Universitat Politècnica de Catalunya

Escola Superior d'Enginyeries Industrial, Aeroespacial
i Audiovisual de Terrassa

Numerical resolution of Burgers equation

Computational Engineering

220027

Professor

Carlos David Pérez Segarra

Francesc Xavier Trias Miquel

Author

Sergio Gutiérrez Sánchez

Saturday 22nd January, 2022

Abstract

In this report, it is presented the theoretical basis of the Burgers equation in Fourier space, alongside its numerical implementation. In addition to that, multiple different studies concerning physical and numerical phenomena involved in Burgers equation are also treated. Among these there can be found Reynolds number and number of modes influence and even a discerning study about the suitability of the implementation of LES numerical techniques.

1 Introduction.

Navier Stokes equations provide a great mathematical tool and description of fluid's behaviour and specially, nonlinear dynamics of turbulence. However, a Direct Numerical Simulation (DNS) is difficult and extremely expensive in what comes to implementation and computational time.

Expression (1) shows incompressible Navier Stokes Momentum Conservation equation

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \cdot \mathbf{u} = \frac{1}{Re}\nabla^2 - \nabla p \qquad (1)$$

The convective term of the equation $((\mathbf{u} \cdot \nabla) \cdot \mathbf{u})$ produces too many scales of fluid motion. These can be approximated as a function of Reynolds (Re) number in the following way [1].

$$\delta t \approx Re^{-1/2} \qquad (2)$$

$$\delta x \approx Re^{-3/4} \qquad (3)$$

On the other hand, these scales increase the necessary memory by a factor of $Re^{9/4}$, and so the computational costs by $Re^{11/4}$. Therefore, it is extremely expensive and unaffordable for regular or personal computers.

Instead of the classical Navier Stokes approach, Johannes Martinus Burgers came with the idea of a simplified model of the equations in Fourier space.

1.1 Burgers equation

Navier Stokes Momentum Conservation equation can be written in 1D as follows.

$$\frac{\partial u}{\partial t} + u \cdot \frac{\partial u}{\partial x} = \frac{1}{Re} \cdot \frac{\partial^2 u}{\partial x} + f \qquad (4)$$

However, considering this equation on an interval Ω with periodic boundary conditions, it can be written in Fourier space in the following way.

$$\frac{\partial \hat{u}_k}{\partial t} + \sum_{k=p+q} \hat{u}_p i q \hat{u}_q = -\frac{k^2}{Re}\hat{u}_k + F_k \qquad (5)$$

Where:

- $k = 0, ..., N$
- N is the total number of Fourier modes
- $F_k = 0$ for k > 1
- $F_1 \quad \frac{\partial \hat{u}_1}{\partial t} = 0$ for t > 0
- $\hat{u}_k$ is the k-th Fourier coefficient of $u(x,t)$ (Navier Stokes 1D) (equation (6))

$$u(x) = \sum_{k=-N}^{k=N} \hat{u}_k \cdot e^{ikx} \qquad (6)$$

Equation (5) can be mathematically developed and simplified using the Kinetic energy transport equation. However, this process will not be shown or explained in this report. Further details can be found in [2].

The final generic Burgers equation that has been computationally solved is presented next

up.

$$\sum_{k=-N}^{k=N} \frac{\partial \hat{u}_k}{\partial t} e^{ikx} + \sum_{p=-N,q=-N}^{p=N,q=N} \hat{u}_p iq \hat{u}_q e^{i(p+q)x} = \sum_{k=-N}^{k=N} (-k^2 \hat{u}_k) e^{ikx} \quad (7)$$

In previous equations, there can be distinguished 3 different terms from conventional Navier Stokes Momentum equation.

First of them is the transient term of the velocity, $\sum_{k=-N}^{k=N} \frac{\partial \hat{u}_k}{\partial t} e^{ikx}$. It basically represents the rate of change of fluid's velocity but written in a Fourier space.

Secondly, the convective term of the equation, $\sum_{p=-N,q=-N}^{p=N,q=N} \hat{u}_p iq \hat{u}_q e^{i(p+q)x}$. This is the source of nonlinearity of the mathematical expression. It is because of this term that energy can "move" throught different scales of motion. Mainly from large scales to smaller ones (high frequency Fourier modes), but also on the opposite direction (energy backscattering). The physical effects of this term will be shown in a more detailed way in results section (5).

Finally, the last term of equation (7) is the diffusive term, $\sum_{k=-N}^{k=N} (-k^2 \hat{u}_k) e^{ikx}$. Its main physical characteristic is the dissipation or damping of energy. This effect becomes much clear and effective for high-frequency modes in Fourier space.

In addition to that, it is necessary to give a brief explanation of the energy term that has been multiple times mentioned before.

These previously mentioned dynamic scales of motion can be characterised by their energy, more precisely, by its kinetic energy.

This kinetic energy is usually computed in conventional Navier Stokes as the product of the velocities of the flow. In a Fourier space, the term E_k is the energy of the k-th mode. This represents the energy of the scale of motion of $\hat{u}_k$. The mathematical expression that allows to compute it is shown next up in equation (8).

$$E_k = \hat{u}_k \cdot \hat{u}_{-k} \quad (8)$$

Where $\hat{u}_{-k}$ is the complex conjugate of $\hat{u}_k$.

The evolution of E_k as a function of k (the Fourier mode) is known as "Energy spectrum". It is the most common way to represent the different scales of motion of a fluid.

2 Description of numerical methods.

Once the theoretical basis of Burgers equation has been presented and briefly introduced, numerical methods applied to its computational resolution are presented.

Burgers equation is used to determine the scales of motion of fluids, as it is possible too with Navier Stokes. However, Burgers equation do not require a computational mesh in order to be solved. Therefore, unlike in other simulations presented in this course, here, it isn't necessary to summarise briefly the mesh generation process.

Instead of that, equations discretization will be treated directly.

2.1 Equations discretization

Each of the Burgers equation terms mentioned in section 1 will be discretized separately.

Transient term

$$\sum_{k=-N}^{k=N} \frac{\partial \hat{u}_k}{\partial t} e^{ikx}$$

As there can be seen, this term represent a sum of the different Fourier modes. In order to calculate each of the it is necessary to discretize it in the following way.

$$\frac{\partial \hat{u}_k}{\partial t} \longrightarrow \frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{\Delta t} \quad (9)$$

Then, it is needed to calculate the appropiate value for the step time value Δt. In order to

do that, Courant - Friedrich - Lewy condition is applied.

$$\Delta t < C_1 \frac{Re}{N^2} \tag{10}$$

Where C_1 is the Courant constant, which vary depending on the simulation characteristics such as the integration scheme or if the simulation is DNS or LES (Large Eddy Simulation).

Convective term

$$\sum_{p=-N,q=-N}^{p=N,q=N} \hat{u}_p i q \hat{u}_q e^{i(p+q)x}$$

As it has been said, the convective term is the reason for the development of Burgers equation. Therefore, it is likely to be the most complex one among the 3 different terms to be discretized.

In this case, it is necessary to calculate what's known as "Triadic Interactions". These are the way in which the energy movement between larger and smaller scales is represented.

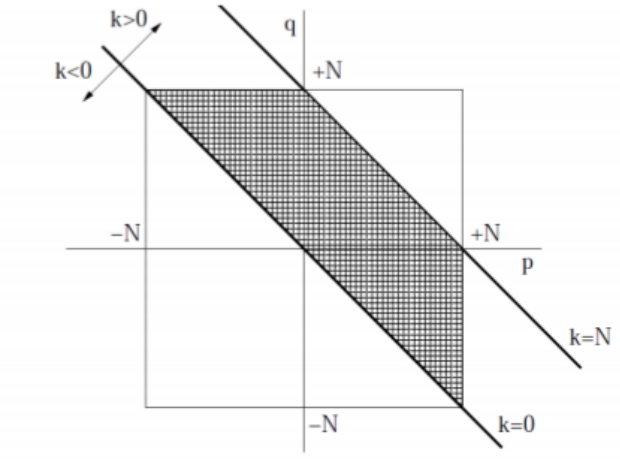

Figure 1: Representation of all triadic interaction between modes. Extracted from [2]

This energy movement are represented by the different interaction between different Fourier modes. It is presented in figure 1 all the possible triadic interactions between these.

However, due to the characteristics of the convective term, only the interaction of the shadowed region are considered for this case.

Again, previous mathematical expression is the sum of all modes of the term, however, in order to compute them, it is necessary to discretize as the multiplication of the following terms.

$$\hat{u}_p i q \hat{u}_q \tag{11}$$

There must be taken into account that $\hat{u}_p$, $\hat{u}_q$ and i are complex numbers, which must be splited into real and imaginary part to be able to operate them.

By doing this, it is possible to arrive to the following expression.

$$C_{Re} = q \cdot [-\hat{u}_{pRe} \cdot \hat{u}_{qIm} - \hat{u}_{pIm} \cdot \hat{u}_{qRe}] \tag{12}$$

$$C_{Im} = q \cdot [\hat{u}_{pRe} \cdot \hat{u}_{qRe} - \hat{u}_{pIm} \hat{u}_{qIm}] \tag{13}$$

Where C_{Re} and C_{Im} denote the real and imaginary terms of the convective contribution. respectively.

It is important to mention that there are several cases in which equations (12) and (13) are slightly modified. These take place with doing the sum of all the p and q modes interaction terms when one of them is negative. In this case, the velocity used is the complex conjugate of the original one.

There is another special case when p or q are zero. In this case no contribution to the sum is considered.

Diffusive term

$$\sum_{k=-N}^{k=N} (-k^2 \hat{u}_k) e^{ikx}$$

The last term of the equation is the diffusive contribution. As it has been said, its main characteristic is the energy damping.

It can be numerically discretized in the following way.

$$D(k)_{Re} = \nu \cdot k^2 \cdot \hat{u}_{kRe} \tag{14}$$

$$D(k)_{Im} = \nu \cdot k^2 \cdot \hat{u}_{kIm} \quad (15)$$

Where ν denotes the kinetic viscosity and k is the Fourier mode.

Again, since $\hat{u}_k$ is a complex number, it is necessary to calculate separately the real and the imaginary contribution of the term.

In order to determine mentioned kinematic viscosity ν, it is possible to do in several ways.

First, if the simulation is a DNS, this viscosity can be directly computed as a function of Re number in the following way.

$$\nu = \frac{1}{Re} \quad (16)$$

However, in case of having a different type if simulation, such as LES or RANS (Reynolds Averaged Navier Stokes), it is necessary to add a new term to this already calculated viscosity from equation (16).

$$\nu_{effective} = \nu + \nu_t \quad (17)$$

For this report, it has been only considered LES simulations. In this case, it has been used the viscosity function from an spectral eddy-viscosity model (equation (18)).

$$\nu_t = \nu_t^{\infty} \cdot (\frac{E_{kN}}{k_N})^{1/2} \cdot \nu_t^*(\frac{k}{k_N}) \quad (18)$$

To found the extended mathematical of equation (18), go to [2].

2.2 Integration scheme

Finally, in what comes to the numerical implementation, it is necessary to give a brief explanation about the temporal integration of Burgers equation.

For this, two different schemes have been implemented in the code.

First of them is a first order explicit Euler scheme.

$$\hat{u}_k^{n+1} = \hat{u}_k^n + \Delta t \cdot [-C(k)^n - D(k)^n] \quad (19)$$

And the other one, a second order Adam - Bashforth scheme.

$$\hat{u}_k^{n+1} = \hat{u}_k^n + \Delta t \cdot [\frac{3}{2} \cdot (-C(k)^n - D(k)^n) - \frac{1}{2} \cdot (-C(k)^{n-1} - D(k)^{n-1})] \quad (20)$$

It must be mentioned that these temporal integration schemes must be applied for both real and imaginary terms of each complex value.

The numerical performance of each method is presented in section 5.

3 Code structure.

In this section, it is presented the main features or aspects of the computational code developed for the resolution of Burgers equation. To see this code go to annex A.

The programming language selected is C++. Unlike in other Computational Engineering subjects reports, in this case, the code is not object-oriented. Instead of that, many different functions have been developed, each of them covering an important task of the simulation process.

In what comes to paralelisation, in this case, due to the tiny computational requirements of the problem, there has been selected to implement a sequential structure for the code, instead of applying paralelisation basis.

Apart from that, next up is presented a brief scheme of Burgers equation numerical resolution process.

1. Declaration of initial input variables
2. Set of initial conditions
3. Resolution process
   (a) Calculation Process
      i. Calculation of viscosity
      ii. Calculation of diffusive term
      iii. Calculation of convective term

iv. Calculation of next step velocity

(b) Checking of convergence criteria

(c) Variables update

4. Final post-processing data calculation

4 Code Verification

In order to verify the results obtained with the developed codes, there are presented two different comparison. One for the DNS type simulations, and the other one for LES.

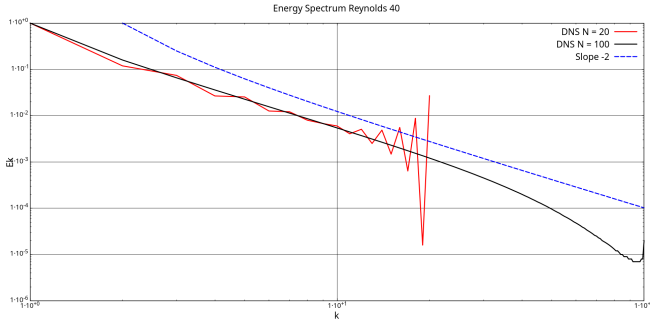

Figure 2: DNS Verification results

As there can be seen in both figures, there is a clear match between the obtained results and the comparison. Apart from minor differences in for N = 20 at the beginning of the plots, the rest are virtually exactly. Therefore, future results obtained in DNS simulations can be considered as correct.

4.1 DNS Verification

The parameters of the simulation are presented in table 2.

Table 1: DNS Verification simulation parameters

Parameter	Value
Re	40
C_1	0.04
C_K	0.4223
Convergence	$1 \cdot 10^{-6}$

The results obtained are shown in figure 2. At the same time, verification results from [1] are presented in figure 3.

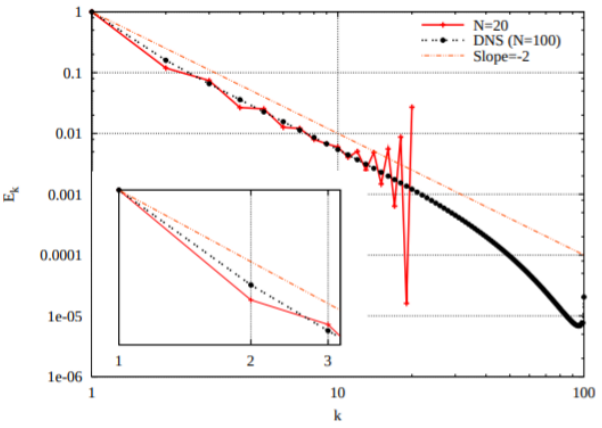

Figure 3: DNS comparison results. Extracted from [1]

4.2 LES Verification

The parameters of the simulation are presented in table 2.

Table 2: DNS Verification simulation parameters

Parameter	Value
Re	40
C_1	0.04
Convergence	$1 \cdot 10^{-6}$

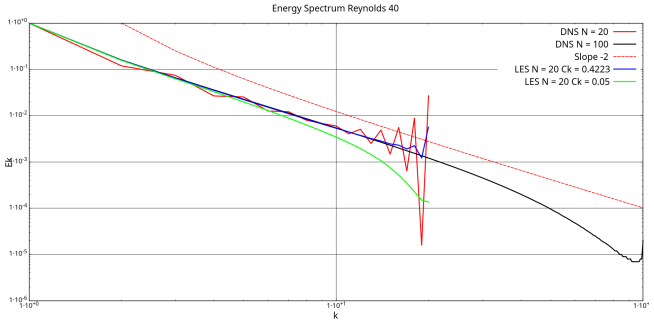

Figure 4: LES Verification results

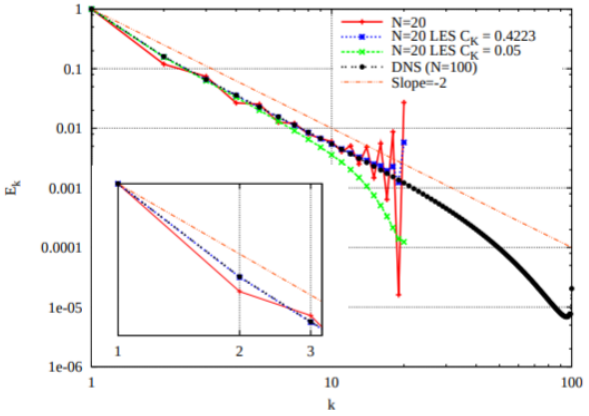

Figure 5: LES comparison results. Extracted from [1]

Again, as in the DNS Verification, there can be seen a clear match between both simulations results.

Therefore, it can be concluded that, both DNS and LES results are correct.

5 Study cases.

Once the theoretical and numerical framework of Burgers Equation have been explained, and so the code verification has been shown, some numerical studies done for this case are presented.

5.1 Reynolds influence study

As it has been mentioned in section 1, the scales of motion are completely related with Reynolds number. It is interesting to study how the energy spectrum of these is affected depending on the Reynolds number. In this case, it is compulsory to select a DNS simulation. Otherwise, it wouldn't be possible to appreciate the smaller scales.

Simulation parameters are presented in table 3.

Table 3: Reynolds Study simulation parameters

Reynolds Number	**Number of modes**	C_1
1.0	100	0.04
5.0	100	0.04
20.0	100	0.04
80.0	200	0.04
200.0	200	0.02

Next up, in figure 6 are presented the obtained results.

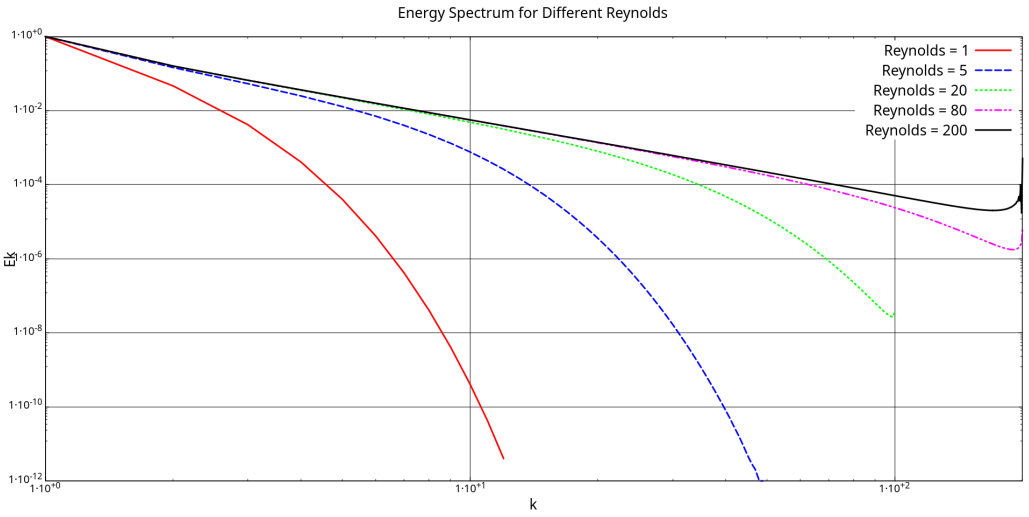

Figure 6: Reynolds Study obtained results

There are several conclusions that can be extracted from the results from figure 6.

First of them is the need of decreasing the Courant number (C_1) (table 3) in order to be able to simulate Reynolds 200 without divergence.

And secondly, what has been mentioned in section 1. Reynolds number is the relationship between inertial and viscous forces. Therefore, the lower the Reynolds, the higher the viscosity and vice versa.

Diffusive term is characterised in Burgers equation for damping energy, and this grows with the viscosity value. As there can be seen in plot 6, for lower Reynolds, the viscosity is much higher and so the energy damping due to diffusion at higher frequency modes.

On the other hand, for higher Reynolds numbers, this energy dissipation is much lower. As it can be observed, the energy spectrum doesn't decay as much as for lower Reynolds for high frequency modes.

5.2 Integration scheme study

Another important aspect that has been mentioned previously is the time integration scheme (section 2). For the present report, there have been implemented two different integration methods, a first order Euler and a second order Adam-Bashforth scheme.

The numerical expression of each of them can be found in section 2.

In order to determine if it is worth to implement a higher than 1 order scheme, it has been decided to simulate the same Reynolds number with both schemes.

The simulation parameters are presented in table 4

Table 4: Reynolds Study simulation parameters

Scheme	**Reynolds Number**	**Number of modes**	C_1
Euler	100.0	200	0.03
Adam-Bashforth	100.0	200	0.05

Next up, in figure 7 are presented the results obtained for both simulation cases.

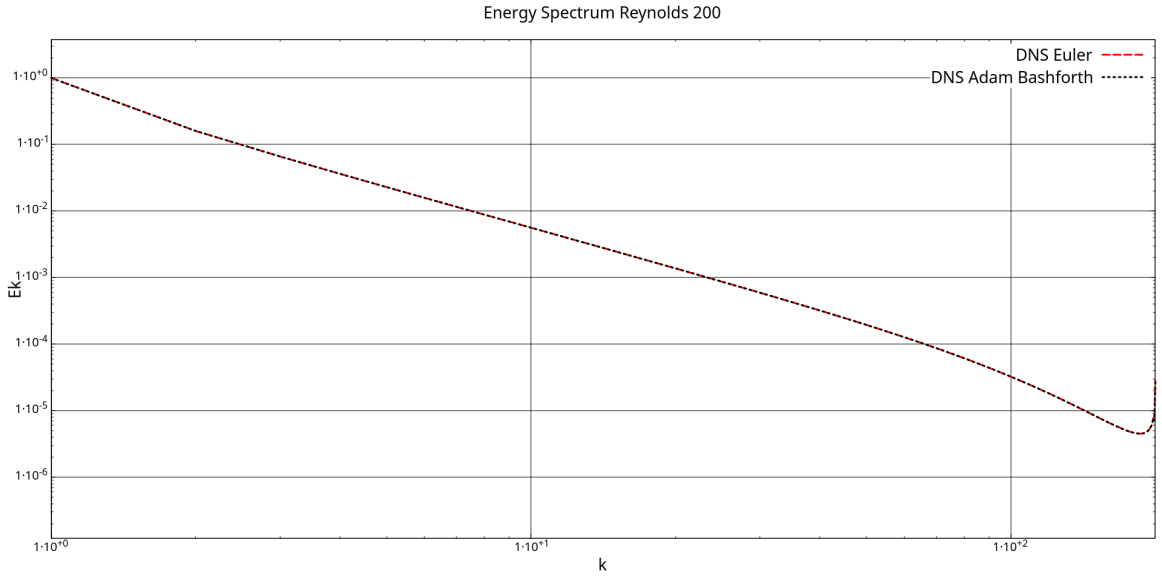

Figure 7: Comparison between Euler and Adam-Bashforth temporal integration schemes

As there can be seen in figure 7, the obtained results for both cases are virtually the same. It is true that, as it can be observed in table 4, the Courant constant (C_1) is higher for the second order scheme. It allows to increase this parameter without diverging, which can't be said for Euler. However, the difference in computational time doesn't represent a great leap in performance.

For the treated scales, the isn't a significant difference between using one or another. Therefore, this option would depend on which scheme is implemented in the code, not on its performance.

5.3 Number of modes influence study

Another important study that has been decided to develop concerns the number of modes for simulation. To do that, there has been set Reynolds to 50 in several different number of modes cases. In this case, the simulations carried out for this study have all the same parameters except the number of modes of course.

THe type of the simulation is DNS, Reynolds number is 50, as it has been mentioned, the integration scheme is a first order Euler, and the Courant constant has been set to 0.02 in order to avoid divergence at low number of modes.

The results obtained are shown in figure 8.

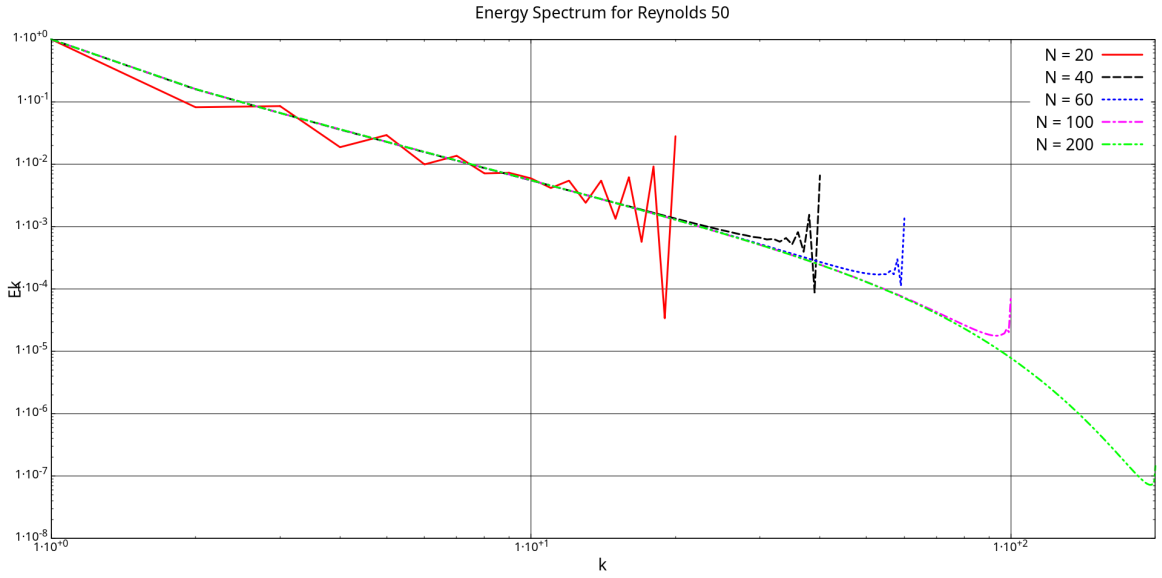

Figure 8: Number of modes study obtained results

As there can be seen in plot 8, there is a clear improvement in the accuracy of the results. For $N = 20$, the solution is close to diverge as it begins to oscillate severely when reaching higher frequency modes. On the other hand, for $N = 100$, the simulation is able to reach very small energy scales without major oscillation or divergence threat. Nevertheless, the computational time increases much faster than the accuracy of the results. It is possible to used higher order integration schemes. But, as it has been shown, there isn't much performance increase with those.

Instead of that, the most common solution is to carry out a LES simulation, in front of expensive DNS.

5.4 DNS vs LES

Following with the conclusions extracted from the results of the previous study, a comparison between DNS and LES results have been made.

In order to compare the performance of each option, there has been performed two different simulations with the following characteristics (table 5).

Table 5: DNS and LES comparison simulation parameters

Type	**Reynolds Number**	**Number of modes**	C_1
DNS	500.0	200	0.005
LES	500.0	200	0.01

The results obtained are shown in figure 9.

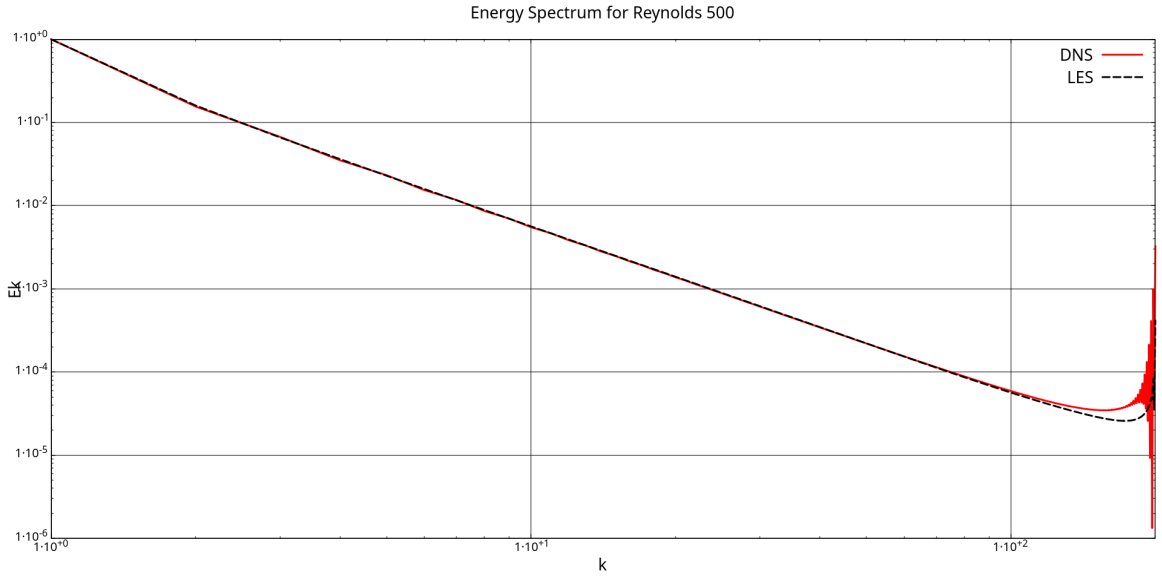

Figure 9: Comparison between DNS and LES performance results obtained

Using "Chrono" C++ library, it is possible to measure the time between two lines of code. For this case, the times calculated are the following (table 6).

Table 6: Simulation times masured

Type	**Time** [s]
DNS	29.706
LES	8.501

There are several aspects that has to be mentioned about previous results.

First, looking at table 5, it is possible to see that, for the same Reynolds number and equal amount of modes, LES allows to carry out the simulation with a Courant constant much higher (double) than DNS. This represents a significant leap in computational time towards the steady solution.

In addition to that, there can be observed in table 6 the absurd difference in computational time required. The results obtained show that, for the mentioned simulation parameters, the time needed for DNS almost quadruples LES. This represents a time difference which makes LES numerical methods much attractive despite its superior theoretical and implementation difficulty.

Finally, in what comes to the results accuracy, there can be seen in figure 9 that there is virtually no difference at all. During most of the plot, both results overlap each other. It isn't until the smallest scales that both begin to separate from each other. Concerned to this high frequency region, there must be said that it looks like DNS solution is close to diverge as k increases. It also can be observed this doesn't happen to LES plot, another advantage of its numerical implementation.

6 Conclusions

There are several conclusions that can be extracted from the report.

First of them is the fulfilment of its scope. The objective was to understand the meaning of Burgers equation and the physical phenomena of the energy and motion scales of fluids. In addition to the numerical implementation and resolution alongside some interesting studies involving the parameters of simulation and their influence on the global result.

Another important conclusion that can be ex-

tracted is the enormous advantage some numerical methods such as LES or RANS have over classical DNS. As it has been shown, there is an enormous difference in computational time required just for solving a one-dimensional equation. Therefore, it is clear the superiority of these methods on full 3D Navier Stokes simulations.

Finally, there must be mentioned that, although Burgers equation may seem as far from a fluid simulation, it isn't as much as it could be thought. It involves many physical and numerical phenomena present in Navier Stokes and so, it is a great leap towards these type of advanced simulation.

References

[1] CTTC (UPC). "Burgers' equation in Fourier space". In: 4 (2014), pp. 1–8.
[2] CTTC (UPC). "Computational Engineering - Burgulence". In: (2019).

A Programming Codes used

```
#include <fstream>
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <cstdio>
#include <time.h>
#include <chrono>

using namespace std;

//Declaraci n de funciones a utilizar
double *AllocateDouble(int NX, int NY, int Dim);
int *AllocateInt(int NX, int NY, int Dim);
double Get_TimeStep(int N, double Re, double Courant);
void InitialConditions(int N, double* UpresRe, double* UpresIm, double* DiffusivePast, dou
double Get_TotalViscosity(string CFD, int N, int k, double Ck, double* UpresRe, double* Up
void Get_Diffusive(double* DiffusivePres, double Viscosity, int k, double* UpresRe, double
void Get_Convective(double* ConvectivePres, int N, int k, double* UpresRe, double* UpresIm
void Get_Velocity(int k, string Esquema, double* DiffusivePres, double* ConvectivePres, do
double Get_StopError(int N, double* UpresRe, double* UpresIm, double* UfutRe, double* Ufut
void Get_Update(int N, double* UpresRe, double* UpresIm, double* UfutRe, double* UfutIm, d
void Get_FinalResults(double Re, int N, double* UfutRe, double* UfutIm, string CFD, string
void Get_SlopeData(int N);

int main(){

     auto start = std::chrono::high_resolution_clock::now();

//Declaraci n de variables del problema

    //Variables f sicas
    double Re = 500.0; //N mero de Reynolds

    double Viscosity;

    //Variables num ricas
    int N = 200; //N mero de modos a calcular
    string Esquema = "Euler"; //Esquema de integraci n a usar (Euler/AdamBashforth)
    string CFD = "DNS"; //Tipo de simulaici n (DNS/LES) (Para la viscosidad)
    double Convergencia = 1e-6; //Convergencia simulaci n
    double Divergencia = 1e10; //Divergencia simulaci n
    double Ck = 0.4223; //Constante de Kolmogorov (LES)
    double Courant = 0.005; //N mero de Courant para el CFL

    //Declaraci n de Arrays
    double *UpresRe; //Step presente parte real
    double *UpresIm; //Step presente parte imaginaria

    double *UfutRe; //Step futuro parte real
    double *UfutIm; //Step futuro parte imaginaria

    double *DiffusivePres; //T rmino difusivo
    double *ConvectivePres; //T rmino convectivo
```

```
double *DiffusivePast; //Término difusivo
double *ConvectivePast; //Término convectivo

//Alojamiento de memoria de los Arrays
UpresRe = AllocateDouble(N, 1, 1); //Step presente parte real
UpresIm = AllocateDouble(N, 1, 1); //Step presente parte imaginaria

UfutRe = AllocateDouble(N, 1, 1); //Step futuro parte real
UfutIm = AllocateDouble(N, 1, 1); //Step futuro parte imaginaria

DiffusivePres = AllocateDouble(2, 1, 1); //Término difusivo
ConvectivePres = AllocateDouble(2, 1, 1); //Término convectivo

DiffusivePast = AllocateDouble(2, 1, 1); //Término difusivo
ConvectivePast = AllocateDouble(2, 1, 1); //Término convectivo

//Declaración de variables necesarias para el problema
double DeltaT = Get_TimeStep(N, Re, Courant);

//Set of the Initial Condition of the Velocity Arrays
InitialConditions(N, UpresRe, UpresIm, DiffusivePast, ConvectivePast);

//Variables para la simulación
int NITER = 0;
double Difference = 2.0*Convergencia;
int k;

//Simulation Process
while(Difference >= Convergencia && Difference <= Divergencia){

    //Velocidades del modo inicial (1)
    UpresRe[0] = 1.0;
    UpresIm[0] = 0.0;

    UfutRe[0] = 1.0;
    UfutIm[0] = 0.0;

    for(k = 1; k < N; k++){

        //Cálculo de la viscosidad
        Viscosity = Get_TotalViscosity(CFD, N, k, Ck, UpresRe, UpresIm, Re);

        //Cálculo del término difusivo
        Get_Diffusive(DiffusivePres, Viscosity, k, UpresRe, UpresIm);

        //Cálculo del término convectivo
        Get_Convective(ConvectivePres, N, k, UpresRe, UpresIm);

        //Cálculo de las velocidades
        Get_Velocity(k, Esquema, DiffusivePres, ConvectivePres, DiffusivePast, Convect

    }

    //Cálculo del error relativo entre Steps
    Difference = Get_StopError(N, UpresRe, UpresIm, UfutRe, UfutIm);
```

```
        //Update de los Arrays
        Get_Update(N, UpresRe, UpresIm, UfutRe, UfutIm, DiffusivePres, ConvectivePres, Dif

        NITER += 1;

        if(NITER%1000 == 0){
            cout<<"Iteraci n: "<<NITER<<", Error Relativo: "<<Difference<<endl;
        }

    }

    cout<<"Final de la simulaci n."<<endl;
    cout<<"Iteraciones Totales: "<<NITER<<endl;
    cout<<"Error Relativo Final: "<<Difference<<endl;

    Get_FinalResults(Re, N, UfutRe, UfutIm, CFD, Esquema);
    Get_SlopeData(N);

    // Record end time
        auto finish = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = finish - start;
        std::cout << "Elapsed time: " << elapsed.count() << " s\n";
}

//Memoria din mica matriz (double)
double *AllocateDouble(int NX, int NY, int Dim){
double *M1;

        M1 = new double [NX*NY*Dim];
        return M1;
}

//Memoria din mica matriz (int)
int *AllocateInt(int NX, int NY, int Dim){
int *M1;

        M1 = new int [NX*NY*Dim];
        return M1;
}

//C lculo del time Step
double Get_TimeStep(int N, double Re, double Courant){
double DeltaT;

    DeltaT = Courant*(Re/pow(N,2.0));

    return DeltaT;

}

//Set de las condiciones iniciales de las variables
void InitialConditions(int N, double* UpresRe, double* UpresIm, double* DiffusivePast, dou
int i;
double I;
```

```
    for(i = 0; i < N; i++){
        I = i;
        UpresRe[i] = 1.0/(I + 1.0);
        UpresIm[i] = 0.0;
    }

    DiffusivePast[0] = 0.0;
    DiffusivePast[1] = 0.0;

    ConvectivePast[0] = 0.0;
    ConvectivePast[1] = 0.0;

}

//C lculo de la viscosidad
double Get_TotalViscosity(string CFD, int N, int k, double Ck, double* UpresRe, double* Up
double K = k;
double m = 2.0;
double Ekn;
double Viscosity = 1.0/Reynolds;

    if(CFD == "LES"){
        Ekn = pow(UpresRe[N-1],2.0) + pow(UpresIm[N-1],2.0);
        Viscosity += (0.31*((5.0 - m)/(m + 1.0))*sqrt(3.0 - m)*pow(Ck, -1.5))*(pow(Ekn/N,
    }

    return Viscosity;
}

//C lculo del t rmino difusivo de la ecuaci n de Momentum
void Get_Diffusive(double* DiffusivePres, double Viscosity, int k, double* UpresRe, double

        DiffusivePres[0] = Viscosity*pow(k + 1,2.0)*UpresRe[k]; //Parte Real Difusivo
        DiffusivePres[1] = Viscosity*pow(k + 1,2.0)*UpresIm[k]; //Parte Imaginaria Difusiv

}

//C lculo del t rmino convectivo de la ecuaci n de Momentum
void Get_Convective(double* ConvectivePres, int N, int k, double* UpresRe, double* UpresIm
int p, q;
int ip, iq;

    ConvectivePres[0] = 0.0;
    ConvectivePres[1] = 0.0;

        for (int p = - N; p <= N; p++) {
                q = k - p + 1;

                ip = abs(p) - 1;
                iq = abs(q) - 1;

                if (q >= - N && q <= N) {
                        if (q == 0 || p == 0) {

                ConvectivePres[0] += 0.0;
                ConvectivePres[1] += 0.0;
```

```
            }
                        else if (q < 0) {

                ConvectivePres[0] += + q*(+ UpresRe[ip]*UpresIm[iq] - UpresRe[iq]*UpresIm[
                                ConvectivePres[1] += + q*(UpresRe[ip]*UpresRe[iq] + UpresIm

                        }
                        else if (p < 0) {

                ConvectivePres[0] += + q*(- UpresRe[ip]*UpresIm[iq] + UpresRe[iq]*UpresIm[
                                ConvectivePres[1] += + q*(UpresRe[ip]*UpresRe[iq] + UpresIm

                        }
                        else {

                                ConvectivePres[0] += + q*(- UpresRe[ip]*UpresIm[iq] - Upres
                                ConvectivePres[1] += + q*(UpresRe[ip]*UpresRe[iq] - UpresIm

                        }
                }
        }

}

void Get_Velocity(int k, string Esquema, double* DiffusivePres, double* ConvectivePres, do

    if(Esquema == "Euler"){
        UfutRe[k] = UpresRe[k] + DeltaT*(- ConvectivePres[0] - DiffusivePres[0]);
        UfutIm[k] = UpresIm[k] + DeltaT*(- ConvectivePres[1] - DiffusivePres[1]);
    }
    else if(Esquema == "AdamBashforth"){
        UfutRe[k] = UpresRe[k] + DeltaT*(1.50*(- ConvectivePres[0] - DiffusivePres[0]) - 0
        UfutIm[k] = UpresIm[k] + DeltaT*(1.50*(- ConvectivePres[1] - DiffusivePres[1]) - 0
    }

}

double Get_StopError(int N, double* UpresRe, double* UpresIm, double* UfutRe, double* UfutI
int k;
double Upres, Ufut;
double MaxDiff = 0.0;

    for(k = 0; k < N; k++){

        Upres = sqrt(pow(UpresRe[k],2.0) + pow(UpresIm[k],2.0));
        Ufut = sqrt(pow(UfutRe[k],2.0) + pow(UfutIm[k],2.0));

        if(abs((Ufut - Upres)/(Upres + 1e-10)) >= MaxDiff){
            MaxDiff = abs((Ufut - Upres)/(Upres + 1e-10));
        }
    }

    return MaxDiff;

}
```

```
void Get_Update(int N, double* UpresRe, double* UpresIm, double* UfutRe, double* UfutIm, d
int k;

    for(k = 0; k < N; k++){
        UpresRe[k] = UfutRe[k];
        UpresIm[k] = UfutIm[k];

        DiffusivePast[0] = DiffusivePres[0];
        DiffusivePast[1] = DiffusivePres[1];

        ConvectivePast[0] = ConvectivePres[0];
        ConvectivePast[1] = ConvectivePres[1];
    }

}

void Get_FinalResults(double Re, int N, double* UfutRe, double* UfutIm, string CFD, string
int k;
double Ek;
char Directorio[500];
int RE = Re;

    sprintf(Directorio,"/home/sergiogus/Desktop/ComputationalEngineering/BurgersEquation/R

    FILE *fp1;
        fp1 = fopen(Directorio,"w");

        for(k = 0; k < N; k++){
            Ek = pow(UfutRe[k],2.0) + pow(UfutIm[k],2.0);

            fprintf(fp1,"%d \t %.12f \n", k + 1, Ek);
        }

        fclose(fp1);

}

void Get_SlopeData(int N){
int k;
double Exponente = -2.0;
int Exp = Exponente;
double K;
char Directorio2[500];

     sprintf(Directorio2,"/home/sergiogus/Desktop/ComputationalEngineering/BurgersEquation

    FILE *fp2;
        fp2 = fopen(Directorio2,"w");

        for(k = 0; k < N; k++){
            K = k;
            fprintf(fp2,"%d \t %f \n", k + 1, pow(K, Exponente));
        }

        fclose(fp2);
```

```
}
```