incurs prohibitive data storage and input/output costs. A more efficient alternative is the use of synthetic turbulence generation algorithms, which mathematically construct a boundary velocity field that closely approximates a true Navier-Stokes solution. A prominent example of this approach is the digital filter method.

The present thesis addresses the problem of generating physically realistic turbulent inflow conditions for a DNS simulation of a turbulent channel flow, which can be extended to a whole different inlet plane dependent scenarios. The main objective is to implement, validate, and integrate the digital filter method of Klein, Sadiki, and Janicka [1], together with the low-storage improvement proposed by Kempf, Wysocki, and Pettit [3], into the high-performance DNS solver CaNS [6], and to use the resulting tool to simulate a fully developed channel flow problem.

This chapter is organized as follows. Section 1.1 introduces the fundamental properties of turbulence and explains why accurate inflow conditions are indispensable in spatially developing simulations. Section 1.2 reviews and classifies the main families of inflow turbulence generation methods available in the literature. Section 1.3 specifies the specific objectives and outlines the structure of the thesis.

1.1 The Random and Chaotic Nature of Turbulence

Turbulent flow is the most common state of fluid motion encountered in nature and engineering. Observing a waterfall, the smoke plume rising from a chimney, or the wake behind a car immediately reveals the characteristic properties of turbulence: it is unsteady, three-dimensional, seemingly random, and chaotic [8]. These properties arise directly from the nonlinear character of the governing equations.

The incompressible Navier-Stokes equations for momentum and mass conservation read as

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} &= -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} \quad \text{momentum equation} \\ \nabla\cdot\mathbf{u} &= 0 \qquad\quad \text{mass conservation} \end{aligned} \tag{1.1}$$

where $\mathbf{u} = (u, v, w)$ is the velocity vector, p the pressure, ρ the (constant) density, and ν the kinematic viscosity. The nonlinear advective term $(\mathbf{u}\cdot\nabla)\mathbf{u}$

is responsible for the extreme sensitivity of the flow to initial and boundary conditions that characterize turbulence.

The last statement can be supported by Lorenz's simplified model of atmospheric convection [9], which consists in a system of three coupled ordinary differential equations

$$\begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x), \\ \dfrac{\mathrm{d}y}{\mathrm{d}t} = \rho\, x - y - xz, \\ \dfrac{\mathrm{d}z}{\mathrm{d}t} = -\beta z + xy, \end{cases} \tag{1.2}$$

characterized by three constant parameters representing the physical properties of the system: σ is the Prandtl number, ρ is the Rayleigh number and β refers to the physical dimensions of the fluid layer itself. The chaotic behavior of the system can be exhibited by choosing $\rho > 24.74$. For $\sigma = 10$, $\beta = 8/3$ and $\rho = 28$ and considering two sets of initial conditions $\{IC\}_1$ and $\{IC\}_2$, where the only difference between the two sets is a small variation in the variable x_0, it is possible to see how the evolution of the variable x, for the two sets of initial conditions IC, changes 1.1.

$$\{IC\}_1 : \begin{cases} x_0 = 0.1 \\ y_0 = 0.1 \\ z_0 = 0.1 \end{cases} \qquad \{IC\}_2 : \begin{cases} x_0 = 0.100001 \\ y_0 = 0.1 \\ z_0 = 0.1 \end{cases}$$

The solution diverges exponentially after a short initial period, as illustrated in Figure 1.1. Although the Lorenz system is far simpler than the Navier-Stokes equations, it captures the essence of chaotic sensitivity: an infinitesimally small perturbation in the initial state leads, after sufficient time, to a completely different solution.

The same sensitivity is present in solutions of the Navier-Stokes equations (1.1). Even a small error in the prescribed inlet velocity field can propagate into the domain interior and alter downstream turbulence statistics. For RANS simulations, this problem is largely bypassed because only mean quantities and a turbulence closure solving related quantities (depending on the used model, e.g. turbulent kinetic energy, dissipation rate, mixing length) must be specified at the inlet. However, high-fidelity approaches such as DNS and LES require three-dimensional, time-dependent velocity fields that not only satisfy the governing flow equations, but also accurately reproduce target turbulent statistics, energy spectra, and spatial correlations. Meeting all of these requirements simultaneously is known as the *inflow turbulence*

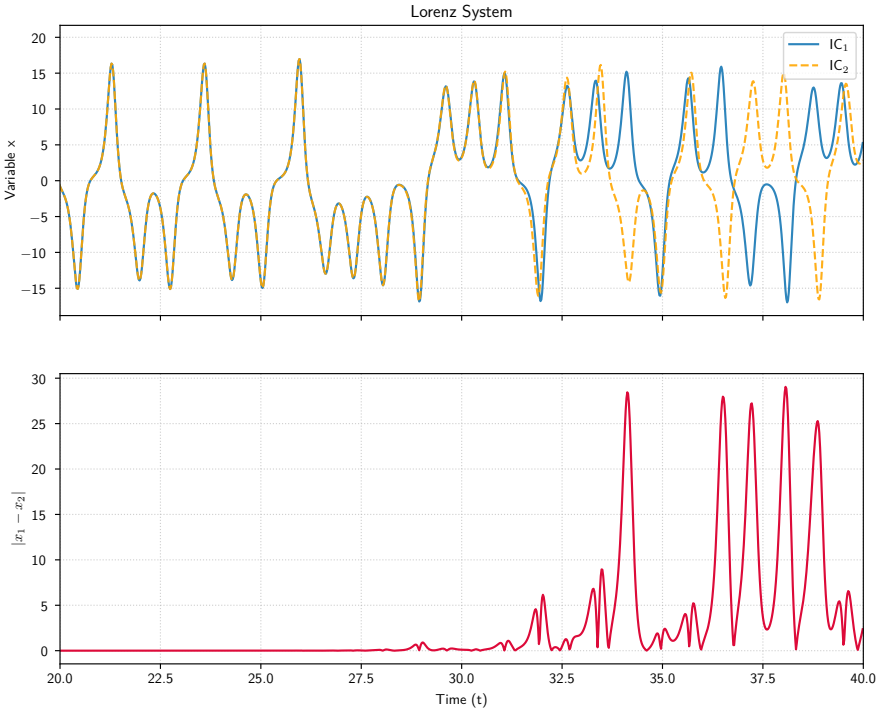


Figure 1.1: Divergence of the x-component solution of the Lorenz system (1.2) for $t > 27$.

generation problem, and it is the central topic of this thesis.

1.2 Inflow Turbulence Generation Methods

An ideal turbulent inflow boundary condition should satisfy a set of requirements. Following Dhamankar et al. [10], the most important of these can be summarized as follows:

1. **Statistical fidelity.** The method should reproduce all known statistical information about the target flow, including single and multi-point first and higher-order moments, spatial autocorrelation functions, and the spectral distribution of turbulent kinetic energy.

2. **Self-sufficiency.** The turbulence fed in at the inflow should be close enough to a fully-developed state that no long adaptation region is required downstream of the inlet plane.

3. **Absence of spurious artefacts.** The method should not introduce non-physical periodicity, spurious acoustic waves, or low-frequency modes that contaminate the downstream flow field.

4. **On-the-fly generation.** The method should be capable of producing inflow planes in real time during the simulation, so that storing large databases of pre-computed slices is not required.

5. **Grid independence.** The method should be applicable to arbitrary spatial discretizations and grid types: structured or unstructured, finite-difference, finite-volume, or finite-element.

6. **Ease of implementation.** The algorithm should be simple to code and to adapt to new flow configurations without major modifications.

7. **Parallel scalability.** Modern DNS and LES are run on hundreds to thousands of processor cores; the inflow generator must scale efficiently without introducing inter-process communication bottlenecks.

In practice, no method satisfies all seven criteria simultaneously, and the choice of approach is always a compromise determined by the specific application. The methods available in the literature can be broadly classified into three families [10, 11, 12]: library-based methods, recycling-rescaling-based methods, and synthetic turbulence generators. These are reviewed in the following subsections. The last family receives the most detailed treatment, as it is the one adopted in this work.

1.2.1 Library-Based Methods

Library-based methods prescribe the inflow velocity field by drawing instantaneous planes from a pre-existing database of turbulent flow data. The data can come from two sources.

Precursor and concurrent simulations. A separate simulation, using the same geometry and operating conditions as the main computation, is run to generate a library of inflow planes. In a *precursor* approach, this auxiliary simulation is completed first and its output is stored to file; during the main simulation, the stored planes are cycled through sequentially. In a *concurrent* approach, the two simulations run simultaneously and exchange data at each time step through a communication channel [10]. The main advantage of library-based methods is the physical realism of the prescribed data: because the database contains actual solutions of the Navier-Stokes equations, all statistical requirements are satisfied by construction. The main drawbacks are the high storage cost of precursor approaches and the complexity of implementing concurrent coupling, as well as the possibility of introducing spurious low-frequency behavior when a finite library is reused periodically [10].

Pre-existing databases. An alternative is to draw the data from an existing database generated for a different geometry or operating condition, rescaling the data to match the target turbulence statistics predicted by a RANS computation. This avoids the cost of a dedicated precursor simulation but requires careful rescaling and typically produces inflow data that needs a longer adaptation distance to develop into realistic turbulence [10].

1.2.2 Recycling-Rescaling-Based Methods

The recycling-rescaling approach, introduced by Lund, Wu, and Squires [2] and building on earlier work by Spalart [13], avoids the need for a separate simulation by generating inflow data from within the main domain itself. A velocity plane is sampled at a downstream *recycling station*, rescaled using similarity arguments appropriate for the inner and outer regions of the boundary layer, and re-injected at the inlet. The rescaling relations exploit the well-known inner scaling y^+ and outer scaling y/δ to map the fluctuations from the recycling station back to the inlet conditions.

For example, when applied to an incompressible zero-pressure-gradient flat-plate boundary layer, this method can produce a realistic, fully-developed turbulent boundary layer at the inlet with little or no adaptation distance.

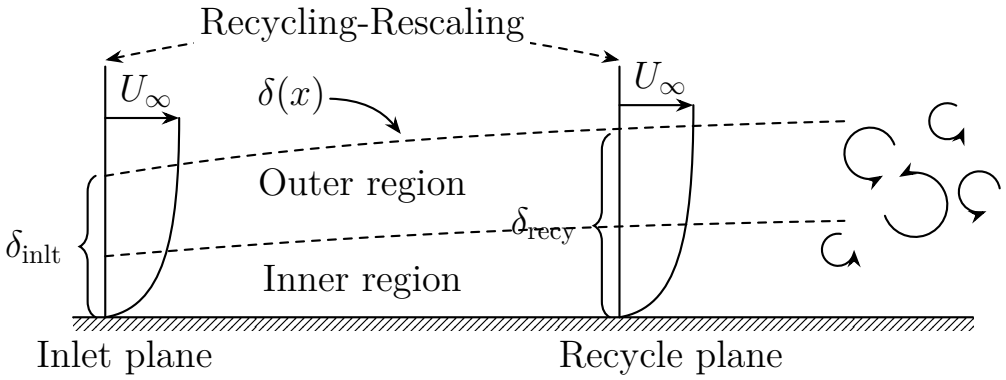


Figure 1.2: Schematic of the recycling–rescaling inflow generator.

It has since been extended to compressible boundary layers [10, 11], rough surfaces, and environmental flows. However, it suffers from an inherent flaw identified by Nikitin [14]: the streamwise periodicity of the auxiliary periodic computation is transmitted to the main simulation, producing mathematically consistent but physically unphysical spatially repetitive features. The resulting spurious low-frequency content can be particularly damaging in computational aeroacoustics and in studies of shock-wave/boundary-layer interaction, where low-frequency physics are of direct interest. Various remedies have been proposed, such as random spanwise shifts of the recycled plane [15] and dynamic multi-scale rescaling [16], but at the cost of additional implementation complexity.

1.2.3 Synthetic Turbulence Generators

Synthetic turbulence generators construct artificial velocity fields that mimic the statistical properties of real turbulence without relying on a pre-existing data source. They are generally the most practical choice for a wide range of applications because they are self-contained, easy to parallelize, and free of the spurious periodicity inherent in recycling approaches, provided that a high-quality pseudo-random number generator is used. The main disadvantage is that synthetic fluctuations are not solutions of the Navier-Stokes equations: they do not satisfy the correct pressure-strain balance or energy cascade, and therefore require a downstream *adaptation distance* over which the flow adjusts to a physically consistent turbulent state. This adaptation distance varies between methods and is one of the key figures of merit by which they are evaluated.

Following Dhamankar et al. [10], synthetic methods can be grouped into six subcategories.

Fourier-Transform-Based Methods

One of the earliest approaches to generating synthetic turbulent velocity fields, proposed by Lee, Lele, and Moin [17], is based on the inverse discrete Fourier transform. Starting from a prescribed energy spectrum and random phase angles, a velocity signal is synthesized that features the correct spectral distribution of turbulent kinetic energy. Since the resulting signal is periodic, random phase shifts must be introduced at irregular intervals to avoid unphysical periodicity. Extensions of this approach, including the Random Flow Generation (RFG) method of Smirnov et al. [18], have been widely adopted and are implemented in commercial CFD packages. The principal limitation is the requirement for a fully three-dimensional energy spectrum, which is rarely available from experiments, and the restriction to equidistant Cartesian grids imposed by the fast Fourier transform.

Proper Orthogonal Decomposition (POD)-Based Methods

POD-based methods exploit experimental or numerical databases to extract the most energetically significant coherent structures of a flow [19]. A synthetic inflow is then constructed as a superposition of these optimal modes, complemented by random noise for the incoherent small-scale content. These methods can in principle reproduce the correct phase information of the large-scale turbulent structures, but they require a high-quality reference dataset that is geometrically and dynamically compatible with the target simulation, which severely limits their applicability.

Deterministic Inner/Outer Layer Methods

Sandham et al. [20] proposed a physically motivated approach that introduces distinct deterministic disturbances within the inner and outer regions of a turbulent boundary layer. Low and high-speed streaks characteristic of the near-wall cycle are modelled using products of harmonic and exponential functions, while large coherent vortices populate the outer layer. The method is simple to implement and geometry independent, but does not enforce a prescribed Reynolds stress tensor, and comparisons by Touber and Sandham [21] showed that it can produce a spurious low-frequency mode that persists far downstream.

Digital Filter-Based Methods

The digital filter method, introduced by Klein, Sadiki, and Janicka [1], generates synthetic turbulence by convolving a three-dimensional field of spatially

uncorrelated Gaussian random numbers with a low-pass filter kernel whose width encodes the prescribed integral length-scale. The shape of the filter is chosen such that the autocorrelation function of the filtered field matches the Gaussian model for homogeneous, fully-developed turbulence derived by Batchelor [22]. A subsequent linear transformation based on the Cholesky decomposition of the Reynolds stress tensor, due to Lund et al. [2], maps the three independent filtered fields onto the desired anisotropic velocity fluctuations.

A key observation exploited by Kempf, Wysocki, and Pettit [3] is that the three-dimensional filter kernel separates into a tensor product of three one-dimensional filters. This allows the three-dimensional convolution to be replaced by three successive one-dimensional passes, reducing the computational cost from $\mathcal{O}(L^3)$ to $\mathcal{O}(3L)$ in terms of the integral length-scale L. Combined with a rolling buffer strategy that stores only the minimum number of random planes needed at any instant, and a deterministic random number seeding scheme that eliminates inter-process communication in parallel computations, the Kempf et al. modification makes the method both memory-efficient and trivially parallelizable. This efficient formulation is the one adopted in the present work and is described in full in Chapter 2.

Variants of the digital filter method have been proposed by Xie and Castro [23], who replaced the Gaussian filter kernel with an exponential one more appropriate for wall-bounded shear flows, and by Kim et al. [24], who extended this idea to all three coordinate directions (the Forward Stepwise Method, FSM). The comparative study of Bercin, Xie, and Turnock [25] showed that exponential kernels reproduce the target autocorrelation functions and energy spectra more faithfully than Gaussian ones for typical turbulent shear flows, as the Gaussian kernel produces a premature drop in spectral energy at wavenumbers above the energy-containing range. The digital filter approach has been applied successfully in diverse settings including boundary layer flows [10], combustion LES [3], and computational aeroacoustics [10].

Diffusion-Based Methods

Kempf, Klein, and Janicka [26] observed that diffusing a white-noise field by solving an unsteady diffusion equation is mathematically equivalent to convolving it with a Gaussian filter, since the Green's function of the diffusion equation is Gaussian. This observation led to an inflow generator that operates directly in physical space, making it naturally applicable to complex geometries and unstructured grids. The main disadvantage is the need to solve a partial differential equation at every time step, which can become a computational bottleneck.

Vortex Method and Synthetic Eddy Method

The Vortex Method (VM) and the Synthetic Eddy Method (SEM), introduced respectively by Sergent [27] and Jarrin et al. [28], take a structural approach: the inflow is represented as a superposition of artificial turbulent eddies convected through the inlet plane. In the SEM, each eddy is characterised by a shape function and a length scale that can in principle be matched to the physical coherent structures of the target flow. Pamies et al. [29] demonstrated that tailoring the eddy shapes to reproduce the known near-wall streaks and hairpin vortex structures of a boundary layer can reduce the adaptation distance to as little as $5.5\,\delta_{99}$. The method is geometry-independent, but ensuring a divergence-free velocity field and correctly prescribing the eddy shapes across a complex inlet require significant additional effort [10].

1.2.4 Comparison and Method Selection

Table 1.1 summarizes the most relevant properties of the three main method families against the criteria listed at the beginning of this section.

Table 1.1: Qualitative comparison of inflow turbulence generation method families. ✓ = satisfies, ∼ = partially satisfies, × = does not satisfy.

Criterion	Library-based	Recycling–rescaling	Synthetic
Statistical fidelity	✓	✓	∼
Self-sufficiency	✓	✓	∼
No spurious artefacts	∼	∼	✓
On-the-fly generation	×	✓	✓
Grid independence	✓	∼	∼–✓
Ease of implementation	∼	×	✓
Parallel scalability	∼	×	✓

For the present application - a turbulent channel flow simulated with the DNS code CaNS on a structured, domain-decomposed parallel grid - the digital filter method of Klein et al. [1] in its low-storage parallel formulation [3] is the most suitable choice. It is self-contained, generates inflow data on the fly, scales trivially across MPI processes, and requires only the mean velocity profile, the Reynolds stress tensor and three length-scales as inputs, both of which can be obtained from existing DNS databases for the target Reynolds number. Its main limitation - the requirement for an adaptation length over which the flow develops into a physically consistent turbulent state - is acceptable, provided the computational domain is sufficiently long.

This constraint is explicitly accounted for in the DNS setup of the channel described in Chapter 5.

1.3 Objectives and Thesis Structure

The overarching goal of this work is to provide high-fidelity DNS results of a turbulent boundary layer by implementing Klein's Digital Filter Method in the CaNS DNS code. The specific goals are the following:

1. Implement and validate the digital filter inflow generator of Klein et al. [1] in the low-storage, parallel formulation of Kempf et al. [3] as a standalone Python module, capable of handling non-uniform (wall-normal stretched) grids and general anisotropic Reynolds stress tensors.

2. Validate the generator against reference DNS data for a turbulent channel flow at $Re_\tau = 180$, confirming accurate recovery of mean velocity profiles, Reynolds stress components, and spatial autocorrelation functions.

3. Port the inflow generation procedure to Fortran and integrate it into the CaNS DNS code, ensuring efficient MPI parallelization and compatibility with CaNS's pressure-velocity coupling algorithm.

4. Conduct a DNS of a fully developed turbulent channel flow at $Re_\tau = 180$ in CaNS with enabled Digital Filter and compare the results to those obtained from Moser, Kim, and Mansour [4] periodic channel.

The development of the digital filter algorithm started by building it first in Python, for a simpler and faster implementation, and once validated the migration to Fortran followed. For a simpler translation and debugging of the code, a standalone Fortran module was built and later a direct integration to CaNS code was obtained.

The remainder of the thesis is organized as follows. Chapter 2 presents the mathematical foundations of the digital filter method and its low-storage implementation. Chapter 3 present the implementation in Python of the Klein's algorithm and validation of the raw inlet velocity data. Chapter 4 describes the integration of the generator into the CaNS code, covering the Fortran implementation, MPI parallelization strategy, and other strategies pointing towards a more efficient generation of the inlet data. Chapter 5 presents the DNS of the turbulent channel: computational domain, grid design, and flow statistics compared with reference data. Chapter 6 summarizes the findings, discusses limitations, and outlines directions for future work.

Chapter 2

Generation of the Turbulent Inlet Velocity Field

Generating physically realistic inflow boundary conditions is one of the central challenges in spatially-developing Direct Numerical Simulation (DNS) and Large-Eddy Simulation (LES). Unlike a channel-flow or periodic-box computation, a spatially developing simulation like a spatial turbulent boundary layer requires time-dependent velocity data to be prescribed at the inflow plane at every time step. The quality of these data directly controls how turbulence evolves inside the domain and, consequently, how accurately the downstream flow field is reproduced compared to reality.

The most rigorous approach - running a separate precursor simulation and extracting instantaneous planes of velocity - is often prohibitively expensive or geometrically incompatible with the target configuration. Simpler alternatives that superimpose uncorrelated white noise on a mean profile fail in practice: because white noise carries energy uniformly across all wavenumbers, the physically important large-scale, low-wavenumber content is absent, and the artificial fluctuations are immediately damped to zero by the governing equations, leaving the simulation effectively laminar near the inlet, thus preventing turbulence from being generated.

The digital filter method proposed by Klein, Sadiki, and Janicka [1] overcomes this limitation by convolving a field of random numbers with a carefully designed low-pass filter whose width encodes the integral length-scale of the target turbulence. The resulting field reproduces prescribed first and second-order one-point statistics as well as a prescribed autocorrelation function. Kempf, Wysocki, and Pettit [3] subsequently showed that the exponential filter kernel separates into a tensor product of three one-dimensional (1-D) filters, reducing the computational cost from $\mathcal{O}(L^3)$ to $\mathcal{O}(3L)$ and eliminating the need for inter-process communication in parallel implementations.

This chapter presents both methods in detail. Section 2.1 derives the mathematical foundations of Klein's original algorithm. Section 2.2 describes the low-storage, separable-filter variant of Kempf et al. Section 2.3 provides a brief introduction to the next chapters regarding the Python implementation of the algorithm 3 and the implementation in a Fortran module 4.

2.1 Klein's Digital Filter Method

2.1.1 Problem Statement and Motivation

In Reynolds-Averaged Navier-Stokes (RANS) modelling it is sufficient to prescribe turbulent kinetic energy and a length scale - or, for second-moment closures, a full Reynolds-stress tensor - at the inlet. In DNS and LES, however, the solver requires time-resolved, spatially correlated velocity fields that are consistent with the Navier-Stokes equations.

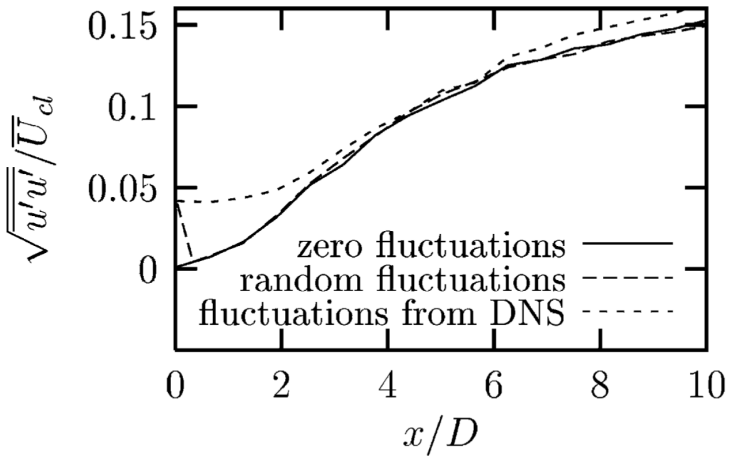


Figure 2.1: Influence of conventional inflow boundary conditions on the simulation of a plane jet. Comparison between inflow with zero fluctuations, random fluctuations and fluctuations from an external channel flow DNS. Adapted from "A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations" by Klein, Sadiki, and Janicka [1].

As demonstrated by Klein et al. [1] and illustrated schematically in Figure 2.1, an inlet condition composed of pure white-noise (random) fluctuations is equivalent to prescribing a laminar (zero fluctuations) inlet: the turbulence collapses within a few mesh spacings because the large-scale content needed to sustain it is absent. Only data drawn from a real or synthetic

source that reproduces the correct energy distribution across wavenumbers can maintain the fluctuation level and let the simulation develop the physically correct flow field.

The central idea of Klein's method is to regard the generation of synthetic turbulence as a two-step process:

1. Generate, for each velocity component $\alpha \in \{u, v, w\}$, a three-dimensional field U_i^α that possesses a prescribed two-point spatial correlation (or equivalently a prescribed length-scale and energy spectrum).

2. If the cross-correlations between velocity components must also be reproduced, apply the linear transformation derived by Lund, Wu, and Squires[2] to map the three uncorrelated fields onto the desired Reynolds stress tensor.

2.1.2 Generation of Correlated Random Fields

Let $r^\alpha_{i,j,k}$ be a three-dimensional array of spatially uncorrelated, normally distributed random numbers with zero mean and unit variance, defined on a Cartesian grid with mesh spacing Δ and global logical coordinates $(i_g,\, j_g,\, k_g)$ in the streamwise (x), spanwise (y), and wall-normal (z) directions, respectively. A spatially correlated field is obtained by convolving r^α with a discrete filter $B_{i,j,k}$:

$$R^\alpha_{I,J,K} = \sum_{i=-N}^{N} \sum_{j=-N}^{N} \sum_{k=-N}^{N} B_{i,j,k}\, r^\alpha_{I+i,\, J+j,\, K+k}, \tag{2.1}$$

where N is the half-width of the filter support. The autocorrelation function of the filtered field is

$$\frac{\overline{R^\alpha_{I,J,K}\, R^\alpha_{I+p,\, J+q,\, K+s}}}{\overline{(R^\alpha)^2}} = \frac{\displaystyle\sum_{i,j,k} B_{i,j,k}\, B_{i-p,\, j-q,\, k-s}}{\displaystyle\sum_{i,j,k} B^2_{i,j,k}}, \tag{2.2}$$

so the shape of B determines the autocorrelation structure of the output field. The right-hand side of the equation (2.2) is the discrete cross-correlation of the filter kernel with itself evaluated at lag (p, q, s); for the symmetric Gaussian kernel (2.5) this coincides with convolution, since $B_{-i,-j,-k} = B_{i,j,k}$.

Note that the three-dimensional array of spatially uncorrelated data $r^\alpha_{i,j,k}$ does not magically vanish. Since $r^\alpha_{i,j,k}$ is spatially uncorrelated and has unit variance $\overline{r^\alpha_{\mathbf{m}}\, r^\alpha_{\mathbf{n}}} = \delta_{\mathbf{mn}}$, the double sum, for each direction, collapses: the only

surviving terms are those where the index pairs coincide, i.e. $I+i = I+p+i'$ provides $i' = i - p$ (and likewise $j' = j - q$, $k' = k - s$). Therefore:

$$\overline{R^{\alpha}_{I,J,K}, R^{\alpha}_{I+p,\,J+q,\,K+s}} = \sum_{i,j,k} B_{i,j,k} B_{i-p,\,j-q,\,k-s}$$

2.1.3 Choice of Filter Kernel and Length-Scale Recovery

For fully developed homogeneous turbulence at a late decay stage, Batchelor [22] showed that the two-point longitudinal velocity autocorrelation takes the Gaussian form

$$R_{uu}(r) = \exp\left(-\frac{\pi r^2}{4L^2}\right), \tag{2.3}$$

where L is the integral length scale. Introducing the normalised length scale $n = L/\Delta$ and expressing the separation in logical coordinates, (2.3) becomes

$$R_{uu}(i,j,k) = \exp\left(-\frac{\pi\left(i^2+j^2+k^2\right)}{4n^2}\right), \quad \Delta^2\left(i^2+j^2+k^2\right) = r^2. \tag{2.4}$$

Klein et al. showed that if the filter coefficients $B_{i,j,k}$ are chosen as an exponential function of r^2, then the autocorrelation of the filtered field matches (2.4) with an error smaller than 10^{-3}, if the provided filter half-width satisfies $N \geq 2n$. The matching condition yields the unnormalised 1-D coefficients

$$\tilde{b}_s = \exp\left(-\frac{\pi s^2}{2n^2}\right), \quad s = -N, \ldots, N, \tag{2.5}$$

which are then normalised to ensure that the convolution preserves the variance of the random field:

$$b_s = \frac{\tilde{b}_s}{\left(\sum_{s'=-N}^{N} \tilde{b}_{s'}^2\right)^{1/2}}. \tag{2.6}$$

In three dimensions the full filter kernel is assembled as the outer product

$$B_{i,j,k} = b_i\, b_j\, b_k, \tag{2.7}$$

where each factor uses the length-scale appropriate to its coordinate direction (n_x, n_y, n_z). Substituting (2.7) into (2.2) reproduces (2.4), confirming that the correct integral length-scale is recovered.

2.1.4 Lund Transformation for Anisotropic Turbulence

The convolution (2.1) applied independently to three random fields r^u, r^v, r^w produces three uncorrelated fields with zero cross-correlation. To recover a prescribed Reynolds stress tensor $\mathbf{R}$ with components $R_{ij} = \overline{u_i' u_j'}$, Lund et al. [2] proposed the linear transformation

$$u_i' = a_{ij}\, U_j^{\alpha}, \tag{2.8}$$

where a_{ij} is the lower-triangular Cholesky factor of $\mathbf{R}$, i.e. $\mathbf{a}\mathbf{a}^{\top} = \mathbf{R}$. Written explicitly, the Lund Matrix is

$$\mathbf{a} = \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \tag{2.9}$$

with

$$\begin{aligned} a_{11} &= R_{11}^{1/2}, \\ a_{21} &= R_{21}/a_{11}, \\ a_{31} &= R_{31}/a_{11}, \\ a_{22} &= \Big(R_{22} - a_{21}^2\Big)^{1/2}, \\ a_{32} &= \Big(R_{32} - a_{21}a_{31}\Big)/a_{22}, \\ a_{33} &= \Big(R_{33} - a_{31}^2 - a_{32}^2\Big)^{1/2}. \end{aligned} \tag{2.10}$$

Because $\mathbf{R}$ is in general a function of the wall-normal coordinate z, a separate Cholesky decomposition is performed at each grid point in that direction. The final inflow velocity at point (J, K) of the inlet plane is

$$u_i(J,K) = \bar{u}_i(K) + a_{ij}(K)\, U_j^{\alpha}(J,K), \tag{2.11}$$

where $\bar{u}_i(K)$ is the prescribed mean velocity profile.

2.1.5 Summary of Klein's Algorithm

The complete procedure for generating one inflow time-step is:

1. Allocate and store three independent random fields r^{α} of size $(1 + 2N_x) \times (M_y + 2N_y) \times (M_z + 2N_z)$, where $M_y \times M_z$ is the size of the inlet plane.

2. Compute the filter coefficients b_s via (2.5)-(2.6) for each direction.

3. Convolve each r^α with $B_{i,j,k}$ via (2.1) to obtain the filtered fields R^α.

4. Normalize R^α to zero mean and unit standard deviation:

$$U^\alpha = (R^\alpha - \overline{R^\alpha})/\sigma_{R^\alpha}$$

5. Apply the Lund transformation (2.8) to obtain the physical fluctuations u_i'.

6. Add the mean profile $\bar{u}_i$ to obtain the total velocity (2.11).

7. Advance the stream-wise random field by one slice (Taylor's hypothesis): discard the oldest y-z plane, shift the buffer, and fill the new plane with fresh random numbers.

The main limitation of this approach in its original form is computational cost: the 3-D convolution (2.1) requires $(2N+1)^3$ multiplications per grid point, so the total cost scales as $\mathcal{O}(L^3/\Delta^3)$. For a typical DNS with $N \approx 100$, this amounts to roughly 8×10^6 operations per point per time step. Moreover, storing the three random fields for the full simulation duration can exhaust the available memory of most computing platforms. These issues are addressed by the low-storage implementation described in the following section.

2.2 Kempf's Low-Storage Implementation

2.2.1 Separable Filtering

The key observation of Kempf et al. [3] relies on a fundamental concept in digital signal processing and linear algebra: separable convolutions. Taking a step back: a 3D filter tensor $B_{i,j,k}$ is defined as separable if it can be represented as the outer product of three 1D vectors.

Because $B_{i,j,k}$ is the tensor product (2.7), the three-dimensional convolution (2.1) can be rewritten as three successive one-dimensional convolutions:

$$R^\alpha_{I,J,K} = \underbrace{\sum_{i=-N}^{N} b_i \underbrace{\sum_{j=-N}^{N} b_j \underbrace{\sum_{k=-N}^{N} b_k\, r^\alpha_{I+i,\,J+j,\,K+k}}_{\text{filter in } k\text{-direction} \to\, r^\alpha|_k}}_{\text{filter in } j\text{-direction} \to\, r^\alpha|_{k,j}}}_{} \;\to\; R^\alpha_{I,J,K}, \tag{2.12}$$

where intermediate results $r^\alpha|_k$ and $r^\alpha|_{k,j}$ denote the field after filtering in the k- and then the j-direction, respectively. This decomposition is illustrated schematically in Figure 2.2.

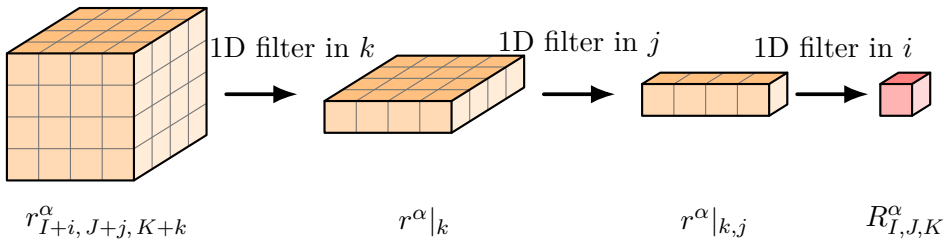


Figure 2.2: Separable filtering as three successive one-dimensional convolutions. The three-dimensional convolution (2.1) is decomposed into a sequence of 1D filters applied along the k, j and i directions, progressively reducing the working set until the final scalar $R^\alpha_{I,J,K}$ is obtained.

Evaluating (2.12) for a single point (I, J, K) requires only $3(2N + 1)$ multiplications-one pass per direction-whereas the original (2.1) needed $(2N+1)^3$. The cost reduction factor is $(2N+1)^2/3$, corresponding to approximately two orders of magnitude for a typical LES ($N \approx 10$) and four orders of magnitude for a typical DNS ($N \approx 100$). So, the application of three 1-D filters reduces computational complexity, optimize memory usage, and facilitate massive parallelization in Large-Eddy Simulations (LES) and Direct Numerical Simulations (DNS).

2.2.2 Low-Storage Rolling Buffer

Instead of storing the entire random field for the duration of the simulation, the low-storage scheme maintains only a rolling buffer of $(1+2N_x)$ inlet slices. At each time step the following operations are performed:

1. The oldest slice (index 0) is discarded; all remaining slices are shifted one position.

2. A new y-z slice of random numbers is generated and appended at the end of the buffer.

3. The buffer is filtered in the y- and z-directions using (2.12).

4. The streamwise filtering reduces to a single weighted dot product of the 1-D coefficient vector b_x with the $(1 + 2N_x)$ slices stored in the buffer.

The memory requirement is therefore proportional to $(1+2N_x) \times M_y \times M_z$ rather than to the total simulated length of the domain. For typical parameters this represents a reduction of several orders of magnitude compared with the original method.

To suppress the velocity discontinuity that arises when a new random slice enters the buffer, Kempf et al. recommend performing a linear interpolation in time between two successive inflow planes. This avoids spurious pressure oscillations and improves numerical stability.

2.2.3 Parallel Implementation Without Communication

In a domain-decomposed parallel computation, each MPI process owns a sub-block of the inlet plane. In Klein's original method this creates a communication problem: the halo region of width N surrounding each block must be filled with data from neighboring processes before the convolution can be performed.

Kempf et al. resolve this by generating the random numbers as a deterministic function of the global logical coordinates (i_g, j_g, k_g). Specifically, the integer triple is used to seed a pseudo-random number generator, so that every process independently and reproducibly computes the same random value at any given physical location. As a result, the halo region can be filled locally without any inter-process communication, making the parallelization trivially scalable.

2.2.4 Computational Cost Comparison

Table 2.1 summarizes the scaling of computational effort E and memory footprint M for the two methods.

Table 2.1: Comparison of computational effort and memory requirements.

	Klein (original)	Kempf (low-storage)
Operations per point	$(2N+1)^3$	$3(2N+1)$
Effort scaling	$\mathcal{O}(L^3)$	$\mathcal{O}(3L)$
Memory (random field)	$\sim L/\Delta \times M_y \times M_z$	$(1+2N_x) \times M_y \times M_z$
Inter-process comms	Required	None

2.3 Algorithm implementation

The mathematical formulation presented in the previous sections constitutes the theoretical foundation of the digital filter method. However, the practical implementation of the algorithm plays a fundamental role in determining its applicability to large-scale Direct Numerical Simulations (DNS). In particular, turbulent inflow generation must satisfy two often conflicting requirements: the accurate reproduction of the prescribed turbulence statistics and a computational overhead that remains negligible compared to the cost of the flow solver. For this reason, the implementation of the Klein's algorithm was developed in two sequential stages: an initial prototype was first built in Python in order to validate the mathematical formulation of the digital filter and subsequently the algorithm was directly implemented in CaNS DNS solver in Fortran which is specifically designed for high-performance computing applications.

The choice of Python as a first step toward the implementation of Klein's algorithm is driven by the major advantages of the language itself: its syntax is intuitive, and its vast ecosystem of libraries enables rapid prototyping. For example, the convolution operations required by the filter can be computed simply by calling the NumPy library, bypassing the need to explicitly write the nested `for` loops that would be required in lower-level languages. While Python is highly effective for algorithmic validation, it lacks the execution speed required for large-scale fluid dynamics simulations. Fortran, on the other hand, addresses this limitation as a strongly typed, compiled language specifically optimized for intensive numerical computations. For DNS applications, the raw computational speed of Fortran is crucial to ensure that the synthetic turbulence generation remains a negligible fraction of the total solver time. Additionally, the Fortran module is fully compatible with CaNS architecture. This seamless integration allows the algorithm to fully leverage modern high-performance computing (HPC) environments by utilizing OpenMP for multi-core CPU execution and OpenACC for GPU acceleration, thereby ensuring optimal scalability across diverse hardware architectures.

Chapter 3

Python implementation and validation of the Digital Filter's algorithm

The implementation of Klein's digital-filter algorithm in a Fortran module was preceded by a Python prototype. The main reason for this choice was practical: before integrating the algorithm into the DNS solver, it was necessary to verify the mathematical formulation, the interpolation strategy, the Lund transformation, and the low-storage buffer management in a flexible environment. Python, together with `NumPy` and `SciPy`, made this rapid prototyping possible.

The first Python version was intentionally explicit and easy to read. Only later was it optimized by vectorization and by replacing the full streamwise convolution with a single dot-product contraction. This chapter describes the development process, the main implementation choices, and the validation of the prototype against reference channel-flow data.

3.1 Motivation for a Python prototype

The final target of the work is a Fortran module coupled with the CaNS solver. Nevertheless, writing the first version directly in Fortran would have made debugging slower and more expensive. A Python prototype allowed the algorithm to be tested independently of the DNS code, with immediate access to array manipulation, interpolation, and plotting tools.

The Python prototype was therefore not meant to be production-ready. Its purpose was to answer three questions before the Fortran implementation:

- Does the digital-filter formulation reproduce the prescribed mean ve-

locity and Reynolds stresses?

- How should non-uniform wall-normal grids be handled?
- Which parts of the algorithm can be optimized without changing the mathematical result?

Once these points were verified, the optimized structure could be ported to Fortran with confidence.

3.2 Introduction to Python Prototype

The Python implementation reproduces the complete turbulent inflow generation procedure. First, a three-dimensional white-noise field is generated for each velocity component. Subsequently, the digital filtering procedure is applied independently along the three spatial directions using the filter coefficients b_i, b_j, b_k introduced in Eq. (2.12). The resulting correlated fluctuations are then transformed through the Lund decomposition in order to recover the prescribed Reynolds stress tensor.

Particular attention was devoted to ensuring that the implementation remained sufficiently general to accommodate both uniform and non-uniform meshes. Since the wall-normal direction is frequently stretched in DNS of wall-bounded turbulence, the filtering operation is performed on an auxiliary uniform grid and subsequently interpolated onto the target computational mesh. This approach guarantees that the prescribed physical length scales remain independent of the local grid spacing.

Moreover, the Python code includes several additional features aimed at improving the robustness of the implementation, including:

- reproducible random number generation through user-defined seeds;
- interpolation of Reynolds stresses onto an auxiliary uniform grid;
- boundary clamping during interpolation to avoid spurious extrapolation effects;
- generation of an `.npz` output file containg the generated velocity field data.

The generated velocity fields are stored and subsequently analyzed in terms of mean velocity profiles, velocity fluctuations and Reynolds stresses. This preliminary validation step proved the working of the algorithm before proceeding with the integration of the algorithm into the DNS solver as a Fortran module.

3.3 Python code

The inflow generator was implemented in Python 3 as the class `InflowGenerator`, defined in the file `tg3.py`. The implementation follows Klein et al.'s separable digital-filter formulation and uses a low-storage rolling buffer to avoid storing the full space–time history of the random field.

A note on the coordinate system is necessary. In the Python prototype and in CaNS, the streamwise direction is x, the spanwise direction is y, and the wall-normal direction is z. The generated arrays have shape

$$(N_t,\, M_y,\, M_z),$$

where the second index is spanwise and the third index is wall-normal. In the post-processing plots, the wall-normal coordinate is often labelled as y^+ to follow the usual literature convention, but the data array axes remain $(t, y_{\mathrm{span}}, z_{\mathrm{wall}})$.

3.3.1 Handling Non-Uniform Grids

In general, a DNS simulation requires coarser spacing in the outer region and a very fine grid spacing near the wall in order to capture the exact behaviour of the flow at the smaller scales. Applying the digital filter directly on such a stretched grid would make the physical width of the Gaussian kernel spatially varying in an uncontrolled way: the same integer half-width N would correspond to a much larger physical length-scale near the wall than at the channel center.

To avoid this inconsistency the code introduces an auxiliary *uniform* grid with M_z points spanning the same wall-normal extent as the physical grid. All filtering operations are performed on this uniform grid, where the kernel width has a well-defined physical meaning. The resulting fluctuation fields are subsequently mapped back onto the stretched physical grid by cubic-spline interpolation. The Reynolds stress tensor, which is available on the physical grid from reference DNS data, is interpolated linearly onto the uniform grid before the Cholesky decomposition is computed.

3.3.2 Filter Coefficients

The 1-D filter coefficients are computed according to (2.5)-(2.6). In code:

```
k = np.arange(-N, N + 1, dtype=np.float64)
b_tilde = np.exp(-np.pi * k**2 / (2.0 * n**2))
b = b_tilde / np.sqrt(np.sum(b_tilde**2))
```

The normalization by $(\sum \tilde{b}^2)^{1/2}$ ensures that the convolution preserves the unit variance of the white-noise input, consistent with (2.6).

3.3.3 Cholesky Decomposition

The Lund matrix **a** is computed by explicit Cholesky factorization following (2.10) at every wall-normal location independently. Diagonal entries are floored at 10^{-12} to prevent `NaN` values in regions of near-zero turbulence (e.g. the free-stream or the channel centerline in a very coarsely resolved test):

```
a[0, 0, k] = np.sqrt(max(R[0, 0], 1e-12))
a[1, 1, k] = np.sqrt(max(R[1,1] - a[1,0,k]**2, 1e-12))
a[2, 2, k] = np.sqrt(max(R[2,2] - a[2,0,k]**2
                              - a[2,1,k]**2, 1e-12))
```

3.3.4 Rolling Buffer and Optimized Streamwise Filtering

The random buffer has shape $(2N_x + 1,\ M_y + 2N_y,\ M_z + 2N_z)$. At each time step, one new y-z plane of unit-normal random numbers is appended and the oldest plane is discarded. In the current Python implementation this is done by a manual slicing shift, in the Fortran module a ring buffer is used to gain speed.

The streamwise convolution is the most expensive step when N_x is large. The original approach would apply `scipy.ndimage.convolve1d` over the full buffer axis, producing $(2N_x + 1)$ output planes and then discarding all but the central one. In the present implementation this is replaced by a single `np.tensordot` contraction:

```
x_filtered = np.tensordot(b_x, buffer, axes=([0], [0]))
```

This computes exactly $\sum_{i=0}^{2N_x} b_x[i] \cdot \text{buffer}[i, :, :]$, which is the convolution output at the central index $i = N_x$, at a cost of $\mathcal{O}((2N_x + 1) \cdot M_y \cdot M_z)$ instead of the $\mathcal{O}((2N_x + 1)^2 \cdot M_y \cdot M_z)$ cost of the full convolution. The spanwise and wall-normal passes then operate on the resulting 2-D plane:

```
temp = convolve1d(x_filtered, b_y, axis=0, mode='wrap')
temp = convolve1d(temp, b_z, axis=1, mode='constant', cval=0.0)
```

Periodic (wrap) boundary conditions are applied in the spanwise direction y, while zero-padding is used in the wall-normal direction z to avoid unphysical correlations across the wall. This is done since the DNS simulations are going to have periodic boundary conditions in spanwise direction.

3.3.5 Time-Step Loop

The main loop over N_t time steps executes the following steps:

1. For each velocity component α, compute the filtered y-z plane via the three-step procedure above: tensordot in x, convolve1d in y, and convolve1d in z.
2. Apply the Lund transformation: for each wall-normal point k, $v_i^{\text{uni}}(\cdot, k) = \sum_j a_{ij}(k) \cdot \xi_j^{\text{uni}}(\cdot, k)$.
3. Interpolate fluctuations from the uniform grid to the physical stretched grid using cubic splines.
4. Add the mean velocity profile: $u_i(J, K) = \bar{u}_i(K) + v_i^{\text{target}}(J, K)$.
5. Advance the buffer: shift by one slice and inject fresh random noise at the far end.

Output arrays of shape (N_t, M_y, M_z) are pre-allocated before the loop to avoid dynamic memory growth, and the same arrays are reused by the `compute_statistics` method to evaluate wall-normal profiles of mean velocities and Reynolds stresses without any additional allocation.

3.3.6 Input Parameters

Table 3.1 lists the key parameters of the class constructor.

Parameter	Type	Description
`nx, ny, nz`	`int`	Length-scale in uniform-grid cells per direction
`My, Mz`	`int`	Inlet plane dimensions (spanwise, wall-normal)
`z_target`	`array` (M_z)	Physical wall-normal coordinates
`u_avg`	`array` ($3 \times M_z$)	Mean velocity profiles $[\bar{u}, \bar{v}, \bar{w}]$
`R_mat`	`array` ($3 \times 3 \times M_z$)	Reynolds stress tensor
`seed`	`int`	RNG seed for reproducibility

Table 3.1: Main parameters of the `InflowGenerator` class.

The mean velocity and Reynolds stress tensor required by the generator can be obtained from experiments, from a precursor DNS/LES, or from reference simulations. In this work, they were taken from the DNS database of Moser, Kim and Mansour [4] at nominal $Re_\tau = 180$.

3.3.7 Determination of the Normalized Length Scales (n)

The normalized filter length scales (`nx`, `ny`, `nz`) provided to the generator are derived from the physical integral length scales of the flow. To obtain these integral length scales (L_i), the two-point spatial correlations extracted from the reference DNS data [5, 4] are integrated.

For a velocity fluctuation component u_i' evaluated in a spatial direction x_k, the normalized two-point correlation is defined as:

$$R_{ii}(r_k) = \frac{\langle u_i'(x)\, u_i'(x + r_k)\rangle}{\langle u_i'(x)^2\rangle} \tag{3.1}$$

where r_k is the spatial separation distance along x_k. The integral length scale $L_{ii,k}$ is computed by integrating this correlation curve from zero separation up to r_0, which represents the point of the first zero-crossing:

$$L_{ii,k} = \int_0^{r_0} R_{ii}(r_k)\, \mathrm{d}r_k \tag{3.2}$$

The integration is deliberately truncated at the first zero-crossing ($R_{ii}(r_0) = 0$). In bounded flows, such as a turbulent channel, spatial correlations inevitably develop negative lobes at larger separation distances due to mass conservation (return flows) and periodic domain boundaries. Including these negative regions in the integral would artificially reduce the calculated length scale, failing to properly represent the size of the primary energy-containing eddies. The numerical integration of the discrete DNS data up to this crossing point is performed using the trapezoidal rule,

$$L_{ii,k} \simeq \sum_{j=0}^{N_0-1} \frac{R_{ii}(r_j) + R_{ii}(r_{j+1})}{2}\, \Delta r_j, \tag{3.3}$$

with the summation truncated at $r_{N_0} = r_0$.

Once the physical integral length scale L_i is determined for a given direction, it must be mapped to the auxiliary uniform grid on which the digital filter operates. The normalized length scale n_i (representing the filter half-width in terms of grid cells) is obtained by dividing the physical length scale by the uniform grid spacing Δ_i in that direction, and rounding to the nearest integer:

$$n_i = \mathrm{round}\left(\frac{L_i}{\Delta_i}\right), \tag{3.4}$$

subject to Klein's condition $N_i \geq 2n_i$, which fixes the actual filter support as $N_i = \max(2n_i, 4)$. In the context of the `InflowGenerator`, the parameters

`nx`, `ny`, and `nz` directly correspond to these computed n_i values, dictating the stencil size of the spatial filter applied to the random field.

It is important to note that, for the wall-normal direction, Δ_i refers to the spacing of the auxiliary uniform grid introduced in Section 3.3.8, and not to the local spacing of the stretched physical mesh. Since all filtering operations are performed on the uniform grid, this is the only spacing that enters the definition of n_i. The physical length scale L_i is therefore preserved regardless of the local stretching of the target mesh, and the interpolation back onto the physical grid is carried out only after the filtering step, as described in the previous section.

Numerical values. Applying this procedure to the reference data yields the values reported in Table 3.2. For the streamwise direction, the integral length scale is $L_x \approx 1.1953$, which with $\Delta_x \approx 0.0982$ gives $n_x \approx 12$. For the spanwise direction, the integral length scale is $L_y \approx 0.0952$, which with $\Delta_y \approx 0.0491$ gives $n_y \approx 1.9$. This value was rounded *up* to $n_y = 2$, so that the filter support is never smaller than the physical correlation length of the underlying flow. The integral length scales used to size the filter are taken from the streamwise-velocity auto-correlation, L_x^{uu} and L_z^{uu} evaluated at the wall-normal position $y^+ \approx 12$ (from Moser et al. coordinate system). The choice of this wall-normal position relies on the peak of kinetic energy production. The peak is located at $y^+ \approx 12$, as is possible to observe in Figure 3.1.

These values are consistent with those commonly reported in the literature. In the digital-filter framework of Klein et al. [1] and Xie and Castro [23], the length-scale parameters are obtained by dividing the physical integral length scale of the flow by the grid spacing in the corresponding direction. For wall-resolved simulations of wall-bounded turbulence, this ratio typically falls in the range $5 \lesssim n_x \lesssim 30$ in the streamwise direction and $2 \lesssim n_{y,z} \lesssim 15$ in the transverse and wall-normal directions, reflecting the anisotropy of the integral scales. The values $n_x = 12$ and $n_y = 2$ obtained above fall within these bounds.

The wall-normal length scale requires a separate argument, because it cannot be obtained from the reference data in the same way. The required quantity is the two-point correlation of w' along z,

$$L_z(z) = \int_0^{r_0(z)} R_{ww}(r_z; z)\, \mathrm{d}r_z, \tag{3.5}$$

which is not available in the database of Moser, Kim and Mansour, since that database provides only one-point statistics. An estimate based on the isotropic proportionality $L_z \approx (2/3)L_x$, sometimes quoted for homogeneous

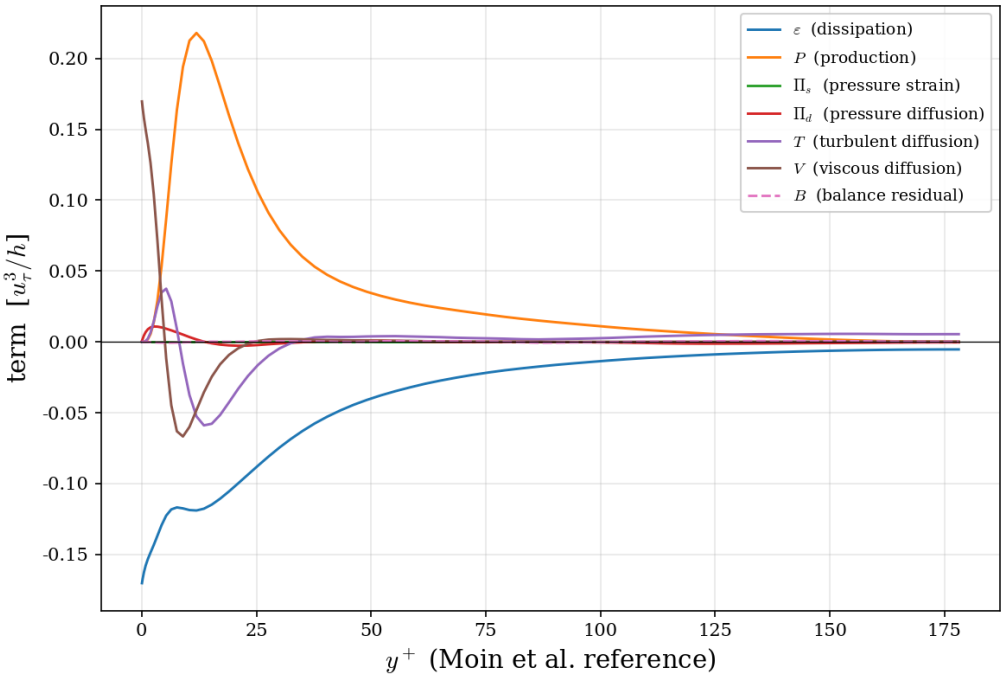


Figure 3.1: Profiles of terms in the balance equation for kinetic energy.

turbulence, would give $L_z \approx 0.7968$ and hence, with the auxiliary grid spacing $\Delta_z^{\text{uni}} = 2/(M_z - 1) = 0.015625$, a value of $L_z \approx 51$. This value is not consistent with the physics of wall-bounded turbulence, where the wall-normal correlation of w' decays much faster than the streamwise one, and it would produce a filter support $2N_z + 1 = 205$ larger than the entire channel height. It also lies well outside the general bound $3 \lesssim n_z \lesssim 15$ reported above.

The value of n_z was therefore chosen from wall-unit literature values for channel flow at $Re_\tau \approx 180$, where the wall-normal integral length scale of the wall-normal velocity component lies in the range $L_z^+ \approx 30$–50. With the auxiliary grid spacing above and with $u_\tau = 1$, $\delta = 1$, $Re_\tau = 180$, this corresponds to $n_z \approx 10$–18. The value $n_z = 12$, corresponding to $L_z^+ \approx 34$, was retained. It lies in the middle of the physical range, satisfies the general bound $3 \lesssim n_z \lesssim 15$, and keeps the filter support $2N_z + 1 = 49$ well within the wall-normal extent of the domain.

3.3.8 Uniform auxiliary grid

The auxiliary uniform grid is not a mathematical requirement of the digital-filter method. The filter could be applied directly on the stretched physical

mesh, but the physical width of the Gaussian kernel would then vary implicitly with the local grid spacing, and the parameters (n_x, n_y, n_z) would lose their unambiguous physical interpretation. Since the purpose of the Python prototype is validation rather than optimization, the simpler and more transparent route was chosen.

The auxiliary grid spans the same wall-normal extent as the physical grid and is uniformly spaced,

$$z_j^{\text{uni}} = z_0 + (j-1)\,\Delta_z^{\text{uni}}, \qquad \Delta_z^{\text{uni}} = \frac{z_{M_z} - z_0}{M_z - 1}, \qquad j = 1, \ldots, M_z. \tag{3.6}$$

Where z_0 and z_{M_z} are directly taken from the input reference data. In case of a plane channel, usually, z_0 corresponds to 0 and z_{M_z} corresponds to the full channel height 2δ. The Reynolds stress tensor is interpolated linearly onto this grid, and the digital filter, the Lund transformation, and the rolling-buffer update are performed entirely on it, so that the kernel half-widths n_i correspond to a well-defined physical length $\Delta_i^{\text{uni}} n_i$ at every wall-normal position.

The resulting fluctuation field is then mapped back onto the physical grid by cubic-spline interpolation, and the mean velocity profile, which is defined on the physical grid, is finally added. The only approximation introduced by this strategy is the interpolation error between the two grids, which is negligible because the fluctuations are correlated over a length scale L_z much larger than Δ_z^{uni}.

The advantage of this approach is that the code remains compact: the filter coefficients are computed once, the same stencil is applied at every wall-normal position, and no position-dependent kernel has to be recomputed at each time step. The results presented in Section 3.4.1 show that this choice is fully adequate, since the mean velocity is recovered within less than one wall unit and the Reynolds stresses match the reference DNS data to within a few percent. The additional complexity of a position-dependent kernel would therefore not have been justified at the prototyping stage.

It should be noted that the auxiliary grid removes the ambiguity in the definition of n_i, but it does not remove the approximation that the same n_i is used at every wall-normal position. The introduction of wall-normal-dependent length scales $(n_x(z), n_y(z), n_z(z))$ is discussed in Section 3.4.1. Further discussions on wall-normal-dependent length scales are present in the Fortran Chapter 4 and Results Chapter 5.

3.4 Generated data and validation

In order to validate the generated velocity fields, some well-known results in turbulence were usede. The data provided as input to the code - mean velocity profiles and Reynolds stress tensor - was directly taken from results obtained by Moser, Kim and Mansour [4] performed using the spectral numerical method of Kim, Moin and Moser [5]. As anticipated, the reference simulation is at nominal $Re_\tau = 180$. The table 3.2 lists different parameters used for the generation of the velocity filed. Numerical values are descend directly from calculations and motivations in Paragraph of Numerical values 3.3.7.

Table 3.2: Main parameters for the generation of the velocity field.

Parameter	Value
L_x, L_y, L_z	1.1953, 0.0952, 0.7968
n_x, n_y, n_z	12, 2, 12
N_t	1000
`My, Mz`	128×129

The wall-normal coordinates were also taken directly from the reference data set. The half-channel height was $\delta = 1$, and the friction velocity was $u_\tau = 1$. Since $Re_\tau = \frac{u_\tau \delta}{\nu}$, these values give $\nu = \frac{1}{Re_\tau}$. For $Re_\tau = 180$, this gives $\nu \approx 0.00556$.

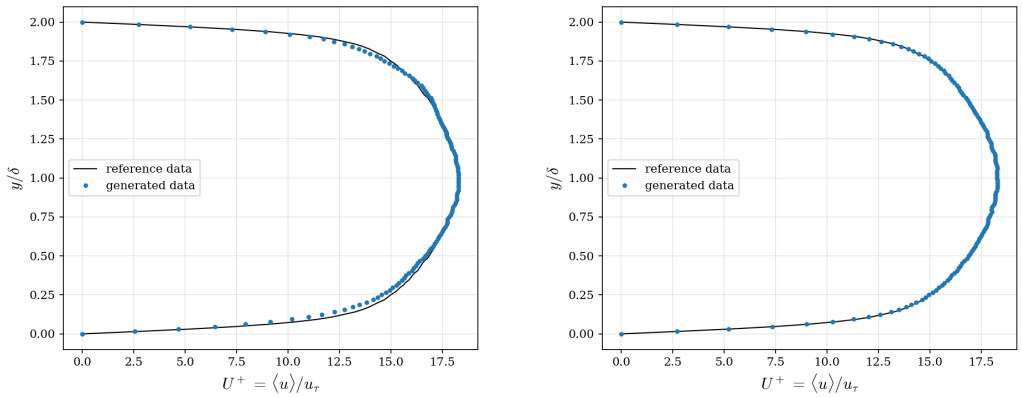
3.4.1 Results

The Python prototype was run for $N_t = 1000$ time steps. A single run required approximately 15 seconds on the available machine, which is fast enough for validation purposes. In order to separate the statistical noise of the random forcing from the systematic behaviour of the algorithm, the generator was run 20 times, each time with a different integer seed passed to the `seed` argument of the constructor, and the resulting time-averaged profiles were further averaged across the 20 realisations. All plots shown in this section refer to this ensemble average, unless explicitly stated otherwise. The single-seed counterparts are shown for comparison in Figures 3.2a, 3.3a, and 3.4a.

Mean velocity profile. Figure 3.2b shows the mean streamwise velocity U^+ as a function of the channel height y/δ, while Figure 3.3b shows the same quantity against y^+ on a logarithmic axis. In the outer layer, $y^+ > 50$, the

agreement is excellent: the log-law is respected with a perfect match between the generated data and the reference data. Indeed, as expected since the mean profile is imposed directly through the u_{avg} input. The same is true in the viscous sublayer, $y^+ < 5$, where the generated points collapse onto $U^+ = y^+$. The single-seed generated mean velocity profile in Figure 3.3a shows more or less the same behaviour with a better match to the reference data in the overlap region, $y^+ > 50$.

The interesting region is the buffer layer, $5 < y^+ < 30$. In the single-seed plot (Figure 3.3a) a small but systematic under-prediction compared to the reference data of about 8% is visible. This value is higher than the expected average error in literature but, when the focus is moved towards the ensemble average of the mean velocity (Figure 3.3b), this error drastically reduces to a maximum value of less than 1%, which is perfectly acceptable. This confirms that the single-seed discrepancy was a finite-sampling effect: although the mean velocity is imposed by construction, the measured mean contains the time average of the fluctuation field, which does not vanish exactly over a finite window. By implementing the algorithm with a prescribed wall-normal varying length scales, the maximum error on a fixed seed velocity generation should be expected lower.

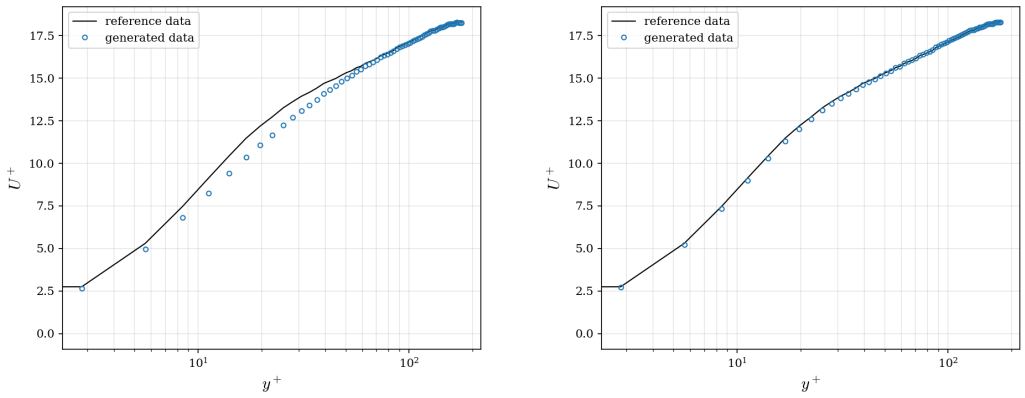


(a) Single seed.

(b) Ensemble average over 20 seeds.

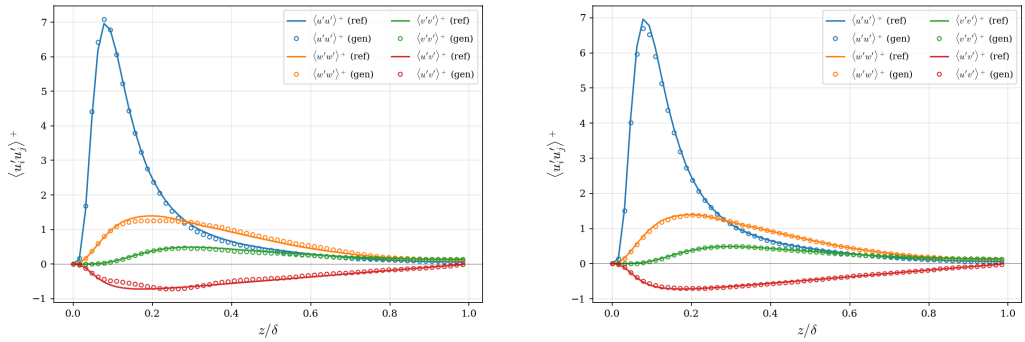
Figure 3.2: Mean streamwise velocity as a function of the channel height.

Reynolds stresses. Figure 3.4b shows the four relevant components of the Reynolds stress tensor, normalised in wall units, for the ensemble average. As in the single-seed case, all second-order moments are of the correct order of magnitude, the qualitative shape of every profile is captured, and the anisotropy between the normal components is reproduced. What changes



(a) Single seed. (b) Ensemble average over 20 seeds.

Figure 3.3: Mean streamwise velocity as a function of y^+ (logarithmic axis).



(a) Single seed. (b) Ensemble average over 20 seeds.

Figure 3.4: Reynolds stress tensor components.

with respect to the single-seed case is the scatter: the ensemble-averaged profiles are visibly smoother, and the residual differences with respect to the reference data are now systematic rather than random. In both cases, the streamwise stressses $\langle u'u' \rangle^+$ are well matched to the reference data of Moser, Kim, and Mansour in the outer layer and which peaks in the reference data at $y/\delta \approx 0.08$ with a maximum of about 6.9. As already described in the previous paragraph, there is an under-predicion in the stress in the buffer layer for the single-seed data. The ensemble average exhibits in the near wall region a really low underprediction but this is consistent with the well-known tendency of the Gaussian digital filter to under-predict the streamwise fluctuations when the prescribed autocorrelation is Gaussian while the true one is not. Anyways, the Gaussian filter provides a really good match to the generated data.

The wall-normal stress $\langle w'w' \rangle^+$ peaks in the reference data at $y/\delta \approx 0.2$ with a maximum of about 1.4. The generated profile peaks at a slightly smaller wall distance and, in the outer region $y/\delta \gtrsim 0.4$, lies systematically above the reference. The position of the peak is therefore shifted toward the wall. As in the single-seed case, this is a direct consequence of using a constant wall-normal length scale n_z: the Gaussian filter is too wide near the wall, where the true turbulent scales are small, and too narrow in the outer region. As a result, the stress peak spreads out. This shifts the highest stress point closer to the wall and overestimates the stress in the outer layer. Averaging the data removes random noise but does not fix this shift, proving it is a consistent error.

The spanwise stress $\langle v'v' \rangle^+$ is the best-reproduced component in both the single-seed and the ensemble-averaged plots. Both the peak value and its location match the reference data closely. This is consistent with the fact that the spanwise direction is statistically homogeneous: the periodic assumption of the filter is well satisfied, and the correlation structure along y is closer to the Gaussian model used by the algorithm.

The Reynolds shear stress $\langle u'v' \rangle^+$ shows the same position mismatch as $\langle w'w' \rangle^+$. The reference minimum of about -0.75 is located at $y/\delta \approx 0.17$, while the generated minimum is displaced a little further away from the wall. Its magnitude is otherwise well reproduced. Since the shear stress is the quantity that closes the mean momentum balance, the correct position of its peak is important for the downstream evolution of flow after the inlet. This suggests that in the DNS simulation there is going to be a distance between the inlet and the fully-developed condition where the flow is going to evolve towards it. The displacement observed here is a known limitation of constant-length-scale digital-filter implementations and is one of the main motivations for the wall-normal-dependent length scale $n_z(z)$ discussed in

the following paragraphs

Role of the random seed. The comparison between the single-seed and the ensemble-averaged results is instructive in itself, because it separates the two sources of error that affect the generated statistics. On one hand, the finite number of time samples introduces a random error that is controlled by the correlation time of the signal and by the length of the averaging window; this error affects mainly the lower-order moments of the fluctuations and is visible in the buffer layer of the single-seed mean velocity profile. On the other hand, the mathematical structure of the algorithm introduces a systematic error that is independent of the sampling, namely the constant-length-scale approximation, the Gaussian filter shape, and the Lund transformation. This error survives the ensemble average and is visible as small deficit in $\langle u'u' \rangle^+$, $\langle w'w' \rangle^+$ and $\langle u'v' \rangle^+$ near the wall region in the range of $0.08 < z/\delta < 0.2$.

Even by increasing the number of generated time steps or the number of different-seed generated data, there is going to be still some sort of error. The error can be lowered with different techniques - prescribing wall-normal dependency of the length scales or providing higher order correlations - but it cannot be eliminated entirely.

3.4.2 Limitations of the prototype

The main limitation of this Digital Filter implementation is that the generated velocity components (u, v, w) do not match the reference data perfectly. This is expected. The digital-filter method and related approaches such as those discussed in [25] use a Cholesky decomposition of the target Reynolds stress tensor to scale a normalized stochastic signal. This forces the second-order moments to match the prescribed data, but it does not guarantee that all structural features of the turbulence are reproduced exactly.

The following reasons explain the remaining differences and indicate possible improvements:

- **Lund Transformation Distortions**: The matrix transformation used to scale the random fields to the target Reynolds stress tensor can inadvertently distort pre-existing statistics. This specifically occurs in the statistically inhomogeneous lateral directions of a flow when anisotropic Reynolds stresses are present. While these deviations are normally small, they prevent the generated data from being an absolutely perfect replication of the target statistics [1, 25].
- **Constant Length Scale Approximations**: The Python prototype uses constant normalized length scales (n_x, n_y, n_z) across the entire

wall-normal direction. This means that the generated structural shape relies on a uniform eddy size, which does not accurately reflect the physical variation of turbulence at different wall-normal positions. So, defining different length scales as a function of the wall-normal position like $(n_x(z), n_y(z), n_z(z))$, would significantly improves the overall reconstruction fidelity of the generated correlation functions and power spectra [1, 25].

- **Non Divergence Free algorithm**: The standard digital filter method does not produces a strictly divergence-free velocity field. Implementation of a divergence-free synthetic eddy method (DFSEM) would allow the algorithm to generate a strictly divergence-free velocity field at the inlet while still successfully reproducing Reynolds stress anisotropy [30].

- **Exponential Correlation Functions**: The choice of the filter shape directly alters the generated data. Using an exponential correlation function rather than a Gaussian one - like the current implemented one - produces better inlet data, as exponential function matches more closely the true correlation functions observed in typical turbulent shear flows [23]. Another way to generate correlation with higher fidelity is to use the correlation function from Laguarda and Hickel [7]. The function is used only for transversal correaltion and is presented in the formulation
$$R(r) = \left(1 - \frac{r}{I_L}\right) exp\left(-\frac{r}{I_L}\right) \tag{3.7}$$
which includes negative values and thus much better resembles transversal correlation function.

These limitations are known but the algorithm, as demonstrated through literature, works in practice. The results presented in this chapter show that, even with the reduced value of $n_z = 12$, the generated statistics match the reference DNS data closely with an error of few percent.

The results shown answers to the three questions of the first Section:

- The digital filter does reproduce the prescribed mean velocity and Reynolds stresses.

- Non-uniform wall-normal grids are not an issue and the calculations are made on an auxiliary uniform grid.

- The rolling buffer can be stored as a ring buffer with a moving head index instead of being physically shifted every time step, and the separable filter can be evaluated only at the interior points that survive

the padding removal. Both reformulations leave the mathematical operations unchanged and are the basis of the Fortran implementation.

A further modification in Fortran, however, does change the mathematical result: the rolling buffer is advanced by $n_{\text{shift}} = U_{\text{bulk}}\Delta t/\Delta x$ planes per time step, with fractional remainder accumulated between steps, instead of one by one plane per time step as in the Python prototype. This solves two major issues:

- it accommodates the adaptive time step in CaNS instead of a fixed one and
- it enforces the correct physical advection rate of the inflow plane which provides the temporal correaltions of the generated velocity field.

Chapter 4

Implementation and working principle of the algorithm in CaNS

The previous Chapter 3 introduced the digital filter algorithm in Python and validated the generated velocity field against the reference data of a fully developed turbulent channel at $Re_\tau = 180$. The results confirmed that the mathematical formulation, the Lund transformation, and the low-storage rolling buffer produce physically realistic inflow data with errors of only a few percent in the near-wall region. The Python prototype therefore fulfilled its purpose: it answered the three validation questions posed at the beginning of Chapter 3 and provided a reliable reference implementation against which the Fortran module could be tested.

The present chapter describes the porting of that validated algorithm into CaNS as a new Fortran module, `mod_tig`, together with the modifications required by the solver architecture. The transition from a high-level prototyping language to a compiled, MPI-parallel DNS solver is not a mere translation exercise. Python offers automatic memory management, vectorized array operations, and library functions for convolution and interpolation; Fortran offers none of these conveniences but provides the raw computational speed required for large-scale simulations. Every filtering pass, every interpolation, and every random-number generation must therefore be written explicitly, and every operation that was implicit in Python must be made explicit in Fortran.

Beyond the language change, three additional requirements shape the Fortran implementation. The first is memory efficiency: the Python prototype stores the full space–time history of the generated field, which is acceptable for 1000 time steps but not for a production DNS with 10^5 or more steps.

The second is parallel scalability: the inflow generator must operate on a domain-decomposed grid, where each MPI rank owns only a sub-block of the inlet plane, and the generator must not introduce inter-process communication bottlenecks. The third is temporal accuracy: the Python prototype advances the buffer by one plane per time step, which is valid only for a fixed time step and a unit CFL-like ratio; in CaNS the time step is adaptive, and the buffer must advance according to the physical advection rate of the inflow plane.

This chapter is organized as follows. Section 4.1 describes the main differences between the Python and Fortran implementations, with emphasis on the explicit loops that replace NumPy operations. Section 4.2 presents the ring-buffer optimization that eliminates the $O(n^3)$ shift of the random field. Section 4.3 discusses the domain decomposition of CaNS and the treatment of halo cells, including schematic representations of the pencil decomposition. Section 4.4 explains the temporally accurate buffer advancement based on Taylor's hypothesis. Section 4.5 documents how the module is initialized and called from the CaNS main loop, and when the inflow velocity field is generated and prescribed as a boundary condition.

4.1 From Python to Fortran: Explicit Loops and Derived Types

The Python prototype relied heavily on the `NumPy` and `SciPy` libraries. The three-dimensional convolution was performed by `scipy.ndimage.convolve1d`, the streamwise contraction by `np.tensordot`, the interpolation by spline routines, and the random field was generated by `np.random.normal`. None of these operations has a direct counterpart in standard Fortran. The Fortran module therefore implements every operation from scratch using explicit `do` loops.

4.1.1 Convolution Without Libraries

The most visible difference is the convolution. In Python, the three filtering passes along x, y, and z were written as

```
x_filtered = np.tensordot(b_x, buffer, axes=([0], [0]))
temp = convolve1d(x_filtered, b_y, axis=0, mode='wrap')
temp = convolve1d(temp, b_z, axis=1, mode='constant', cval=0.0)
```

In Fortran, the same operation is expressed as three nested loops. The streamwise pass, which in Python is a single `tensordot` call, becomes the explicit accumulation in `apply_filter_center_plane`:

```
acc = 0.0_rp
do ix = 1, seg1_len
    ix_phys = ix_head + ix - 1
    acc = acc + bx(ix) * buf(ix_phys, iy, iz)
end do
do ix = seg1_len + 1, Lx_buf
    ix_phys = ix - seg1_len
    acc = acc + bx(ix) * buf(ix_phys, iy, iz)
end do
x_filtered(iy,iz) = acc
```

The two-loop split around `seg1_len` is a direct consequence of the ring buffer: the coefficient vector `bx` is stored in logical order, but the buffer is stored in physical order with the head at position `ix_head`. The split realigns the two so that the dot product is mathematically identical to the one computed by `np.tensordot`, without the need to physically rotate the buffer.

The spanwise pass uses periodic wrapping, implemented by an explicit modular index:

```
iy_acc = mod(iy_acc - 1 + Ly_buf, Ly_buf) + 1
acc = acc + by(m) * x_filtered(iy_acc, iz)
```

The wall-normal pass uses clamping instead of periodicity, because the filter must not correlate across the wall:

```
iz_acc = (iz + Nz_l) + (m - Nz_l - 1)
if (iz_acc >= 1 .and. iz_acc <= Lz_buf) then
    plane(iy,iz) = plane(iy,iz) + bz(m) * temp_y(iy, iz_acc)
end if
```

This asymmetry between the spanwise and wall-normal boundary treatments was already present in the Python prototype, where it was expressed by the `mode='wrap'` and `mode='constant'` arguments. In Fortran, the same behaviour is encoded in the index arithmetic.

The three filtering processes using the kernels b_x, b_y, and b_z are represented, for a general u_i component of velocity, in Figure 4.1

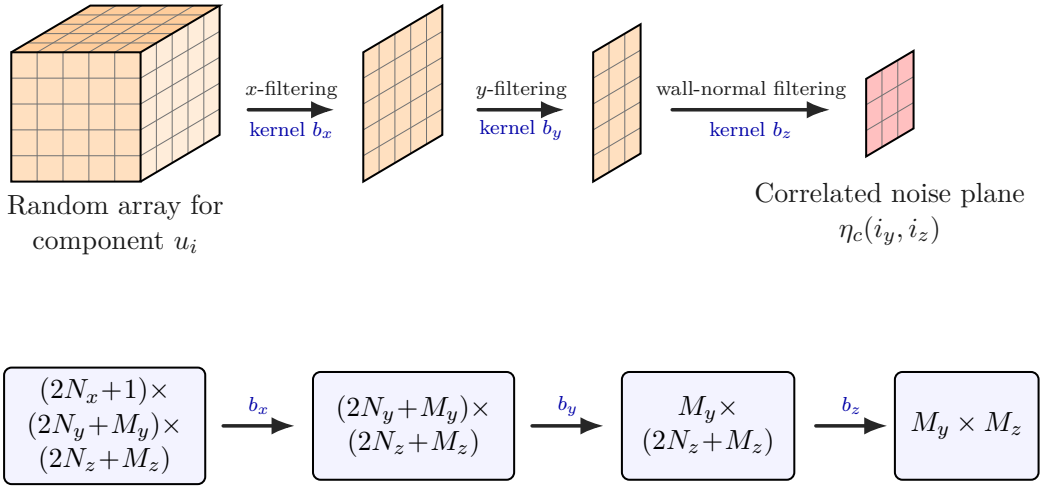


Figure 4.1: Schematic of the Klein digital-filter inflow generation: the padded random field is successively filtered in the x, y and z directions to produce the correlated noise plane $\eta_c(i_y, i_z)$.

4.1.2 Interpolation Without `SciPy`

The Python prototype used `scipy.interpolate.CubicSpline` to map the filtered fluctuations from the auxiliary uniform grid back to the stretched physical grid. The Fortran module implements a natural cubic spline from scratch, in two routines: `spline_construct`, which builds the tridiagonal system and performs the forward elimination and back substitution, and `spline_eval`, which evaluates the spline at a query point using binary search to locate the bracketing interval. The result is mathematically equivalent to `scipy.interpolate.CubicSpline` called with the `bc_type='natural'` option, but it is self-contained and requires no external library.

The explicit implementation of these two subroutines introduces a further practical advantage related to the wall-normal discretization. In the Python prototype, the reference dataset providing the mean velocity component profiles and the Reynolds stress tensor as functions of the wall-normal coordinate had to be brought onto the target simulation grid by a pre-processing step performed outside the generator. Any change in the reference dataset therefore required manual intervention, and the generator could not be pointed at a different database without rewriting the interpolation logic. The Fortran implementation removes this constraint, because the spline construction and evaluation are decoupled: the spline is built once on the source grid of the input file, and it is then evaluated on the target grid of the simulation at any desired set of wall-normal positions. As a preview of the DNS setup

discussed in Chapter 5, the reference file `MKM_Re180.dat` used in this work is discretized with a hyperbolic tangent function with 129 points in the wall-normal direction, whereas the DNS uses a stretched grid type from Pirozzoli and Orlandi [31] with 100 points. The two grids are neither aligned nor equal in size, and the spline routines perform the mapping between them automatically, without any pre-processing of the input data. The same mechanism applies to any other reference dataset, at any other Reynolds number, with no change to the module

For the mean velocity and Reynolds stress profiles, the Python prototype used linear interpolation, which the Fortran module preserves through the explicit weighted sum in `interpolate_to_grid`:

```
tg%u_temp_interp(iy,iz) = (1.0_rp - wt) * tg%vel_u(iy,kf) &
                        + wt * tg%vel_u(iy,kf+1)
```

4.1.3 Storage Layout of the `turbulentInlet_type`

All variables used for the inflow generator are stored in a single instance of the derived type `turbulentInlet_type`, declared in the main program as

```
type(turbulentInlet_type) :: tg
```

and passed by reference to every subroutine of `mod_tig`. The members of `tg` are grouped into five functional categories, listed in Table 4.1.

The first two groups are read or derived once during `init_tg` and are never modified again: the reference profiles, the integral length scales, and the filter coefficients are constants of the simulation. The precomputed matrices form the bridge between these inputs and the runtime state: the Lund matrix `a_lund` is the Cholesky factor of `R` at every wall-normal position, and `r_buffer` is the random block that the ring buffer cycles through. The buffer-state group is the only part of `tg` that changes at every time step, and it contains just four scalars: the head index, the two advection parameters, and the sub-cell accumulator described in Section 4.4. Keeping these three groups separate means that the generator can be re-initialized with a different reference dataset without touching the algorithmic core.

The working arrays are the largest part of `tg`. They are divided into three tiers that follow the three stages of the algorithm. The `noise_*` arrays hold the filtered random planes after the three convolution passes but before the Lund transformation. The `vel_*` arrays hold the correlated fluctuations after the Lund transformation, and the `z_uniform` member stores the auxiliary uniform grid on which the filtering is performed (Section 3.3.8). The `*_out`

Table 4.1: Members of the derived type `turbulentInlet_type`. Array shapes are given in Fortran order.

Category	Member	Shape / Type
Input data	`u_avg`	`(3, Mz)`
	`R`	`(3, 3, Mz)`
	`z_target`	`(Mz)`
	`L_s`	`(3)`
Filter parameters	`n_x, n_y, n_z`	`integer`
	`Nx, Ny, Nz`	`integer`
	`My, Mz`	`integer`
	`Lx_buf, Ly_buf, Lz_buf`	`integer`
	`b_x, b_y, b_z`	`(2N+1)`
Precomputed matrices	`a_lund`	`(3, 3, Mz)`
	`z_uniform`	`(Mz)`
	`r_buffer`	`(Lx, Ly, Lz, 3)`
Buffer state	`ix_head`	`integer`
	`U_bulk`	`real(rp)`
	`dx`	`real(rp)`
	`x_shift_accum`	`real(rp)`
Working arrays	`noise_u,v,w`	`(My, Mz)`
	`vel_u,v,w`	`(My, Mz)`
	`u,v,w_out`	`(My, Mz)`
	`u,v,w_local`	`(0:n(2)+1, 0:n(3)+1)`
	`u,v,w_temp_interp`	`(My, Mz)`

arrays add the mean profile to obtain the full inlet velocity, and the `*_local` arrays are the per-rank sub-blocks, including halo cells, that are copied into the CaNS velocity fields by the `set_tg_*` routines. The `*_temp_interp` arrays are pure scratch space, used only inside `interpolate_to_grid`. They are kept as members of `tg` rather than as local variables of that subroutine only because Fortran cannot allocate arrays of this size on the stack.

4.1.4 Random Number Generation

The Python prototype used `np.random.default_rng(seed)` with a user-supplied seed for reproducibility. In Fortran, the intrinsic `random_number` subroutine is used, and the seed is set once in `init_tg`:

```
call random_seed(size=seed_size)
allocate(seed_array(seed_size))
seed_array = 42
```

```
call random_seed(put=seed_array)
```

The normal distribution is generated by the Marsaglia polar method, implemented in `gen_normal`. The method produces exact samples from $\mathcal{N}(0,1)$: each accepted polar pair (v_1, v_2) yields two independent standard normal variates, and the acceptance rate of $\pi/4 \approx 0.785$ means that roughly 1.27 `random_number` calls are required per output value on average. The implementation reuses the second sample of each pair through the `have_saved` flag, which halves the number of `random_number` calls relative to a naive implementation that discards the second value:

```
if (have_saved) then
    val = val_saved
    have_saved = .false.
    return
end if
```

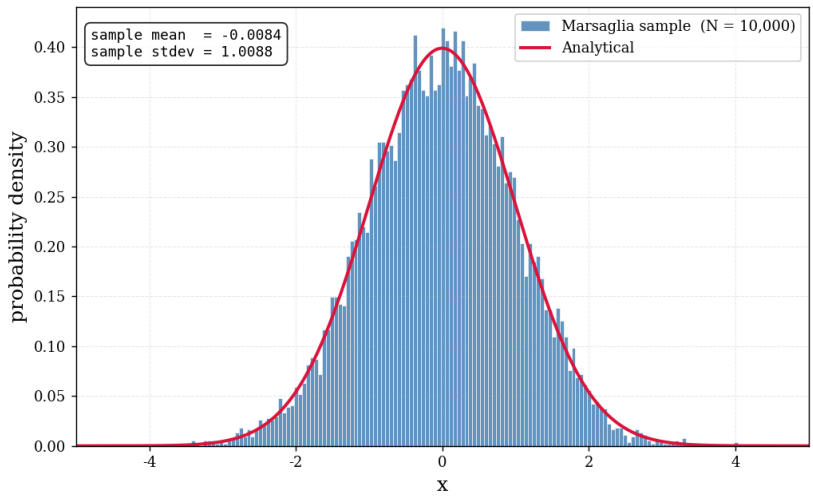


Figure 4.2: Empirical histogram of $N = 10^4$ samples drawn from `gen_normal`, compared with the analytic standard normal density $\mathcal{N}(0,1)$.

Figure 4.2 shows the empirical histogram of $N = 10^4$ samples drawn from `gen_normal`, overlapped to the analytic standard normal density. The values of mean and standard deviation agree with the theoretical values 0 and 1, and the shape of the histogram follows the analytic curve across the full range of the distribution. The residual scatter is a finite-sample effect of the histogram itself, not a limitation of the method; as N increases,

the empirical density converges to $\mathcal{N}(0,1)$ at the standard rate. In case of $N = 10^5$ samples, the mean value is $+0.0047$ and the standard deviation is 1.0032, for $N = 10^6$ the mean gets to $+0.0009$ and the standard deviation to 1.0003, which demonstrates the convergence towards values of 0 and 1.
This optimization is particularly important because `gen_normal` is called $3 \times L_x^{\text{buf}} \times L_y^{\text{buf}} \times L_z^{\text{buf}}$ times at every buffer advance, and the random-number generation is one of the dominant costs of the module.

4.2 Ring Buffer with Moving Head

The Python prototype advanced the random buffer by physically shifting the entire array at every time step:

```
buffer[:-1] = buffer[1:]
buffer[-1] = np.random.normal(size=(Ly_buf, Lz_buf))
```

This operation copies $(2N_x) \times L_y^{\text{buf}} \times L_z^{\text{buf}}$ floating-point values per step, i.e. it scales as $\mathcal{O}(N_x L_y L_z)$, which is the same order as the convolution itself and negates the benefit of the low-storage formulation when N_x is large. The Fortran module eliminates this cost entirely through the ring buffer.

4.2.1 Moving Head Index

Instead of shifting the whole array, the module maintains an integer index `ix_head` that indicates the position of the oldest plane in the buffer. When a new plane is appended, it is written at the position `ix_head`, and the index is incremented through the *mod*:

```
ix_old_head = tg%ix_head
do iz = 1, Lz_buf
    do iy = 1, Ly_buf
        call gen_normal(rnd_val)
        tg%r_buffer(ix_old_head, iy, iz, i_comp) = rnd_val
    end do
end do
tg%ix_head = mod(ix_old_head, Lx_buf) + 1
```

In Fortran, array indices start at 1, not at 0. The presence of $+1$ is related to Fortran indices since *mod* function returns the value 0 when `ix_old_head` equals `Lx_buf`. The same buffer memory is reused indefinitely, and the cost of appending a plane is exactly $L_y^{\text{buf}} \times L_z^{\text{buf}}$ operations, independent of N_x.

Over a full simulation the ring buffer therefore reduces the total cost of buffer management from $\mathcal{O}(N_t N_x L_y L_z)$ to $\mathcal{O}(N_t L_y L_z)$, which for $N_x \approx 100$ is a two-order-of-magnitude saving on this part of the algorithm.

4.2.2 Filtering with the Moving Head

The moving head has a direct consequence for the convolution: the coefficient vector `b_x` is stored in logical order, but the buffer is stored in physical order with the head at `ix_head`. The filter must therefore access the buffer in two segments, the first from `ix_head` to the end of the array and the second from the beginning to `ix_head`-1. This is precisely what the two-loop split in `apply_filter_center_plane` accomplishes:

```
seg1_len = Lx_buf - ix_head + 1
do ix = 1, seg1_len
    ix_phys = ix_head + ix - 1
    acc = acc + bx(ix) * buf(ix_phys, iy, iz)
end do
do ix = seg1_len + 1, Lx_buf
    ix_phys = ix - seg1_len
    acc = acc + bx(ix) * buf(ix_phys, iy, iz)
end do
```

The two segments together span the entire coefficient vector, and the result is identical to the dot product that would be computed on a physically shifted buffer. The ring buffer therefore preserves the mathematical formulation exactly while eliminating the shifting cost.

4.2.3 Memory Layout

The random buffer has shape $(L_x^{\text{buf}}, L_y^{\text{buf}}, L_z^{\text{buf}}, 3)$ with the three spatial indices first and the velocity-component index last. Fortran stores arrays in column-major order, making the last index varying faster in memory. This layout is chosen deliberately: elements with consecutive i_x values lie at consecutive addresses, whereas elements with consecutive i_{comp} values are separated by a stride of $L_x^{\text{buf}} L_y^{\text{buf}} L_z^{\text{buf}}$. Therefore, the buffer is stored as three large adjacent blocks, one per velocity component, rather than a single array of three-component vectors.

This layout matches the access pattern of the filtering step, which dominates the cost of the generator. In `apply_filter_center_plane`, the streamwise convolution uses the innermost loop over i_x:

```
do ix = 1, seg1_len
    ix_phys = ix_head + ix - 1
    acc = acc + bx(ix) * buf(ix_phys, iy, iz)
end do
```

Since i_x is the contiguous index, this loop reads memory in sequential order and benefits from cache locality and vectorization. The outer loops over i_y and i_z are distributed across OpenMP threads or GPU work-items, so each thread reads a distinct column of the buffer without contention. The velocity component index does not appear in the filtering loop at all: the subroutine is called three times, once per component, and each call works on a single contiguous $(L_x^{\text{buf}}, L_y^{\text{buf}}, L_z^{\text{buf}})$ block. If i_{comp} came first, the three velocity components would be interleaved in memory and the filter would lose its sequential access pattern.

4.3 Parallelization and Halo Cells

4.3.1 Why Parallelization Is Mandatory

A DNS simulation typically has millions of cells (order of 10^6 and more). This relevant number of cells, coupled with a number of time steps, in order of 10^4, can easily lead to an order of 10^{10} operations, which is massive for a single CPU core. If only one core were used, the actual time used to solve the simulation could require days, even for an easy problem. In order to speed up the whole process, the calculations can be spread on multiple cores or even CPUs/GPUs. This process can be done on distributed-memory clusters, where the domain is partitioned across ten, hundreds or thousands of MPI ranks and each rank owns a small sub-block of the grid. One CPU can have multiple cores, i.e. in case of 64 cores, there are going to be 64 available ranks.

Rank is a logical way of numbering the available units that can compute calculations. It represents on how many cores is possible to run the job or part of it. If the turbulent generator is run on only rank 0 (index for ranks spaces from 0 to $N - 1$ where N are the available cores) and the generated inlet plane broadcasted to all other ranks, the broadcast would become a serial bottleneck at every time step: the cost of sending $M_y \times M_z$ floating-point values over the network is comparable to the cost of computing them, and the latency of the collective operation would dominate the simulation time. The generator must therefore be parallelized in such a way that every rank computes its own portion of the inlet plane locally, with no inter-process communication in the inner loop.

4.3.2 Domain Decomposition in CaNS

CaNS uses the 2DECOMP&FFT library for domain decomposition and distributed FFTs. The global grid has $N_x \times N_y \times N_z$ points, and the domain is divided into $P =$ `dims(1)` $\times$ `dims(2)` sub-blocks, where P is the number of MPI ranks. The array `dims(1:2)` is read from the input `input.nml` file:

```
dims(1:2) = 0, 0, ipencil_axis = 1
```

A value of zero in `dims(1:2)` instructs 2DECOMP to choose an optimal decomposition automatically based on the global grid and the number of ranks. The parameter `ipencil_axis = 1` selects the axis along which the first pencil decomposition is performed, which determines how the data is distributed across ranks for the FFTs. In CaNS [6], when `ipencil_axis = 1` the direction of the decomposition of the domain is in x direction.

4.3.3 Slab and Pencil Decomposition

The 2DECOMP library supports two decomposition strategies: a 1D slab decomposition and a 2D pencil decomposition. In a slab decomposition, the domain is divided along only one spatial direction. For example, if decomposed along the x-axis, each MPI rank owns full, unbroken y-z planes and only a subset of the x layers. In a pencil decomposition, the domain is divided along two directions simultaneously, assigning each rank a long, thin pencil of grid points. This distributes the computational load and memory more evenly across ranks for large P.

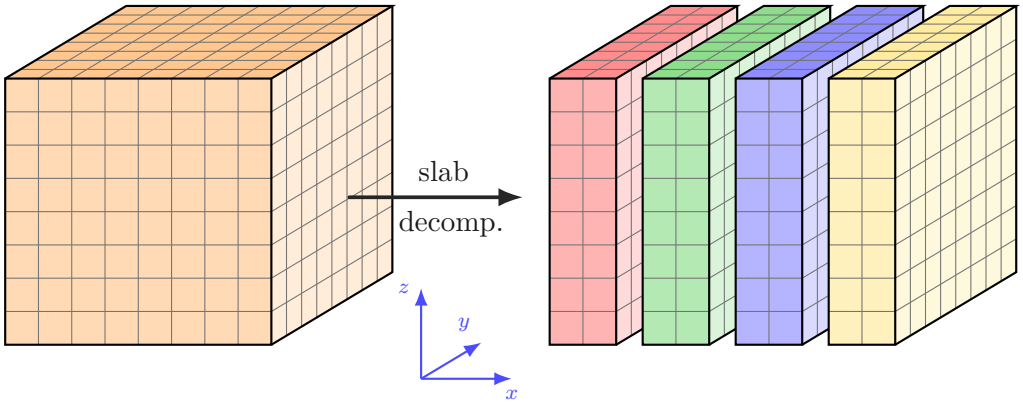


Figure 4.3: Slab decomposition of the $16 \times 16 \times 16$ inlet block into four x-aligned slabs.

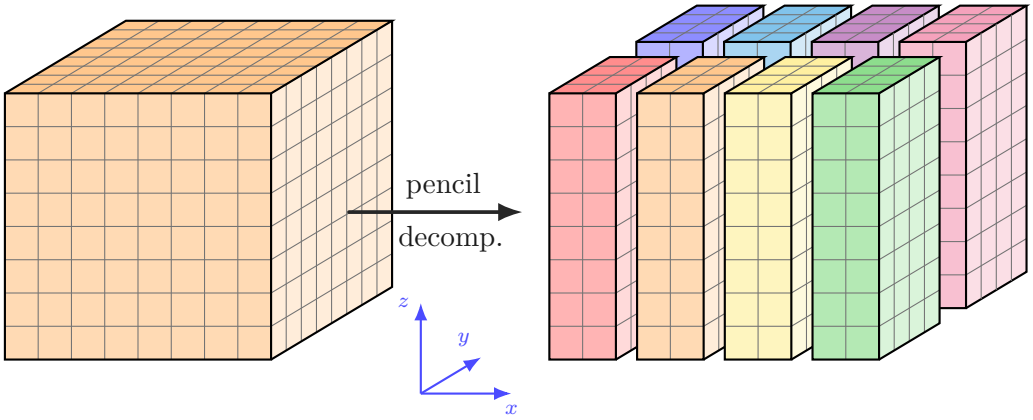


Figure 4.4: Pencil decomposition of the $16 \times 16 \times 16$ inlet block into eight z-aligned towers.

Figure 4.3 shows a schematic of the slab decomposition for a three-dimensional example. The global domain is divided into P_x sub-blocks, and each sub-block is assigned to one MPI rank. Since data on y and z axis remain untouched, each rank performs N_x/P_x distinct 2D-FFTs. In this case $P_x = 4$ and $N_x = N_y = N_z = 16$.

Figure 4.4 shows a schematic of the pencil decomposition for a three-dimensional example with decomposition in x and y direction. This type of decomposition brings to the so-called *z-aligned pencil*. The global domain is divided into $P_x \times P_y$ sub-blocks, and each sub-block is assigned to one MPI rank. The local grid size on each rank is roughly $N_x/P_x \times N_y/P_y \times N_z/P_z$, with the exact split determined by the integer division performed by 2DECOMP. Since only the data on z axis remains untouched, each rank computes a number of $(N_x/P_x) \times (N_y/P_y)$ distinct 1D-FFTs. In this case $P_x = P_y = 4$ and $N_x = N_y = N_z = 16$, giving a total of 16 MPI ranks.

A pencil decomposition offers exceptional scalability. While a slab decomposition restricts the maximum processor count to N (the number of grid points along the split axis), a pencil decomposition allows the use of up to N^2 processors. This makes pencil decomposition the mandatory strategy for large-scale, high-performance simulations utilizing massive grids (e.g., 2048^3 points and larger) that require optimal load balancing across high core counts.

For the inlet generator, the relevant decomposition is the one on the inlet plane, i.e. the y-z plane. Each rank receives a local sub-block of size $n(2) \times n(3)$, where $n(2)$ and $n(3)$ are the local grid dimensions read from the `mod_common_mpi` module. The generator must produce the correct velocity values on this sub-block, including the halo cells that are needed by the

finite-difference stencils of the solver.

4.3.4 Halo Cells

In a domain-decomposed computation, each MPI rank owns a sub-block of the global grid. The finite-difference and interpolation stencils at the edges of a sub-block, however, need values that live on the neighbouring rank. To make those values available locally, a layer of *halo cells* is stored around the interior of each sub-block, holding copies of the neighbour's data.

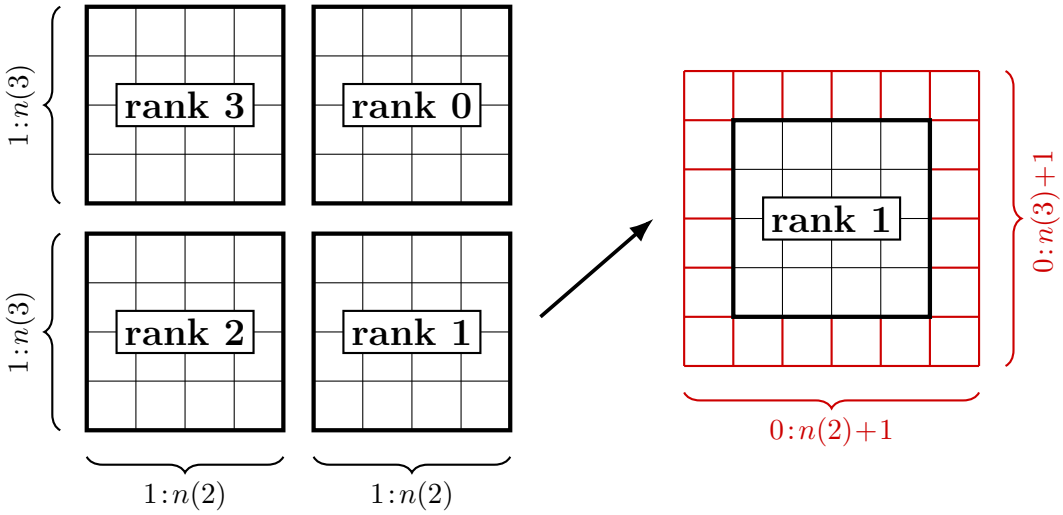


Figure 4.5: Left: partition of the computational domain into four MPI ranks. Right: the subdomain of rank 1 surrounded by one layer of halo cells.

The stencils used by CaNS require one such layer on each side of the local sub-block. Figure 4.5 shows the arrangement for a two-dimensional example: the interior cells owned by one rank are surrounded by a ring of halo cells. For the inlet plane, this means that each rank must fill not only its own $n(2) \times n(3)$ cells but also the surrounding cells at indices 0 and $n(2)+1$ in the y direction and 0 and $n(3)+1$ in the z direction. Without these halo values, the finite-difference operators near the sub-block boundaries would use incorrect data and introduce spurious oscillations.

The finite-difference and interpolation stencils used by CaNS require one layer of halo cells on each side of the local sub-block. For the inlet plane, in cases of an *x-aligned pencil*, this means that each rank must fill not only its own $n(2) \times n(3)$ cells but also the surrounding cells at indices 0 and $n(2)+1$ in the y direction and 0 and $n(3)+1$ in the z direction. Without these halo

values, the finite-difference operators near the sub-block boundaries would use incorrect data and introduce spurious oscillations.

In a conventional parallel implementation, the halo cells would be filled by exchanging messages with the neighbouring ranks through MPI. This is the standard approach for the velocity and pressure fields in CaNS, where the `bounduvw` and `boundp` routines perform the halo exchange at every time step. For the inlet generator, however, the halo exchange is not necessary. Since the generator computes the full global inlet plane on every rank, each rank already contains the value that its neighbours would send. The local sub-block, including the halo cells, can be directly extracted from the global plane by a simple index shift:

```
do k = 0, n(3)+1
    kg = lo(3)-1+k
    if (kg > 0 .and. kg <= tg%Mz) then
        do j = 0, n(2)+1
            jg = lo(2)-1+j
            if (jg > 0 .and. jg <= tg%My) then
                tg%u_local(j,k) = tg%u_out(jg,kg)
                tg%v_local(j,k) = tg%v_out(jg,kg)
                tg%w_local(j,k) = tg%w_out(jg,kg)
            end if
        end do
    end if
end do
```

The variables `lo(2)` and `lo(3)` are the global indices of the first interior point owned by the rank in the y and z directions, respectively. The expression `jg = lo(2)-1+j` converts the local index `j` into the corresponding global index, so that `u_local(j,k)` receives the value `u_out(jg,kg)` at the same physical location. The bounds check skips halo cells that fall outside the physical domain, for example below the wall at $z = 0$ or above the channel centreline, where no neighbour exists and no value is available. The result is a local array `u_local` of shape $(0 : n(2) + 1, 0 : n(3) + 1)$ that contains exactly the data required by the boundary-condition routines.

This approach could be more efficient from a computational point of view: currently every rank performs the full global filtering process. The overall cost per rank is therefore $\mathcal{O}(L_x^{\mathrm{buf}} \times M_y \times M_z)$ which is a factor of P more work than the strictly necessary which would be $\mathcal{O}(L_x^{\mathrm{buf}} \times n(2) \times n(3))$. However, this extra cost is small compared with the Poisson solver and the momentum equations, and in return the generator requires no inter-process communication at all.

4.3.5 OpenMP and OpenACC Parallelization

Within each MPI rank, the generator is parallelized over threads (OpenMP) and over GPU work-items (OpenACC). The filtering loops in `apply_filter_center_plane` are decorated with `!$OMP PARALLEL DO` and `!$ACC PARALLEL LOOP` instructions, with `collapse(2)` to distribute both the y and z loops across threads. The Lund transformation and the interpolation are parallelized in the same way. The random-number generation is kept serial deliberately, because the Marsaglia polar method maintains a saved second sample between calls and is therefore not safe to parallelize without a per-thread state.

The GPU offloading is managed through the `!$acc enter data` and `!$acc update device` instructions. The local arrays `u_local`, `v_local`, and `w_local` are registered on the device once in `init_tg`:

```
!$acc enter data create(tg%u_local, tg%v_local, tg%w_local)
```

At every call of `generate_inflow`, the freshly computed CPU values are pushed to the device with `!$acc update device`, and the boundary condition routines `set_tg_u`, `set_tg_v`, and `set_tg_w` write the values into the global velocity arrays on the device. This avoids repeated host-device transfers within the time-stepping loop and keeps the generator overhead negligible.

4.4 Temporal Advancement and Taylor's Hypothesis

4.4.1 The Frozen-Turbulence Approximation

The random buffer represents a three-dimensional block of turbulence that is advected through the inlet plane by the mean flow. In a spatially developing simulation, the inflow plane is not a solid boundary; it is a virtual surface through which turbulence enters the domain. The velocity field at this surface evolves in time as the turbulent structures are carried past it by the streamwise mean velocity.

Taylor's hypothesis of frozen turbulence states that, for a flow with a sufficiently high mean velocity, the temporal evolution of the velocity at a fixed point is dominated by the advection of frozen turbulent structures, so that

$$\frac{\partial u}{\partial t} \approx -U_{\text{bulk}} \frac{\partial u}{\partial x}, \tag{4.1}$$

where U_{bulk} is the bulk streamwise velocity. In the digital-filter context, this means that the inflow plane at time $t + \Delta t$ is approximately the inflow plane at time t shifted downstream by a distance $U_{\text{bulk}}\Delta t$. Equivalently, the random buffer must be advanced by

$$n_{\text{shift}} = \frac{U_{\text{bulk}}\Delta t}{\Delta x} \tag{4.2}$$

planes per time step, where Δx is the streamwise grid spacing. When n_{shift} is an integer, the buffer is simply advanced by that many planes, and the filter produces a new inflow plane that is statistically consistent with the previous one.

4.4.2 Sub-Cell Shift Accumulation

In CaNS, the time step Δt is adaptive and is recomputed at every step from the CFL condition. The ratio $U_{\text{bulk}}\Delta t/\Delta x$ is therefore not an integer in general, and rounding it to the nearest integer at every step would introduce a systematic drift in the effective advection velocity. If the ratio is consistently rounded down, the turbulence would appear to move slower than the mean flow; if it is consistently rounded up, it would move faster. Over thousands of time steps, the accumulated error would degrade the temporal correlation of the generated field.

The Fortran module solves this problem with an accumulator. The physical shift in cell units is accumulated in `x_shift_accum`, and the integer part is extracted at every time step:

```
tg%x_shift_accum = tg%x_shift_accum + tg%U_bulk * dt / tg%dx
n_shift = int(tg%x_shift_accum)
tg%x_shift_accum = tg%x_shift_accum - real(n_shift, rp)
```

The fractional remainder is retained for the next step, so that the long-run average shift per time step is exactly $U_{\text{bulk}}\Delta t/\Delta x$, with no drift. This is the discrete equivalent of a phase accumulator in a numerically controlled oscillator, and it guarantees that the temporal correlation of the generated field is preserved over arbitrarily long simulations.

4.4.3 Choosing the Bulk Velocity

The bulk velocity U_{bulk} is computed in `init_tg` by trapezoidal integration of the mean streamwise profile over the channel height:

```
tg%U_bulk = 0.0_rp
do k = 1, tg%Mz - 1
    tg%U_bulk = tg%U_bulk + 0.5_rp * (tg%u_avg(1,k) +
                tg%u_avg(1,k+1)) * (tg%z_target(k+1)
                - tg%z_target(k))
end do
tg%U_bulk = tg%U_bulk / l(3)
```

The bulk velocity is the appropriate advection velocity for the frozen turbulence approximation in a channel flow, because the turbulence at the inlet plane is not localized near the wall but spans the entire channel height. Using the centreline velocity would overestimate the advection rate in the near-wall region, while using the friction velocity would underestimate it in the outer region. The bulk velocity is the natural compromise.

4.4.4 Retiring and Replenishing Planes

When the integer shift is n_{shift}, the module retires the oldest n_{shift} planes of the buffer and replaces them with fresh random data. Each retired plane is overwritten in place at the position `ix_head`, and the head is incremented modulo L_x^{buf}:

```
do i_shift = 1, n_shift
    ix_old_head = tg%ix_head
    do i_comp = 1, 3
        do iz = 1, Lz_buf
            do iy = 1, Ly_buf
                call gen_normal(rnd_val)
                tg%r_buffer(ix_old_head,iy,iz,i_comp) = rnd_val
            end do
        end do
    end do
    tg%ix_head = mod(ix_old_head, Lx_buf) + 1
end do
```

The loop over `i_shift` is kept serial because `gen_normal` maintains a saved sample between calls. The cost of replenishing n_{shift} planes is $n_{\text{shift}} \times 3 \times L_y^{\text{buf}} \times L_z^{\text{buf}}$ random-number generations, which is proportional to the physical time elapsed and is therefore negligible compared to the cost of the solver.

Two safety checks bound the value of n_{shift}. The lower bound $n_{\text{shift}} \geq 0$ handles the case of a very small time step, where the physical shift is less

than one cell; in this case the buffer is not advanced, and the filter produces a nearly identical inflow plane to the previous step. This is physically correct: for $\Delta t \ll \Delta x/U_{\text{bulk}}$, the inflow plane barely changes between consecutive steps, and the generated field should reflect that. The upper bound $n_{\text{shift}} \leq L_x^{\text{buf}} - 1$ prevents the buffer from being flushed entirely, which would destroy all correlation. When the bound is exceeded, the excess is returned to the accumulator so that no shift is silently discarded.

4.5 Integration with the CaNS Solver

The turbulent inlet generator is integrated into CaNS through three interfaces: the initialization routine defined as `init_tg`, the generator routine `generate_inflow`, and the boundary-condition routines `set_tg_u`, `set_tg_v`, and `set_tg_w`. This section documents when and how these routines are called from the main program.

4.5.1 Initialization

The generator is initialized in the main program immediately after the flow field has been set up and before the main time-stepping loop begins:

```
if(is_t_inlet) call init_tg(tg,ng,l,n,zc_g(1:ng(3)))
```

The `is_t_inlet` flag is read from the input file and activates the turbulent inlet. The arguments are the derived type `tg`, the global grid sizes `ng`, the domain lengths `l`, the local grid sizes `n`, and the global wall-normal coordinates `zc_g`. The initialization performs the following steps:

1. Sets the global inlet plane dimensions `My = ng(2)` and `Mz = ng(3)`.
2. Reads the mean velocity profile and Reynolds stress tensor from the reference file `*.dat` if it exists, or falls back to a default hardcoded analytical profile.
3. Computes the normalized length scales n_x, n_y, n_z from the prescribed integral length scales and the grid spacing.
4. Computes the filter half-widths $N_x = \max(2n_x, 4)$, and similarly for N_y and N_z.
5. Builds the Lund matrix `a_lund` at every wall-normal position by Cholesky decomposition.

6. Computes the 1-D filter coefficients `b_x`, `b_y`, `b_z`.

7. Allocates and fills the initial random buffer with unit-variance Gaussian noise.

8. Registers the local arrays on the GPU device with `!$acc enter data create`.

After initialization, the generator is ready to produce inflow planes.

4.5.2 The Time-Stepping Loop

The main time-stepping loop of CaNS increments the time-step counter and the physical time at the top of the loop:

```
do while(.not.is_done)
    istep = istep + 1
    time = time + dt
    ...
```

Immediately after these increments, and before the Runge-Kutta stages begin, the generator is called:

```
if(is_inlet .and. is_t_inlet .and. is_bound(0,1)) then
    call generate_inflow(tg, n, lo, dt)
    call set_tg_u(tg,ng,n,lo,is_bound,0,u)
    call set_tg_v(tg,ng,n,lo,is_bound,0,v)
    call set_tg_w(tg,ng,n,lo,is_bound,0,w)
end if
```

The logical condition `is_bound(0,1)` checks that the inlet boundary is located at the lower x boundary of the domain, which is the standard configuration for a spatially developing flow. The call to `generate_inflow` produces a new inflow plane based on the current time step `dt`, advances the random buffer according to Taylor's hypothesis, and fills the local arrays `u_local`, `v_local`, `w_local`. The three calls to `set_tg_u`, `set_tg_v`, and `set_tg_w` then copy these local arrays into the corresponding slices of the global velocity arrays.

4.5.3 Application During the Runge-Kutta Stages

The velocity boundary condition at the inlet must be consistent across all three stages of the Runge-Kutta scheme and across the pressure-correction step. The boundary-condition routines are therefore called multiple times within each time step. Inside the Runge-Kutta loop:

```
do irk=1,3
    if(is_inlet .and. is_t_inlet .and. is_bound(0,1)) then
        call set_tg_u(tg,ng,n,lo,is_bound,0,u)
        call set_tg_v(tg,ng,n,lo,is_bound,0,v)
        call set_tg_w(tg,ng,n,lo,is_bound,0,w)
    end if
    ...
```

Note that `generate_inflow` is not called inside the Runge-Kutta loop; only the boundary-condition routines are called. The inflow plane is generated once per time step, and the same plane is applied at every stage. This is the correct behaviour: the inflow plane is a physical boundary condition that varies on the time scale of the large eddies, not on the time scale of the Runge-Kutta sub-steps. Regenerating the plane at every stage would introduce spurious high-frequency content and would break the temporal correlation of the generated field.

After the pressure-correction step, the boundary conditions are applied once more:

```
if(is_inlet .and. is_t_inlet .and. is_bound(0,1)) then
    call set_tg_v(tg,ng,n,lo,is_bound,0,v)
    call set_tg_w(tg,ng,n,lo,is_bound,0,w)
end if
```

Only v and w components are re-imposed at the boundary, not u velocity component. The reason lies in how the velocities and pressures are stored in CaNS's domain. CaNS uses a *staggered grid* arrangement where u is saved on $x-direction$ faces, v is saved on $y-direction$ faces, w is saved on $z-direction$ faces and pressure p is saved at the center of the grid. The v and w components are therefore cell-centred in the streamwise x direction, at the same x-position as the pressure. For the inlet plane at $x = 0$, this means that u's inlet value is written at the face index `i = ibound * n(1)`, while v and w are written at the cell-centre index `i = ibound * (n(1)+1)`.

The reason `set_tg_u` is not re-applied is subtler than it appears. The corrector routine does include the inlet face $i = 0$ in its loop for u, but the

update is $u(0,j,k) \leftarrow u(0,j,k) - \Delta t/\Delta x\,(pp(1,j,k) - pp(0,j,k))$, and the discrete pressure correction satisfies $pp(0,j,k) = pp(1,j,k)$ at the inlet. The correction term therefore vanishes identically and the face value $u(0,j,k)$ is unchanged by the projection, so re-imposing it would only rewrite a value that is already correct. The components v and w, by contrast, sit on cell centres x-position, where the pressure correction has non-zero spanwise and wall-normal directions and therefore modifies their values. They must be re-imposed after the corrector step to restore the turbulent fluctuations produced by the digital filter at the inlet plane.

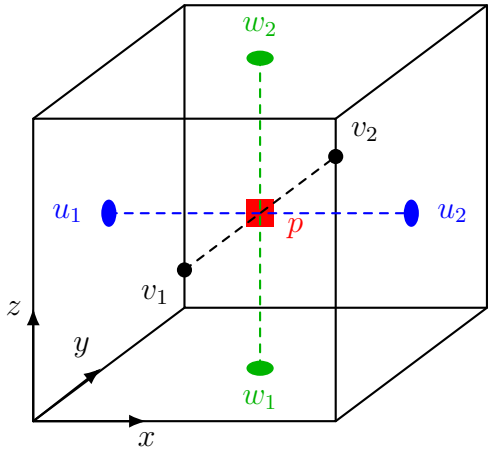


Figure 4.6: Staggered grid arrangement of a single computational cell. Pressure p stored at the cell centre; velocity components u, v and w stored on the faces normal to the x, y and z directions, respectively.

The consequence of this asymmetry is that the inlet velocity field is not exactly divergence-free after the pressure-correction step: overwriting v and w with the generated values reintroduces a small residual divergence at the inlet and slightly perturbs the correlation between the three components with respect to the one produced by the digital filter. This is an inherent limitation of prescribing synthetic turbulence in a fractional-step solver with a staggered grid.

4.5.4 The Boundary-Condition Routines

The routines `set_tg_u`, `set_tg_v`, and `set_tg_w` copy the local inlet arrays into the global velocity arrays at the inlet plane. They differ only in the index of the inlet plane and in the halo treatment. All three routines exclude the halo cells at the global boundaries, which are owned by neighbouring subdo-

mains or lie outside the physical domain, because those cells are overwritten later by the wall or periodicity boundary conditions:

```
j0 = 0
j1 = n(2)+1
if (lo(2) == 1) j0 = 1
if (lo(2)+n(2)-1 == ng(2)) j1 = n(2)
```

The same guard is applied in the wall-normal direction. This ensures that the periodic boundary condition in the spanwise direction and the no-slip condition at the walls are not overwritten by the inlet generator. In the streamwise direction the routines are called only when $is_bound(0, 1)$ is true, i.e. when the local subdomain owns the inlet plane, so no equivalent guard is needed there.

4.6 Summary and Discussion

The Fortran module `mod_tig` implements the Klein digital-filter algorithm with the Kempf low-storage formulation, ported from the Python prototype of Chapter 3 and integrated into the CaNS solver. The main modifications relative to the Python prototype are:

- **Explicit loops.** All NumPy and SciPy operations, including the three convolution passes, the cubic-spline interpolation, and the random-number generation, are implemented as explicit Fortran `do` loops.
- **Ring buffer.** The random buffer is advanced by a moving head index rather than by a physical shift of the array, reducing the cost of buffer management from $\mathcal{O}(N_x L_y L_z)$ to $\mathcal{O}(L_y L_z)$ per time step.
- **Taylor's hypothesis.** The buffer is advanced by $n_{\text{shift}} = U_{\text{bulk}} \Delta t / \Delta x$ planes per time step, with the fractional remainder accumulated between steps. This accommodates the adaptive time step of CaNS and enforces the correct physical advection rate of the inflow plane.
- **Parallelization.** The generator is parallelized with OpenMP and OpenACC within each MPI rank, and the local sub-block is extracted from the global inlet plane with a simple index shift that populates the halo cells without inter-process communication.
- **GPU offloading.** The local arrays are registered on the GPU device once and updated at every time step, keeping the host-device transfer overhead negligible.

The design choice to compute the full global inlet plane on every MPI rank, rather than distributing the filtering operation across ranks, is a deliberate trade-off. It sacrifices a factor of P in computational cost for the complete elimination of inter-process communication. The additional cost is small compared to the cost of the momentum equations and the Poisson solver, and the simplicity of the implementation is a significant advantage for maintenance and debugging. The same choice was made by Kempf et al. [3] in their original low-storage formulation, and it has been validated in the present work by the DNS results reported in Chapter 5.

The following chapter presents the DNS of a turbulent channel flow at $Re_\tau = 180$ using the Fortran module described here, and compares the results with the reference data of Moin, Kim, and Mansour [4].

Chapter 5

Channel flow's setup and results

Before jumping to simulation of a turbulent channel flow with Digital Filter's velocity field imposed as boundary conditions, a few preliminary simulations of a doubly-periodic channel were conducted. The reasons are two:

- Verify that CaNS reproduces the MKM [4] reference statistics at at $Re_\tau = 180$.
- Establish a doubly-periodic baseline that uses the same wall-normal discretization as the main simulation, so that deviations observed in the turbulent-inlet case can be attributed to the inlet generator alone.

Once the doubly-periodic channel was validated in CaNS, then the simulation with the Digital filter was run. The second doubly-periodic simulation is not a necessary step but it provides better compatible data to compare with the Digital filter on simulation.

5.1 Preliminary Simulations

Two preliminary simulations were run where the aim is to observe how the simulations in CaNS compare to the reference data of Moser et al. [4] obtained through the spectral code of Kim et al. [5]. Since the original reference data is generated using highly accurate global basis functions, the first objective is to evaluate if CaNS - which employs a second-order central finite-difference scheme - can successfully recover these statistics using the exact same number of grid points and wall-normal discretization. As will be shown, the velocity profiles and Reynolds stresses match the spectral reference data very closely.

A minor underestimation is observed at higher y^+ and y/δ, which is an expected consequence of the differing spatial discretization schemes.

Following this direct validation, a second doubly-periodic channel simulation is introduced. This intermediate simulation has a domain decomposed in wall-normal direction following the decomposition of Pirozzoli and Orlandi [31], mirroring the exact numerical setup that will be utilized in the main spatially developing simulation. When comparing this second baseline against the spectral benchmark, the results once again demonstrate strong agreement, preserving only the previously mentioned minor outer-layer underestimations in Section 3.4.

The parameters of each simulation âĂŤ domain dimensions, grid resolution, viscosity, boundary conditions, and stopping criteria âĂŤ are all set in the input file `input.nml` read by CaNS at start-up. Because the reference simulations use $u_\tau = 1$ and $\delta = 1$ as non-dimensionalisation units, the friction Reynolds number reduces to the inverse of the kinematic viscosity, $Re_\tau = 1/\nu$, so that $\nu = 1/Re_\tau$. The value $\nu = 1/180$ is therefore imposed for all simulations at nominal $Re_\tau = 180$.

5.1.1 First Doubly-periodic simulation

The first run simulation of the doubly-periodic streamwise and spanwise turbulent channel had the same number of points and lengths of the domain as the ones in MKM's simulations [4]. Table 5.1 lists the parameters for the first doubly-periodic simulation, the reference system used is the one commonly found in literature: x for streamwise direction, y for wall-normal direction and z for spanwise direction.

Table 5.1: Simulation parameters for the first doubly-periodic channel simulation at $Re_\tau = 180$

L_x	L_y	L_z	$N_x \times N_y \times N_z$	Δx^+	Δz^+
$4\pi\delta$	2δ	$\frac{4}{3}\pi\delta$	$128 \times 129 \times 128$	17.7	5.9

The dimensions of the domain (L_x, L_y, L_z) are a function of the half-channel height δ which is equal to 1.

The whole simulation was split in two stages: the first part was responsible to reach the condition of fully-developed flow, the second part was instead needed to compute the statistics on the fully-developed quasi-steady state. Each of the two stages was run for 40000 time steps, for a total of 80000. The bulk velocity U_{bulk} for this type of channel at $Re_\tau = 180$ is about 15.67 which,

given the streamwise domain length L_x of 4π, provides the *flow-through time* t_{ft} of:

$$t_{ft} = \frac{L_x}{U_{bulk}} = \frac{4\pi}{15.67} \approx 0.80 \tag{5.1}$$

The 40000 time steps provided a total simulation time of $t = 82.4s$ and the number of flow-through times is:

$$N_{ft} = \frac{t}{t_{ft}} = \frac{82.4}{0.80} \approx 102.8 \tag{5.2}$$

Consequently, the flow completed approximately 103 flow-through times ($t/t_{ft} \approx 103$). Because standard practices in direct numerical simulations of channel flows typically require 10 to 20 flow-through times to properly wash out initial perturbations, this initialization window is highly conservative and rigorously guarantees that the flow has reached a fully developed, statistically steady state. Even by taking into account the *Large Eddy Turnover Time* τ, calculated by the ratio between the half channel height δ and the friction velocity u_τ, in theory, the simulation would still be far conservative with 40000 time steps. This can be demonstrated easily since, for this case, from Moser, Kim, and Mansour simulations, $\delta = 1$ and $u_\tau = 1$ whose ratio provides:

$$\tau = \frac{\delta}{u_\tau} = \frac{1}{1} = 1 \tag{5.3}$$

Since the total simulation time is of 82.4 seconds for the whole 40000 time steps, the ratio between t and τ would provide a number of large eddy turnovers of ≈ 82.4, still ensuring the fully developed state. The difference between the two quantities is slight but they still provide two different point of views:

- **Flow-through time** (L_x/U_{bulk}): Measures the bulk advection. It answers the question: How many times did a volume of fluid completely traverse the 4π periodic box?
- **Eddy turnover time** (δ/u_τ): Measures the characteristic lifespan of the largest, energy-containing turbulent structures (eddies roughly the size of the channel half-height). It answers: How many times did the largest eddies rotate, break down, and transfer their kinetic energy down the turbulent cascade?

The second part of the simulation, as anticipated, was run for another 40000 number of time steps. Here, the three-dimensional velocity field was saved every 400 time steps, thus providing 101 snapshots to process.

5.1.2 Results for the first Doubly-periodic simulation

The obtained results match nearly perfectly with the reference data from Moser, Kim, and Mansour for the turbulent channel at nominal $Re_\tau = 180$.

Table 5.2: Comparison of different quantities between the results from the first doubly-periodic simulation and reference ones at nominal $Re_\tau = 180$

Data set	Actual Re_τ	U_{bulk}	u_τ
Reference [4]	178.13	15.67	1.000
Computed	177.14	15.67	0.994

The Table 5.2 shows that the match between the reference set of data and the just calculated ones is really good with less than 1% in terms of error which is tolerated. The bulk velocity and the friction velocity are matching perfectly, the computed actual friction Reynolds number is the only parameter with slightly a bigger error of 0.5% which is still a good result.

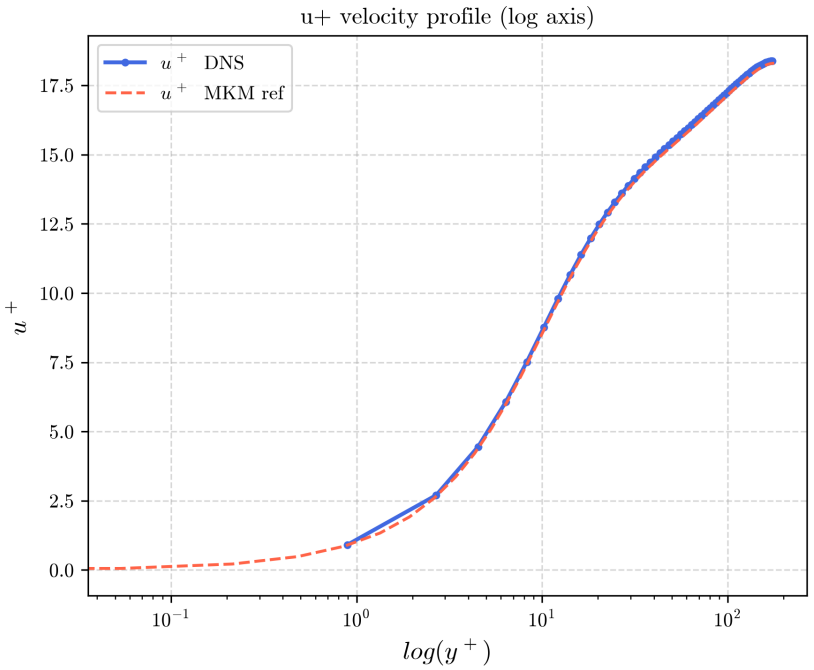


Figure 5.1: Adimensionalized streamwise velocity as a function of y^+

What about the mean velocity profiles and the Reynolds stresses? Fig-

ure 5.1 shows the adimensionalized streamwise velocity. Since the calculated friction velocity has a value of 0.994 which is less than 1% of difference to the reference one of 1, the two lines are well overlapped.

Figure 5.2 shows the behaviour of the Reynolds stresses. In general, the shape is mathing the shape of the reference data but there are some evident under and overestimations. The streamwise stresses $\langle u'u' \rangle^+$ is slightly underestimated at higher distences from the wall, for $y/\delta > 0.35$,the shear stresses $\langle u'v' \rangle^+$ are instead overestimated more near the wall.

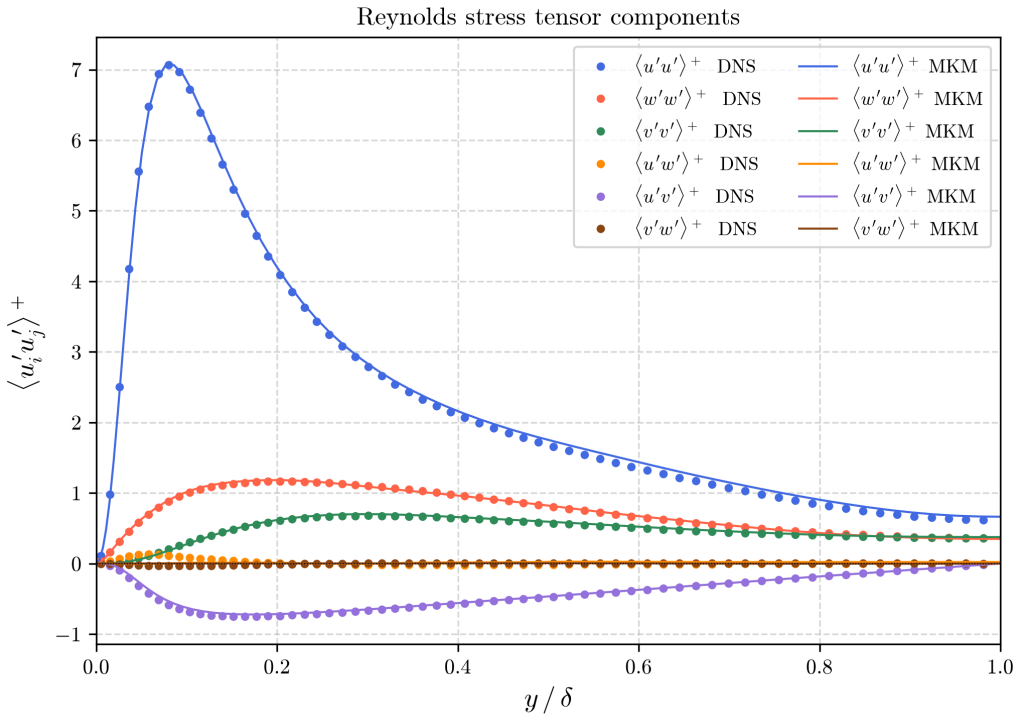


Figure 5.2: Reynolds stress components.

The same under and over estimation can be also seen in the viscous stress profile in Figure 5.3 and even more in the plot representing the total stress in Figure 5.4 where the overestimation in $\langle u'v' \rangle^+$ brings not to a fully linearly decay behaviour of the total stress in the first zones near the wall (see the red bump at about $y/\delta \approx 0.05$) but far away from the wall the recovery of the reference data is good.

The reason for these small visual inconsistencies are related to the differences in the actual DNS codes. One of these is the truncation error derived from a Taylor series expansion in the finite-difference scheme of CaNS and the second one derives from the modified wavenumber derived from Fourier

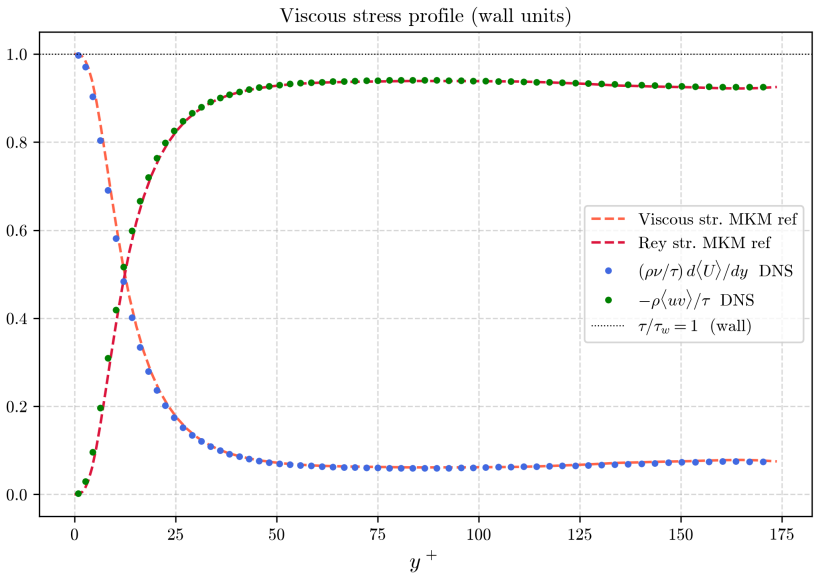


Figure 5.3: Viscous and Reynolds stress adimensionalized with τ.

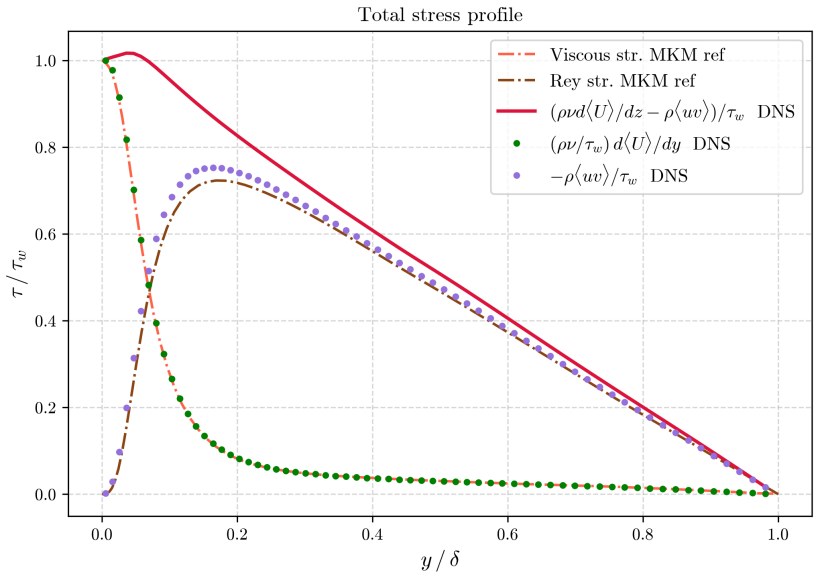


Figure 5.4: Total stress profile adimensionalized with τ_w.

analysis. So, the physiscal consequences in the outer layer are three:

1. **Energy Attenuation at High Frequencies**: The finite difference scheme systematically underestimates the magnitude of spatial derivatives for the smallest resolved scales.

2. **Impact on grid**: The grid designed by Kim et al. was sized assuming the solver could perfectly resolve up to maximum wavenumber value of k_{max}. Since the second-order scheme in CaNS severely damps wavenumbers well before k_{max}, the effective resolution of this first doubly-periodic simulation is lower than the spectral one, even though the physical number of grid points is identical.

3. **Underestimated stresses**: This numerical damping of high-frequency fluctuations artificially filters out a portion of the turbulent kinetic energy at the grid scale. Consequently, the integrated turbulent statistics - such as the Reynolds stresses - exhibit a slight underestimation in the outer layer.

The results shows that the fully-developed state was definitely obtained with the first 40000 number of time steps. The first doubly-periodic simulations shows that CaNS is able to provide the same results as the ones from Moser, Kim, and Mansour even with a wall-normal discretization that is not the most suitable for this case due to the differences codes. This then leads to not a perfect match in the calculated value of the friction velocity that, when squared to adimensionalize the stresses $\langle u_i' u_j' \rangle$, brings to a bigger discrepancy in the results.

5.1.3 Second Doubly-periodic simulation

The only difference between the first doubly-periodic channel and the second, current, one is the discretization in the wall normal direction. In this second simulation, the number of wall-normal points is fixed to 100 since the discretization from Pirozzoli and Orlandi [31] is used. This type of discretization was developed for flows at high friction Reynolds number ($Re_\tau \approx 10^3$) providing a good balance between accuracy and efficiency. The main purpose of this type of stretching is to reduce the number of point in wall-normal direction once the target Re_τ is chosen. In the case of this simulation, for a target friction Reynolds number of 180, the number of points was of 100, approximately 22% less points. This had a big impact on the duration of a single time step: for 129 points, the time needed was about 0.15 seconds,

with 100 points the time lowered to 0.11, bringing an increase in speed of roughly 27%.

Also this second simulation was split in two stages: first stage of 40000 time step to reach the fully-developed state and a second stage of other 40000 time steps where the statistics are computed.

5.1.4 Results for the second Doubly-periodic simulation

Also in this case, the obtained results shows a classical shape, as the previous ones, but the errors are slightly higher due to the type of the stretching of the wall-normal grid which is not optimal for low Reynolds applications.

Table 5.3: Comparison of different quantities between the results for the second doubly-periodic simulation and reference ones at nominal $Re_\tau = 180$

Data set	Actual Re_τ	U_{bulk}	u_τ
Reference [4]	178.13	15.67	1.000
Computed	177.18	15.67	0.994

As is possible to see from the Table 5.3, the values are matching with the ones for the first doubly-periodic channel from Table 5.2. The Figures 5.5,5.6,5.7, and 5.8 exhibits the same behaviour of the ones for the first doubly-periodic channel from Figures 5.1,5.2,5.3, and 5.4. The same explanation as for the first case applies to the small over and under predictions.

The effects of the stretched grid from Pirozzoli are not bringing any difference in results. Actually this discretization provided an increase in performance of the code since the total time to compute 80000 time steps was lower due the fact each time step needed ≈ 0.11 second instead of ≈ 0.15 seconds as in the case of the first doubly-periodic turbulent channel flow.

These results obtained from this second doubly-periodic channel were then exported as a `.dat` file. This generated file is then used to provide the needed information to the digital filter algorithm so that the differences in the results can be attributed to the algorithm itself.

5.2 Main simulation with Digital Filter

The two doubly-periodic simulations established that CaNS reproduces the MKM reference statistics at $Re_\tau = 180$ to within 1% on the integral quan-

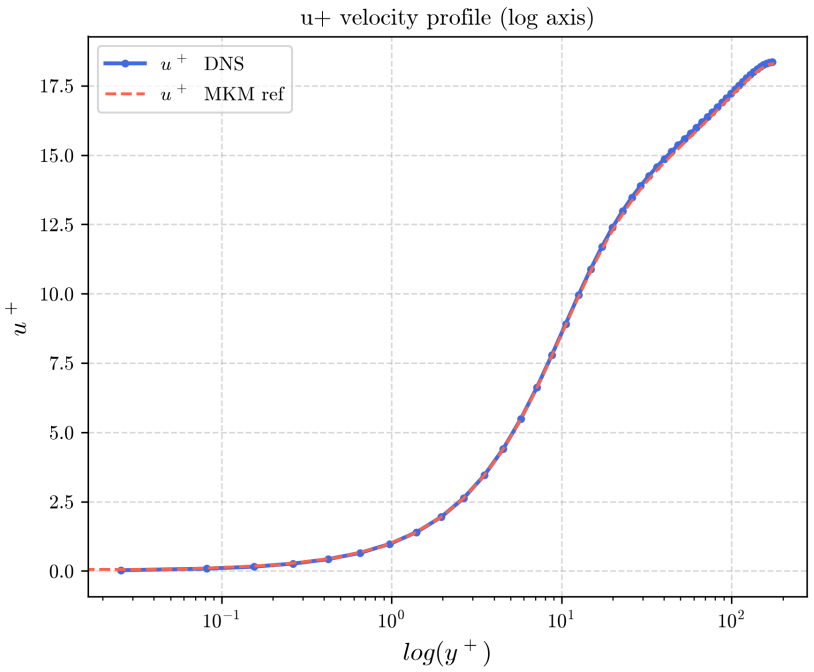


Figure 5.5: Adimensionalized streamwise velocity as a function of $\log(y^+)$

tities, and that the Pirozzoli wall-normal stretching used in the target setup does not degrade the accuracy of the solution. With that baseline in place, the main simulation can now be carried out: a spatially developing turbulent channel flow in which the inflow velocity field is generated on the fly by the digital-filter module described in Chapters 2 and 4. The purpose of this simulation is to verify that the synthetic inflow reproduces the target statistics at the inlet plane, and to quantify the downstream adaptation distance over which the flow recovers a fully developed state.

5.2.1 Setup for channel's simulation

The computational domain is a rectangular box of dimensions $L_x \times L_y \times L_z = 18\pi\delta \times 2\pi\delta \times 2\delta$, i.e. approximately $56.54 \times 6.28 \times 2$ in units of the half-channel height $\delta = 1$, as shown in Figure 5.9. The streamwise length is about 4.5 times the box used in the doubly-periodic simulations of Section 5.1.3, which provides the flow with sufficient distance to develop after the inlet plane before reaching the outflow. The spanwise and wall-normal extents are the same as in the preliminary simulations, so the two configurations differ only in the streamwise direction and the boundary conditions. Currently only the spanwise direction in periodic.

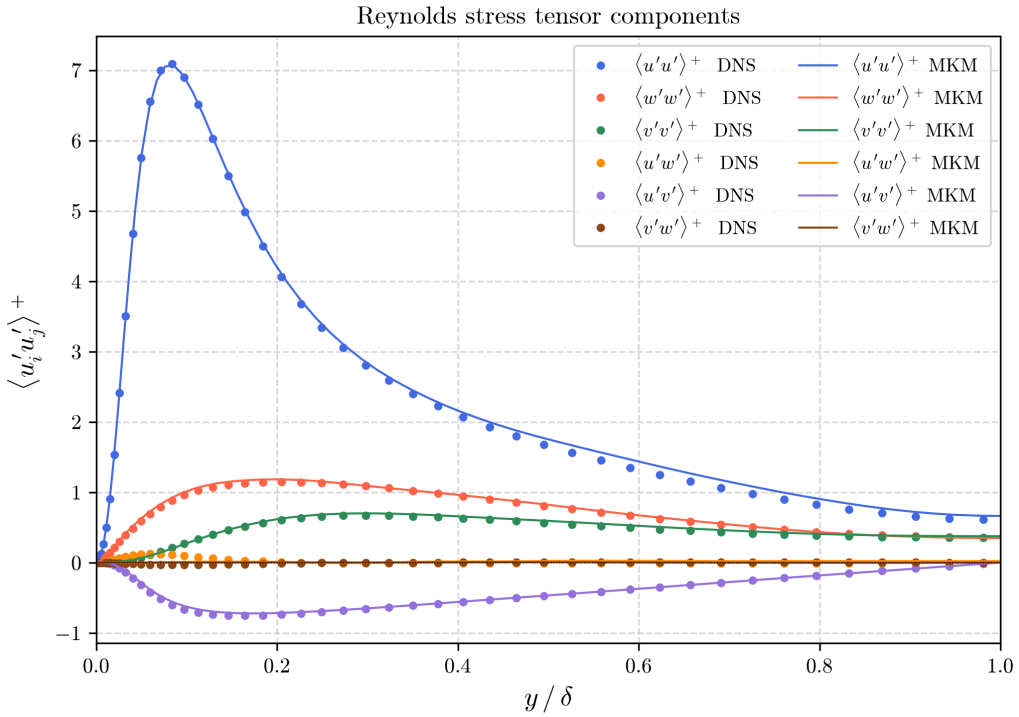


Figure 5.6: Reynolds stress components.

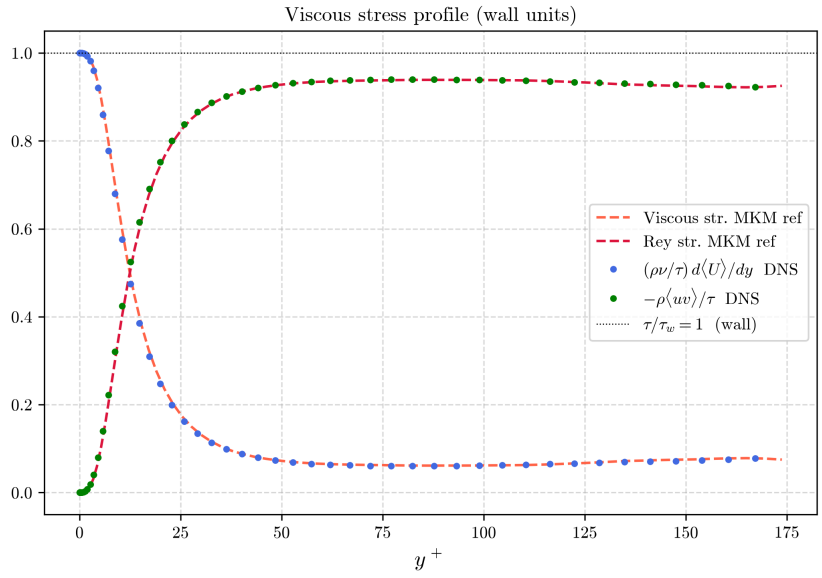


Figure 5.7: Viscous and Reynolds stress adimensionalized with τ.

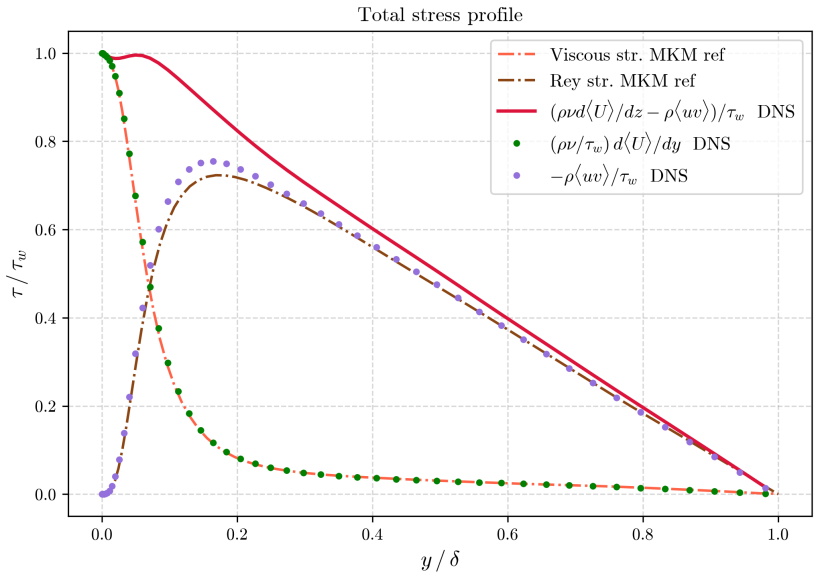


Figure 5.8: Total stress profile adimensionalized with τ_w.

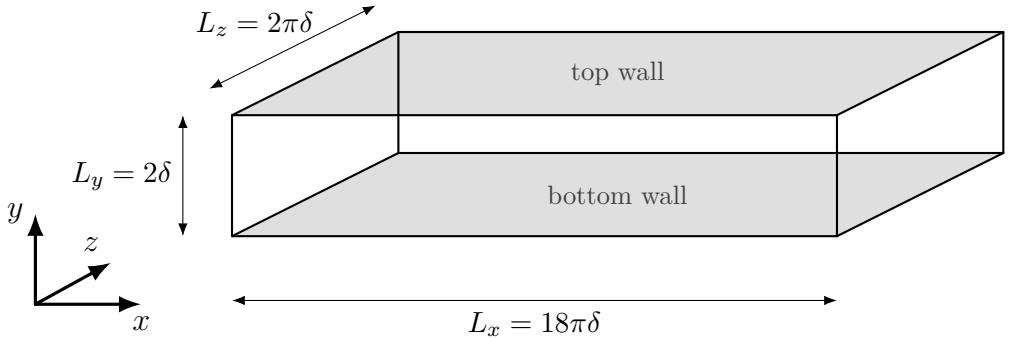


Figure 5.9: Computational domain of the channel-flow DNS. The streamwise direction is periodic with length $L_x = 18\pi\delta$, the wall-normal direction spans the channel height $L_y = 2\delta$ between the two solid walls, and the spanwise direction is periodic with length $L_z = 2\pi\delta$.

The grid has $N_x \times N_y \times N_z = 1272 \times 282 \times 100$ points. Since the simulation is run on 64 cores, this numbers allows a pencil decomposition of 8×8. In the streamwise and spanwise directions the grid is uniform, with $\Delta x^+ = 8.0$ and $\Delta y^+ = 4.0$ at the nominal friction velocity, finer than the doubly-periodic simulations. In the wall-normal direction the grid follows the same Pirozzoli & Orlandi stretching used in the second preliminary simulation in this Chapter, with $N_z = 100$ points over the full channel height. The computational parameters are summarized in Table 5.4.

Table 5.4: Computational parameters for CaNS for the main spatially developing channel simulation with turbulent inlet.

Parameter	Value
$L_x \times L_y \times L_z$	$18\pi \times 2\pi \times 2$
$N_x \times N_y \times N_z$	$1272 \times 282 \times 100$
Δx^+	8.0
Δy^+	4.0
Wall-normal stretching	Pirozzoli & Orlandi, 100 points
Re_τ (nominal)	180
u_τ (reference)	1.0
ν	1/180

Note that in CaNS y direction refers to the spanwise direction which is different from y direction in different Figures that represent the wall-normal direction, as used in literature.

The boundary conditions are summarized as follows. At the inlet plane ($x = 0$) the three velocity components are prescribed through the digital filter module: the mean velocity profiles and Reynolds stress tensor are taken from the MKM reference dataset at $Re_\tau = 180$, and the fluctuating part is generated on the fly by the low-storage formulation described in Chapter 4. At the outlet ($x = L_x$) Neumann conditions `N` are applied. In the spanwise direction (y) the flow is periodic, as in the doubly-periodic simulations. In the wall-normal direction (z) no-slip and impermeability conditions, through Dirichlet condition `D`, are imposed at both walls. The initial velocity field is set to zero; the turbulent inlet alone drives the flow into the domain.

As for the two doubly-periodic simulations, the main simulation was split into two stages: a first stage of 60000 time steps to let the initial quiescent

field reach a fully developed quasi-steady state, and a second stage of 120000 time steps over which the statistics are computed. The total number of time steps is therefore 180000, higher than in the doubly-periodic cases because the streamwise domain is roughly 4.5 times longer and the perturbations injected at the inlet require correspondingly more time to traverse the domain and reach a statistically steady state.

5.2.2 Digital filter inlet configuration

The turbulent inlet generator is configured from the statistics of the second doubly-periodic simulation rather than directly from the MKM reference data. The reason is not the wall-normal discretization: the MKM file is provided on 129 points while the inlet plane has 100, but the natural cubic-spline interpolation described in Section 4.1.2 maps the two grids onto each other automatically, so the point-count mismatch is not a limitation. The reason is numerical consistency. The second doubly-periodic simulation was run with the same solver, the same numerical schemes, and the same wall-normal stretching as the main simulation, so its output statistics are the ones that CaNS would produce in a fully developed channel at this resolution. Using them at the inlet therefore removes the finite-difference truncation error from the inflow signal, which would otherwise be present if the spectral MKM data were used directly. Since the second doubly-periodic results match the MKM reference to within 1% on the integral quantities, this choice does not bias the target statistics; it only makes the inlet consistent with the solver.

The streamwise and spanwise integral length scales used to size the digital filter are also taken from the MKM reference file. These are physical quantities that do not depend on the numerical scheme, so there is no reason to derive them from the CaNS output; using the MKM values keeps the filter parameters directly comparable with the literature values quoted in Chapter 3 and with the Python prototype.

The buffer advance per time step is set by Taylor's hypothesis, $n_{\text{shift}} = \lfloor U_{\text{bulk}} \Delta t / \Delta x \rfloor$, with the sub-cell remainder accumulated between steps as described in Section 4.4. Because the time step in CaNS is adaptive and is recomputed at every step from the CFL condition, this is the only formulation that preserves the correct physical advection rate of the inflow plane over an arbitrarily long simulation. For this simulation, the value of seed was fixed to 42.

5.2.3 Flow development and averaging window

The simulation is started from a quiescent field. The turbulent inlet begins injecting fluctuations from the first time step, and the flow downstream of the inlet develops progressively as the turbulence is advected into the domain. The transient lasts for approximately one convective time based on the bulk velocity,

$$t_{\text{conv}} = \frac{L_x}{U_{\text{bulk}}} = \frac{56.54}{15.67} \approx 3.6,$$

after which the flow reaches a statistically steady state in the sense that the time-averaged statistics become independent of the averaging window. The first stage of the simulation is run for 60000 time steps, corresponding to a total physical time of 123.7 seconds and therefore to approximately 34 flow-through times, long enough to reach a statistically steady state. The first stage instead is run for 120000 time steps, corresponding to a total physical time of ≈ 250 seconds and therefore to approximately 70 flow-through times. This number of flow-through times is exactly in the range $50 - 100$ which is commonly used to compute statistics.

5.2.4 Results

The first quantity analyzed is the local skin friction coefficient $c_f(x)$, defined as:

$$c_f(x) = \frac{\tau_w(x)}{\frac{1}{2}\rho U_{bulk}^2} \tag{5.4}$$

Where $\tau_w(x)$ is the local wall shear stress and ρ is the density, assumed as 1.

Because the inlet plane carries a synthetic turbulent field rather than a genuinely developed one, the local wall shear stress at the inlet does not match the value found in a fully developed channel. The streamwise evolution of $c_f(x)$ therefore provides a direct measure of the adaptation distance: downstream of the inlet, the synthetic fluctuations gradually decay while the physically consistent turbulence develops, and once this process is complete $c_f(x)$ settles to the value it would take in a fully developed flow. The point at which it flattens marks the beginning of the fully developed region.

Figure 5.10 shows the profile of $c_f(x)$ along the streamwise direction, compared with the global friction coefficient C_f obtained from the second doubly-periodic simulation. The behaviour is the one expected from the discussion of Chapter 3: the curve first drops to a lower values of $c_f(x)$ and subsequenlty rises back to a plateau at approximately $x \approx 30\delta$, and oscillates around $C_f = 0.008$ from that point onward, the value commonly reported

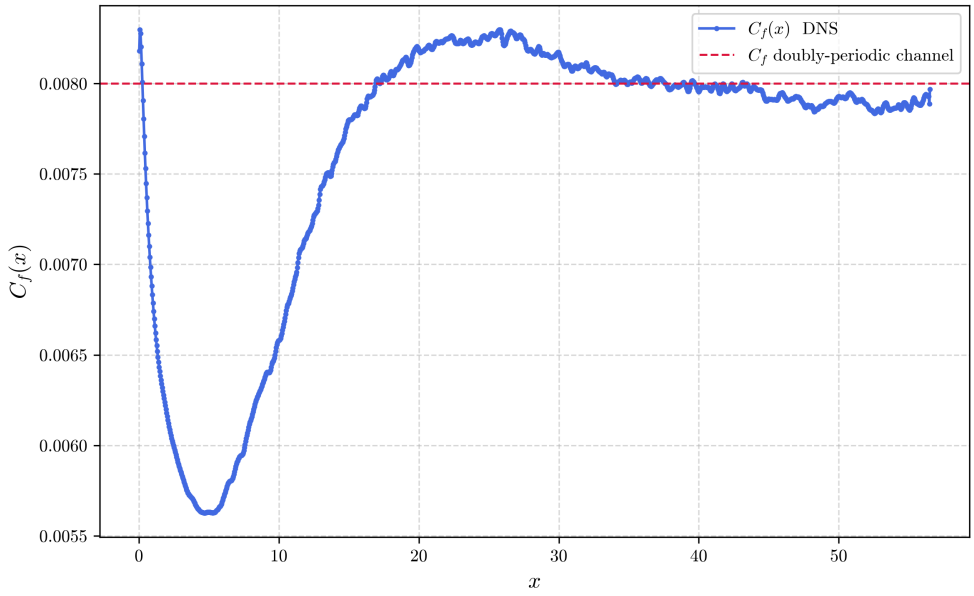


Figure 5.10: Evolution of the local friction coefficient $c_f(x)$ in streamwise direction compared to the value of the global friction coefficient C_f obtained from the second doubly-periodic simulation.

in the literature for a channel flow at $Re_\tau = 180$. The adaptation length of about 30δ is somewhat longer than the typical range of 10δ to 20δ reported in the literature for digital-filter inlets. This is consistent with the constant wall-normal length-scale approximation used in the current implementation, which is known to delay the recovery of the correct near-wall turbulence structure. As already noted in Chapter 3, the adaptation distance is expected to shorten considerably if the wall-normal length scale n_z is allowed to vary with the wall-normal coordinate.

Once $c_f(x)$ has reached its plateau, at approximately $x \approx 30\delta$, the flow can be considered fully developed and its statistics can be averaged in the streamwise direction. Two regions must nevertheless be excluded from the averaging window. The first is the adaptation region downstream of the inlet, which extends up to the plateau location. The second is the vicinity of the outlet plane, where the Neumann condition imposed on the velocity field perturbs the solution in the last few channel half-heights of the domain. The statistics presented in this section are therefore computed over the interval $x \in [30\delta, 50\delta]$, which excludes both regions and covers approximately 20 channel half-heights of fully developed flow.

The behaviour of the velocity and of the stresses is discussed below. Figure 5.11 shows the behaviour of the adimensionalized streamwise velocity in

function of $\log(y^+)$. The matching between the generated velocity field and the reference one from the second doubly-periodic simulation is perfect. Both the inner and outer layer regions are recovered.

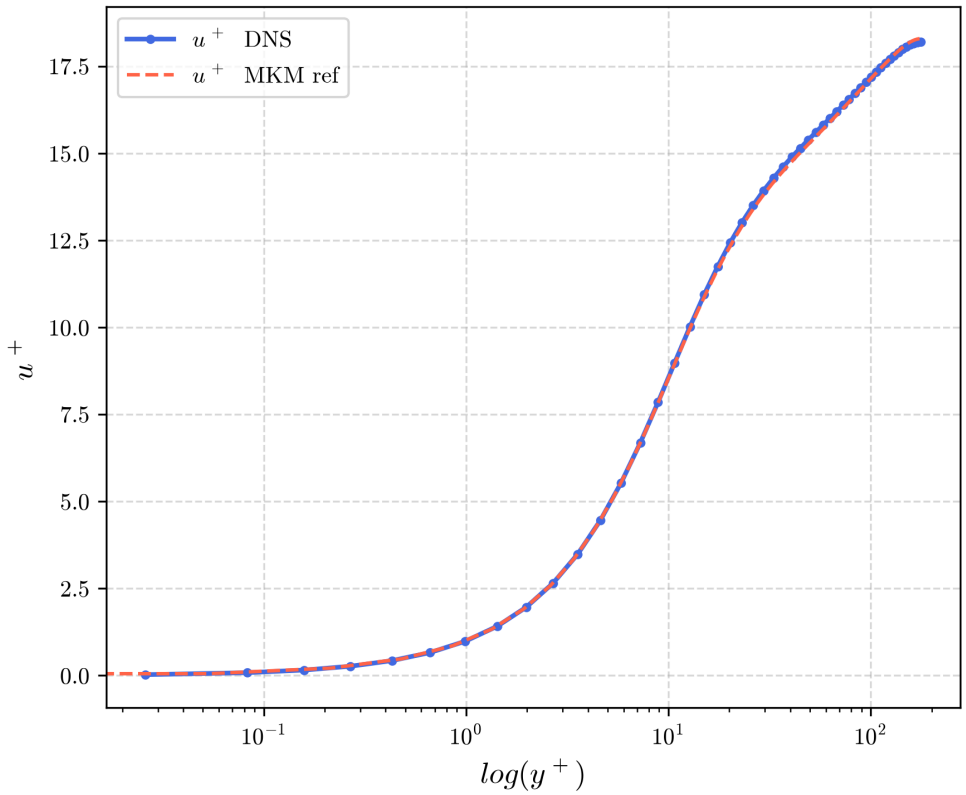


Figure 5.11: Adimensionalized streamwise velocity compared to the reference one.

Reynolds stresses are shown in Figure 5.12, and even in this case the matching to the reference data is pretty accurate. A small underestimation of the components $\langle u'u' \rangle^+$, $\langle v'v' \rangle^+$, and $\langle w'w' \rangle^+$ is observed at lower-mid values of y/δ. The reasons are multiple:

- **Cause 1 - the Gaussian filter kernel**: The digital filter reproduces the target Reynolds stress tensor exactly at the one-point level, because the Lund transformation forces the Cholesky factor of $\mathbf{R}$ onto the random field. What it does not reproduce is the spectral content of the fluctuations. The Gaussian kernel of Klein et al. was derived for homogeneous, isotropic turbulence at a late decay stage, therefore suppresses the streamwise energy content in the buffer layer, which is

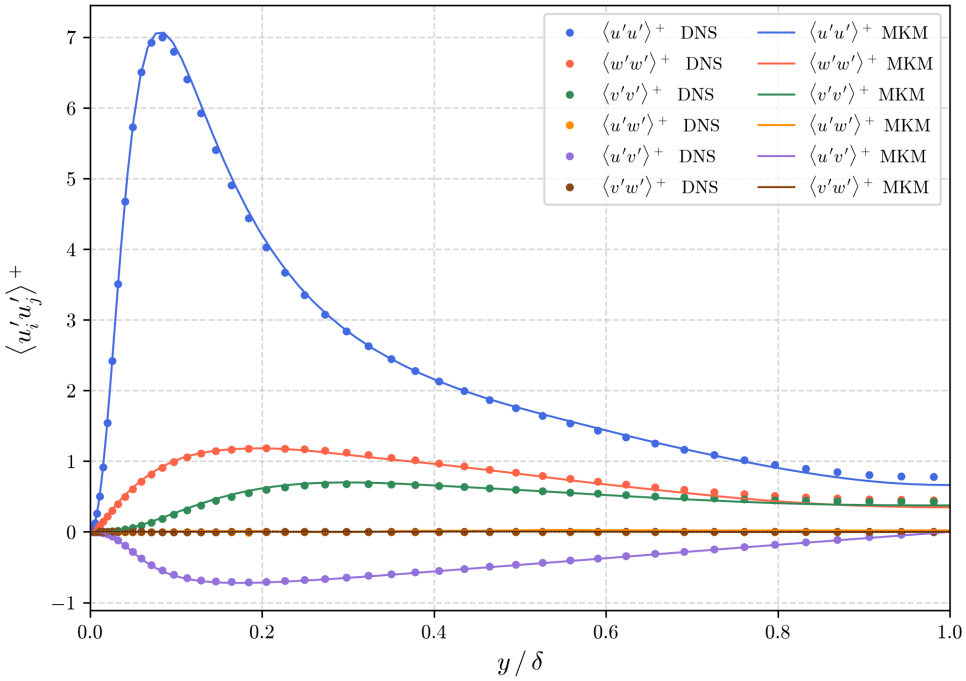


Figure 5.12: Components of Reynolds stress tensor.

precisely the region where $\langle u'u' \rangle^+$ peaks. An exponential filter reproduces the target spectra much more faithfully, as Xie and Castro [23] showed.

- **Cause 2 - continuous injection of biased turbulence**: The statistics in the TIG simulation are not the equilibrium statistics of a closed system, as in the doubly-periodic case. They are the steady state of an open system whose inflow is continuously biased by the Gaussian filter. The flow equilibrates over the adaptation length of about 30δ, after which the mean profile and $c_f(x)$ match the reference, but the second-order structure retains the imprint of the inlet because the turbulence has been conditioned by its history through the synthetic generator.

- **Cause 3 - finite sampling from a single seed**: The inlet field is produced from a single, fixed seed, so the statistics reported here are a single realisation of the underlying stochastic process rather than an ensemble average. As shown in Chapter 3, averaging over multiple seeds significantly reduces the residual differences with respect to the reference data (Figure 3.4), and the single-seed counterpart exhibits a visible scatter that the ensemble average does not. The same effect

is present here: part of the deficit in the Reynolds stresses is a finite-sampling fluctuation rather than a systematic bias, and it could be reduced by repeating the simulation with a different seed and averaging the results. This contribution is however statistical in nature, unlike the two mechanisms discussed above, and its sign is not fixed: a different seed could shift the profiles either above or below the reference.

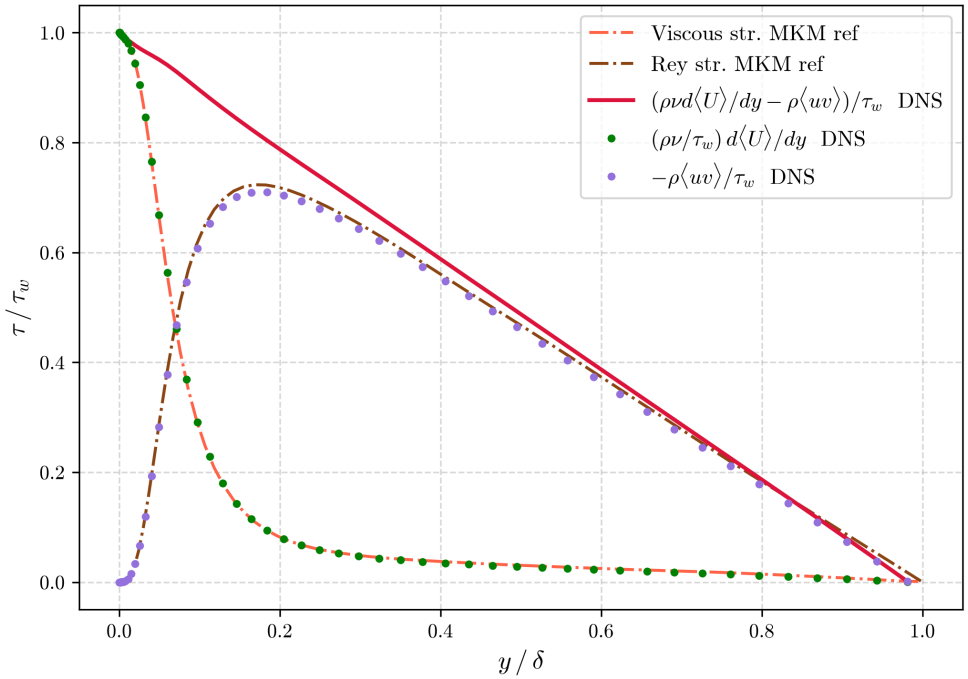


Figure 5.13: Shear stresses in function of the adimensionalized half-channel height.

Figure 5.13 shows how the viscous and Reynolds shear stresses compare to the reference data. The linear behaviour of the total stress - red line - is well recovered and also the general behaviour of the two individual viscous and Reynolds shear stresses. A small underestimation is seen in the Reynolds shear stress and the reason can be attributed to an imperfect match of $\langle u'v'\rangle^+$ and τ_w to the reference values. Figure 5.14 shows the fractional contribution of the two components of the total stress.

In Table 5.5 are presented all the relevant results compared between the main simulation and the reference second double-periodic one.

Taken together, the results presented in this section show that the digital-filter inlet reproduces the target flow at $Re_\tau = 180$ to within a few percent on the integral quantities, and that the profile shapes of the mean velocity

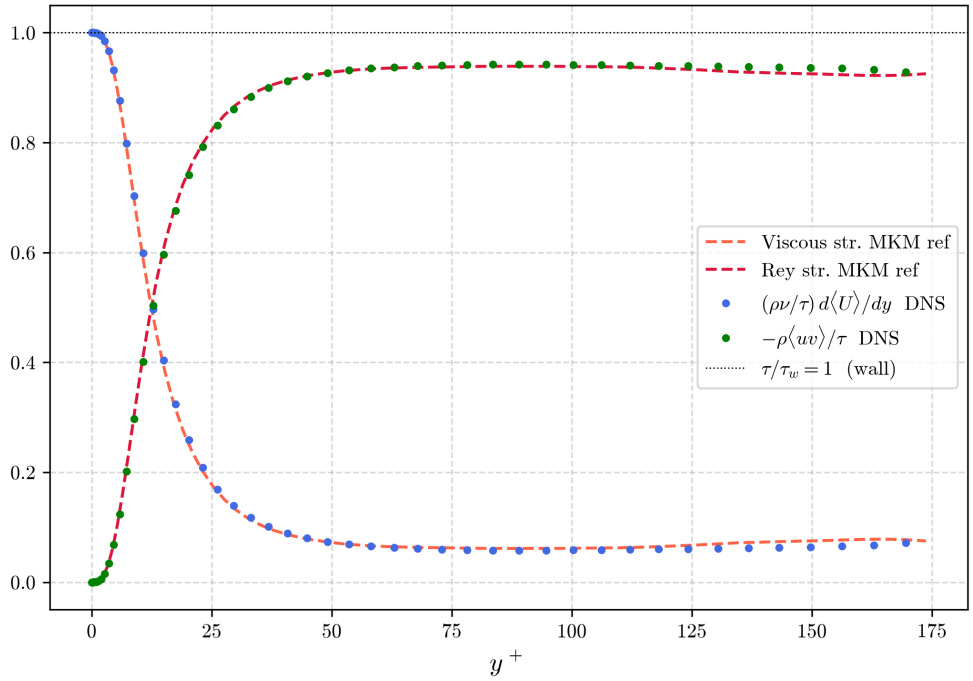


Figure 5.14: Fractional contribution of the viscous and Reynolds stresses to the total stress.

and of the Reynolds stress tensor are captured correctly across the entire channel height. The residual deviations are small and systematic, and they are attributable to the three mechanisms discussed above - the Gaussian filter kernel, the continuous injection of biased turbulence, and the finite sampling from a single seed - rather than to a failure of the algorithm itself. The adaptation length of approximately 30δ is longer than the 10δ to 20δ typically reported for digital-filter inlets, but the plateau value of $c_f(x)$ confirms that the synthetic turbulence does not bias the equilibrium state of the flow: it only delays the distance over which that state is reached. Figure 5.15 shows exactly how the streamwise component of the Reynolds stress $\langle u'u'\rangle^+$ evolves in streamwise direction, proving again that after the adaption distance of 30δ the stress matches the reference data.

The main limitations of the current implementation are the constant wall-normal length scale n_z, the Gaussian shape of the filter kernel, and the single-seed sampling. All three have been identified in the course of the analysis and all three have clear remedies, which are discussed in Section 6 together with the broader outlook of the work.

Table 5.5: Integral quantities for the turbulent channel at nominal $Re_\tau = 180$, comparing the main simulation with the digital-filter inlet (TIG) against the second doubly-periodic reference run.

Quantity	D-P reference	TIG (main)
U_{bulk}	15.67	15.68
u_τ	0.994	0.998
Re_τ	177.18	179.77

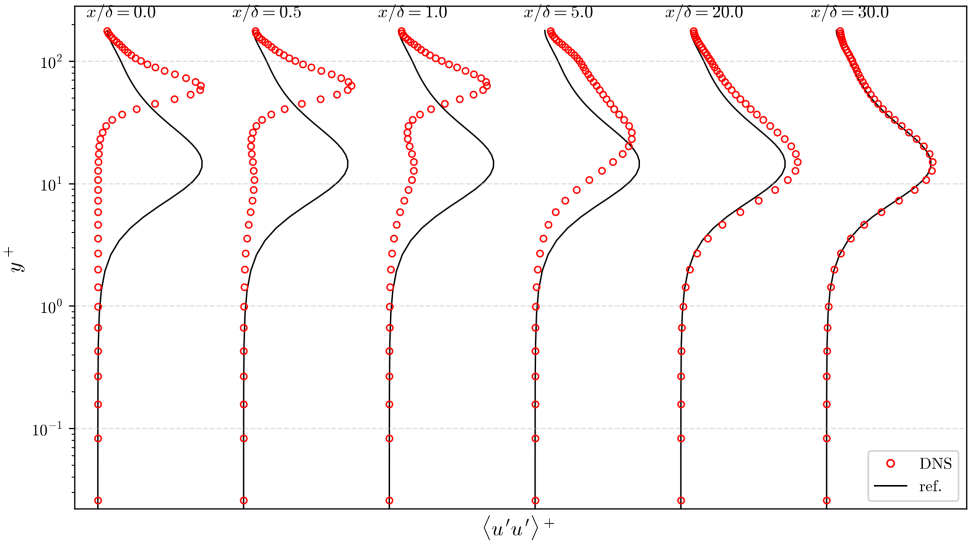


Figure 5.15: Streamwise development of the dimensionless time-averaged streamwise velocity variance.

Chapter 6

Conclusions and Future Work

This thesis addressed the problem of generating physically realistic turbulent inflow conditions for Direct Numerical Simulation of wall-bounded flows. The digital filter method of Klein, Sadiki, and Janicka [1], in the low-storage formulation of Kempf, Wysocki, and Pettit [3], was implemented in a standalone Python prototype for validation, ported to Fortran as the module `mod_tig`, and integrated into the high-performance DNS solver CaNS [6]. The resulting tool was used to simulate a spatially developing turbulent channel flow at a nominal $Re_\tau = 180$, with the inflow plane prescribed by the digital filter and the flow allowed to develop over a domain of length $L_x = 18\pi\delta$.

6.1 Summary of Results

The main simulation confirms that the digital-filter inlet reproduces the target flow to within a few percent on all integral quantities. Table 5.5 shows that the bulk velocity, the friction velocity, and the friction Reynolds number recovered by the main simulation agree with the second doubly-periodic baseline to within 1%: $U_{bulk} = 15.68$ against 15.67, $u_\tau = 0.998$ against 0.994, and $Re_\tau = 179.77$ against 177.18. The mean streamwise velocity profile in Figure 5.11 recovers the viscous sublayer and the log-law region without visible deviation from the reference, and the four components of the Reynolds stress tensor in Figure 5.12 reproduce both the shape and the peak location of the reference profiles across the full channel height. The streamwise development of $\langle u'u' \rangle^+$ shown in Figure 5.15 further confirms that the stress converges to the reference value once the flow has adapted.

The negative aspect is the length over which the synthetic turbulence develops to reach a fully developed state. The local friction coefficient $c_f(x)$ in Figure 5.10 does not reach its plateau until $x \approx 30\delta$ downstream of the inlet,

a value longer than the typical range 10δ to 20δ reported in the literature for digital-filter inlets. The plateau value itself is correct: it oscillates around $C_f \approx 0.008$, consistent with a fully developed channel at $Re_\tau = 180$, so the synthetic inlet does not bias the equilibrium state of the flow; it only delays the distance over which that state is recovered. As already noted in Chapter 5, the longer adaptation is a direct consequence of using a constant wall-normal length scale n_z, which makes the Gaussian kernel too wide in the near-wall region and too narrow in the outer region.

6.2 Answers to the Objectives

The four objectives stated in Section 1.3 can be addressed directly as follows.

1. **Implement and validate the digital filter as standalone Python module.** The algorithm was implemented as the class `InflowGenerator` in Python, capable of handling non-uniform wall-normal grids through an auxiliary uniform grid and general anisotropic Reynolds stress tensors through the Lund transformation. The prototype was validated against the reference channel-flow data at $Re_\tau = 180$ and produced mean velocity and Reynolds stress profiles within a few percent of the reference, as documented in Chapter 3.

2. **Validate against reference DNS data at $Re_\tau = 180$.** The validation confirmed the accurate recovery of the mean velocity profile, of the four relevant Reynolds stress components, and of the target autocorrelation function. The residual deviations observed near the wall were attributed to the Gaussian shape of the filter kernel and to the constant length-scale approximation, rather than to a failure of the algorithm itself.

3. **Port to Fortran and integrate into CaNS.** The algorithm was ported to Fortran as the module `mod_tig`, with explicit loops replacing all NumPy and SciPy operations, a ring buffer with a moving head replacing the physical array shift, and a temporally accurate buffer advance based on Taylor's hypothesis. The module is parallelized with OpenMP and OpenACC and communicates with the solver through the derived type `turbulentInlet_type`. No inter-process communication is required inside the generator, because each rank constructs its own sub-block directly from the global inlet plane.

4. **Run a DNS of a fully developed turbulent channel with the digital-filter inlet.** The main simulation was conducted in a domain

of $L_x \times L_y \times L_z = 18\pi \times 2\pi \times 2$ with a grid of $1272 \times 282 \times 100$ points, and the results were compared with the reference data of Moser, Kim, and Mansour. The comparison showed agreement to within a few percent on the integral quantities, as summarized in Section 6.1.

6.3 Limitations and Future Work

The validation of the algorithm is currently complete only at a single friction Reynolds number. The natural extension is to repeat the analysis at a higher Re_τ, and in particular to validate the digital-filter inlet against the MKM data at $Re_\tau = 395$, where the separation of scales between the inner and outer regions is larger and the constant length-scale approximation is expected to be more strongly challenged. This will require new integral length scales and a fresh set of filter parameters, but no change to the module structure. Beyond the study of turbulent channels, one of the ultimate goals is to use this algorithm for the simulation of turbulent boundary layers.

Several improvements to the algorithm itself can shorten the adaptation distance and increase the fidelity of the generated statistics.

Wall-normal-dependent length scales. The constant value of n_z used in the present implementation is the main cause of the long adaptation distance. A wall-normal profile $n_z(z)$, derived from the local integral length scale of the reference data, would let the Gaussian kernel follow the physical variation of the turbulence and would substantially reduce the adaptation length. The cost of this change is small: a separate stencil per wall-normal position, precomputed once during initialization.

Exponential and Laguarda-Hickel kernels. The Gaussian kernel of Klein et al. was derived for homogeneous, isotropic turbulence at a late decay stage and produces a premature drop in spectral energy above the energy-containing range. Exponential kernels, as proposed by Xie and Castro [23], and the transversal correlation of Laguarda and Hickel [7], which includes negative values and better resembles the true correlation function of a shear flow, reproduce the target spectra more faithfully. The change is confined to the coefficient computation and does not affect the rest of the algorithm. This is a small improvement that can already provide results with higher fidelity.

Divergence-free formulation. The standard digital filter does not produce a strictly divergence-free velocity field, and the staggered-grid correc-

tion in CaNS reintroduces a small residual divergence at the inlet plane. The divergence-free synthetic eddy method of [30] would remove this inconsistency at the cost of a more elaborate eddy representation.

Ensemble averaging. The statistics reported in Chapter 5 are a single realisation from a fixed seed. A true ensemble average requires repeating the simulation with a different global seed and averaging the results, which multiplies the computational effort by the number of realisations. Re-seeding the generator at every time step would not achieve the same effect, because the downstream field retains the memory of the inlet through advection and different inlet planes within the same run are therefore not independent realisations of the equilibrium state.

Higher-order statistics. The digital filter currently reproduces only the first and second one-point moments of the target flow. Extending it to prescribe higher-order statistics, for example the third moments that appear in the budget of the Reynolds stresses, would improve the structural fidelity of the generated field, but at the cost of additional computational effort and a more complex implementation. This is a natural direction for future work if the residual deficits observed in the second-order moments near the wall prove to be limiting for a particular application.

Taken together, the results of this thesis show that the low-storage digital filter of Kempf et al. can be integrated into a finite-difference DNS solver without loss of accuracy and without sacrificing parallel scalability, and that the resulting inflow generator produces physically realistic turbulence at $Re_\tau = 180$ with an adaptation distance that is longer than the literature values but still acceptable for domains of the length used here. The improvements listed above address the remaining limitations and point the way towards a generator that can be applied at higher Reynolds numbers and in more complex configurations.

Bibliography

[1] Klein M., Sadiki A., and Janicka J. Digital filter based generation of inlet conditions for dns and les. *Journal of Computational Physics. doi: https://doi.org/10.1016/S0021-9991(03)00090-1*, 2003.

[2] Lund T.S., Wu X., and Squires K.D. Generation of turbulent inflow data for spatially-developing boundary layer simulations. *Journal of Computational Physics. doi: https://doi.org/10.1006/jcph.1998.5882*, 1998.

[3] Kempf A.M., Wysocki S., and Pettit M. An efficient, parallel low-storage implementation of klein's turbulence generator for les and dns. *Computers and Fluids. doi: https://doi.org/10.1016/j.compfluid.2012.02.027*, 2012.

[4] Robert Moser, John Kim, and Nagi Mansour. Direct numerical simulation of turbulent channel flow up to $re_\tau = 590$. *Physics of Fluids - PHYS FLUIDS*, 11:943–945, 04 1999.

[5] John Kim, Parviz Moin, and Robert Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of Fluid Mechanics*, 177:133–166, 1987.

[6] Costa P. Cans: Canonical navier-stokes solver. *https://github.com/p-costa/CaNS*, 2020.

[7] Luis Laguarda and Stefan Hickel. Analysis of improved digital filter inflow generation methods for compressible turbulent boundary layers. *Computers and Fluids*, 268, 2024.

[8] Pope S. B. Turbulent flows. *Journal of Computational Physics.*, 2000.

[9] Lorenz E. N. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences. 20 (2): 130âĂŞ141*, 1963.

[10] Dhamankar N. S., Baba-Ahmandi M. H., and Lyrintzis A. S. Overview of turbulent inflow boundary conditions for large-eddy simulations. *AIAA J 2017:1-18. doi: https://doi.org/10.2514/1.J055528*, 2015.

[11] Wu X. Inflow turbulence generation methods. *Annual Review of Fluid Mechanics. doi: https://doi.org/10.1146/annurev-fluid-010816-060322*, 2017.

[12] Tabor G.R. and Baba-Ahmadi M.H. Inlet conditions for large eddy simulation: A review. *Computers and Fluids. doi: https://doi.org/10.1016/j.compfluid.2009.10.007*, 2010.

[13] Spalart P.R. and Leonard A. Direct numerical simulation of equilibrium turbulent boundary layers. *Turbulent Shear Flows 5, Springer Berlin Heidelberg. doi: https://doi.org/10.1007/978-3-642-71435-1_20*, 1987.

[14] Nikitin N.V. Spatial periodicity of spatially evolving turbulent flow caused by inflow boundary condition. *Physics of Fluids. doi: https://doi.org/10.1063/1.2781596*, 2007.

[15] Morgan B., Larsson J., Kawai S., and Lele S.K. Improving low-frequency characteristics of recycling/rescaling inflow turbulence generation. *AIAA Journal. doi: https://doi.org/10.2514/1.J050705*, 2011.

[16] Araya G., Castillo L., Meneveau C., and Jansen K. A dynamic multi-scale approach for turbulent inflow boundary conditions in spatially developing flows. *Journal of Fluid Mechanics. doi: https://10.1017/S0022112010005616*, 2011.

[17] Lee S., Lele S.K., and Moin P. Simulation of spatially evolving turbulence and the applicability of taylor's hypothesis in compressible flow. *Physics of Fluids. doi: https://doi.org/10.1063/1.858425*, 1992.

[18] Smirnov A., Shi S., and Celik I. Random flow generation technique for large eddy simulations and particle-dynamics modeling. *Journal of Fluids Engineering*, 2001.

[19] Druault P., Lardeau S., Bonnet J., Coiffet F., Delville J., Lamballais E., Largeau J., and Perret L. Generation of three-dimensional turbulent inlet conditions for large-eddy simulation. *AIAA Journal. doi: https://doi.org/10.2514/1.3946*, 2004.

[20] Sandham N.D., Yao Y.F., and Lawal A.A. Large-eddy simulation of transonic turbulent flow over a bump. *International Journal of Heat and Fluid Flow. doi: https://doi.org/10.1016/S0142-727X(03)00052-3*, 2003.

[21] Touber E. and Sandham N.D. Large-eddy simulation of low-frequency unsteadiness in a turbulent shock-induced separation bubble. *Theoretical and Computational Fluid Dynamics. doi: https://doi.org/10.1007/s00162-009-0103-z*, 2009.

[22] Batchelor G. The theory of homogeneous turbulence. *Cambridge University Press*, 1953.

[23] Xie Z.T. and Castro I.P. Efficient generation of inflow conditions for large eddy simulation of street-scale flows. *Flow, Turbulence and Combustion. doi: https://doi.org/10.1007/s10494-008-9151-5*, 2008.

[24] Kim Y., Xie Z.T., and Castro I.P. Divergence-free turbulence inflow conditions for large-eddy simulations with incompressible flow solvers. *Computers and Fluids. doi: https://doi.org/10.1016/j.compfluid.2013.06.001*, 2013.

[25] Bercin K.M., Xie Z.T., and Turnock S.R. Exploration of digital-filter and forward-stepwise synthetic turbulence generators and an improvement for their skewness-kurtosis. *Computers and Fluids. doi: https://doi.org/10.1016/j.compfluid.2018.03.070*, 2018.

[26] Kempf A., Klein M., and Janicka J. Efficient generation of initial- and inflow-conditions for transient turbulent flows in arbitrary geometries. *Flow, Turbulence and Combustion. doi: https://doi.org/10.1007/s10494-005-3140-8*, 2005.

[27] Sergent E. *Vers une mthodologie de couplage entre la simulation des grandes chelles et les modles statistiques*. PhD thesis, Ecole centrale de Lyon, 2002.

[28] Jarrin N., Benhamadouche S., Laurence D., and Prosser R. A synthetic-eddy-method for generating inflow conditions for large-eddy simulations. *International Journal of Heat and Fluid Flow. doi: https://doi.org/10.1016/j.ijheatfluidflow.2006.02.006*, 2006.

[29] PamiÃĺs M., Weiss P.E., Garnier E., Deck S., and Saganut P. Generation of synthetic turbulent inflow data for large eddy simulation of spatially

evolving wall-bounded flows. *Physics of Fluids. doi: https://doi.org/10.1063/1.3103881*, 2009.

[30] Yann Dreze, Muting Hao, and Luca di Mare. Divergence-free turbulent inflow data from realistic covariance tensor. *Physics of Fluids. doi: https://doi.org/10.1063/5.0136568*, 35(2), 2023.

[31] Sergio Pirozzoli and Paolo Orlandi. Natural grid stretching for dns of wall-bounded flows. *Journal of Computational Physics*, 439:110408, 2021.