POLITECNICO DI MILANO

Facoltà di Ingegneria Industriale

Corso di laurea in Ingegneria Aeronautica

Linear stability of 2D incompressible flows using Trilinos

Relatore: Prof. Franco Auteri

Autore: Andrea Penza 735235

Anno accademico 2010-2011

"The only way to do a great work is to love what you do.
If you haven't found it yet, keep looking, don't settle"
S. Jobs

"Never say never,
because limits, like fears, are often just an illusion"
M. Jordan

Contents

List of Figures

List of Tables

Abstract

This work focuses on the numerical investigation of the linear stability of two-dimensional incompressible flows. For this purpose, an innovative software based on the object-oriented C++ Trilinos library has been developed. The finite element solver has been built exploiting the Sundance package of Trilinos, which showed an excellent flexibility and a high level of abstraction leading to a rapid development of the Navier-Stokes simulator. Good results have been obtained for the well-known case of lid-driven cavity problem, which has been employed to validate the solver of the base flow. Good results have been obtained also for the linear stability problem of the flow around a circular cylinder, which has been used to validate the linear stability solver. About this last test case part of the eigenvalue spectrum and the eigenmodes of the most unstable eigenvalue are presented as final results of the stability analysis.
Particular attention has been dedicated to the description of the configuration of the software and to its installation, a quite involved task, while the whole code produced and all modifications introduced in the existing Sundance classes are completely listed.

Sommario

In questo lavoro ci si è prefissi l'obiettivo di realizzare un software innovativo per lo studio della stabilità delle correnti fluidodinamiche. A questo scopo è stato realizzato un codice basato sul linguaggio di programmazione $C++$, provato poi su due casi di validazione. L'obiettivo finale è stato quello di realizzare uno strumento per poter studiare le caratteristiche di stabilità degli oscillatori fluidici che sono attualmente oggetto di ricerca. Gli oscillatori fluidici sono strumenti che utilizzano un fluido per ottenere operazione logiche di tipo analagico o digitale simili a quelle utilizzate in campo elettronico. L'obiettivo di tale ricerca è migliorare la comprensione del funzionamento degli oscillatori fluidici utili come alternativa a componenti elettronici maggiormente soggetti alla sollecitazione termica e all'usura. In figura 1 è riportato un esempio di oscillatore fluidico di geometria molto semplice.

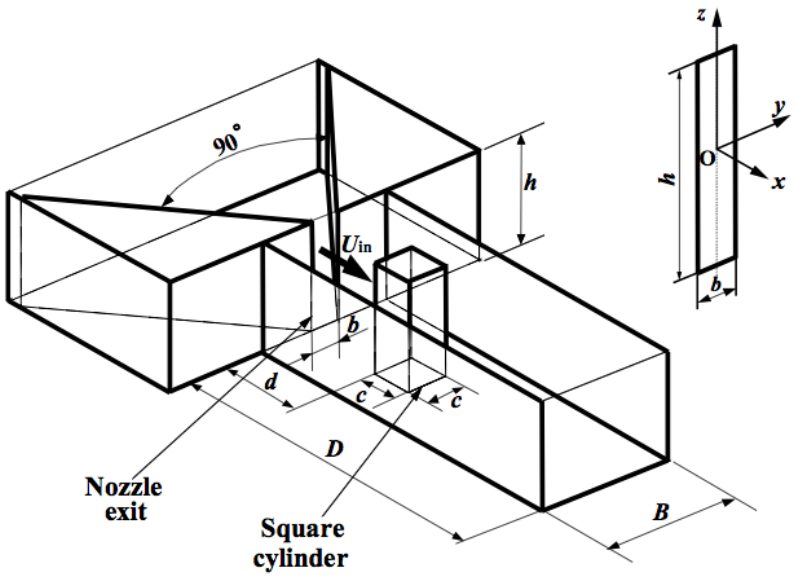


Figure 1: Esempio di osciallatore fluidico di geometria semplice (figura tratta da [8]

In molti casi la documentazione si è rivelata inadeguata o assente. Questo lavoro

si propone anche di colmare alcune delle numerose lacune nella documentazione, soprattutto per quanto riguarda la parte di installazione, documentazione che qui è stata fornita dettagliatamente con indicazioni passo-passo sul da farsi.
Il primo passo da fare è studiare la stabilità in campo lineare delle correnti incomprimibili bidimensionali, il che comporta risolvere due problemi. Il primo è il calcolo del flusso base relativo al problema stazionario, mentre il secondo è calcolare gli autovalori del problema non stazionario linearizzato partendo dalla condizione stazionaria precedentemente calcolata.
La formulazione fisica del problema parte delle equazioni di Navier-Stokes per un flusso incomprimibile

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\, \boldsymbol{u} - \dfrac{1}{Re}\nabla^2 \boldsymbol{u} + \boldsymbol{\nabla} p = \boldsymbol{g} \\ \\ \boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \end{cases} \tag{1}$$

e dalla loro linearizzazione nell'ambito della teoria delle piccole perturbazioni che prevede una scomposizione della soluzione in una parte stazionaria $(\boldsymbol{U}_0, P_0)$, il cosiddetto flusso base, sommata a una piccola pertubazione $(\boldsymbol{u}_\delta, p_\delta)$.

$$\boldsymbol{u} = \boldsymbol{U}_0 + \boldsymbol{u}_\delta$$
$$p = P_0 + p_\delta$$

Si ottengono così le equazioni linearizzate

$$\begin{cases} \dfrac{\partial \boldsymbol{u}_\delta}{\partial t} + ((\boldsymbol{U}_0 \cdot \boldsymbol{\nabla})\, \boldsymbol{u}_\delta + (\boldsymbol{u}_\delta \cdot \boldsymbol{\nabla})\, \boldsymbol{U}_0) - \dfrac{1}{Re}\nabla^2 \boldsymbol{u}_\delta + \boldsymbol{\nabla} p_\delta = 0 \\ \\ \boldsymbol{\nabla} \cdot \boldsymbol{u}_\delta = 0 \end{cases} \tag{2}$$

che discretizzate danno luogo al problema algebrico agli autovalori

$$\lambda \boldsymbol{M} \boldsymbol{v} = \boldsymbol{A} \boldsymbol{v} \tag{3}$$

che può essere riformulato nel modo seguente per accelerare la convergenza del processo risolutivo:

$$(\boldsymbol{A} - \sigma \boldsymbol{M})^{-1}\, \boldsymbol{M} \boldsymbol{v} = \nu \boldsymbol{v} \tag{4}$$

Il software realizzato si è dimostrato estremamente compatto e flessibile in fase di impostazione del problema da studiare. A ciò ha contribuito senza dubbio l'eccellente qualità di programmazione della libreria su cui il software si basa, ovvero Trilinos. Trilinos è una libreria di pacchetti per lo sviluppo di software che

usano un linguaggio di programmazione a oggetti, il *C++*. In particolare è stato utilizzato il pacchetto Sundance, realizzato inizialmento presso i *Sandia National Laboratories* e sviluppato ora dal Dipartimento di Matematica presso la *Texas Tech University*.
Il solutore del problema agli autovalori è stato implementato usando il pacchetto Sundance distribuito all'interno della libreria Trilinos. Sundance è un sistema per il rapido sviluppo di solutori per problemi differenziali alle derivate parziali basato sul metodo degli elementi finiti. Il codice prodotto, riportato interamente in appendice A, ha la struttura tipica di un programma per la risoluzione di un problema differenziale con il metodo degli elementi finiti. La griglia di calcolo non strutturata composta da soli triangoli è stata realizzata tramite il software commerciale Ansys IcemCFD e una volta caricata nel programma la griglia vengono costruiti gli elementi finiti. È possibile scegliere diversi tipi di elementi e in questo lavoro si è optato per elementi lagrangiani di Taylor-Hood, quadratici per la velocità e lineari per la pressione. Successivamente, utilizzando le classi disponibili in Sundance, viene formulata molto velocemente e con estrema semplicità la forma debole delle equazioni. La soluzione stazionaria è stata calcolata tramite il metodo di Newton, implementato nel pacchetto di solutori non lineari di Trilinos chiamato NOX. Per l'analisi di stabilità invece è stato necessario risolvere il problema agli autovalori associato alla formulazione (2) tramite un interfaccia al pacchetto Anasazi. Si è optato per una trasformazione spettrale di tipo *shift-invert* (4) per aumentare l'efficienza del metodo risolutivo. Questo tipo di trasformazione aumenta la velocità di convergenza calcolando per primi gli autovalori di modulo minore invece che quelli di modulo maggiore. Per tale trasformazione è stato necessario modificare la struttura di alcune classi di Sundance in quanto essa non era stata inizialmente prevista degli sviluppatori.
Il programma sviluppato è stato applicato a flussi già studiati e noti in letteratura in modo da validare il codice realizzato. Ottimi risultati sono stati ottenuti per il problema della cavità, utilizzato nella fase iniziale dello sviluppo del codice come caso prova per validare il solutore di Navier-Stokes stazionario. La soluzione calcolata è risultata in accordo con i dati numerici riportati da Botella & Peyret (1998) [6] e il confronto è riportato nelle tabelle 1 e 2. I valori riportati sono stati calcolati su 17 punti lungo la linea di mezzeria orizzontale e verticale.
Di maggiore interesse ai fini dell'obiettivo finale è però il caso della corrente bidimensionale che investe un cilindro a sezione circolare. In questo caso è stata fatta un' analisi di stabilità della corrente. Partendo dalla soluzione stazionaria del flusso base visibile in figura 2 si è passati al calcolo degli autovalori del problema linearizzato. È stata calcolata parte dello spettro di autovalori, riportata in figura 3 in cui si può notare come il numero di Reynolds influenzi la stabilità della corrente. Infatti i risultati hanno dimostrato che la corrente si mantiene stabile per un numero di Reynolds inferiore a 46.5, che rappresenta pertanto il numero di Reynolds critico per la stabilità della corrente attorno al cilindro. L'autovalore a parte reale più grande è quello più instabile, a esso è associato il modo instabile presentato in figura 4. L'accuratezza del calcolo degli autovalori è stata verificata

y	u_x, Ref. [6]	u_x
1.0000	-1.000	-1.000
0.9766	-0.664	-0.664
0.9688	-0.581	-0.581
0.9609	-0.517	-0.506
0.9531	-0.472	-0.469
0.8516	-0.337	-0.337
0.7344	-0.189	-0.194
0.6172	-0.057	-0.057
0.5000	0.062	0.062
0.4531	0.108	0.108
0.2813	0.280	0.280
0.1719	0.388	0.387
0.1016	0.300	0.300
0.0703	0.223	0.223
0.0625	0.202	0.204
0.0547	0.181	0.181
0.0000	0.000	0.000

Table 1: Componente orizzontale di velocità lungo la mezzeria verticale, Re = 1000

riferendosi a quanto già presentato da Giannetti & Luchini (2007) [7] nonchè da Sipp & Lebedev (2007) [14].

x	u_y, Ref. [6]	u_y
0.0000	0.000	0.000
0.0312	-0.228	-0.233
0.0391	-0.294	-0.291
0.0469	-0.355	-0.357
0.0547	-0.410	-0.410
0.0947	-0.526	-0.527
0.1406	-0.426	-0.427
0.1953	-0.320	-0.311
0.5000	0.025	0.025
0.7656	0.325	0.325
0.7734	0.334	0.334
0.8437	0.377	0.377
0.9062	0.333	0.326
0.9219	0.310	0.310
0.9297	0.296	0.289
0.9375	0.281	0.281
1.0000	0.000	0.000

Table 2: Componente verticale di velocità lungo la mezzeria orizzontale, Re = 1000

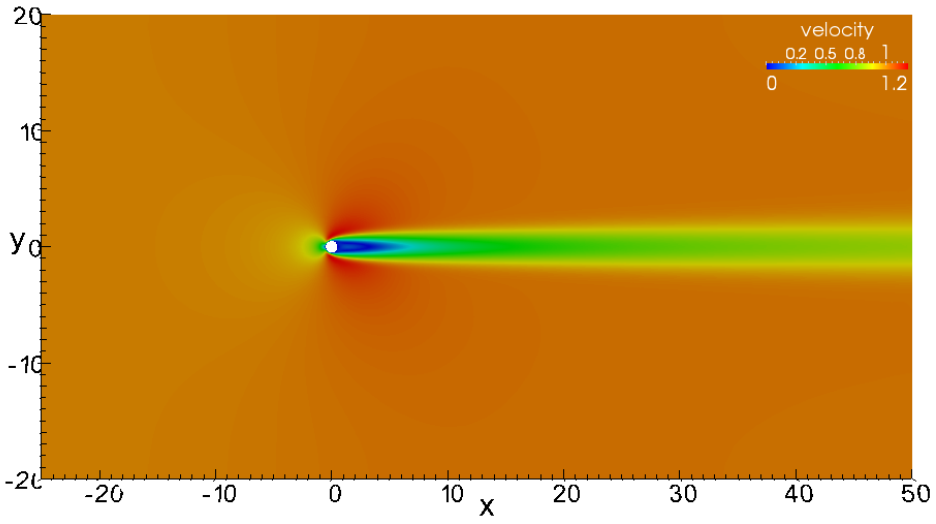


Figure 2: Corrente attorno al cilindro: modulo della velocità del flusso base ($Re = 46.5$)

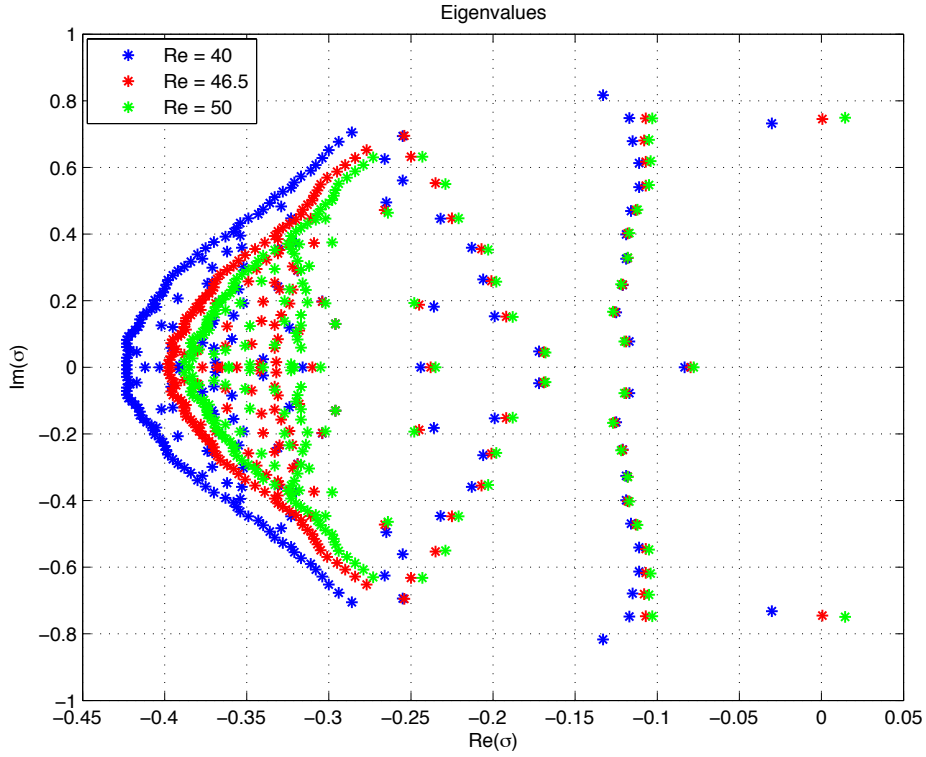
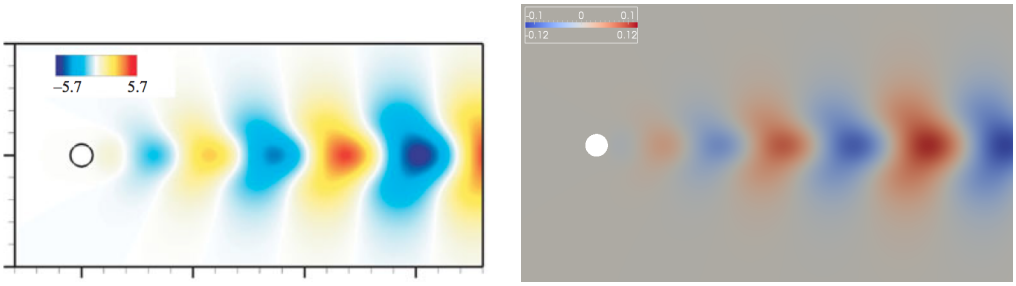


Figure 3: Corrente attorno al cilindro: autovalori del problema linearizzato



(a) Sipp & Lebedev (2007), [14] (b) Risultati attuali

Figure 4: Corrente attorno al cilindro: primo modo instabile relativo alla componente verticale della velocità

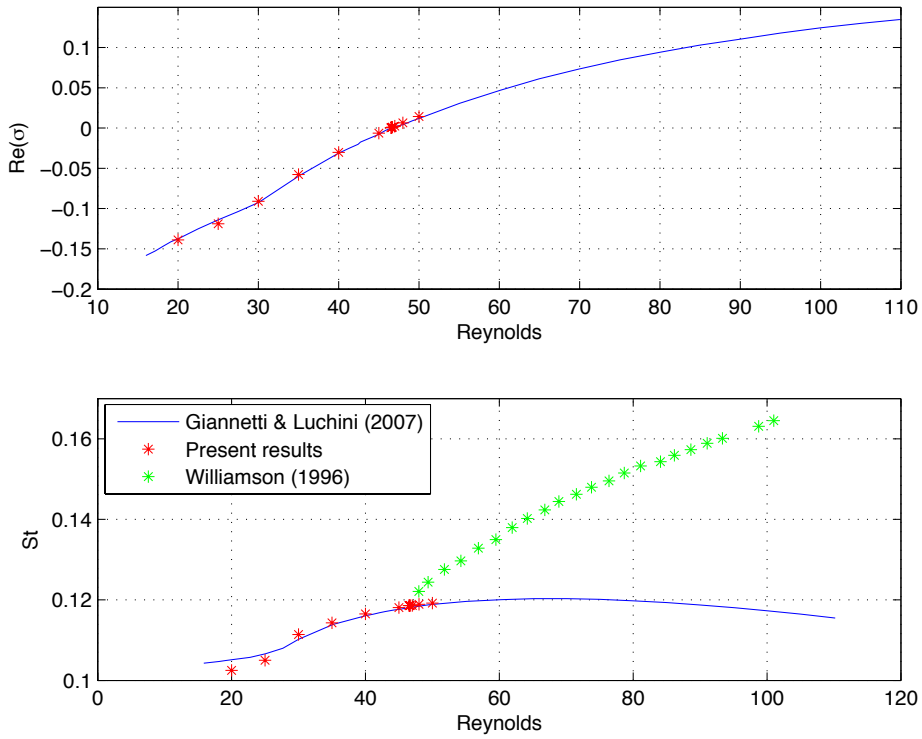


Figure 5: Primo autovalore instabile: dipendenza dal numero di Reynolds

La figura 5 è particolarmente importante. Presenta i risultati relativi al primo autovalore instabile in termini di parte reale e numero di Strouhal al variare del numero di Reynolds. Si può riscontrare un buon accordo coi dati pubblicati in [7], in particolare per la curva $Re - St$. La linearità dell'approssimazione vale per piccole perturbazioni ed è per questo valida solo in prossimità del numero di

Reynolds critico che, come detto, è stato individuato in 46.5. Allontanandosi da tale regione gli effetti non lineari prevalgono e la soluzione si allontana da quella sperimentale proposta da Williamson (1996) [18].
Il software sviluppato ha prodotto dei buoni risultati, e la sua validazione può ritenersi conclusa. Tuttavia ha mostrato alcune criticità che dovranno essere oggetto di futuri miglioramenti. Prima di tutto è cruciale la possibilità di poter usare il codice in parallelo, in modo da poter distribuire la memoria impiegata ed effettuare calcoli su mesh più fini. In questo lavoro sono già riportate tutte le informazioni necessarie e il codice da utilizzare per partizionare la mesh, ma occorrerà senza dubbio cambiare il solutore. In quest'ottica i solutori diretti paralleli MUMPS e SuperLU Dist dovrebbero fornire le prestazioni desiderate in termini di calcolo parallelo, e c'è la possibilità di interfacciarli con Sundance tramite il pacchetto Amesos. Questo permetterebbe oltre che di poter lavorare su mesh composte da un numero maggiore di elementi anche di ridurre sensibilmente i tempi di calcolo, che al momento sono risultati eccessivi per il tipo di problema che si è risolto. In quest'ottica l'ottimizzazione nella scelta del solutore è una questione cruciale e va affrontata come priorità. Per quanto riguarda la scelta del solutore una ulteriore possibile strada è quella dei solutori iterativi contenuti nel pacchetto AztecOO di Trilinos. In questo caso la scelta del precondizionatore adeguato per ottenere la convergenza del metodo risolutivo non è banale e la ricerca di un precondizionatore robusto e affidabile nonché efficiente è ancora in corso. In letteratura sono presenti diversi articoli che trattano questo argomento.
Come detto, la trasformazione *shift-invert* utilizzata per rendere più efficiente il calcolo degli autovalori è stata introdotta in Sundance durante questo lavoro. Per il momento è stato possibile implementare solamente uno shift reale. Si potrebbero limitare i tempi di calcolo rendendo possibile l'utilizzo di un shift complesso, che permetterebbe così di calcolare lo spettro degli autovalori su una regione più ristretta nell'intorno dell'autovalore di interesse (tipicamente quello più instabile) senza dover calcolare una regione ampia come è stato fatto in questo lavoro poichè l'autovalore instabile era abbastanza lontano dall'asse reale.
Un altro possibile miglioramento al lavoro proposto è sicuramente quello di rendere la procedura di realizzazione della griglia di calcolo indipendente dal software commerciale Cubit, che è stato utilizzato solamente per assegnare dei numeri identificativi ai lati del dominio, in modo da poter applicare le condizioni al contorno. In questo senso potrebbe essere necessario modificare una o alcune classi di Sundance in modo da interfacciarlo con un numero maggiore di formati del file di mesh oltre al formato ExodusII utilizzato per questo lavoro.
Infine lo scopo ultimo del lavoro, come anticipato, è quello di studiare la stabilità lineare di correnti in geometrie complicate, come ad esempio le correnti all'interno di oscillatori fluidici. Il prossimo passo previsto sarà proprio quello di utilizzare il codice sviluppato in questo lavoro per studiare la prima instabilità della corrente all'interno dell'oscillatore fluidico proposto. A puro titolo di esempio la figura 6 riporta la mesh relativa ad un geometria molto semplice, simile a quella della figura 1, e utilizzata nel lavoro di Hirata (2009) [8].

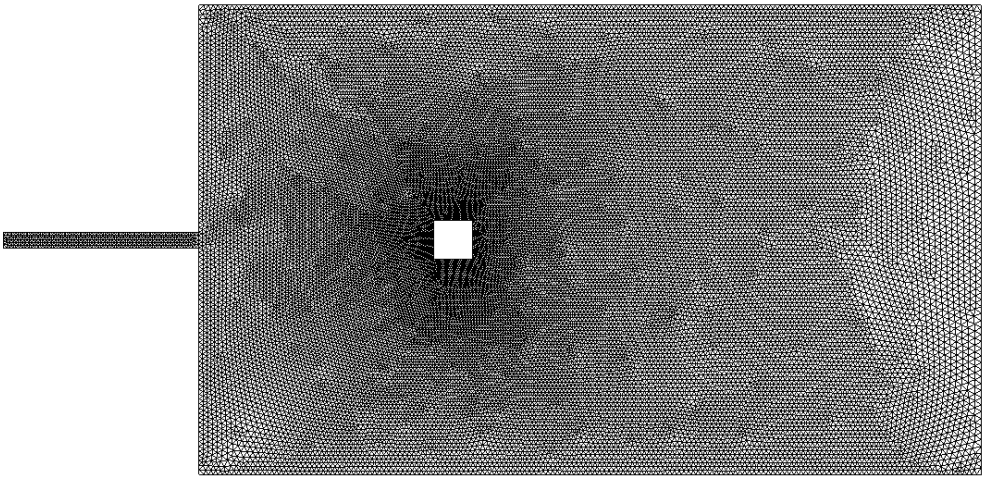


Figure 6: Mesh di calcolo per un semplice oscillatore fluidico

Introduction

A confined jet sometimes causes a self-excited oscillation characterized by a well-defined frequency due to the existence of a downstream target. This phenomenon has been investigated in recent years in order to study some fluidic oscillators, flip-flop jet nozzles, heat/material mixers, flow controls and flowmeters, in order to build high-reliability devices without mechanically moving components. *Fluidics* addresses the use of a fluid to perform analog or digital operations similar to those performed with electronics. The term *fluidics* is normally used when devices have no moving parts. A jet of fluid can be deflected by a weaker jet striking it at the side. This provides nonlinear amplification, similar to what happens in a transistor used in electronic digital logic. Fluidic devices are used mostly in environments where electronic digital logic would be unreliable, or where electronic components are subject to overheating. Some examples of different geometries of fluidic devices are shown in figure 7. The shape of a simple fluidic oscillator is shown in figure 8 and it refers to the geometry described by Hirata, Matoba, Naruse, Haneda and Funaky (2009) [8], who carried out velocity measurements by an UVP (ultrasonic velocity profiler) and by PIV (particle image velocimetry).

The characterization of a simple fluidic oscillator may be performed by the same approach employed in the study of bluff-body wakes, and a numerical modal analysis can be used to determine the properties of the instability and to find its critical Reynolds number. One of the most common examples of a bluff-body flow is given by the flow around an infinitely long circular cylinder, for this reason such a flow has been employed here as a test case. In this flow, the transition from the steady to the unsteady state breaks the symmetry of the flow field and the Von Karman vortex street is generated. Experimental results on the 2D cylinder wake were reported by Williamson (1996) [18], who suggested the threshold of 47 for the critical Reynolds where the transition from the steady to the unsteady state of the cylinder wake occurs.

Several approaches can be used to deal with this problem. In recent years, direct numerical simulations have been used to study the development of the Von Karman vortex street and to identify the onset of the instability. The linear theory provides an alternative way to the solution of the full nonlinear problem in order to reduce the computational cost. In this approach the stability of the flow is investigated by solving the two-dimensional generalized eigenvalue problem derived from the discretization of the linearized Navier-Stokes equations.

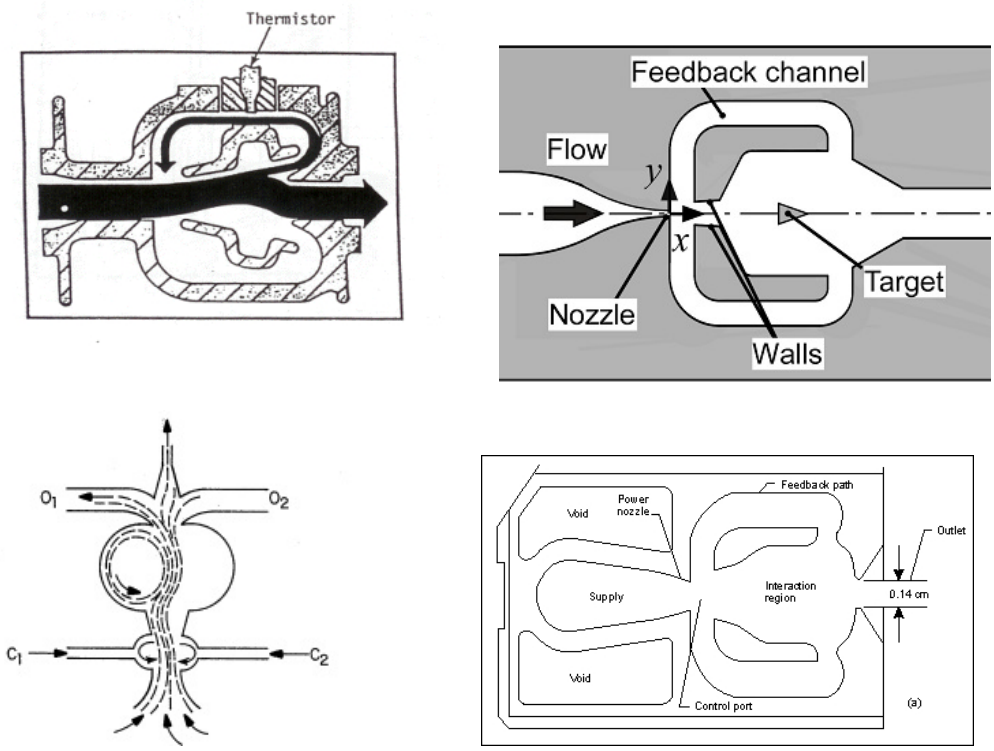


Figure 7: Examples of different fluidic oscillators

Numerical benchmarks have been provided in recent years. Sipp & Lebedev (2007) [14] published an investigation on the case of the 2D cylinder wake, using Taylor-Hood (P2, P2, P1) finite elements for the spatial discretization of the primitive variable (u_x, u_y, p) equations. In this work, the eigenvalue problem is solved thanks to an Arnoldi method based on a shift-invert strategy, available in the ARPACK library. Moreover, Giannetti & Luchini (2006) [7] used a finite volume discretization and employed an immersed-boundary technique to represent the cylinder surface on a Cartesian mesh. In this work, the eigenvalue problem is solved by a variant of the classical inverse-iteration algorithm, simultaneously applied to both the direct and the adjoint problems in order to procede in a coupled way. Only a LU decomposition for each step was required. Lanzerstorfer & Kuhlmann (2012) [9] employed a finite volume discretization applied to a backward-facing step on a staggered rectangular grid. Contrary to Giannetti & Luchini (2006) [7] they did not use the immersed boundary technique. In this work, an implicitly restarted Arnoldi algorithm provided in the ARPACK software library and the MATLAB `eig` command were employed. A shift-invert transformation with zero shift was used. Barkley, Blackburn & Sherwin (2008) [4] employed a spectral element method applied to the backward-facing step problem using the linear theory to investigate the stabil-

ity of the flow. In this work, the eigenvalue problem is solved by a Krylov method implemented by the authors using the LAPACK library. Experimental results reported by Williamson (1996) [18] substantially confirm the threshold of the critical Reynolds number suggested by the quoted numerical investigations.
The numerical approach employed in this work is based on a finite element discretization of the Navier-Stokes equations implemented using the Trilinos library, and an innovative software has been produced. The first problem to deal with was the correct configuration and installation of the Trilinos package Sundance, this package was employed to develop the code of the Navier-Stokes simulator.
In chapter 1 the problem formulation is introduced starting from the differential equations up to the weak formulation, the approach employed to compute the base flow and the discretization of the problem by the to finite element method is also presented.
In chapter 2 the framework of the Trilinos library and the main features of the Sundance package are presented. Furthermore, detailed instructions to install Sundance and all necessary third-party softwares on Linux Ubuntu 11.04 operative system, on which this work was carried out and all components tested, are given.
In chapter 3, the software structure and the whole code developed to implement the Navier-Stokes simulator are presented.
Chapter 4 is devoted to the validation of the code. Results obtained by the steady solver for the lid-driven cavity and flow the circular cylinder in crossflow are compared to well established benchmark [6], [3],[7] and [14]. The eigenvalues and eigenfunction computed by means of the linear stability solver are also reported and validated with respect to accurate results available in the recent literature [7] and [14]. The last chapter is devoted to some concluding remarks and to envisage future developments.
In the appendix, in which all codes used to accomplish the work are reported. Of particular interest are the modified lines implementing the shift-invert transformation in the Sundance package.

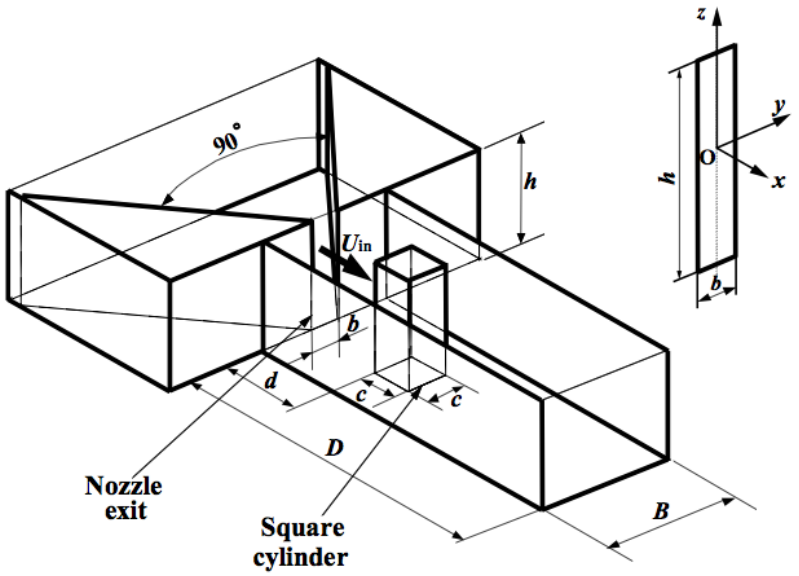


Figure 8: Geometry of the fluidic oscillator investigated in [8]

Chapter 1

Problem formulation

In order to investigate the linear stability of two-dimensional incompressible flows the first step is to obtain the mathematical formulation of the problem. This section deals with this task. First, the differential form of the Navier-Stokes equations is presented, the description of how the base flow is computed and how the linear stability is treated is given. Finally a brief introduction to the finite element method, which is employed to discretize the equations, is provided. The stability analysis requires the solution of the nonlinear steady Navier-Stokes equations first, to compute the base flow, and then the solution of the generalized eigenvalue problem obtained from the linerization of the evolution equations in the neighbourhood of the base flow. The nonlinear system of equations has been solved by the Newton method, which requires solving a linear system per iteration.

The differential form of the Navier-Stokes equations in their incompressible and dimensionless form (1.1) is provided below.

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\, \boldsymbol{u} - \dfrac{1}{Re} \nabla^2 \boldsymbol{u} + \boldsymbol{\nabla} p = \boldsymbol{g} \\ \\ \boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \end{cases} \tag{1.1}$$

Here $\boldsymbol{u}$ is the velocity vector, with components (u_x, u_y), and p is the pressure. Equations (1.1) are made dimensionless through the *Reynolds number*, R, which is based on the cylinder diameter D as the characteristic length scale, the uniform velocity U_∞ of the undisturbed stream as the reference velocity and the kinematic viscosity ν.

$$R = \frac{U_\infty D}{\nu}$$

1.1 The differential problem

We start considering the steady-state flows solution related to the nonlinear differential system (1.1), in this way the governing equations result as follows:

$$\begin{cases} (\boldsymbol{u}\cdot\boldsymbol{\nabla})\,\boldsymbol{u} - \dfrac{1}{Re}\nabla^2\boldsymbol{u} + \boldsymbol{\nabla}p = 0 \\ \boldsymbol{\nabla}\cdot\boldsymbol{u} = 0 \end{cases} \tag{1.2}$$

The force field on right-hand side is neglected in this discussion. Moreover, boundary conditions are required on the edges of the computational domain. Tipical conditions are the no-slip conditions expressed by (1.3a), the conditions of symmetry (1.3c) which can also be employed on egdes in the far field which are approximately parallel to the flow velocity. The velocity vector is assigned on the inlet side (1.3b), while the pressure distribuition and the normal derivative of the normal component of the velocity is assigned on the outlet side (1.3d).

$$u_x = 0, \quad u_y = 0 \tag{1.3a}$$
$$u_x = 1, \quad u_y = 0 \tag{1.3b}$$
$$\frac{\partial u_x}{\partial y} = 0, \quad u_y = 0 \tag{1.3c}$$
$$\frac{\partial u_x}{\partial x} = 0, \quad p = 0 \tag{1.3d}$$

The weak form is derived directly from the strong form (1.2) by multiplying each scalar equation (1.4) by the relative test function and integrating by parts.

$$\begin{cases} \dfrac{\partial u_x}{\partial x} + \dfrac{\partial u_y}{\partial y} = 0 \\ u_x\dfrac{\partial u_x}{\partial x} + u_y\dfrac{\partial u_x}{\partial y} = -\dfrac{\partial p}{\partial x} + \dfrac{1}{Re}\nabla^2 u_x \\ u_x\dfrac{\partial u_y}{\partial x} + u_y\dfrac{\partial u_y}{\partial y} = -\dfrac{\partial p}{\partial y} + \dfrac{1}{Re}\nabla^2 u_y, \end{cases} \tag{1.4}$$

The weak form is obtained:

$$\begin{cases} \displaystyle\int_\Omega \left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y}\right) q = 0 \\ \displaystyle -\int_{\partial\Omega}\frac{1}{Re}\left(\frac{\partial u_x}{\partial n}\right)v_x + \int_\Omega \frac{1}{Re}\boldsymbol{\nabla}v_x\cdot\boldsymbol{\nabla}u_x + \int_\Omega\left(u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y}\right)v_x + \int_\Omega\frac{\partial p}{\partial x}v_x = 0 \\ \displaystyle -\int_{\partial\Omega}\frac{1}{Re}\left(\frac{\partial u_y}{\partial n}\right)v_y + \int_\Omega \frac{1}{Re}\boldsymbol{\nabla}v_y\cdot\boldsymbol{\nabla}u_y + \int_\Omega\left(u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y}\right)v_y + \int_\Omega\frac{\partial p}{\partial y}v_y = 0 \end{cases}$$

1.2 Computation of the base flow

In order to address the iterative solution of the steady base flow, the system (1.2) is brought in form (1.5) and the Newton method is applied to the system (1.5). The resulting nonlinear problem can be compactly written as

$$\boldsymbol{F}(\boldsymbol{v}) = 0 \tag{1.5}$$

where $\boldsymbol{F}(\boldsymbol{v})$ is a vector of n nonlinear equations and n is the number of scalar unknowns in vector unknown $\boldsymbol{v}$. In the case of Navier-Stokes equations expressed in primitive-variable formulation the vector $\boldsymbol{v}$ is defined as follows.

$$\boldsymbol{v} = \begin{pmatrix} \boldsymbol{u} \\ \\ p \end{pmatrix}$$

The left-hand side of system (1.5) can be expanded in Taylor series truncated at first order in the neighbourhood of the n-th approximation of the solution obtaining

$$\boldsymbol{F}(\boldsymbol{v}) \approx \boldsymbol{F}(\boldsymbol{v}_n) + \partial_{\boldsymbol{v}}\boldsymbol{F}(\boldsymbol{v}_n)(\boldsymbol{v} - \boldsymbol{v}_n) \tag{1.6}$$

The derivative term in this formulation assumes the meaning of *Jacobian matrix*, and contains the derivative of each equation with respect to all unknowns of the problem. So the expression (1.6) can be rewritten as

$$\boldsymbol{F}(\boldsymbol{v}) \approx \boldsymbol{F}(\boldsymbol{v}_0) + \boldsymbol{J}(\boldsymbol{v}_0)(\boldsymbol{v} - \boldsymbol{v}_0) \tag{1.7}$$

According to (1.5) right-hand term side of (1.7) is zero, so the formulation of Newton's method is obtained as follows:

$$\boldsymbol{F}(\boldsymbol{v}_n) + \boldsymbol{J}(\boldsymbol{v}_n)(\boldsymbol{v} - \boldsymbol{v}_n) = 0 \tag{1.8}$$

And the linear system leading to the *n+1* approximation reads

$$\boldsymbol{J}(\boldsymbol{v}_n)\boldsymbol{v}_{n+1} = \boldsymbol{J}(\boldsymbol{v}_n)\boldsymbol{v}_n - \boldsymbol{F}(\boldsymbol{v}_n) \tag{1.9}$$

Chapter 3 shows how the Newton's method can be implemented by means of the Trilinos library.
In any iterative method a convergence criterion for stopping the computation is needed. In his case, the iteration is stopped when the residual $F(v_n)$ is less than 10^{-8} for the cavity problem and 10^{-10} for the cylinder problem. For the Newton's method an initial guess is required, and in this work a zero initial guess was sufficient to achieve convergence in all the test cases. In this way the steady solution of problem (1.1) can be computed. This solution serves as base flow for the stability analysis presented in the next paragraph.

1.3 The linear stability problem

In order to investigate the onset of the instability in an incompressible flow the linear theory is employed. Thus the solution field $\boldsymbol{v} = \{\boldsymbol{u}, p\}^T$ is decomposed into the sum of a steady part and a small unsteady perturbation.

$$\boldsymbol{u}(x, y, t) = \boldsymbol{U}_0(x, y) + \boldsymbol{u}_\delta(x, y, t)$$

$$p(x, y, t) = P_0(x, y) + p_\delta(x, y, t)$$

where $\boldsymbol{v}_0 = \{\boldsymbol{U}_0, P_0\}^T$ represents of the base flow computed through the procedure descripted in the previous paragraph. The linearization of equation (1.1) leads to the decomposition of velocity and pressure into a steady part and an unsteady perturbation. Neglecting second and higher order terms, the linearized evolution problem reads

$$\begin{cases} \dfrac{\partial \boldsymbol{u}_\delta}{\partial t} + \left(\left(\boldsymbol{U}_0 \cdot \boldsymbol{\nabla}\right) \boldsymbol{u}_\delta + \left(\boldsymbol{u}_\delta \cdot \boldsymbol{\nabla}\right) \boldsymbol{U}_0\right) - \dfrac{1}{Re}\nabla^2 \boldsymbol{u}_\delta + \boldsymbol{\nabla} p_\delta = 0 \\ \\ \boldsymbol{\nabla} \cdot \boldsymbol{u}_\delta = 0 \end{cases} \tag{1.10}$$

It is important to notice that boundary conditions are homogeneous in this case, because the pertubation on velocity and pressure is assumed to be null on all edges of the domain. Essential Dirichlet boundary conditions are employed for $\boldsymbol{u}_\delta$ and p_δ.
Introducing the following expression for the velocity and pressure perturbation in the linearized problem

$$\boldsymbol{u}_\delta(x, y, t) = \hat{\boldsymbol{u}}_\delta(x, y) e^{\lambda t} \tag{1.11a}$$

$$p_\delta(x, y, t) = \hat{p_\delta}(x, y) e^{\lambda t} \tag{1.11b}$$

The following system is obtained:

$$\begin{cases} \lambda e^{\lambda t} \hat{\boldsymbol{u}_\delta} + \left(\left(\boldsymbol{U}_0 \cdot \boldsymbol{\nabla}\right) \hat{\boldsymbol{u}_\delta} + \left(\hat{\boldsymbol{u}_\delta} \cdot \boldsymbol{\nabla}\right) \boldsymbol{U}_0\right) e^{\lambda t} - \dfrac{1}{Re} e^{\lambda t} \nabla^2 \hat{\boldsymbol{u}} + e^{\lambda t} \boldsymbol{\nabla} \hat{p_\delta} = 0 \\ \\ \boldsymbol{\nabla} \cdot \hat{\boldsymbol{u}_\delta} = 0 \end{cases} \tag{1.12}$$

After discretization by the finite element method, see the next paragraph, a non-symmetric, generalized eigenvalue problem is obtained.

$$\lambda \boldsymbol{M} \hat{\boldsymbol{v}} = \boldsymbol{A} \hat{\boldsymbol{v}} \tag{1.13}$$

The computed eigenvalues and eigenvector, solution to the problem (1.13), will be in general complex, and taking a look to equations (1.11) it becomes evident that if their real part is positive the corresponding mode amplitude grows exponentially in time. If the real part of an eigenvalue is positive, the corresponding eigenfunction is linearly unstable. The computed base flow is therefore linearly unstable if one or more eigenvalues have positive real part. This work deals with the investigation of the first instability of the flow and for this reason the eigenvalue with largest real part is sought for. A few different techniques have been proposed so far to solve the eigenvalue problem. In this work a shift-invert transformation (also known as *spectral transformation*) is employed to increase the efficienty of convergence of the employed Block Krylov Schur solver. In this method the following change of variable is introduced,

$$\lambda = \sigma + \frac{1}{\nu} \tag{1.14}$$

which leads to the solution of the transformed eigenvaue problem

$$\left(\boldsymbol{A} - \sigma \boldsymbol{M}\right)^{-1} \boldsymbol{M}\hat{\boldsymbol{v}} = \nu \hat{\boldsymbol{v}} \tag{1.15}$$

The value of σ is a shift that can be used when the position in the complex plane of the most unstable eigenvalue is approximately known a priori. A real or complex shift can be used to converge to the eigenvalues nearest to the shift. In this work a complex shift has not been employed due to limitations of the eigenvalue solver package. It is important to notice that the eigenfunctions of the transformed problem are the same as those of the original one (1.13).

1.4 The Finite Element Method

The Finite Element Method (also known as FEM) is employed to discretize the differential problems. The Finite Element Method is a numerical technique for the efficient solution of partial differential equations, such as the Navier-Stokes equations. The discretization and the solution of a partial differential equation requires several steps as listed below and shown in figure 1.1.

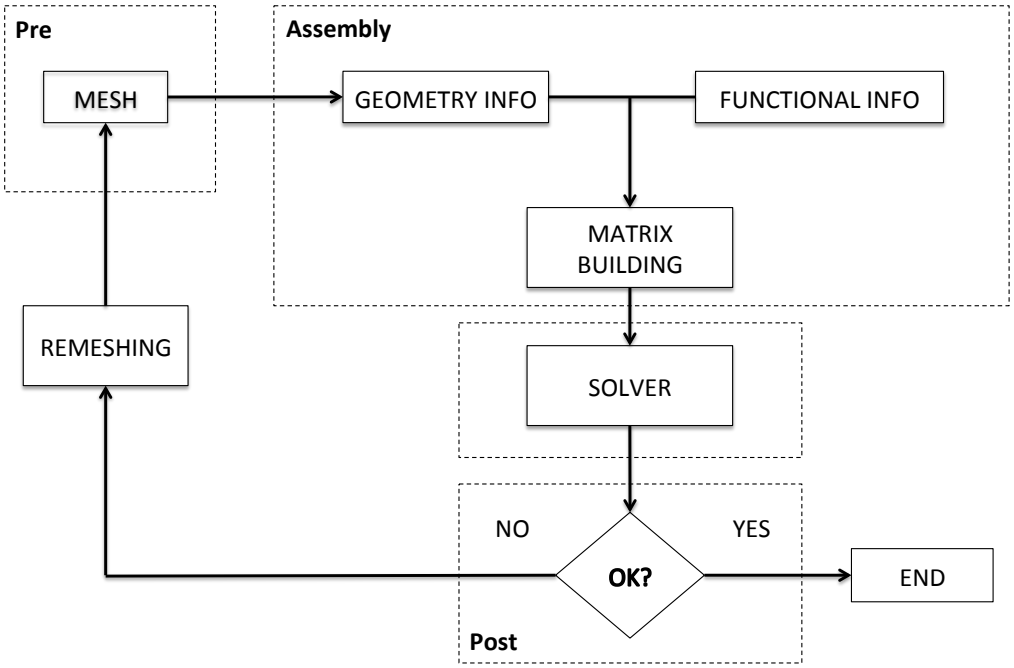


Figure 1.1: Typical block scheme for a FEM code

- Pre-processing and discretization of the problem.
- Assembly of the coefficient matrix and vector of known term.
- Solution of the algebraic system.

Each of these steps will be explained in more depth in the following sections.

1.4.1 Discretization of the problem

The first step is to model the geometry of the computational domain using a CAD software. The geometry model is then fed to the mesh generator to obtain the

computational grid. In this phase, for 2D and 3D problems, the shape of the elements must be choosen to suit the solver capabilities. Figure 1.2 shows some typical element shapes. The user should also take care of the quality of the mesh. In particular, attention should be paid to refine the element size in some critical zones of the domain, such as corners, near-wall regions, wakes. In this work, ustructured two-dimensional meshes of triangles are employed.

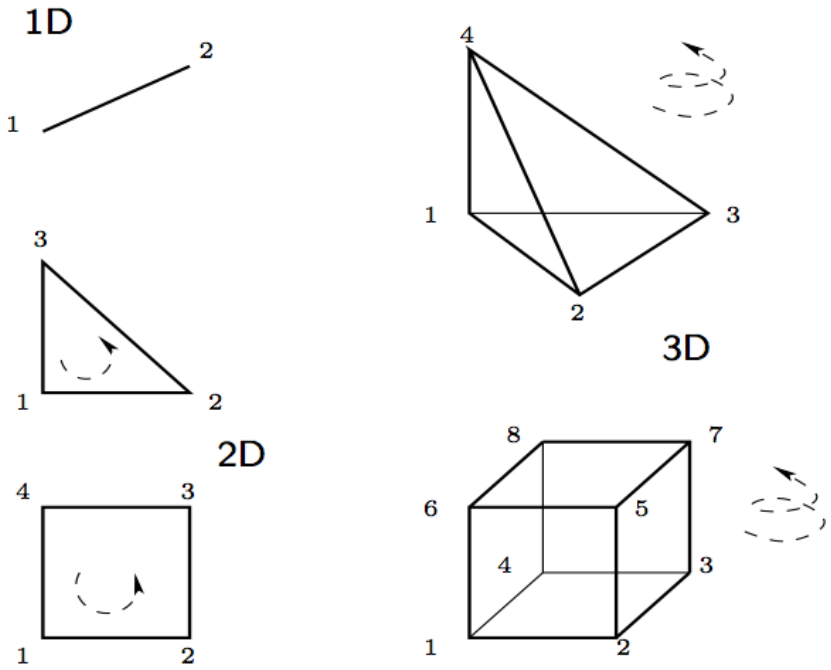


Figure 1.2: Examples of available element shapes

In the Finite Element Method the test and trial functions are approximated by continuous piecewise polynomial functions built on the tessellation:

$$f \approx f_{h,p}$$

where h is the spacing due to discretization and p is the order of polynomials used to approximate the solution on a single element.
The discrete approximation to the unknowns and the test function reads

$$f_{h,p}(x,y,t) = \sum_{i=1}^{n} \Phi_i(x,y) f_i(t)$$

where ϕ represents a spatially-dependant function used to describe the solution with support on the single finite element bubble. Differential operators can be

applied directly to the discrete approximation of the solution, for instance the discrete Laplacian is built as follows:

$$\nabla^2 f_{h,p}(x,y,t) = \sum_{i=1}^{n} \left(\nabla^2 \Phi_i(x,y)\right) f_i(t)$$

1.4.2 Assembly of the linear system

When a linear equation is discretized by the finite element method a system of the following kind is obtained.

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{1.16}$$

In this phase the stiffness matrix $\boldsymbol{A}$ is generated starting from geometric information deriving from the mesh, and functional information deriving from the polynomials choosen to describe the solution on the finite element.
In order to satisfy the compatibility condition between the velocity and pressure approximations, Taylor-Hood elements are used. Second-order Lagrangian elements are used to describe the velocity unknowns, while first-order Lagrangian elements are used for the pressure.
To complete the assembly of the linear system, boundary conditions should be applied. The simplest way to impose Dirichlet boundary conditions is to eliminate rows and columns corresponding to the prescribed degrees of freedom. As figure 1.3, shows the number of unknowns is thus reduced.

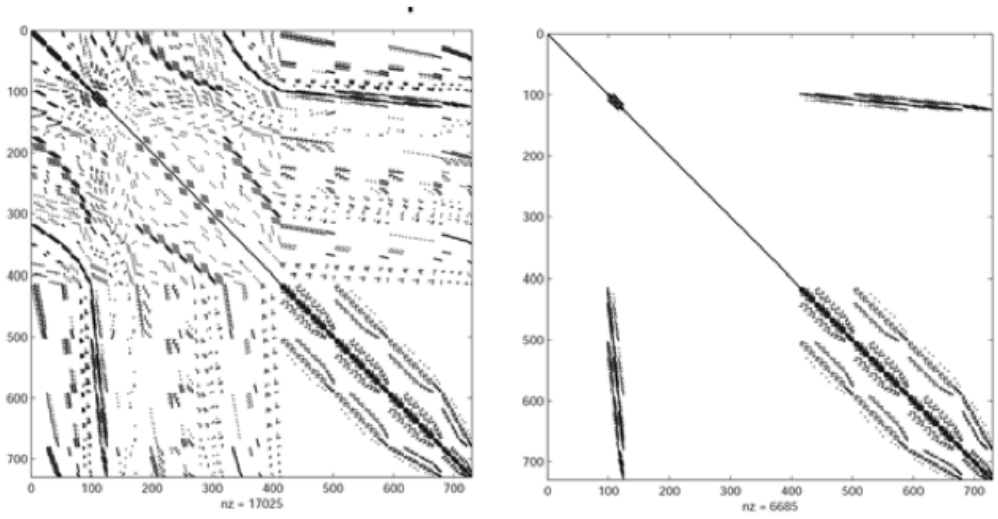


Figure 1.3: Pattern of FEM matrix before (left) and after (right) enforcing the Dirichlet boundary conditions for a 3D problem

1.4.3 Solution of the algebraic system

The most demanding operation, in terms of computing power, in a FEM simulation is represented by the solution of the linear system (1.16). This operation has to be performed once for each iteration in the solution of a nonlinear problem, therefore an efficient linear solver must be used. In this work, the direct linear solver provided with the Amesos package has been employed.

Chapter 2

Introduction to Trilinos library

Trilinos is a collection of object-oriented open source software libraries, called packages, developed by *Sandia National Laboratories* and intended to be used as building blocks for the development of scientific applications.
The Sandia National Laboratories are two major United Stated Department of Energy research and development national laboratories. The primary campus is located on Kirtland Air Force Base in Albuquerque, New Mexico and the other is in Livermore, California. Their primary mission is to develop, engineer, and test the non-nuclear components of nuclear weapons, but other aims include research and development in energy and environmental programs, as well as the security of critical national infrastructures. In addition, Sandia is home to a wide variety of research including computational biology, mathematics (through its Computer Science Research Institute), materials science, alternative energy, psychology, MEMS[1], and cognitive science initiatives.

2.1 The Trilinos Project

The Trilinos Project is an open-source *C++* library aimed to facilitate the design, development and intergration of mathematical software. Its goal is to develop algorithms and technologies within an object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific applications. Particularly the use of an object-oriented framework gives a very useful flexibility in scientific programming. The Trilinos library is made of different packages, each of them is a self-contained, independent piece of software with its own set of requirements, its own development team and group of users. Because of this, Trilinos itself is designed to respect the autonomy of packages. Trilinos offers a variety of ways for a particular package to interact with other Trilinos packages. It also offers a set of tools that can assist package developers with builds across multiple platforms, generating documentation and regression testing across a set of target platforms. Trilinos packages provide a development framework for differ-

[1] Microelectromechanical systems

ents aspects of mathematical problems, such as constructing and using sparse and dense matrices, graphs and vectors, iterative and direct solution of linear systems, parallel multilevel and algebraic preconditioning, solution of non-linear, eigenvalue and time-dependent problems, PDE-constrained optimization problem, partitioning and load balancing of distributed data structures, automatic differentiation, and PDE-discretizations.
Below the complete list of Trilinos packages sorted into groups is reported and shortly described.

1. **Basic Linear Algebra Libraries**:
   - **Epetra**: core linear algebra package. Facilitates construction and manipulation of serial graphs, sparse ande dense matrices, vectors and multivectors.
   - **EpetraExt**: extension to the core linear algebra package, Epetra.
   - **Tpetra**: next-generation templated version of Petra, taking advantage of the newer advanced features of C++.
   - **Jpetra**: experimental Java of the Petra library.
   - **Kokkos**: core kernel package.
2. **Preconditioners**:
   - **AztecOO**: ILU-type preconditioner.
   - **IFPACK**: distributed algebraic preconditioner package. Includes incomplete factorizations and relaxation-based preconditioners in domain decomposition framework. Compatible with AztecOO.
   - **Ifpack2**: Contains preconditioners that operate on the templated linear-algebra objects provided by the Tpetra package. Intended as a templated replacement for Ifpack.
   - **ML**: multilevel, distributed memory algebraic preconditioners. Provides multi-level, multigrid-like preconditioners for distributed linear systems. Compatible with AztecOO.
   - **Teko**: blocked and segregated preconditioning package.
3. **Linear Solvers**
   - **Epetra**: provides wrappers for select BLAS and LAPACK routines.
   - **Teuchos**: provides wrappers for select BLAS and LAPACK routines.
   - **Pliris**: an object-oriented interface to a LU solver for dense matrices on parallel platforms.
   - **AztecOO**: preconditioned Krylov solver package. Supercedes Aztec 2.1. Solves linear systems of equations via preconditioned Krylov methods. Uses Epetra objects, compatible with IFPACK, ML and Aztec.

- **Belos**: next-generation iterative solvers written using a traits interface, meaning that it has no explicit dependence on any concrete linear algebra library. Instead, it can be used with any concrete linear algebra library that implements the Thyra abstract interfaces and even Epetra directly.
- **Komplex**: complex linear solver package. Solves complex-valued linear systems via equivalent real formulations.
- **Amesos**: direct solver classes. Supports use of a growing list of third party direct solvers, including DSCPACK, SuperLU, SuperLUDist and UMFPACK. Compatible with Epetra.
- **Amesos2**: direct solver library/interface in Trilinos. Amesos2 provides interfaces to third-party direct solvers for templated matrices and vectors in Trilinos

4. **Nonlinear, Transient, and Optimization Solvers**

   - **NOX**: nonlinear solver package. Abstract and concrete classes for construction and solution of nonlinear problems.
   - **LOCA**: LOCA is a software library for performing bifurcation analysis of large-scale applications.
   - **MOOCHO**: solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems.
   - **Piro**: Piro is striving to be the single unifying layer above all nonlinear solver, time integration, optimization, and UQ packages.
   - **Rythmos**: Rythmos is a transient integrator for ordinary differential equations.
   - **TriKota**: TriKota is a convenience package that builds the Dakota framework underneath Trilinos as if it were a Trilinos package. Dakota contains a wide array of algorithms for optimization and UQ.
   - **GlobiPack**: the GlobiPack package contains a set of interfaces and implementations for 1D globalization capabilities to be used in nonlinear solvers, optimization solvers, and similar algorithms that require globalization methods (e.g. line search and trust region methods).
   - **OptiPack**: the OptiPack package contains interfaces and concrete implementations of some basic optimization algorithms based on Thyra. The globalization methods used are implemented in GlobiPack.

5. **Eigensolvers**

   - **Anasazi**: Anasazi is an extensible and interoperable framework for large-scale eigenvalue algorithms.

6. **Automatic Differentiation**

- **Sacado**: Sacado is a package for automatic differentiation of C++ programs.
- **Stokhos**: Stokhos is a package for intrusive stochastic Galerkin uncertainty quantification methods.

7. **Domain Decomposition**

   - **Claps**: Claps is a collection of domain decomposition preconditioners and solvers.

8. **Mortar Methods**

   - **Moertel**: surface coupling of different physical models, discretization schemes or non-matching triangulations along interior interfaces of a domain.

9. **Partitioning / Load Balancing**

   - **Isorropia**: Isorropia is a partitioning and load balancing package, intended to assist with redistributing objects such as matrices and matrix-graphs in a parallel execution setting.
   - **Zoltan**: Zoltan is a toolkit of parallel services for dynamic, unstructured, and/or adaptive simulations, parallel dynamic load balancing for applications, including finite element methods, matrix operations, particle methods, and crash simulations, parallel graph coloring, matrix ordering, unstructured communication tools, and distributed data directories.

10. **Abstract Interfaces and Adapters**

    - **Thyra**: Abstract linear solver package. Replaces the now-deprecated TSF family.
    - **PyTrilinos**: Python interfaces to selected Trilinos packages.
    - **CTrilinos**: C interfaces to selected Trilinos packages.
    - **ForTrilinos**: ForTrilinos provides a set of standards-conforming, potrable Fortran interfaces to Trilinos packages.
    - **WebTrilinos**: Web interface to experiment with Trilinos through a browser.
    - **Stratimikos**: Stratimikos contains a unified set of Thyra-based wrappers to linear solver and preconditioner capabilities in Trilinos.
    - **FEI**: a general interface for assembling finite-element data into a system of linear equations.
    - **TrilinosCouplings**: a collection of interfaces between packages.

11. **Mesh Generation, Improvement, and Adaptivity**

- **Mesquite**: Applies a variety of node-movement algorithms to improve the quality and/or adapt a given mesh.
- **PAMGEN**: PAMGEN creates hexahedral or quadrilateral (in 2D) finite element meshes of simple shapes (cubes and cylinders) in parallel.

12. **Discretization Utilities**

- **Intrepid**: Intrepid is a library of interoperable tools for compatible discretizations of Partial Differential Equations (PDEs).
- **phdMesh**: the Parallel Heterogeneous Dynamic unstructured Mesh (phdMesh) data structure library is intended to be component used within a finite element or finite volume library or code. The phdMesh data structure supports arbitrary unstructured mesh connectivity, application-defined groupings of mesh entities, and application-defined computational field data.
- **STK**: contains capabilities intended to support massively parallel multiphysics computations on dynamically changing unstructured meshes.

13. **Utilities**

- **Teuchos**: Common tools package.
- **TriUtils**: TriUtils is a package of utilites used by many of the Trilinos packages.
- **EpetraExt**: Matrix/Vector read/write utilities.
- **RTOp**: RTOp (reduction/transformation operators) provides the basic mechanism for implementing vector operations in a flexible and efficient manner.
- **Galeri**: Galeri is a package for generating linear systems used by many of the Trilinos packages for examples and tests.
- **ThreadPool**: minimalistic interface for orchestrating the thread-parallel execution of functions within a pool of threads.
- **Optika**: the Optika package provides trilinos developers with easy access to GUI input methods for their programs.
- **SEACAS**: the Sandia Engineering Analysis Code Access System (SEACAS) is a collection of applications for manipulating Exodus databases.

14. **PDE Discretization Tools**

- **Phalanx**: Phalanx is a local field evaluation kernel specifically designed for general partial differential equation solvers.
- **Intrepid**: Intrepid is a library of interoperable tools for compatible discretizations of Partial Differential Equations (PDEs).

- **Shards**: Shards is a suite of common tools for numerical and topological data that facilitate interoperability between typical software modules used to solve Partial Differential Equations (PDEs) by finite element, finite volume and finite difference methods.

15. **Instructional**

- **Didasko**: Didasko is the Trilinos tutorial, and contains several examples, detailed descriptions, tips, and suggestions for most Trilinos packages.
- **New_Package**: a sample Trilinos package containing all of the infrastructure to install a new package into the Trilinos framework. Contains the basic directory structure, a collection of sample configuration and build files and a sample "Hello World" package.

16. **PDE Toolbox**

- **Sundance**: Sundance is a system for rapid development of high-performance finite-element solutions of partial differential equations.

Since Sundance is the package that will be employed in this work to develop a code to investigate the linear stability of 2D incompressible flows, it will be described in more detail in section 2.2.

2.1.1 About the GNU Lesser General Public License

The Trilinos library is licensed under the GNU Lesser General Public License (or LGPL). The LGPL is a free software license published by the Free Software Foundation, and it was designed as a compromise between the GNU General Public License (or GPL) and *permissive license*. While permissive license permit the redistributor to add other license terms and potentially restrictions to a derived work, the copyleft-GPL do not allow further restrictions. However both free software licenses offer the same freedoms in terms of how the software can be used, studied and privately midified. The main difference between LGPL and GPL is that the first allows the software to be linked with other softwares with different licenses. So there are two cases: works deriving from software, and works thas use the software. This latter case is not authorized by the license.

2.2 Sundance

Sundance is a multi-package toolbox of Trilinos, aimed to be a system for rapid development of high-performance parallel finite-element solutions of PDEs, so it uses a general, powerful, and quite elegant method for turning a PDE into a discrete system of algebraic equations. Sundance is described with expressions, function spaces, and domains instead of low-level concepts such as matrix entries, elements, and nodes. Sundance does not require the user to provide the "stiffness matrix" to te software but the user just writes a set of simbolic equations and Sundance will provide to manage them. Here is the power of high-level programming of Sundance, and it is possible thank to the C++ object-oriented framework, on which it's based on. So Sundance is intended to let the user make choiches by selecting and combining high-level objects rather than by writing low-level code. The idea is that the user should be able to code a finite element problem using the same level of abstraction he would use to describe the problem and its discretization on a blackboard. The high-level nature of the components means also that the user need not worry about tedious and error-prone bookkeeping details. In addition to the advantage of conceptual simplicity and freedom from bookkeeping, this component-based approach allows a high degree of flexibility in the formulation, discretization, and solution of a problem.
Solution of partial differential equations is a complicated endeavor with many subtle difficulties, and there can be no *one-size-fits-all* simulation code. For this reason Sundance is not a simulation code but it is a set of high-level objects that will let the user build his own simulation code. These objects shield the user from rather tedious bookkeeping details required in writing a finite-element code, but they do not shield from the need to understand how to do a proper formulation and discretization of the problem.
Here is a list of the main features of Sundance:

- Sundance provide a finite-element model of PDE based on a weak form of PDE set. The way to describe the PDEs's set is to write the weak form using a high-level symbolic notation.

- The geometric domain of the problem is decomposed into a discrete mesh of cells. Sundance provide two choices to generate the mesh: the user can generate the mesh using the built-in mesh generator with simple capability (useful just for some toy problems), or carry out the mesh with a third-party software and then import it in Sundance.

- Due to the complexity in solving a large system of linear and nonlinear equations $\mathbf{Kx} = \mathbf{f}$ Sundance lets the user to choose the best solver to suit the structure of the problem.

- The solution of nonlinear problem is reduced to solving a sequence of linear problems. There are many choices of linearization method and iteration

for a nonlinear problem, each one resulting in a different sequence of linear problems.

As the reader could note, in solving a PDE set of equations with the finite-element method the designer will be faced with a large number of choices. In this way Sundance is intended to let the user make these choices combining high-level objects, but it cannot help the user make good choices. The user is required to possess an adequate knowledge of the problem and of the library to produce an efficient code.

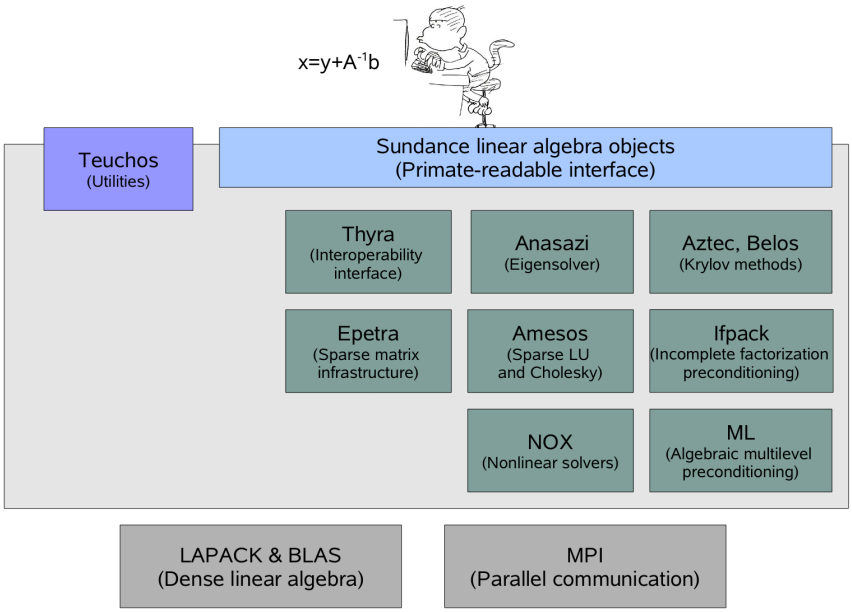


Figure 2.1: Sundance's structure

Figure 2.1 shows which Trilinos packages make up the Sundance toolbox. For the description of each package the reader is referred to paragraph 2.1.

2.3 Set up the environment

Before starting to configure the installation of Trilinos the environment of the system must be set up.
First of all, the user needs to get the Trilinos distribution from website[2]. The download of the desired release of Trilinos is available after e-mail registration. After the download the user has to choose the destination of Trilinos installation and extract the tarball file in the desired directory.

[2]http://trilinos.sandia.gov

```
$ tar zxvf trilinos-10.8.3-Source /<USER_PATH>
```

For example, in this job the <USER_PATH> variable was set to /home/andrea/Projects. It's really recommended to have two different locations for the installation directory and the Trilinos's source directory. For this reason, the user is invited to create a new directory within the Trilinos source using very simple terminal command like this:

```
$ mkdir BUILD
$ cd BUILD/
```

Once the source and build directories are ready, the user has to set up the environment variable TRILINOS_HOME with the location of Trilinos source directory. In the present case the following command was employed:

```
$ export TRILINOS_HOME=/<USER_PATH>/trilinos-10.8.3-Source
```

To view the complete list of environment variables and verify the correct set up described above the bash command env can be used.

Moreover, since most of Trilinos packages are written in C++ with Fortran kernels, the user needs to install a C++ compiler and a Fortran compiler too. The last step required to install cmake. The easiest way is to get it is using the following command from terminal:

```
$ non l'ho installato con: apt-get install cmake

$ ./configure
$ gmake
$ make install
```

and this will download and install the last version of cmake available for the Linux release found on the system. After that the environment is fully equipped.

2.4 Third-Party software

Sundance manages the most of the problem but, as already said, it lets the user make some choices regarding preprocessing and postprocessing modes. For what concerns the preprocessing fase the user can choose to generate the mesh with an external third-party software. The current release supports two input file formats; the first one is **Shewchuk's Triangle Mesher**, a smart Two-Dimensional mesh generator based on Delaunay Triangulation developed by University of California at Berkeley, while the second one is the Exodus file format. ExodusII format is provided by **Cubit** software, developed and distribuited by Sandia under commercial license. ExodusII file format is provided also by other commercial software, one of these is **Ansys IcemCFD**. Thanks to Politecnico di Milano it could be

possible to use IcemCFD under academic license to generate all meshes needed for this work, that will be shown in details in chapters 4.
Sundance lets also the user choose the output format of the results, and which third-party software is preferred. As described in chapter 3, in this work it was choosen the VTK output format and **ParaView** as visualization software.
In the following subparagraphs all required and optional third-party libraries needed to reach the aim of this work are presented.

2.4.1 BLAS and LAPACK

As the figure 2.1 shows, Sundance is a multi-package toolbox of Trilinos that exploits also some external libraries. It is the case of the **Basic Linear Algebra Subprograms** library, better known as **BLAS**, and the **Linear Algebra PACKage**, **LAPACK**. **BLAS** and **LAPACK** are required third-party libraries without which the configuration of any package of Trilinos will crash.
On Debian or Ubuntu distributions, the quickest way to install these libraries is with terminal, as follows:

```
$ apt-get install libblas-dev liblapack-dev
```

The system will require the root privileges to start this operation. This will complete the installation of **BLAS** and **LAPACK** libraries.

2.4.2 MPI

The **Message Passing Interface**, or **MPI**, is a protocol for parallel computer communication between nodes belonging to a cluster of computers. Unlike BLAS and LAPACK, this is just an optional library, that is necessary if the user want to run his Trilinos's simulation in parallel. One of the goals of this work was to be able to run the Sundance's code in parallel, to reduce execution time and to enable memory demanding computations. Indeed, it is very frequent to deal with large-scale problems in scientific applications and the capability to run program on a distributed memory system could be really useful and appreciated.
The user can use terminal command to install MPI on the system in very simple and easy way:

```
$ apt-get install mpi
```

To run an executable program in parallel mode the user should use the following command:

```
$ mpirun -np <NP> ./<FILENAME>.exe
```

where `<NP>` is the number of concurrent parallel processes.

2.4.3 Chaco Mesh Partitioner

In order to run simulations in parallel mode Sundance needs the mesh already partitioned. If the choosen mesh format is Exodus, as it was for this work, the geometry should be partitioned using **Chaco Mesh Partitoner**, a program developed at Sandia and available free on Sandia's open-source software portal[3].
Once the mesh partitioner has been downloaded, the next step is unpacking the downloaded file into a directory and use the command `make` in the main directory to compile Chaco. Once the installation ends the variable `PATH` in the `bashrc` file needs to be updated by adding the location of the executable file `Chaco` generated during installation, that will be located in the `<CHACO_DIR>/Chaco-2.2/exec` directory. So, regarding this work, the final state of the variable `PATH` to export in the `bashrc` file appeared like this:

```
 export PATH=/home/andrea/Projects/trilinos-10.8.3-Source/MyApps/
MeshPartitioner/Chaco-2.2/exec:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin
```

The user should pay attention not to cancel other entries in the variable `PATH`, but just update it.
The procedure to be followed to partition the mesh is quite simple, and amounts to run a short Sundance code in which the user should just specify the name of the source mesh file, `<MeshFile>`, and the number of processors of the parallel simulation, `<NP>`. Indeed Chaco will divide the domain in the number of parallel processes. Particular thanks to Kevin Long for providing the author with the code reported in appendix A.1. As the reader can notice, information about the namefile to partition and the number of parts to create are to be reported in the source file as variables. Next the code should be compiled and run to divide the mesh in the desired number.

2.4.4 Exodus II

Since the user will probably want to use ExodusII meshes ExodusII library has to be installed. There are different ways to do this. The simplest one, requires the installation of the **SEACAS** Trilinos package. Trilinos optional packages can be installed by following the instructions reported in section 2.5.

2.4.5 NetCDF

In order to use an ExodusII mesh the **NetCDF** (Network Common Data Form) library also has to be installed. First, the user should download the package from the Unidata website[4], and then extract the content of the tarball file into a destination directory. To install the library the user has to run the `configure` script located in the main directory of NetCDF in this way:

[3] http://www.cs.sandia.gov/web1400/1400_download.html
[4] http://www.unidata.ucar.edu/software/netcdf/

```
 $ ./configure --prefix=/home/andrea/Projects/netcdf-4.1.3/BUILD
--disable-netcdf-4
 $ make check install
```

where `prefix` is the path of a new directory previously created from the user within the installation directory. After the installation's end it could be necessary fix an environment variable. The best way to do it and make this automatic is to add `bashrc` file the following line:

```
export LD_LIBRARY_PATH=/home/andrea/Projects/netcdf-4.1.3/BUILD/lib
```

to the `.bashrc` file in the user home directory.

2.5 Trilinos installation

2.5.1 Configuration

Once the environment is ready (refer to 2.3) and all third-party software has been installed, the user must to choose which Trilinos packages are going to be built. This can be done running a configuration script containing the instructions of what packages the user wants to enable, from the BUILD directory created before. In this work the following script was used.

```
EXTRA_ARGS=$@

cmake \
 -D CMAKE_BUILD_TYPE:STRING=DEBUG \
 -D Trilinos_ENABLE_Sundance:BOOL=ON \
 -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
 -D Trilinos_ENABLE_TESTS:BOOL=ON \
 -D Trilinos_ENABLE_SEACAS:BOOL=ON \
 -D Trilinos_ENABLE_Pamgen:BOOL=TRUE \
 -D TPL_ENABLE_MPI:BOOL=ON \
 -D MPI_INCLUDE_DIRS:PATH=/usr/lib/openmpi/include \
 -D MPI_LIBRARY_NAMES:STRING="mpi;mpi_cxx;mpi_f77;mpi_f90;
       openmpi_malloc;open-pal;open_rte;otf;vt;vt.fmpi;vt.mpi;
       vt.omp;vt.ompi" \
 -D MPI_LIBRARY_DIRS:PATH=/usr/lib/openmpi/lib \
 -D TPL_ENABLE_Netcdf:BOOL=ON \
 -D Netcdf_INCLUDE_DIRS:PATH=/<MY_PATH>/netcdf-4.1.3/include \
 -D Netcdf_LIBRARY_NAMES:STRING="netcdf;netcdf_c++;netcdff" \
 -D Netcdf_LIBRARY_DIRS:PATH=/<MY_PATH>/netcdf-4.1.3/BUILD/lib \
$EXTRA_ARGS \
${TRILINOS_HOME}
```

As the reader can see, the script is composed by two main parts. In the first the user must give information about which packages have to be enabled, if test cases have to be included, and if all optional packages have to be built. Particularly this last instruction is strongly recommended in the case of Sundance, because it is a

multi-package toolbox with a lot of external dependencies. The compilation of a Sundance executable will fail if all the required library are not installed.
The second part of the configuration script starts with `TPL` (third-party library) and deals with the third-party software that the user wants to let work with Trilinos. As explained in [21] the user should add four lines for each TPL he wants to include in the installation. Particularly, the first line is:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=ON
```

where `<TPLNAME>` is first MPI, and then Netcdf in the given example. Refer to [20] for the complete list of names of supported third-party library.
Notice that also the order of the TPLs in relevant.
The second line deals with the path to the header include directories. The syntax is:

```
-D <TPLNAME>_INCLUDE_DIRS:PATH=<INCLUDE_PATH>
```

where `<INCLUDE_PATH>` is the path of the include directory of the third-party library in the current installation. e.g. in my installation, the MPI include directory path is `/usr/lib/openmpi/include`, regarding to MPI.
The third line is the list of unadorned library names, put in the same order needed by linker. Notice that the platform-specific prefixes (e.g. 'lib') and postfixes (e.g. '.a', '.lib', or '.dll') will be added automatically. The fourth line is the path of the directory where the library files can be found in the thirdparty installation. Examples are given below:

```
-D <TPLNAME>_LIBRARY_NAMES:STRING=<LIST_OF_LIBRARY>
-D <TPLNAME>_LIBRARY_DIRS:PATH=<LIBRARY_PATH>
```

where for instance `<LIST_OF_LIBRARY>` for the MPI libraries `libmpi.a`, `libmpi_cxx.a` and `libmpi_f77.a` is something like `mpi;mpi_cxx;mpi_f77;`, and `<LIBRARY_PATH>` is `/usr/lib/openmpi/lib`, for every.
These four lines are to be replicated each third-party library the user wants to include.
The user should run the following command in order to start the configuration:

```
$ ./do-configure.sh
```

where `do.configure` is the name choosen for the configuration script described above. Once the system give a positive responce like this:

```
$ Configuring done!
```

the configuration ends and the system is ready to complete the installation.

2.5.2 Building

The next step is to compile and build the library. To do this the following instruction must be given from terminal:

```
$ make
```

This will take the most part of installation time. On my system[5] it took approximately 75 minutes. The last step is the installation, so as root do:

```
$ make install
```

and this will end the installation of Trilinos and Sundance.

[5]MacBookPro 8,1, Dual Core Intel i7 2.7 GHz, 4 GB RAM, working with Linux Ubuntu 11.04 distribution.

Chapter 3

Software's structure

3.1 How Sundance works

The reasons that led Trilinos developers to build Sundance was the desire to create a tool for the rapid development of partial differential equation solvers based on finite element method so that it was the most user-friendly as possible. This goal is achieved the way passes through an intuitive formulation of weak form of the original problem so that the users can programm PDEs simulators quite rapidly. The very high-level programming suits fine to this requirement, managing individually a large number of issues that remain hidden to the user, and for this reason Sundance is a powerful tool. Figure 3.1 shows the whole Sundance simulation process, that anc be split into two main blocks. The first one is the problem formulation. This phase involves mesh generation through a compatible third-party software and the formulation of equations and boundary conditions in weak form. The second one is the problem solution in which the user should choose the right solver from the Trilinos package. Building the system very simple using Sundance objects.

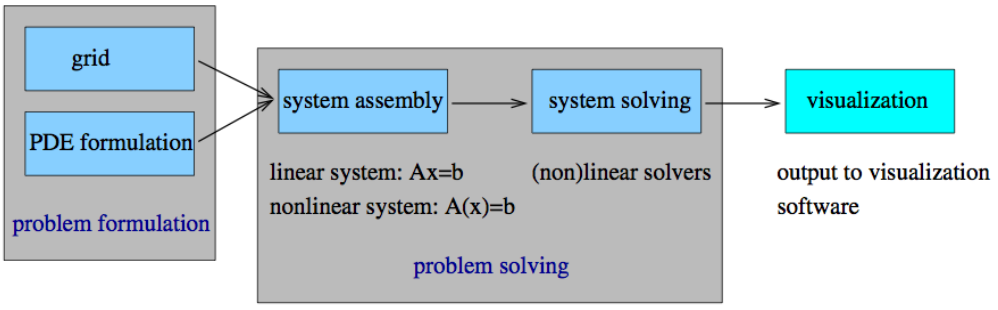


Figure 3.1: Sundance simulation process, from [5]

3.2 Building the code

In this paragraph the main parts of the code are presented and the reader can refer to appendix A.2 for the complete code used for the Navier-Stokes simulation.

3.2.1 Iniatilization and finalization

The first step is to show the standard C++ format common to every Sundance codes.

```
#include "Sundance.hpp"

int main(int argc, char** argv)
{
   try
       {
         Sundance::init(&argc, &argv);


         /****************************
              code body goes here
          ****************************/


       }
   catch(std::exception& e)
       {
         Sundance::handleException(e);
       }
   Sundance::finalize();
}
```

These few lines are necessary since they control the initialization and the finalization of the Sundance software, and all the rest of the code should be added in place of the comment `code body goes here`. As any C++ source code all header files to include should be listed at the top of the file.

3.2.2 Getting the mesh and defining domains of integrations

Sundance lets the user two choices to provide the mesh grid. The first is to generate it using the `MeshGenerator` object, but this is functional for very simple geometries as squares and rectangles. For more complex designs the only way is to import the mesh through the `MeshSource` class. As described in paragraph 2.4 the choice is limited to two mesh formats for the current release of Sundance, the Shewchuk's Triangle mesh format and the ExodusII file format. In this work the

ExodusII format has been preferred.
Sundance is designed to work with different kinds of mesh, so first of all the user should specify which type of mesh is going to be used with the `MeshType` object. In the following example, as for the whole job, a `BasicSimplicialMeshType` was considered, which is the constructor for a grid made of simplicial elements, e.g. triangles in the two-dimensional case or tetrahedra in three-dimensions. The Sundance mesh object is built by the following lines.

```
MeshType meshType = new BasicSimplicialMeshType();
MeshSource mesher = new ExodusMeshReader( meshFile, meshType, comm );
Mesh mesh = mesher.getMesh();
```

`meshFile` is the path of the location of the Exodus II mesh file. `mesher` is an object of `MeshSource` class built through the `ExodusMeshReader` contructor, since the choosen mesh format is ExodusII. Alternatively the user could use the `TriangleMeshReader` to load a mesh made by Shewchuk's Triangle Mesher. `comm` is a communication parameter needed to run simulation in parallel. See A for the complete code and the meaning of this parameter. Once the mesh is read the next step is to partition the mesh in an interior zone and a boundary zone. To do this Sundance uses a `CellFilter` object to represent subregions of a geometric domain.

```
CellFilter interior = new MaximalCellFilter();
CellFilter edges = new DimensionalCellFilter(1);
```

The `DimensionalCellFilter(int nOfDimensions)` method is helpful to filter mesh cells by dimension. According to a two-dimensional mesh, the edges of inlet, outlet and walls are one-dimensional.
The `edges` instance should be splitted using the `labeledSubset(int number)` method into the desired parts of the domain of integration as follows.

```
CellFilter left = edges.labeledSubset(5);
CellFilter right = edges.labeledSubset(3);
CellFilter up = edges.labeledSubset(2);
CellFilter down = edges.labeledSubset(4);
CellFilter cylinder = edges.labeledSubset(1);
```

These edges have been labeled by Cubit command window through the command:

```
sideset 1 curve 1
```

and so on for all boundaries.
Notice that through the `CellFilter` object the user could also redefine names of boundaries, that could be useful in case of same boundary conditions on different edges. The following lines show an example.

```
CellFilter walls = cylinder;
CellFilter up_down = up + down;
CellFilter inflow = left;
CellFilter outflow = right;
```

The reader should refer to paragraph 3.2.4 in order to fully understand the use of these objects.

3.2.3 Defining unknowns, test functions and operators

According to the Navier-Stokes problem in primitive variables formulation we use second order Lagrange P2 elements for horizontal and vertical velocity, and first order Lagrange P1 elements for pressure. The same kind of elements is used for unknown functions and their associated test functions. Defining this is really easy, as follows.

```
BasisFamily L1 = new Lagrange(1);
BasisFamily L2 = new Lagrange(2);

Expr ux = new UnknownFunction(L2, "ux");
Expr vx = new TestFunction(L2, "vx");
Expr uy = new UnknownFunction(L2, "uy");
Expr vy = new TestFunction(L2, "vy");
Expr p = new UnknownFunction(L1, "p");
Expr q = new TestFunction(L1, "q");

Expr u = List(ux, uy);
Expr v = List(vx, vy);
```

In order to write the weak form of the Navier-Stokes equations it is necessary to define coordinates, derivative and gradient operators. This is simply done because the gradient operator is just formed by making a `List` containing the partial differentiation operators in the x e y directions.

```
Expr x = new CoordExpr(0);
Expr y = new CoordExpr(1);
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
```

As the reader could notice the gradient thus defined is treated as a vector, and the directions are always numbered starting from zero as in any C o C++ code. The gradient is simply applied with an overloading of multiplication operator, so that an operation such as `grad*u` expands correctly to {`dx*u, dy*u`}. Now everything is set up and the weak form of Navier-Stokes equations can be written.

3.2.4 Weak form and boundary conditions

In order to compute the main flow and to show how the weak form of two-dimensional steady incompressible Navier-Stokes formulation is encoded into Sundance equations (1.4) are recalled.

$$\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} = 0$$

$$u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\nabla^2 u_x$$

$$u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\nabla^2 u_y$$

We come to the integral weak formulation by multiplying for test functions and integrating by parts the strong form. The terms of velocity derivatives in normal direction are cut off because they are zero due to Neumann homogeneous boundary conditions.

$$\int_\Omega \left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} \right) q = 0$$

$$\int_\Omega \frac{1}{Re} \boldsymbol{\nabla} v_x \cdot \boldsymbol{\nabla} u_x + \int_\Omega \left(u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} \right) v_x + \int_\Omega \frac{\partial p}{\partial x} v_x = 0$$

$$\int_\Omega \frac{1}{Re} \boldsymbol{\nabla} v_y \cdot \boldsymbol{\nabla} u_y + \int_\Omega \left(u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} \right) v_y + \int_\Omega \frac{\partial p}{\partial y} v_y = 0$$

This is very simply coded in Sundance using high-level programming with the following instructions, required to build the nonlinear problem and solve the base flow.

$$\int_K f(\boldsymbol{x}) d\boldsymbol{x} \approx \sum_{iq=1}^{nqn} f(\boldsymbol{x}_{iq}) w_{iq} \tag{3.1}$$

```
QuadratureFamily quad2 = new GaussianQuadrature(2);

Expr eqn1 = Integral( interior, ( dx * ux + dy * u y) * q, quad2 );

Expr eqn2 = Integral( interior, ( 1 / reynolds ) * ( ( grad * vx ) *
            ( grad * ux ) + ( grad * vy ) * ( grad * uy ) ) + vx *
            ( u * grad ) * ux + vy * ( u * grad ) * uy - p * ( dx *
            vx + dy * vy ), quad2 );

Expr eqn = eqn1 + eqn2;
```

First of all the user should define a quadrature rule, needed to compute integrals. However, if any exactly integrable term is detected the quadrature rule will be ignored. Equation (3.1) shows how integrals will be approximated using a quadrature rule.
The second step is to enforce the appropriate boundary conditions. As told, Neumann boundary conditions are applied simply cutting out normal derivative terms from the weak form, while Dirichlet boundary conditions are managed by the `EssentialBC` object as follows.

$$\int_{walls} \boldsymbol{v} \cdot \boldsymbol{u} = 0 \tag{3.2}$$

$$\int_{inflow} v_x \left(u_x - 1.0\right) + v_y u_y = 0 \tag{3.3}$$

$$\int_{outflow} qp = 0 \tag{3.4}$$

$$\int_{up,down} v_y u_y = 0 \tag{3.5}$$

In equation 3.2 the no-slip condition is enforced on all walls, while in 3.3 and 3.5 both the velocity components are imposed at the inflow boundary and on the bottom and top sides of domain. Condition 3.3 enforces also the uniform unitary horizontal velocity field at inflow. At last, equation 3.4 mposes a null pressure at outflow. Essential boundary conditions are coded as follows.

```
Expr bcWalls = EssentialBC( walls, v * u, quad2 );
Expr bcInflow = EssentialBC( inflow, vx * ( ux - 1.0 ) + vy * uy, quad2 );
Expr bcOutflow = EssentialBC( outflow, q * p, quad2 );
Expr bcUpDown = EssentialBC( up_down, vy * uy, quad2 );

Expr bc = bcWalls + bcInflow + bcOutflow + bcUpDown;
```

As the last line shows all boundary conditions should be joint to be used in the building of nonlinear problem discussed in the next paragraph.

3.2.5 Initial guess and nonlinear problem

To build a nonlinear problem the user needs to create an expression for the initial guess. This is easily done by creating a discrete space with a vector of basis functions.

```
DiscreteSpace discSpace( mesh, Sundance::List( L2, L2, L1 ), vecType );
Expr u0 = new DiscreteFunction( discSpace, 0.0, "u0" );
```

Here the initial guess $\boldsymbol{u}_0$ is set to zero for all three components. It results to work fine with Navier-Stokes equations solved by the Newton method.
Constructing the nonlinear problem is done by using the `NonlinearProblem` constructor, as follows.

```
NonlinearProblem prob( mesh, eqn, bc, List(vx, vy, q), List(ux, uy, p),
                u0, vecType);
```

All objects previously created will be passed to the `NonlinearProblem` constructor, and it is relevant to notice the order of list of functions, because test functions should be passed before unknown functions, as shown above.
The computation of the base flow involves two different classes of problems, since the nonlinear system is solved by an iterative technique and a linear subproblem is build and processed at each nonlinear iteration. The NOX package of Trilinos was developed to provide the user with a tool for solving large-scale systems of nonlinear equations in the form 1.5 using the **Newton's method**.

```
ParameterXMLFileReader reader("nox-amesos.xml");
ParameterList solverParams = reader.getParameters();
NOXSolver solver(solverParams);

NOX::StatusTest::StatusType status = prob.solve(solver);
```

A quick clarification about the method used to solve the nonlinear system is needed. As told, the NOX package of Trilinos was employed in order to use the Newton's method. The procedure of such a method was explained in paragraph 1.2. As shown, Newton's method solve a linear system for each nonlinear iteration. So also a linear method is needed, and NOX leaves the choice of which employ to the user. Here the direct solvers contained in the Amesos package of Trilinos were used. Particularly the direct solver called KLU was used as default choice for Amesos. The list of parameters needed to call the Amesos solver through the xml file is reported below. Tolerance and maximum number of iteration are typical parameters of the Newton's method.

```
<ParameterList>
  <ParameterList name="NOX Solver">
    <Parameter name="Nonlinear Solver" type="string" value=
"Line Search Based"/>
    <ParameterList name="Line Search">
      <Parameter name="Method" type="string" value="More'-Thuente"/>
    </ParameterList>
    <ParameterList name="Status Test">
      <Parameter name="Max Iterations" type="int" value="50"/>
      <Parameter name="Tolerance" type="double" value="1e-10"/>
    </ParameterList>
    <ParameterList name="Linear Solver">
      <Parameter name="Type" type="string" value="Amesos"/>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

3.2.6 Eigenvalue problem

In order to investigate the linear stability of 2D incompressible flows the generalized eigenvalue problem 3.6 is built.

$$\lambda \mathcal{M} \boldsymbol{x} = \mathcal{A} \boldsymbol{x} \tag{3.6}$$

The computed steady solution $\boldsymbol{U}_0$ plays the role of base flow in the linearization for small perturbations of the equations 4.9, as follows.

$$\boldsymbol{u} = \boldsymbol{U}_0 + \boldsymbol{u}_\delta$$

$$p = p_0 + p_\delta$$

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t}_{\delta} = -Re\left(\left(\boldsymbol{U}_0 \cdot \boldsymbol{\nabla}\right)\boldsymbol{u}_\delta + \left(\boldsymbol{u}_\delta \cdot \boldsymbol{\nabla}\right)\boldsymbol{U}_0\right) + \nabla^2 \boldsymbol{u}_\delta - \boldsymbol{\nabla} p_\delta \\ 0 = \boldsymbol{\nabla} \cdot \boldsymbol{u}_\delta \end{cases} \tag{3.7}$$

Since the only term which has changed is the nonlinear term the weak form of the problem 3.6 is modified as follows.

```
Expr U = List( u0[0], u0[1] ) ;

Expr eqn3 = Integral( interior, - ( 1 / reynolds ) * ( ( grad * vx )
            * ( grad *ux ) + ( grad * vy ) * ( grad * uy ) ) + vx
            * ( U * grad ) * ux + vy * ( U * grad ) * uy + vx *
            ( u * grad ) * ux + vy * ( u * grad ) * uy - p *
            ( dx * vx + dy * vy ), quad2 ) ;

Expr eqn4 = Integral( interior, - ( dx * ux + dy * uy ) * q, quad2 ) ;

Expr sigma_M = Integral(interior, - sigma *
              ( vx*ux + vy*uy + 0.000001*q*p), quad2 );

Expr eqnEig = eqn3 + eqn4 + sigma_M ;
```

The left-hand terms form the mass matrix. A trick is employed to set to zero the term regarding the pressure, according to the formulation (3.7).

```
Expr massExpr = vx * ux + vy * uy + 0.000001 * q * p ;
```

Since the linearized eigenvalue problem is formulated in terms of a pertubation of velocity and pressure just homogeneous boundary conditions are applied.

```
Expr bcWallsEig = EssentialBC( walls, v * u, quad2 );
Expr bcInflowEig = EssentialBC( inflow, vx * ux + vy * uy, quad2 );
Expr bcOutflowEig = EssentialBC( outflow, q * p, quad2 );
Expr bcUpDownEig = EssentialBC( up_down, vy * uy, quad2 );

Expr bcEig = bcWallsEig + bcInflowEig + bcOutflowEig + bcUpDownEig;
```

The eigenvalue problem is built with a syntax which is similar to he one used for the nonlinear problem for the solution of the base flow. Sundance offers different ways of building the problem, and here a constructor which allows to input the mass matrix and the boundary conditions is employed.

```
LinearEigenproblem probEig( mesh, eqnEig, bcEig, massExpr,
   List( vx, vy, q ), List( ux, uy, p ), vecType, lumpedMass );
```

As already done, the eigenvalue solver is built reading parameters from an *eXtensible Markup Language* file (.xml) through the `ParameterXMLFileReader` object and then passed to the `Eigensolver` constructor. Finally the solution is computed calling the `solve_inverse` A.3 member function from the `LinearEigenproblem` object.

```
ParameterXMLFileReader readerEig( solverFileEig ) ;
ParameterList paramsEig = readerEig.getParameters() ;
ParameterList solverParamsEig = paramsEig.sublist( "Eigensolver" ) ;
Eigensolver<double> solverEig = new AnasaziEigensolver<double>(
solverParamsEig ) ;

Eigensolution solnEig = probEig.solve_inverse( solverEig ) ;
```

The reader could note that the called member function to solve the problem is just an inverse trasformation, instead of the shift-inverse one. The reason of this will soon explanined. It was impossible to subtract the operators $\mathcal{A}$ and $\mathcal{M}$ in the modified file of Sundance (see appendix A.3 for the complete code).

$$(\mathcal{A} - \sigma\mathcal{M})^{-1}\,\mathcal{M}\boldsymbol{x} = \nu\boldsymbol{x}$$

$$\mathcal{F}^{-1}\mathcal{M}\boldsymbol{x} = \nu\boldsymbol{x}$$

The shift σ is introduced directly in the weak formulation of the problem, in order to build the operator $\mathcal{F}$. This is simply done adding the expression `sigmaM` to the equations.

```
Expr sigmaM = Integral(interior, - sigma * massExpr, quad2);
Expr eqnEig = eqn3 + eqn4 + sigmaM ;
```

Since an inverse transformation is required to solve the initial eigenvalue problem, the eigenvalues of the original problem must be recovered by the following transformation.

$$\lambda = \sigma + \frac{1}{\nu}$$

```
for(int i = 0; i < solnEig.numEigenfunctions(); i++)
 {
   const std::complex<double>& ew = solnEig.eigenvalue(i);
   std::complex<double> eigval;
   eigval = solnEig.eigenvalue(i);
   eigval = sigma + (1.0 / solnEig.eigenvalue(i));
   cout << "ew=(" << eigval  << ")" << std::endl;
 }
```

The complete parameter list used to call the **Block Krylov-Schur** method provided by the Anasazi package to solve the non-hermitian eigenvalue problem is reported hereafter for convenience.

```
  <ParameterList>
    <ParameterList name="Eigensolver">
      <Parameter name="Method" type="string" value="Block Krylov Schur"/>
      <Parameter name="Number of Eigenvalues" type="int" value="10"/>
      <Parameter name="Block Size" type="int" value="1"/>
      <Parameter name="Num Blocks" type="int" value="100"/>
      <Parameter name="Verbosity" type="int" value="0"/>
      <Parameter name="Which" type="string" value="LM"/>
      <Parameter name="Use Preconditioner" type="bool" value="false"/>
      <Parameter name="Is Hermitian" type="bool" value="false"/>
      <Parameter name="Maximum Restarts" type="int" value="100"/>
      <Parameter name="Use Locking" type="bool" value="false"/>
      <Parameter name="Max Locked" type="int" value="1"/>
      <Parameter name="Convergence Tolerance" type="double" value=
"1.0e-06"/>
      <ParameterList name="Preconditioner">
         <Parameter name="Type" type="string" value="ML"/>
         <Parameter name="Problem Type" type="string" value="SA"/>
         <ParameterList name="ML Settings">
            <Parameter name="output" type="int" value="0"/>
         </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>
```

The boolean variable "Use Preconditioner" must be set to "false", since the Block Krylov-Schur method does not support any preconditioning. For this reason the lines regarding the ML preconditioner are ignored.
"Maximum restarts" is the maximum number of times that the subspace will be expanded to compute approximate eigenpairs to the desired precision, which is set with the "Convergence Tolerance" parameter. The Block Krylov-Schur method will keep the minimum multiple of the blocksize that is larger than the number of requested eigenvalues, and expand the subspace to the dimension obtained by multiplying "Num Blocks" and "Block Size". The desired approximated Ritz eigenpairs (determined by the "Which" parameter) are found for the expanded system and undesired eigenpairs are discarded, reducing the projected system back to the minimum multiple of the blocksize needed for the requested number of eigenpairs. More details about Krylov-Schur method are provided by Stewart [15] and its block version used in this job is in-depth described by Zhou et Saad [19].

3.2.7 Output

The last step described is the data output. Sundance provides different output file formats to visualize the computed solution or to extract numerical data, such

as VTK file or Matlab ".dat" file. Fields are added through the `FieldWriter` class and the `addField` member function. Desired fields should be listed as in the following example, in order to generate a VTK file.

```
  Expr expr_vector(List(u0[0], u0[1]));

  FieldWriter w = new VTKWriter("NavierStokesCylinder-" + Teuchos::
toString(reynolds));
  w.addMesh(mesh);
  w.addField("ux", new ExprFieldWrapper(u0[0]));
  w.addField("uy", new ExprFieldWrapper(u0[1]));
  w.addField("p", new ExprFieldWrapper(u0[2]));
  w.addField("vel", new ExprFieldWrapper(expr_vector));
  w.write();
```

Chapter 4

Results

This chapter 4 focuses on the numerical results obtained by the Trilinos code previously described. The aim is to validate the code with results available in literature. Two cases are analyzed with two different goals. The first one is the well-known case of the Lid-driven cavity flow in a square domain, and its aim is to compare the present results with the benchmark of Botella & Peyret [6] in order to validate the solution of the steady flow. Moreover the stability of the wake of a two-dimensional cylinder is investigated. Results show a good agreement with those obtained by Giannetti & Luchini [7] and confirm the correctness and accuracy of our solution for the eigenvalue problem associated to the linearized unsteady formulation presented in section 1.3.
In section 4.1 both the strong and the weak formulation of the problem and their boundary conditions are introduced, and then we show the computational grid employed. Numerical results are finally reported.

4.1 Lid-driven cavity flow

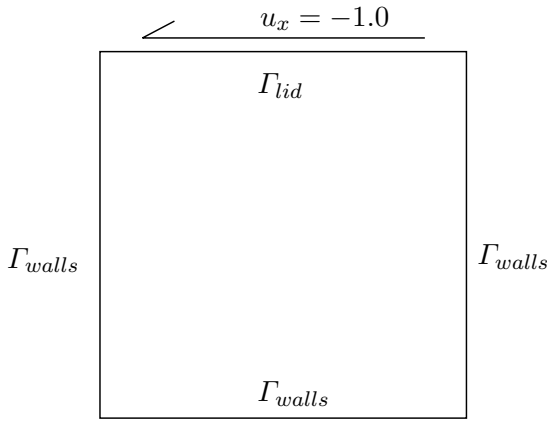


Figure 4.1: Lid-driven cavity geometry sketch, side length $L = 1.0$

4.1.1 Problem formulation

This paragraph introduces the reader to the the lid-driven cavity flow problem. A square domain is employed and a uniform horizontal velocity is enforced on the upper edge Γ_{lid}. We start from the steady Navier-Stokes equations

$$\begin{cases} (\boldsymbol{u}\cdot\boldsymbol{\nabla})\,\boldsymbol{u} - \dfrac{1}{Re}\nabla^2\boldsymbol{u} + \boldsymbol{\nabla}p = 0 \\ \boldsymbol{\nabla}\cdot\boldsymbol{u} = 0 \end{cases} \tag{4.1}$$

and then we write their scalar version useful to obtain the weak form, which is necessary for the finite element formulation.

$$\begin{cases} u_x\dfrac{\partial u_x}{\partial x} + u_y\dfrac{\partial u_x}{\partial y} = -\dfrac{\partial p}{\partial x} + \dfrac{1}{Re}\nabla^2 u_x \\ u_x\dfrac{\partial u_y}{\partial x} + u_y\dfrac{\partial u_y}{\partial y} = -\dfrac{\partial p}{\partial y} + \dfrac{1}{Re}\nabla^2 u_y \\ \dfrac{\partial u_x}{\partial x} + \dfrac{\partial u_y}{\partial y} = 0 \end{cases} \tag{4.2}$$

The partial differential equations are supplemented by the following boundary conditions. The no-slip condition is as usual applied to walls Γ_{walls}, while the upper edge slides from right to left as a lid. A condition of uniform velocity is on the lid employed.

$$u_x = 1, \quad u_y = 0, \quad \text{on } \Gamma_{lid} \tag{4.3a}$$
$$u_x = 0, \quad u_y = 0, \quad \text{on } \Gamma_{walls} \tag{4.3b}$$

The weak form is obtained by multiplying the Navier-Stokes equations by a test function and integrating by parts as follows:

$$\begin{cases} -\displaystyle\int_\Gamma \frac{1}{Re}\left(\frac{\partial u_x}{\partial n}\right)v_x + \int_\Omega \frac{1}{Re}\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}v_x + \int_\Omega \left(u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y}\right)v_x + \int_\Omega \frac{\partial p}{\partial x}v_x = 0 \\ -\displaystyle\int_\Gamma \frac{1}{Re}\left(\frac{\partial u_y}{\partial n}\right)v_y + \int_\Omega \frac{1}{Re}\boldsymbol{\nabla}u_y\cdot\boldsymbol{\nabla}v_y + \int_\Omega \left(u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y}\right)v_y + \int_\Omega \frac{\partial p}{\partial y}v_y = 0 \\ \displaystyle\int_\Omega \left(\frac{\partial u_x}{\partial x} + \frac{\partial U_y}{\partial y}\right)q = 0 \end{cases} \tag{4.4}$$

Since Dirichlet boundary conditions are imposed on velocity, the corresponding test functions annihilate there, therefore the boundary integrals are all zero.

$$\int_{\Gamma_{lid}} (u_x - 1.0)\, v_x = 0 \qquad \int_{\Gamma_{lid}} u_y v_y = 0, \quad \text{on } \Gamma_{lid} \tag{4.5a}$$

$$\int_{\Gamma_{walls}} u_x v_x = 0 \qquad \int_{\Gamma_{walls}} u_y v_y = 0, \quad \text{on } \Gamma_{walls} \tag{4.5b}$$

The formulation (4.4) becomes:

$$\begin{cases} \displaystyle\int_\Gamma \frac{1}{Re} \boldsymbol{\nabla} u_x \cdot \boldsymbol{\nabla} v_x + \int_\Gamma \left(u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} \right) v_x + \int_\Gamma \frac{\partial p}{\partial x} v_x = 0 \\ \displaystyle\int_\Gamma \frac{1}{Re} \boldsymbol{\nabla} u_y \cdot \boldsymbol{\nabla} v_y + \int_\Gamma \left(u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} \right) v_y + \int_\Gamma \frac{\partial p}{\partial y} v_y = 0 \\ \displaystyle\int_\Gamma \left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} \right) q = 0 \end{cases} \tag{4.6}$$

4.1.2 Computational grid

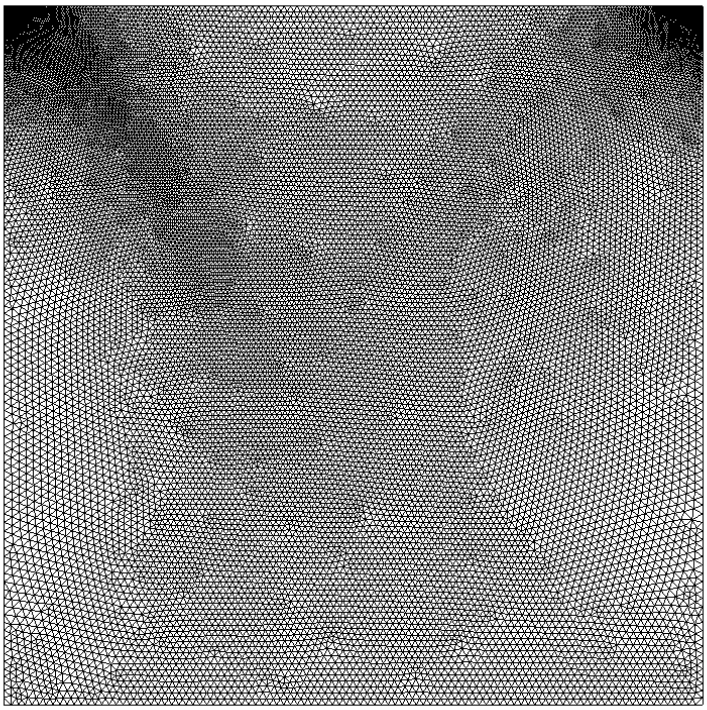


Figure 4.2: Mesh domain used for cavity problem

It is known that the lid-driven cavity problem formulated in primitive variables presents two singularity points in the corners where the vertical walls intersect the sliding lid. We must take this into account when the mesh geometry is created. **Ansys ICEM Cfd** mesh generator was employed to obtain the computational mesh shown in figure 4.2. Suitable stretching laws were used along the cavity walls to refine the size of the mesh elements in critical points. Table 4.1 shows in detail how the mesh shown in figure 4.2 was obtained, while parameters involved in the use of hyperbolic grid refinement law are listed below. S_i is the spacing of the i-th element on the boundary, while the only parameters left to the control of the user are the spacings of both the first and the last element along the edge, $Sp1$ and $Sp2$, and the number on nodes N to employ on each side. The length of the edge is represented by b. It is to be noted that in vertical edges $Sp1$ is above $Sp2$. The total number of triangles is 40739.

edge	law	n. of nodes	Sp1	Sp2
top	hyperbolic	200	0.001	0.001
left	hyperbolic	140	0.001	0.010
right	hyperbolic	140	0.001	0.010
bottom	uniform	101	0.010	0.010

Table 4.1: Square cavity mesh parameters

$$S_i = \frac{U_i}{2 \cdot A + (1 - A)\, U_i} \tag{4.7a}$$

$$U_i = 1 + \frac{\tanh\left(b - R_i\right)}{\tanh\left(\frac{b}{2}\right)} \tag{4.7b}$$

$$R_i = \frac{i - 1}{N - 1} - \frac{1}{2} \tag{4.7c}$$

$$A = \sqrt{\frac{Sp1}{Sp2}} \tag{4.7d}$$

$$\sinh b = \frac{b}{(N - 1) \cdot \sqrt{Sp1 \cdot Sp2}} \tag{4.7e}$$

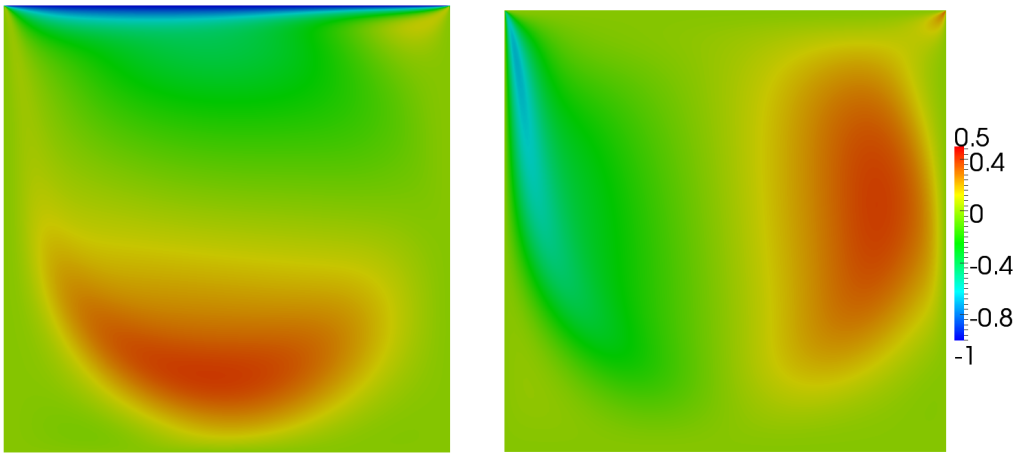
4.1.3 Numerical results

In this paragraph numerical results for the lid-driven cavity flow problem are presented. Our aim is to compare results produced by Sundance to reference data available in literature. In particular, our numerical results are compared to those reported by Botella & Peyret [6] who provided very accurate benchmark data with a desingularized spectral Navier-Stokes solver formulated in primitive variable. Vorticity levels are compared to those reported by Auteri, Quartapelle & Vigevano [3], who developed a desingularized spectral for the ω-ψ formulation; an excellent agreement in the "eye-ball norm" was found between the vorticity levels computed with our $\boldsymbol{u}$-p formulation using equation (4.8) and those reported in [3].

$$\omega = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \tag{4.8}$$

Figure 4.3 illustrates the velocity fields computed for the horizontal and vertical components. The boudary condition is properly satisfied on the sliding lid, as well as on the walls where both velocities are zero, as expected. For what concerns the

velocity field a change of direction can be noticed in the lower region of the domain. This represents correctly the recirculation region, where the mass conservation coupled with the presence of walls induces the fluid to flow upstream. The figure on the right shows how vertical component of the field is influenced by the movement of the lid. An upward flow is triggered in the right-hand side of the domain, as opposed to what happens on the left-hand side.
Figure 4.4 illustrates the circulating motion established in the cavity due to the sliding of the upper lid. All figures are referred to a Reynolds number of 1000.



(a) Horizontal velocity component u_x (b) Vertical velocity component u_y

Figure 4.3: Lid-driven cavity flow: velocity components at Re = 1000

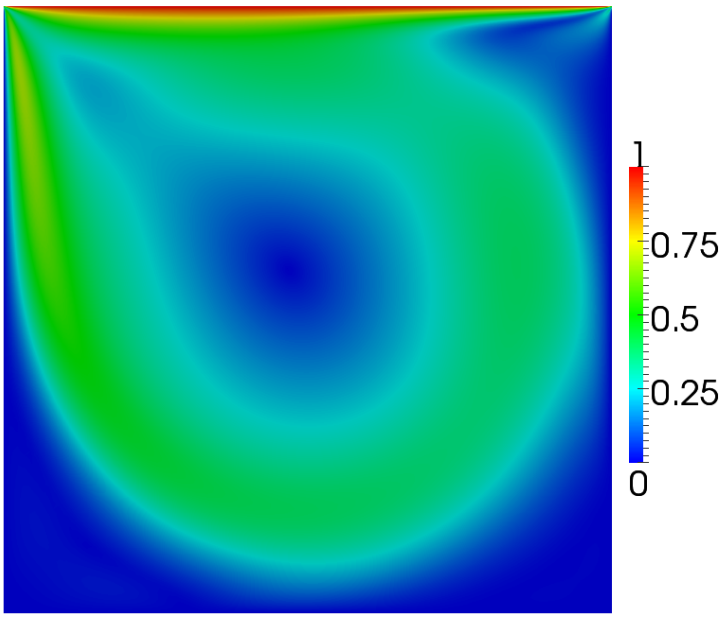


Figure 4.4: Lid-driven cavity flow: velocity magnitude $||\boldsymbol{u}||$ at Re = 1000

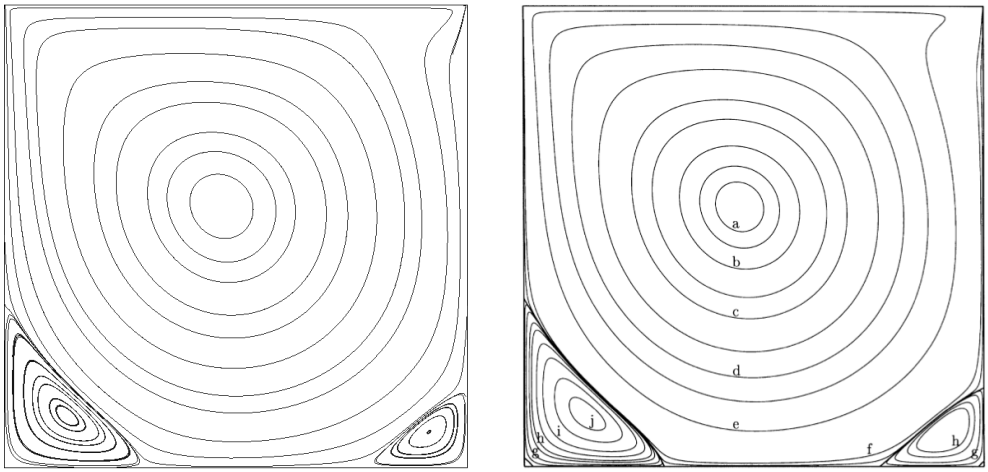
y	u_x, Ref. [6]	u_x
1.0000	-1.000	-1.000
0.9766	-0.664	-0.664
0.9688	-0.581	-0.581
0.9609	-0.517	-0.506
0.9531	-0.472	-0.469
0.8516	-0.337	-0.337
0.7344	-0.189	-0.194
0.6172	-0.057	-0.057
0.5000	0.062	0.062
0.4531	0.108	0.108
0.2813	0.280	0.280
0.1719	0.388	0.387
0.1016	0.300	0.300
0.0703	0.223	0.223
0.0625	0.202	0.204
0.0547	0.181	0.181
0.0000	0.000	0.000

Table 4.2: Lid-driven cavity flow: horizontal velocity component through the vertical centerline at Re = 1000

x	u_y, Ref. [6]	u_y
0.0000	0.000	0.000
0.0312	-0.228	-0.233
0.0391	-0.294	-0.291
0.0469	-0.355	-0.357
0.0547	-0.410	-0.410
0.0947	-0.526	-0.527
0.1406	-0.426	-0.427
0.1953	-0.320	-0.311
0.5000	0.025	0.025
0.7656	0.325	0.325
0.7734	0.334	0.334
0.8437	0.377	0.377
0.9062	0.333	0.326
0.9219	0.310	0.310
0.9297	0.296	0.289
0.9375	0.281	0.281
1.0000	0.000	0.000

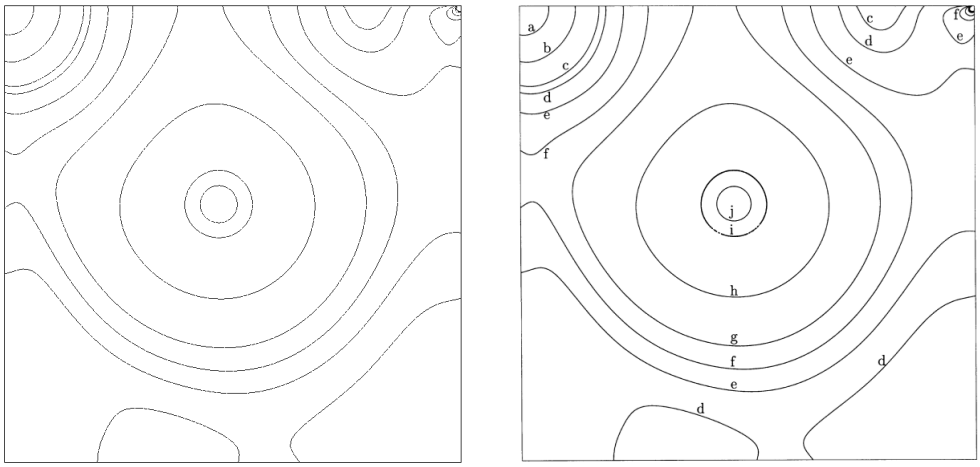
Table 4.3: Lid-driven cavity flow: vertical velocity component through the horizontal centerline at Re = 1000

A direct comparison with the reference results is shown in tables 4.2 and 4.3. The horizontal velocity component along the vertical centerline and the vertical component along the horizontal centerline are reported respectively. The computed data show a quite close agreement with the results of Botella & Peyret [6] at a Reynolds number of 1000. Numerical values of pressure are not included in the quantitative analysis because it is known that pressure is defined modulo an additive constant. However, a qualitative analysis of the trend could be useful to understand to what extent is the computed solution correct. Figure 4.6 shows the computed isobar lines compared to those reported in [6]: an excellent match can be noted.
In figure 4.5 streamlines are illutrated. The Paraview *stream tracer* tool allows the user to draw the streamlines from a line source, as in this case, or from a point source. Also in this case a qualitative comparison with [6] confirms the correctness of the computed solution for the lid-driven cavity problem. Finally, vorticity levels are printed. Figure 4.7 shows a quantitative comparison of the results computed with Sundance with those reported in [3] for a Reynolds number of 1000. As it can be seen, they agree. Level values are reported in the caption.



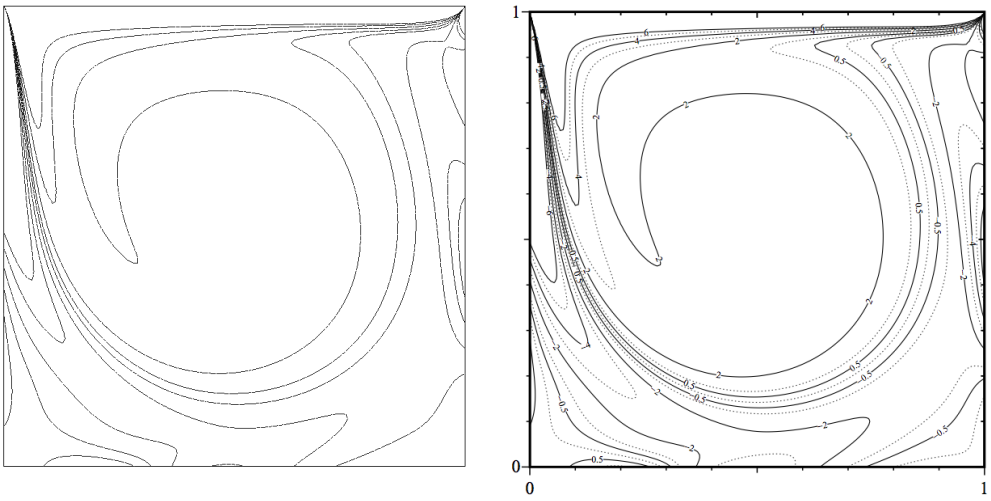
(a) Present results (b) Botella & Peyret (1998), [6]

Figure 4.5: Lid-driven cavity flow: streamline comparison



(a) Present results (b) Botella & Peyret (1998), [6]

Figure 4.6: Lid-driven cavity flow: isobar line comparison



(a) Present results (b) Auteri, Quartapelle & Vigevano (2002), [3]

Figure 4.7: Lid-driven cavity flow: vorticity levels. Values: -6, -4, -2, -0.5, 0.5, 2, 4, 6 (corresponding to the thick lines in the reference plot)

4.2 Stability analysis of 2D cylinder wake

Once the steady solver is validated, the case of a two-dimensional cylinder in cross-flow is investigated. The aim is to compute the eigenvalues of the linearized problem in order to study the linear stability of the flow. To reach this target we have to compute the, possibly unstable, steady solution of the Navier-Stokes equations, which represents the base flow for the unsteady linearized problem described in section 1.3.

4.2.1 Problem formulation

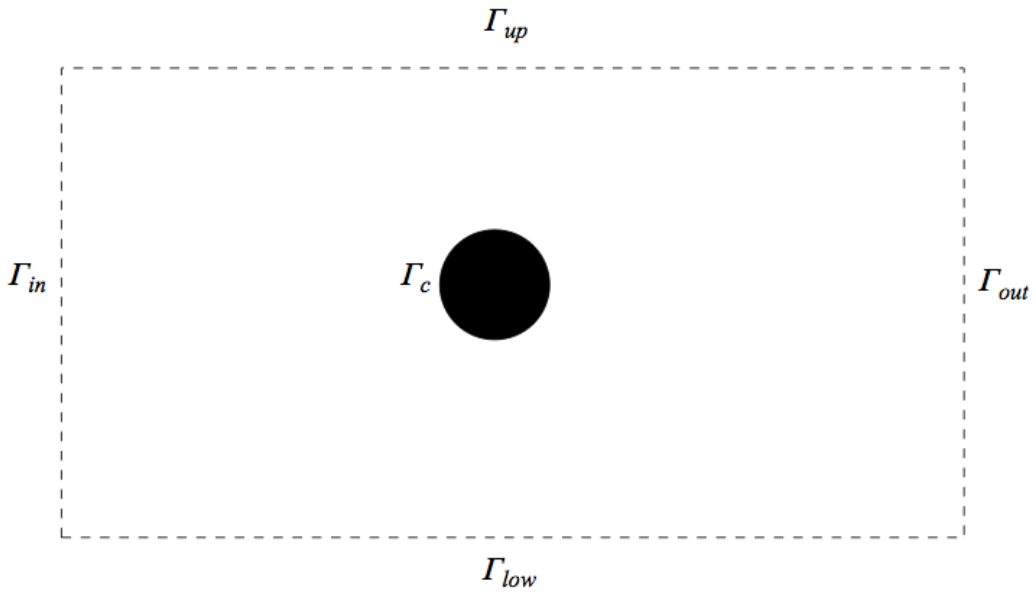


Figure 4.8: Computational domain

To obtain the base flow, the Navier-Stokes equations

$$\begin{cases} (\boldsymbol{u}\cdot\boldsymbol{\nabla})\,\boldsymbol{u} - \dfrac{1}{Re}\nabla^2\boldsymbol{u} + \boldsymbol{\nabla}p = 0 \\ \boldsymbol{\nabla}\cdot\boldsymbol{u} = 0 \end{cases} \tag{4.9}$$

have to be solved. The field force is neglected in this discussion, while the velocity time derivative is zero. The following boundary conditions have been enforced to compute the base flow:

$$u_x = 1, \quad u_y = 0 \quad \text{on } \Gamma_{in} \tag{4.10a}$$

$$\frac{\partial u_x}{\partial y} = 0, \quad u_y = 0 \quad \text{on } \Gamma_{up} \cup \Gamma_{low} \tag{4.10b}$$

$$\frac{\partial u_x}{\partial x} = 0, \quad p = 0 \quad \text{on } \Gamma_{out} \tag{4.10c}$$

$$u_x = 0, \quad u_y = 0 \quad \text{on } \Gamma_c \tag{4.10d}$$

The no-slip boundary condition (4.10d) is imposed on the cylinder surface, while different conditions are enforced on the external boundary of the rectangular computational domain. A uniform free stream horizontal velocity is enforced (4.10a) on the inflow edge, symmetry boundary conditions are enforced on the upper and lower sides (4.10b), while pressure and the streamwise derivative of the horizontal velocity components are set to zero at outflow edge (4.10c). A homogeneous Neumann boundary condition is enforced on the horizontal component of the velocity at the outlet boundary and for this reason normal derivative terms do not appear in equations (4.11). The weak form is obtained by multiplying the equations (4.2) by the test functions (v_x, v_y, q) and integrating by parts.

$$\begin{cases} \displaystyle\int_\Gamma \frac{1}{Re} \boldsymbol{\nabla} u_x \cdot \boldsymbol{\nabla} v_x + \int_\Gamma \left(u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} \right) v_x + \int_\Gamma \frac{\partial p}{\partial x} v_x = 0 \\ \displaystyle\int_\Gamma \frac{1}{Re} \boldsymbol{\nabla} u_y \cdot \boldsymbol{\nabla} v_y + \int_\Gamma \left(u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} \right) v_y + \int_\Gamma \frac{\partial p}{\partial y} v_y = 0 \\ \displaystyle\int_\Gamma \left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} \right) q = 0 \end{cases} \tag{4.11}$$

The boundary conditions in their weak form are formulated in a similar way.

$$\int_{\Gamma_{in}} (u_x - 1.0)\, v_x = 0, \quad \int_{\Gamma_{in}} u_y v_y = 0, \quad \text{on } \Gamma_{in} \tag{4.12a}$$

$$\int_{\Gamma_{up}} u_y v_y = 0, \quad \int_{\Gamma_{low}} u_y v_y = 0, \quad \text{on } \Gamma_{up} \cup \Gamma_{low} \tag{4.12b}$$

$$\int_{\Gamma_{out}} pq = 0, \quad \text{on } \Gamma_{out} \tag{4.12c}$$

$$\int_{\Gamma_c} u_x v_x = 0, \quad \int_{\Gamma_c} u_y v_y = 0, \quad \text{on } \Gamma_c \tag{4.12d}$$

As shown in Section 1.3, the linearization of the Navier-Stokes equations (4.9) leads to a non-symmetric, generalized eigenvalue problem. Velocity and pressure can be decomposed in the sum of two contributions: the solution $(\boldsymbol{U}_0, P_0)$ of the steady problem and a small perturbation $(\boldsymbol{u}_\delta, p_\delta)$. After dropping quadratic terms, the following system of linear equations is obtained in strong form:

$$\begin{cases} \dfrac{\partial \boldsymbol{u}_\delta}{\partial t} + \left(\left(\boldsymbol{U}_0 \cdot \boldsymbol{\nabla}\right) \boldsymbol{u}_\delta + \left(\boldsymbol{u}_\delta \cdot \boldsymbol{\nabla}\right) \boldsymbol{U}_0\right) - \dfrac{1}{Re} \nabla^2 \boldsymbol{u}_\delta + \boldsymbol{\nabla} p_\delta = 0 \\ \boldsymbol{\nabla} \cdot \boldsymbol{u}_\delta = 0 \end{cases} \tag{4.13}$$

after reformulating this problem in weak form and discretizing it, we obtain

$$\lambda \boldsymbol{M} \boldsymbol{v} = \boldsymbol{A} \boldsymbol{v} \tag{4.14}$$

As explained in section 1.3, a spectral transformation is employed to solve the eigenvalue problem, and this leads to the final expression of the shift-invert problem used to compute the perturbation of pressure and velocity.

$$\left(\boldsymbol{A} - \sigma \boldsymbol{M}\right)^{-1} \boldsymbol{M} \boldsymbol{v} = \nu \boldsymbol{v} \tag{4.15}$$

According to [7] homogeneous boundary conditions of the same kind as those employed for the base flow are enforced to obtain the eigenproblem:

$$u_{x,\delta} = 0, \quad u_{y,\delta} = 0 \quad \text{on } \Gamma_{in} \tag{4.16a}$$

$$\frac{\partial u_{x,\delta}}{\partial y} = 0, \quad u_{y,\delta} = 0 \quad \text{on } \Gamma_{up} \cup \Gamma_{low} \tag{4.16b}$$

$$\frac{\partial u_{x,\delta}}{\partial x} = 0, \quad p_\delta = 0 \quad \text{on } \Gamma_{out} \tag{4.16c}$$

$$u_{x,\delta} = 0, \quad u_{y,\delta} = 0 \quad \text{on } \Gamma_{c} \tag{4.16d}$$

4.2.2 Computational grid

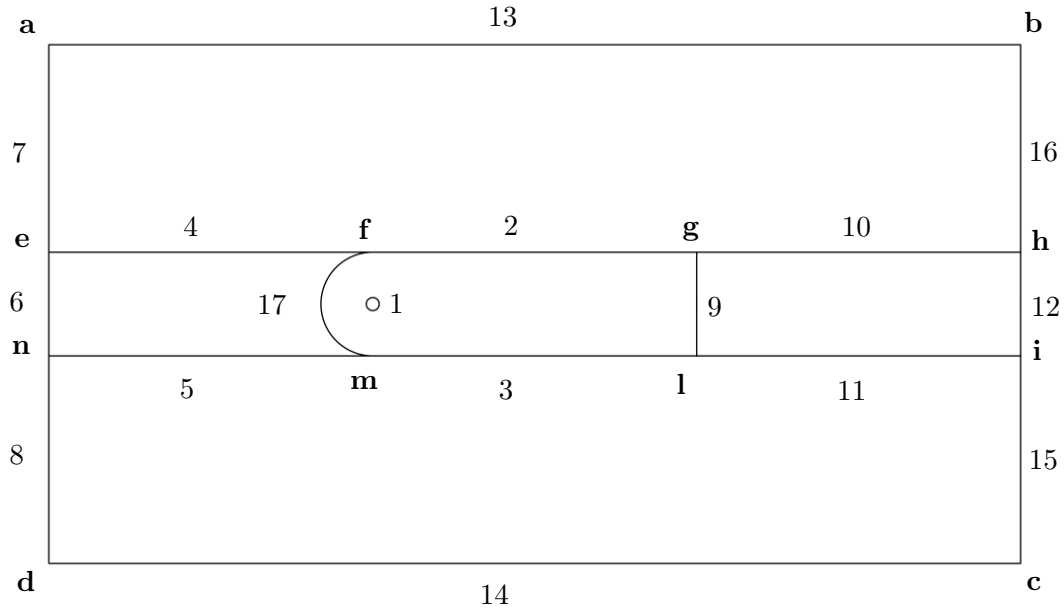


Figure 4.9: Cylinder geometry sketch

The mesh geometry was generated by assigning the maximum spacing between adjacent nodes and/or the number of nodes along each side, which are defined by points from **a** to **m**, using a commercial mesh generator. Table 4.4 reports the coordinates of all points. The size of the computational domain is the same employed by Giannetti & Luchini in [7].

point	x	y
a	-25.0	20.0
b	50.0	20.0
c	50.0	-20.0
d	-25.0	-20.0
e	-25.0	4.0
f	0.0	4.0
g	25.0	4.0
h	50.0	4.0
i	50.0	-4.0
l	25.0	.4.0
m	0.0	-4.0
n	-25.0	-4.0

Table 4.4: Coordinates of the auxiliary points used to build the mesh

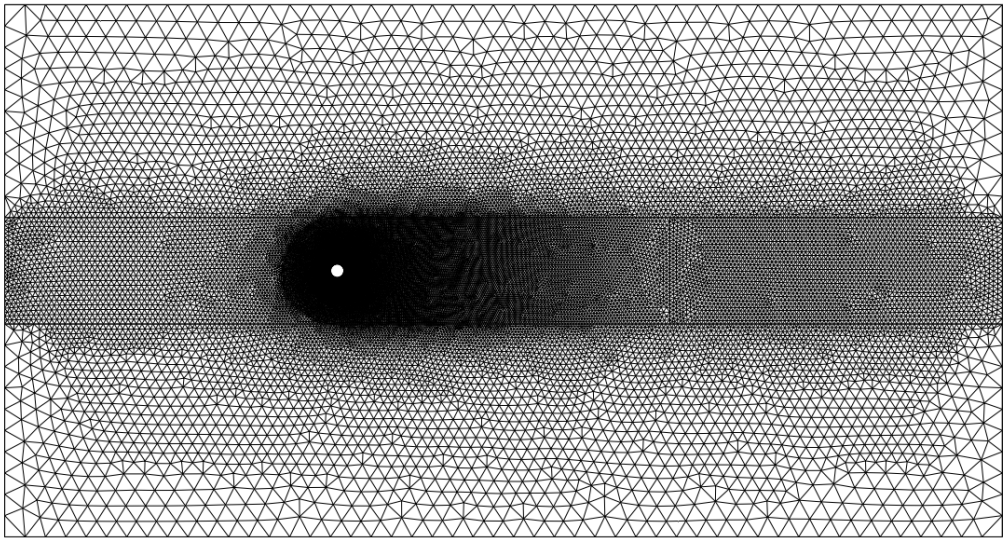


Figure 4.10: Cylinder mesh

Figure 4.10 presents the unstructured mesh used for the computation. It was created with the commercial software **Ansys IcemCFD**. The mesh is composed of triangles, and the total number of elements is 55918. Table 4.5 reports all information needed to generate the mesh. See equations 4.7 for the meaning of the $Sp1$ and $Sp2$ parameters.

edge	law	n. of nodes	Sp1	Sp2
1	uniform	100	0.03173	0.03173
2	hyperbolic	150	0.12823	0.30000
3	hyperbolic	150	0.12823	0.30000
4	hyperbolic	80	0.12823	0.35000
5	hyperbolic	80	0.12823	0.35000
6	uniform	28	0.30000	0.30000
7	uniform	11	1.53061	1.53061
8	uniform	11	0.53061	1.53061
9	uniform	28	0.30000	0.30000
10	hyperbolic	80	0.30000	0.40000
11	hyperbolic	80	0.30000	0.40000
12	uniform	21	0.40000	0.40000
13	uniform	50	1.53061	1.53061
14	uniform	50	1.53061	1.53061
15	uniform	11	1.53061	1.53061
16	uniform	11	1.53061	1.53061
17	uniform	100	0.12823	0.12823

Table 4.5: Cylinder mesh parameters, refer to equations 4.7

4.2.3 Numerical results

The numerical procedure described in the previous section was used to evaluate the steady flow. Calculations were performed on the grid shown in section 4.2.2. Numerical results are reported below, first for the steady problem, and then for the eigenvalue problem.
Figures 4.11 and 4.12 show the velocity field around the two-dimensional cylinder. A recirculation region can be seen in the cylinder wake where velocity magnitude slightly tends to increase in the x direction. Velocity increases correctly along the upstream side of the cylinder due to the geometry. Flow velocity decreases along the rear side of the cylinder while pressure tends to increase, as expected. The front stagnation point is clearly visible in figure 4.12 at the leading edge of the cylinder and the horizontal streamline confirms its correct position. The recirculation region within the wake near the rear side of the cylinder is highlighted in figure 4.14. Figures 4.15 and 4.16 illustrate the vertical velocity component. A rapid increase of velocity modulus can be seen near the upstream side of the cylinder, where the flow is deflected because of the geometry. A different behavior can be noted downstream, where the horizontal velocity is negative because the flow is separated, as figure 4.18 shows. Isobar lines are provided in figure 4.17 and it can be seen how separation influences the downstream spatial distribution of pressure. The solution agrees with the enforced condition of zero-pressure at outflow boundary. Finally figure 4.19 shows the dependence of the wake length on Reynolds number. As expected the bubble length increases with Reynolds number and figure 4.20 compares the present results to those obtained by Giannetti & Luchini [7].

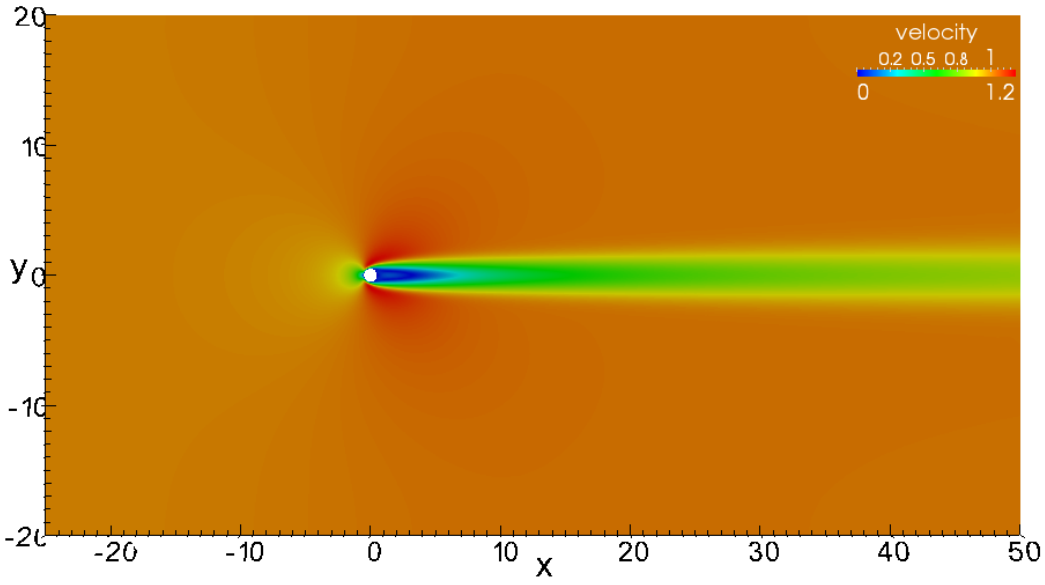


Figure 4.11: Cylinder flow: velocity magnitude

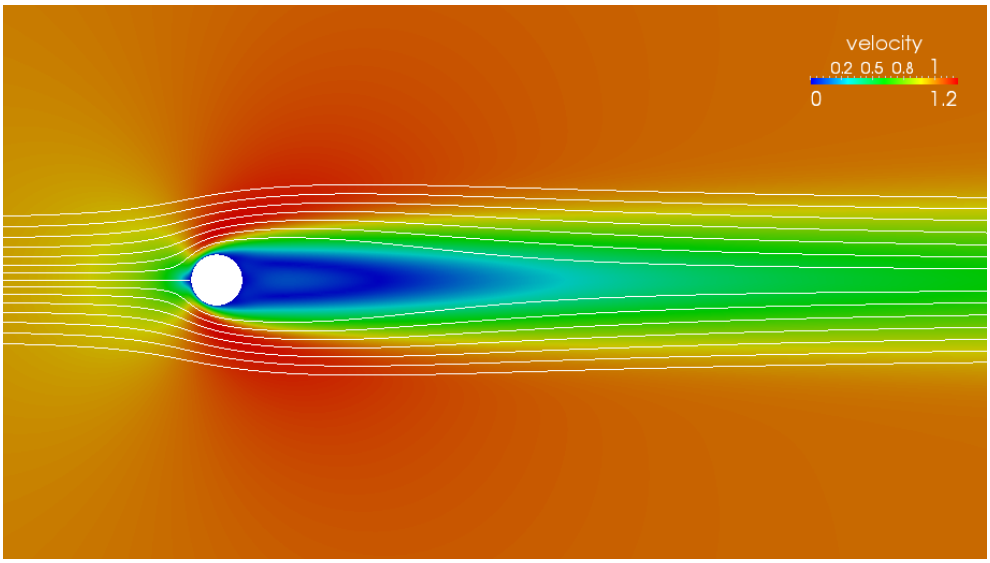


Figure 4.12: Cylinder flow: velocity magnitude and streamlines near the cylinder

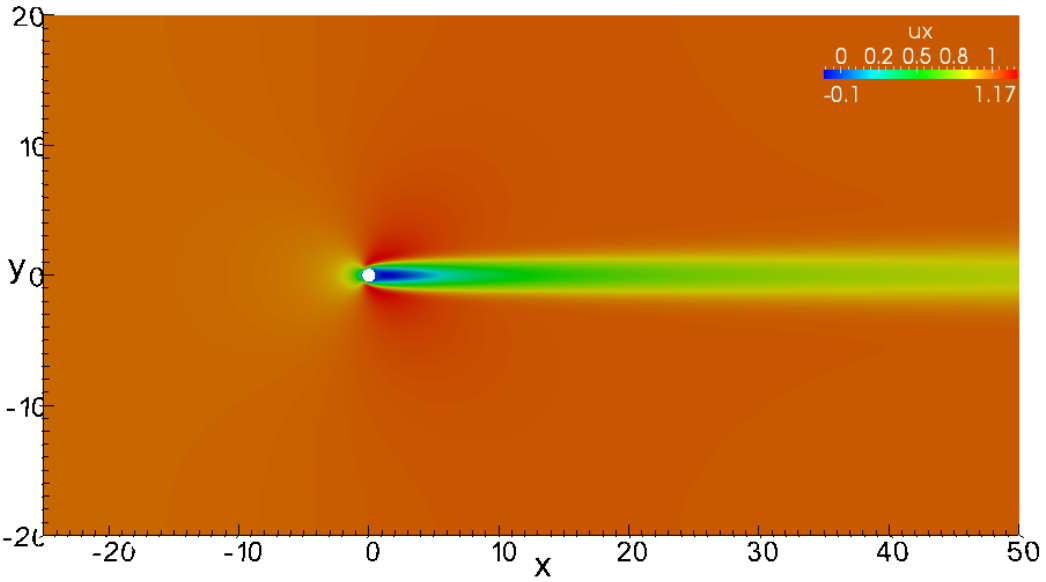


Figure 4.13: Cylinder flow: horizontal component of the velocity field

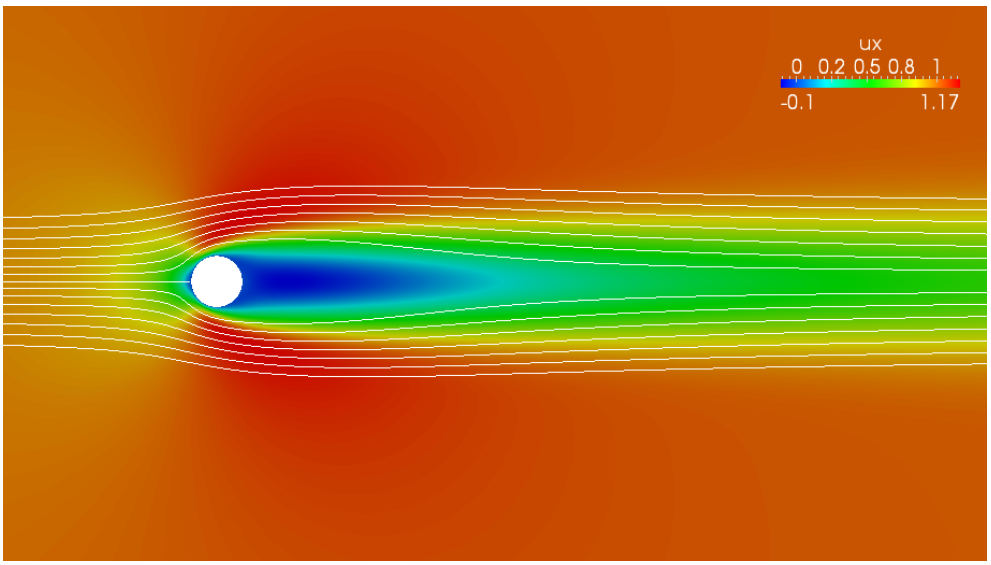


Figure 4.14: Cylinder flow: horizontal component of the velocity field and streamlines near the cylinder

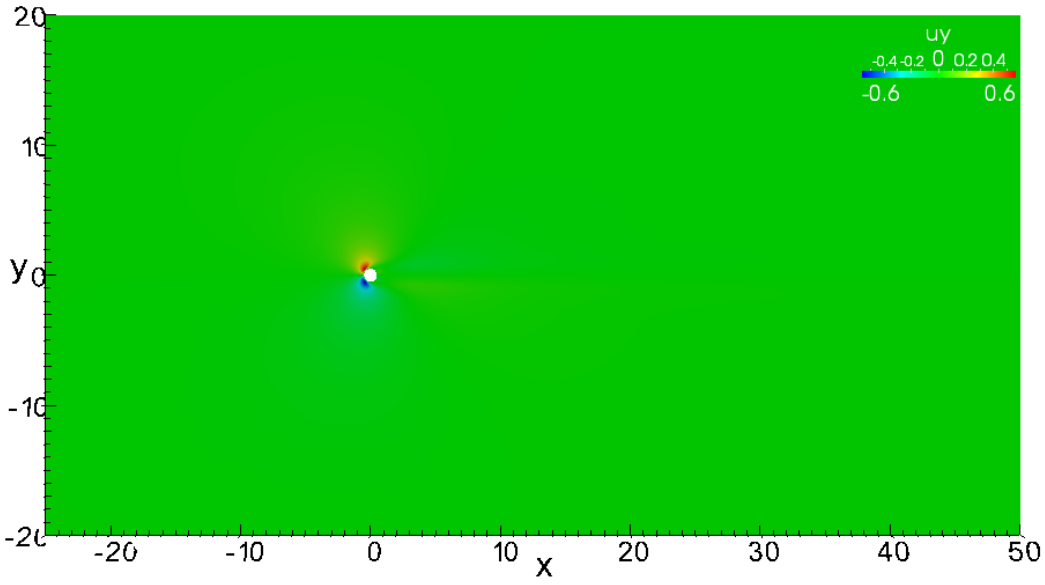


Figure 4.15: Cylinder flow: vertical component of the velocity field

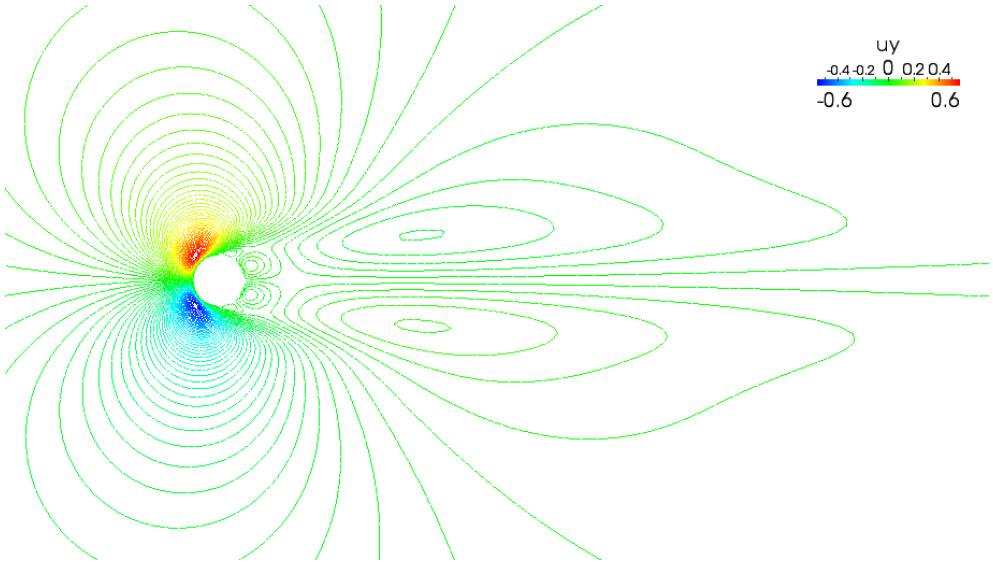


Figure 4.16: Cylinder flow: vertical component of the velocity field near the cylinder

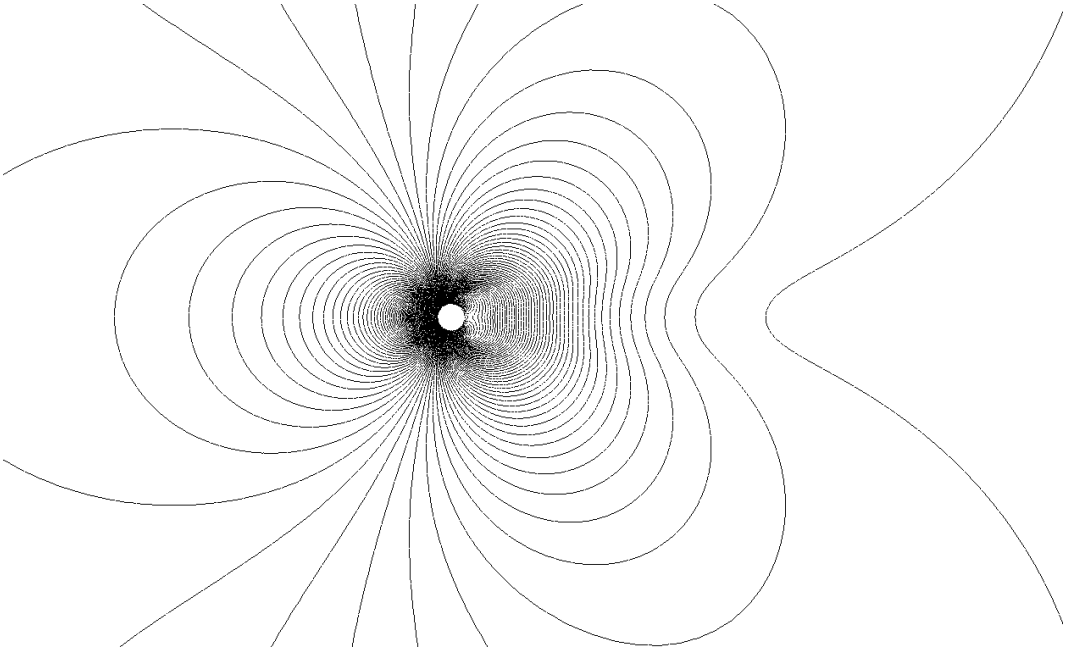


Figure 4.17: Cylinder flow: pressure field

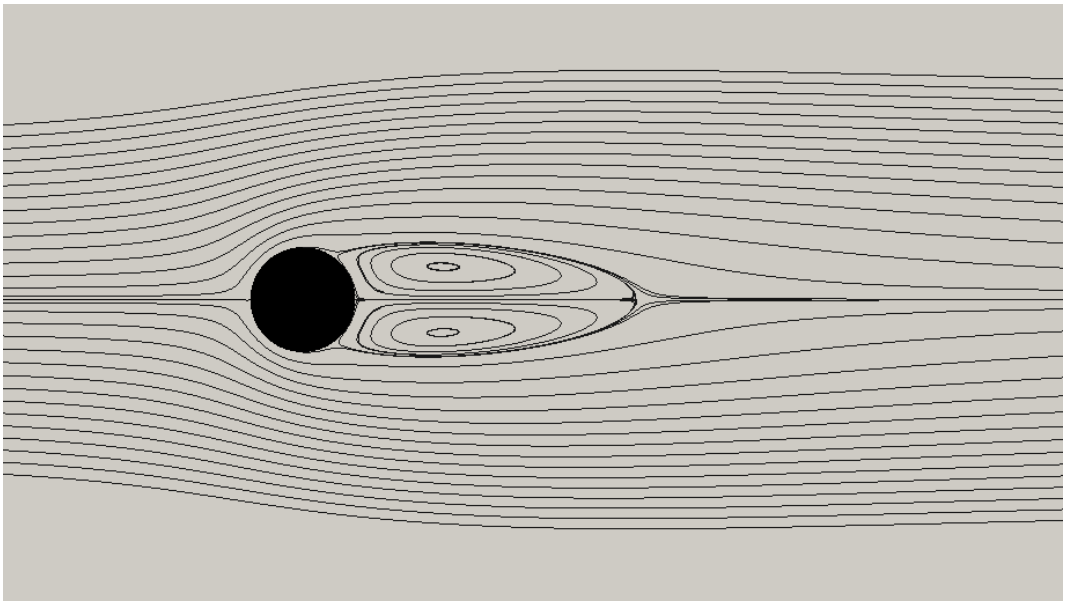
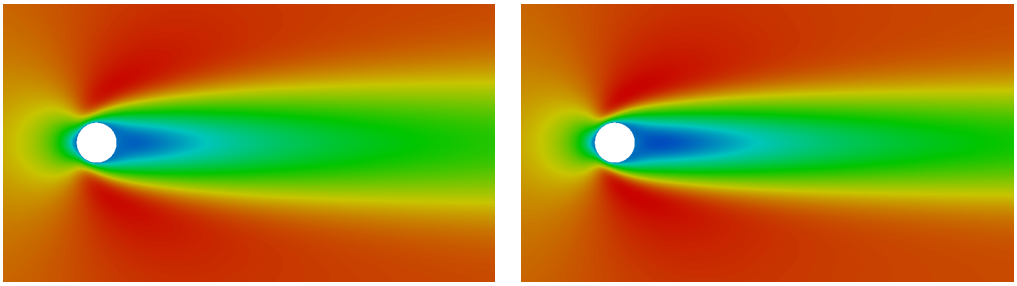
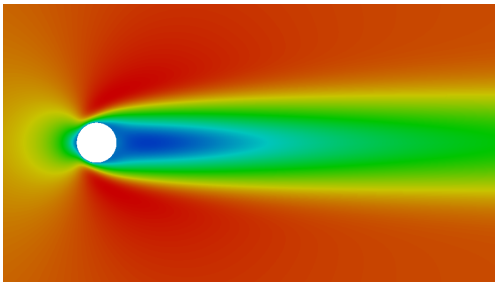


Figure 4.18: Cylinder flow: streamlines

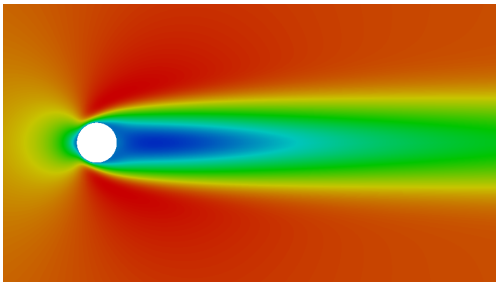


(a) $Re = 20$

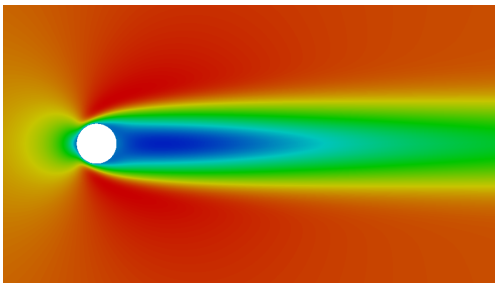
(b) $Re = 30$



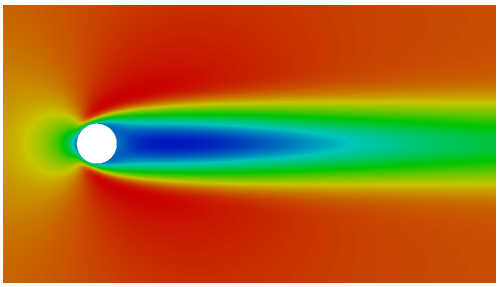
(c) $Re = 40$



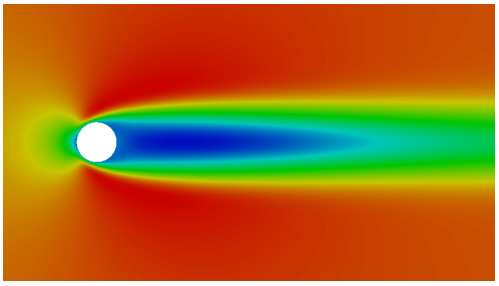
(d) $Re = 50$



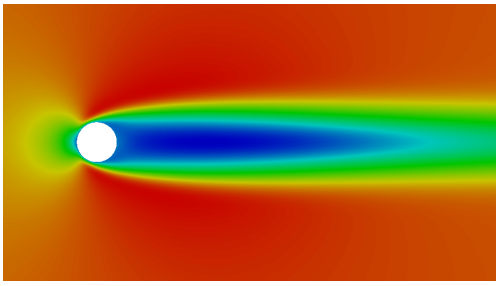
(e) $Re = 60$



(f) $Re = 70$



(g) $Re = 80$



(h) $Re = 100$

Figure 4.19: Cylinder flow: wake length dependence on Reynolds number

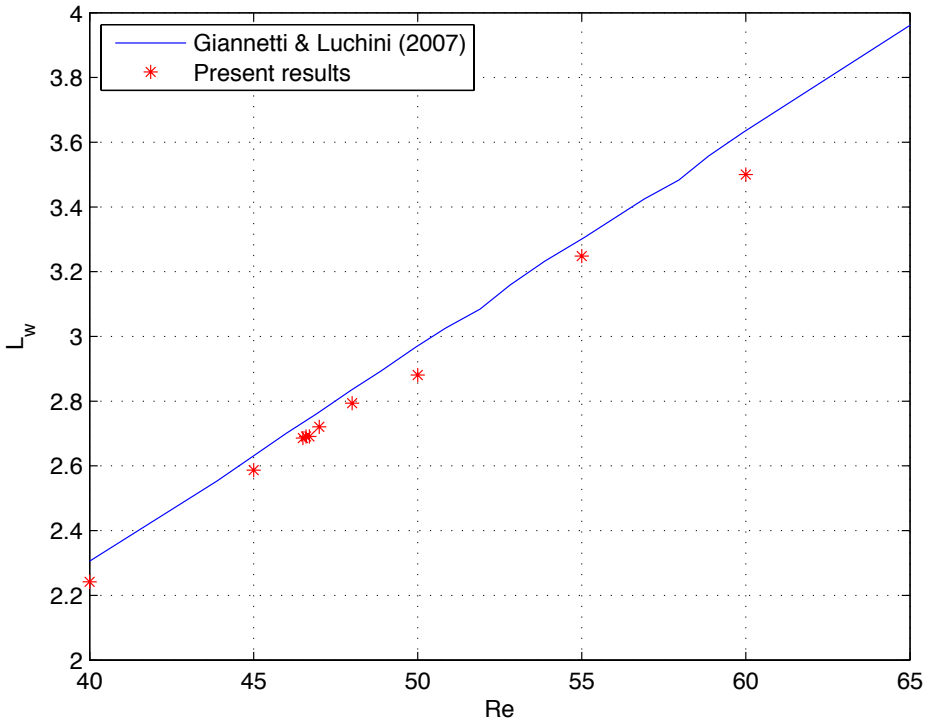


Figure 4.20: Cylinder flow: wake bubble length

The stability characteristics of the flow are assesed by monitoring the behavior of the most unstable mode of the linearized equations of motion. Since the matrix of the linearized problem is real the complex modes of this problem occur in conjugate pairs. The critical Reynolds number starting from which on the steady flow becomes unstable was estimated to be equal to 46.7 by Giannetti & Luchini [7], and 46.6 by Sipp & Lebedev [14]. According to our calculation the onset of the first unstable mode occurs at $Re_c = 46.5$. This result substantially agrees with the reference results. In the experiments provided by Williamson (1996) [18] a threshold of 47 was observed. Figure 4.21 shows the computed eigenvalue spectrum for three Reynolds numbers in the neighborhood of the computed threshold of 46.5. As expected, by increasing the Reynolds number a shift of all eigenvalues is noted, and the eigenvalue having the largest real part moves to the right-hand side of the complex plane. Thus the flow becomes unstable, and the computed critical eigenvalue computed is $\sigma = (0.000172 \pm j0.745)$: this is in line with results obtained by Sipp & Lebedev [14] (2007) who reported an imaginary part of 0.74 in the neighborhood of the imaginary axis, as shown in figure 4.23. Several clusters of eigenvalues are to be noted in the reported pictures. Differences between the present ones and those reported in [14] could depend on different geometry size, mesh characteristics and employed boundary condition at outflow and on the

non-normality of the problem. Figure 4.22 shows the amplification rate $Re(\sigma)$ and the Strouhal number $St = Im(\sigma_1)/2\pi$ for the most unstable mode. The relation between Reynolds and Strouhal number is in agreement with results reported by Giannetti & Luchini [7]. However, it is important to note that only in the neighborhood of the critical point the predicted Strouhal number is in good agreement with the experimental data of Williamson (1996). In fact the linear theory is unable to predict the real vortex-shedding frequency in the unstable regime far from the critical point, where nonlinear effects become important and the relationship $Re - St$ shown in figure 4.22 is no longer accurate.
Finally, figure 4.26 shows the vertical component of the real part of the most unstable eigenfunction. It is self-evident the good agreement with the picture taken from Sipp & Lebedev (2007) [14]. The effects of Reynolds number parameter on the horizontal velocity and pressure modes are presented in figures 4.24 and 4.25.

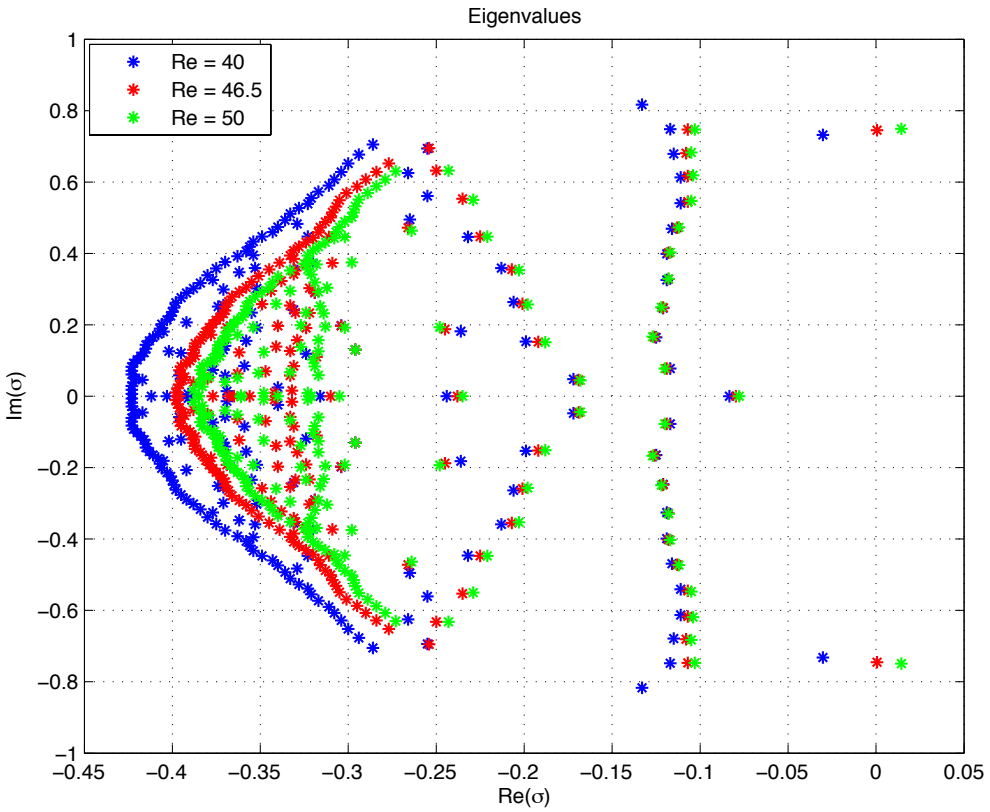


Figure 4.21: Cylinder flow: eigenvalue spectrum for three different Reynolds numbers (40, 46.5, 50)

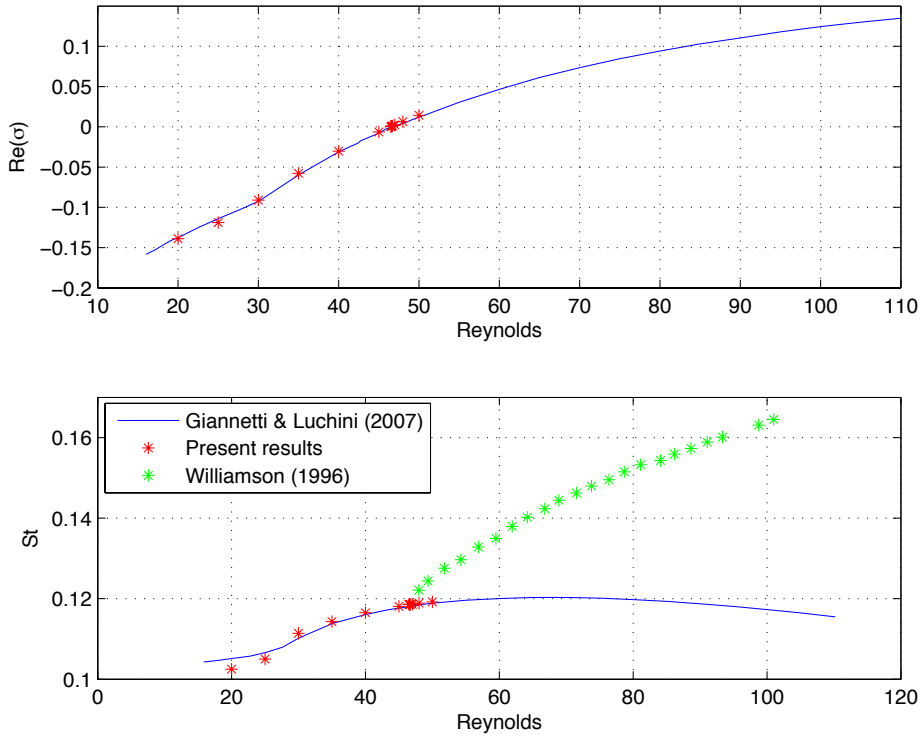
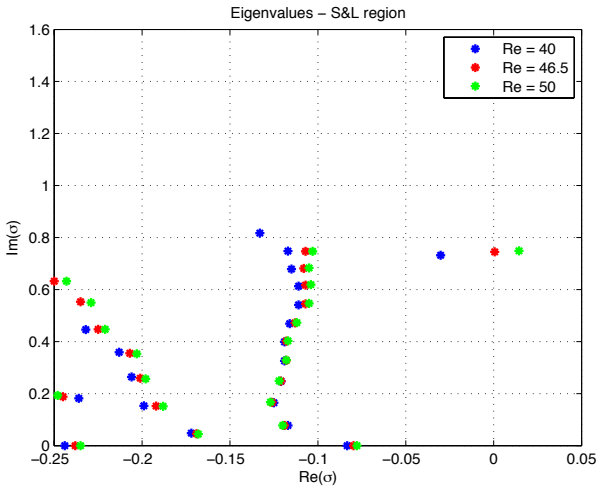
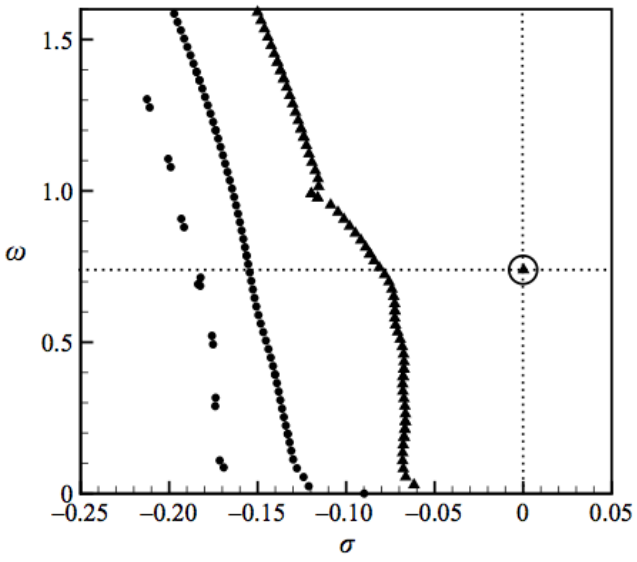


Figure 4.22: Cylinder flow: unstable eigenvalue: effect of Reynolds number on the real and imaginary parts

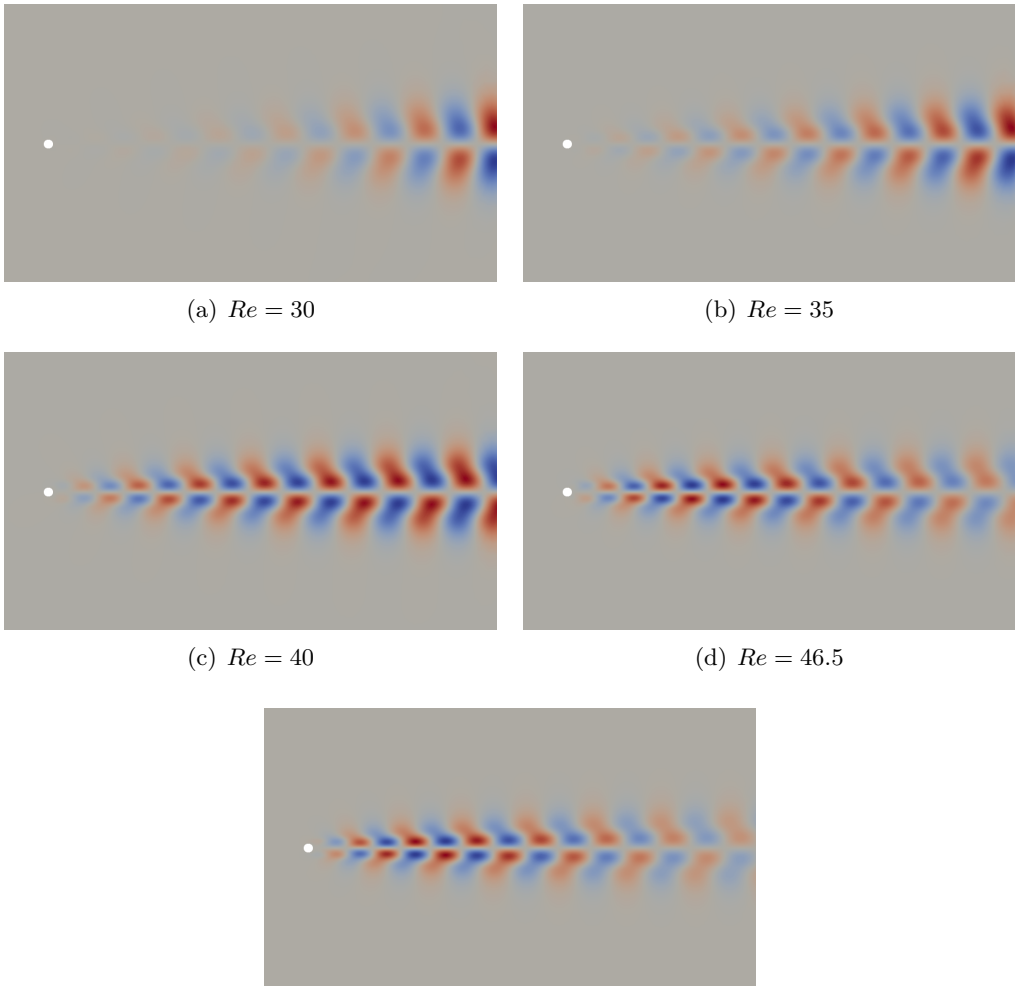


(a) Present results



(b) Sipp & Lebedev (2007), [14]

Figure 4.23: Cylinder flow: computed eigenvalues near imaginary axis

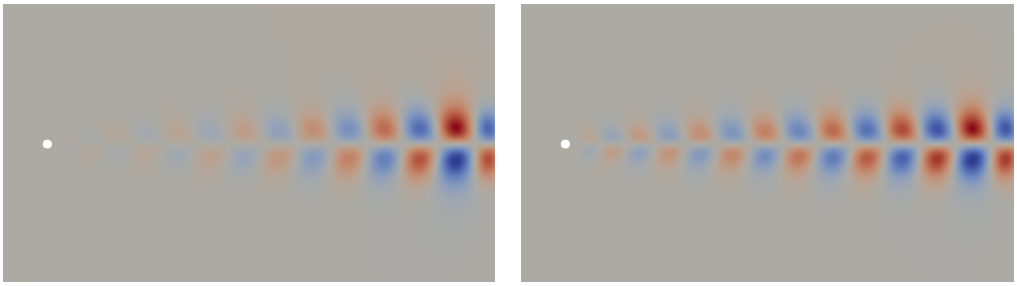


(a) $Re = 30$ (b) $Re = 35$

(c) $Re = 40$ (d) $Re = 46.5$

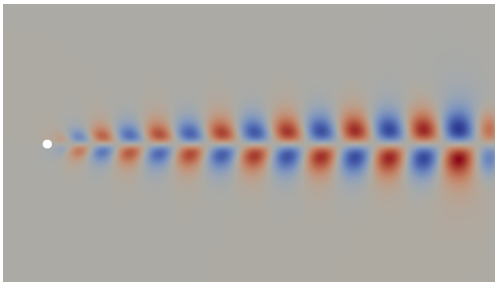
(e) $Re = 50$

Figure 4.24: Cylinder flow: dependence on Reynolds number of the most unstable eigenfuction (horizontal component of the velocity)

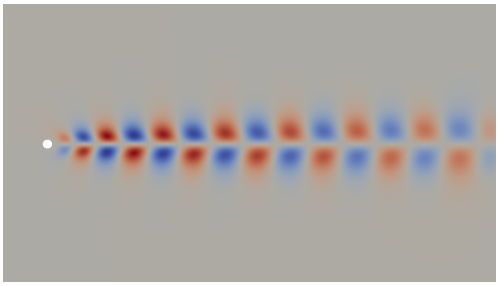


(a) $Re = 30$

(b) $Re = 35$



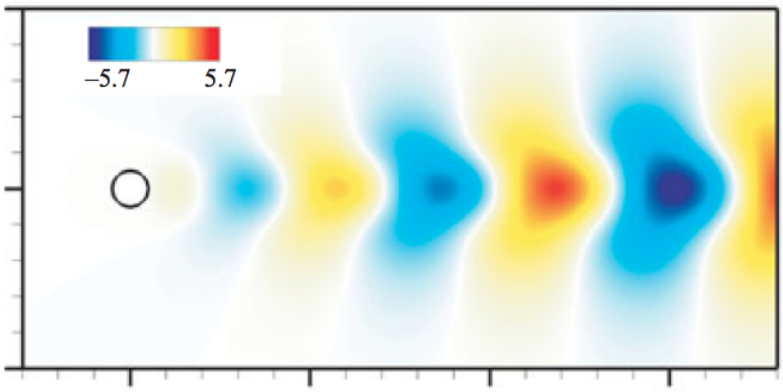
(c) $Re = 40$



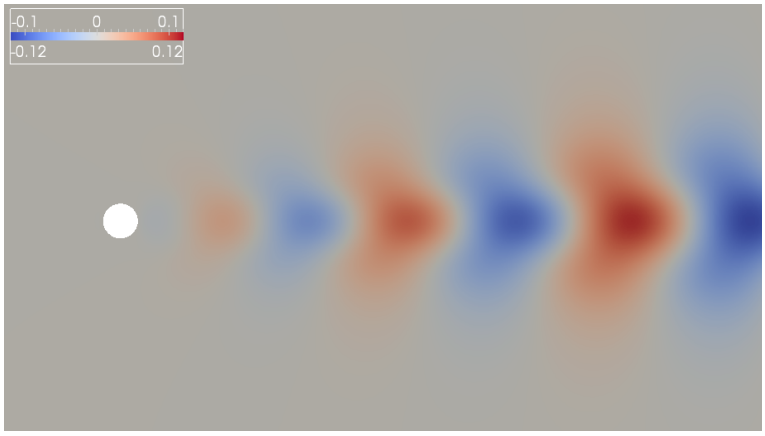
(d) $Re = 46.5$

(e) $Re = 50$

Figure 4.25: Cylinder flow: dependence on Reynolds number of the most unstable eigenfuction (pressure)



(a) Sipp & Lebedev (2007), [14]



(b) Present results

Figure 4.26: Cylinder flow: $Re(u_y)$ at Reynolds 46.6

Chapter 5

Conclusions and future steps

This work has focused on the numerical investigation of the stability of incompressible flows by the Finite Element Method. A new software has been developed by leveraging the Trilinos C++ library and the obtained results have been shown to be correct. In particular, it is shown that the software works properly for the computation of the base flow in the two examples provided, i.e. the lid-driven cavity flow and the 2D cylinder wake. For this latter test case, computed results the linear stability analysis has been performed and the result have been shown to be in close agreement with a couple of quite recent benchmarks.
The proposed Navier-Stokes proposed solver has revealed itself flexible and compact. A few lines of software are sufficient to implement the solver, and this is certainly an important feature. Changing the problem is very quick: the user can pass from the cavity problem to the cylinder wake one by just modifying a few lines. This makes the code extremely flexible. A small change to the Sundance class was necessary in order to implement the spectral transformation of *shifh-invert* type. This was possible only by virtue of to the open-source nature of the Trilinos library, a feature that offers a lot of possibilities to develop and improve the code. The Trilinos community is very active: a users mailing list exists, and I have often addressed it to submit questions and requests for clarifications. It was very useful and allowed to fill some gaps in the documentation. In particular the biggest effort was necessary to build and solve the eigenvalue problem, because the documentation was totally absent. In conclusion Sundance has revealed a good tool to develop in finite element solvers for partial differential equations.

5.1 Further developments

Further developments are needed at the present to obtain a fully capable solver for the linear stability problems. Hereafter a list is reported sorted according to what I deem more important.

1. **Parallel mode**: all computations were run in serial mode and the computational cost is too large at the moment. The impossibility to run in parallel

was caused by the choosen solver for the linear step of Newton's method, that has revealed itself inappropriate to suit the parallel mode. This aspect has limited the capability of the code, because it limits the number of mesh cells that can be employed. To improve our results using more refined meshes it would be crucial to solve this issue and change the linear solver. The iterative solvers provided by the AztecOO package of Trilinos could to be useful for reach this goal if efficientpreconditioners were available.

2. **Matrix factorization**: The coefficient matrix arising from the discretization of the linearized problem must be factored to implement the *shift-invert* method. This was done using the direct method provided by the Amesos package of Trilinos and it has revealed quite inefficient. It could be useful to investigate some alternative linear solver, such as UMFpack, MUMPS or SuperLU.

3. **Mesh format**: the current release of Sundance supports just Shewchuck's Triangle or ExodusII mesh format. It could be useful to extend the interface to other common mesh formats. The `MeshSource` object could be modified in order to make it possible to employ different input files for the mesh. In particular, it was really inconvenient to use the software Cubit just to assign labels to the mesh edges in order to enforce boundary conditions. Cubit is a commercial software developed at Sandia Labs and it is available for free only in a trial 30-days version. The `labeledSubset(int)` member function of the `CellFilter` class should be improved in order to read not only `int` type inputs but also strings. This should allow to use ExodusII mesh files generated by softwares different from Cubit.

4. **Preconditioners**: in order to find a different, more efficient way to solve the nonlinear problem related to the computation of the base flow, a preconditioned iterative solver can be considered. This seems to be a critical aspect and the choice of the best preconditioner is the subject of current investigations.

5. **Documentation**: Sundance documentation lacks in several parts. It would be extremely useful to complete the documentation. This could help the Trilinos community to know Sundance and its great potential.

6. **Complex shift**: the shift used in the spectral transformation is just a real shift. This aspect could be improved in order to reduce the dimension of the Krylov subspace required to compute the first unstable eigenvalue and, in this way, reduce the required computational time. It is not clear if Sundance can manage a complex shift, and more investigations are needed.

Appendix A

Complete codes

A.1 Chaco Mesh Partitioner script

```
#include "Sundance.hpp"
#include "SundanceMeshIOUtils.hpp"

int main(int argc, char** argv)
{
  try
  {
    std::string infile="<MeshFile>";
    std::string outfile=infile;
    int numProc = <NP>;
    bool help=false;
    Sundance::setOption("i", infile, "Input mesh filename");
    Sundance::setOption("o", outfile, "Output mesh filename");
    Sundance::setOption("np", numProc, "Number of partitions");
    Sundance::setOption("h", "nohelp", help, "Help");

    Sundance::init(&argc, &argv);

    if (help || infile.length()==0 || numProc<1)
    {
      cout << "Usage: partitionExo --i=inputFilename --o=outputFilename "
        "--np=numberOfPartitions" << std::endl;
      cout << "Do not include .exo suffix on filenames" << std::endl;
      return 0;
    }
    else
    {
      TEST_FOR_EXCEPT(infile.length()==0);
      TEST_FOR_EXCEPT(outfile.length()==0);
      TEST_FOR_EXCEPT(numProc<=1);

      MeshType meshType = new BasicSimplicialMeshType();

      MeshSource mesher
        = new ExodusMeshReader(infile, meshType);

      RCP<SerialPartitionerBase> part
        = rcp(new FileIOChacoPartitioner("part"));

      serialPartition(part, numProc, mesher, outfile);
    }
```

```
    TimeMonitor::summarize();
  }
catch(std::exception& e)
  {
    std::cerr << "Detected exception: " << e.what() << std::endl;
  }
}
```

A.2 Sundance complete code

```
/* ************************************************************************ */
/*                  Navier-Stokes Flow around 2D cylinder                   */
/* ************************************************************************ */

#include "Sundance.hpp"
#include "SundanceEvaluator.hpp"
#include "PlayaNOXSolver.hpp"
#include "PlayaMPIComm.hpp"
#include "PlayaAnasaziEigensolverDecl.hpp"
#include "PlayaAnasaziEigensolverImpl.hpp"

// #include "SundanceElementIntegral.hpp"
using Sundance::List;


int main(int argc, char** argv)
{
 try
 {

  // Declare solver:
  std::string meshFile="../Mesh/cylinder2";
  std::string solverFile = "../Solvers/nox-amesos.xml";
  Sundance::setOption("meshFile", meshFile, "mesh file");
  Sundance::setOption("solver", solverFile, "name of XML file for solver");
  std::string solverFileEig = "../Solvers/AnasaziKrylovSchur-04LM-300.xml";

  Sundance::init(&argc, &argv);
  MPIComm comm = MPIComm::world();
  int myrank = MPIComm::world().getRank();

  // Define case ID number:
  double numCase = 0.0;

  // Define Reynolds number:
  double reynolds = 50.0;

  std::cout << "                                    " << std::endl;
  std::cout << "START CASE: " + Teuchos::toString(numCase) << std::endl;
  std::cout << "Mesh File: " + meshFile << std::endl;
  std::cout << "Reynolds: " + Teuchos::toString(reynolds) << std::endl;
  std::cout << "Solver: " + solverFile << std::endl;
  std::cout << "Solver Eigen: " + solverFileEig << std::endl;



  // Define vector type:
  VectorType<double> vecType = new EpetraVectorType();

  // Input Triangle mesh:
  MeshType meshType = new BasicSimplicialMeshType();
  MeshSource mesher = new ExodusMeshReader( meshFile, meshType, comm );
  Mesh mesh = mesher.getMesh();

  cout << "Nr Cells: "<<mesh.numCells(2) << std::endl;
  cout << "My Rank is: " << myrank << std::endl;

  // Define mesh domain:
  CellFilter interior = new MaximalCellFilter();
  CellFilter edges = new DimensionalCellFilter(1);
```

```
  // CellFilter point = new DimensionalCellFilter(0);

  CellFilter left = edges.labeledSubset(3);
  CellFilter right = edges.labeledSubset(4);
  CellFilter up = edges.labeledSubset(2);
  CellFilter down = edges.labeledSubset(5);
  CellFilter cylinder = edges.labeledSubset(1);

  CellFilter walls = cylinder;
  CellFilter up_down = up + down;
  CellFilter inflow = left;
  CellFilter outflow = right;

  // Define basis family and functions:
  BasisFamily L1 = new Lagrange(1);
  BasisFamily L2 = new Lagrange(2);

  Expr ux = new UnknownFunction(L2, "ux");
  Expr vx = new TestFunction(L2, "vx");

  Expr uy = new UnknownFunction(L2, "uy");
  Expr vy = new TestFunction(L2, "vy");

  Expr p = new UnknownFunction(L1, "p");
  Expr q = new TestFunction(L1, "q");

  Expr u = List(ux, uy);
  Expr v = List(vx, vy);

  // Define coordinates, derivatives and gradient:
  Expr x = new CoordExpr(0);
  Expr y = new CoordExpr(1);
  Expr dx = new Derivative(0);
  Expr dy = new Derivative(1);
  Expr grad = List(dx, dy);
  // Expr h = new CellDiameterExpr();

  // Define quadrature rule:
  QuadratureFamily quad1 = new GaussianQuadrature(1);
  QuadratureFamily quad2 = new GaussianQuadrature(2);
  QuadratureFamily quad4 = new GaussianQuadrature(4);

  // Define the weak form:
  Expr eqn1 = Integral(interior, (1/reynolds)*((grad*vx)*(grad*ux) +
(grad*vy)*(grad*uy)) + vx*(u*grad)*ux + vy*(u*grad)*uy - p*(dx*vx+dy*vy), quad2 );

  Expr eqn2 = Integral( interior, (dx*ux + dy*uy) * q, quad2 );

  Expr eqn = eqn1 + eqn2;

  // Define the boundary conditions:
  // Expr Uinflow = 0.5 * (1.0-x*x); // parabolic u profile at inflow

  Expr bcWalls = EssentialBC(walls, v*u, quad2);
  Expr bcInflow = EssentialBC(inflow, vx*(ux-1.0) + vy*uy, quad2);
  Expr bcOutflow = EssentialBC(outflow, q*p, quad2);
  Expr bcUpDown = EssentialBC(up_down, vy*uy, quad2);

  Expr bc = bcWalls + bcInflow + bcOutflow + bcUpDown;
```

```
/* ******************************************************** */
/*                    NON LINEAR PROBLEM                    */
/* ******************************************************** */

  // Define discrete space and initial guess for iterative method:
  DiscreteSpace discSpace(mesh, Sundance::List(L2, L2, L1), vecType);
  Expr u0 = new DiscreteFunction(discSpace, 0.0, "u0");

  // Define the NON linear problem:
  NonlinearProblem prob( mesh, eqn, bc, List(vx, vy, q), List(ux, uy, p), u0, vecType);

  // Define solver parameters and build the solver:
  ParameterXMLFileReader reader(solverFile);
  ParameterList solverParams = reader.getParameters();
  NOXSolver solver(solverParams);

  // Solve the nonlinear system
  NOX::StatusTest::StatusType status = prob.solve(solver);
  DiscreteSpace discSpace2(mesh, L1, vecType);
  L2Projector projector(discSpace2, dx*u0[1]-dy*u0[0]);
  Expr omega = projector.project();

  // Write the field in VTK format
  FieldWriter w = new VTKWriter("cylinder2-Re-" + Teuchos::toString(reynolds));
  w.addMesh(mesh);

  Expr expr_vector(List(u0[0], u0[1]));

  w.addField("ux", new ExprFieldWrapper(u0[0]));
  w.addField("uy", new ExprFieldWrapper(u0[1]));
  w.addField("vel", new ExprFieldWrapper(expr_vector));
  w.addField("p", new ExprFieldWrapper(u0[2]));
  w.addField("vorticity", new ExprFieldWrapper(omega));
  w.write();

/*********************************************************/
/*             Wake-Bubble measurement                   */
/*********************************************************/

  // ux interpolation
  DiscreteSpace discSpace3(mesh, L1, vecType);
  L2Projector projUx(discSpace3, u0[0]);
  Expr u_x = projUx.project();

  AToCPointLocator locator(mesh, interior, createVector(tuple(200, 200)));
  CToAInterpolator interpolator(locator, u_x);

  int nPts = 1000;
  Array<double> pos(2*nPts);
  Array<double> Ux(u_x.size() * nPts);

  vector<double> X(nPts);
  vector<double> Y(nPts);
  double X0_b = 3.0;
  double XF_b = 4.0;
  double h = (XF_b - X0_b) / nPts;

  for (int i=0; i<nPts; i++)
        {
```

```
X[i] = X0_b + i * h;
Y[i] = 0.0;
}

  for (int i=0; i<nPts; i++)
   {
pos[2*i] = X[i];
pos[2*i+1] = Y[i];
}

  interpolator.interpolate(pos, Ux);

  std::cout << " ************** ux ************** " << std::endl;

  for (int i=0; i<nPts; i++)
   {
std::cout << "x = " << X[i] << "   y = " << Y[i] << "   ux = " << Ux[i] << std::endl;
}




/*****************************************************************************/
/*                              EIGENVALUE PROBLEM:                          */
/*****************************************************************************/

  bool lumpedMass = false;
  double sigma = -0.01;

  Expr U = List(u0[0], u0[1]);

  // Define the new weak form for the operator A:
  Expr eqn3 = Integral(interior, - ( (1/reynolds)*((grad*vx)*(grad*ux) +
(grad*vy)*(grad*uy)) + vx*(U*grad)*ux + vy*(U*grad)*uy + vx*(u*grad)*U[0]
+ vy*(u*grad)*U[1] - p*(dx*vx+dy*vy) ), quad2 );

  Expr eqn4 = Integral( interior, - (dx*ux + dy*uy) * q, quad2 );

  Expr sigma_M = Integral(interior, - sigma * ( vx*ux + vy*uy + 0.000001*q*p), quad2 );

  Expr eqnEig = eqn3 + eqn4 + sigma_M;

  // Define bc for eigenproblem:
  Expr bcInflowEig = EssentialBC(inflow, vx*ux + vy*uy, quad2);
  Expr bcEig = bcWalls + bcInflowEig + bcOutflow + bcUpDown;

  // Define the weak form to build the operator M:
  Expr massExpr = vx*ux + vy*uy + 0.000001*q*p;

  LinearEigenproblem probEig(mesh, eqnEig, bcEig, massExpr, List(vx, vy, q),
 List(ux, uy, p), vecType, lumpedMass);

  // Define solver parameters and build the solver:
  ParameterXMLFileReader readerEig(solverFileEig);
  ParameterList paramsEig = readerEig.getParameters();
  ParameterList solverParamsEig = paramsEig.sublist("Eigensolver");
  Eigensolver<double> solverEig = new AnasaziEigensolver<double>(solverParamsEig);

  Eigensolution solnEig = probEig.solve_inverse(solverEig);

  // qui recuperare gli autovalori del problema originario !!!
  for(int i = 0; i < solnEig.numEigenfunctions(); i++)
```

```
    {
      const std::complex<double>& ew = solnEig.eigenvalue(i);
      std::complex<double> eigval;
      Expr ev = solnEig.eigenfunction(i);
      eigval = solnEig.eigenvalue(i);
      eigval = sigma + (1.0 / solnEig.eigenvalue(i));
      cout << " ********************************** " << std::endl;
      cout << "ew=(" << eigval  << ")" << std::endl;
      cout << "ev=(" << ev  << ")" << std::endl;
    }


  DiscreteSpace discSpaceU_m(mesh, L2, vecType);
  DiscreteSpace discSpaceP_m(mesh, L1, vecType);

  FieldWriter wef = new VTKWriter( "UnstableEigenmodes-2-Re" + Teuchos::toString(reynolds));
  wef.addMesh( mesh );
  for (int i=0; i< solnEig.numEigenfunctions(); i++)
  {
    Expr ef = solnEig.eigenfunction(i);

    Expr Re_ux = ef[0].real();
    Expr Im_ux = ef[0].imag();
    Expr Re_uy = ef[1].real();
    Expr Im_uy = ef[1].imag();
    Expr Re_p = ef[2].real();
    Expr Im_p = ef[2].imag();
    Expr ux_m = sqrt((Re_ux * Re_ux) + (Im_ux * Im_ux));
    Expr uy_m = sqrt((Re_uy * Re_uy) + (Im_uy * Im_uy));
    L2Projector projectorU_m(discSpaceU_m, sqrt( (ux_m * ux_m) + (uy_m * uy_m) ));
    Expr u_m = projectorU_m.project();
    L2Projector projectorP_m(discSpaceP_m, sqrt( (Re_p * Re_p) + (Im_p * Im_p) ));
    Expr p_m = projectorP_m.project();
    // Expr u_mode(List(ef[0], ef[1]));
    // Expr p_mode(List(ef[2], ef[2]));
    wef.addField("u[" + Teuchos::toString(i) + "]", new ExprFieldWrapper(u_m));
    wef.addField("p[" + Teuchos::toString(i) + "]", new ExprFieldWrapper(p_m));
   }
    wef.write();


  // End run message:
  std::cout << "                                    " << std::endl;
  std::cout << "END OF PROGRAM " + Teuchos::toString(numCase) << std::endl;
  std::cout << "-----------------------------------" << std::endl;
  std::cout << "                                    " << std::endl;

 }
 catch(exception& e)
 {
  Sundance::handleException(e);
 }
 Sundance::finalize();
 return Sundance::testStatus();
}
```

A.3 Sundance linear eigenproblem class

SundanceLinearEigenproblem.cpp

```
  // ************************************************************************
/* @HEADER@ */

#include "PlayaLinearSolverBuilder.hpp"
#include "PlayaAmesosSolver.hpp"
#include "SundanceLinearEigenproblem.hpp"
#include "SundanceOut.hpp"
#include "PlayaTabs.hpp"
#include "SundanceTestFunction.hpp"
#include "SundanceUnknownFunction.hpp"
#include "SundanceEssentialBC.hpp"
#include "SundanceIntegral.hpp"
#include "SundanceListExpr.hpp"
#include "SundanceZeroExpr.hpp"
#include "SundanceEquationSet.hpp"
#include "SundanceQuadratureFamily.hpp"
#include "SundanceAssembler.hpp"
#include "SundanceMaximalCellFilter.hpp"
#include "SundanceGaussianQuadrature.hpp"

#include "PlayaLinearCombinationDecl.hpp"
#include "PlayaLinearCombinationImpl.hpp"
#include "PlayaLinearOperatorDecl.hpp"
#include "PlayaVectorDecl.hpp"
#include "PlayaSimpleDiagonalOpDecl.hpp"
#include "PlayaSimpleIdentityOpDecl.hpp"
#include "PlayaSimpleIdentityOpImpl.hpp"
#include "PlayaSimpleComposedOpDecl.hpp"
#include "PlayaSimpleComposedOpImpl.hpp"

#include "PlayaInverseOperatorDecl.hpp"
#include "PlayaInverseOperatorImpl.hpp"
#include "PlayaLinearSolverImpl.hpp"

#ifndef HAVE_TEUCHOS_EXPLICIT_INSTANTIATION
#include "PlayaVectorImpl.hpp"
#include "PlayaLinearCombinationImpl.hpp"
#include "PlayaLinearOperatorImpl.hpp"
#include "PlayaSimpleDiagonalOpImpl.hpp"
#include "PlayaSimpleIdentityOpDecl.hpp"
#endif

using namespace Sundance;
using namespace Sundance;
using namespace Sundance;
using namespace Sundance;
using namespace Teuchos;
using namespace Playa;
using namespace PlayaExprTemplates;

static Time& normalizationTimer()
{
  static RCP<Time> rtn
    = TimeMonitor::getNewTimer("Eigenfunction normalization");
  return *rtn;
}

static Time& makeEigensystemTimer()
```

```
{
  static RCP<Time> rtn
    = TimeMonitor::getNewTimer("Building eigensystem stiffness matrix");
  return *rtn;
}

LinearEigenproblem::LinearEigenproblem(
  const Mesh& mesh, const Expr& eqn,
  const Expr& v, const Expr& u,
  const VectorType<double>& vecType)
  : lumpMass_(false),
    kProb_(),
    mProb_(),
    M_(),
    MUnlumped_(),
    discSpace_()
{
  Expr empty;

  kProb_ = LinearProblem(mesh, eqn, empty, v, u, vecType);
  discSpace_ = *(kProb_.solnSpace()[0]);
}

LinearEigenproblem::LinearEigenproblem(
  const Mesh& mesh, const Expr& eqn,
  const Expr& v, const Expr& u,
  const VectorType<double>& vecType,
  bool lumpedMass)
  : lumpMass_(lumpedMass),
    kProb_(),
    mProb_(),
    M_(),
    MUnlumped_(),
    discSpace_()
{
  Expr empty;

  kProb_ = LinearProblem(mesh, eqn, empty, v, u, vecType);
  mProb_ = makeMassProb(mesh, empty, v, u, vecType);
  discSpace_ = *(kProb_.solnSpace()[0]);
  MUnlumped_ = mProb_.getOperator();
  if (lumpMass_)
  {
    M_ = lumpedOperator(MUnlumped_);
  }
  else
  {
    M_ = MUnlumped_;
  }
}


LinearEigenproblem::LinearEigenproblem(
  const Mesh& mesh, const Expr& eqn,
  const Expr& massExpr,
  const Expr& v, const Expr& u,
  const VectorType<double>& vecType,
  bool lumpedMass)
  : lumpMass_(lumpedMass),
    kProb_(),
    mProb_(),
    M_(),
```

```
    MUnlumped_(),
    discSpace_()
{
  Expr bc;
  kProb_ = LinearProblem(mesh, eqn, bc, v, u, vecType);
  mProb_ = makeMassProb(mesh, massExpr, v, u, vecType);
  discSpace_ = *(kProb_.solnSpace()[0]);

  MUnlumped_ = mProb_.getOperator();
  if (lumpMass_)
  {
    M_ = lumpedOperator(MUnlumped_);
  }
  else
  {
    M_ = MUnlumped_;
  }

}


// modificato 21/02/2012
LinearEigenproblem::LinearEigenproblem(
  const Mesh& mesh, const Expr& eqn, const Expr& bc,
  const Expr& massExpr,
  const Expr& v, const Expr& u,
  const VectorType<double>& vecType,
  bool lumpedMass)
  : lumpMass_(lumpedMass),
    kProb_(),
    mProb_(),
    M_(),
    MUnlumped_(),
    discSpace_()
{
  kProb_ = LinearProblem(mesh, eqn, bc, v, u, vecType);
  mProb_ = makeMassProb(mesh, massExpr, v, u, vecType);
  discSpace_ = *(kProb_.solnSpace()[0]);

  MUnlumped_ = mProb_.getOperator();
  if (lumpMass_)
  {
    M_ = lumpedOperator(MUnlumped_);
  }
  else
  {
    M_ = MUnlumped_;
  }

}


LinearProblem LinearEigenproblem::makeMassProb(
  const Mesh& mesh,
  const Expr& massExpr,
  const Expr& v, const Expr& u,
  const VectorType<double>& vecType) const
{
  Expr eqn;

  CellFilter interior = new MaximalCellFilter();
  QuadratureFamily quad = new GaussianQuadrature( 4 );
```

```
  if (massExpr.ptr().get()==0)
  {
    eqn = Integral(interior, v*u, quad);
  }
  else
  {
    eqn = Integral(interior, massExpr, quad);
  }
  Expr bc;
  LinearProblem rtn(mesh, eqn, bc, v, u, vecType);
  return rtn;
}


Array<Expr> LinearEigenproblem::makeEigenfunctions(
  Array<Vector<double> >& ev) const
{
  TimeMonitor timer(normalizationTimer());

  Array<Expr> x(ev.size());
  CellFilter interior = new MaximalCellFilter();
  QuadratureFamily q = new GaussianQuadrature(2);
  for (int i=0; i<ev.size(); i++)
  {
    x[i] = new DiscreteFunction(discSpace_, ev[i], "ev[" + Teuchos::toString(i)+"]");
    double N = 1.0;
    if (MUnlumped_.ptr().get())
    {
      N = ev[i] * (MUnlumped_ * ev[i]);
    }
    else
    {
      N = evaluateIntegral(discSpace_.mesh(),
        Integral(interior, x[i]*x[i], q));
    }
    ev[i].scale(1.0/sqrt(N));
  }

  return x;
}


LinearOperator<double>
LinearEigenproblem::lumpedOperator(const LinearOperator<double>& M) const
{
  Vector<double> ones = M.domain().createMember();
  ones.setToConstant(1.0);
  Vector<double> m = M * ones;
  LinearOperator<double> rtn = diagonalOperator(m);

  return rtn;
}


Eigensolution LinearEigenproblem::solve(const Eigensolver<double>& solver) const
{
  Array<std::complex<double> > ew;
  Array<Vector<double> > ev;

  LinearOperator<double> K;
  {
    TimeMonitor timer(makeEigensystemTimer());
```

```
    K = kProb_.getOperator();
  }

  solver.solve(K, M_, ev, ew);

  Array<Expr> eigenfuncs = makeEigenfunctions(ev);

  return Eigensolution(eigenfuncs, ew);
}

// Added by F.A. on 17/2/2012: inverse, no shift
Eigensolution LinearEigenproblem::solve_inverse(const Eigensolver<double>& solver) const
{
  Array<std::complex<double> > ew;
  Array<Vector<double> > ev;
  LinearSolver<double> linInvSolver = LinearSolverBuilder::createSolver("../Solvers/
amesos.xml");

  LinearOperator<double> K, Ainv, AinvM;
  {
    TimeMonitor timer(makeEigensystemTimer());
    K = kProb_.getOperator();
    Ainv = inverse(K, linInvSolver);
    AinvM = Ainv * M_;
  }

  LinearOperator<double> Id = identityOperator(K.domain());


  solver.solve(AinvM, Id, ev, ew);

  Array<Expr> eigenfuncs = makeEigenfunctions(ev);

  return Eigensolution(eigenfuncs, ew);
}
```

SundanceLinearEigenproblem.hpp.

```
// ************************************************************************
/* @HEADER@ */

#ifndef SUNDANCE_LINEAREIGENPROBLEM_H
#define SUNDANCE_LINEAREIGENPROBLEM_H

#include "SundanceDefs.hpp"
#include "SundanceLinearProblem.hpp"
#include "SundanceEigensolution.hpp"
#include "SundanceFunctionalEvaluator.hpp"
#include "PlayaEigensolver.hpp"

namespace Sundance
{
using namespace Sundance;
using namespace Sundance;
using namespace Sundance;
using namespace Sundance;
using namespace Sundance;
using namespace Teuchos;
using namespace Playa;


/**
 *
 */
class LinearEigenproblem
{
public:
  /** */
  LinearEigenproblem(){}
  /** */
  LinearEigenproblem(
    const Mesh& mesh, const Expr& eqn,
    const Expr& v, const Expr& u,
    const VectorType<double>& vecType) ;
  /** */
  LinearEigenproblem(
    const Mesh& mesh, const Expr& eqn,
    const Expr& v, const Expr& u,
    const VectorType<double>& vecType,
    bool lumpMass) ;
  /** */
  LinearEigenproblem(
    const Mesh& mesh, const Expr& eqn,
    const Expr& massExpr,
    const Expr& v, const Expr& u,
    const VectorType<double>& vecType,
    bool lumpMass) ;
  /** */
  LinearEigenproblem(
    const Mesh& mesh, const Expr& eqn, const Expr& bc,
    const Expr& massExpr,
    const Expr& v, const Expr& u,
    const VectorType<double>& vecType,
    bool lumpMass) ;
  /** */
  Eigensolution solve(const Eigensolver<double>& solver) const ;

  /** */
  Eigensolution solve_inverse(const Eigensolver<double>& solver) const ;
```

```
  /** */
  Eigensolution solve_shift_inverse(const Eigensolver<double>& solver) const ;

  /** */
  LinearOperator<double> getK() const {return kProb_.getOperator();}

  /** */
  LinearOperator<double> getM() const {return mProb_.getOperator();}


private:
  /** */
  Array<Expr> makeEigenfunctions(Array<Vector<double> >& ev) const ;

  /** */
  LinearProblem makeMassProb(
    const Mesh& mesh,
    const Expr& massExpr,
    const Expr& v, const Expr& u,
    const VectorType<double>& vecType) const ;

  /** */
  LinearOperator<double>
  lumpedOperator(const LinearOperator<double>& M) const ;

  bool lumpMass_;
  LinearProblem kProb_;
  LinearProblem mProb_;
  LinearOperator<double> M_;
  LinearOperator<double> MUnlumped_;
  DiscreteSpace discSpace_;
};


}

#endif
```

Acknowledgements

A special mention goes to my supervisor Franco Auteri. His patient guide has been crucial for the good result of this work. In every meeting with him I have learned several knowledges, which will be certainly useful also in my future career. To him my total admiration.
Particular thanks go to the Professor Kevin Long, for his precious advice in the very beginning of the work, and to Heidi Thornquist for the support in the last period of work.

Bibliography

[1] F. Auteri, L. Quartapelle, "Newton–Krylov Method with Stokes Preconditioning for Steady Incompressible Flow in a Sphere", in preparation, 2011

[2] F. Auteri, L. Quartapelle, "Stability analysis of Navier-Stokes equations", in preparation, 2011

[3] F. Auteri, L. Quartapelle, L. Vigevano, "Accurate ω-Ψ Spectral Solution of the Singular Driven Cavity Problem", *Journal of Computational Physics*, vol. **180**, pp. 597–615, 2002

[4] D. Barkley, H. M. Blackburn, S. J. Sherwin, "Direct Optimal Growth Analysis for Timesteppers", *International Journal For Numerical Method In Fluids*, vol. **57**, pp. 1435–1458, 2008

[5] J. Benk, M. Mehl, M. Ulbrich, "Sundance PDE Solvers on Cartesian Fixed Grid in Complex and Variables Geometries", *CFD and OPTIMIZATION 2011 - An ECCOMAS Thematic Conference*, Anatalya (Turkey), 2011

[6] O. Botella, R. Peyret, "Benchmark Spectral Results on the Lid-Driven Cavity Flow", *Computers & Fluids*, vol. **24**, No. 4, pp. 421-433, 1998

[7] F. Giannetti, P. Luchini, "Structural Sensitivity of the First Instability of the Cylinder Wake", *Journal of Fluid Mechanics*, vol. **581**, pp. 167-197, 2007

[8] K. Hirata, N. Matoba, T. Naruse, Y. Haneda, J. Funaki, "On the Stable-Oscillation Domain of a Simple Fluidic Oscillator", *Journal of Fluid Science and Technology*, vol. **4**, No. 3, pp. 623–635, 2009

[9] D. Lanzerstorfer, H. C. Kuhlmann, "Global Stability of the Two-Dimensional Flow Over a Backward-Facing Step", *Journal of Fluid Mechanics*, vol. **693**, pp. 1-27, 2012

[10] K. Long, "Sundance 2.0 Tutorial", Computational Science and Mathematics Research Department, Sandia National Laboratories, Livermore, CA, 2004

[11] K. Long "Sundance: a Trilinos package for efficient development of efficient simulators", *Copper Mountain Trilinos Workshop*, 2008

[12] A. Quarteroni, *Modellistica Numerica per Problemi Differenziali*, Springer-Verlag Italia, Milano, 2008

[13] M. Sala, M. A. Heroux, D. M. Day, J. M. Willenbring, "Trilinos Tutorial - For Trilinos Release 10.8", Available on http://trilinos.sandia.gov, 2011

[14] D. Sipp, A. Lebedev, "Global Stability of Base and Mean Flows: a General Approach and its Applications to Cylinder and Open Cavity Flows", *Journal of Fluid Mechanics, vol. **593**, pp. 333–358*, 2007

[15] G. W. Stewart, "A Krylov-Schur algorithm for large scale eigenproblems", *SIAM J. Matrix Anal. Appl.*, vol **23**, No. 3, pp. 601–614, 2001

[16] R. Temam, *Theory and numerical analysis of the Navier–Stokes equations*, North-Holland, Amsterdam, 1977

[17] V. Theofilis, "Global Linear Instability", *Annual Reviews of Fluid Mechanics*, vol. **43**, pp. 319-352, 2011

[18] C. H. K. Williamson, "Vortex Dynamics in the Cylinder Wake", *Annual Review of Fluid Mechanics*, vol. **28**, pp. 477–539, 1996

[19] Y. Zhou, Y. Saad, "Block Krylov-Schur method for large symmetric eigenvalue problems", *Kluwer Academic Publishers*, 2008

[20] "Third-Party Libraries - A discussion of the various third-party libraries often used in conjunction with Trilinos", Available on http://trilinos.sandia.gov/third-party_libraries.html

[21] "Trilinos CMake Quickstart - Instructions for building Trilinos 10.0 and later", Available on http://trilinos.sandia.gov, 2009