Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 2095-2110 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

AN OBJECT ORIENTED VERSION OF THE KINEMATIC LAPLACIAN EQUATION METHOD

Gabriel H. Burzstyn [*a], **Javier Quinteros**[b] **and Alejandro D. Otero**[a,c,d]

[a]*Departamento de Computación, FCEN, Universidad de Buenos Aires, Intendente Güiraldes 2160 - Ciudad Universitaria, C1428EGA, Buenos Aires, Argentina*

[b]*Deutsches GeoForschungsZentrum, Section 2.5 - Geodynamical Modeling, Telegrafenberg, 14473 Potsdam, Germany*

[c]*Grupo ISEP, Depto. de Electrotecnia, FI, Universidad de Buenos Aires, Paseo Colón 850, C1063ACV, Buenos Aires, Argentina*

[d]*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Av. Rivadavia 1917, C1033AAJ, Buenos Aires, Argentina*

Abstract. The kinematic Laplacian equation (KLE) method solves the Navier–Stokes equations by means of its vorticity-velocity formulation. This method has been used to calculate the time dependent flow around moving bodies and other fluid dynamic problems with great success.

The KLE computes the time evolution of the vorticity as an ordinary differential equation (ODE) in each node of the discretized space. The input data for the vorticity transport equation at each time step is provided by a modified version of the Poisson linear partial differential equation for the velocity, called KLE Equation.

This paper presents an object oriented implementation based on a general purpose and high performance framework for solving partial differential equations by the finite and spectral element methods. The framework can interact with different high-performance linear algebra libraries, either for dense or sparse matrices.

Different matrix assembly and boundary condition imposition methods were tested as well as two different solvers in order to find the code version with the best performance. The method was validated against a problem with known analytical solution. Scalability tests were performed to study the behavior of the method as the complexity of the problem increases. Results showing the benefits obtained with this implementation are presented.

[*]Corresponding author: gburszty@dc.uba.ar

1 INTRODUCTION

The classic formulation of Navier–Stokes equations is expressed as a function of velocity and pressure. The kinematic Laplacian equation (KLE) method (Ponta, 2005), instead solves the Navier–Stokes equations starting from a vorticity-velocity, also known as hybrid, formulation. The KLE calculates the evolution in time of the vorticity as an ordinary differential equation (ODE) in each node of the discretization. The input data for the vorticity transport equation at each time step is computed using a modified version of the linear Poisson partial differential equation for velocity, called *KLE equation*. There are several implementations of this method, ranging from functional prototypes to the ones developed on interpreted languages or C++. This work is an evolution of Bursztyn et al. (2008) which presented an object oriented implementation of the KLE method based on a general purpose and high performance framework for solving partial differential equations by the finite and spectral element methods. Due to the interfaces developed, this framework is able to make use of different high-performance linear algebra libraries, either for dense or sparse matrices.

A brief overview of the KLE method is presented in section 2. The framework used and the KLE implementation are explained in sections 3 and 4 respectively. The main improvements of this implementation with respect to that of Bursztyn et al. (2008) are exposed in section 5. Finally, the results about the performance achieved by means of different approaches to impose boundary conditions and different solvers are shown in section 6.

2 THE KLE METHOD

Ponta (2005) presented a new method based on the vorticity-velocity formulation ($\boldsymbol{\omega}$, $\boldsymbol{v}$) to model the Navier–Stokes equations. The KLE method is characterized by a complete decoupling of the two variables, vorticity in time - velocity in space. In this way, it reduces to three the number of variables to be solved in the temporal integration process with respect to the six that have other hybrid formulations. Furthermore, this decomposition of the problem allows the use of algorithms such as ODE solvers of variable order and adaptive time step to improve the efficiency and robustness of the integration process.

The KLE method consists of two parts. First, the so-called kinematic Laplacian equation, which is a modified version of the Poisson equation for velocity, responsible for solving the kinematics of the problem. Second, the algorithm to integrate the equations related to the dynamics of the problem, in other words, the vorticity transport equation, which is resolved as an ODE on each node of the discretization. Both sides feed each other and together are known as KLE method.

The expression of the variational formulation of the KLE for incompressible flow, which is used in this work can be expressed as

$$\int_\Omega \boldsymbol{\nabla v} : \boldsymbol{\nabla \delta v} + \alpha_{\mathcal{D}} (\boldsymbol{\nabla \cdot v})(\boldsymbol{\nabla \cdot \delta v}) + \alpha_\omega (\boldsymbol{\nabla} \times \boldsymbol{v}) \cdot (\boldsymbol{\nabla} \times \boldsymbol{\delta v}) \, \mathrm{d}\Omega = \int_\Omega (\boldsymbol{\nabla} \times \boldsymbol{\omega}) \cdot \boldsymbol{\delta v} + \alpha_\omega \boldsymbol{\omega} \cdot (\boldsymbol{\nabla} \times \boldsymbol{\delta v}) \, \mathrm{d}\Omega, \tag{1}$$

where $\boldsymbol{v}$ is the velocity, $\boldsymbol{\omega}$ is the vorticity and $\alpha_{\mathcal{D}}$ and α_ω are the penalty constants for the constrains $\boldsymbol{\nabla \cdot v} = \mathcal{D}$ and $\boldsymbol{\nabla} \times \boldsymbol{v} = \boldsymbol{\omega}$, where $\mathcal{D}$ is the expansion rate.

2.1 Flow temporal evolution

The KLE method represents the flow dynamics by means of the Navier–Stokes equations in terms of the vorticity. Namely,

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = -\boldsymbol{v}\cdot\boldsymbol{\nabla}\boldsymbol{\omega} + \nu\nabla^2\boldsymbol{\omega} + \boldsymbol{\omega}\cdot\boldsymbol{\nabla}\boldsymbol{v}. \tag{2}$$

It is obtained from the full three-dimensional incompressible flow in a domain Ω, with a solid boundary $\partial\Omega$ and far from the outer boundary in a reference frame fixed to the solid.

If at any moment the velocity field $\boldsymbol{v}$ is known for the whole domain Ω, then eq. 2 can be rewritten as

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = -\boldsymbol{v}\cdot\boldsymbol{\nabla}(\boldsymbol{\nabla}\times\boldsymbol{v}) + \nu\nabla^2(\boldsymbol{\nabla}\times\boldsymbol{v}) + (\boldsymbol{\nabla}\times\boldsymbol{v})\cdot\boldsymbol{\nabla}\boldsymbol{v}, \tag{3}$$

so that we know the time derivative of the vorticity $\boldsymbol{\omega}$ at that moment at each point of discretization in the Ω domain, and $\boldsymbol{\omega}$ can be determined at a later time by integrating eq. 3 by means of an ODE integration algorithm.

In this way, the variational formulation of the kinematic Laplacian equation 1 is used as a counterpart of the vorticity transport equation 2 in a vorticity - velocity formulation to approximate the solution of the Navier–Stokes equations. The time integration of the vorticity transport equation produces the distribution of vorticity $\boldsymbol{\omega}$ in the Ω domain. For incompressible flows, the rate of deformation $\mathcal{D}$, i.e. the divergence of velocity is assumed zero.

2.2 The discretization method

The variational formulation of eq. 1 is discretized by the spectral element method as explained in Bursztyn et al. (2008). This method is a particular implementation of the p version of the finite element method (FEM) where the nodes of the elements are located at points of a Gauss–Lobatto grid. This way of locating the nodes of the spectral elements presents an advantage: low-order elements ($p = 1$ or 2) correspond to the classical finite elements with 2 and 3 nodes in each dimension. Thus, there is a variable order method which contains classical low-order elements as special cases. For high order elements, the distribution of nodes according to a Gauss–Lobatto grid is more convenient than classical equidistant spaced nodes (Hourigan et al., 2001). Once the nodes are located in the master element as described in Otero (2008), interpolating functions are constructed as Lagrange polynomials associated with these nodes. Then, their derivatives with respect to natural coordinates are calculated.

The discretization of eq. 1 in each element (Otero and Ponta, 2006; Otero, 2008) can be written as

$$\boldsymbol{\delta}\hat{\mathbf{V}}^{eT}\underbrace{\left(\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{L}}+\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\mathcal{D}}}+\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\omega}}\right)}_{\boldsymbol{K}^{\boldsymbol{e}}}\hat{\mathbf{V}}^{e} = \boldsymbol{\delta}\hat{\mathbf{V}}^{eT}\underbrace{\left(\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{L}}+\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{\omega}}\right)}_{\boldsymbol{R}^{\boldsymbol{e}}}\hat{\boldsymbol{\omega}}^{e}, \tag{4}$$

where

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{L}} = \int_{\Omega^e}\mathbf{B}^{eT}\mathbf{B}^{e}\,\mathrm{d}\Omega = \int_{-1}^{1}\int_{-1}^{1}\mathbf{B}^{eT}\mathbf{B}^{e}\,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\mathcal{D}}} = \int_{-1}^{1}\int_{-1}^{1}\alpha_{\mathcal{D}}\,\mathbf{B}^{eT}\mathbf{m}^{T}\mathbf{m}\mathbf{B}^{e}\,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\omega}} = \int_{-1}^{1}\int_{-1}^{1}\alpha_{\omega}\,\mathbf{B}^{eT}\mathbf{r}^{T}\mathbf{r}\mathbf{B}^{e}\,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

$$\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{L}} = \int_{-1}^{1}\int_{-1}^{1}\mathbf{H}^{eT}\mathbf{B}^{e}_{\boldsymbol{\omega}}\,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

$$\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{\omega}} = \int_{-1}^{1}\int_{-1}^{1}\alpha_{\omega}\,\mathbf{B}^{eT}\mathbf{r}^{T}\mathbf{H}^{e}_{\boldsymbol{\omega}}\,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

and $\boldsymbol{\delta}\hat{\mathbf{V}}^{e}$ is the array of nodal values of the components of an arbitrary field $\boldsymbol{\delta v}$.

The element matrices $\boldsymbol{K}^e$ and $\boldsymbol{R}^e$ are assembled in the respective global matrices to obtain the global system that represents the discretization of the KLE

$$\boldsymbol{K}\ \hat{\mathbf{V}} = \boldsymbol{R}\ \hat{\boldsymbol{\omega}}. \tag{5}$$

To evaluate the right-hand-side of the system of ODE describing the temporal evolution, eq. 5 is rewritten as

$$\frac{\partial \boldsymbol{\omega}}{\partial t} = \boldsymbol{\mathcal{F}}\left(\boldsymbol{\omega}, t\right) = \boldsymbol{\nabla} \times \left(\nu\ \boldsymbol{\nabla}\cdot\boldsymbol{\nabla}\boldsymbol{v} - \boldsymbol{v}\cdot\boldsymbol{\nabla}\boldsymbol{v}\right), \tag{6}$$

and after the discretization, it results

$$\boldsymbol{\mathcal{F}}(\hat{\boldsymbol{\omega}}, t) = \hat{\boldsymbol{C}}_{\boldsymbol{url}} \left(\nu\ \hat{\boldsymbol{D}}_{\boldsymbol{iv}} - \hat{\boldsymbol{V}}_{\boldsymbol{adv}}\right) \hat{\boldsymbol{G}}_{\boldsymbol{rad}} \hat{\mathbf{V}}. \tag{7}$$

3 THE FRAMEWORK

We use the framework presented in Quinteros et al. (2007) to implement numerical models based in finite element method. This framework is based on a class structure that represents each of the entities that are often part of a problem solved by the FEM. According to the concepts of object-oriented programming, each class provides a specific functionality that clearly represents an object. The developer must take care that each object will provide the consistent functionality expected in this type of implementations. A detailed description of the characteristics of the framework can be found in Quinteros (2008), whereas here we only mention some of them.

In this work we use a particular type of finite element called spectral element to implement the KLE. These type of elements had not been included in the original version of the framework, so an extension was included in the class structure (Bursztyn et al., 2008).

3.1 General characteristics of the class structure

One of the major purposes of the FEM is to define a base of functions, based on subdividing the domain into subregions called finite elements. Each element is composed of nodes. In the framework, the domain is represented by the `Domain` class which includes all necessary information to represent the geometry to be modeled by a set of elements and nodes stored in the `Elements` and `Nodes` attributes. Each element contains references to nodes that belong to it, storing in the `idNodes` attribute the corresponding global identification numbers. To calculate the stiffness matrix of each element, we must evaluate the interpolating functions and their derivatives at the Gauss points determined by the order and type of the element used. This information is represented by the `GaussPoint` class.

Figure 1 shows a schematic plot of a domain composed by 2 elements. These elements are defined by 4 nodes located in the corners. Also, there are 4 Gauss points where the numeric interpolation is calculated. Figure 2 shows part of the class diagram that provides the framework. One can see the classes `Domain`, `Element`, `Node`, `GaussPoint` and `Boundary` among others.

In order to implement the KLE method by using spectral elements, a `Spectral` class was created to implement the spectral elements with configurable number of nodes. This class, according to the framework structure, inherits from `Element` all its properties and methods.

3.2 Libraries used by the framework

The framework isolates the linear algebra operations so that they are carried out by means of external libraries with optimum performance and independently from the implementation

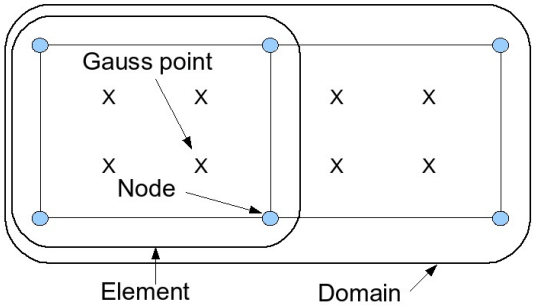


Figure 1: Domain composed of two 4-node elements with their Gauss points (modified from Quinteros et al., 2007).

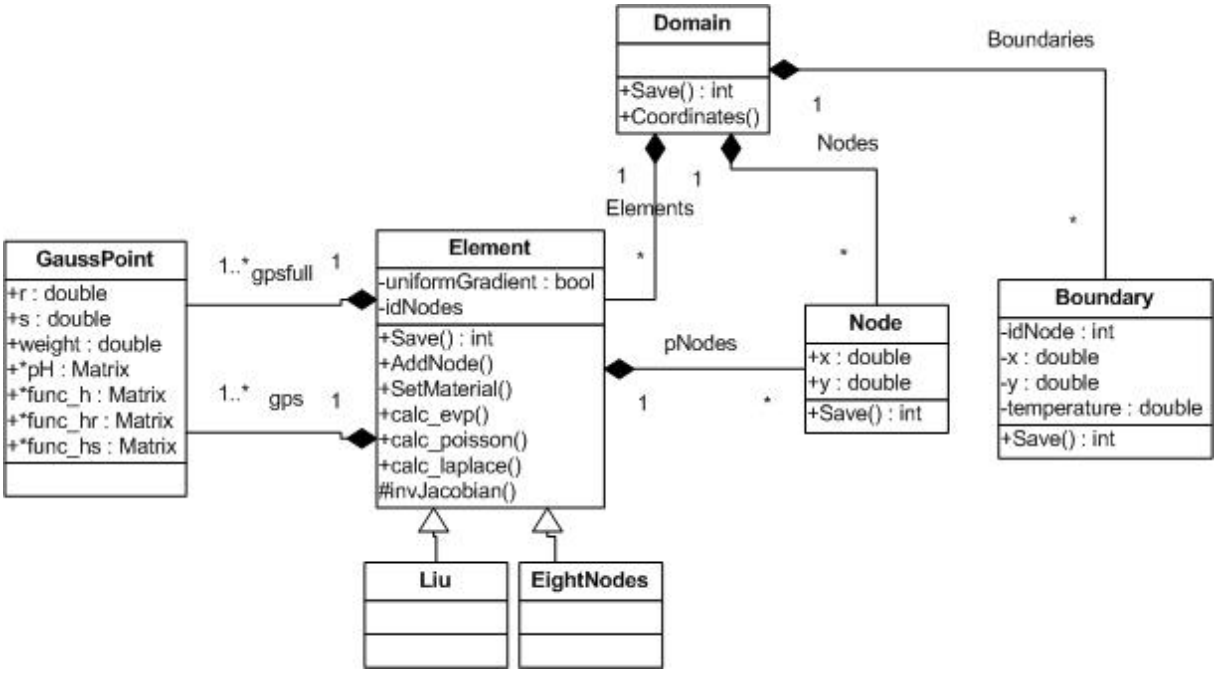


Figure 2: Class diagram.

of the FEM in particular. The stiffness matrix associated with each instance of the `Element` class is computed using dense matrices related to geometric properties of the element. A class called `Matrix` provides a simple interface to operate with dense matrices, including the most common operations needed, by means of operator overloading and polymorphism. In this case, the algebraic operations are performed by the Lapack library[1] (Anderson et al., 1999).

Another class called `SparseMatrix` is designed to improve memory management and performance in operations with large sparse matrices, as it is often the case of global stiffness or mass matrices. Like the `Matrix` class, it separates the user from the actual implementation of algebraic operations, which in this case could be either `SuperLU`[2] (Demmel et al., 1999) or `Pardiso`[3] (Schenk et al., 2001).

[1] http://www.netlib.org/lapack/
[2] http://crd.lbl.gov/~xiaoye/SuperLU/
[3] http://www.pardiso-project.org/

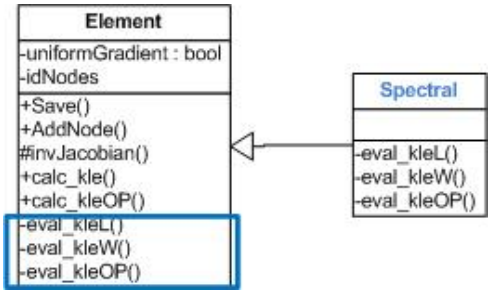


Figure 3: Class `Element` y `Spectral`.

3.3 Boundary condition parser

A new interface to a parser called `muParser`[4] was implemented to calculate dynamically the boundary conditions defined by a multi-variable-dependent function. It computes at runtime the value of the function from its definition. `muParser` interprets any string as a function, receiving the information of the variables, constants and functions to use. In this case, the variables defined were the spatial coordinates (x, y) and time (t). This extension allows to define more complex problems in a simple way. The model receives as a parameter the name of a configuration file where the vorticity w, the boundary conditions on the velocity and its partial derivatives, as a function of time, are defined.

4 KLE METHOD IMPLEMENTATION

The methods *calc_kle* and *calc_kleOP* which implement the KLE method were added to the `Element` class, as shown in figure 3. The methodology explained in Bursztyn et al. (2008), where a `calc_equation` method invokes another method called `eval_equation`, was maintained (here `equation` is the name which defines the problem).

The *calc_kle* method is responsible for carrying out the integration at Gauss points for the matrices of eq. 4. It calculates the element matrices $\boldsymbol{K}^e$ and $\boldsymbol{R}^e$ by means of the `eval_kleL` and `eval_kleW` methods, that perform the operations related to the basic Laplacian formulation and the penalty terms of vorticity and divergence of the velocity, respectively. The *calc_kleOP* method also performs the integrations in the Gauss points, but for the differential operators needed in eq. 7. This method calls the `eval_kleOP` method to calculate the matrices evaluated at the integration points necessary to build the derivation operators, which are used to evaluate the right-hand-side of the ODE system in successive time steps (Ponta, 2005; Otero and Ponta, 2006).

Figure 4 shows the KLE method steps which will be explained in this section.

4.1 Domain discretization

This implementation accepts as input data text files with a format similar to the one used by the *GID*[5] application. Through this file, an instance of the `Domain` class which represents the domain is created, with all the elements, nodes and boundary conditions. Finally, using the `GaussPointGenerator` class the Gauss points are generated and the shape functions and its derivatives calculated. Gauss points and all the associated information are stored in the static variables `gpsfull` and `gps`, as already explained in Bursztyn et al. (2008).

[4] http://muparser.sourceforge.net/
[5] http://www.gidhome.com/

- Domain discretization
- Matrices structure generation
- For each element
  - Calculation of the element matrices and integration in the Gauss points
  - Assembly of global matrices
  - Calculation of the differential operators
- Boundary condition imposition
- Solution of the system of equations

Figure 4: KLE method diagram.

4.2 Matrices structure generation

The matrices $\boldsymbol{K}$ and $\boldsymbol{R}$ in eq. 5 usually have a small amount of non-zero values. These matrices are represented as instances of the `SparseMatrix` class and their structure is built based on the connectivity list that relates elements and nodes. Similarly, the empty structures of `BRot`, `BGrad` and `BDiv`, which represent necessary arrays to calculate the differential operators of eq. 7, are built.

After this step, the assembly of the element matrices in the global matrix is substantially improved, as all the positions were previously allocated and structured, avoiding memory blocks reallocation during this stage.

4.3 Elemental integration and global assembly

As shown in figure 4, for each element, element matrices are calculated. To this end, we integrate in the Gauss points by means of the `calc_kle` method. Using `eval_kleL` and `eval_kleW` methods the corresponding integrations to calculate these matrices are performed (for details see eq. 4). Taking as example two of these matrices

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{L}} = \int_{\Omega^e} \mathbf{B}^{eT}\mathbf{B}^e \,\mathrm{d}\Omega = \int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^{eT}\mathbf{B}^e \,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

$$\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{L}} = \int_{-1}^{1}\int_{-1}^{1} \mathbf{H}^{eT}\mathbf{B}^e_{\boldsymbol{\omega}} \,|\boldsymbol{J}|\,\mathrm{d}r\mathrm{d}s,$$

the `calc_kle` method uses for each Gauss point the `eval_kleL` method to calculate $\mathbf{B}^{eT}\mathbf{B}^e|\boldsymbol{J}|$ and $\mathbf{H}^{eT}\mathbf{B}^e_{\boldsymbol{\omega}}\,|\boldsymbol{J}|$ and sums up the result multiplied by the weight corresponding to the point of integration for the double integral, to compute $\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{L}}$ and $\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{L}}$. By the summation of $\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{L}}$, $\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{D}}$, $\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\omega}}$, $\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{L}}$ and $\boldsymbol{R}^{\boldsymbol{e}}_{\boldsymbol{\omega}}$ the element matrices are obtained.

Once obtained the element matrices $\boldsymbol{K}^{\boldsymbol{e}}$ and $\boldsymbol{R}^{\boldsymbol{e}}$, they are assembled into the global matrices which are sparse. `SparseMatrix` class has an interface that facilitates the operation

$$\boldsymbol{K} = \sum_e \boldsymbol{K}^{\boldsymbol{e}} \tag{8}$$

where the summatory implies a mapping between elemental and global numbering of the nodes. The same is done with the global matrix $\boldsymbol{R}$ and element matrices $\boldsymbol{R}^{\boldsymbol{e}}$.

4.4 Calculation of the differential operators

For the temporal integration, differential operators $\hat{\boldsymbol{C}}_{\boldsymbol{url}}$, $\hat{\boldsymbol{G}}_{\boldsymbol{rad}}$ and $\hat{\boldsymbol{D}}_{\boldsymbol{iv}}$ are calculated in order to be used in the evaluation of the right-hand-side of the vorticity transport equation 7. These operators are calculated with full integration over Gauss points, carried out by the `calc_kleOP` method from the `Element` class. This method has its corresponding `eval_kleOP` method. Finally, the element matrices are assembled into global operator matrices. With the purpose of optimizing the process time, this calculation is performed within the same element cycle along with the calculation of the matrices $\boldsymbol{K}$ and $\boldsymbol{R}$ as shown in figure 4 so that only one pass through all the elements is required.

5 IMPROVEMENTS IN B.C. IMPOSITION AND SYSTEM RESOLUTION

In the previous versions, once the global matrices were assembled, boundary conditions were imposed in order to solve the system expressed by eq. 5. The boundary conditions and the vorticity vector $\hat{\boldsymbol{\omega}}$ are part of the input information of the problem. If we define $\boldsymbol{F} = \boldsymbol{R}\,\hat{\boldsymbol{\omega}}$, the problem can be expressed as

$$\boldsymbol{K}\,\hat{\mathbf{V}} = \boldsymbol{F}, \tag{9}$$

where $\hat{\mathbf{V}}$ is the vector which contains the unknowns we want to obtain after solving the system.

Depending on the size of the problem, different stages of the resolution can be the most time-consuming. In particular, the assemblage of element/global matrices, imposition of boundary conditions and resolution of the system of equations. We tried to improve two of them in order to reduce the computational time and increase the overall performance.

We compared two different ways of assembling and imposing boundary conditions. The original one was to assemble the element matrices $\boldsymbol{K}^{\boldsymbol{e}}$ and $\boldsymbol{R}^{\boldsymbol{e}}$ into the global ones $\boldsymbol{K}$ and $\boldsymbol{R}$. Then, boundary conditions were imposed overwriting the values of $\boldsymbol{K}$ and $\boldsymbol{F}$ where needed. A second way is to impose the boundary conditions at the moment of the element matrix calculation. The matrix $\boldsymbol{K}$ is split in four submatrices to separate the degrees of freedom that were imposed by boundary conditions from those which were not. Matrix $\boldsymbol{R}$ is split into two submatrices so that $\boldsymbol{F}$ is decomposed in an analogous way. Namely,

$$\boldsymbol{K} = \begin{bmatrix} K_{uu} & K_{uk} \\ K_{ku} & K_{kk} \end{bmatrix}, \qquad \boldsymbol{F} = \begin{bmatrix} F_u \\ F_k \end{bmatrix}, \tag{10}$$

where the subscript k represents the known degrees of freedom, related to Dirichlet boundary conditions, and the subscript u represents the unknown degrees of freedom.

Then, equation 9 can be expressed as

$$\begin{bmatrix} K_{uu} & K_{uk} \\ K_{ku} & K_{kk} \end{bmatrix} * \begin{bmatrix} V_u \\ V_k \end{bmatrix} = \begin{bmatrix} F_u \\ F_k \end{bmatrix}, \tag{11}$$

where V_k is the velocity array containing the known (input) values, V_u is the velocity array containing the unknown values and F_u and F_k contain the values of F corresponding to the degrees of freedom whose values are unknown and known respectively.

It is not necessary to calculate the matrices K_{ku}, K_{kk} and R_k from eq. 10 because it is enough to solve just the first line of eq. 11. Thus, while in the original implementation 2 big matrices are assembled, only 3 smaller matrices are assembled in this new approach. Immediate benefits are a decrease of allocated memory (allowing to solve bigger problems) and the execution time.

At the resolution stage, we tested and compared a second version of the `SparseMatrix` class (Quinteros et al., 2009) based on the `Pardiso` solver. The rest of the code remained unchanged because the class declaration was the same.

6 RESULTS

6.1 Model setup

All the results shown in this section were obtained on a PC with a processor Intel Core 2 Quad 2.4 Ghz and 8GB RAM with Debian GNU/Linux kernel 2.6.32-3.

In this work we used a two-dimensional regular mesh with N_{el} elements in each direction. Each element, according to the order p, have $N_{GL} = p+1$ nodes in each direction, giving a total of ${N_{GL}}^2$ nodes per element. N^* is the inverse of the mean internodal distance, such that the total number of nodes in the mesh is $(N_{GL}+1)^2$. The problem was solved by changing the order of the elements (modifying the number of nodes in the element), as well as the number of elements within the mesh in order to study the behavior regarding p and h refinement respectively.

6.2 Validation of results

To validate the results obtained by this implementation, we used a velocity field given by the `error function` which corresponds to the solution of the flow over an infinite flat plate (see Sec. 4.3 from Batchelor, 2000, among others) which has been proposed as a benchmark for vorticity-velocity methods by Otero and Ponta (2006). First, we defined the velocity from which the vorticity was calculated. Then, imposing that vorticity in the whole domain and the given velocity in the mesh boundaries, eq. 5 was solved to recover the velocities which were compared against the original values. The exact solution of the streamwise velocity for different values of the time parameter $\tau = \sqrt{4\,\nu\,t}/Y$ is shown in figure 5.

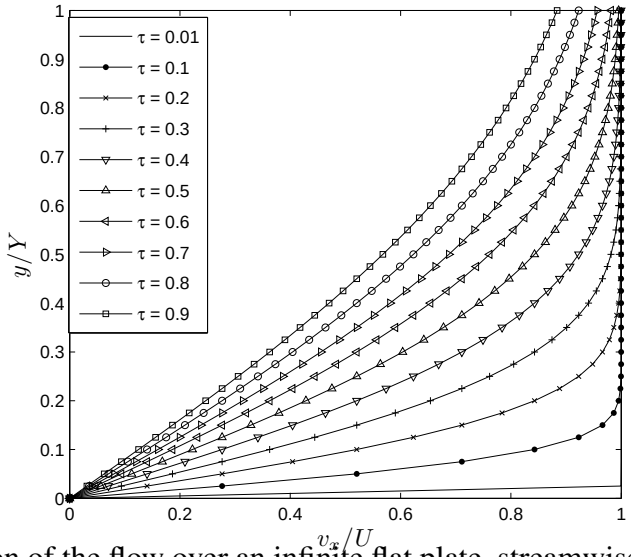


Figure 5: Exact solution of the flow over an infinite flat plate, streamwise velocity (see Sec. 4.3 from Batchelor, 2000, among others).

Validation results were already shown in Bursztyn et al. (2008). There, one can see the expected spectral convergence for the p refinement in comparison with the normal expected convergence of the classical FEM approach and h refinement, which shows a straight line of slope equal to the order p of the element in logarithmic plot.

Similar experiments were performed to validate the accuracy of the calculation of differential operators. It was observed that the h refinement gives a straight line whose slope decreased in magnitude with the order of the derivatives. In contrast, the curves associated with the p

refinement showed the typical behavior of spectral convergence (Boyd, 2000). Also, it was found that error increased as a higher order derivatives were computed.

6.3 Performance of the different approaches

The different implementations were tested varying the solver and the method for assembling and imposing the boundary conditions used, as explained in section 4. The first version, which assembles into 2 matrices and uses the `SuperLU` library, is the one already presented in Bursztyn et al. (2008). In the second version (B.C. in E.M.), the assemblage is made into 3 smaller matrices and the boundary conditions are imposed on the element matrices. The third version (B.C. in E.M. + Pardiso) is similar to the second, but using the new `SparseMatrix` class, which uses the `Pardiso` solver.

Figure 6 shows the total execution time for each approach varying the number of elements of order $p = 2$ (h refinement). Other elements with different orders behave in the same manner. p refinement is not shown because when the size of the problem increases both approaches tend to be exactly the same, as the domain is formed by a small number of elements. It can be seen that the new BC imposition has a better performance than the previous one and that the difference becomes bigger when the size of the problem increases. When `SuperLU` is replaced by `Pardiso` the performance is again enhanced, showing a better behavior for the whole range of experiments. Moreover, given that the slope of the third version is significantly lower than that of the version of Bursztyn et al. (2008), it will have better scaling capabilities and could perform much better in problems bigger than those exposed here.

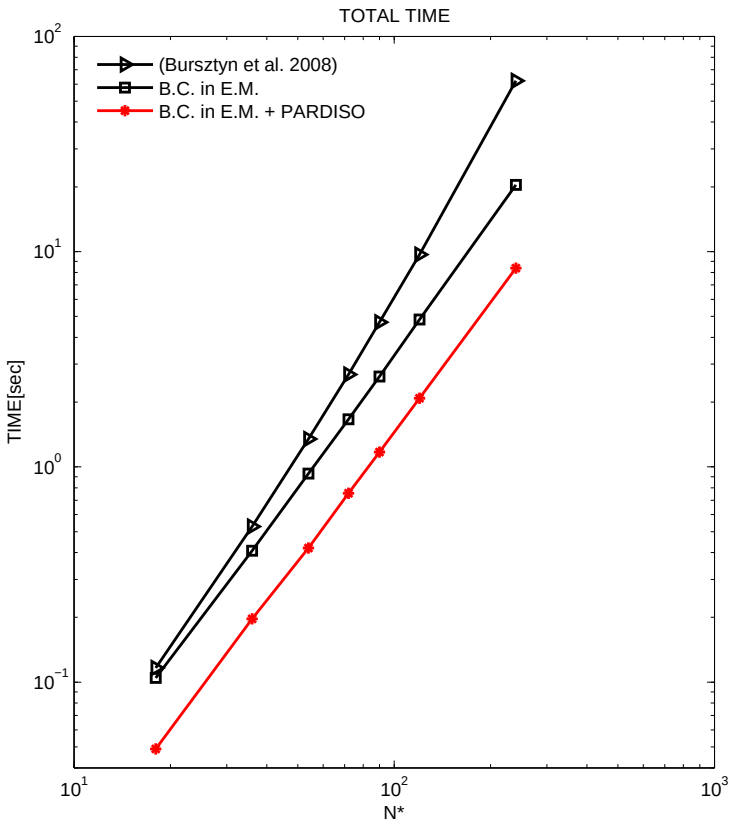
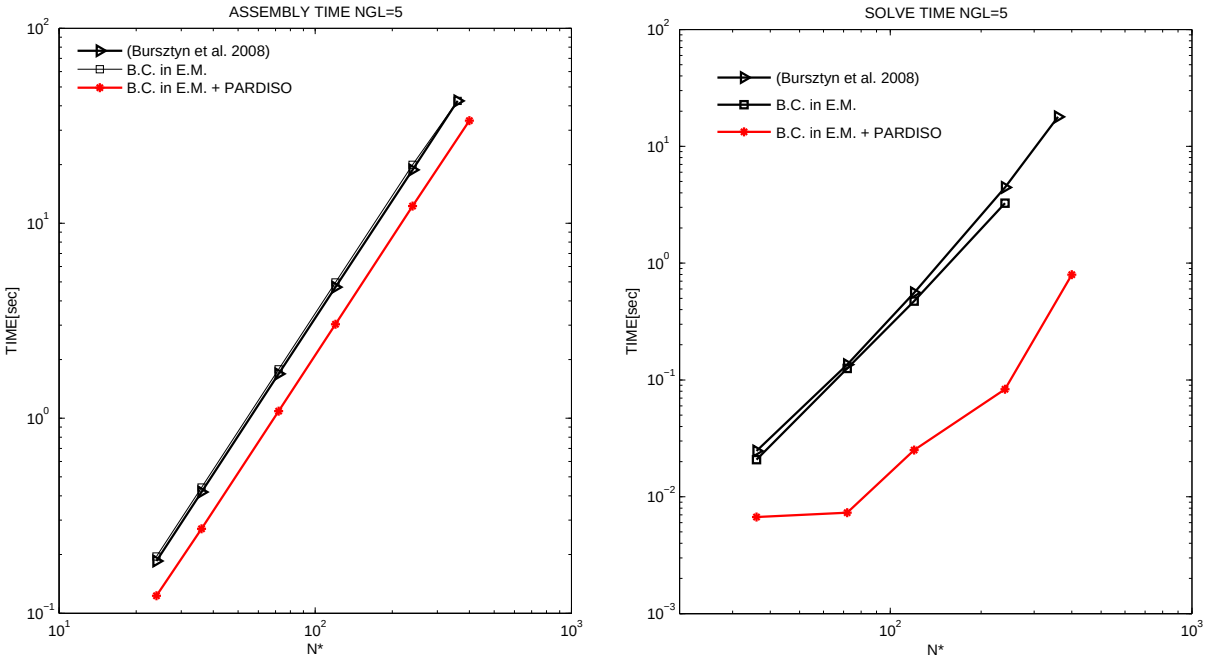


Figure 6: Total execution time for the different versions using a grid with elements of order $p = 2$ varying the number of elements.



(a) (b)

Figure 7: Time spent in different implementations using a grid with elements of order $p = 4$ varying the number of elements for the assembling (a) and solving the system (b).

It is important to mention that `SuperLU` and `Pardiso` solvers do not use the same sparse matrix representation in terms of memory allocation. Thus, if we use the same method to assemble and to impose the boundary conditions, they will not necessarily use the same amount of time. A similar situation happens when we keep the same solver but change the method to impose the BC, as we need to assemble 3 smaller matrices compared with the 2 original ones, which means that the final system of equations will differ. This means that an influence between these two stages will be seen in the results.

As an example of this, it can be seen in fig. 7a, which shows the time to assemble for a grid composed by elements of order $p = 4$, that the time needed to assemble the global matrices is reduced almost by a factor of 2. This is related to the different storage specifications of `SuperLU` and `Pardiso`. While the former needs to store all the non-zero values in memory, the latter allows to store only those in the upper half part in case of a symmetric matrix. It can be seen in the same figure that when the same solver (`SuperLU`) is used, the overhead in time due to the inclusion of the BC imposition in the assemblage is negligible, despite of the size of the problem. The advantage in this case is the elimination of the boundary condition imposition step over the assembled matrices.

The time needed to solve the system of equations for each approach is shown in fig. 7b. The second approach shows a better performance than the previous one when the problem is small, but the difference slowly vanishes when the size of the problem increases. This is a direct cause of the size of the system of equations to solve, which is reduced by splitting the matrix K in submatrices related to the known and unknown degrees of freedom (see eq. 11). However, when the problem size increases, the number of degrees of freedom related with BC increases linearly, while the unknowns increase quadratically. Due to this, the difference in computational time will be less important for bigger problems.

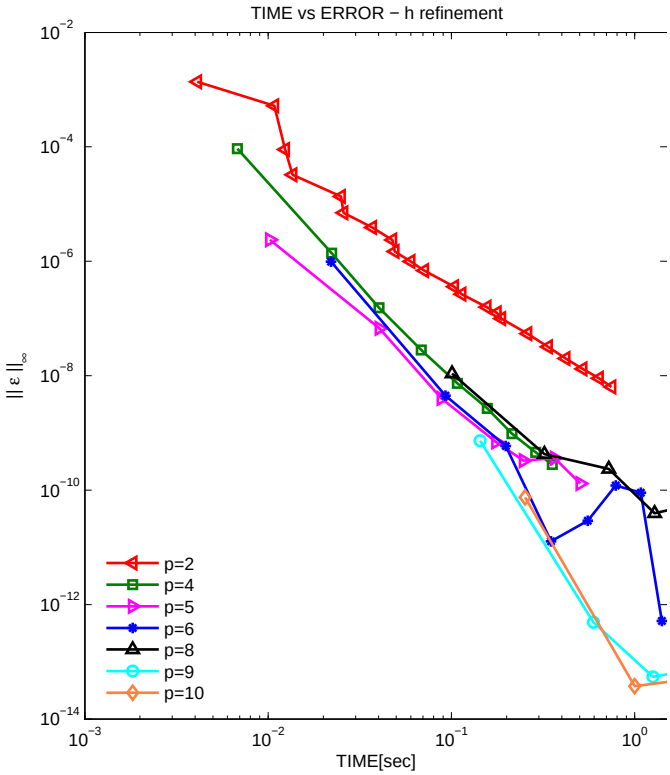


Figure 8: Error as a function of total computational time for elements of different order.

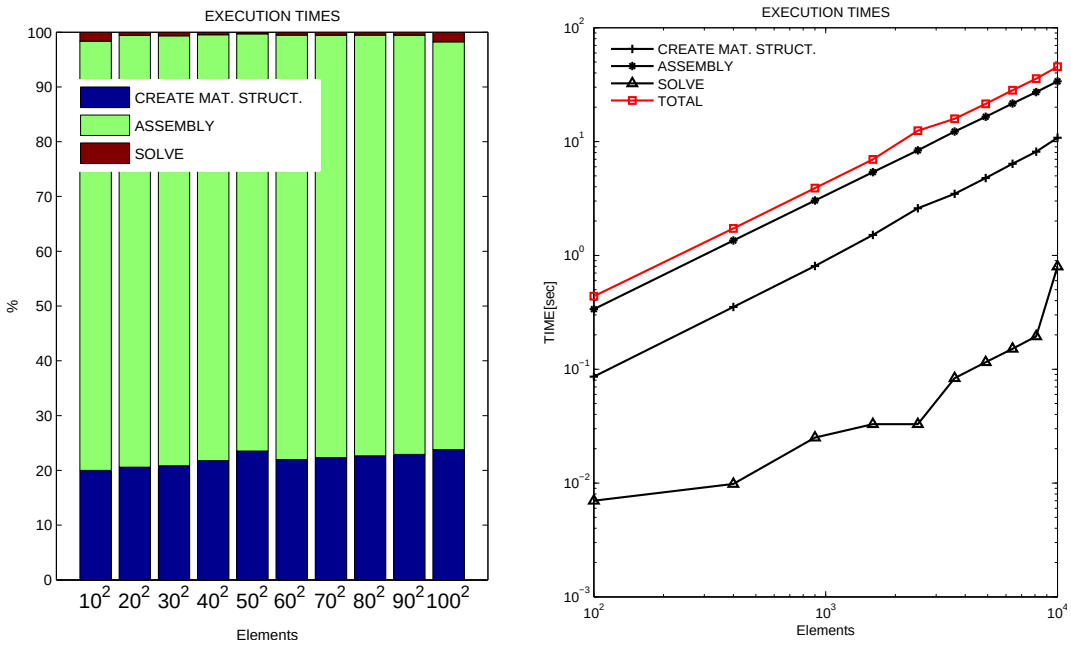
On the other hand, the performance with the inclusion of `Pardiso` enhances the computational time by more than one order of magnitude. The cause of this improvement is the reduced storage required by `Pardiso` when the global stiffness matrix is symmetric and positive definite.

These results lend us to infer that the imposition of BC in the assemblage and the use of `Pardiso` solver seem to be the best approach, so in the next sections we will study how it behaves and compares against the previous KLE implementations.

6.4 Influence of the element order on precision and execution time

An interesting performance measure is the relation between precision and execution time for a given problem size. In order to test the order p that performs the best, the problem was run for different meshes with increasing number of elements with the same order. Figure 8 shows curves of constant order p for different meshes. It can be seen that to achieve the same numerical precision (error), more computational time is needed when low order elements ($p = 2$) are used, in comparison with medium or high order ($p \geq 4$) elements, which show some sort of convergence to a performance limit. An improvement in performance can be achieved increasing the order up to $p \approx 6$, but no further benefits are obtained for higher order, at least in terms of computational time. This behavior deserves a deeper study to evaluate if this keeps on happening for different solution smoothness degree. There is also an apparently anomalous behavior for $p = 8$. Another performance measure that could be considered and we plan to study in the future is the relation between precision and memory requirements. According to that it might be useful to work with orders higher than 6.

To further study the contribution of each step of the problem solution, we split the execution time in 3 different stages: the matrix structure creation as explained in section 3 (CREATE



(a) (b)

Figure 9: Execution times in meshes composed by elements of $p = 4$ order in percent of total time (a) and in logarithmic scale (b).

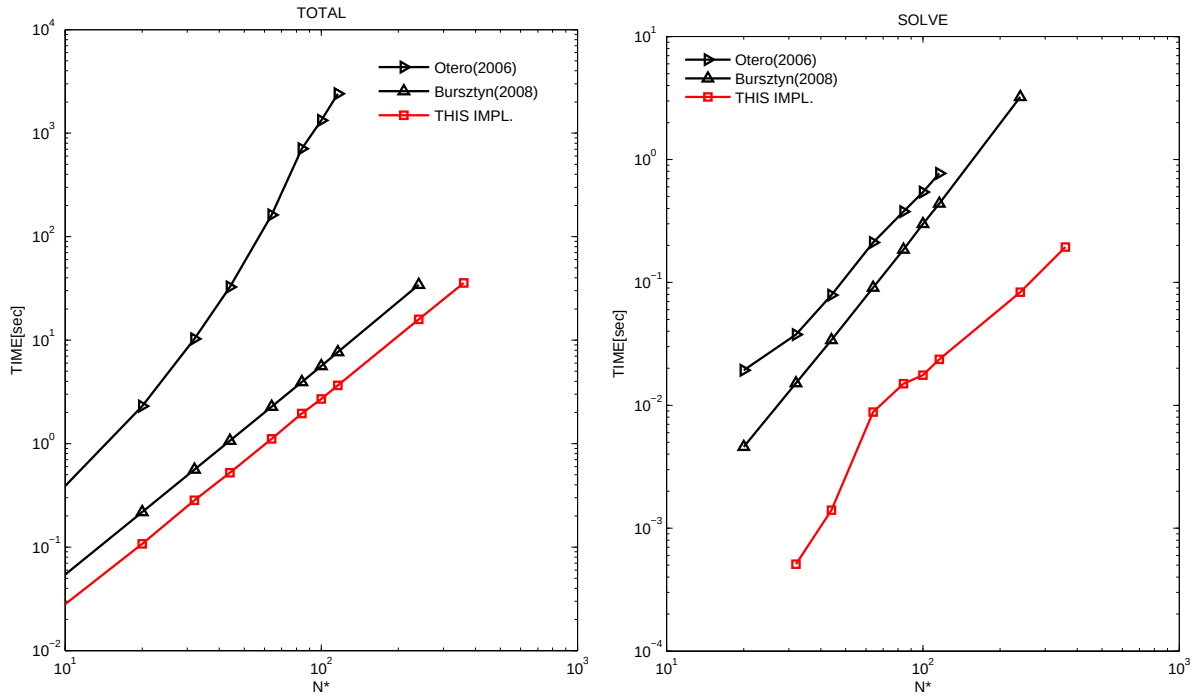
MAT. STRUCT.), the assembly time to build the matrices from eq. 5 imposing the boundary conditions (ASSEMBLY) and the time to solve the system of equations 5 (SOLVE).

Figure 9 shows the contribution to the total time of each stage in meshes of increasing number of elements of order $p = 4$. The contribution of each stage is almost constant in the range of problems shown. Figure 9a shows the percent of total time while figure 9b shows the time, in logarithmic scale, spent at each stage.

The assembly time is the higher among the three stages. Due to the way that KLE method works, this assemble operation needs to be performed only at the beginning of a simulation or when the shape of the mesh changes, since all matrices assembled during this stage depend mainly on the geometry. Regarding the creation of the matrix structures, it only needs to be done if the connectivity of the mesh is modified, which is something very unusual during a simulation. Something important related to the time for solving the system of equations is that in the first resolution, it is necessary to reorder and compute the triangular factors of the global matrix. In the case that the geometry of the mesh does not change all along a simulation, these factors can be stored and used in successive resolutions saving the corresponding amount of time. In the case that the geometry of the mesh does change, but its connectivity does not, the reordering can be reused, which also allows to save time. The solution time presented in the results in this section always includes the reordering and factorization time mentioned.

6.5 Comparison with previous implementations

Several experiments with identical meshes were run in order to compare the implementation presented by Otero and Ponta (2006) and the one presented by Bursztyn et al. (2008) with this implementation. The main difference is that the former was completely implemented in Matlab. Between both implementations in C++ the principal differences are the way boundary conditions are imposed and the solver used to find the solution of the system of equations. The



(a) (b)

Figure 10: Execution time comparing the previous implementations and this implementation using meshes of elements with $p = 4$ order: (a) total and (b) solve.

problem was solved for meshes with elements of different order varying the number of elements in the mesh. Results for meshes composed by elements of order $p = 4$ are shown in fig. 10. As the main contribution to the total time comes from the assemblage stage we present, in fig. 10a, results of total time, which can be directly related to the assemble. On the other hand, the solve time becomes almost negligible when compared with the total, so results for that stage are shown alone in fig. 10b.

Figure 10 shows that for small problems Bursztyn et al. (2008) implementation needs twice the time that the implementation presented here. This relation is kept almost constant in the whole range shown. Comparing Otero and Ponta (2006) implementation, it can be seen that it takes one order of magnitude more time than this implementation for small problems, while for medium sized problems it takes almost three orders of magnitude more. Regarding the solution of the system of equations this implementation is considerably better than both implementation, improving by at least one order of magnitude with respect to Bursztyn et al. (2008) implementation and one and a half order of magnitude with respect to Otero and Ponta (2006) implementation. Both behaviors are strongly related to the capability of assembling only the half of the stiffness matrix.

A convenient way to evaluate the scalability of each implementation is to study the slope when plotting the time against the size of the problem in a logarithmic plot. These slopes represent the order of dependence of time respect to the problem size. In this case, as for general nonstructured meshes the parameter N^* has no direct meaning, it is better to use the total number of nodes in the mesh which, roughly, grows as the square of N^*. Thus, the slopes will be almost half of those seen in plots against N^*. In table 1, the slopes measured for different element orders are presented. Results are given for individual stages and for the total time.

	ASSEMBLE + B.C.			SOLVE			TOTAL		
NGL	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
3	2.0	1.5	1.0	1.6	1.7	1.2	2.0	1.4	1.0
5	2.1	1.4	1.0	1.8	1.7	1.1	2.2	1.4	1.0
9	2.0	1.1	1.0	1.6	1.2	1.0	2.0	1.1	1.1

Table 1: Slopes in logarithmic plots of time vs total number of nodes for: (a) Otero and Ponta (2006); (b) Bursztyn et al. (2008); (c) this implementation.

In table 1 it can be seen that slopes of Otero and Ponta (2006) implementation almost double those of the implementation presented herein while those of Bursztyn et al. (2008) implementation are about 40 – 50 % higher in the assemble and boundary condition imposition, about 50 % higher in the solution and about 40 % higher in total. In view of these results, this implementation is clearly the best option at the moment, considering the scalability in bigger problems.

7 CONCLUSIONS AND FUTURE WORK

In this work, a new implementation of the KLE method was presented which improves the performance of previous implementations. Two main modifications were added, namely, the way boundary conditions are imposed and the library used to solve the sparse system of equations. The boundary conditions are now imposed at elemental level before assembling global matrices. `Pardiso` library was used to replace `SuperLU` in the solution of the sparse system of equations. With these two modifications a better memory use and a fast resolution is achieved. From the analysis of time measurements for different element orders it can be concluded that significant time savings are obtained. In the assemble stage, this is mainly due to the possibility of storing only one triangular part of the global stiffness matrix. This also helps to reduce the time needed to solve the system.

The scalability of this new version was also analyzed and compared with the previous ones. The dependence of the time required on the problem size, represented by the slopes in logarithmic plots, shows an improvement on the scalability in every stage as well as the complete process. This indicates that this version is better suited for increasing problem sizes and as a base of future parallel implementations.

Work has already been initiated to incorporate an ODE solver to the C++ framework. As the code is already capable of computing the differential operator matrices required to evaluate the right hand side of equation 7, after finishing this task, this implementation will be ready to simulate time dependent bidimensional incompressible flows. Possible extensions in the future could be towards the solution of three dimensional problems, turbulent flows by the addition of a subgrid model in a large eddies simulation formulation and complex multiphysics problems by the coupling with ODEs representing other phenomena.

REFERENCES

Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., and Sorensen D. *LAPACK users' guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

Batchelor G.K. *An introduction to fluid dynamics*. Cambridge University Press, Cambridge, UK, 2000.

Boyd J.P. *Chebyshev and Fourier spectral methods*. Dover, Mineola, New York, USA, 2000.

Bursztyn G.H., Otero A.D., and Quinteros J. Una nueva implementación para las ecuaciones de navier–stokes mediante kle y elementos espectrales. *ENIEF 2008 - XVII Congreso sobre Métodos Numéricos y sus Aplicaciones*, pages 2367–2383, 2008.

Demmel J.W., Eisenstat S.C., Gilbert J.R., Li X.S., and Liu J.W.H. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

Hourigan K., Thompson M.C., and Tan B.T. Self–sustained oscilations in flows around long blunt plates. *J. Fluids Struct.*, 15:387–398, 2001.

Otero A.D. *Análisis no-lineal del comportamiento estructural de sistemas avanzados de conversión eoloeléctrica*. Ph.D. thesis, Universidad de Buenos Aires, 2008.

Otero A.D. and Ponta F.L. Spectral–element implementation of the KLE method: a $(\omega, \mathbf{v})$ formulation of the Navier–Stokes equations. *Mecánica Computacional*, 25:2649–2667, 2006.

Ponta F.L. The kinematic Laplacian equation method. *J. Comput. Phys.*, 207:405–426, 2005.

Quinteros J. *Numerical modeling of an Andean system and its responseto climatic and rheological variations*. Ph.D. thesis, Universidad de Buenos Aires, 2008.

Quinteros J., Jacovkis P.M., and Ramos V.A. Diseño flexible y modular de modelos numéricos basados en elementos finitos. *Mecánica Computacional*, 26:1724–1740, 2007.

Quinteros J., Ramos V.A., and M.Jacovkis P. An elasto-visco-plastic model using the Finite ElementMethod for crustal and lithospheric deformation. *Journal of Geodynamics*, 48:83–94, 2009. doi:10.1016/j.jog.2009.06.006.

Schenk O., Gaertner K., Fichtner W., and Stricker A. Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18:69–78, 2001.