Nektar++: Spectral/hp Element Framework

Version 4.3.2

User Guide

June 4, 2016

Department of Aeronautics, Imperial College London, UK
Scientific Computing and Imaging Institute, University of Utah, USA

Contents

Introduction

Nektar++ [7] is a tensor product based finite element package designed to allow one to construct efficient classical low polynomial order h-type solvers (where h is the size of the finite element) as well as higher p-order piecewise polynomial order solvers. The framework currently has the following capabilities:

- Representation of one, two and three-dimensional fields as a collection of piecewise continuous or discontinuous polynomial domains.
- Segment, plane and volume domains are permissible, as well as domains representing curves and surfaces (dimensionally-embedded domains).
- Hybrid shaped elements, i.e triangles and quadrilaterals or tetrahedra, prisms and hexahedra.
- Both hierarchical and nodal expansion bases.
- Continuous or discontinuous Galerkin operators.
- Cross platform support for Linux, Mac OS X and Windows.

The framework comes with a number of solvers and also allows one to construct a variety of new solvers.

Our current goals are to develop:

- Automatic auto-tuning of optimal operator implementations based upon not only h and p but also hardware considerations and mesh connectivity.
- Temporal and spatial adaption.
- Features enabling evaluation of high-order meshing techniques.

For further information and to download the software, visit the Nektar++ website at http://www.nektar.info.

Part I

Getting Started

Chapter 1

Installation

Nektar++ is available in both a source-code distribution and as pre-compiled binary packages for a number of operating systems. We recommend using the pre-compiled packages if you wish to use the existing Nektar++ solvers for simulation and do not need to perform additional code development.

1.1 Installing Debian/Ubuntu Packages

Binary packages are available for current Debian/Ubuntu based Linux distributions. These can be installed through the use of standard system package management utilities, such as Apt, if administrative access is available.

1. Add the appropriate line for the Debian-based distribution to the end of the file `/etc/apt/sources.list`

Distribution	Repository
Debian 8.0 (jessie)	`deb http://www.nektar.info/debian-jessie jessie contrib`
Debian 7.0 (wheezy)	`deb http://www.nektar.info/debian-wheezy wheezy contrib`
Ubuntu 14.04 (trusty)	`deb http://www.nektar.info/ubuntu-trusty trusty contrib`

2. Update the package lists

```
apt-get update
```

3. Install the required Nektar++ packages, or the complete suite with:

```
apt-get install nektar++
```

   Any additional dependencies required for Nektar++ to function will be automatically installed.

Tip

Nektar++ is split into multiple packages for the different components of the software. A list of available Nektar++ packages can be found using:

```
apt-cache search nektar++
```

1.2 Installing Redhat/Fedora Packages

Add a file to the directory `/etc/yum.repos.d/nektar.repo` with the following contents

```
[Nektar]
name=nektar
baseurl=<baseurl>
```

substituting `<baseurl>` for the appropriate line from the table below.

Distribution	`<baseurl>`
Fedora 20	`http://ww.nektar.info/fedora/20/$basearch`

Note

The `$basearch` variable is automatically replaced by Yum with the architecture of your system.

1.3 Installing from Source

This section explains how to build Nektar++ from the source-code package.

Nektar++ uses a number of third-party libraries. Some of these are required, others are optional. It is generally more straightforward to use versions of these libraries supplied pre-packaged for your operating system, but if you run into difficulties with compilation errors or failing regression tests, the Nektar++ build system can automatically build tried-and-tested versions of these libraries for you. This requires enabling the relevant options in the CMake configuration.

1.3.1 Obtaining the source code

There are two ways to obtain the source code for *Nektar++*:

- Download the latest source-code archive from the Nektar++ downloads page.

- Clone the git repository
  - Using anonymous access. This does not require credentials but any changes to the code cannot be pushed to the public repository. Use this initially if you would like to try using Nektar++ or make local changes to the code.

```
git clone http://gitlab.nektar.info/clone/nektar/nektar.git nektar++
```

  - Using authenticated access. This will allow you to directly contribute back into the code.

```
git clone git@gitlab.nektar.info:nektar/nektar.git nektar++
```

Tip
You can easily switch to using the authenticated access from anonymous access at a later date.

1.3.2 Linux

1.3.2.1 Prerequisites

Nektar++ uses a number of external programs and libraries for some or all of its functionality. Some of these are *required* and must be installed prior to compiling Nektar++, most of which are available as pre-built *system* packages on most Linux distributions or can be installed manually by a *user*. Others are optional and required only for specific features, or can be downloaded and compiled for use with Nektar++ *automatically* (but not installed system-wide).

		Installation			
Package	Req.	Sys.	User	Auto.	Note
C++ compiler	✓	✓			gcc, icc, etc
CMake > 2.8.7	✓	✓	✓		Ncurses GUI optional
BLAS	✓	✓	✓		Or MKL, ACML, OpenBLAS
LAPACK	✓	✓	✓		
Boost >= 1.52	✓	✓	✓	✓	Compile with iostreams
ModMETIS	✓			✓	
FFTW > 3.0		✓	✓	✓	For high-performance FFTs
ARPACK > 2.0		✓	✓		For arnoldi algorithms
OpenMPI		✓			For parallel execution
GSMPI				✓	For parallel execution
PETSc			✓	✓	Alternative linear solvers
Scotch		✓	✓	✓	Alternative mesh partitioning
VTK > 5.8		✓	✓		Visualisation utilities

Warning
Boost version 1.51 has a bug which prevents *Nektar++* working correctly. Please use a newer version.

1.3.2.2 Quick Start

Open a terminal.

If you have downloaded the tarball, first unpack it:

```
tar -zxvf nektar++-4.3.2.tar.gz
```

Change into the `nektar++` source code directory

```
mkdir -p build && cd build
ccmake ../
make install
```

1.3.2.3 Detailed instructions

From a terminal:

1. If you have downloaded the tarball, first unpack it

   ```
   tar -zxvf nektar++-4.3.2.tar.gz
   ```

2. Change into the source-code directory, create a `build` subdirectory and enter it

   ```
   mkdir -p build && cd build
   ```

3. Run the CMake GUI and configure the build by pressing `c`

   ```
   ccmake ../
   ```

   - Select the components of Nektar++ (prefixed with `NEKTAR_BUILD_`) you would like to build. Disabling solvers which you do not require will speed up the build process.
   - Select the optional libraries you would like to use (prefixed with `NEKTAR_USE_`) for additional functionality.
   - Select the libraries not already available on your system which you wish to be compiled automatically (prefixed with `THIRDPARTY_BUILD_`)

A full list of configuration options can be found in Section 1.3.5.

Note

Selecting `THIRDPARTY_BUILD_` options will request CMake to automatically download thirdparty libraries and compile them within the *Nektar++* directory. If you have administrative access to your machine, it is recommended to install the libraries system-wide through your package-management system.

4. Press `c` to configure the build. If errors arise relating to missing libraries, review the `THIRDPARTY_BUILD_` selections in the configuration step above or install the missing libraries manually or from system packages.

5. When configuration completes without errors, press `c` again until the option `g` to generate build files appears. Press `g` to generate the build files and exit CMake.

6. Compile the code

```
make install
```

During the build, missing third-party libraries will be automatically downloaded, configured and built in the *Nektar++* `build` directory.

Tip

If you have multiple processors/cores on your system, compilation can be significantly increased by adding the `-jX` option to make, where X is the number of simultaneous jobs to spawn. For example, use

```
make -j4 install
```

on a quad-core system.

7. Test the build by running unit and regression tests.

```
ctest
```

1.3.3 OS X

1.3.3.1 Prerequisites

Nektar++ uses a number of external programs and libraries for some or all of its functionality. Some of these are *required* and must be installed prior to compiling Nektar++, most of which are available on *MacPorts* (www.macports.org) or can be installed manually by a *user*. Others are optional and required only for specific features,

or can be downloaded and compiled for use with Nektar++ *automatically* (but not installed system-wide).

Note

To compile *Nektar++* on OS X, Apple's Xcode Developer Tools must be installed. They can be installed either from the App Store (only on Mac OS 10.7 and above) or downloaded directly from http://connect.apple.com/ (you are required to have an Apple Developer Connection account). Xcode includes Apple implementations of BLAS and LAPACK (called the Accelerate Framework).

		Installation			
Package	Req.	MacPorts	User	Auto.	Note
Xcode	✓				Provides developer tools
CMake > 2.8.7	✓	`cmake`	✓		Ncurses GUI optional
BLAS	✓				Part of Xcode
LAPACK	✓				Part of Xcode
Boost >= 1.52	✓	`boost`	✓	✓	Compile with iostreams
TinyXML	✓	`tinyxml`	✓	✓	
ModMETIS	✓			✓	
FFTW > 3.0		`fftw-3`	✓	✓	For high-performance FFTs
ARPACK > 2.0		`arpack`	✓		For arnoldi algorithms
OpenMPI		`openmpi`			For parallel execution
GSMPI				✓	For parallel execution
PETSc		`petsc`	✓	✓	Alternative linear solvers
Scotch		`scotch`	✓	✓	Alternative mesh partitioning
VTK > 5.8		`vtk`	✓		Visualisation utilities

Tip

CMake, and some other software, is available from MacPorts (`http://macports.org`) and can be installed using, for example,

```
sudo port install cmake
```

Package names are given in the table above. Similar packages also exist in other package managers such as Homebrew.

1.3.3.2 Quick Start

Open a terminal (Applications->Utilities->Terminal).

If you have downloaded the tarball, first unpack it:

```
tar -zxvf nektar++-4.3.2.tar.gz
```

Change into the `nektar++` source code directory

```
mkdir -p build && cd build
ccmake ../
make install
```

1.3.3.3 Detailed instructions

From a terminal (Applications->Utilities->Terminal):

1. If you have downloaded the tarball, first unpack it

   ```
   tar -zxvf nektar++-4.3.2.tar.gz
   ```

2. Change into the source-code directory, create a `build` subdirectory and enter it

   ```
   mkdir -p build && cd build
   ```

3. Run the CMake GUI and configure the build

   ```
   ccmake ../
   ```

   Use the arrow keys to navigate the options and `ENTER` to select/edit an option.

   - Select the components of Nektar++ (prefixed with `NEKTAR_BUILD_`) you would like to build. Disabling solvers which you do not require will speed up the build process.
   - Select the optional libraries you would like to use (prefixed with `NEKTAR_USE_`) for additional functionality.
   - Select the libraries not already available on your system which you wish to be compiled automatically (prefixed with `THIRDPARTY_BUILD_`)
   -

   A full list of configuration options can be found in Section 1.3.5.

Note

Selecting `THIRDPARTY_BUILD_` options will request CMake to automatically download thirdparty libraries and compile them within the *Nektar++* directory. If you have administrative access to your machine, it is recommended to install the libraries system-wide through MacPorts.

4. Press `c` to configure the build. If errors arise relating to missing libraries (variables set to `NOTFOUND`), review the `THIRDPARTY_BUILD_` selections in the previous step or install the missing libraries manually or through MacPorts.

5. When configuration completes without errors, press `c` again until the option `g` to generate build files appears. Press `g` to generate the build files and exit CMake.

6. Compile the code

```
make install
```

   During the build, missing third-party libraries will be automatically downloaded, configured and built in the *Nektar++* `build` directory.

Tip

If you have multiple processors/cores on your system, compilation can be significantly increased by adding the `-jX` option to make, where X is the number of simultaneous jobs to spawn. For example,

```
make -j4 install
```

7. Test the build by running unit and regression tests.

```
ctest
```

1.3.4 Windows

Windows compilation is supported, but the build process is somewhat convoluted at present. As such, only serial execution is supported with a minimal amount of additional build packages. These can either be installed by the user, or automatically in the build process.

		Installation		
Package	Req.	User	Auto.	Note
MS Visual Studio	✓	✓		2012 and 2013 known working
CMake $\geq$ 3.0	✓	✓		
BLAS	✓	✓	✓	
LAPACK	✓	✓	✓	
Boost $\geq$ 1.55	✓	✓	✓	Compile with iostreams
ModMETIS	✓	✓	✓	

1.3.4.1 Detailed instructions

1. Install Microsoft Visual Studio 2013 (preferred) or 2012 (known to work). This can be obtained from Microsoft free of charge by using their Express developer tools from http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx.

2. Install WinRAR from http://www.rarlab.com/download.htm.

3. Install CMake 3.0+ from http://www.cmake.org/download/. When prompted, select the option to add CMake to the system PATH.

4. (Optional) Install Git from http://git-scm.com/download/win to use the development versions of *Nektar++*. When prompted, select the option to add Git to the system PATH. You do not need to select the option to add Unix tools to the PATH.

5. (Optional) If you do not wish to build boost during the compilation process (which can take some time), then boost binaries can be found at http://sourceforge.net/projects/boost/files/boost-binaries/1.57.0/. By default these install into `C:\local\boost_1_57_0`. If you use these libraries, you will need to:

   - Rename `libs-msvc12.0` to `lib`
   - Inside the `lib` directory, create duplicates of `boost_zlib.dll` and `boost_bzip2.dll` called `zlib.dll` and `libbz2.dll`.

6. Unpack `nektar++-4.3.2.tar.gz` using WinRAR.

Note

Some Windows versions do not recognise the path of a folder which has `++` in the name. If you think that your Windows version can not handle path containing special characters, you should rename `nektar++-4.3.2` to `nektar-4.3.2`.

7. Create a `builds` directory within the `nektar++-4.3.2` subdirectory.

8. Open a Visual Studio terminal. From the start menu, this can be found in *All Programs > Visual Studio 2013 > Visual Studio Tools > Developer Command Prompt for VS2013*.

9. Change directory into the `builds` directory and run the CMake graphical utility:

```
cd C:\path\to\nektar\builds
cmake-gui ..
```

10. Select the build system you want to generate build scripts for. Note that *Visual Studio 2013* is listed as *Visual Studio 12* in the drop-down list. If you have a 64-bit installation of Windows 7, you should select the *Win64* variant, otherwise 32-bit executables will be generated. Select the option to use the native compilers.

11. Click the *Configure* button, then the *Generate* button.

12. Return to the command line and issue the command:

    ```
    msbuild INSTALL.vcxproj /p:Configuration=Release
    ```

    To build in parallel with, for example, 12 processors, issue:

    ```
    msbuild INSTALL.vcxproj /p:Configuration=Release /m:12
    ```

13. After the installation process is completed, the executables will be available in `builds\dist\bin`.

14. To use these executables, you need to modify your system `PATH` to include the library directories where DLLs are stored. To do this, navigate to *Control Panel > System and Security > System*, select *Advanced System Settings*, and in the *Advanced* tab click the *Environment Variables*. In the *System Variables* box, select *Path* and click *Edit*. To the end of this list, add the **full paths** to directories:

    - `builds\dist\lib\nektar++-4.3.2`
    - `builds\dist\bin`
    - Optionally, if you installed Boost from the binary packages, `C:\local\boost_1_57_0 \lib`

15. To run the test suite, open a **new** command line window, change to the `builds` directory, and then issue the command

    ```
    ctest -C Release
    ```

1.3.5 CMake Option Reference

This section describes the main configuration options which can be set when building Nektar++. The default options should work on almost all systems, but additional features (such as parallelisation and specialist libraries) can be enabled if needed.

1.3.5.1 Components

The first set of options specify the components of the Nektar++ toolkit to compile. Some options are dependent on others being enabled, so the available options may change.

Components of the *Nektar++* package can be selected using the following options:

- `NEKTAR_BUILD_DEMOS` (Recommended)

  Compiles the demonstration programs. These are primarily used by the regression testing suite to verify the *Nektar++* library, but also provide an example of the basic usage of the framework.

- `NEKTAR_BUILD_DOC`

  Compiles the Doxygen documentation for the code. This will be put in

  ```
  $BUILDDIR/doxygen/html
  ```

- `NEKTAR_BUILD_LIBRARY` (Required)

  Compiles the Nektar++ framework libraries. This is required for all other options.

- `NEKTAR_BUILD_SOLVERS` (Recommended)

  Compiles the solvers distributed with the *Nektar++* framework.

  If enabling `NEKTAR_BUILD_SOLVERS`, individual solvers can be enabled or disabled. See Part III for the list of available solvers. You can disable solvers which are not required to reduce compilation time. See the `NEKTAR_SOLVER_X` option.

- `NEKTAR_BUILD_TESTS` (Recommended)

  Compiles the testing program used to verify the *Nektar++* framework.

- `NEKTAR_BUILD_TIMINGS`

  Compiles programs used for timing *Nektar++* operations.

- `NEKTAR_BUILD_UNIT_TESTS`

  Compiles tests for checking the core library functions.

- `NEKTAR_BUILD_UTILITIES`

  Compiles utilities for pre- and post-processing simulation data.

- `NEKTAR_SOLVER_X`

  Enabled compilation of the 'X' solver.

A number of ThirdParty libraries are required by *Nektar++*. There are also optional libraries which provide additional functionality. These can be selected using the following options:

- `NEKTAR_USE_ACML`

  Use the optimised BLAS library for AMD processors.

- `NEKTAR_USE_ACCELERATE_FRAMEWORK`

  Use the Mac Osx accelerate framework for BLAS and LAPACK methods. This option should only be required under in a Mac OSX environment.

- `NEKTAR_USE_ARPACK`

  Build *Nektar++* with support for ARPACK. This provides routines used for linear stability analyses. Alternative Arnoldi algorithms are also implemented directly in *Nektar++*.

- `NEKTAR_USE_BLAS_LAPACK` (Required)

  Enables the use of Basic Linear Algebra Subroutines libraries for linear algebra operations.

- `NEKTAR_USE_SYSTEM_BLAS_LAPACK` (Recommended)

  On Linux systems, use the default BLAS and LAPACK library on the system. This may not be the implementation offering the highest performance for your architecture, but it is the most likely to work without problem.

- `NEKTAR_USE_CCM`

  Use the ccmio library provided with the Star-CCM package for reading ccm files. This option is required as part of NekMesh if you wish to convert a Star-CCM mesh into the Nektar format. It is possible to read a Tecplot plt file from Star-CCM but this output currently needs to be converted to ascii format using the Tecplot package.

- `NEKTAR_USE_FFTW`

  Build *Nektar++* with support for FFTW for performing Fast Fourier Transforms (FFTs). This is used only when using domains with homogeneous coordinate directions.

- `NEKTAR_USE_MKL`

  Use the Intel MKL library. This is typically available on cluster environments and should offer performance tuned for the specific cluster environment.

- `NEKTAR_USE_MPI` (Recommended)

  Build Nektar++ with MPI parallelisation. This allows solvers to be run in serial or parallel.

- `NEKTAR_USE_OPENBLAS`

  Use OpenBLAS for the BLAS library. OpenBLAS is based on the Goto BLAS implementation and generally offers better performance than a non-optimised system BLAS. However, the library must be installed on the system.

- `NEKTAR_USE_PETSC`

  Build *Nektar++* with support for the PETSc package for solving linear systems.

- `NEKTAR_USE_SCOTCH`

  Build *Nektar++* with support for the SCOTCH graph partitioning library. This provides an alternative mesh partitioning algorithm to METIS. However, METIS is still required as it is used by the multi-level static condensation algorithm.

- `NEKTAR_USE_SMV`

  Build *Nektar++* with support for optimised sparse matrix-vector operations.

- `NEKTAR_USE_VTK`

  Build *Nektar++* with support for VTK libraries. This is only needed for specialist utilities and is not needed for general use.

  Note

  The VTK libraries are not needed for converting the output of simulations to VTK format for visualization as this is handled internally.

The `THIRDPARTY_BUILD_X` options select which third-party libraries are automatically built during the *Nektar++* build process. Below are the choices of X:

- `BOOST`

  The *Boost* libraries are frequently provided by the operating system, so automatic compilation is not enabled by default. If you do not have Boost on your system, you can enable this to have Boost configured automatically.

- `GSMPI`

  (MPI-only) Parallel communication library.

- `LOKI`

  An implementation of a singleton.

- `METIS`

  A graph partitioning library used for substructuring of matrices and mesh partitioning when *Nektar++* is run in parallel.

- `PETSC`

  A package for the parallel solution of linear algebra systems.

- `SCOTCH`

  An alternative graph partitioning library used for mesh partitioning when *Nektar++* is run in parallel.

- `TINYXML`

  Library for reading and writing XML files.

1.4 Compiling Documentation

Documentation for Nektar++ is provided in a number of forms:

- User Guide (LaTeX, compiled to pdf or html)
- Source code documentation (Doxygen compiled to html)

1.4.1 Dependencies

To build the LaTeX documents (user guide or tutorials), the following dependencies are required:

- texlive-base
- texlive-latex-extra
- texlive-science
- imagemagick

To build the Doxygen documentation, the following dependencies are required:

- doxygen
- graphviz

1.4.2 Compiling the User Guide

To compile the User Guide:

1. Configure the Nektar++ build tree as normal.
2. Run

   ```
   make user-guide-pdf
   ```

   to make the PDF version, or run

   ```
   make user-guide-html
   ```

   to make the HTML version.

1.4.3 Compiling the code documentation

To compile the code documentation enable the `NEKTAR_BUILD_DOC` option in the `ccmake` configuration tool.

You can then compile the HTML code documentation using:

```
make doc
```

1.5 Compiling Tutorials

If you are using a clone of the *Nektar++* git repository, you can also download the source for the *Nektar++* tutorials which is available as a *git submodule*.

1. From a *Nektar++* working directory (e.g. `$NEKPP`):

   ```
   git submodule init
   git submodule update --remote
   ```

2. From your build directory (e.g. `$NEKPP/build`), re-run `cmake` to update the build system to include the tutorials

   ```
   cmake ../
   ```

3. Compile each required tutorial, for example

   ```
   make flow-stability-channel
   ```

Chapter 2

Mathematical Formulation

2.1 Background

The spectral/hp element method combines the geometric flexibility of classical h-type finite element techniques with the desirable resolution properties of spectral methods. In this approach a polynomial expansion of order P is applied to every elemental domain of a coarse finite element type mesh. These techniques have been applied in many fundamental studies of fluid mechanics [31] and more recently have gained greater popularity in the modelling of wave-based phenomena such as computational electromagnetics [14] and shallow water problems [4] - particularly when applied within a Discontinuous Galerkin formulation.

There are at least two major challenges which arise in developing an efficient implementation of a spectral/hp element discretisation:

- implementing the mathematical structure of the technique in a digestible, generic and coherent manner, and
- designing and implementing the numerical methods and data structures in a matter so that both high- and low-order discretisations can be efficiently applied.

In order to design algorithms which are efficient for both low- and high-order spectral/hp discretisations, it is important clearly define what we mean with low- and high-order. The spectral/hp element method can be considered as bridging the gap between the high-order end of the traditional finite element method and low-order end of conventional spectral methods. However, the concept of high- and low-order discretisations can mean very different things to these different communities. For example, the seminal works by Zienkiewicz & Taylor [36] and Hughes list examples of elemental expansions only up to third or possibly fourth-order, implying that these orders are considered to be high-order for the traditional h-type finite element community. In contrast the text books of the spectral/hp element community typically show examples of problems ranging from a

low-order bound of minimally fourth-order up to anything ranging from 10^{th}-order to 15^{th}-order polynomial expansions. On the other end of the spectrum, practitioners of global (Fourier-based) spectral methods [12] would probably consider a 16^{th}-order global expansion to be relatively low-order approximation.

One could wonder whether these different definitions of low- and high-order are just inherent to the tradition and lore of each of the communities or whether there are more practical reasons for this distinct interpretation. Proponents of lower-order methods might highlight that some problems of practical interest are so geometrically complex that one cannot computationally afford to use high-order techniques on the massive meshes required to capture the geometry. Alternatively, proponents of high-order methods highlight that if the problem of interest can be captured on a computational domain at reasonable cost then using high-order approximations for sufficiently *smooth* solutions will provide a higher accuracy for a given computational cost. If one however probes even further it also becomes evident that the different communities choose to implement their algorithms in different manners. For example the standard h-type finite element community will typically uses techniques such as sparse matrix storage formats (where only the non-zero entries of a global matrix are stored) to represent a global operator. In contrast the spectral/hp element community acknowledges that for higher polynomial expansions more closely coupled computational work takes place at the individual elemental level and this leads to the use of elemental operators rather than global matrix operators. In addition the global spectral method community often make use of the tensor-product approximations where products of one-dimensional rules for integration and differentiation can be applied.

2.2 Methods overview

Here a review of some terminology in order to situate the spectral/hp element method within the field of the finite element methods.

2.2.1 The finite element method (FEM)

Nowadays, the finite element method is one of the most popular numerical methods in the field of both solid and fluid mechanics. It is a discretisation technique used to solve (a set of) partial differential equations in its equivalent variational form. The classical approach of the finite element method is to partition the computational domain into a mesh of many small subdomains and to approximate the unknown solution by piecewise linear interpolation functions, each with local support. The FEM has been widely discussed in literature and for a complete review of the method, the reader is also directed to the seminal work of Zienkiewicz and Taylor [36].

2.2.2 High-order finite element methods

While in the classical finite element method the solution is expanded in a series of linear basis functions, high-order FEMs employ higher-order polynomials to approximate the

solution. For the high-order FEM, the solution is locally expanded into a set of $P+1$ linearly independent polynomials which span the polynomial space of order P. Confusion may arise about the use of the term *order*. While the order, or *degree*, of the expansion basis corresponds to the maximal polynomial degree of the basis functions, the order of the method essentially refers to the accuracy of the approximation. More specifically, it depends on the convergence rate of the approximation with respect to mesh-refinement. It has been shown by Babuska and Suri [3], that for a sufficiently smooth exact solution $u \in H^k(\Omega)$, the error of the FEM approximation u^δ can be bounded by:

$$||u - u^\delta||_E \leq Ch^P||u||_k.$$

This implies that when decreasing the mesh-size h, the error of the approximation algebraically scales with the P^{th} power of h. This can be formulated as:

$$||u - u^\delta||_E = O(h^P).$$

If this holds, one generally classifies the method as a P^{th}-order FEM. However, for non-smooth problems, i.e. $k < P+1$, the order of the approximation will in general be lower than P, the order of the expansion.

2.2.2.1 h-version FEM

A finite element method is said to be of h-type when the degree P of the piecewise polynomial basis functions is fixed and when any change of discretisation to enhance accuracy is done by means of a mesh refinement, that is, a reduction in h. Dependent on the problem, local refinement rather than global refinement may be desired. The h-version of the classical FEM employing linear basis functions can be classified as a first-order method when resolving smooth solutions.

2.2.2.2 p-version FEM

In contrast with the h-version FEM, finite element methods are said to be of p-type when the partitioning of domain is kept fixed and any change of discretisation is introduced through a modification in polynomial degree P. Again here, the polynomial degree may vary per element, particularly when the complexity of the problem requires local enrichment. However, sometimes the term p-type FEM is merely used to indicated that a polynomial degree of $P > 1$ is used.

2.2.2.3 hp-version FEM

In the hp-version of the FEM, both the ideas of mesh refinement and degree enhancement are combined.

2.2.2.4 The spectral method

As opposed to the finite element methods which builds a solution from a sequence of local elemental approximations, spectral methods approximate the solution by a truncated series of global basis functions. Modern spectral methods, first presented by Gottlieb and Orzag [12], involve the expansion of the solution into high-order orthogonal expansion, typically by employing Fourier, Chebyshev or Legendre series.

2.2.2.5 The spectral element method

Patera [25] combined the high accuracy of the spectral methods with the geometric flexibility of the finite element method to form the spectral element method. The multi-elemental nature makes the spectral element method conceptually similar to the above mentioned high-order finite element. However, historically the term spectral element method has been used to refer to the high-order finite element method using a specific nodal expansion basis. The class of nodal higher-order finite elements which have become known as spectral elements, use the Lagrange polynomials through the zeros of the Gauss-Lobatto(-Legendre) polynomials.

2.2.2.6 The spectral/hp element method

The spectral/hp element method, as its name suggests, incorporates both the multi-domain spectral methods as well as the more general high-order finite element methods. One can say that it encompasses all methods mentioned above. However, note that the term spectral/hp element method is mainly used in the field of fluid dynamics, while the terminology p and hp-FEM originates from the area of structural mechanics.

2.2.3 The Galerkin formulation

Finite element methods typically use the Galerkin formulation to derive the weak form of the partial differential equation to be solved. We will primarily adopt the classical Galerkin formulation in combination with globally C^0 continuous spectral/hp element discretisations.

To describe the Galerkin method, consider a steady linear differential equation in a domain Ω denoted by

$$L(u) = f,$$

subject to appropriate boundary conditions. In the Galerkin method, the weak form of this equation can be derived by pre-multiplying this equation with a test function v and integrating the result over the entire domain Ω to arrive at: Find $u \in \mathcal{U}$ such that

$$\int_\Omega vL(u)d\boldsymbol{x} = \int_\Omega vfd\boldsymbol{x}, \quad \forall v \in \mathcal{V},$$

where $\mathcal{U}$ and $\mathcal{V}$ respectively are a suitably chosen trial and test space (in the traditional Galerkin method, one typically takes $\mathcal{U} = \mathcal{V}$). In case the inner product of v and $\mathbb{L}(u)$

can be rewritten into a bi-linear form $a(v,u)$, this problem is often formulated more concisely as: Find $u \in \mathcal{U}$ such that

$$a(v,u) = (v,f), \quad \forall v \in \mathcal{V},$$

where (v,f) denotes the inner product of v and f. The next step in the classical Galerkin finite element method is the discretisation: rather than looking for the solution u in the infinite dimensional function space $\mathcal{U}$, one is going to look for an approximate solution u^δ in the reduced finite dimensional function space $\mathcal{U}^\delta \subset \mathcal{U}$. Therefore we represent the approximate solution as a linear combination of basis functions Φ_n that span the space $\mathcal{U}^\delta$, i.e.

$$u^\delta = \sum_{n \in \mathcal{N}} \Phi_n \hat{u}_n.$$

Adopting a similar discretisation for the test functions v, the discrete problem to be solved is given as: Find $\hat{u}_n$ $(n \in \mathcal{N})$ such that

$$\sum_{n \in \mathcal{N}} a(\Phi_m, \Phi_n)\hat{u}_n = (\Phi_m, f), \quad \forall m \in \mathcal{N}.$$

It is customary to describe this set of equations in matrix form as

$$\boldsymbol{A}\hat{\boldsymbol{u}} = \hat{\boldsymbol{f}},$$

where $\hat{\boldsymbol{u}}$ is the vector of coefficients $\hat{u}_n$, $\boldsymbol{A}$ is the system matrix with elements

$$\boldsymbol{A}[m][n] = a(\Phi_m, \Phi_n) = \int_\Omega \Phi_m L(\Phi_n) d\boldsymbol{x},$$

and the vector $\hat{\boldsymbol{f}}$ is given by

$$\hat{\boldsymbol{f}}[m] = (\Phi_m, f) = \int_\Omega \Phi_m f d\boldsymbol{x}.$$

Chapter 3

XML Session File

The Nektar++ native file format is compliant with XML version 1.0. The root element is NEKTAR which contains a number of other elements which describe configuration for different aspects of the simulation. The required elements are shown below:

```
<NEKTAR>
  <GEOMETRY>
    ...
  </GEOMETRY>
  <EXPANSIONS>
    ...
  </EXPANSIONS>
  <CONDITIONS>
    ...
  </CONDITIONS>
  ...
</NEKTAR>
```

The different sub-elements can be split across multiple files, however each file must have a top-level NEKTAR tag. For example, one might store the geometry information separate from the remaining configuration in two separate files as illustrated below:

`geometry.xml`

```
<NEKTAR>
  <GEOMETRY>
    ...
  </GEOMETRY>
</NEKTAR>
```

`conditions.xml`

```
<NEKTAR>
  <CONDITIONS>
    ...
  </CONDITIONS>
  <EXPANSIONS>
```

```
    ...
  </EXPANSIONS>
  ...
</NEKTAR>
```

Note

When specifying multiple files, repeated XML sub-elements are not merged. The sub-elements from files appearing later in the list will, in general, override those elements from earlier files.

For example, the `NekMesh` utility will produce a default `EXPANSIONS` element and blank `CONDITIONS` element. Specifying a custom-written XML file containing these sections *after* the file produced by `NekMesh` will override these defaults.
The exception to this rule is when an empty XML sub-element would override a non-empty XML sub-element. In this case the empty XML sub-element will be ignored. If the custom-written XML file containing `CONDITIONS` were specified before the file produced by `NEKMESH`, the empty `CONDITIONS` tag in the latter file would be ignored.

3.1 Geometry

This section defines the mesh. It specifies a list of vertices, edges (in two or three dimensions) and faces (in three dimensions) and how they connect to create the elemental decomposition of the domain. It also defines a list of composites which are used in the Expansions and Conditions sections of the file to describe the polynomial expansions and impose boundary conditions.

The GEOMETRY section is structured as

```
<GEOMETRY DIM="2" SPACE="2">
  <VERTEX> ... </VERTEX>
  <EDGE> ... </EDGE>
  <FACE> ... </FACE>
  <ELEMENT> ... </ELEMENT>
  <CURVED> ... </CURVED>
  <COMPOSITE> ... </COMPOSITE>
  <DOMAIN> ... </DOMAIN>
</GEOMETRY>
```

It has two (required) attributes:

- `DIM` specifies the dimension of the expansion elements.
- `SPACE` specifies the dimension of the space in which the elements exist.

These attributes allow, for example, a two-dimensional surface to be embedded in a three-dimensional space.

Note

The attribute `PARTITION` may also appear in a partitioned mesh. However, this attribute should not be explicitly specified by the user.

Each of the `VERTEX`, `EDGE`, `FACE`, `ELEMENT` and `CURVED` sections may optionally be compressed and stored in base64-encoded gzipped binary form, using either little-endian or big-endian ordering, as specified by the `COMPRESSED` attribute to these sections. Currently supported values are:

- `B64Z-LittleEndian`: Base64 Gzip compressed using little-endian ordering.
- `B64Z-BigEndian`: Base64 Gzip compressed using big-endian ordering.

Note

The description below explains how the `GEOMETRY` section is laid out in uncompressed ascii format. From Jan 2016 the distribution uses the compressed format for each of the above sections. To convert a compressed xml file into ascii format use

```
NekMesh file.msh newfile.xml:xml:uncompress
```

3.1.1 Vertices

Vertices have three coordinates. Each has a unique vertex ID. In uncompressed form, they are defined within `VERTEX` subsection as follows:

```
<V ID="0"> 0.0  0.0  0.0 </V> ...
```

The `VERTEX` subsection has optional attributes which can be used to apply a transformation to the mesh:
`XSCALE`, `YSCALE`, `ZSCALE`, `XMOVE`, `YMOVE`, `ZMOVE`

They specify scaling factors (centred at the origin) and translations to the vertex coordinates. For example, the following snippet

```
<VERTEX XSCALE="5">
  <V ID="0"> 0.0  0.0  0.0 </V>
  <V ID="1"> 1.0  2.0  0.0 </V>
</VERTEX>
```

defines two vertices with coordinates $(0.0, 0.0, 0.0), (1.0, 2.0, 0.0)$.

All of these attributes can be arbitrary analytic expressions depending on pre- defined constants and parameters defined in the XML file and mathematical operations/functions of the latter. If omitted, default scaling factors 1.0, and translations of 0.0, are assumed.

3.1.2 Edges

Tip

The `EDGES` section is only necessary when `DIM=2` or `DIM=3` in the parent `GEOMETRY` element and may be omitted for one-dimensional meshes.

Edges are defined by two vertices. Each edge has a unique edge ID. In uncompressed form, they are defined in the file with a line of the form

```
<E ID="0"> 0 1 </E>
```

3.1.3 Faces

Tip

The `FACES` section is only necessary when `DIM=3` in the parent `GEOMETRY` element and may otherwise be omitted.

Faces are defined by three or more edges. Each face has a unique face ID. They are defined in the file with a line of the form

```
<T ID="0"> 0 1 2 </T>
<Q ID="1"> 3 4 5 6 </Q>
```

The choice of tag specified (T or Q), and thus the number of edges specified depends on the geometry of the face (triangle or quadrilateral).

3.1.4 Element

Elements define the top-level geometric entities in the mesh. Their definition depends upon the dimension of the expansion. For two-dimensional expansions, an element is defined by a sequence of three or four edges. For three-dimensional expansions, the element is defined by a list of faces. Elements are defined in the file with a line of the form

```
<T ID="0"> 0 1 2 </T>
<H ID="1"> 3 4 5 6 7 8 </H>
```

Again, the choice of tag specified depends upon the geometry of the element. The element tags are:

- `S` Segment
- `T` Triangle
- `Q` Quadrilateral
- `A` Tetrahedron
- `P` Pyramid
- `R` Prism
- `H` Hexahedron

3.1.5 Curved Edges and Faces

Tip

The `CURVED` section is only necessary if curved edges or faces are present in the mesh and may otherwise be omitted.

For mesh elements with curved edges and/or curved faces, a separate entry is used to describe the control points for the curve. Each curve has a unique curve ID and is associated with a predefined edge or face. The total number of points in the curve (including end points) and their distribution is also included as attributes. The control points are listed in order, each specified by three coordinates. Curved edges are defined in the file with a line of the form

```
<E ID="3" EDGEID="7" TYPE="PolyEvenlySpaced" NUMPOINTS="3">
   0.0  0.0  0.0    0.5  0.5  0.0    1.0  0.0  0.0
</E>
```

Note

In the compressed form, this section contains different sub-elements to efficiently encode the high-order curvature data. This is not described further in this document.

3.1.6 Composites

Composites define collections of elements, faces or edges. Each has a unique composite ID associated with it. All components of a composite entry must be of the same type. The syntax allows components to be listed individually or using ranges. Examples include

```
<C ID="0"> T[0-862] </C>
<C ID="1"> E[68,69,70,71] </C>
```

3.1.7 Domain

This tag specifies composites which describe the entire problem domain. It has the form of

```
<DOMAIN> C[0] </DOMAIN>
```

3.2 Expansions

This section defines the polynomial expansions used on each of the defined geometric composites. Expansion entries specify the number of modes, the basis type. The short-hand version has the following form

```
<E COMPOSITE="C[0]" NUMMODES="5" FIELDS="u" TYPE="MODIFIED" />
```

or, if we have more then one variable we can apply the same basis to all using

```
<E COMPOSITE="C[0]" NUMMODES="5" FIELDS="u,v,p" TYPE="MODIFIED" />
```

The expansion basis can also be specified in detail as a combination of one-dimensional bases, and thus the user is able to, for example, increase the quadrature order. For tet elements this takes the form:

```
<E COMPOSITE="C[0]"
   BASISTYPE="Modified_A,Modified_B,Modified_C"
   NUMMODES="3,3,3"
   POINTSTYPE="GaussLobattoLegendre,GaussRadauMAlpha1Beta0,GaussRadauMAlpha2Beta0"
   NUMPOINTS="4,3,3"
   FIELDS="u" />
```

and for prism elements:

```
<E COMPOSITE="C[1]"
   BASISTYPE="Modified_A,Modified_A,Modified_B"
   NUMMODES="3,3,3"
   POINTSTYPE="GaussLobattoLegendre,GaussLobattoLegendre,GaussRadauMAlpha1Beta0"
   NUMPOINTS="4,4,3"
   FIELDS="u" />
```

3.3 Conditions

The final section of the file defines parameters and boundary conditions which define the nature of the problem to be solved. These are enclosed in the `CONDITIONS` tag.

3.3.1 Parameters

Parameters may be required by a particular solver (for instance time-integration parameters or solver-specific parameters), or arbitrary and only used within the context of the session file (e.g. parameters in the definition of an initial condition). All parameters are enclosed in the `PARAMETERS` XML element.

```
<PARAMETERS>
    ...
</PARAMETERS>
```

A parameter may be of integer or real type and may reference other parameters defined previous to it. It is expressed in the file as

```
<P> [PARAMETER NAME] = [PARAMETER VALUE] </P>
```

For example,

```
<P> NumSteps = 1000              </P>
<P> TimeStep = 0.01              </P>
<P> FinTime  = NumSteps*TimeStep </P>
```

3.3.2 Solver Information

These specify properties to define the actions specific to solvers, typically including the equation to solve, the projection type and the method of time integration. The property/value pairs are specified as XML attributes. For example,

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvection"    />
  <I PROPERTY="Projection"            VALUE="Continuous"           />
  <I PROPERTY="TimeIntegrationMethod" VALUE="ClassicalRungeKutta4" />
</SOLVERINFO>
```

The list of available solvers in Nektar++ can be found in Part III.

3.3.2.1 Drivers

Drivers are defined under the `CONDITIONS` section as properties of the `SOLVERINFO` XML element. The role of a driver is to manage the solver execution from an upper level.

The default driver is called `Standard` and executes the following steps:

1. Prints out on screen a summary of all the conditions defined in the input file.
2. Sets up the initial and boundary conditions.
3. Calls the solver defined by `SolverType` in the `SOLVERINFO` XML element.
4. Writes the results in the output (.fld) file.

In the following example, the driver `Standard` is used to manage the execution of the incompressible Navier-Stokes equations:

```
<SOLVERINFO>
    <I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes" />
    <I PROPERTY="SolverType" VALUE="VelocityCorrectionScheme" />
    <I PROPERTY="Projection" VALUE="Galerkin" />
    <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2" />
    <I PROPERTY="Driver" VALUE="Standard" />
</SOLVERINFO>
```

If no driver is specified in the session file, the driver `Standard` is called by default. Other drivers can be used and are typically focused on specific applications. As described in Sec. 10.3.1 and 10.4.1, the other possibilities are:

- `ModifiedArnoldi` - computes of the leading eigenvalues and eigenmodes using modified Arnoldi method.
- `Arpack` - computes of eigenvalues/eigenmodes using Implicitly Restarted Arnoldi Method (ARPACK).
- `SteadyState` - uses the Selective Frequency Damping method (see Sec. 10.1.5) to obtain a steady-state solution of the Navier-Stokes equations (compressible or incompressible).

3.3.3 Variables

These define the number (and name) of solution variables. Each variable is prescribed a unique ID. For example a two-dimensional flow simulation may define the velocity variables using

```
<VARIABLES>
  <V ID="0"> u </V>
  <V ID="1"> v </V>
</VARIABLES>
```

3.3.4 Global System Solution Information

Many *Nektar++* solvers use an implicit formulation of their equations to, for instance, improve timestep restrictions. This means that a large matrix system must be constructed and a global system set up to solve for the unknown coefficients. There are several approaches in the spectral/*hp* element method that can be used in order to improve efficiency in these methods, as well as considerations as to whether the simulation is run in parallel or serial. *Nektar++* opts for 'sensible' default choices, but these may or may not be optimal depending on the problem under consideration.

This section of the XML file therefore allows the user to specify the global system solution parameters, which dictates the type of solver to be used for any implicit systems that are constructed. This section is particularly useful when using a multi-variable solver such as the incompressible Navier-Stokes solver, as it allows us to select different preconditioning

and residual convergence options for each variable. As an example, consider the block defined by:

```
<GLOBALSYSSOLNINFO>
  <V VAR="u,v,w">
    <I PROPERTY="GlobalSysSoln"             VALUE="IterativeStaticCond" />
    <I PROPERTY="Preconditioner"            VALUE="LowEnergyBlock"/>
    <I PROPERTY="IterativeSolverTolerance"  VALUE="1e-8"/>
  </V>
  <V VAR="p">
    <I PROPERTY="GlobalSysSoln"             VALUE="IterativeStaticCond" />
    <I PROPERTY="Preconditioner"     VALUE="FullLinearSpaceWithLowEnergyBlock"/>
    <I PROPERTY="IterativeSolverTolerance"  VALUE="1e-6"/>
  </V>
</GLOBALSYSSOLNINFO>
```

The above section specifies that the variables `u,v,w` should use the `IterativeStaticCond` global solver alongside the `LowEnergyBlock` preconditioner and an iterative tolerance of 10^{-8} on the residuals. However the pressure variable `p` is generally stiffer: we therefore opt for a more expensive `FullLinearSpaceWithLowEnergyBlock` preconditioner and a larger residual of 10^{-6}. We now outline the choices that one can use for each of these parameters and give a brief description of what they mean.

Note

Defaults for all fields can be defined by setting the equivalent property in the `SOLVERINFO` section. Parameters defined in this section will override any options specificed there.

3.3.4.1 `GlobalSysSoln` options

Nektar++ presently implements four methods of solving a global system:

- **Direct** solvers construct the full global matrix and directly invert it using an appropriate matrix technique, such as Cholesky factorisation, depending on the properties of the matrix. Direct solvers **only** run in serial.
- **Iterative** solvers instead apply matrix-vector multiplications repeatedly, using the conjugate gradient method, to converge to a solution to the system. For smaller problems, this is typically slower than a direct solve. However, for larger problems it can be used to solve the system in parallel execution.
- **Xxt** solvers use the XX^T library to perform a parallel direct solve. This option is only available if the `NEKTAR_USE_MPI` option is enabled in the CMake configuration.
- **PETSc** solvers use the PETSc library, giving access to a wide range of solvers and preconditioners. See section 3.3.4.4 below for some additional information on how

to use the PETSc solvers. This option is only available if the `NEKTAR_USE_PETSC` option is enabled in the CMake configuration.

Both the **Xxt** and **PETSc** solvers are considered advanced and are under development – either the direct or iterative solvers are recommended in most scenarios.

These solvers can be run in one of three approaches:

- The **Full** approach constructs the global system based on all of the degrees of freedom contained within an element. For most of the *Nektar++* solvers, this technique is not recommended.

- The **StaticCond** approach applies a technique called *static condensation* to instead construct the system using only the degrees of freedom on the boundaries of the elements, which reduces the system size considerably. This is the **default option in parallel**.

- **MultiLevelStaticCond** methods apply the static condensation technique repeatedly to further reduce the system size, which can improve performance by 25-30% over the normal static condensation method. It is therefore the **default option in serial**. Note that whilst parallel execution technically works, this is under development and is likely to be slower than single-level static condensation: this is therefore not recommended.

The `GlobalSysSoln` option is formed by combining the method of solution with the approach: for example `IterativeStaticCond` or `PETScMultiLevelStaticCond`.

3.3.4.2 Preconditioner options

Preconditioners can be used in the iterative and PETSc solvers to reduce the number of iterations needed to converge to the solution. There are a number of preconditioner choices, the default being a simple Jacobi (or diagonal) preconditioner, which is enabled by default. There are a number of choices that can be enabled through this parameter, which are all generally discretisation and dimension-dependent:

Name	Dimensions	Discretisations
`Null`	All	All
`Diagonal`	All	All
`FullLinearSpace`	2/3D	CG
`LowEnergyBlock`	3D	CG
`Block`	2/3D	All
`FullLinearSpaceWithDiagonal`	All	CG
`FullLinearSpaceWithLowEnergyBlock`	2/3D	CG
`FullLinearSpaceWithBlock`	2/3D	CG

For a detailed discussion of the mathematical formulation of these options, see the developer guide.

3.3.4.3 SuccessiveRHS options

The `SuccessiveRHS` option can be used in the iterative solver only, to attempt to reduce the number of iterations taken to converge to a solution. It stores a number of previous solutions, dictated by the setting of the `SuccessiveRHS` option, to give a better initial guess for the iterative process.

3.3.4.4 PETSc options and configuration

The PETSc solvers, although currently experimental, are operational both in serial and parallel. PETSc gives access to a wide range of alternative solver options such as GMRES, as well as any packages that PETSc can link against, such as the direct multi-frontal solver MUMPS.

Configuration of PETSc options using its command-line interface dictates what matrix storage, solver type and preconditioner should be used. This should be specified in a `.petscrc` file inside your working directory, as command line options are not currently passed through to PETSc to avoid conflict with *Nektar++* options. As an example, to select a GMRES solver using an algebraic multigrid preconditioner, and view the residual convergence, one can use the configuration:

```
-ksp_monitor
-ksp_view
-ksp_type gmres
-pc_type gamg
```

Or to use MUMPS, one could use the options:

```
-ksp_type preonly
-pc_type lu
-pc_factor_mat_solver_package mumps
```

```
-mat_mumps_icntl_7 2
```

A final choice that can be specified is whether to use a *shell* approach. By default, *Nektar++* will construct a PETSc sparse matrix (or whatever matrix is specified on the command line). This may, however, prove suboptimal for higher order discretisations. In this case, you may choose to use the *Nektar++* matrix-vector operators, which by default use an assembly approach that can prove faster, by setting the `PETScMatMult SOLVERINFO` option to `Shell`:

```
<I PROPERTY="PETScMatMult" VALUE="Shell" />
```

The downside to this approach is that you are now constrained to using one of the *Nektar++* preconditioners. However, this does give access to a wider range of Krylov methods than are available inside *Nektar++* for more advanced users.

3.3.5 Boundary Regions and Conditions

Boundary conditions are defined by two XML elements. The first defines the boundary regions in the domain in terms of composite entities from the `GEOMETRY` section of the file. Each boundary region has a unique ID and are defined as,

```
<BOUNDARYREGIONS>
    <B ID=[id]> [composite-list] </B>
    ...
</BOUNDARYREGIONS>
```

For example,

```
<BOUNDARYREGIONS>
  <B ID="0"> C[2] </B>
  <B ID="1"> C[3] </B>
</BOUNDARYREGIONS>
```

The second XML element defines, for each variable, the condition to impose on each boundary region, and has the form,

```
<BOUNDARYCONDITIONS>
    <REGION REF="[regionID]">
      <[type1] VAR="[variable1]" VALUE="[expression1]" />
      ...
      <[typeN] VAR="[variableN]" VALUE="[expressionN]" />
    </REGION>
    ...
</BOUNDARYCONDITIONS>
```

There should be precisely one `REGION` entry for each `B` entry defined in the `BOUNDARYREGION` section above. For example, to impose a Dirichlet condition on both variables for a domain with a single region,

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="u" VALUE="sin(PI*x)*cos(PI*y)" />
    <D VAR="v" VALUE="sin(PI*x)*cos(PI*y)" />
  </REGION>
</BOUNDARYCONDITIONS>
```

Boundary condition specifications may refer to any parameters defined in the session file. The `REF` attribute corresponds to a defined boundary region. The tag used for each variable specifies the type of boundary condition to enforce.

3.3.5.1 Dirichlet (essential) condition

Dirichlet conditions are specified with the `D` tag.

Projection	Homogeneous support	Time-dependent support	Dimensions
CG	Yes	Yes	1D, 2D and 3D
DG	Yes	Yes	1D, 2D and 3D
HDG	Yes	Yes	1D, 2D and 3D

Example:

```
<!-- homogeneous condition -->
<D VAR="u" VALUE="0" />
<!-- inhomogeneous condition -->
<D VAR="u" VALUE="x^2+y^2+z^2" />
<!-- time-dependent condition -->
<D VAR="u" USERDEFINEDTYPE="TimeDependent" VALUE="x+t" />
```

3.3.5.2 Neumann (natural) condition

Neumann conditions are specified with the `N` tag.

Projection	Homogeneous support	Time-dependent support	Dimensions
CG	Yes	Yes	1D, 2D and 3D
DG	No	No	1D, 2D and 3D
HDG	?	?	?

Example:

```
<!-- homogeneous condition -->
<N VAR="u" VALUE="0" />
<!-- inhomogeneous condition -->
<N VAR="u" VALUE="x^2+y^2+z^2" />
<!-- time-dependent condition -->
<N VAR="u" USERDEFINEDTYPE="TimeDependent" VALUE="x+t" />
<!-- high-order pressure boundary condition (for IncNavierStokesSolver) -->
<N VAR="u" USERDEFINEDTYPE="H" VALUE="0" />
```

3.3.5.3 Periodic condition

Periodic conditions are specified with the `P` tag.

Projection	Homogeneous support	Dimensions
CG	Yes	1D, 2D and 3D
DG	No	2D and 3D

Example:

```
<BOUNDARYREGIONS>
  <B ID="0"> C[1] </B>
  <B ID="1"> C[2] </B>
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <P VAR="u" VALUE="[1]" />
  </REGION>
  <REGION REF="1">
    <P VAR="u" VALUE="[0]" />
  </REGION>
</BOUNDARYCONDITIONS>
```

Periodic boundary conditions are specified in a significantly different form to other conditions. The `VALUE` property is used to specify which `BOUNDARYREGION` is periodic with the current region in square brackets.

Caveats:

- A periodic condition must be set for "'both"' boundary regions; simply specifying a condition for region 0 or 1 in the above example is not enough.
- The order of the elements inside the composites defining periodic boundaries is important. For example, if 'C[0]' above is defined as edge IDs '0,5,4,3' and 'C[1]' as '7,12,2,1' then edge 0 is periodic with edge 7, 5 with 12, and so on.
- For the above reason, the composites must also therefore be of the same size.
- In three dimensions, care must be taken to correctly align triangular faces which are intended to be periodic. The top (degenerate) vertex should be aligned so that, if the faces were connected, it would lie at the same point on both triangles.

3.3.5.4 Time-dependent boundary conditions

Time-dependent boundary conditions may be specified through setting the `USERDEFINEDTYPE` attribute and using the parameter `t` where the current time is required. For example,

```
<D VAR="u" USERDEFINEDTYPE="TimeDependent" VALUE="sin(PI*(x-t))" />
```

3.3.5.5 Boundary conditions from file

Boundary conditions can also be loaded from file. The following example is from the Incompressible Navier-Stokes solver,

```
<REGION REF="1">
  <D VAR="u" FILE="Test_ChanFlow2D_bcsfromfiles_u_1.bc" />
  <D VAR="v" VALUE="0" />
  <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
</REGION>
```

Boundary conditions can also be loaded simultaneously from a file and from an analytic expression. For example in the scenario where a spatial boundary condition is read from a file, but needs to be modulated by a time-dependent analytic expression:

```
<REGION REF="1">
  <D VAR="u" VALUE="USERDEFINEDTYPE="TimeDependent" VALUE="sin(PI*(x-t))"
             FILE="bcsfromfiles_u_1.bc" />
</REGION>
```

In the case where both `VALUE` and `FILE` are specified, the values are multiplied together to give the final value for the boundary condition.

3.3.6 Functions

Finally, multi-variable functions such as initial conditions and analytic solutions may be specified for use in, or comparison with, simulations. These may be specified using expressions (`<E>`) or imported from a file (`<F>`) using the Nektar++ FLD file format

```
<FUNCTION NAME="ExactSolution">
  <E VAR="u" VALUE="sin(PI*x-advx*t))*cos(PI*(y-advy*t))" />
</FUNCTION>
<FUNCTION NAME="InitialConditions">
  <F VAR="u" FILE="session.rst" />
</FUNCTION>
```

A restart file is a solution file (in other words an .fld renamed as .rst) where the field data is specified. The expansion order used to generate the .rst file must be the same as that for the simulation. .pts files contain scattered point data which needs to be interpolated to the field. For further information on the file format and the different interpolation schemes, see section 5.3.12. All filenames must be specified relative to the location of the .xml file.

With the additional argument `TIMEDEPENDENT="1"`, different files can be loaded for each timestep. The filenames are defined using boost::format syntax where the step time is used as variable. For example, the function `Baseflow` would load the files `U0V0_1.00000000E-05.fld`, `U0V0_2.00000000E-05.fld` and so on.

```
<FUNCTION NAME="Baseflow">
       <F VAR="U0,V0" TIMEDEPENDENT="1"FILE="U0V0_%14.8E.fld"/>
</FUNCTION>
```

For .pts files, the time consuming computation of interpolation weights in only performed for the first timestep. The weights are stored and reused in all subsequent steps, which is why all consecutive .pts files must use the same ordering, number and location of data points.

Other examples of this input feature can be the insertion of a forcing term,

```
<FUNCTION NAME="BodyForce">
  <E VAR="u" VALUE="0" />
  <E VAR="v" VALUE="0" />
</FUNCTION>
<FUNCTION NAME="Forcing">
  <E VAR="u" VALUE="-(Lambda + 2*PI*PI)*sin(PI*x)*sin(PI*y)" />
</FUNCTION>
```

or of a linear advection term

```
<FUNCTION NAME="AdvectionVelocity">
  <E VAR="Vx" VALUE="1.0" />
  <E VAR="Vy" VALUE="1.0" />
  <E VAR="Vz" VALUE="1.0" />
</FUNCTION>
```

3.3.6.1 Remapping variable names

Note that it is sometimes the case that the variables being used in the solver do not match those saved in the FLD file. For example, if one runs a three-dimensional incompressible Navier-Stokes simulation, this produces an FLD file with the variables `u`, `v`, `w` and `p`. If we wanted to use this velocity field as input for an advection velocity, the advection-diffusion-reaction solver expects the variables `Vx`, `Vy` and `Vz`.

We can manually specify this mapping by adding a colon to the

```
<FUNCTION NAME="AdvectionVelocity">
  <F VAR="Vx,Vy,Vz" FILE="file.fld:u,v,w" />
</FUNCTION>
```

There are some caveats with this syntax:

- You must specify the same number of fields for both the variable, and after the colon. For example, the following is not valid.

  ```
  <FUNCTION NAME="AdvectionVelocity">
    <F VAR="Vx,Vy,Vz" FILE="file.fld:u" />
  </FUNCTION>
  ```

- This syntax is not valid with the wildcard operator `*`, so one cannot write for example:

  ```
  <FUNCTION NAME="AdvectionVelocity">
    <F VAR="*" FILE="file.fld:u,v,w" />
  </FUNCTION>
  ```

3.3.6.2 Time-dependent file-based functions

With the additional argument `TIMEDEPENDENT="1"`, different files can be loaded for each timestep. The filenames are defined using boost::format syntax where the step time is used as variable. For example, the function `Baseflow` would load the files `U0V0_1.00000000E-05.fld`, `U0V0_2.00000000E-05.fld` and so on.

```
<FUNCTION NAME="Baseflow">
  <F VAR="U0,V0" TIMEDEPENDENT="1" FILE="U0V0_%14.8R.fld" />
</FUNCTION>
```

Section 3.6 provides the list of acceptable mathematical functions and other related technical details.

3.3.7 Quasi-3D approach

To generate a Quasi-3D appraoch with Nektar++ we only need to create a 2D or a 1D mesh, as reported above, and then specify the parameters to extend the problem to a 3D case. For a 2D spectral/hp element problem, we have a 2D mesh and along with the parameters we need to define the problem (i.e. equation type, boundary conditions, etc.). The only thing we need to do, to extend it to a Quasi-3D approach, is to specify some additional parameters which characterise the harmonic expansion in the third direction. First we need to specify in the solver information section that that the problem will be extended to have one homogeneouns dimension; here an example

```
<SOLVERINFO>
  <I PROPERTY="SolverType"            VALUE="VelocityCorrectionScheme"/>
  <I PROPERTY="EQTYPE"                VALUE="UnsteadyNavierStokes"/>
  <I PROPERTY="AdvectionForm"         VALUE="Convective"/>
  <I PROPERTY="Projection"            VALUE="Galerkin"/>
  <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2"/>
  <I PROPERTY="HOMOGENEOUS"           VALUE="1D"/>
</SOLVERINFO>
```

then we need to specify the parameters which define the 1D harmonic expanson along the z-axis, namely the homogeneous length (LZ) and the number of modes in the homogeneous direction (HomModesZ). HomModesZ corresponds also to the number of quadrature points in the homogenous direction, hence on the number of 2D planes discretized with a specral/hp element method.

```
<PARAMETERS>
  <P> TimeStep      = 0.001   </P>
  <P> NumSteps      = 1000    </P>
  <P> IO_CheckSteps = 100     </P>
  <P> IO_InfoSteps  = 10      </P>
  <P> Kinvis        = 0.025   </P>
  <P> HomModesZ     = 4       </P>
  <P> LZ            = 1.0     </P>
</PARAMETERS>
```

In case we want to create a Quasi-3D approach starting form a 1D spectral/hp element mesh, the procedure is the same, but we need to specify the parameters for two harmonic directions (in Y and Z direction). For Example,

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvectionDiffusion" />
  <I PROPERTY="Projection"            VALUE="Continuous"/>
  <I PROPERTY="HOMOGENEOUS"           VALUE="2D"/>
  <I PROPERTY="DiffusionAdvancement"  VALUE="Implicit"/>
  <I PROPERTY="AdvectionAdvancement"  VALUE="Explicit"/>
  <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2"/>
</SOLVERINFO>
<PARAMETERS>
  <P> TimeStep      = 0.001 </P>
  <P> NumSteps      = 200   </P>
  <P> IO_CheckSteps = 200   </P>
  <P> IO_InfoSteps  = 10    </P>
  <P> wavefreq      = PI    </P>
  <P> epsilon       = 1.0   </P>
  <P> Lambda        = 1.0   </P>
  <P> HomModesY     = 10    </P>
  <P> LY            = 6.5   </P>
  <P> HomModesZ     = 6     </P>
  <P> LZ            = 2.0   </P>
</PARAMETERS>
```

By default the opeartions associated with the harmonic expansions are performed with the Matix-Vector-Multiplication (MVM) defined inside the code. The Fast Fourier Transofrm (FFT) can be used to speed up the operations (if the FFTW library has been compiled in ThirdParty, see the compilation instructions). To use the FFT routines we need just to insert a flag in the solver information as below:

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvectionDiffusion" />
  <I PROPERTY="Projection"            VALUE="Continuous"/>
  <I PROPERTY="HOMOGENEOUS"           VALUE="2D"/>
  <I PROPERTY="DiffusionAdvancement"  VALUE="Implicit"/>
  <I PROPERTY="AdvectionAdvancement"  VALUE="Explicit"/>
  <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2"/>
  <I PROPERTY="USEFFT"                VALUE="FFTW"/>
</SOLVERINFO>
```

The number of homogenenous modes has to be even. The Quasi-3D apporach can be created starting from a 2D mesh and adding one homogenous expansion or starting form a 1D mesh and adding two homogeneous expansions. Not other options available. In case of a 1D homogeneous extension, the homogeneous direction will be the z-axis. In case of a 2D homogeneous extension, the homogeneous directions will be the y-axis and the z-axis.

3.4 Filters

Filters are a method for calculating a variety of useful quantities from the field variables as the solution evolves in time, such as time-averaged fields and extracting the field variables at certain points inside the domain. Each filter is defined in a `FILTER` tag inside a `FILTERS` block which lies in the main `NEKTAR` tag. In this section we give an overview of the modules currently available and how to set up these filters in the session file.

Here is an example `FILTER`:

```
<FILTER TYPE="FilterName">
    <PARAM NAME="Param1"> Value1 </PARAM>
    <PARAM NAME="Param2"> Value2 </PARAM>
</FILTER>
```

A filter has a name – in this case, `FilterName` – together with parameters which are set to user-defined values. Each filter expects different parameter inputs, although where functionality is similar, the same parameter names are shared between filter types for consistency. Numerical filter parameters may be expressions and so may include session parameters defined in the `PARAMETERS` section.

In the following we document the filters implemented. Note that some filters are solver-specific and will therefore only work for a given subset of the available solvers.

3.4.1 Time-averaged fields

This filter computes time-averaged fields for each variable defined in the session file. Time averages are computed by either taking a snapshot of the field every timestep, or alternatively at a user-defined number of timesteps N. An output is produced at the end of the simulation into `session_avg.fld`, or alternatively every M timesteps as defined by the user, into a sequence of files `session_*_avg.fld`, where `*` is replaced by a counter. This latter option can be useful to observe statistical convergence rates of the averaged variables.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which average fields are written.
`SampleFrequency`	✗	1	Number of timesteps at which the average is calculated, N.
`OutputFrequency`	✗	`NumSteps`	Number of timesteps after which output is written, M.

As an example, consider:

```
<FILTER TYPE="AverageFields">
    <PARAM NAME="OutputFile">MyAverageField</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency"> 10 </PARAM>
</FILTER>
```

This will create a file named `MyAverageField.fld` averaging the instantaneous fields every 10 time steps. The averaged field is however only output every 100 time steps.

3.4.2 Moving average of fields

This filter computes the exponential moving average (in time) of fields for each variable defined in the session file. The moving average is defined as:

$$\bar{u}_n = \alpha u_n + (1 - \alpha)\bar{u}_{n-1}$$

with $0 < \alpha < 1$ and $\bar{u}_1 = u_1$.

The same parameters of the time-average filter are supported, with the output file in the form `session_*_movAvg.fld`. In addition, either α or the time-constant τ must be defined. They are related by:

$$\alpha = \frac{t_s}{\tau + t_s}$$

where t_s is the time interval between consecutive samples.

As an example, consider:

```
<FILTER TYPE="MovingAverage">
    <PARAM NAME="OutputFile">MyMovingAverage</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency"> 10 </PARAM>
    <PARAM NAME="tau"> 0.1 </PARAM>
</FILTER>
```

This will create a file named `MyMovingAverage_movAvg.fld` with a moving average sampled every 10 time steps. The averaged field is however only output every 100 time steps.

3.4.3 Reynolds stresses

Note

This filter is only supported for the incompressible Navier-Stokes solver.

This filter is an extended version of the time-average filter. It outputs not only the time-average of the fields, but also the Reynolds stresses. The same parameters supported in the time-average case can be used, for example:

```
<FILTER TYPE="ReynoldsStresses">
    <PARAM NAME="OutputFile">MyAverageField</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency"> 10 </PARAM>
</FILTER>
```

By default, this filter uses a simple average. Optionally, an exponential time average can be used, in which case the output contain the moving averages and the Reynolds stresses calculated based on them. For example:

```
<FILTER TYPE="ReynoldsStresses">
    <PARAM NAME="OutputFile">MyAverageField</PARAM>
    <PARAM NAME="MovingAverage">true</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency"> 10 </PARAM>
    <PARAM NAME="alpha"> 0.01 </PARAM>
</FILTER>
```

3.4.4 Checkpoint fields

The checkpoint filter writes a checkpoint file, containing the instantaneous state of the solution fields at at given timestep. This can subsequently be used for restarting the simulation or examining time-dependent behaviour. This produces a sequence of files, by default named `session_*.chk`, where `*` is replaced by a counter. The initial condition is written to `session_0.chk`.

Note

This functionality is equivalent to setting the `IO_CheckSteps` parameter in the session file.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the checkpoints are written.
`OutputFrequency`	✓	-	Number of timesteps after which output is written.

For example, to output the fields every 100 timesteps we can specify:

```
<FILTER TYPE="Checkpoint">
    <PARAM NAME="OutputFile">IntermediateFields</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
</FILTER>
```

3.4.5 History points

The history points filter can be used to evaluate the value of the fields in specific points of the domain as the solution evolves in time. By default this produces a file called `session.his`. For each timestep, and then each history point, a line is output containing the current solution time, followed by the value of each of the field variables. Commented lines are created at the top of the file containing the location of the history points and the order of the variables.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the checkpoints are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`OutputPlane`	✗	0	If the simulation is homogeneous, the plane on which to evaluate the history point. (No Fourier interpolation is currently implemented.)
`Points`	✓	-	A list of the history points. These should always be given in three dimensions.

For example, to output the value of the solution fields at three points $(1, 0.5, 0)$, $(2, 0.5, 0)$ and $(3, 0.5, 0)$ into a file `TimeValues.his` every 10 timesteps, we use the syntax:

```
<FILTER TYPE="HistoryPoints">
    <PARAM NAME="OutputFile">TimeValues</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="Points">
        1 0.5 0
        2 0.5 0
        3 0.5 0
    </PARAM>
</FILTER>
```

3.4.6 ThresholdMax

The threshold value filter writes a field output containing a variable m, defined by the time at which the selected variable first exceeds a specified threshold value. The default name of the output file is the name of the session with the suffix `_max.fld`. Thresholding is applied based on the first variable listed in the session by default.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	*session*_max.fld	Output filename to which the threshold times are written.
`ThresholdVar`	✗	*first variable name*	Specifies the variable on which the threshold will be applied.
`ThresholdValue`	✓	-	Specifies the threshold value.
`InitialValue`	✓	-	Specifies the initial time.
`StartTime`	✗	0.0	Specifies the time at which to start recording.

An example is given below:

```
<FILTER TYPE="ThresholdMax">
    <PARAM NAME="OutputFile"> threshold_max.fld </PARAM>
    <PARAM NAME="ThresholdVar"> u </PARAM>
    <PARAM NAME="ThresholdValue"> 0.1 </PARAM>
    <PARAM NAME="InitialValue">  0.4 </PARAM>
</FILTER>
```

which produces a field file `threshold_max.fld`.

3.4.7 ThresholdMin value

Performs the same function as the `ThresholdMax` filter but records the time at which the threshold variable drops below a prescribed value.

3.4.8 One-dimensional energy

This filter is designed to output the energy spectrum of one-dimensional elements. It transforms the solution field at each timestep into a orthogonal basis defined by the functions

$$\psi_p(\xi) = L_p(\xi)$$

where L_p is the p-th Legendre polynomial. This can be used to show the presence of, for example, oscillations in the underlying field due to numerical instability. The resulting output is written into a file called `session.eny` by default. The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the energy spectrum is written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.

An example syntax is given below:

```
<FILTER TYPE="Energy1D">
    <PARAM NAME="OutputFile">EnergyFile</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
</FILTER>
```

3.4.9 Modal energy

Note

This filter is only supported for the incompressible Navier-Stokes solver.

This filter calculates the time-evolution of the kinetic energy. In the case of a two- or three-dimensional simulation this is defined as

$$E_k(t) = \frac{1}{2}\int_\Omega \|\mathbf{u}\|^2 \, dx$$

However if the simulation is written as a one-dimensional homogeneous expansion so that

$$\mathbf{u}(\mathbf{x}, t) = \sum_{k=0}^{N} \hat{\mathbf{u}}_k(t) e^{2\pi i k \mathbf{x}}$$

then we instead calculate the energy spectrum

$$E_k(t) = \frac{1}{2}\int_\Omega \|\hat{\mathbf{u}}_k\|^2 \, dx.$$

Note that in this case, each component of $\hat{\mathbf{u}}_k$ is a complex number and therefore $N =$ `HomModesZ`$/2$ lines are output for each timestep. This is a particularly useful tool in examining turbulent and transitional flows which use the homogeneous extension. In either case, the resulting output is written into a file called `session.mdl` by default.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the energy spectrum is written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.

An example syntax is given below:

```
<FILTER TYPE="ModalEnergy">
    <PARAM NAME="OutputFile">EnergyFile</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
</FILTER>
```

3.4.10 Aerodynamic forces

Note
This filter is only supported for the incompressible Navier-Stokes solver.

This filter evaluates the aerodynamic forces along a specific surface. The forces are projected along the Cartesian axes and the pressure and viscous contributions are computed in each direction.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the forces are written.
`Frequency`	✗	1	Number of timesteps after which output is written.
`Boundary`	✓	-	Boundary surfaces on which the forces are to be evaluated.

An example is given below:

```
<FILTER TYPE="AeroForces">
    <PARAM NAME="OutputFile">DragLift.frc</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="Boundary"> B[1,2] </PARAM>
</FILTER>
```

During the execution a file named `DragLift.frc` will be created and the value of the aerodynamic forces on boundaries 1 and 2, defined in the `GEOMETRY` section, will be output every 10 time steps.

3.4.11 Kinetic energy and enstrophy

Note
This filter is only supported for the incompressible and compressible Navier-Stokes solvers **in three dimensions**.

The purpose of this filter is to calculate the kinetic energy and enstrophy

$$E_k = \frac{1}{2\mu(\Omega)} \int_\Omega \|\mathbf{u}\|^2 \, dx, \qquad \mathcal{E} = \frac{1}{2\mu(\Omega)} \int_\Omega \|\omega\|^2 \, dx$$

where $\mu(\Omega)$ is the volume of the domain Ω. This produces a file containing the time-evolution of the kinetic energy and enstrophy fields. By default this file is called `session.eny` where `session` is the session name.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session.eny`	Output file name to which the energy and enstrophy are written.
`OutputFrequency`	✓	-	Number of timesteps at which output is written.

To enable the filter, add the following to the `FILTERS` tag:

```
<FILTER TYPE="Energy">
    <PARAM NAME="OutputFrequency"> 1 </PARAM>
</FILTER>
```

3.5 Forcing

An optional section of the file allows forcing functions to be defined. These are enclosed in the `FORCING` tag. The forcing type is enclosed within the `FORCE` tag and expressed in the file as:

```
<FORCE TYPE="[NAME]">
    ...
</FORCE>
```

The force type can be any of the following:

- "Absorption"
- "Body"
- "Programmatic"
- "Noise"

3.5.1 Absorption

This force type allows the user to apply an absorption layer (essentially a porous region) anywhere in the domain. The user may also specify a velocity profile to be imposed at the start of this layer, and in the event of a time-dependent simulation, this profile can be modulated with a time-dependent function. These velocity functions and the function defining the region in which to apply the absorption layer are expressed in the `CONDITIONS` section, however the name of these functions are defined here by the `COEFF`

tag for the layer, the `REFFLOW` tag for the velocity profile, and the `REFFLOWTIME` for the time-dependent function.

```
<FORCE TYPE="Absorption">
    <COEFF> [FUNCTION NAME] <COEFF/>
    <REFFLOW> [FUNCTION NAME] <REFFLOW/>
    <REFFLOWTIME> [FUNCTION NAME] <REFFLOWTIME/>
</FORCE>
```

3.5.2 Body

This force type specifies the name of a body forcing function expressed in the `CONDITIONS` section.

```
<FORCE TYPE="Body">
    <BODYFORCE> [FUNCTION NAME] <BODYFORCE/>
</FORCE>
```

3.5.3 Programmatic

This force type allows a forcing function to be applied directly within the code, thus it has no associated function.

```
<FORCE TYPE="Programmatic">
</FORCE>
```

3.5.4 Noise

This force type allows the user to specify the magnitude of a white noise force. Optional arguments can also be used to define the frequency in time steps to recompute the noise (default is never) and the number of time steps to apply the noise (default is the entire simulation).

```
<FORCE TYPE="Noise">
    <WHITENOISE> [VALUE] <WHITENOISE/>
    <!-- Optional arguments -->
    <UPDATEFREQ> [VALUE] <UPDATEFREQ/>
    <NSTEPS> [VALUE] <NSTEPS/>
</FORCE>
```

3.6 Analytic Expressions

This section discusses particulars related to analytic expressions appearing in Nektar++. Analytic expressions in Nektar++ are used to describe spatially or temporally varying properties, for example

- velocity profiles on a boundary
- some reference functions (e.g. exact solutions)

which can be retrieved in the solver code.

Analytic expressions appear as the content of `VALUE` attribute of

- boundary condition type tags within `<REGION>` subsection of `<BOUNDARYCONDITIONS>`, e.g. `<D>`, `<N>` etc.
- expression declaration tag `<E>` within `<FUNCTION>` subsection.

The tags above declare analytic expressions as well as link them to one of the field variables declared in `<EXPANSIONS>` section. For example, the declaration

```
<D VAR="u" VALUE="sin(PI*x)*cos(PI*y)" />
```

registers expression $\sin(\pi x)\cos(\pi y)$ as a Dirichlet boundary constraint associated with field variable `u`.

Enforcing the same velocity profile at multiple boundary regions and/or field variables results in repeated re-declarations of a corresponding analytic expression. Currently one cannot directly link a boundary condition declaration with an analytic expression uniquely specified somewhere else, e.g. in the `<FUNCTION>` subsection. However this duplication does not affect an overall computational performance.

3.6.1 Variables and coordinate systems

Declarations of analytic expressions are formulated in terms of problem space-time coordinates. The library code makes a number of assumptions to variable names and their order of appearance in the declarations. This section describes these assumptions.

Internally, the library uses 3D global coordinate space regardless of problem dimension. Internal global coordinate system has natural basis (1,0,0),(0,1,0),(0,0,1) with coordinates "'x"','y"' and "'z"'. In other words, variables "'x"','y"' and "'z"' are considered to be first, second and third coordinates of a point ("'x"','y"','z"').

Declarations of problem spatial variables do not exist in the current XML file format. Even though field variables are declarable as in the following code snippet,

```
<VARIABLES>
  <V ID="0"> u </V>
  <V ID="1"> v </V>
</VARIABLES>
```

there are no analogous tags for space variables. However an attribute `SPACE` of `<GEOMETRY>` section tag declares the dimension of problem space. For example,

```
<GEOMETRY DIM="1" SPACE="2"> ...
</GEOMETRY>
```

specifies 1D flow within 2D problem space. The number of spatial variables presented in expression declaration should match space dimension declared via `<GEOMETRY>` section tag.

The library assumes the problem space also has natural basis and spatial coordinates have names "'x"','y"' and"'z"'.

Problem space is naturally embedded into the global coordinate space: each point of

- 1D problem space with coordinate x is represented by 3D point (x,0,0) in the global coordinate system;
- 2D problem space with coordinates (x,y) is represented by 3D point (x,y,0) in the global coordinate system;
- 3D problem space with coordinates (x,y,z) has the same coordinates in the global space coordinates.

Currently, there is no way to describe rotations and translations of problem space relative to the global coordinate system.

The list of variables allowed in analytic expressions depends on the problem dimension:

- For 1D problem analytic expressions must make use of variable "'x"' only;
- For 2D problem analytic expressions should make use of variables "'x"' and "'y"'.
- 3D problems may use variables "'x"', "'y"' and "'z"' in their analytic expressions.

Violation of these constraints yields unpredictable results of expression evaluation. The current implementation assigns magic value -9999 to each dimensionally excessive spacial variable appearing in analytic expressions. For example, the following declaration

```
<GEOMETRY DIM="2" SPACE="2"> ...
</GEOMETRY> ...
<CONDITIONS> ...
  <BOUNDARYCONDITIONS>
     <REGION REF="0">
       <D VAR="u" VALUE="x+y+z" /> <D VAR="v" VALUE="sin(PI*x)*cos(PI*y)" />
     </REGION>
   </BOUNDARYCONDITIONS>
...
</CONDITIONS>
```

results in expression $x + y + z$ being evaluated at spatial points $(x_i, y_i, -9999)$ where x_i and y_i are the spacial coordinates of boundary degrees of freedom. However, the library behaviour under this constraint violation may change at later stages of development (e.g., magic constant 0 may be chosen) and should be considered unpredictable.

Another example of unpredictable behaviour corresponds to wrong ordering of variables:

```
<GEOMETRY DIM="1" SPACE="1"> ...
</GEOMETRY> ...
<CONDITIONS> ...
  <BOUNDARYCONDITIONS>
     <REGION REF="0">
       <D VAR="u" VALUE="sin(y)" />
     </REGION>
   </BOUNDARYCONDITIONS>
...
</CONDITIONS>
```

Here one declares 1D problem, so Nektar++ library assumes spacial variable is "'x"'. At the same time, an expression $sin(y)$ is perfectly valid on its own, but since it does not depend on "'x"', it will be evaluated to constant $sin(-9999)$ regardless of degree of freedom under consideration.

3.6.1.1 Time dependence

Variable "'t"' represents time dependence within analytic expressions. The boundary condition declarations need to add an additional property `USERDEFINEDTYPE="TimeDependent"` in order to flag time dependency to the library.

3.6.1.2 Syntax of analytic expressions

Analytic expressions are formed of

- brackets (). Bracketing structure must be balanced.
- real numbers: every representation is allowed that is correct for `boost::lexical_cast<double>()`, e.g.

```
  1.2, 1.2e-5, .02
```

- mathematical constants

Identifier	Meaning	Real Value
	Fundamental constants	
E	Natural Logarithm	2.71828182845904523536
PI	π	3.14159265358979323846
GAMMA	Euler Gamma	0.57721566490153286060
DEG	deg/radian	57.2957795130823208768
PHI	golden ratio	1.61803398874989484820
	Derived constants	
LOG2E	$\log_2 e$	1.44269504088896340740
LOG10E	$\log_{10} e$	0.43429448190325182765
LN2	$\log_e 2$	0.69314718055994530942
PI_2	$\frac{\pi}{2}$	1.57079632679489661923
PI_4	$\frac{\pi}{4}$	0.78539816339744830962
1_PI	$\frac{1}{\pi}$	0.31830988618379067154
2_PI	$\frac{2}{\pi}$	0.63661977236758134308
2_SQRTPI	$\frac{2}{\sqrt{\pi}}$	1.12837916709551257390
SQRT2	$\sqrt{2}$	1.41421356237309504880
SQRT1_2	$\frac{1}{\sqrt{2}}$	0.70710678118654752440

- parameters: alphanumeric names with underscores, e.g. `GAMMA_123`, `GaM123_45a_`, `_gamma123` are perfectly acceptable parameter names. However parameter name cannot start with a numeral. Parameters must be defined with `<PARAMETERS>...</PARAMETERS>`. Parameters play the role of constants that may change their values in between of expression evaluations.

- variables (i.e., `x, y, z` and `t`)

- unary minus operator (e.g. `-x`)

- binary arithmetic operators `+, -, *, /, ^` Powering operator allows using real exponents (it is implemented with `std::pow()` function)

- boolean comparison operations `<, <=, >, >=, ==` evaluate their sub-expressions to real values 0.0 or 1.0.

- mathematical functions of one or two arguments:

Identifier	Meaning
`abs(x)`	absolute value $\lvert x \rvert$
`asin(x)`	inverse sine $\arcsin x$
`acos(x)`	inverse cosine $\arccos x$
`ang(x,y)`	computes polar coordinate $\theta = \arctan(y/x)$ from (x, y)
`atan(x)`	inverse tangent $\arctan x$
`atan2(y,x)`	inverse tangent function (used in polar transformations)
`ceil(x)`	round up to nearest integer $\lceil x \rceil$
`cos(x)`	cosine $\cos x$
`cosh(x)`	hyperbolic cosine $\cosh x$
`exp(x)`	exponential e^x
`fabs(x)`	absolute value (equivalent to `abs`)
`floor(x)`	rounding down $\lfloor x \rfloor$
`log(x)`	logarithm base e, $\ln x = \log x$
`log10(x)`	logarithm base 10, $\log_{10} x$
`rad(x,y)`	computes polar coordinate $r = \sqrt{x^2 + y^2}$ from (x, y)
`sin(x)`	sine $\sin x$
`sinh(x)`	hyperbolic sine $\sinh x$
`sqrt(x)`	square root $\sqrt{x}$
`tan(x)`	tangent $\tan x$
`tanh(x)`	hyperbolic tangent $\tanh x$

These functions are implemented by means of the cmath library: http://www.cplusplus.com/reference/clibrary/cmath/. Underlying data type is `double` at each stage of expression evaluation. As consequence, complex-valued expressions (e.g. $(-2)^{0}.123$) get value `nan` (not a number). The operator `^` is implemented via call to `std::pow()` function and accepts arbitrary real exponents.

- random noise generation functions. Currently implemented is `awgn(sigma)` - Gaussian Noise generator, where σ is the variance of normal distribution with zero mean. Implemented using the `boost::mt19937` random number generator with boost variate generators (see http://www.boost.org/libs/random)

3.6.1.3 Examples

Some straightforward examples include

- Basic arithmetic operators: `0.5*0.3164/(3000^0.25)`
- Simple polynomial functions: `y*(1-y)`
- Use of values defined in `PARAMETERS` section: `-2*Kinvis*(x-1)`
- More complex expressions involving trigonometric functions, parameters and constants: `(LAMBDA/2/PI)*exp(LAMBDA*x)*sin(2*PI*y)`

- Boolean operators for multi-domain functions: `(y<0)*sin(y) + (y>=0)*y`

3.6.2 Performance considerations

Processing analytic expressions is split into two stages:

- parsing with pre-evaluation of constant sub-expressions,
- evaluation to a number.

Parsing of analytic expressions with their partial evaluation take place at the time of setting the run up (reading an XML file). Each analytic expression, after being pre-processed, is stored internally and quickly retrieved when it turns to evaluation at given spatial-time point(s). This allows to perform evaluation of expressions at a large number of spacial points with minimal setup costs.

3.6.2.1 Pre-evaluation details

Partial evaluation of all constant sub-expressions makes no sense in using derived constants from table above. This means, either make use of pre-defined constant `LN10^2` or straightforward expression `log10(2)^2` results in constant `5.3018981104783980105` being stored internally after pre-processing. The rules of pre-evaluation are as follows:

- constants, numbers and their combinations with arithmetic, analytic and comparison operators are pre-evaluated,
- appearance of a variable or parameter at any recursion level stops pre-evaluation of all upper level operations (but doesn't stop pre-evaluation of independent parallel sub-expressions).

For example, declaration

```
1     <D VAR="u" VALUE="exp(-x*sin(PI*(sqrt(2)+sqrt(3))/2)*y )" />
```

results in expression `exp(-x*(-0.97372300937516503167)*y )` being stored internally: sub-expression `sin(PI*(sqrt(2)+sqrt(3))/2)` is evaluated to constant but appearance of `x` and `y` variables stops further pre-evaluation.

Grouping predefined constants and numbers together helps. Its useful to put brackets to be sure your constants do not run out and become factors of some variables or parameters.

Expression evaluator does not do any clever simplifications of input expressions, which is clear from example above (there is no point in double negation). The following subsection addresses the simplification strategy.

3.6.2.2 Preparing analytic expression

The total evaluation cost depends on the overall number of operations. Since evaluator is not making simplifications, it worth trying to minimise the total number of operations in input expressions manually.

Some operations are more computationally expensive than others. In an order of increasing complexity:

- `+, -, <, >, <=, >=, ==,`
- `*, /, abs, fabs, ceil, floor,`
- `^, sqrt, exp, log, log10, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan`.

For example,

- `x*x` is faster than `x^2` — it is one double multiplication vs generic calculation of arbitrary power with real exponents.
- `(x+sin(y))^2` is faster than `(x+sin(y))*(x+sin(y))` - sine is an expensive operation. It is cheaper to square complicated expression rather than compute it twice and add one multiplication.
- An expression `exp(-41*( (x+(0.3*cos(2*PI*t)))^2 + (0.3*sin(2*PI*t))^2 ))` makes use of 5 expensive operations (`exp`, `sin`, `cos` and power `^` twice) while an equivalent expression `exp(-41*( x*x+0.6*x*cos(2*PI*t) + 0.09 ))` uses only 2 expensive operations.

If any simplifying identity applies to input expression, it may worth applying it, provided it minimises the complexity of evaluation. Computer algebra systems may help.

3.6.2.3 Vectorized evaluation

Expression evaluator is able to calculate an expression for either given point (its space-time coordinates) or given array of points (arrays of their space-time coordinates, it uses SoA). Vectorized evaluation is faster then sequential due to a better data access pattern. Some expressions give measurable speedup factor 4.6. Therefore, if you are creating your own solver, it worth making vectorized calls.

Part II

Preprocessing & Postprocessing

Chapter 4

NekMesh

`NekMesh` is a utility bundled with *Nektar++* which has two purposes:

- allow foreign mesh file formats to be converted into *Nektar++*'s XML format;
- aide in the generation of high-order meshes through a series of supplied processing modules.

Note

`NekMesh` replaces a previous utility called MeshConvert. This change is to reflect the fact that the program no longer only converts and manipulates meshes but can now also generate them from a CAD definition. This mesh generator is in a early beta stage of development and as such is disabled by default. For the time being those not using the beta mesh generator can use `NekMesh` as they would have used MeshConvert, none of the functionality or methodology has changed.

There is also some limited support for other output formats. We begin by running through a basic example to show how a mesh can be converted from the widely-used mesh-generator `Gmsh` to the XML file format.

Note

The default since Jan 2016 is to output the .xml files in a compressed form where the VERTEX, EDGES, FACES, ELEMENTS and CURVED information is compressed into binary format which is then converted into base64. This is identified for each section by the attribute `COMPRESSED="B64Z-LittleEndian"`. To output in ascii format add the module option ":xml:uncompress" to the .xml file, i.e.

```
NekMesh file.msh newfile.xml:xml:uncompress
```

4.1 Exporting a mesh from Gmsh

To demonstrate how NekMesh works, we will define a simple channel-like 3D geometry. First, we must define the Gmsh geometry to be used. The Gmsh definition is given below, and is visualised in figure 4.1.

```
Point(1) = {-1, 0, 0, 1.0};
Point(2) = {-0.3, 0, 0, 1.0};
Line(3) = {1, 2};
s[] = Extrude {0, 0, 7} {
  Line{3}; Layers{5}; Recombine;
};
v[] = Extrude {{0, 0, 1}, {0, 0, 0}, Pi} {
  Surface{s[1]}; Layers{10}; Recombine;
};
```

Whilst a full tutorial on Gmsh is far beyond the scope of this document, note the use of the Recombine argument. This allows us to generate a structured hexahedral mesh; remove the first Recombine to generate a prismatic mesh and both occurances to generate a tetrahedral mesh. Increasing the Layers numbers refines the mesh in the radial and azimuthal direction respectively.

4.2 Defining physical surfaces and volumes

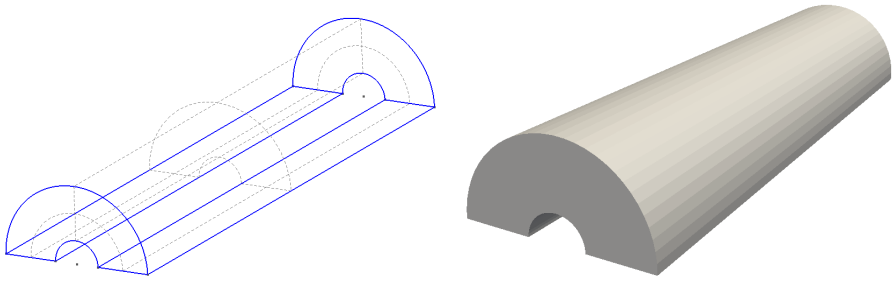


Figure 4.1 Geometry definition in Gmsh (left) and resulting high-order mesh visualised in ParaView (right).

In order for us to use the mesh, we need to define the physical surfaces which correspond to the inflow, outflow and walls so that we can set appropriate boundary conditions. The numbering resulting from the extrusions in this case is not straightforward. In the graphical interface, select `Geometry > Physical Groups > Add > Surface`, and then hover over each of the surfaces which are shown by the dashed gray lines. The numbering will be revealed in the toolbar underneath the geometry as a ruled surface. In this case:

- **Walls:** surfaces 7, 8, 28, 29.
- **Inflow:** surface 16.

- **Outflow:** surface 24.

We also need to define the physical volumes, which can be done in a similar fashion. For this example, there is only one volume having ID 1. Adding these groups to the end of the `.geo` file is very straightforward:

```
1 Physical Volume(0) = {1};
2 Physical Surface(1)= {7,8,28,29};
3 Physical Surface(2) = {16};
4 Physical Surface(3) = {24};
```

Either choose the option `File->Save Mesh` or, assuming this is saved in a file named `test.geo`, run the command

```
gmsh -3 test.geo
```

which will produce the resulting MSH file `test.msh`. One can generate a high-order mesh by specifying the order on the command line, for example

```
gmsh -3 -order 6 test.geo
```

will generate a sixth-order mesh. Note that you will need to use a current version of `Gmsh` in order to do this, most likely from subversion.

4.3 Converting the MSH to Nektar++ format

Assuming that you have compiled *Nektar++* according to the compilation instructions, run the command

```
NekMesh test.msh test.xml
```

to generate the XML file.

Note

This file contains only the geometry definition (and a default `EXPANSIONS` definition). In order to use this mesh, a `CONDITIONS` section must be supplied detailing the solver and parameters to use.

To validate the mesh visually, we can use a utility such as Paraview or VisIt. To do this, run the command

```
XmlToVtk test.xml
```

which generates an unstructured VTK file `test.vtu`.

It is possible that, when the high-order information was inserted into the mesh by `Gmsh`, invalid elements are generated which self intersect. In this case, the Jacobian of the mapping defining the curvature will have negative regions, which will generate warnings such as:

```
Warning: Level 0 assertion violation
3D deformed Jacobian not positive (element ID = 48) (first vertex ID = 105)
```

This tells you the element ID that is invalid, and the ID of the first vertex of the element. Whilst a resulting simulation may run, the results may not be valid because of this problem, or excessively large amounts of time may be needed to solve the resulting linear system.

4.4 NekMesh modules

`NekMesh` is designed to provide a pipeline approach to mesh generation. To do this, we break up tasks into three different types. Each task is called a *module* and a chain of modules specifies the pipeline.

- **Input** modules read meshes in a variety of formats;
- **Processing** modules modify meshes to aide in generation processes;
- **Output** modules write meshes in a variety of formats.

The figure below depicts how these might be coupled together to form a pipeline: On the

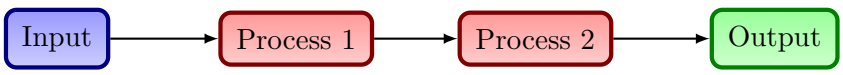


Figure 4.2 Illustrative pipeline of the `NekMesh` process.

command line, we would define this as:

```
NekMesh -m process1 -m process2 input.msh output.xml
```

Process modules can also have parameters passed to them, that can take arguments, or not.

```
NekMesh -m process1:p1=123:booleanparam input.msh output.xml
```

To list all available modules use the `-l` command line argument:

```
Available classes:
  Input: dat:
    Reads Tecplot polyhedron ascii format converted from Star CCM (.dat).
...
```

and then to see the options for a particular module, use the `-p` command line argument:

```
Options for module detect:
        vol: Tag identifying surface to process.
```

Note

Module names change when you use the `-p` option. Input modules should be preceded by `in:`, processing modules by `proc:` and output modules by `out:`.

4.4.1 Input modules

Input and output modules use file extension names to determine the correct module to use. Not every module is capable of reading high-order information, where it exists. The table below indicates support currently implemented.

Format	Extension	High-order	Notes
Gmsh	`msh`	✓	Only reads nodes, elements and physical groups (which are mapped to composites).
Nektar	`rea`	✓	Reads elements, fluid boundary conditions. Most curve types are unsupported: high-order information must be defined in an accompanying .hsf file.
Nektar++	`xml`	✓	Fully supported.
PLY	`ply`	✗	Reads only the ASCII format..
Semtex	`sem`	✓	Reads elements and boundary conditions. In order to read high-order information, run `meshpr session.sem > session.msh` and place in the same directory as the session file.
Star-CCM+	`dat`	✗	Star outputs plt file which currently needs to be coverted to ascii using Tecplot. Reads mesh only, only support for quads and triangles (2D) and hexes, prisms, tetrahedra (3D).
Star-CCM+	`ccm`	✗	Reads start ccm format. Reads mesh only, only support for quads and triangles (2D) and hexes, prisms, tetrahedra (3D). Requires NEKTAR_USE_CCM option to be activated in cmake and then requires ccmio library to be compiled by user.
VTK	`vtk`	✗	Experimental support. Only ASCII triangular data is supported.

Note that you can override the module used on the command line. For example, `Semtex` session files rarely have extensions. So for a session called `pipe-3d` we can convert this using the syntax

```
NekMesh pipe-3d:sem pipe-3d.xml
```

Typically, mesh generators allow physical surfaces and volumes to contain many element types; for example a cube could be constructed from a mixture of hexes and prisms. In *Nektar++*, a composite can only contain a single element type. Whilst the converter will attempt to preserve the numbering of composites from the original mesh type, sometimes a renumbering will occur when a domain contains many element types. For example, for a domain with the tag `150` containing quadrilaterals and triangles, the Gmsh reader will print a notification along the lines of:

```
Multiple elements in composite detected; remapped:
```

```
- Tag 150 => 150 (Triangle), 151 (Quadrilateral)
```

The resulting file therefore has two composites of IDs `150` and `151` respectively, containing the triangular and quadrilateral elements of the original mesh.

4.4.2 Output modules

The following output formats are supported:

Format	Extension	High-order	Notes
Gmsh	`msh`	✓	Curvature output is highly experimental.
Nektar++	`xml`	✓	Most functionality supported.
VTK	`vtk`	✗	Experimental. Only ASCII triangular data is supported.

Note that for both `Gmsh` and `VTK`, it is highly likely that you will need to experiment with the source code in order to successfully generate meshes since robustness is not guaranteed.

The default for `xml` is into binary data which has been converted into base64. If you wish to see an ascii output you need to specify the output module option `uncompress` by executing:

```
NekMesh Mesh.msh output.xml:xml:uncompress
```

In the rest of these subsections, we discuss the various processing modules available within `NekMesh`.

4.4.3 Extract surfaces from a mesh

To extract composite surfaces 2 and 3 from a mesh use the module `extract` module:

```
NekMesh -m extract:surf=2,3 Mesh.xml output.xml
```

If you also wish to have the boundaries of the extracted surface detected add the `detectbnd` option

```
NekMesh -m extract:surf=2,3:detectbnd Mesh.xml output.xml
```

4.4.4 Negative Jacobian detection

To detect elements with negative Jacobian determinant, use the `jac` module:

```
NekMesh -m jac Mesh.xml output.xml
```

To get a detailed list of elements which have negative Jacobians, one may use the `list` option:

```
NekMesh -m jac:list Mesh.xml output.xml
```

and to extract the elements for the purposes of visualisation within the domain, use the `extract` boolean parameter:

```
NekMesh -m jac:extract Mesh.xml MeshWithNegativeElements.xml
```

To turn off curvature associated with negative jacobians one can try to use the `removecurveifsingular` boolean parameter:

```
NekMesh -m jac:removecurveifsingular  Mesh.xml output.xml
```

This option will remove the high order curvature on prismatic faces with singular jacobians. This does not guarantee a non-singular mesh since it is possible for neighbouring element then to have singular jacobians. Multiple calls to the module might help with this scenario.

4.4.5 Spherigon patches

Where high-order information is not available (e.g. when using meshes from imaging software), various techniques can be used to apply a smoothing to the high-order element. In `NekMesh` we use *spherigons*, a kind of patch used in the computer graphics community used for efficiently smoothing polygon surfaces.

Spherigons work through the use of surface normals, where in this sense 'surface' refers to the underlying geometry. If we have either the exact or approximate surface normal at each given vertex, spherigon patches approximate the edges connecting two vertices by arcs of a circle. In `NekMesh` we can either approximate the surface normals from the linear elements which connect to each vertex (this is done by default), or supply a file which gives the surface normals.

To apply spherigon patches on two connected surfaces 11 and 12 use the following command:

```
NekMesh -m spherigon:surf=11,12 \
    MeshWithStraighEdges.xml MeshWithSpherigons.xml
```

If the two surfaces "11" and "12" are not connected, or connect at a sharp edge which is C^0 continuous but not C^1 smooth, use two separate instances of the spherigon module.

```
NekMesh -m spherigon:surf=11 -m spherigon:surf=12 \
    MeshWithStraighEdges.xml MeshWithSpherigons.xml
```

This is to avoid the approximated surface normals being incorrect at the edge.

If you have a high-resolution mesh of the surfaces 11 and 12 in `ply` format it can be used to improve the normal definition of the spherigons. Run:

```
NekMesh -m spherigon:surf=11,12:usenormalfile=Surf_11-12_Mesh.ply \
    MeshWithStraighEdges.xml MeshWithSpherigons.xml
```

This can be useful, for example, when meshing the Leading edge of an airfoil. Starting from a linear mesh (left figure) the spherigon patches curve the surface elements producing leading edge closer to the underlying geometry:

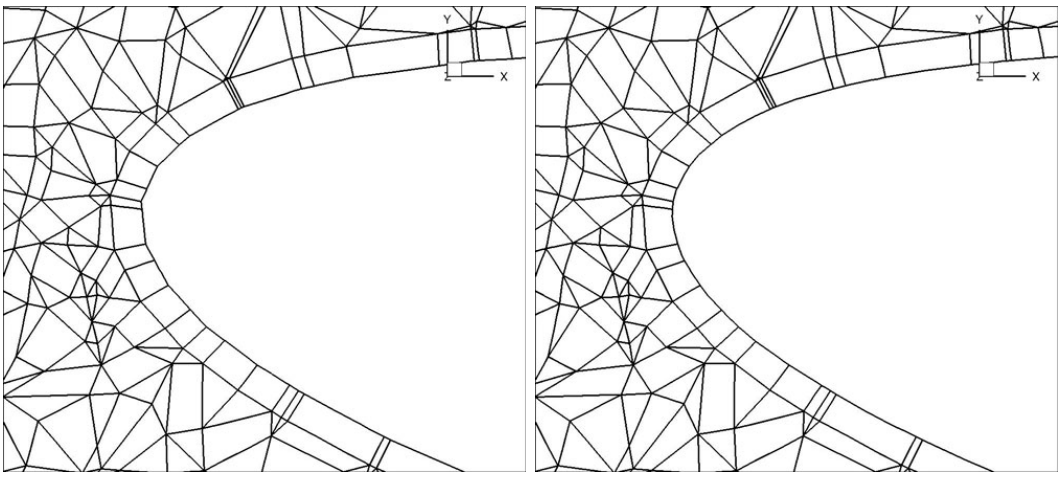


Figure 4.3 (a) Leading edge without spherigons, (b) Leading edge with spherigons

4.4.6 Periodic boundary condition alignment

When using periodic boundary conditions, the order of the elements within the boundary composite determines which element edges are periodic with the corresponding boundary composite.

To counteract this issue, `NekMesh` has a periodic alignment module which attempts to identify pairs of mutually periodic edges. Given two surfaces `surf1` and `surf2`, which for example correspond to the physical surface IDs specified in `Gmsh`, and an axis which defines the periodicity direction, the following command attempts to reorder the composites:

```
NekMesh -m peralign:surf1=11:surf2=12:dir=y \
    -m peralign:surf1=13:surf2=14:dir=z Mesh.xml Mesh_aligned.xml
```

Here the surfaces with IDs 11 and 12 will be aligned normal to the y-axis and the surfaces 13 and 14 will be aligned normal to the z-axis.

Note that this command cannot perform magic – it assumes that any given edge or face lying on the surface is periodic with another face on the opposing surface, that there are the same number of elements on both surfaces, and the corresponding edge or face is the same size and shape but translated along the appropriate axis.

In 3D, where prismatic or tetrahedral elements are connected to one or both of the surfaces, additional logic is needed to guarantee connectivity in the XML file. In this case we append the `orient` parameter:

```
NekMesh -m peralign:surf1=11:surf2=12:dir=y:orient input.dat output.xml
```

Note

One of the present shortcomings of `orient` is that it throws away all high-order information and works only on the linear element. This can be gotten around if you are just doing e.g. spherigon patches by running this `peralign` module before the `spherigon` module.

4.4.7 Boundary layer splitting

Often it is the case that one can generate a coarse boundary layer grid of a mesh. `NekMesh` has a method for splitting prismatic and hexahedral elements into finer elements based on the work presented in [22] and [23]. You must have a prismatic mesh that is O-type – that is, you can modify the boundary layer without modifying the rest of the mesh.

Given n layers, and a ratio r which defines the relative heights of elements in different layers, the method works by defining a geometric progression of points

$$x_k = x_{k-1} + ar^k, \quad a = \frac{2(1-r)}{1-r^{n+1}}$$

in the standard segment $[-1, 1]$. These are then projected into the coarse elements to construct a sequence of increasingly refined elements, as depicted in figure 4.4.

To split a prism boundary layer on surface 11 into 3 layers with a growth rate of 2 and 7 integration points per element use the following command:

```
NekMesh -m bl:surf=11:layers=3:r=2:nq=7 MeshWithOnePrismLayer.xml \
      MeshWith3PrismsLayers.xml
```

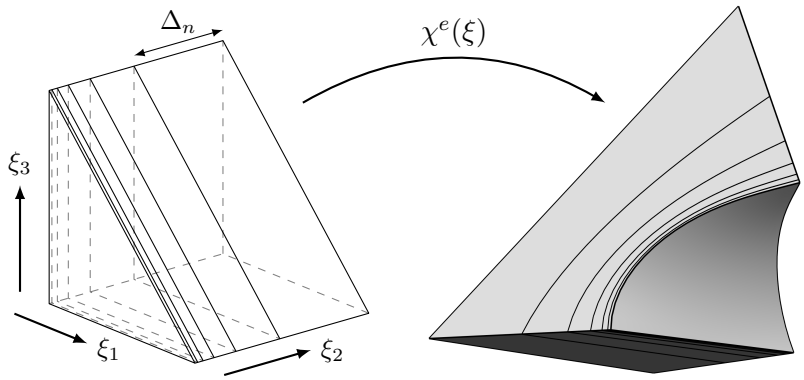


Figure 4.4 Splitting Ω_{st} and applying the mapping χ^e to obtain a high-order layer of prisms from the macro-element.

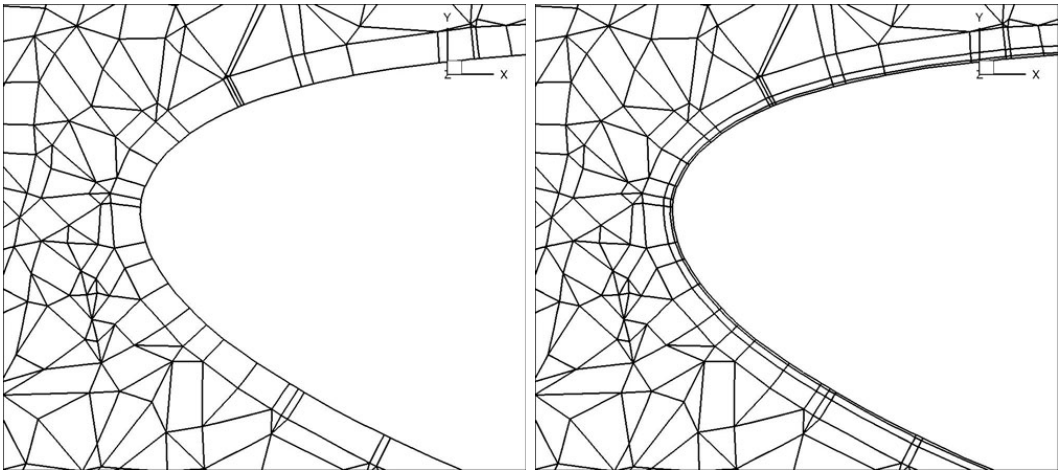


Figure 4.5 (a) LE with Spherigons but only one prism layer for resolving the boundary layer, (b) LE with Spherigons with 3 growing layers of prisms for better resolving the boundary layer.

Note

You can also use an expression in terms of coordinates (x, y, z) for r to make the ratio spatially varying; e.g. `r=sin(x)`. In this case the function should be sufficiently smooth to prevent the elements self-intersecting.

4.4.8 High-order cylinder generation

Generating accurate high-order curved geometries in `Gmsh` is quite challenging. This module processes an existing linear cylindrical mesh, with axis aligned with the z-coordinate axis, to generate accurate high-order curvature information along the edges.

```
NekMesh -m cyl:surf=2:r=1.0:N=5 LinearCylinder.xml HighOrderCylinder.xml
```

The module parameters are:

- `surf`: Surface on which to apply curvature. This should be the outer surface of the cylinder.
- `r`: Radius of the cylinder.
- `N`: Number of high-order points along each element edge.

Note

The module could also be used to apply curvature along the interior of a hollow cylinder. However, there are no checks to ensure the resulting elements are not self-intersecting.

4.4.9 Surface extraction

Often one wants to visualise a particular surface of a 3D mesh. `NekMesh` supports extraction of two-dimensional surfaces which can be converted using `XmlToVtk` or similar programs for visualisation purposes, or combined with `FieldConvert` in order to extract the value of a 3D field on a given surface.

To extract a surface use the command:

```
NekMesh -m extract:surf=12,3,4 volume-mesh.xml surface-mesh.xml
```

where the integers are surface IDs to be extracted.

An optional arguemnt of `detectbnd` can be added to identify the boundary composites as part of the surface extraction.

4.4.10 Linearisation

The ability to remove all the high-order information in a mesh can be useful at times.

To do this in NekMesh use the command:

```
NekMesh -m linearise high-order-mesh.xml linear-mesh.xml
```

The output will contain only the linear mesh information, all curved information is removed.

4.4.11 Extracting interface between tetrahedra and prismatic elements

When the mesh is three-dimensional and comprised of a prismatic boundary layer with tetrahedra in the interior of the domain, this module extracts the prismatic elements only, and constructs a boundary region for the interface between the tetrahedra and prisms. This is useful in, for example, the study of aortic flows, where the prismatic boundary layer can be extracted and refined to study unsteady advection-diffusion problems on a more refined grid inside the boundary layer.

To use this module you therefore use the command:

```
NekMesh -m extracttetprisminterface input.xml output.xml
```

There are no configuration options for this module, as it is highly specific to a certain class of meshes.

4.4.12 Boundary identification

Some mesh formats lack the ability to identify boundaries of the domain they discretise. `NekMesh` has a rudimentary boundary identification routine for conformal meshes, which will create a composite of edges (2D) or faces (3D) which are connected to precisely one element. This can be done using the `detect` module:

```
NekMesh -m detect volume.xml volumeWithBoundaryComposite.xml
```

4.4.13 Scalar function curvature

This module imposes curvature on a surface given a scalar function $z = f(x, y)$. For example, if on surface 1 we wish to apply a surface defined by a Gaussian $z = \exp[-(x^2 + y^2)]$ using 7 quadrature points in each direction, we may issue the command

```
NekMesh -m scalar:surf=1:nq=7:scalar=exp\(x*x+y*y\) mesh.xml deformed.xml
```

Note

This module makes no attempt to apply the curvature to the interior of the domain. Elements must therefore be coarse in order to prevent self-intersection. If a boundary layer is required, one option is to use this module in combination with the splitting module described earlier.

Chapter 5

FieldConvert

FieldConvert is a utility embedded in *Nektar++* with the primary aim of allowing the user to convert the *Nektar++* output binary files (.chk and .fld) into a format which can be read by two common visualisation softwares: Paraview (.vtu format) or Tecplot (.dat format). FieldConvert also allows the user to manipulate the *Nektar++* output binary files by using some additional modules which can be called with the option `-m` which stands for `m`odule. Note that another flag, `-r` (which stand for `r`ange) allows the user to specify a sub-range of the domain on which the conversion or manipulation of the *Nektar++* output binary files will be performed.

Almost all of the FieldConvert functionalities can be run in parallel if *Nektar++* is compiled using MPI (see the installation documentation for additional info on how to implement *Nektar++* using MPI). [1]

5.1 Convert .fld / .chk files into Paraview, VisIt or Tecplot format

To convert the *Nektar++* output binary files (.chk and .fld) into a format which can be read by two common visualisation softwares: Paraview (.vtu format), VisIt (.vtu format) or Tecplot (.dat format) the user can run the following commands:

- Paraview or VisIt (.vtu format)

```
FieldConvert test.xml test.fld test.vtu
```

- Tecplot (.dat format)

```
FieldConvert test.xml test.fld test.dat
```

[1] Modules that do not have parallel support will be specified in the appropriate section.

where `FieldConvert` is the executable associated to the utility FieldConvert, `test.xml` is the session file and `test.dat`, `test.vtu` are the desired format outputs, either Tecplot or Paraview format respectively.

Tip

Note that the session file is also supported in its compressed format `test.xml.gz`.

5.2 Range option *-r*

The Fieldconvert range option `-r` allows the user to specify a sub-range of the mesh (computational domain) by using an additional flag, `-r` (which stands for `r`ange and either convert or manipulate the *Nektar++* output binary files. Taking as an example the conversion of the *Nektar++* binary files (.chk or .fld) shown before and wanting to convert just the 2D sub-range defined by $-2 \leq x \leq 3$, $-1 \leq y \leq 2$ the additional flag `-r` can be used as follows:

- Paraview or VisIt (.vtu format)

```
FieldConvert -r -2,3,-1,2 test.xml test.fld test.vtu
```

- Tecplot (.dat format)

```
FieldConvert -r 2,3,-1,2 test.xml test.fld test.dat
```

where `-r` defines the range option of the FieldConvert utility, the two first numbers define the range in x direction and the the third and fourth number specify the y range. A sub-range of a 3D domain can also be specified. For doing so, a third set of numbers has to be provided to define the z range.

5.3 FieldConvert modules *-m*

FieldConvert allows the user to manipulate the *Nektar++* output binary files (.chk and .fld) by using the flag `-m` (which stands for `m`odule).. Specifically, FieldConvert has these additional functionalities

1. `C0Projection`: Computes the C0 projection of a given output file;
2. `QCriterion`: Computes the Q-Criterion for a given output file;
3. `addFld`: Sum two .fld files;

4. `combineAvg`: Combine two *Nektar++* binary output (.chk or .fld) field file containing averages of fields (and possibly also Reynolds stresses) into single file;
5. `concatenate`: Concatenate a *Nektar++* binary output (.chk or .fld) field file into single file;
6. `equispacedoutput`: Write data as equi-spaced output using simplices to represent the data for connecting points;
7. `extract`: Extract a boundary field;
8. `homplane`: Extract a plane from 3DH1D expansions;
9. `innerproduct`: take the inner product between one or a series of fields with another field (or series of fields).
10. `interpfield`: Interpolates one field to another, requires fromxml, fromfld to be defined;
11. `interppointdatatofld`: Interpolates given discrete data using a finite difference approximation to a fld file given an xml file;
12. `interppoints`: Interpolates a set of points to another, requires fromfld and fromxml to be defined, a line or plane of points can be defined;
13. `isocontour`: Extract an isocontour of "fieldid" variable and at value "fieldvalue". Optionally "fieldstr" can be specified for a string defiition or "smooth" for smoothing;
14. `jacobianenergy`: Shows high frequency energy of Jacobian;
15. `meanmode`: Extract mean mode (plane zero) of 3DH1D expansions;
16. `pointdatatofld`: Given discrete data at quadrature points project them onto an expansion basis and output fld file;
17. `printfldnorms`: Print L2 and LInf norms to stdout;
18. `scalargrad`: Computes scalar gradient field;
19. `scaleinputfld`: Rescale input field by a constant factor;
20. `shear`: Computes time-averaged shear stress metrics: TAWSS, OSI, transWSS, TAAFI, TACFI, WSSG;
21. `surfdistance`: Computes height of a prismatic boundary layer mesh and projects onto the surface (for e.g. y^+ calculation).
22. `vorticity`: Computes the vorticity field.
23. `wss`: Computes wall shear stress field.

The module list above can be seen by running the command

```
FieldConvert -l
```

In the following we will detail the usage of each module.

5.3.1 Smooth the data: *C0Projection* module

To smooth the data of a given .fld file one can use the `C0Projection` module of FieldConvert

```
FieldConvert -m C0Projection test.xml test.fld test-C0Proj.fld
```

where the file `test-C0Proj.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

The option `localtoglobalmap` will do a global gather of the coefficients and then scatter them back to the local elements. This will replace the coefficients shared between two elements with the coefficients of one of the elements (most likely the one with the highest id). Although not a formal projection it does not require any matrix inverse and so is very cheap to perform.

The option `usexmlbcs` will enforce the boundary conditions specified in the input xml file.

The option `helmsmoothing=`L will perform a Helmholtz smoothing projection of the form

$$\left(\nabla^2 + \left(\frac{2\pi}{L}\right)^2\right)\hat{u}^{new} = \left(\frac{2\pi}{L}\right)^2 \hat{u}^{orig}$$

which can be interpreted in a Fourier sense as smoothing the original coefficients using a low pass filter of the form

$$\hat{u}_k^{new} = \frac{1}{(1+k^2/K_0^2)}\hat{u}_k^{orig} \text{ where } K_0 = \frac{2\pi}{L}$$

and so L is the length scale below which the coefficients values are halved or more. Since this form of the Helmholtz operator is not possitive definite, currently a direct solver is necessary and so this smoother is mainly of use in two-dimensions.

5.3.2 Calculate Q-Criterion: *QCriterion* module

To perform the Q-criterion calculation and obtain an output data containing the Q-criterion solution, the user can run

```
FieldConvert -m QCriterion test.xml test.fld test-QCrit.fld
```

where the file `test-QCrit.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.3 Sum two .fld files: *addFld* module

To sum two .fld files one can use the `addFld` module of FieldConvert

```
FieldConvert -m addfld:fromfld=file1.fld:scale=-1 file1.xml file2.fld \
file3.fld
```

In this case we use it in conjunction with the command `scale` which multiply the values of a given .fld file by a constant `value`. `file1.fld` is the file multiplied by `value`, `file1.xml` is the associated session file, `file2.fld` is the .fld file which is summed to `file1.fld` and finally `file3.fld` is the output which contain the sum of the two .fld files. `file3.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.4 Combine two .fld files containing time averages: *combineAvg* module

To combine two .fld files obtained through the AverageFields or ReynoldsStresses filters, use the `combineAvg` module of FieldConvert

```
FieldConvert -m combineAvg:fromfld=file1.fld file1.xml file2.fld \
file3.fld
```

`file3.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.5 Concatenate two files: *concatenate* module

To concatenate `file1.fld` and `file2.fld` into `file-conc.fld` one can run the following command

```
FieldConvert -m concatenate file.xml  file1.fld  file2.fld  file-conc.fld
```

where the file `file-conc.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.6 Equi-spaced output of data: *equispacedoutput* module

This module interpolates the output data to an truly equispaced set of points (not equispaced along the collapsed coordinate system). Therefore a tetrahedron is represented by a tetrahedral number of poinst. This produces much smaller output files. The points are then connected together by simplices (triangles and tetrahedrons).

```
FieldConvert -m equispacedoutput test.xml test.fld test.dat
```

or

```
FieldConvert -m equispacedouttput test.xml test.fld test.vtu
```

Note

Currently this option is only set up for triangles, quadrilaterals, tetrahedrons and prisms.

5.3.7 Extract a boundary region: *extract* module

The boundary region of a domain can be extracted from the output data using the following command line

```
FieldConvert -m extract:bnd=2:fldtoboundary=1 test.xml \
        test.fld test-boundary.fld
```

The option `bnd` specifies which boundary region to extract. Note this is different to NekMesh where the parameter `surf` is specified and corresponds to composites rather boundaries. If `bnd` is not provided, all boundaries are extracted to different fields. The `fldtoboundary` is an optional command argument which copies the expansion of test.fld into the boundary region before outputting the .fld file. This option is on by default. If it turned off using `fldtoboundary=0` the extraction will only evaluate the boundary condition from the xml file. The output will be placed in test-boundary-b2.fld. If more than one boundary region is specified the extension -b0.fld, -b1.fld etc will be outputted. To process this file you will need an xml file of the same region. This can be generated using the command:

```
NekMesh -m extract:surf=5  test.xml test-b0.xml
```

The surface to be extracted in this command is the composite number and so needs to correspond to the boundary region of interest. Finally to process the surface file one can use

```
FieldConvert  test-b0.xml test-b0.fld test-b0.dat
```

This will obviously generate a Tecplot output if a .dat file is specified as last argument. A .vtu extension will produce a Paraview or VisIt output.

5.3.8 Compute the gradient of a field: *gradient* module

To compute the spatial gradients of all fields one can run the following command

```
FieldConvert -m gradient test.xml  test.fld  test-grad.fld
```

where the file `file-grad.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.9 Extract a plane from 3DH1D expansion: *homplane* module

To obtain a 2D expansion containing one of the planes of a 3DH1D field file, use the command:

```
FieldConvert -m homplane:planeid=value file.xml file.fld file-plane.fld
```

If the option `wavespace` is used, the Fourier coefficients corresponding to `planeid` are obtained. The command in this case is:

```
FieldConvert -m homplane:wavespace:planeid=value file.xml \
    file.fld file-plane.fld
```

The output file `file-plane.fld` can be processed in a similar way as described in section 5.1 to visualise it either in Tecplot or in Paraview.

5.3.10 Inner Product of a single or series of fields with respect to a single or series of fields: *innerproduct* module

You can take the inner product of one field with another field using the following command:

```
FieldConvert -m innerproduct:fromfld=file1.fld  file2.xml file2.fld \
out.stdout
```

This command will load the `file1.fld` and `file2.fld` assuming they both are spatially defined by `files.xml` and determine the inner product of these fields. The input option `fromfld` must therefore be specified in this module.

Optional arguments for this module are `fields` which allow you to specify the fields that you wish to use for the inner product, i.e.

```
FieldConvert -m innerproduct:fromfld=file1.fld:fields=''0,1,2'' file2.xml \
file2.fld out.stdout
```

will only take the inner product between the variables 0,1 and 2 in the two fields files. The default is to take the inner product between all fields provided.

Additional options include `multifldids` and `allfromflds` which allow for a series of fields to be evaluated in the following manner:

```
FieldConvert -m innerproduct:fromfld=file1.fld:multifldids=''0-3''\
file2.xml  file2.fld out.stdout
```

will take the inner product between a file names field1_0.fld, field1_1.fld, field1_2.fld and field1_3.fld with respect to field2.fld.

Analogously including the options `allfromflds`, i.e.

```
FieldConvert -m innerproduct:fromfld=file1.fld:multifldids=''0-3'':\
allfromflds  file2.xml file2.fld out.stdout
```

Will take the inner product of all the from fields, i.e. field1_0.fld,field1_1.fld,field1_2.fld and field1_3.fld with respect to each other. This option essentially ignores file2.fld. Only the unique inner products are evaluated so if four from fields are given only the related trianuglar number $4 \times 5/2 = 10$ of inner products are evaluated.

This option can be run in parallel.

5.3.11 Interpolate one field to another: *interpfield* module

To interpolate one field to another, one can use the following command:

```
FieldConvert -m interpfield:fromxml=file1.xml:fromfld=file1.fld \
        file2.xml file2.fld
```

This command will interpolate the field defined by `file1.xml` and `file1.fld` to the new mesh defined in `file2.xml` and output it to `file2.fld`. The `fromxml` and `fromfld` must be specified in this module. In addition there are two optional arguments `clamptolowervalue` and `clamptouppervalue` which clamp the interpolation between these two values. Their default values are -10,000,000 and 10,000,000.

Tip

This module can run in parallel where the speed is increased not only due to using more cores but also, since the mesh is split into smaller sub-domains, the search method currently adopted performs faster.

5.3.12 Interpolate scattered point data to a field: *interppointdatatofld* module

To interpolate discrete point data to a field, use the interppointdatatofld module:

```
FieldConvert -m interppointdatatofld file1.xml file1.pts file1.fld
```

This command will interpolate the data from `file1.pts` to the mesh and expansions defined in `file1.xml` and output the field to `file1.fld`. The file `file.pts` is of the form:

```
<?xml version="1.0" encoding="utf-8" ?>
<NEKTAR>
  <POINTS DIM="1" FIELDS="a,b,c">
    1.0000 -1.0000 1.0000 -0.7778
    2.0000 -0.9798 0.9798 -0.7980
    3.0000 -0.9596 0.9596 -0.8182
    4.0000 -0.9394 0.9394 -0.8384
  </POINTS>
</NEKTAR>
```

where `DIM="1" FIELDS="a,b,c` specifies that the field is one-dimensional and contains three variables, a, b, and c. Each line defines a point, while the first column contains its x-coordinate, the second one contains the a-values, the third the b-values and so on. In case of n-dimensional data, the n coordinates are specified in the first n columns accordingly. In order to interpolate 1D data to a nD field, specify the matching coordinate in the output field using the `interpcoord` argument:

```
FieldConvert -m interppointdatatofld:interppointdatatofld=1 3D-file1.xml \
               1D-file1.pts 3D-file1.fld
```

This will interpolate the 1D scattered point data from `1D-file1.pts` to the y-coordinate of the 3D mesh defined in `3D-file1.xml`. The resulting field will have constant values along the x and z coordinates. For 1D Interpolation, the module implements a quadratic scheme and automatically falls back to a linear method if only two data points are given. A modified inverse distance method is used for 2D and 3D interpolation. Linear and quadratic interpolation require the data points in the `.pts`-file to be sorted by their location in ascending order. The Inverse Distance implementation has no such requirement.

5.3.13 Interpolate a field to a series of points: *interppoints* module

You can interpolate one field to a series of given points using the following command:

```
FieldConvert -m interppoints:fromxml=file1.xml:fromfld=file1.fld \
        file2.pts file2.dat
```

This command will interpolate the field defined by `file1.xml` and `file1.fld` to the points defined in `file2.pts` and output it to `file2.dat`. The `fromxml` and `fromfld` must be specified in this module. The format of the file `file2.pts` is of the same form as for the *interppointdatatofld* module:

```
<?xml version="1.0" encoding="utf-8" ?>
<NEKTAR>
  <POINTS DIM="2" FIELDS="">
    0.0 0.0
    0.5 0.0
    1.0 0.0
  </POINTS>
</NEKTAR>
```

There are three optional arguments `clamptolowervalue`, `clamptouppervalue` and `defaultvalue` the first two clamp the interpolation between these two values and the third defines the default value to be used if the point is outside the domain. Their default values are -10,000,000, 10,000,000 and 0.

In addition, instead of specifying the file `file2.pts`, a module list of the form

```
FieldConvert -m interppoints:fromxml=file1.xml:fromfld= \
        file1.fld:line=npts,x0,y0,x1,y1
```

can be specified where `npts` is the number of equispaced points between $(x0, y0)$ to $(x1, y1)$ which can also be used in 3D by specifying $(x0, y0, z0)$ to $(x1, y1, z1)$.

An extraction of a plane of points can also be specified by

```
FieldConvert -m interppoints:fromxml=file1.xml:fromfld=file1.fld:\
      plane=npts1,npts2,x0,y0,z0,x1,y1,z1,x2,y2,z2,x3,y3,z3
```

where `npts1,npts2` is the number of equispaced points in each direction and $(x0, y0, z0)$, $(x1, y1, z1)$, $(x2, y2, z2)$ and $(x3, y3, z3)$ define the plane of points specified in a clockwise or anticlockwise direction.

In addition an extraction of a box of points can also be specified by

```
FieldConvert -m interppoints:fromxml=file1.xml:fromfld=file1.fld:\
     box=npts1,npts2,npts3,xmin,xmax,ymin,ymax,zmin,zmax
```

where `npts1,npts2,npts3` is the number of equispaced points in each direction and $(xmin, ymin, zmin)$ and $(xmax, ymax, zmax3)$ define the limits of the box of points.

For the plane and box interpolation there is an additional optional argument `cp=p0,q` which adds to the interpolated fields the value of $c_p = (p - p0)/q$ and $c_{p0} = (p - p0 +$

$0.5u^2)/q$ where $p0$ is a reference pressure and q is the free stream dynamics pressure. If the input does not contain a field "p" or a velocity field "u,v,w" then cp and $cp0$ are not evaluated accordingly

Note

This module runs in parallel for the plane and box extraction of points. In this case a series of .dat files are generated that can be concatinated together. Other options do not run in parallel.

5.3.14 Isocontour extraction: *iscontour* module

Extract an isocontour from a field file. This option automatically take the field to an equispaced distribution of points connected by linear simplicies of triangles or tetrahedrons. The linear simplices are then inspected to extract the isocontour of interest. To specify the field `fieldid` can be provided giving the id of the field of interest and `fieldvalue` provides the value of the isocontour to be extracted.

```
FieldConvert -m isocontour:fieldid=2:fieldvalue=0.5 test.xml test.fld \
        test-isocontour.dat
```

Alternatively `fieldstr`="u+v" can be specified to calculate the field u^2 and extract its isocontour. You can also specify `fieldname`="UplusV" to define the name of the isocontour in the .dat file, i.e.

```
FieldConvert -m isocontour:fieldstr="u+v":fieldvalue=0.5:\
  fieldname="UplusV" test.xml  test.fld test-isocontour.dat
```

Optionally `smooth` can be specified to smooth the isocontour with default values `smoothnegdiffusion`=0.495, `smoothnegdiffusion`=0.5 and `smoothiter`=100. This option typically should be used wiht the `globalcondense` option which removes multiply defined verties from the simplex definition which arise as isocontour are generated element by element. The `smooth` option preivously automatically called the `globalcondense` option but this has been depracated since it is now possible to read isocontour files directly and so it is useful to have these as separate options.

In addition to the `smooth` or `globalcondense` options you can specify `removesmallcontour`=100 which will remove separate isocontours of less than 100 triangles.

Note

Currently this option is only set up for triangles, quadrilaterals, tetrahedrons and prisms.

5.3.15 Show high frequency energy of the Jacobian: *jacobianenergy* module

```
FieldConvert -m jacobianenergy file.xml file.fld jacenergy.fld
```

The option `topmodes` can be used to specify the number of top modes to keep.

The output file `jacenergy.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.16 Extract mean mode of 3DH1D expansion: *meanmode* module

To obtain a 2D expansion containing the mean mode (plane zero in Fourier space) of a 3DH1D field file, use the command:

```
FieldConvert -m meanmode file.xml file.fld file-mean.fld
```

The output file `file-mean.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot or in Paraview or VisIt.

5.3.17 Project point data to a field: *pointdatatofld* module

To project a series of points given at the same quadrature distribution as the .xml file and write out a .fld file use the pointdatatofld module:

```
FieldConvert --noequispaced -m pointdatatofld file.pts file.xml file.fld
```

This command will read in the points provided in the `file.pts` and assume these are given at the same quadrature distribution as the mesh and expansions defined in `file.xml` and output the field to `file.fld`. If the points do not match an error will be dumped.

The file `file.pts` which is assumed to be given by an interpolation from another source is of the form:

```
<?xml version="1.0" encoding="utf-8" ?>
<NEKTAR>
  <POINTS DIM="3" FIELDS="p">
  1.70415 -0.4       -0.0182028      -0.106893
  1.70415 -0.395683  -0.0182028      -0.106794
  1.70415 -0.3875    -0.0182028      -0.106698
  1.70415 -0.379317  -0.0182028      -0.103815
  </POINTS>
</NEKTAR>
```

where `DIM="3" FIELDS="p` specifies that the field is three-dimensional and contains one variable, p. Each line defines a point, the first, second, and third columns contains the x, y, z-coordinate and subsequent columns contain the field values, in this case the p-value

So in the general case of n-dimensional data, the n coordinates are specified in the first n columns accordingly followed by the field data.

The default argument is to use the equipapced (but potentially collapsed) coordinates which can be obtained from the command.

```
FieldConvert file.xml file.dat
```

In this case the pointdatatofld module shoudl be used without the `-noequispaced` option. However this can lead to problems when peforming an elemental forward projection/transform since the mass matrix in a deformed element can be singular as the equispaced points do not have a sufficiently accurate quadrature rule that spans the polynomial space. Therefore it is adviseable to use the set of points given by

```
FieldConvert --noequispaced file.xml file.dat
```

which produces a set of points at the gaussian collapsed coordinates. In this case one must also use the `-noequispaced` option when projecting to a field.

Finally the option `setnantovalue=0` can also be used which sets any nan values in the interpolation to zero or any specified value in this option.

5.3.18 Print L2 and LInf norms: *printfldnorms* module

```
FieldConvert -m printfldnorms test.xml test.fld out.stdout
```

This module does not create an output file which is reinforced by the out.stdout option. The L2 and LInf norms for each field variable are then printed to the stdout.

5.3.19 Computes the scalar gradient: *scalargrad* module

The scalar gradient of a field is computed by running:

```
FieldConvert -m scalargrad:bnd=0 test.xml test.fld test-scalgrad.fld
```

The option `bnd` specifies which boundary region to extract. Note this is different to NekMesh where the parameter `surf` is specified and corresponds to composites rather boundaries. If `bnd` is not provided, all boundaries are extracted to different fields. To process this file you will need an xml file of the same region.

5.3.20 Scale a given .fld: *scaleinputfld* module

To scale a .fld file by a given scalar quantity, the user can run:

```
FieldConvert -m scaleinputfld:scale=value test.xml test.fld test-scal.fld
```

The argument `scale=value` rescales of a factor `value` `test.fld` by the factor value. The output file `file-conc.fld` can be processed in a similar way as described in section 5.1 to visualise the result either in Tecplot, Paraview or VisIt.

5.3.21 Time-averaged shear stress metrics: *shear* module

Time-dependent wall shear stress derived metrics relevant to cardiovascular fluid dynamics research can be computed using this module. They are

- TAWSS: time-averaged wall shear stress;
- OSI: oscillatory shear index;
- transWSS: transverse wall shear stress;
- TACFI: time-averaged cross-flow index;
- TAAFI: time-averaged aneurysm formation index;
- |WSSG|: wall shear stress gradient.

To compute these, the user can run:

```
FieldConvert -m shear:N=value:fromfld=test_id_b0.fld test.xml test-multishear.fld
```

The argument `N` and `fromfld` are compulsory arguments that respectively define the number of `fld` files corresponding to the number of discrete equispaced time-steps, and the first `fld` file which should have the form of `test_id_b0.fld` where the first underscore in the name marks the starting time-step file ID.

The input `.fld` files are the outputs of the *wss* module. If they do not contain the surface normals (an optional output of the *wss* modle), then the *shear* module will not compute the last metric, |WSSG|.

5.3.22 Boundary layer height calculation: *surfdistance* module

The surface distance module computes the height of a prismatic boundary layer and projects this value onto the surface of the boundary, in a similar fashion to the `extract` module. In conjunction with a mesh of the surface, which can be obtained with `NekMesh`, and a value of the average wall shear stress, one potential application of this module is to determine the distribution of y^+ grid spacings for turbulence calculations.

To compute the height of the prismatic layer connected to boundary region 3, the user can issue the command:

```
FieldConvert -m surfdistance:bnd=3 input.xml output.fld
```

Note that no `.fld` file is required, since the mesh is the only input required in order to calculate the element height. This produces a file `output_b3.fld`, which can be visualised with the appropriate surface mesh from `NekMesh`.

5.3.23 Calculate vorticity: *vorticity* module

To perform the vorticity calculation and obtain an output data containing the vorticity solution, the user can run

```
FieldConvert -m vorticity test.xml test.fld test-vort.fld
```

where the file `test-vort.fld` can be processed in a similar way as described in section 5.1.

5.3.24 Computing the wall shear stress: *wss* module

To obtain the wall shear stres vector and magnitude, the user can run:

```
FieldConvert -m wss:bnd=0:addnormals=1 test.xml test.fld test-wss.fld
```

The option `bnd` specifies which boundary region to extract. Note this is different to NekMesh where the parameter `surf` is specified and corresponds to composites rather boundaries. If `bnd` is not provided, all boundaries are extracted to different fields. The `addnormals` is an optional command argument which, when turned on, outputs the normal vector of the extracted boundary region as well as the shear stress vector and magnitude. This option is off by default. To process the output file(s) you will need an xml file of the same region.

5.3.25 Manipulating meshes with FieldConvert

FieldConvert has support for two modules that can be used in conjunction with the linear elastic solver, as shown in chapter 11. To do this, FieldConvert has an XML output module, in addition to the Tecplot and VTK formats.

The `deform` module, which takes no options, takes a displacement field and applies it to the geometry, producing a deformed mesh:

```
FieldConvert -m deform input.xml input.fld deformed.xml
```

The `displacement` module is designed to create a boundary condition field file. Its intended use is for mesh generation purposes. It can be used to calculate the displacement between the linear mesh and a high-order surface, and then produce a `fld` file, prescribing the displacement at the boundary, that can be used in the linear elasticity solver.

Presently the process is somewhat convoluted and must be used in conjunction with NekMesh to create the surface file. However the bash input below describes the procedure. Assume the high-order mesh is in a file called `mesh.xml`, the linear mesh is `mesh-linear.xml` that can be generated by removing the `CURVED` section from `mesh.xml`, and that we are interested in the surface with ID 123.

```
# Extract high order surface
NekMesh -m extract:surf=123 mesh.xml mesh-surf-curved.xml

# Use FieldConvert to calculate displacement between two surfaces
FieldConvert -m displacement:id=123:to=mesh-surf-curved.xml \
    mesh-linear.xml mesh-deformation.fld

# mesh-deformation.fld is used as a boundary condition inside the
# solver to prescribe the deformation conditions.xml contains
# appropriate Nektar++ parameters (mu, E, other BCs, ...)
LinearElasticSolver mesh-linear.xml conditions.xml

# This produces the final field mesh-linear.fld which is the
# displacement field, use FieldConvert to apply it:
FieldConvert-g -m deform mesh-linear.xml mesh-linear.fld mesh-deformed.xml
```

5.4 FieldConvert in parallel

To run FieldConvert in parallel the user needs to compile *Nektar++* with MPI support and can employ the following command

```
mpirun -np <nprocs> FieldConvert test.xml test.fld test.dat
```

or

```
mpirun -np <nprocs> FieldConvert test.xml test.fld test.vtu
```

replacing `<nprocs>` with the number of processors. This will produce multiple `.dat` or `.vtu` files of the form `test_P0.dat`, `test_P1.dat`, `test_P2.dat` or `test_P0.vtu`, `test_P1.vtu`, `test_P2.vtu`. Note when producing a parallel `.vtu` file an additional file called `.pvtu` is written out which allows for parallel reading of the individual `.vtu` files. Similarly functions that produce a `.fld` file output can be processed in this manner. In the case when producing a .fld file a directory called `test.fld` (or the specified output name) will be produced with the standard parallel field files placed within the directory.

5.5 Processing large files in serial

When processing large files, it is not always convenient to run in parallel but process each parallel partition in serial, for example when interpolating a solution field from one

mesh to another.

5.5.1 Using the nprocs and procid options

One option is to use the `-nprocs` and `-procid` command line options. For example, the following command will interpolate partition 2 of a decomposition into 10 partitions of `fiile2.xml` from `file1.fld`

```
FieldConvert --nprocs 10 --procid 2 \
        -m interpfield:fromxml=file1.xml:fromfld=file1.fld \
        file2.xml file2.fld
```

This call will only therefore consider the interpolation process across one partition (namely, partition 2). To create the full interpolated field requires a loop over each of the partitions, which, in a bash shell can be run as

```
for n in ‘seq 0 9‘; do
    FieldConvert --nprocs 10 --procid $n \
            -m interpfield:fromxml=file1.xml:fromfld=file1.fld \
            file2.xml file2.fld
done
```

The resulting output will lie in a directory called `file2.fld`, with each of the different parallel partitions in files with names `P0000000.fld`, `P0000001.fld`, . . . , `P0000009.fld`. This is nearly a complete parallel field file. However, the `Info.xml` file, which contains the information about which elements lie in each partition, is missing. This can be generated by using the command

```
FieldConvert --nprocs 10 file2.xml file2.fld/Info.xml:info
```

Note the final `:info` extension on the last argument is necessary to tell FieldConvert that you wish to generate an info file, but with the extension `.xml`. This syntax allows the routine not to get confused with the input/output XML files.

5.5.2 Using the –part-only and –part-only-overlapping options

Another approach to serially proessing a large file is to initially process the file into multiple partitions. This can be done with the `-part-only` option. So the command

```
FieldConvert --part-only 10 file.xml file.fld
```

will partition the mesh into 10 paritions and write each partition into a directory called `file_xml`. If you enter this directory you will find partitioned XML files `P0000000.xml`, `P0000001.xml`, . . . , `P0000009.xml` which can then be processed individually as outlined above.

There is also a `-part-only-overlapping` option, which can be run in the same fashion.

```
FieldConvert --part-only-overlapping 10 file.xml file.fld
```

In this mode, the mesh is partitioned into 10 partitions in a similar manner, but the elements at the partition edges will now overlap, so that the intersection of each partition with its neighbours is non-empty. This is sometime helpful when, for example, producing a global isocontour which has been smoothed. Applying the smoothed isocontour extraction routine with the `-part-only` option will produce a series of isocontour where there will be a gap between partitions, as the smoother tends to shrink the isocontour within a partition. using the `-part-only-overlapping` option will still yield a shrinking isocontour, but the overlapping partitions help to overlap the partiiton boundaries.

Part III

Solver Applications

Chapter 6

Advection-Diffusion-Reaction Solver

6.1 Synopsis

The ADRSolver is designed to solve partial differential equations of the form:

$$\alpha\frac{\partial u}{\partial t} + \lambda u + \nu\nabla u + \epsilon\nabla\cdot(D\nabla u) = f \tag{6.1}$$

in either discontinuous or continuous projections of the solution field. For a full list of the equations which are supported, and the capabilities of each equation, see the table below.

Equation to solve	EquationType	Dimensions	Projections
$\nabla^2 u = 0$	`Laplace`	All	Continuous/Discontinuous
$\nabla^2 u = f$	`Poisson`	All	Continuous/Discontinuous
$\nabla^2 u + \lambda u = f$	`Helmholtz`	All	Continuous/Discontinuous
$\epsilon\nabla^2 u + \mathbf{V}\nabla u = f$	`SteadyAdvectionDiffusion`	2D only	Continuous/Discontinuous
$\epsilon\nabla^2 u + \lambda u = f$	`SteadyDiffusionReaction`	2D only	Continuous/Discontinuous
$\epsilon\nabla^2 u + \mathbf{V}\nabla u + \lambda u = f$	`SteadyAdvectionDiffusionReaction`	2D only	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + \mathbf{V}\nabla u = f$	`UnsteadyAdvection`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} = \epsilon\nabla^2 u$	`UnsteadyDiffusion`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + \mathbf{V}\nabla u = \epsilon\nabla^2 u$	`UnsteadyAdvectionDiffusion`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + u\nabla u = 0$	`UnsteadyInviscidBurger`	1D only	Continuous/Discontinuous

Table 6.1 Equations supported by the ADRSolver with their capabilities.

6.2 Usage

```
ADRSolver session.xml
```

6.3 Session file configuration

The type of equation which is to be solved is specified through the EquationType SOLVERINFO option in the session file. This can be set as in table 6.1. At present, the Steady non-symmetric solvers cannot be used in parallel.

6.3.1 Solver Info

The solver info are listed below:

- **Eqtype**: This sets the type of equation to solve, according to the table above.
- **TimeIntegrationMethod**: The following types of time integration methods have been tested with each solver:

EqType	Explicit	Diagonally Implicit	IMEX	Implicit
`UnsteadyAdvection`	✓			
`UnsteadyDiffusion`	✓	✓		
`UnsteadyAdvectionDiffusion`			✓	
`UnsteadyInviscidBurger`	✓			

- **Projection**: The Galerkin projection used may be either:
  - `Continuous` for a C0-continuous Galerkin (CG) projection.
  - `Discontinuous` for a discontinous Galerkin (DG) projection.
- **DiffusionAdvancement**: This specifies how to treat the diffusion term. This will be restricted by the choice of time integration scheme:
  - `Explicit` Requires the use of an explicit time integration scheme.
  - `Implicit` Requires the use of a diagonally implicit, IMEX or Implicit scheme.
- **AdvectionAdvancement**: This specifies how to treat the advection term. This will be restricted by the choice of time integration scheme:
  - `Explicit` Requires the use of an explicit or IMEX time integration scheme.
  - `Implicit` Not supported at present.
- **AdvectionType**: Specifies the type of advection:

  - `NonConservative` (for CG only).
  - `WeakDG` (for DG only).

- **DiffusionType**:
  - `LDG`.
- **UpwindType**:
  - `Upwind`.

6.3.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `epsilon`: sets the diffusion coefficient ϵ.
  Can be used in: SteadyDiffusionReaction, SteadyAdvectionDiffusionReaction, UnsteadyDiffusion, UnsteadyAdvectionDiffusion.
  Default value: 0.
- `d00`, `d11`, `d22`: sets the diagonal entries of the diffusion tensor D.
  Can be used in: UnsteadyDiffusion
  Default value: All set to 1 (i.e. identity matrix).
- `lambda`: sets the reaction coefficient λ.
  Can be used in: SteadyDiffusionReaction, Helmholtz, SteadyAdvectionDiffusionReaction
  Default value: 0.

6.3.3 Functions

The following functions can be specified inside the `CONDITIONS` section of the session file:

- `AdvectionVelocity`: specifies the advection velocity $\mathbf{V}$.
- `InitialConditions`: specifies the initial condition for unsteady problems.
- `Forcing`: specifies the forcing function f.

6.4 Examples

Example files for the ADRSolver are provided in `solvers/ADRSolver/Examples`

6.4.1 1D Advection equation

In this example, it will be demonstrated how the Advection equation can be solved on a one-dimensional domain.

6.4.1.1 Advection equation

We consider the hyperbolic partial differential equation:

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0, \tag{6.2}$$

where $f = au$ is the advection flux.

6.4.1.2 Input file

The input for this example is given in the example file `Advection1D.xml`

The geometry section defines a 1D domain consisting of 10 segments. On each segment an expansion consisting of 4 Lagrange polynomials on the Gauss-Lobotto-Legendre points is used as specified by

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" FIELDS="u" TYPE="GLL_LAGRANGE_SEM" NUMMODES="4"/>
</EXPANSIONS>
```

Since we are solving the unsteady advection problem, we must specify this in the solver information. We also choose to use a discontinuous flux-reconstruction projection and use a Runge-Kutta order 4 time-integration scheme.

```
<I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvection"    />
<I PROPERTY="Projection"            VALUE="DisContinuous"        />
<I PROPERTY="AdvectionType"         VALUE="FRDG"                 />
<I PROPERTY="UpwindType"            VALUE="Upwind"               />
<I PROPERTY="TimeIntegrationMethod" VALUE="ClassicalRungeKutta4"/>
```

We choose to advect our solution for 20 time units with a time-step of 0.01 and so provide the following parameters

```
<P> FinTime         = 20                </P>
<P> TimeStep        = 0.01              </P>
<P> NumSteps        = FinTime/TimeStep  </P>
```

We also specify the advection velocity. We first define dummy parameters

```
<P> advx            = 1                  </P>
<P> advy            = 0                  </P>
```

and then define the actual advection function as

```
<FUNCTION NAME="AdvectionVelocity">
    <E VAR="Vx" VALUE="advx" />
</FUNCTION>
```

Two boundary regions are defined, one at each end of the domain, and periodicity is enforced

```
<BOUNDARYREGIONS>
    <B ID="0"> C[1] </B>
    <B ID="1"> C[2] </B>
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <P VAR="u" VALUE="[1]" />
    </REGION>
    <REGION REF="1">
        <P VAR="u" VALUE="[0]" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Finally, we specify the initial value of the solution on the domain

```
<FUNCTION NAME="InitialConditions">
    <E VAR="u" VALUE="exp(-20.0*x*x)" />
</FUNCTION>

<FUNCTION NAME="ExactSolution">
    <E VAR="u" VALUE="exp(-20.0*x*x)" />
</FUNCTION>
```

6.4.1.3 Running the code

```
ADRSolver Advection1D.xml
```

To visualise the output, we can convert it into either TecPlot or VTK formats

```
FieldConvert Advection1D.xml Advection1D.fld Advection1D.dat
FieldConvert Advection1D.xml Advection1D.fld Advection1D.vtu
```

6.4.2 2D Helmholtz Problem

In this example, it will be demonstrated how the Helmholtz equation can be solved on a two-dimensional domain.

6.4.2.1 Helmholtz equation

We consider the elliptic partial differential equation:

$$\nabla^2 u + \lambda u = f \tag{6.3}$$

where ∇^2 is the Laplacian and λ is a real positive constant.

6.4.2.2 Input file

The input for this example is given in the example file `Helmholtz2D_modal.xml`

The geometry for this problem is a two-dimensional octagonal plane containing both triangles and quadrilaterals. Note that a mesh composite may only contain one type of element. Therefore, we define two composites for the domain, while the rest are used for enforcing boundary conditions.

```
<COMPOSITE>
    <C ID="0"> Q[22-47] </C>
    <C ID="1"> T[0-21] </C>
    <C ID="2"> E[0-1] </C>
    .
    .
    <C ID="10"> E[84,75,69,62,51,40,30,20,6] </C>
</COMPOSITE>

<DOMAIN> C[0-1] </DOMAIN>
```

For both the triangular and quadrilateral elements, we use the modified Legendre basis with 7 modes (maximum polynomial order is 6).

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="7" FIELDS="u" TYPE="MODIFIED" />
    <E COMPOSITE="C[1]" NUMMODES="7" FIELDS="u" TYPE="MODIFIED" />
</EXPANSIONS>
```

Only one parameter is needed for this problem. In this example $\lambda = 1$ and the Continuous Galerkin Method is used as projection scheme to solve the Helmholtz equation, so we need to specify the following parameters and solver information.

```
<PARAMETERS>
    <P> Lambda = 1 </P>
</PARAMETERS>

<SOLVERINFO>
    <I PROPERTY="EQTYPE" VALUE="Helmholtz" />
    <I PROPERTY="Projection" VALUE="Continuous" />
</SOLVERINFO>
```

All three basic boundary condition types have been used in this example: Dirichlet, Neumann and Robin boundary. The boundary regions are defined, each of which corresponds to one of the edge composites defined earlier. Each boundary region is then assigned an appropriate boundary condition.

```
<BOUNDARYREGIONS>
    <B ID="0"> C[2] </B>
    .
    .
    <B ID="8"> C[10] </B>
</BOUNDARYREGIONS>

```

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="sin(PI*x)*sin(PI*y)" />
    </REGION>
    <REGION REF="1">
        <R VAR="u" VALUE="sin(PI*x)*sin(PI*y)-PI*sin(PI*x)*cos(PI*y)"
           PRIMCOEFF="1" />
    </REGION>
    <REGION REF="2">
        <N VAR="u" VALUE="(5/sqrt(61))*PI*cos(PI*x)*sin(PI*y)-
                          (6/sqrt(61))*PI*sin(PI*x)*cos(PI*y)" />
    </REGION>
        .
        .
</BOUNDARYCONDITIONS>
```

We know that for $f = -(\lambda+2\pi^2)sin(\pi x)cos(\pi y)$, the exact solution of the two-dimensional Helmholtz equation is $u = sin(\pi x)cos(\pi y)$. These functions are defined specified to initialise the problem and verify the correct solution is obtained by evaluating the L_2 and L_{inf} errors.

```
<FUNCTION NAME="Forcing">
    <E VAR="u" VALUE="-(Lambda + 2*PI*PI)*sin(PI*x)*sin(PI*y)" />
</FUNCTION>

<FUNCTION NAME="ExactSolution">
    <E VAR="u" VALUE="sin(PI*x)*sin(PI*y)" />
</FUNCTION>
```

6.4.2.3 Running the code

```
ADRSolver Test_Helmholtz2D_modal.xml
```

This execution should print out a summary of input file, the L_2 and L_{inf} errors and the time spent on the calculation.

6.4.2.4 Post-processing

Simulation results are written in the file Helmholtz2D_modal.fld. We can choose to visualise the output in Gmsh

```
FieldConvert Helmholtz2D_modal.xml Helmholtz2D_modal.fld Helmholtz2D_modal.vtu
```

which generates the file Helmholtz2D_modal.vtu which can be visualised and is shown in Fig. 6.1

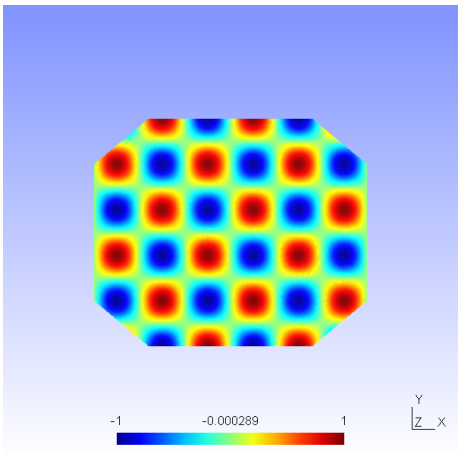


Figure 6.1 Solution of the 2D Helmholtz Problem.

6.4.3 Advection dominated mass transport in a pipe

The following example demonstrates the application of the ADRsolver for modelling advection dominated mass transport in a straight pipe. Such a transport regime is encountered frequently when modelling mass transport in arteries. This is because the diffusion coefficient of small blood borne molecules, for example oxygen or adenosine triphosphate, is very small $O(10^{-10})$.

6.4.3.1 Background

The governing equation for modelling mass transport is the unsteady advection diffusion equation:

$$\frac{\partial u}{\partial t} + v\nabla u + \epsilon\nabla^2 u = 0$$

For small diffusion coefficient, ϵ, the transport is dominated by advection and this leads to a very fine boundary layer adjacent to the surface which must be captured in order to get a realistic representation of the wall mass transfer processes. This creates problems not only from a meshing perspective, but also numerically where classical oscillations are observed in the solution due to under-resolution of the boundary layer.

The Graetz-Nusselt solution is an analytical solution of a developing mass (or heat) transfer boundary layer in a pipe. Previously this solution has been used as a benchmark for the accuracy of numerical methods to capture the fine boundary layer which develops for high Peclet number transport (the ratio of advection to diffusion). The solution is derived based on the assumption that the velocity field within the mass transfer boundary layer is linear i.e. the Schmidt number (the relative thickness of the momentum to mass transfer boundary layer) is sufficiently large. The analytical solution for the

non-dimensional mass transfer at the wall is given by:

$$Sh(z) = \frac{2^{4/3}(PeR/z)^{1/3}}{g^{1/3}\Gamma(4/3)},$$

where z is the streamwise coordinate, R the pipe radius, $\Gamma(4/3)$ an incomplete Gamma function and Pe the Peclet number given by:

$$Pe = \frac{2UR}{\epsilon}$$

In the following we will numerically solver mass transport in a pipe and compare the calculated mass transfer at the wall with the Graetz-Nusselt solution. The Peclet number of the transport regime under consideration is 750000, which is physiologically relevant.

6.4.3.2 Input file

The geometry under consideration is a pipe of radius, $R = 0.5$ and length $l = 0.5$

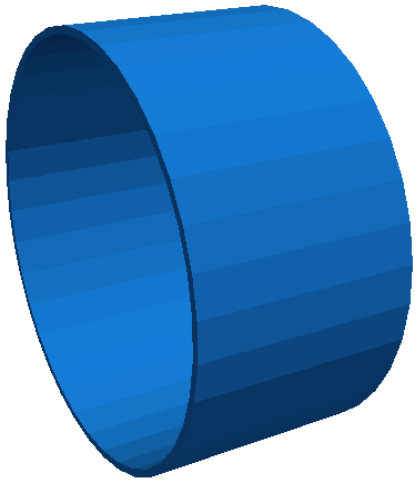


Figure 6.2 Pipe.

Since the mass transport boundary layer will be confined to a very small layer adjacent to the wall we do not need to mesh the interior region, hence the mesh consists of a layer of ten prismatic elements over a thickness of 0.036R. The elements progressively grow over the thickness of domain.

In this example we utilise heterogeneous polynomial order, in which the polynomial order normal to the wall is higher so that we avoid unphysical oscillations, and hence the incorrect solution, in the mass transport boundary layer. To do this we specify explicitly the expansion type, points type and distribution in each direction as follows:

```
1 <EXPANSIONS>
```

```
  <E COMPOSITE="C[0]"
     NUMMODES="3,5,3"
     BASISTYPE="Modified_A,Modified_A,Modified_B"
     NUMPOINTS="4,6,3"
     POINTSTYPE="GaussLobattoLegendre,GaussLobattoLegendre,GaussRadauMAlpha1Beta0"
     FIELDS="u" />
</EXPANSIONS>
```

The above represents a quadratic polynomial order in the azimuthal and streamwise direction and 4th order polynomial normal to the wall for a prismatic element.

We choose to use a continuous projection and an first-order implicit-explicit time-integration scheme. The `DiffusionAdvancement` and `AdvectionAdvancement` parameters specify how these terms are treated.

```
<I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvectionDiffusion" />
<I PROPERTY="Projection"            VALUE="Continuous" />
<I PROPERTY="DiffusionAdvancement"  VALUE="Implicit" />
<I PROPERTY="AdvectionAdvancement"  VALUE="Explicit" />
<I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder1" />
<I PROPERTY="GlobalSysSoln"         VALUE="IterativeStaticCond" />
```

We integrate for a total of 30 time units with a time-step of 0.0005, necessary to keep the simulation numerically stable.

```
<P> TimeStep = 0.0005              </P>
<P> FinalTime = 30                 </P>
<P> NumSteps = FinalTime/TimeStep  </P>
```

The value of the ϵ parameter is $\epsilon = 1/Pe$

```
<P> epsilon = 1.33333e-6           </P>
```

The analytical solution represents a developing mass transfer boundary layer in a pipe. In order to reproduce this numerically we assume that the inlet concentration is a uniform value and the outer wall concentration is zero; this will lead to the development of the mass transport boundary layer along the length of the pipe. Since we do not model explicitly the mass transfer in the interior region of the pipe we assume that the inner wall surface concentration is the same as the inlet concentration; this assumption is valid based on the large Peclet number meaning the concentration boundary layer is confined to the region in the immediate vicinity of the wall. The boundary conditions are specified as follows in the input file:

```
<BOUNDARYREGIONS>
   <B ID="0"> C[3] </B>  <!-- inlet -->
   <B ID="1"> C[4] </B>  <!-- outlet -->
   <B ID="2"> C[2] </B>  <!-- outer surface -->
   <B ID="3"> C[5] </B>  <!-- inner surface -->
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
```

```
  <REGION REF="0">
    <D VAR="u" VALUE="1" />
  </REGION>
  <REGION REF="1">
    <N VAR="u" VALUE="0" />
  </REGION>
  <REGION REF="2">
    <D VAR="u" VALUE="0" />
  </REGION>
  <REGION REF="3">
    <D VAR="u" VALUE="1" />
  </REGION>
</BOUNDARYCONDITIONS>
```

The velocity field within the domain is fully devqeloped pipe flow (Poiseuille flow), hence we can define this through an analytical function as follows:

```
<FUNCTION NAME="AdvectionVelocity">
  <E VAR="Vx" VALUE="0" />
  <E VAR="Vy" VALUE="0" />
  <E VAR="Vz" VALUE="2.0*(1-(x*x+y*y)/0.25)" />
</FUNCTION>
```

We assume that the initial domain concentration is uniform everywhere and the same as the inlet. This is defined by,

```
<FUNCTION NAME="InitialConditions">
  <E VAR="u" VALUE="1" />
</FUNCTION>
```

6.4.3.3 Results

To compare with the analytical expression we numerically calculate the concentration gradient at the surface of the pipe. This is then plotted against the analytical solution by extracting the solution along a line in the streamwise direction, as shown in Fig. 6.3.

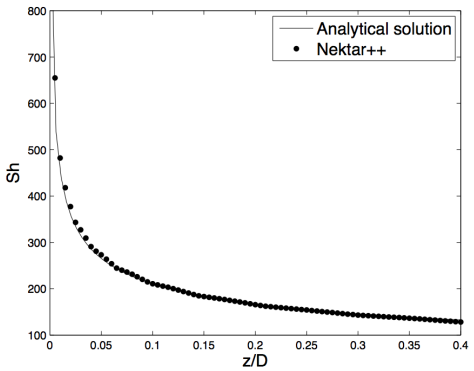


Figure 6.3 Concentration gradient at the surface of the pipe.

Chapter 7

Acoustic Perturbation Equations Solver

7.1 Synopsis

The aim of APESolver is to predict aerodynamic sound generation. Through the application of a splitting technique, the flow-induced acoustic field is totally decoupled from the underlying incompressible hydrodynamic field. The acoustic perturbation equations (APE-1/APE-4) proposed by Ewert and Shroeder are employed as the governing equations of the acoustic field and they assure stable aeroacoustic simulation due to the suppression of the term related to the production of perturbed vorticity. These equations are similar to the linearised perturbed compressible equations, but while in the original formulation the flow decomposition is based on solenoidal vortical perturbations as well as irrotational acoustic perturbations, in this case perturbations are assumed to be exclusively of acoustic nature.

$$\frac{\partial p'}{\partial t} + \overline{c}^2 \frac{\partial \overline{\rho} u_i'}{\partial x_i} + \overline{c}^2 \frac{\partial \overline{u}_i p'/\overline{c}^2}{\partial x_i} = \overline{c}^2 q_c$$
$$\frac{\partial u_i'}{\partial t} + \frac{\partial \overline{u}_j u_j'}{\partial x_i} + \frac{\partial p'/\overline{\rho}}{\partial x_i} = 0$$

where $(\overline{u}_i, \overline{p}, \overline{\rho}, \overline{c}^2 = \gamma \overline{p}/\overline{\rho})$ represents the base flow and (u_i', p') the perturbations. $\overline{c}^2 q_c$ is the acoustic source term.

7.2 Usage

```
APESolver session.xml
```

7.3 Session file configuration

7.3.1 Solver Info

- `Eqtype` Specifies the equation to solve. This should be set to `APE`.
- `UpwindType` Specifies the numerical interface flux scheme. Currently, only `APEUpwind` supported.

7.3.2 Parameters

- `Gamma`: Ratio of specific heats

7.3.3 Functions

- `BaseFlow` Baseflow $(\overline{u}_i, \overline{p}, \overline{\rho})$ defined by the variables `u0,v0,w0,p0,rho0`
- `Source` Source term $\overline{c}^2 q_c$
- `InitialConditions`

7.4 Examples

7.4.1 Aeroacoustic Wave Propagation

In this section we explain how to set up a simple simulation of aeroacoustics in Nektar++. We will study the propagation of an acoustic wave in the simple case where the base flow is $\overline{u}_i = 0$, $\overline{p} = p_\infty = 10^6$, $\overline{\rho} = \rho_0 = 1.204$. The geometry consists of 64 quadrilateral elements, as shown in Fig. 7.1.

7.4.1.1 Input file

We require a discontinuous Galerkin projection and use an explicit fourth-order Runge-Kutta time integration scheme. We therefore set the following solver information:

```
<I PROPERTY="EQType" VALUE="APE"/>
<I PROPERTY="Projection" VALUE="DisContinuous"/>
<I PROPERTY="TimeIntegrationMethod"  VALUE="ClassicalRungeKutta4"/>
<I PROPERTY="UpwindType"  VALUE="APEUpwind"/>
```

To maintain numerical stability we must use a small time-step. The total simulation time is 150 time units. Finally, we set the density, heat ratio and ambient pressure.

```
<P> TimeStep       = 0.00001            </P>
<P> NumSteps       = 150                </P>
<P> FinTime        = TimeStep*NumSteps  </P>
<P> Rho0           = 1.204              </P> <!-- Incompressible density -->
<P> Gamma          = 1.4                </P> <!-- Ratio of specific heats -->
<P> Pinfinity      = 100000             </P> <!-- Ambient pressure -->
```

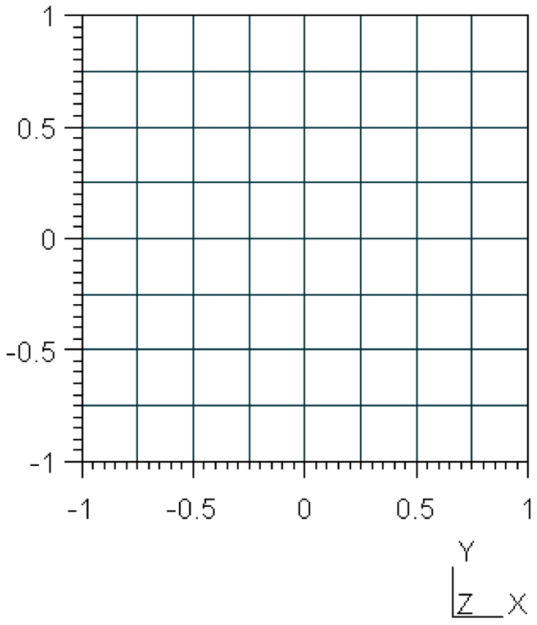


Figure 7.1 Geometry used for the example case of modelling propagation of acoustic waves where $\overline{u}_i = 0$, $\overline{p} = p_\infty = 10^6$, $\overline{\rho} = \rho_0 = 1.204$

Let us note that to solve efficiently this problem a discontinuous Garlerkin approach was used. The system is excited via the initial conditions putting a Gaussian pulse for pulse fluctuations. Finally, it is necessary to specify the base flow and the eventual source terms using the following functions:

```
<FUNCTION NAME="Baseflow">
    <E VAR="u0"   VALUE="0" />
    <E VAR="v0"   VALUE="0" />
    <E VAR="p0"   VALUE="Pinfinity" />
    <E VAR="rho0" VALUE="Rho0" />
</FUNCTION>

<FUNCTION NAME="Source">
    <E VAR="S"  VALUE="0" />
</FUNCTION>

<!-- Gaussian pulse located at the origin -->
<FUNCTION NAME="InitialConditions">
    <E VAR="p" VALUE="100*exp(-32*(((x)*(x))+((y)*(y))))" />
    <E VAR="u" VALUE="0" />
    <E VAR="v" VALUE="0" />
</FUNCTION>
```

7.4.1.2 Running the code

```
APESolver Test_pulse.xml
```

7.4.1.3 Results

Fig. 7.2 shows the pressure profile at different time steps, showing the acoustic propagation.

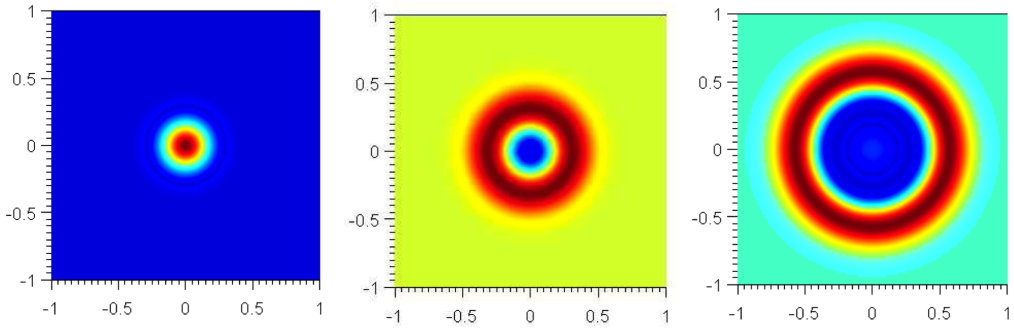


Figure 7.2

It is possible to show the profile of the pressure perturbations with respect to the spatial coordinate. The pressure fluctuations, that are concentrated in a specific locations at the beginning (as specified by the initial conditions), propagate with time and for sufficiently large time the decay is exponential as predicted by literature [10].

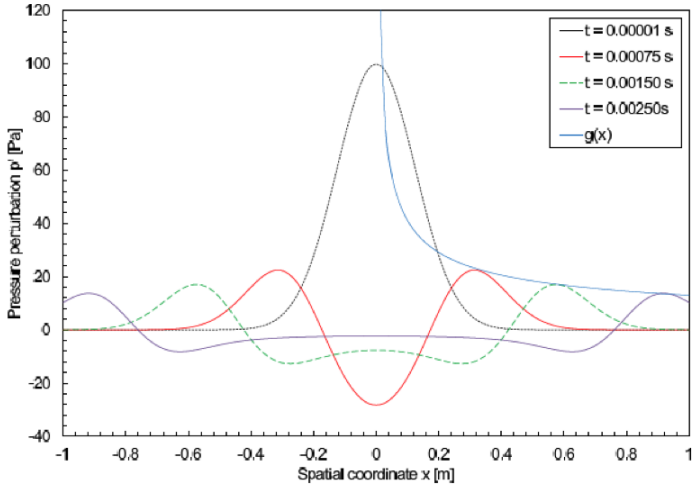


Figure 7.3

Chapter 8

Cardiac Electrophysiology Solver

8.1 Synopsis

The CardiacEPSolver is used to model the electrophysiology of cardiac tissue, specifically using the monodomain or bidomain model. These models are continuum models and represent an average of the electrical activity over many cells. The system is a reaction-diffusion system, with the reaction term modeling the flow of current in and out of the cells using a separate set of ODEs.

8.1.1 Bidomain Model

The Bidomain model is given by the following PDEs,

$$g_{ix}\frac{\partial^2 V_i}{\partial x^2} + g_{iy}\frac{\partial^2 V_i}{\partial y^2} = \chi\left[C_m\frac{\partial(V_i - V_e)}{\partial t} + G_m(V_i - V_e)\right]$$
$$g_{ex}\frac{\partial^2 V_e}{\partial x^2} + g_{ey}\frac{\partial^2 V_e}{\partial y^2} = -\chi\left[C_m\frac{\partial(V_i - V_e)}{\partial t} + G_m(V_i - V_e)\right].$$

However, when solving numerically, one often rewrites these equations in terms of the transmembrane potential and extracellular potential,

$$\chi\left[C_m\frac{\partial V_m}{\partial t} + J_{ion}\right] = g_{ex}\frac{\partial^2 V_e}{\partial x^2} + g_{ey}\frac{\partial^2 V_e}{\partial y^2}$$
$$(g_{ix} + g_{ex})\frac{\partial^2 V_e}{\partial x^2} + (g_{iy} + g_{ey})\frac{\partial^2 V_e}{\partial y^2} = -g_{ix}\frac{\partial^2 V_m}{\partial x^2} - g_{iy}\frac{\partial^2 V_m}{\partial y^2}$$

8.1.2 Monodomain Model

In the case where the intracellular and extracellular conductivities are proportional, that is $g_{ix} = kg_{ex}$ for some k, then the above two PDEs can be reduced to a single PDE:

$$\chi\left[C_m\frac{\partial V_m}{\partial t} + J_{ion}\right] = \nabla\cdot(\sigma\nabla V_m)$$

8.1.3 Cell Models

The action potential of a cardiac cell can be modelled at either a biophysical level of detail, including a number of transmembrane currents, or as a phenomenological model, to reproduce the features of the action potential, with fewer variables. Each cell model will include a unique system of ODEs to represent the gating variables of that model.

A number of ionic cell models are currently supported by the solver including:

- Courtemanche, Ramirez, Nattel, 1998
- Luo, Rudy, 1991
- ten Tusscher, Panfilov, 2006 (epicardial, endocardial and mid-myocardial variants)

Phenomological cell models are also supported:

- Aliev-Panfilov
- Fitzhugh-Nagumo

It is important to ensure that the units of the voltage and currents from the cell model are consistent with the units expected by the tissue level solver (monodomain/bidomain). We will show as an example the Courtemanche, Ramirez, Nattel, 1998 human atrial model.

The monodomain equation:

$$\chi \left[C_m \frac{\partial V_m}{\partial t} + J_{ion} \right] = \nabla \cdot (\sigma \nabla V_m)$$

8.2 Usage

```
CardiacEPSolver session.xml
```

8.3 Session file configuration

8.3.1 Solver Info

- `Eqtype` Specifies the PDE system to solve. The following values are supported:
  - `Monodomain`: solve the monodomain equation.
  - `BidomainRoth`: solve the bidomain equations using the Roth formulation.
- `CellModel` Specifies the cell model to use. Available cell models are

Value	Description	No. of Var.	Ref.
`AlievPanfilov`	Phenomological	1	[2]
`CourtemancheRamirezNattel98`	Human atrial	20	[19]
`FitzHughNagumo`			
`Fox02`			
`LuoRudy91`	Mammalian ventricular	7	[18]
`PanditGilesDemir03`			
`TenTusscher06`	Human ventricular	18	[32]
`Winslow99`			

- `Projection` Specifies the Galerkin projection type to use. Only `Continuous` has been extensively tested.

- `TimeIntegrationMethod` Specifies the time integration scheme to use for advancing the PDE system. This must be an IMEX scheme. Suitable choices are: `IMEXOrder1`, `IMEXOrder2`, `IMEXOrder3`, `IMEXdirk_3_4_3`. The cell model state variables are time advanced using Forward Euler for the ion concentrations, and Rush-Larsen for the cell model gating variables.

- `DiffusionAdvancement` Specifies whether the diffusion is handled implicitly or explicitly in the time integration scheme. The current code only supports `Implicit` integration of the diffusion term. The cell model is always integrated explicitly.

8.3.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file. Example values are taken from [11].

- `Chi` sets the surface-to-volume ratio (Units: mm^{-1}).
  Example: $\chi = 140\mathrm{mm}^{-1}$

- `Cm` sets the specific membrane capacitance (Units: $\mu\mathrm{F}\,\mathrm{mm}^{-2}$).
  Example: $C_m = 0.01\mu\mathrm{F}\,\mathrm{mm}^{-2}$

- `Substeps` sets the number of substeps taken in time integrating the cell model for each PDE timestep.
  Example: 4

- `d_min`, `d_max`, `o_min`, `o_max` specifies a bijective map to assign conductivity values σ to intensity values μ when using the `IsotropicConductivity` function. The intensity map is first thresholded to the range $[d_{\min}, d_{\max}]$ and then the conductivity is calculated as

$$\sigma = \frac{o_{\max} - o_{\min}}{d_{\max} - d_{\min}}(1 - \mu) + o_{\min}$$

8.3.3 Functions

The following functions can be specified inside the `CONDITIONS` section of the session file. If both are specified, the effect is multiplicative. Example values are taken from [11].

- `IsotropicConductivity` specifies the conductivity σ of the tissue.
  Example: $\sigma = 0.13341\,\mathrm{mS\,mm^{-1}}$, based on $\sigma = \frac{\sigma_i \sigma_e}{\sigma_i + \sigma_e}$, $\sigma_i = 0.17, \sigma_e = 0.62\mathrm{mS\,mm^{-1}}$

  The variable name to use is `intensity` since the conductivity may be derived from late-Gadolinium enhanced MRA imaging. Example specifications are

```
<E VAR="intensity" VALUE="0.13341" />
<F VAR="intensity" FILE="scarmap.con" />
```

  where `scarmap.con` is a Nektar++ field file containing a variable `intensity` describing the conductivity across the domain.

- `AnisotropicConductivity` specifies the conductivity σ of the tissue.

8.3.4 Filters

- `CellHistoryPoints` writes all cell model states over time at fixed points. Can be used along with the `HistoryPoints` filter to record all variables at specific points during a simulation.

```
<FILTER TYPE="CellHistoryPoints">
    <PARAM NAME="OutputFile">crn.his</PARAM>
    <PARAM NAME="OutputFrequency">1</PARAM>
    <PARAM NAME="Points">
        0.00 0.0 0.0
    </PARAM>
</FILTER>
```

  - `OutputFile` specifies the filename to write history data to.
  - `OutputFrequency` specifies the number of steps between successive outputs.
  - `Points` lists coordinates at which history data is to be recorded.

- `CheckpointCellModel` checkpoints the cell model. Can be used along with the `Checkpoint` filter to record complete simulation state and regular intervals.

```
<FILTER TYPE="CheckpointCellModel">
  <PARAM NAME="OutputFile"> session </PARAM>
  <PARAM NAME="OutputFrequency"> 1 </PARAM>
</FILTER>
```

  - `OutputFile` (optional) specifies the base filename to use. If not specified, the session name is used. Checkpoint files are suffixed with the process ID and the extension '.chk'.

- `OutputFrequency` specifies the number of timesteps between checkpoints.

- `Electrogram` Computes virtual unipolar electrograms at a prescribed set of points.

```
<FILTER TYPE="Electrogram">
  <PARAM NAME="OutputFile"> session </PARAM>
  <PARAM NAME="OutputFrequency"> 1 </PARAM>
  <PARAM NAME="Points">
      0.0  0.5  0.7
      1.0  0.5  0.7
      2.0  0.5  0.7
  </PARAM>
</FILTER>
```

  - `OutputFile` (optional) specifies the base filename to use. If not specified, the session name is used. The extension '.ecg' is appended if not already specified.
  - `OutputFrequency` specifies the number of resolution of the electrogram data.
  - `Points` specifies a list of coordinates at which electrograms are desired. *They must not lie within the domain.*

- `Benchmark` Records spatially distributed event times for activation and repolarisation (recovert) during a simulation, for undertaking benchmark test problems.

```
<FILTER TYPE="Benchmark">
  <PARAM NAME="ThresholdValue"> -40.0 </PARAM>
  <PARAM NAME="InitialValue">     0.0 </PARAM>
  <PARAM NAME="OutputFile"> benchmark </PARAM>
  <PARAM NAME="StartTime">        0.0 </PARAM>
</FILTER>
```

  - `ThresholdValue` specifies the value above which tissue is considered to be depolarised and below which is considered repolarised.
  - `InitialValue` specifies the initial value of the activation or repolarisation time map.
  - `OutputFile` specifies the base filename of activation and repolarisation maps output from the filter. This name is appended with the index of the event and the suffix '.fld'.
  - `StartTime` (optional) specifies the simulation time at which to start detecting events.

8.3.5 Stimuli

Electrophysiological propagaion is initiated through the stimulus current I_{ion}. The `STIMULI` section describes one or more regions of stimulus and the time-dependent protocol with which they are applied.

```
<STIMULI>
  ...
</STIMULI>
```

A number of stimulus types are available

8.3.5.1 Stimulus types

- `StimulusRect` stimulates a cuboid-shaped region of the domain, specified by two coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2). An additional parameter specifies the "smoothness" of the boundaries of the region; higher values produce a sharper boundary. Finally, the maximum strength of the stimulus current is specified in $\mu\text{A}/\text{mm}^3$

```
<STIMULUS TYPE="StimulusRect" ID="0">
  <p_x1> -15.24 </p_x1>
  <p_y1>  14.02 </p_y1>
  <p_z1>   6.87 </p_z1>
  <p_x2>  12.23 </p_x2>
  <p_y2>  16.56 </p_y2>
  <p_z2>   8.88 </p_z2>
  <p_is> 100.00 </p_is>
  <p_strength> 50.0 </p_strength>
</STIMULUS>
```

- `StimulusCirc` stimulates a spherical region of the domain, as specified by a centre and radius. The smoothness and strength parameters are also specified as for ‘StimulusRect‘.

```
<STIMULUS TYPE="StimulusCirc" ID="0">
  <p_x1> -15.24 </p_x1>
  <p_y1>  14.02 </p_y1>
  <p_z1>   6.87 </p_z1>
  <p_r1>  12.23 </p_r1>
  <p_is> 100.00 </p_is>
  <p_strength> 50.0 </p_strength>
</STIMULUS>
```

8.3.5.2 Protocols

A protocol specifies the time-dependent function indicating the strength of the stimulus and one such `PROTOCOL` section should be included within each `STIMULUS`. This can be expressed as one of:

- `ProtocolSingle` a single stimulus is applied at a given start time and for a given duration

```
<PROTOCOL TYPE="ProtocolSingle">
  <START>    0.0   </START>
  <DURATION> 2.0   </DURATION>
</PROTOCOL>
```

- `ProtocolS1` a train of pulses of fixed duration applied at a given start time and with a given cycle length.

```
<PROTOCOL TYPE="ProtocolS1">
  <START>    0.0   </START>
  <DURATION> 2.0   </DURATION>
  <S1CYCLELENGTH> 300.0 </S1CYCLELENGTH>
  <NUM_S1>   5     </NUM_S1>
</PROTOCOL>
```

- `ProtocolS1S2` same as 'ProtocolS1' except with an additional single pulse applied at a different cycle length at the end of the train of S1 pulses.

```
<PROTOCOL TYPE="ProtocolS1S2">
  <START>    0.0   </START>
  <DURATION> 2.0   </DURATION>
  <S1CYCLELENGTH> 300.0 </S1CYCLELENGTH>
  <NUM_S1>   5     </NUM_S1>
  <S2CYCLELENGTH> 100.0 </S2CYCLELENGTH>
</PROTOCOL>
```

Chapter 9

Compressible Flow Solver

9.1 Synopsis

The CompressibleFlowSolver allows us to solve the unsteady compressible Euler and Navier-Stokes equations for 1D/2D/3D problems using a discontinuous representation of the variables. In the following we describe both the compressible Euler and the Navier-Stokes equations.

9.1.1 Euler equations

The Euler equations can be expressed as a hyperbolic conservation law in the form

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}_i}{\partial x} + \frac{\partial \mathbf{g}_i}{\partial y} + \frac{\partial \mathbf{h}_i}{\partial z} = 0, \tag{9.1}$$

where $\mathbf{q}$ is the vector of the conserved variables, $\mathbf{f}_i = \mathbf{f}_i(\mathbf{q})$, $\mathbf{g}_i = \mathbf{g}_i(\mathbf{q})$ and $\mathbf{h}_i = \mathbf{h}_i(\mathbf{q})$ are the vectors of the inviscid fluxes

$$\mathbf{q} = \begin{Bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{Bmatrix}, \quad \mathbf{f}_i = \begin{Bmatrix} \rho u \\ p + \rho u^2 \\ \rho uv \\ \rho uw \\ u(E+p) \end{Bmatrix}, \quad \mathbf{g}_i = \begin{Bmatrix} \rho v \\ \rho uv \\ p + \rho v^2 \\ \rho vw \\ v(E+p) \end{Bmatrix}, \quad \mathbf{h}_i = \begin{Bmatrix} \rho w \\ \rho uw \\ \rho vw \\ p + \rho w^2 \\ w(E+p) \end{Bmatrix}, \tag{9.2}$$

where ρ is the density, u, v and w are the velocity components in x, y and z directions, p is the pressure and E is the total energy. In this work we considered a perfect gas law for which the pressure is related to the total energy by the following expression

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho(u^2 + v^2 + w^2), \tag{9.3}$$

where γ is the ratio of specific heats.

9.1.2 Compressible Navier-Stokes equations

The Navier-Stokes equations include the effects of fluid viscosity and heat conduction and are consequently composed by an inviscid and a viscous flux. They depend not only on the conserved variables but also, indirectly, on their gradient. The second order partial differential equations for the three-dimensional case can be written as:

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} + \frac{\partial \mathbf{g}}{\partial y} + \frac{\partial \mathbf{h}}{\partial z} = 0, \tag{9.4}$$

where $\mathbf{q}$ is the vector of the conserved variables, $\mathbf{f} = \mathbf{f}(\mathbf{q}, \nabla(\mathbf{q}))$, $\mathbf{g} = \mathbf{g}(\mathbf{q}, \nabla(\mathbf{q}))$ and $\mathbf{h} = \mathbf{h}(\mathbf{q}, \nabla(\mathbf{q}))$ are the vectors of the fluxes which can also be written as:

$$\mathbf{f} = \mathbf{f}_i - \mathbf{f}_v, \mathbf{g} = \mathbf{g}_i - \mathbf{g}_v, \mathbf{h} = \mathbf{h}_i - \mathbf{h}_v, \tag{9.5}$$

where $\mathbf{f}_i$, $\mathbf{g}_i$ and $\mathbf{h}_i$ are the inviscid fluxes of Eq. (9.2) and $\mathbf{f}_v$, $\mathbf{g}_v$ and $\mathbf{h}_v$ are the viscous fluxes which take the following form:

$$\begin{gathered}
\mathbf{f}_v = \begin{Bmatrix} 0 \\ \tau_{xx} \\ \tau_{yx} \\ \tau_{zx} \\ u\tau_{xx} + v\tau_{yx} + w\tau_{zx} + kT_x \end{Bmatrix}, \quad \mathbf{g}_v = \begin{Bmatrix} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{zy} \\ u\tau_{xy} + v\tau_{yy} + w\tau_{zy} + kT_y \end{Bmatrix}, \\
\mathbf{h}_v = \begin{Bmatrix} 0 \\ \tau_{xz} \\ \tau_{yz} \\ \tau_{zz} \\ u\tau_{xz} + v\tau_{yz} + w\tau_{zz} + kT_z \end{Bmatrix},
\end{gathered} \tag{9.6}$$

where τ_{xx}, τ_{xy}, τ_{xz}, τ_{yx}, τ_{yx}, τ_{yy}, τ_{yz}, τ_{zx}, τ_{zy} and τ_{zz} are the components of the stress tensor[1]

$$\begin{aligned}
\tau_{xx} &= 2\mu\left(u_x - \tfrac{u_x+v_y+w_z}{3}\right), \quad \tau_{yy} = 2\mu\left(v_y - \tfrac{u_x+v_y+w_z}{3}\right), \\
\tau_{zz} &= 2\mu\left(w_z - \tfrac{u_x+v_y+w_z}{3}\right), \quad \tau_{xy} = \tau_{yx} = \mu(v_x + u_y), \\
\tau_{yz} &= \tau_{zy} = \mu(w_y + v_z), \quad \tau_{zx} = \tau_{xz} = \mu(u_z + w_x).
\end{aligned} \tag{9.7}$$

where μ is the dynamic viscosity calculated using the Sutherland's law and k is the thermal conductivity.

9.1.3 Numerical discretisation

In Nektar++ the spatial discretisation of the Euler and of the Navier-Stokes equations is projected in the polynomial space via a discontinuous projection. Specifically we make use either of the discontinuous Galerkin (DG) method or the Flux Reconstruction (FR)

[1] Note that we use Stokes hypothesis $\lambda = -2/3$.

approach. In both the approaches the physical domain Ω is divided into a mesh of N non-overlapping elements Ω_e and the solution is allowed to be discontinuous at the boundary between two adjacent elements. Since the Euler as well as the Navier-Stokes equations are defined locally (on each element of the computational domain), it is necessary to define a term to couple the elements of the spatial discretisation in order to allow information to propagate across the domain. This term, called numerical interface flux, naturally arises from the discontinuous Galerkin formulation as well as from the Flux Reconstruction approach.

For the advection term it is common to solve a Riemann problem at each interface of the computational domain through exact or approximated Riemann solvers. In Nektar++ there are different Riemann solvers, one exact and nine approximated. The exact Riemann solver applies an iterative procedure to satisfy conservation of mass, momentum and energy and the equation of state. The left and right states are connected either with the unknown variables through the Rankine-Hugoniot relations, in the case of shock, or the isentropic characteristic equations, in the case of rarefaction waves. Across the contact surface, conditions of continuity of pressure and velocity are employed. Using these equations the system can be reduced to a non-linear algebraic equation in one unknown (the velocity in the intermediate state) that is solved iteratively using a Newton method. Since the exact Riemann solver gives a solution with an order of accuracy that is related to the residual in the Newton method, the accuracy of the method may come at high computational cost. The approximated Riemann solvers are simplifications of the exact solver.

Concerning the diffusion term, the coupling between the elements is achieved by using a local discontinuous Galerkin (LDG) approach as well as five different FR diffusion terms.

The boundary conditions are also implemented by exploiting the numerical interface fluxes just mentioned. For a more detailed description of the above the interested reader can refer to [8] and [21].

9.2 Usage

```
CompressibleFlowSolver session.xml
```

9.3 Session file configuration

In the following we describe the session file configuration. Specifically we consider the sections under the tag `<CONDITIONS>` in the session (.xml) file.

Parameters

Under this section it is possible to set the parameters of the simulation.

```
<PARAMETERS>
  <P> TimeStep            = 0.0000001                 </P>
```

```
 <P> FinTime             = 1.0                        </P>
 <P> NumSteps            = FinTime/TimeStep           </P>
 <P> IO_CheckSteps       = 5000                       </P>
 <P> IO_InfoSteps        = 1                          </P>
 <P> Gamma               = 1.4                        </P>
 <P> pInf                = 101325                     </P>
 <P> rhoInf              = 1.225                      </P>
 <P> TInf                = pInf/(287.058*rhoInf)      </P>
 <P> Twall               = pInf/(287.058*rhoInf)+15.0 </P>
 <P> uInf                = 147.4                      </P>
 <P> vInf                = 0.0                        </P>
 <P> wInf                = 0.0                        </P>
 <P> mu                  = 1e-5                       </P>
 <P> Pr                  = 0.72                       </P>
 <P> thermalConductivity = 0.02                       </P>
</PARAMETERS>
```

- `TimeStep` is the time-step we want to use;
- `FinTime` is the final physical time at which we want our simulation to stop;
- `NumSteps` is the equivalent of `FinTime` but instead of specifying the physical final time we specify the number of time-steps;
- `IO_CheckSteps` sets the number of steps between successive checkpoint files;
- `IO_InfoSteps` sets the number of steps between successive info stats are printed to screen;
- `Gamma` ratio of the specific heats. Default value = 1.4;
- `pInf` farfield pressure (i.e. p_∞). Default value = 101325 Pa;
- `rhoInf` farfield density (i.e. ρ_∞). Default value = 1.225 Kg/m^3;
- `TInf` farfield temperature (i.e. T_∞). Default value = 288.15 K;
- `Twall` temperature at the wall when isothermal boundary conditions are employed (i.e. T_w). Default value = 300.15K;
- `uint` farfield X-component of the velocity (i.e. u_∞). Default value = 0.1 m/s;
- `vInf` farfield Y-component of the velocity (i.e. v_∞). Default value = 0.0 m/s;
- `wInf` farfield Z-component of the velocity (i.e. w_∞). Default value = 0.0 m/s;
- `mu` dynamic viscosity (i.e. μ_∞). Default value = 1.78e-05 Pas;
- `Pr` Prandtl number. Default value = 0.72;
- `thermalConductivity` thermal conductivity (i.e. κ_∞). Default value = 0.0257 $W/(Km)$;

Solver info

Under this section it is possible to set the solver information.

```
<SOLVERINFO>
  <I PROPERTY="EQType"                VALUE="NavierStokesCFE"     />
  <I PROPERTY="Projection"            VALUE="DisContinuous"       />
  <I PROPERTY="AdvectionType"         VALUE="WeakDG"              />
  <I PROPERTY="DiffusionType"         VALUE="LDGNS"               />
  <I PROPERTY="TimeIntegrationMethod" VALUE="ClassicalRungeKutta4"/>
  <I PROPERTY="UpwindType"            VALUE="ExactToro"           />
  <I PROPERTY="ProblemType"           VALUE="General"             />
  <I PROPERTY="ViscosityType"         VALUE="Constant"            />
</SOLVERINFO>
```

- `EQType` is the tag which specify the equations we want solve:
  - `NavierStokesCFE` (Compressible Navier-Stokes equations);
  - `EulerCFE` (Compressible Euler equations).
- `Projection` is the type of projection we want to use:
  - `DisContinuous`.
    Note that the Continuous projection is not supported in the Compressible Flow Solver.
- `AdvectionType` is the advection operator we want to use:
  - `WeakDG` (classical DG in weak form);
  - `FRDG` (Flux-Reconstruction recovering nodal DG scheme);
  - `FRSD` (Flux-Reconstruction recovering a spectral difference (SD) scheme);
  - `FRHU` (Flux-Reconstruction recovering Huynh (G2) scheme);
  - `FRcmin` (Flux-Reconstruction with $c = c_{min}$);
  - `FRcinf` (Flux-Reconstruction with $c = \infty$).
- `DiffusionType` is the diffusion operator we want to use:
  - `WeakDG` (classical DG in weak form);
  - `FRDG` (Flux-Reconstruction recovering nodal DG scheme);
  - `FRSD` (Flux-Reconstruction recovering a spectral difference (SD) scheme);
  - `FRHU` (Flux-Reconstruction recovering Huynh (G2) scheme);
  - `FRcmin` (Flux-Reconstruction with $c = c_{min}$);
  - `FRcinf` (Flux-Reconstruction with $c = \infty$).
- `TimeIntegrationMethod` is the time-integration scheme we want to use. Note that only an explicit discretisation is supported:

  - `ForwardEuler`;
  - `RungeKutta2_ImprovedEuler`;
  - `ClassicalRungeKutta4`.

- `UpwindType` is the numerical interface flux (i.e. Riemann solver) we want to use for the advection operator:
  - `AUSM0`;
  - `AUSM1`;
  - `AUSM2`;
  - `AUSM3`;
  - `Average`;
  - `ExactToro`;
  - `HLL`;
  - `HLLC`;
  - `LaxFriedrichs`;
  - `Roe`.

- `ProblemType` is the problem type we want to solve. This tag is supported for solving ad hoc problems such as the isentropic vortex or the Ringleb flow.
  - `General`;
  - `IsentropicVortex`;
  - `RinglebFlow`;

- `ViscosityType` is the viscosity type we want to use:
  - `Constant` (Constant viscosity);
  - `Variable` (Variable viscosity through the Sutherland's law.);

Boundary conditions

In this section we can specify the boundary conditions for our problem. First we need to define the variables under the section `VARIABLES`. For a 1D problem we have:

```
<VARIABLES>
  <V ID="0"> rho  </V>
  <V ID="1"> rhou </V>
  <V ID="4"> E    </V>
</VARIABLES>
```

For a 2D problem we have

```
<VARIABLES>
  <V ID="0"> rho  </V>
  <V ID="1"> rhou </V>
  <V ID="2"> rhov </V>
  <V ID="4"> E    </V>
</VARIABLES>
```

For a 3D problem we have:

```
<VARIABLES>
  <V ID="0"> rho  </V>
  <V ID="1"> rhou </V>
  <V ID="2"> rhov </V>
  <V ID="3"> rhow </V>
  <V ID="4"> E    </V>
</VARIABLES>
```

After having defined the variables depending on the dimensions of the problem we want to solve it is necessary to specify the boundary regions on which we want to define the boundary conditions:

```
<BOUNDARYREGIONS>
  <B ID="0"> C[100] </B>
</BOUNDARYREGIONS>
```

Finally we can specify the boundary conditions on the regions specified under `BOUNDARYREGIONS`. In the following some examples for a 2D problem:

- Slip wall boundary conditions:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="Wall" VALUE="0" />
    <D VAR="rhou" USERDEFINEDTYPE="Wall" VALUE="0" />
    <D VAR="rhov" USERDEFINEDTYPE="Wall" VALUE="0" />
    <D VAR="E"    USERDEFINEDTYPE="Wall" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

- No-slip wall boundary conditions:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="WallViscous" VALUE="0" />
    <D VAR="rhou" USERDEFINEDTYPE="WallViscous" VALUE="0" />
    <D VAR="rhov" USERDEFINEDTYPE="WallViscous" VALUE="0" />
    <D VAR="E"    USERDEFINEDTYPE="WallViscous" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

- Adiabatic wall boundary conditions:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
    <D VAR="rhou" USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
    <D VAR="rhov" USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
    <D VAR="E"    USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

- Farfield boundary conditions (including inviscid characteristic boundary conditions):

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  VALUE="rhoInf" />
    <D VAR="rhou" VALUE="rhoInf*uInf" />
    <D VAR="rhov" VALUE="rhoInf*vInf" />
    <D VAR="E"
    VALUE="pInf/(Gamma-1)+0.5*rhoInf*(uInf*uInf+vInf*vInf+wInf*wInf)"/>
  </REGION>
</BOUNDARYCONDITIONS>
```

- Pressure outflow boundary conditions:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="PressureOutflow" VALUE="0" />
    <D VAR="rhou" USERDEFINEDTYPE="PressureOutflow" VALUE="0" />
    <D VAR="rhov" USERDEFINEDTYPE="PressureOutflow" VALUE="0" />
    <D VAR="E"    USERDEFINEDTYPE="PressureOutflow" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

Initial conditions and exact solution

Under the two following sections it is possible to define the initial conditions and the exact solution (if existent).

```
<FUNCTION NAME="InitialConditions">
  <E VAR="rho"    VALUE="rhoInf"/>
  <E VAR="rhou"   VALUE="rhoInf*uInf"   />
  <E VAR="rhov"   VALUE="rhoInf*vInf"   />
  <E VAR="E"
  VALUE="pInf/(Gamma-1)+0.5*rhoInf*(uInf*uInf+vInf*vInf+wInf*wInf)"/>
</FUNCTION>

<FUNCTION NAME="ExactSolution">
  <E VAR="rho"    VALUE="rhoInf"          />
  <E VAR="rhou"   VALUE="rhoInf*uInf"   />
  <E VAR="rhov"   VALUE="rhoInf*vInf"   />
  <E VAR="E"
  VALUE="pInf/(Gamma-1)+0.5*rhoInf*(uInf*uInf+vInf*vInf+wInf*wInf)"/>
</FUNCTION>
```

9.4 Examples

9.4.1 Shock capturing

Compressible flow is characterised by abrupt changes in density within the flow domain often referred to as shocks. These discontinuities lead to numerical instabilities (Gibbs phenomena). This problem is prevented by locally adding a diffusion term to the equations to damp the numerical fluctuations. These fluctuations in an element are identified using a sensor algorithm which quantifies the smoothness of the solution within an element. The value of the sensor in an element is defined as

$$S_e = \frac{||\rho_e^p - \rho_e^{p-1}||_{L_2}}{||\rho_e^p||_{L_2}} \tag{9.8}$$

An artificial diffusion term is introduced locally to the Euler equations to deal with flow discontinuity and the consequential numerical oscillations. Two models are implemented, a non-smooth and a smooth artificial viscosity model.

9.4.1.1 Non-smooth artificial viscosity model

For the non-smooth artificial viscosity model the added artificial viscosity is constant in each element and discontinuous between the elements. The Euler system is augmented by an added laplacian term on right hand side of equation 9.1. The diffusivity of the system is controlled by a variable viscosity coefficient ϵ. The value of ϵ is dependent on ϵ_0, which is the maximum viscosity that is dependent on the polynomial order (p), the mesh size (h) and the maximum wave speed and the local sensor value. Based on pre-defined sensor threshold values, the variable viscosity is set accordingly

$$\epsilon = \epsilon_0 \begin{cases} 0 & \text{if } \; s_e < s_\kappa - \kappa \\ 0.5\left(1 + \sin\frac{\pi(S_e - s_\kappa)}{2\kappa}\right) & \text{if } \; s_\kappa - \kappa < S_e < s_\kappa + \kappa \\ 1 & \text{if } \; s_e > s_\kappa + \kappa \end{cases} \tag{9.9}$$

9.4.1.2 Smooth artificial viscosity model

For the smooth artificial viscosity model an extra PDE for the artificial viscosity is appended to the Euler system

$$\begin{aligned} \frac{\partial \epsilon}{\partial t} &= \nabla \cdot (\nabla \epsilon) + \frac{1}{\tau}\left(\frac{h}{p}\lambda_{max} S_\kappa - \epsilon\right) \qquad \text{on} \qquad \Omega \\ \frac{\partial \epsilon}{\partial n} &= 0 \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{on} \qquad \Gamma \end{aligned} \tag{9.10}$$

where S_κ is a normalised sensor value and serves as a forcing term for the artificial viscosity. A smooth artificial viscosity distribution is obtained.

To enable the smooth viscosity model, the following line has to be added to the `SOLVERINFO` section:

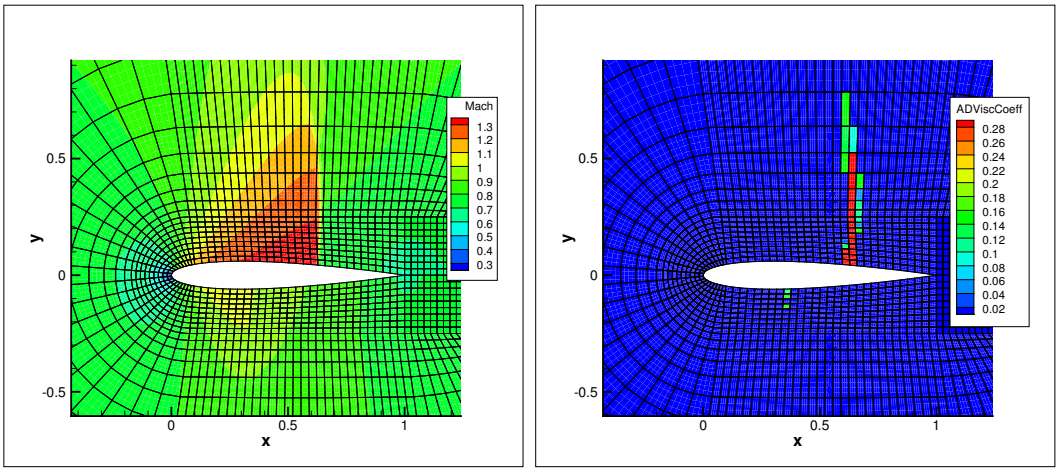


Figure 9.1 (a) Steady state solution for $M = 0.8$ flow at $\alpha = 1.25°$ past a NACA 0012 profile, (b) Artificial viscosity (ϵ) distribution

```
 <SOLVERINFO>
<I PROPERTY="ShockCaptureType"      VALUE="Smooth"              />
 <SOLVERINFO>
```

Furthermore, the extra viscosity variable `eps` has to be added to the variable list:

```
<VARIABLES>
  <V ID="0"> rho  </V>
  <V ID="1"> rhou </V>
  <V ID="2"> rhov </V>
  <V ID="4"> E    </V>
  <V ID="5"> eps </V>
</VARIABLES>
```

A similar addition has to be made for the boundary conditions and initial conditions. The tests that have been run started with a uniform homogeneous boundary condition and initial condition. The following parameters can be set in the xml session file:

```
<PARAMETERS>
<P> Skappa              = -1.3                  </P>
<P> Kappa               = 0.2                   </P>
<P> mu0                 = 1.0                   </P>
<P> FH                  = 3                     </P>
<P> FL                  = 0.01*FH               </P>
<P> C1                  = 0.03                  </P>
<P> C2                  = 5/3*C1                </P>
</PARAMETERS>
```

where for now `FH` and `FL` are used to tune which range of the sensor is used as a forcing term and `C1` and `C2` are fixed constants which can be played around with to make the model more diffusive or not. However these constants are generally fixed.

9.4.2 Variable polynomial order

A sensor based p-adaptive algorithm is implemented to optimise the computational cost and accuracy. The DG scheme allows one to use different polynomial orders since the fluxes over the elements are determined using a Riemann solver and there is now further coupling between the elements. Furthermore, the initial p-adaptive algorithm uses the same sensor as the shock capturing algorithm to identify the smoothness of the local solution so it rather straightforward to implement both algorithms at the same time.

The polynomial order in each element can be adjusted based on the sensor value that is obtained. Initially, a converged solution is obtained after which the sensor in each element is calculated. Based on the determined sensor value and the pre-defined sensor thresholds, it is decided to increase, decrease or maintain the degree of the polynomial approximation in each element and a new converged solution is obtained.

$$p_e = \begin{cases} p_e - 1 & \text{if } s_e > s_{ds} \\ p_e + 1 & \text{if } s_{sm} < s_e < s_{ds} \\ p_e & \text{if } s_{fl} < s_e < s_{sm} \\ p_e - 1 & \text{if } s_e < s_{fl} \end{cases} \tag{9.11}$$

For now, the threshold values s_e, s_{ds}, s_{sm} and s_{fl} are determined empirically by looking at the sensor distribution in the domain. Once these values are set, two .txt files are outputted, one that has the composites called VariablePComposites.txt and one with the expansions called VariablePExpansions.txt. These values have to copied into a new .xml file to create the adapted mesh.

Chapter 10

Incompressible Navier-Stokes Solver

10.1 Synopsis

10.1.1 Velocity Correction Scheme

A useful tool implemented in Nektar++ is the incompressible Navier Stokes solver that allows to solve the governing equation for viscid Newtonians fluid using two different algorithms.

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{V} \cdot \nabla \mathbf{V} = -\nabla p + \nu \nabla^2 \mathbf{V} + \mathbf{f} \tag{10.1a}$$

$$\nabla \cdot \mathbf{V} = 0 \tag{10.1b}$$

where V is the velocity, p the pressure and ν the kinematic viscosity. The first approach uses a splitting/projection method where the velocity matrix system and the pressure are typically decoupled. Splitting schemes are typically favourite for their numerical efficiency since for a Newtonian fluid the velocity and pressure are handled independently, requiring the solution of three (in two dimensions) elliptic systems of rank N (opposed to a single system of rank 3N solved in the Stokes problem). However, a drawback of this approach is the splitting scheme error which is introduced when decoupling the pressure and the velocity system, although this can be made consistent with the overall temporal accuracy of the scheme by appropriate discretisation of the pressure boundary conditions. When the incompressible Navier-Stokes equations are solved using the velocity correction splitting scheme, a stiffly-stable time integration is applied (derived from the work of Karniadakis, Israeli and Orszag [16], see also Guermond and Shen [13]). Briefly, the time integration scheme consists of the following steps:

1. Calculate a first intermediate velocity field evaluating the advection term explicitly

and combining it with the solution at previous time-steps:

$$\frac{\tilde{\mathbf{V}} - \sum_{q=0}^{J-1} \frac{\alpha_q}{\gamma_0} \mathbf{V}^{n-q}}{\Delta t} = \sum_{q=0}^{J-1} \frac{\beta_q}{\gamma_0} [-(\mathbf{V} \cdot \nabla)\mathbf{V}]^{n-q} + \mathbf{f}^{n+1} \tag{10.2}$$

2. Solve a Poisson equation to obtain the pressure solution at the new time level:

$$\Delta p^{n+1} = \Big(\frac{\gamma_0}{\Delta t}\Big) \nabla \cdot \tilde{\mathbf{V}} \tag{10.3}$$

with consistent Neumann boundary conditions prescribed as

$$\frac{\partial p}{\partial n}^{n+1} = -\Big[\frac{\partial \mathbf{V}}{\partial t}^{n+1} + \nu \sum_{q=0}^{J-1} \beta_q (\nabla \times \nabla \times \mathbf{V})^{n-q} + \sum_{q=0}^{J-1} \beta_q [(\mathbf{V} \cdot \nabla)\mathbf{V}]^{n-q}\Big] \cdot \mathbf{n} \tag{10.4}$$

and on the outflow boundary, either setting $p^{n+1} = 0$ or using high order Dirichlet boundary condition, which is given as[9]

$$p^{n+1} = \nu \sum_{q=0}^{J-1} \nabla \mathbf{V}^{n-q} \cdot \mathbf{n} - \frac{1}{2} \mid \sum_{q=0}^{J-1} \mathbf{V}^{n-q} \mid^2 S_o(\mathbf{n} \cdot \sum_{q=0}^{J-1} \mathbf{V}^{n-q}) - \mathbf{f}_b^{n+1} \cdot \mathbf{n} \tag{10.5}$$

with a step function defined by $S_o(n \cdot \mathbf{V}) = \frac{1}{2}(1 - \tanh \frac{n \cdot \mathbf{V}}{\mathbf{V_0} \delta})$, where $\mathbf{V_0}$ is the characteristic velocity scale and δ is a non-dimensional positive constant chosen to be sufficiently small. $\mathbf{f}_b$ is the forcing term in this case the analytical conditions can be given but if these are not known explicitly, it is set to zero, i.e. $\mathbf{f}_b = 0$.

3. Calculate a second intermediate velocity field:

$$\tilde{\tilde{\mathbf{V}}} = \tilde{\mathbf{V}} - \Big(\frac{\Delta t}{\gamma_0}\Big) \nabla p^{n+1} \tag{10.6}$$

4. Use the second intermediate velocity field as a forcing term in a Helmholtz problem to obtain the velocity field at the new time level:

$$\Big(\Delta - \frac{\gamma_0}{\Delta t \nu}\Big) \mathbf{V}^{n+1} = -\Big(\frac{\gamma_0}{\Delta t \nu}\Big) \tilde{\tilde{\mathbf{V}}} \tag{10.7}$$

with the outflow boundary conditions either set as $n \cdot \nabla \mathbf{V}^{n+1} = 0$ or as high order Neumann boundary conditions prescribed by[9]:

$$\mathbf{n} \cdot \nabla \mathbf{V}^{n+1} = \frac{1}{\nu}\Big[p^{n+1}\mathbf{n} + \frac{1}{2} \mid \sum_{q=0}^{J-1} \mathbf{V}^{n-q} \mid^2 S_o(\mathbf{n} \cdot \sum_{q=0}^{J-1} \mathbf{V}^{n-q})\mathbf{n} - \nu \sum_{q=0}^{J-1} (\nabla \cdot \mathbf{V}^{n-q})\mathbf{n} + \mathbf{f}_b^{n+1}\Big] \tag{10.8}$$

Here, J is the integration order and γ_0, α_q, β_q are the stiffly stable time integration coefficients given in the table below.

	$1st\ order$	$2nd\ order$	$3rd\ order$
γ_0	1	3/2	11/6
α_0	1	2	3
α_1	0	−1/2	−3/2
α_2	0	0	1/3
β_0	1	2	3
β_1	0	−1	−3
β_2	0	0	1

(10.9)

This multi-step implicit-explicit splitting scheme decouples the velocity field $\mathbf{V}$ from the pressure p, leading to an explicit treatment of the advection term and an implicit treatment of the pressure and the diffusion term.

10.1.2 Substepping the Velocity Correction Scheme

It is possible to use difference time steps in the velocity correction scheme using a substepping [28] or auxiliary semi-Lagrangian approach [35]. Originally the scheme was proposed by Maday, Patera and Ronquist who referred to as an operator-integration-factor splitting method [20]

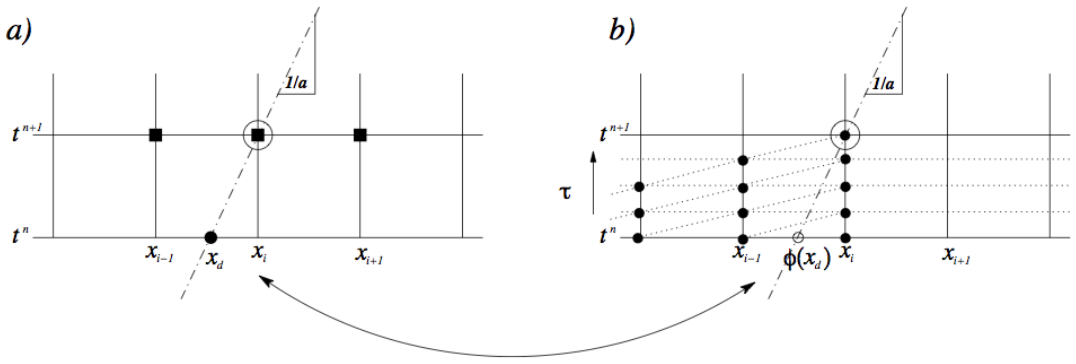


Figure 10.1 Schematic representation of the substepping approach. (a) Making an explicit time step the hyperbolic solution, travelling with a speed a, can be understood as being related to the solution at point x_d (the departure point). (b) Making smaller explicit time steps we can evaluate the solution $\phi(x_d)$ at the departure point and then use this value to make a semi-Lagrangian discretisation of the implicit components usually associated with diffusion.

A schematic of the approach can be understood from figure 10.1.2 where we observe that smaller time steps can be used for the explicit advection steps whilst a larger overall time step is adopted for the more expensive implicit solve for the diffusion operator. More details of the implementation can be found in [35] and [28]. In the following sections

we outline the parameters that can be used to set up this scheme. Since the explicit part is advanced using a DG scheme it is necessary to use a `Mixed_CG_Discontinuous` expansion with this option.

Note

Example of the substepping scheme can be found in the regression tests directory under `$NEKHOME/Solver/IncNavierStokesSolver/Tests/` directory. For exmaple the test cases `KovaFlow_SubStep_2order.xml`, `Hex_Kovasnay_SubStep.xml` and `Tet_Kovasnay_SubStep.xml`

10.1.3 Direct solver (coupled approach)

The second approach consists of directly solving the matrix problem arising from the discretization of the Stokes problem. The direct solution of the Stokes system introduces the problem of appropriate spaces for the velocity and the pressure systems to satisfy the inf-sup condition and it requires the solution of the full velocity-pressure system. However, if a discontinuous pressure space is used then all but the constant mode of the pressure system can be decoupled from the velocity. When implementing this approach with a spectral/hp element discretization, the remaining velocity system may then also be statically condensed to decouple the so called interior elemental degrees of freedom, reducing the Stokes problem to a smaller system expressed on the elemental boundaries. The direct solution of the Stokes problem provides a very natural setting for the solution of the pressure system which is not easily dealt with in a splitting scheme. Further, the solution of the full coupled velocity system allows the introduction of a spatially varying viscosity, which arise for non-Newtonian flows, with only minor modifications.

We consider the weak form of the Stokes problem for the velocity field $boldsymbolu = [u, v]^T$ and the pressure field p:

$$(\nabla\phi, \nu\nabla\boldsymbol{u}) - (\nabla\cdot\phi, p) = (\phi, \boldsymbol{f}) \tag{10.10a}$$

$$(q, \nabla\cdot\boldsymbol{u}) = 0 \tag{10.10b}$$

where the components of A,B and C are $\nabla\phi_b, \nu\nabla\boldsymbol{u_b}$, $\nabla\phi_b, \nu\nabla\boldsymbol{u_i}$ and $\nabla\phi_i, \nu\nabla\boldsymbol{u_i}$ and the components D_b and D_i are $q, \nabla\boldsymbol{u_b}$ and $q, \nabla\boldsymbol{u_i}$. The indices b and i refer to the degrees of freedom on the elemental boundary and interior respectively. In constructing the system we have lumped the contributions form each component of the velocity field into matrices A,B and C. However, we note that for a Newtonian fluid the contribution from each field is decoupled. Since the interior degrees of freedom of the velocity field do not overlap, the matrix C is block diagonal and to take advantage of this structure we can statically condense out the C matrix to obtain the system:

$$\begin{bmatrix} A - BC^{-1}B^T & D_b^T - BC^{-1}D_i & 0 \\ D_b - D_i^T C^{-1}B^T & -D_i^T C^{-1}D_i & 0 \\ B^T & D_i & C \end{bmatrix} \begin{bmatrix} \boldsymbol{u_b} \\ p \\ \boldsymbol{u_i} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f_b} - BC^{-1}\boldsymbol{f_i} \\ -D_i^T C^{-1}\boldsymbol{f_i} \\ \boldsymbol{f_i} \end{bmatrix} \tag{10.11}$$

To extend the above Stokes solver to an unsteady Navier-Stokes solver we first introduce the unsteady term, $\partial \boldsymbol{u}/\partial t$, into the Stokes problem. This has the principal effect of modifying the weak Laplacian operator $\nabla\phi, \nu\nabla\boldsymbol{u}]$ into a weak Helmholtz operator $\nabla\phi, \nu\nabla\boldsymbol{u}) - \lambda(\phi, \boldsymbol{u}$ where λ depends on the time integration scheme. The second modification requires the explicit discretisation of the non-linear terms in a similar manner to the splitting scheme and this term is then introduced as the forcing term $\boldsymbol{f}$. For more details see [1, 29].

10.1.4 Linear Stability Analysis

Hydrodynamic stability is an important part of fluid-mechanics that has a relevant role in understanding how an unstable flow can evolve into a turbulent state of motion with chaotic three-dimensional vorticity fields and a broad spectrum of small temporal and spatial scales. The essential problems of hydrodynamic stability were recognised and formulated in 19th century, notably by Helmholtz, Kelvin, Rayleigh and Reynolds.

Conventional linear stability assumes a normal representation of the perturbation fields that can be represented as independent wave packets, meaning that the system is self-adjoint. The main aim of the global stability analysis is to evaluate the amplitude of the eigenmodes as time grows and tends to infinity. However, in most industrial applications, it is also interesting to study the behaviour at intermediate states that might affects significantly the functionality and performance of a device. The study of the transient evolution of the perturbations is seen to be strictly related to the non-normality of the linearised Navier-Stokes equations, therefore the normality assumptiong of the system is no longer assumed. The eigenmodes of a non-normal system do not evolve independently and their interaction is responsible for a non-negligible transient growth of the energy. Conventional stability analysis generally does not capture this behaviour, therefore other techniques should be used.

A popular approach to study the hydrodynamic stability of flows consists in performing a direct numerical simulation of the linearised Navier-Stokes equations using iterative methods for computing the solution of the associated eigenproblem. However, since linearly stable flows could show a transient increment of energy, it is necessary to extend this analysis considering the combined effect of the direct and adjoint evolution operators. This phenomenon has noteworthy importance in several engineering applications and it is known as transient growth.

In Nektar++ it is then possible to use the following tools to perform stability analysis:

- direct stability analysis;

- adjoint stability analysis;
- transient growth analysis;

10.1.4.1 Direct stability analysis

The equations that describe the evolution of an infinitesimal disturbance in the flow can be derived decomposing the solution into a basic state $(\mathbf{U}, p)$ and a perturbed state $\mathbf{U} + \varepsilon\mathbf{u}'$ with $\varepsilon \ll 1$ that both satisfy the Navier-Stokes equations. Substituting into the Navier Stokes equations and considering that the quadratic terms $\mathbf{u}' \cdot \nabla\mathbf{u}'$can be neglected, we obtain the linearised Navier-Stokes equations:

$$\frac{\partial \mathbf{u}'}{\partial t} + \mathbf{U} \cdot \nabla\mathbf{u}' + \mathbf{u}' \cdot \nabla\mathbf{U} = -\nabla p + \nu\nabla^2\mathbf{u}' + \mathbf{f} \tag{10.12a}$$

$$\nabla \cdot \mathbf{u}' \tag{10.12b}$$

The linearised Navier-Stokes equations are identical in form to the non-linear equation, except for the non-linear advection term. Therefore the numerical techniques used for solving Navier-Stokes equations can still be applied as long as the non-linear term is substituted with the linearised one. It is possible to define the linear operator that evolved the perturbation forward in time:

$$\mathbf{u}'(\mathbf{x}, t) = \mathcal{A}(\mathbf{U})\mathbf{u}'(\mathbf{x}, 0) \tag{10.13}$$

Let us assume that the base flow $\mathbf{U}$ is steady, then the perturbations are autonomous and we can assume that:

$$\mathbf{u}'(\mathbf{x}, t) = \mathbf{q}'(\mathbf{x}) \exp(\lambda t) \quad \text{where } \lambda = \sigma + i\omega \tag{10.14}$$

Then we obtain the associated eigenproblem:

$$\mathcal{A}(\mathbf{U})\mathbf{q}' = \lambda\mathbf{q}' \tag{10.15}$$

The dominant eigenvalue determines the behaviour of the flow. If it the real part is positive there exists exponentially growing solutions, conversely if every single eigenvalues has negative real part then the flow is linearly stable. If the real part of the eigenvalue is zero, it is present a bifurcation point.

10.1.4.2 Adjoint Stability Analysis

The adjoint of a linear operator is one of the most important concept in functional analysis and has an it has important role to tackle the transition to turbulence. Let us write the linearised Navier-Stokes equation in a compact form:

$$\mathcal{H}\mathbf{q} = 0 \quad \text{where} \quad \mathcal{H} = \left(\begin{array}{c|c} -\partial_t - (\mathbf{U}\cdot\nabla) + (\nabla\mathbf{U})\cdot + \frac{1}{Re}\nabla^2 & -\nabla \\ \hline \nabla\cdot & 0 \end{array} \right) \tag{10.16}$$

The adjoint operator ($\mathcal{H}^*$ is defines as:

$$\langle \mathcal{H}\mathbf{q}, \mathbf{q} \rangle = \langle \mathbf{q}, \mathcal{H}^*\mathbf{q}^* \rangle \tag{10.17}$$

Integrating by parts and employing the divergence theorem, it is possible to express the adjoint equations:

$$-\frac{\partial \mathbf{u}^*}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{u}^* + (\nabla\mathbf{U})^T\cdot\mathbf{u}^* = -\nabla p^* + \frac{1}{Re}\nabla^2\mathbf{u} \tag{10.18a}$$

$$\nabla\cdot\mathbf{u}^* = 0 \tag{10.18b}$$

The adjoint fields are in fact related to the concept of **receptivity**. The value of the adjoint velocity at a point in the flow indicates the response that arises from an unsteady momentum source at that point. The adjoint pressure and the adjoint stream function play instead the same role for mass and vorticity sources respectively. Therefore, the adjoint modes can be seen as a powerful tool to understand where to act in order to ease/inhibit the transition.

10.1.4.3 Transient Growth Analysis

Transient growth is a phenomenon that occurs when a flow that is linearly stable, but whose perturbations exhibit a non-negligible transient response due to regions of localised convective instabilities. This situation is common in many engineering applications, for example in open flows where the geometry is complex, producing a steep variation of the base flow. Therefore, the main question to answer is if it exists a bounded solution that exhibit large growth before inevitably decaying. Let us introduce a norm to quantify the size of a perturbation. It is physically meaningful to use the total kinetic energy of a perturbation on the domain Ω. This is convenient because it is directly associated with the standard-$L2$ inner product:

$$\mathcal{A}(\tau)\mathbf{v} = \sigma\mathbf{u}, \quad \|\mathbf{u}\| = 1 \tag{10.19}$$

where $\sigma = \|\mathbf{u}'(\tau)\|$. This is no other that the singular value decomposition of $\mathcal{A}(\tau)$. The phenomenology of the transient growth can be explained considering the non-normality of the linearised Navier-Stokes evolution operator. This can be simply understood using the simple geometric example showed in figure 10.1.4.3. Let us assume a unit-length vector $\mathbf{f}$ represented in a non-orthogonal basis .This vector is defined as the difference of the nearly collinear vectors $\mathbf{\Phi_1}$ and $\mathbf{\Phi_2}$. With the time progression, the component of these two vectors decrease respectively by 20% and 50%. The vector $\mathbf{f}$ increases substantially in length before decaying to zero. Thus, the superposition of decaying non-orthogonal eigenmode can produce in short term a growth in the norm of the perturbations.

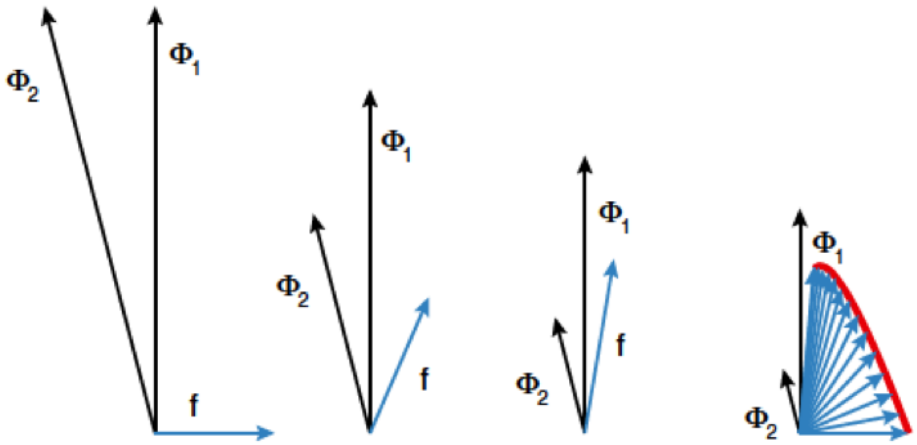


Figure 10.2 Geometric interpretation of the transient growth. Adapted from Schmid, 2007

10.1.5 Steady-state solver using Selective Frequency Damping

To compute linear stability analysis, the choice of the base flow, around which the system will be linearised, is crucial. If one wants to use the steady-state solution of the Navier-Stokes equations as base flow, a steady-state solver is implemented in *Nektar++*. The method used is the encapsulated formulation of the Selective Frequency Damping method [15]. Unstable steady base flows can be obtained with this method. The SFD method is based on the filtering and control of unstable temporal frequencies within the flow. The time continuous formulation of the SFD method is

$$\begin{cases} \dot{q} = NS(q) - \chi(q - \bar{q}), \\ \dot{\bar{q}} = \frac{q-\bar{q}}{\Delta}. \end{cases} \tag{10.20}$$

where q represents the problem unknown(s), the dot represents the time derivative, NS represents the Navier-Stokes equations, $\chi \in \mathbb{R}_+$ is the control coefficient, $\bar{q}$ is a filtered version of q, and $\Delta \in \mathbb{R}_+^*$ is the filter width of a first-order low-pass time filter. The steady-state solution is reached when $q = \bar{q}$.

The convergence of the method towards a steady-state solution depends on the choice of the parameters χ and Δ. They have to be carefully chosen: if they are too small, the instabilities within the flow can not be damped; but if they are too large, the method may converge extremely slowly. If the dominant eigenvalue of the flow studied is known (and given as input), the algorithm implemented can automatically select parameters that ensure a fast convergence of the SFD method. Most of the time, the dominant eigenvalue is not know, that is why an adaptive algorithm that adapts χ and Δ all along the solver execution is also implemented.

Note that this method can not be applied for flows with a pure exponential growth of the instabilities (*e.g.* jet flow within a pipe). In other words, if the frequency of the dominant eigenvalue is zero, then the SFD method is not a suitable tool to obtain a steady-state solution.

10.2 Usage

```
IncNavierStokesSolver session.xml
```

10.3 Session file configuration

In the following the possible options are shown for the incompressible Navier-Stokes. The `Expansion` section for an incompressible flow simulation can be set as for other solvers regardless of the projection type. Here an example for a 3D simulation (for 2D simulations the specified fields would be just `u,v,p`).

```
<EXPANSIONS>
  <E COMPOSITE="C[0]" NUMMODES="6" FIELDS="u,v,w,p" TYPE="MODIFIED" />
</EXPANSIONS>
```

In case of a simulation using the Direct Solver we need to set `FIELDS=u,v` as the pressure expansion order will be automatically set to fulfil the inf-sup condition. Possible choices for the expansion `TYPE` are:

Basis	TYPE
Modal	MODIFIED
Nodal	GLL_LAGRANGE
Nodal SEM	GLL_LAGRANGE_SEM

10.3.1 Solver Info

The following parameters can be specified in the `SOLVERINFO` section of the session file:

- `EqType`: sets the kind of equations we want to solve on the domain as:

```
<I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes"/>
```

Possible values are:

Equations	EQTYPE	Dim.	Projections	Alg.
Steady Stokes (SS)	SteadyStokes	All	CG	VCS
Steady Oseen (SO)	SteadyOseen	All	CG	DS
Unsteady Stokes (US)	UnsteadyStokes	All	CG	VCS
Steady Linearised NS (SLNS)	SteadyLinearisedNS	All	CG	DS
Unsteady Linearised NS (ULNS)	UnsteadyLinearisedNS	All	CG	VCS,DS
Unsteady NS (UNS)	UnsteadyNavierStokes	All	CG,CG-DG	VCS

- `SolverType`: sets the scheme we want to use to solve the set of equations as

```
<I PROPERTY="SolverType"  VALUE="VelocityCorrectionScheme"/>
```

Possible values are:

Algorithm	SolverType	Dimensions	Projections
Velocity Correction Scheme	VelocityCorrectionScheme	2D, Quasi-3D, 3D	CG, CG-DG
Direct solver	CoupledLinearisedNS	2D, Quasi-3D, 3D	CG

- `Driver`: this specifies the type of problem to be solved:

Driver	Description	Dimensions	Projections
Standard	Time integration of the equations	All	CG, DG
SteadyState	Steady-state solver (see Sec. 10.1.5)	All	CG, DG

- `Projection`: sets the Galerkin projection type as

```
<I PROPERTY="Projection" VALUE="Continuous"/>
```

Possible values are:

Galerkin Projection	Projection	Dimensions	Equations	Algorithms
Continuous (CG)	Continuous	All	All	All
Discontinuous (DG)	DisContinuous	All	...	...
Mixed CG and DG (CG-DG)	Mixed_CG_Discontinuous	2D,3D	just UNS	VCS-substepping

- `TimeIntegrationMethod`: sets the time integration method as

```
<I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2"/>
```

Possible values are

Time-Integration Method	`TimeIntegrationMethod`	Dimensions	Equations	Projections
IMEX Order 1	`IMEXOrder1`	all	US, UNS	CG
IMEX Order 2	`IMEXOrder2`	all	US, UNS	CG
IMEX Order 3	`IMEXOrder3`	all	US, UNS	CG
Backward Euler	`BackwardEuler`	all	US, UNS	CG-DG
BDF Order 1	`BDFImplicitOrder1`	all	US, UNS	CG-DG
BDF Order 2	`BDFImplicitOrder2`	all	US, UNS	CG-DG

- `Extrapolation`: Specify the extrapolation method (standard or substetping) to be used in velocity correction scheme. Essentially this activates the sub-stepping routine which requires the mixed CG-DG projection

```
1 <I PROPERTY="Extrapolation" VALUE="SubStepping"/>
```

  Possible values are `SubStepping` or `Standard` with "Standard" being the default value if nothing is specifiied.

- `SubStepIntScheme`: choose the substep DG time integration scheme so that a different order schems can be used as compared to the overal time integraiton scheme.

```
1 <I PROPERTY="SubStepIntScheme" VALUE="RungeKutta2\_ImprovedEuler"/>
```

  Possible values are

Time-Integration Method	`SubStepIntScheme`
Forward Euler	`ForwardEuler`
RK 2	`RungeKutta2_ImprovedEuler`

  This option is useful if you wish to use an overall scheme that is first order accurate for example with TimeIntegrationMethod as BDFImplicitOrder1 but using a second order RungeKutta2_ImprovedEuler for greater stability in the substep.

- `GlobalSysSoln`: sets the approach we use to solve the the linear systems of the type $Ax = b$ appearing in the solution steps, such as the Poisson equation for the pressure in the splitting-scheme. It can be set as

```
1 <I PROPERTY="GlobalSysSoln" VALUE="IterativeStaticCond"/>
```

  Possible values are

System solution	`GlobalSysSoln`	Parallel
Direct Solver (DS)	`DirectFull`	just quasi-3D
DS with Static Condensation	`DirectStaticCond`	just Quasi-3D
DS with Multilevel Static Condensation	`DirectMultiLevelStaticCond`	just Quasi-3D
Iterative Solver (IS)	`IterativeFull`	just Quasi-3D
IS with Static Condensation	`IterativeStaticCond`	quasi-3D
IS with Multilevel Static Condensation	`IterativeMultiLevelStaticCond`	quasi-3D

Default values are `DirectMultiLevelStaticCond` in serial and `IterativeStaticCond` in parallel.

- `SmoothAdvection`: activates a stabilization technique which smooths the advection term using the pressure inverse mass matrix. It can be used just in combination with nodal expansion basis for efficiency reasons.

```
<I PROPERTY="SmoothAdvection" VALUE="True"/>
```

- `SpectralVanishingViscosity`: activates a stabilization technique which increases the viscosity on the highest Fourier frequencies of a Quasi-3D approach.

```
<I PROPERTY="SpectralVanishingViscosity" VALUE="True"/>
```

- `DEALIASING`: activates the 3/2 padding rule on the advection term of a Quasi-3D simulation.

```
<I PROPERTY="DEALIASING" VALUE="ON"/>
```

- `SPECTRALHPDEALIASING`: activates the spectral/hp dealiasing to stabilize the simulation. This method is based on the work of Kirby and Sherwin [7].

```
<I PROPERTY="SPECTRALHPDEALIASING" VALUE="True" />
```

10.3.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `TimeStep`: sets the time-step for the integration in time formula
- `NumSteps`: sets the number of time-steps
- `Kinvis`: sets the cinematic viscosity coefficient formula
- `SubStepCFL`: sets the CFL safety limit for the sub-stepping algorithm (default value = 0.5)
- `MinSubSteps`: perform a minimum number of substeps in sub-stepping algorithm (default is 1)
- `MaxSubSteps`: perform a maxmimum number of substeps in sub-stepping algorithm otherwise exit (default is 100)
- `SVVCutoffRatio`: sets the ratio of Fourier frequency not affected by the SVV technique (default value = 0.75, i.e. the first 75% of frequency are not damped)
- `SVVDiffCoeff`: sets the SVV diffusion coefficient (default value = 0.1)

10.3.3 Forcing

10.3.3.1 MovingBody

Note

This force type is only supported for the Quasi-3D incompressible Navier-Stokes solver.

This force type allows the user to solve the interaction system of an incompressible fluid flowing past a flexible moving bodies [24]. By this forcing function, one can eliminate the difficulty of moving mesh by using body-fitted coordinates, so that an additional acceleration term(i.e., forcing term) is introduced to the momentum equations by the non-inertial transform from the deformed and moving coordinate system to non-deformed and stationary one.

```
<FORCE TYPE="MovingBody">
</FORCE>
```

Available options of the motion type for the moving body include free, constrained and forced vibrations, which can be specified in the `SOLVERINFO` section. The free type of motion allows the body to move in both streamwise and crossflow directions, while the constrained type limits the motion only in the crossflow direction. For the forced type, the vibration profiles of the body should be specified as a given function or read from input file in `MovingBody` section. here an example,

```
<SOLVERINFO>
    <I PROPERTY="EQTYPE"     VALUE="UnsteadyNavierStokes" />
    <I PROPERTY="SolverType"     VALUE="VelocityCorrectionScheme" />
    <I PROPERTY="EvolutionOperator"     VALUE="SkewSymmetric" />
    <I PROPERTY="Projection"     VALUE="Galerkin" />
    <I PROPERTY="GlobalSysSoln"     VALUE="DirectStaticCond"/>
    <I PROPERTY="TimeIntegrationMethod"     VALUE="IMEXOrder2" />
    <I PROPERTY="HOMOGENEOUS"     VALUE="1D"/>
    <I PROPERTY="USEFFT"     VALUE="FFTW"/>
    <I PROPERTY="VibrationType"     VALUE="FREE"/>
</SOLVERINFO>
```

A moving body type boundary condition should be specified in `BOUNDARYCONDITIONS` for the velocities on the moving body,

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" USERDEFINEDTYPE="MovingBody" VALUE="0" />
        <D VAR="v" USERDEFINEDTYPE="MovingBody" VALUE="0" />
        <D VAR="w" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
     </REGION>
</BOUNDARYCONDITIONS>
```

For the simulation of low mass ratio, there is an option to activate fictitious mass method for stabilizing explicit coupling between the fluid solver and structural dynamic solver. Here we need to specify the values of fictitious mass and damping in `PARAMETERS`, for example,

```
<SOLVERINFO>
    <I PROPERTY="FictitiousMassMethod"    VALUE="True"/>
</SOLVERINFO>
<PARAMETERS>
    <P> FictDamp      = 1000    </P>
    <P> FictMass       = 1.5       </P>
</PARAMETERS>
```

A filter calles as `MovingBody` is encapsulated in this module to evaluate the aerodynamic forces along the moving body surface. The forces for each computational plane are projected along the Cartesian axes and the pressure and viscous contributions are computed in each direction.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the forces are written.
`Frequency`	✗	1	Number of timesteps after which output is written.
`Boundary`	✓	-	Boundary surfaces on which the forces are to be evaluated.

To enable the filter, add the following to the `FORCE` tag::

```
 <FORCE TYPE="MovingBody">
     <PARAM NAME="OutputFile">DragLift.frc</PARAM>
     <PARAM NAME="OutputFrequency">10</PARAM>
     <PARAM NAME="Boundary"> B[0] </PARAM>
 </FORCE>
```

During the execution a file named `DragLift.frc` will be created and the value of the aerodynamic forces on boundary 0, defined in the `GEOMETRY` section, will be output every 10 time steps.evaluates the aerodynamic forces along the moving body surface. The forces for each computational plane are projected along the Cartesian axes and the pressure and viscous contributions are computed in each direction.

Also, to use this module a `MAPPING` needs to be specified, as described in section 10.6. In the case of free and constrained motion presented here, the functions defined by the mapping act as initial conditions. Also, when using the MovingBody forcing, it is not necessary to set the `TIMEDEPENDENT` property of the mapping.

10.4 Stability analysis Session file configuration

The type of equation which is to be solved is specified through the `EqType` option in the session file. This can be set to any of the following:

Equation to solve
$\frac{\partial \mathbf{u}'}{\partial t} + \mathcal{L}(\mathbf{U}, \mathbf{u}') = -\nabla p + \nu \nabla^2 \mathbf{u}'$

Equation Type	Dimensions	Projections	Algorithms
UnsteadyNavierStokes	2D, Quasi-3D	Continuous	VCS,DS

10.4.1 Solver Info

- `Eqtype`: sets the type of equation to solve, according to the table above.
- `TimeIntegrationMethod`: the following types of time integration methods have been tested with each solver:

	Explicit	Diagonally Implicit	IMEX	Implicit
`UnsteadyNavierStokes`	X		X	

- `Projection`: the Galerkin projection used may be either
  - `Continuous`: for a C0-continuous Galerkin (CG) projection;
  - `Discontinuous`: for a discontinous Galerkin (DG) projection.
- `EvolutionOperator`:
  - `Nonlinear` (non-linear Navier-Stokes equations).
  - `Direct` (linearised Navier-Stokes equations).
  - `Adjoint` (adjoint Navier-Stokes equations).
  - `TransientGrowth` ((transient growth evolution operator).
- `Driver`: specifies the type of problem to be solved:
  - `Standard` (time integration of the equations)
  - `ModifiedArnoldi` (computations of the leading eigenvalues and eigenmodes using modified Arnoldi method)
  - `Arpack` (computations of eigenvalues/eigenmodes using Implicitly Restarted Arnoldi Method (ARPACK)).
- `ArpackProblemType`: types of eigenvalues to be computed (for Driver Arpack only)

  - `LargestMag` (eigenvalues with largest magnitude).
  - `SmallestMag` (eigenvalues with smallest magnitude).
  - `LargestReal` (eigenvalues with largest real part).
  - `SmallestReal` (eigenvalues with smallest real part).
  - `LargestImag` (eigenvalues with largest imaginary part).
  - `SmallestIma` (eigenvalues with smallest imaginary part).
- `Homogeneous`: specifies the Fourier expansion in a third direction (optional)
  - `1D` (Fourier spectral method in z-direction).
- `ModeType`: this specifies the type of the quasi-3D problem to be solved.
  - `MultipleMode` (stability analysis with multiple modes).
  - `SingleMode` (BiGlobal Stability Analysis: full-complex mode).
  - `HalfMode` (BiGlobal Stability Analysis: half-complex mode u.Re v.Re w.Im p.Re).

10.4.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `Re`: sets the Reynolds number
- `Kinvis`: sets the kinematic viscosity ν.
- `kdim`: sets the dimension of the Krylov subspace κ. Can be used in: `ModifiedArnoldi` and `Arpack`. Default value: 16.
- `evtol`: sets the tolerance of the eigenvalues. Can be used in:
- `imagShift`: provide an imaginary shift to the direct sovler eigenvalue problem by the specified value. lttModifiedArnoldi and `Arpack`. Default value: 10^{-6}.
- `nits`: sets the maximum number of iterations. Can be used in: `ModifiedArnoldi` and `Arpack`. Default value: 500.
- `LZ`: sets the length in the spanswise direction L_z. Can be used in `Homogeneous` set to `1D`. Default value: 1.
- `HomModesZ`: sets the number of planes in the homogeneous directions. Can be used in `Homogeneous` set to `1D` and `ModeType` set to `MultipleModes`.
- `N_slices`: sets the number of temporal slices for Floquet stability analysis.

- `period`: sets the periodicity of the base flow.
- `realShift`: provide a real shift to the direct sovler eigenvalue problem by the specified value.

10.4.3 Functions

When using the direct solver for stability analysis it is necessary to specify a Forcing function "StabilityCoupledLNS" in the form:

```
<FORCING>
   <FORCE TYPE="StabilityCoupledLNS">
   </FORCE>
</FORCING>
```

This is required since we need to tell the solver to use the existing field as a forcing function to the direct matrix inverse as part of the Arnoldi iteration.

Note

Examples of the set up of the direct solver stability analysis (and other incompressible Navier-Stokes solvers) can be found in the regression test directory `NEKTAR/solvers/IncNavierStokesSolver/Tests`. See for example the files `PPF_R15000_ModifiedArnoldi_Shift.tst` and `PPF_R15000_3D.xml` noting that some parameters are specified in the .tst files.

10.5 Steady-state solver Session file configuration

In this section, we detail how to use the steady-state solver (that implements the selective frequency damping method, see Sec. 10.1.5). Two cases are detailed here: the execution of the classical SFD method and the *adaptive* SFD method, where the control coefficient χ and the filter width Δ of the SFD method are updated all along the solver execution. For the second case, the parameters of the SFD method do not need to be defined by the user (they will be automatically calculated all along the solver execution) but several session files must be defined in a very specific way.

10.5.1 Execution of the classical steady-state solver

10.5.1.1 Solver Info

The definition of `Eqtype`, `TimeIntegrationMethod` and `Projection` is similar as what is explained in 10.4.1. The use of the steady-state solver is enforced through the definition of the `Driver` which has to be `SteadyState`. `EvolutionOperator` does not need to be defined to run the unadapted SFD method (by default, it is set to `Nonlinear`).

10.5.1.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `Kinvis`: sets the kinematic viscosity ν. It is typically $1/Re$ if both the characteristic velocity and characteristic length are chosen to be 1.
- `ControlCoeff`: sets the control coefficient χ of the SFD method. Default value: 1.
- `FilterWidth`: sets the filter width Δ of the SFD method. Default value: 2.
- `GrowthRateEV` and `FrequencyEV`: if the growth rate and the frequency of the dominant eigenvalue are known, they can be given given as input and the code will automatically select the optimum parameters χ and Δ (and overwrite the values of `ControlCoeff` and `GrowthRateEV` that may be given in the session file)
- `TOL`: sets the tolerance of the SFD method. The code will run until $||q - \bar{q}||_{inf} < TOL$. Default value: 10^{-8}.

Note that for the steady-state solver, the parameter `NumSteps` is not taken into account. The solver will run until a steady-state solution is found and not for a pre-defined number of time steps.

10.5.2 Execution of the adaptive steady-state solver

Running the adaptive selective frequency damping method requires to set up the session files in a very specific manner. First, the `Geometry` section must be in a separated archive file. If the test case studied is called "Session", the mesh file must be called `Session.xml.gz` (the linux command "gzip" can be used to obtain this file).

The requirements for the file `Session.xml` are similar as for the ones for the classical SFD method. The `Geometry` section being removed and placed in `Session.xml.gz`. This file defines the properties of the nonlinear problem solved (*i.e.* the flow for which we want a steady-state). Also, the `SOLVERINFO` section must contain the line:

```
1 <I PROPERTY="EvolutionOperator" VALUE="AdaptiveSFD" />
```

The adaptive SFD method used is coupled with a stability analysis method. Then `kdim`, `nvec`, `evtol` and `nits` should be defined into the `PARAMETERS` section of `Session.xml`. If not, these parameters will take the default values presented in Sec. 10.4.

The goal of running the stability analysis is to evaluate the dominant eigenvalue of a "partially converged" steady base flow. This approximation is then used by the steady-state solver to select a control coefficient χ and a filter width Δ then ensure a fast convergence towards a steady-state solution.

To define the linear stability problem, another file, that must be called `Session_LinNS.xml`, has to be defined. This file **must be an exact copy/paste of** `Session.xml`, only three things have to be modified:

1. The boundary conditions must be modified to be homogeneous (*i.e.* equal to zero) at all inflow boundaries.
2. A non-zero function `InitialConditions` has to be defined.
3. A random function `BaseFlow` has to be defined (it will be overwritten all along the solver execution). We recommend it to be a copy of `InitialConditions`.

Once these three files (the `Geometry` in `Session.xml.gz`, the nonlinear problem definition in `Session.xml` and the homogeneous linear problem in `Session_LinNS.xml`) are correctly defined, the adaptive SFD method must be executed using:

```
IncNavierStokesSolver Session.xml.gz Session.xml
```

10.6 Coordinate transformations Session file configuration

This section describes how to include a coordinate transformation to the solution of the incompressible Navier-Stokes equations. In some cases, this approach allows a slightly deformed geometry to be mapped into a geometry with a homogeneous direction, which can be treated using a quasi-3D method. It is also useful for problems with a moving body, where otherwise a moving mesh would have to be employed.

10.6.1 Solver Info

To activate the mapping technique, `SolverType` needs to be set as:

```
<I PROPERTY="SolverType" VALUE="VCSMapping" />
```

Also, there are other optional properties in the `SolverInfo` section:

```
<I PROPERTY="MappingImplicitPressure" VALUE="TRUE"/> <!-- Default = FALSE -->
<I PROPERTY="MappingImplicitViscous" VALUE="TRUE"/>  <!-- Default = FALSE -->

<I PROPERTY="MappingNeglectViscous" VALUE="FALSE"/>  <!-- Default = FALSE -->
```

the first two options determine if the pressure and viscous terms resulting from the coordinate transformation are treated implicitly using an iterative procedure. If the last option is set to true, the viscous terms in the mapping are not computed. This leads to a faster solution, but the effect on the results need to be determined for the specific case. By default, all mapping terms are computed and treated explicitly.

10.6.2 Parameters

When treating the mapping terms implicitly, the following parameters can be set:

```
<P> MappingPressureTolerance    = 1e-8  </P>  <!-- Default = 1e-12  -->
<P> MappingViscousTolerance     = 1e-8  </P>  <!-- Default = 1e-12  -->
<P> MappingPressureRelaxation   = 0.9   </P>  <!-- Default = 1.0    -->
<P> MappingViscousRelaxation    = 0.9   </P>  <!-- Default = 1.0    -->
```

They determine the tolerance of the iterative solution of the equations, and a relaxation parameter which can improve the numerical stability of the method.

10.6.3 Mapping

The particular transformation employed is specified by:

```
<MAPPING TYPE="XYofZ">
    <COORDS>Mapping</COORDS>
    <VEL>MappingVel</VEL>
    <TIMEDEPENDENT>True</TIMEDEPENDENT> <!-- Default is False -->
</MAPPING>
```

where `TIMEDEPENDENT` indicates if the transformation varies with time.

The available values for `TYPE`, and the transformations they represent, are:

Mapping type	$\bar{x}$	$\bar{y}$	$\bar{z}$
`Translation`	$x + f(t)$	$y + g(t)$	$z + h(t)$
`XofZ`	$x + f(z,t)$	y	z
`XofXZ`	$f(x,z,t)$	y	z
`XYofZ`	$x + f(z,t)$	$y + g(z,t)$	z
`XYofXY`	$f(x,y,t)$	$g(x,y,t)$	z
`General`	$f(x,y,z,t)$	$g(x,y,z,t)$	$h(x,y,z,t)$

where $(\bar{x}, \bar{y}, \bar{z})$ are the Cartesian (physical) coordinates and (x, y, z) are the transformed coordinates. Note that for quasi-3D problems, the z coordinate cannot be transformed.

10.6.4 Functions

The function `COORDS` (and `VEL` for time dependent mappings) indicated in the `MAPPING` section need to be defined, for example as:

```
        <FUNCTION NAME="Mapping">
            <E VAR="x" VALUE="x + cos(PI*z)" />
            <E VAR="y" VALUE="y + cos(2*PI*t)" />
        </FUNCTION>

        <FUNCTION NAME="MappingVel">
            <E VAR="vx" VALUE="0.0" />
```

```
            <E VAR="vy" VALUE="-1.0*2*PI*sin(2*PI*t)" />
        </FUNCTION>
```

the transformation defined by these functions need to be valid (non-zero Jacobian). By default, any component of `COORDS` that is not specified is taken as a trivial transformation, e.g. $\bar{x} = x$, and any velocity not specified is considered to be zero.

10.6.5 Boundary conditions

In case of a time-dependent mapping, a moving body boundary condition is available:

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" USERDEFINEDTYPE="MovingBody" VALUE="0" />
        <D VAR="v" USERDEFINEDTYPE="MovingBody" VALUE="0" />
        <D VAR="w" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
     </REGION>
</BOUNDARYCONDITIONS>
```

When using the `MovingBody` boundary condition, the Dirichlet condition is relative to the boundary, while the regular Dirichlet boundary condition is taken in an absolute sense.

All Dirichlet boundary conditions are specified in the Cartesian (physical) space, and are automatically transformed to the computational frame of reference.

Note

Examples of the use of mappings can be found in the test directory `NEKTAR/solvers/IncNavierStokesSolver/Tests`. See for example the files `KovaFlow_3DH1D_P8_16modes_Mapping-implicit.xml` and `CylFlow_Mov_mapping.xml`.

10.7 Adaptive polynomial order Session file configuration

An adaptive polynomial order procedure is available for 2D and Quasi-3D simulations. This procedure consists of the following steps:

- Advance the equations for a determined number of time steps
- Use the sensor defined in equation 9.8 to estimate an error measure (the variable used in the sensor can be specified). The error is defined here as the square of the sensor.
- Use the error to determine if the order in each element should be increased by one, decreased by one, or left unaltered.

- Project the solution in each element to the new polynomial order and use it as an initial condition to restart the equation, repeating all steps a given number of times.

It is important to note that restarting the simulation after the refinement can be an expensive operation (in a typical case 200 times the cost of a single time step). Therefore, the number of steps between successive refinements needs to be carefully chosen, since if this value is too low the procedure becomes inefficient, while if it is too high the refinement might not capture accurately structures that are convected by the flow.

10.7.1 Solver Info

The use of the adaptive polynomial order procedure is enforced through the definition of the `Driver` which has to be `Adaptive`.

10.7.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `NumSteps`: when using the adaptive order procedure, this parameter determines the number of time steps between successive refinements.
- `NumRuns`: this parameter defines the number of times the sequence of solving the equation and refining is performed. Therefore, the total number of time steps in the simulation is $NumSteps \times NumRuns$.
- `AdaptiveMaxModes`: sets the maximum number of modes (in each direction) that can be used in an element during the adaptive procedure. The solution will not be refined beyond this point, even if the error is higher than the tolerance. Default value: 12.
- `AdaptiveMinModes`: sets the minimal number of modes (in each direction) that can be used in an element during the adaptive procedure. Default value: 4.
- `AdaptiveUpperTolerance`: defines a constant tolerance. The polynomial order in an element is increased whenever the error is higher than this value. This can be replaced by a spatially-varying function, as described below. Default value: 10^{-6}.
- `AdaptiveLowerinModes`: defines a constant tolerance. The polynomial order in an element is decreased whenever the error is lower than this value. This can also be replaced by a spatially-varying function. Default value: 10^{-8}.
- `AdaptiveSensorVariable`: integer defining which variable will be considered when calculating the error. For example, if this parameter is set to 1 in the Incompressible Navier-Stokes Solver, the error will be estimated using the v velocity. Default value: 0.

10.7.3 Functions

Spatially varying tolerances can be specified by defining the functions `AdaptiveLowerinModes` and/or `AdaptiveUpperTolerance`. In this case, the tolerance in an element is taken as the average of the function in the quadrature points of the element. If these functions are defined, the values set for the tolerances in the `PARAMETERS` section are ignored.

10.8 Examples

10.8.1 Kovasznay Flow 2D

This example demonstrates the use of the velocity correction o solve the 2D Kovasznay flow at Reynolds number $Re = 40$. In the following we will numerically solve for the two dimensional velocity and pressure fields with steady boundary conditions.

10.8.1.1 Input file

The input for this example is given in the example file `KovaFlow_m8.xml`. The mesh consists of 12 quadrilateral elements.

We will use a 7th-order polynomial expansions ($N = 8$ modes) using the modified Legendre basis and therefore require the following expansion definition.

```
<EXPANSIONS>
  <E COMPOSITE="C[0]" NUMMODES="6" FIELDS="u,v,p" TYPE="MODIFIED" />
</EXPANSIONS>
```

We next specify the solver information for our problem. In particular, we select the velocity correction scheme formulation, using a continuous Galerkin projection. For this scheme, an implicit-explicit ime-integration scheme must be used and we choose one of second order.

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes" />
  <I PROPERTY="SolverType" VALUE="VelocityCorrectionScheme" />
  <I PROPERTY="AdvectionForm" VALUE="Convective" />
  <I PROPERTY="Projection" VALUE="Galerkin" />
  <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2" />
</SOLVERINFO>
```

The key parameters are listed below. Since the problem is unsteady we prescribe the time step and the total number of time steps. We also know the required Reynolds number, but we must prescribe the kinematic viscosity to the solver. We first define a *dummy* parameter for the Reynolds number, and then define the kinematic viscosity as the inverse of this. The value of λ is used when defining the boundary conditions and exact solution. Note that `PI` is a pre-defined constant.

```
<PARAMETERS>
  <P> TimeStep      = 0.001  </P>
  <P> NumSteps      = 100    </P>
```

```
  <P> Re            = 40     </P>
  <P> Kinvis        = 1/Re   </P>
  <P> LAMBDA        = 0.5*Re-sqrt(0.25*Re*Re+4*PI*PI)</P>
</PARAMETERS>
```

We choose to impose a mixture of boundary condition types as defined below.

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="1-exp(LAMBDA*x)*cos(2*PI*y)" />
        <D VAR="v" VALUE="(LAMBDA/2/PI)*exp(LAMBDA*x)*sin(2*PI*y)" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="1">
        <D VAR="u" VALUE="1-exp(LAMBDA*x)*cos(2*PI*y)" />
        <D VAR="v" VALUE="(LAMBDA/2/PI)*exp(LAMBDA*x)*sin(2*PI*y)" />
        <D VAR="p" VALUE="0.5*(1-exp(2*LAMBDA*x))" />
    </REGION>
    <REGION REF="2">
        <N VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <N VAR="p" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Initial conditions are obtained from the file KovaFlow_m8.rst, which is a *Nektar++* field file. This is the output of an earlier simulation, renamed with the extension `rst` to avoid being overwritten, and is used in this case to reduce the integration time necessary to obtain the steady flow.

```
<FUNCTION NAME="InitialConditions">
    <F FILE="KovaFlow_m8.rst" />
</FUNCTION>
```

Note the use of the `F` tag to indicate the use of a file. In contrast, the exact solution is prescribed using analytic expressions which requires the use of the `E` tag.

```
<FUNCTION NAME="ExactSolution">
    <E VAR="u" VALUE="1-exp(LAMBDA*x)*cos(2*PI*y)" />
    <E VAR="v" VALUE="(LAMBDA/2/PI)*exp(LAMBDA*x)*sin(2*PI*y)" />
    <E VAR="p" VALUE="0.5*(1-exp(2*LAMBDA*x))" />
</FUNCTION>
```

10.8.1.2 Running the simulation

Launch the simulation using the following command

```
IncNavierStokesSolver KovaFlow_m8.xml
```

After completing the prescribed 100 time-steps, the difference between the computed solution and the exact solution will be displayed. The actual mantissas may vary slightly, but the overall magnitude should be as shown.

```
L 2 error (variable u) : 3.75296e-07
L inf error (variable u) : 5.13518e-07
L 2 error (variable v) : 1.68897e-06
L inf error (variable v) : 2.23918e-06
L 2 error (variable p) : 1.46078e-05
L inf error (variable p) : 5.18682e-05
```

The output of the simulation is written to `KovaFlow_m8.fld`. This can be visualised by converting it to a visualisation format. For example, to use ParaView, convert the output into VTK format using the tility.

```
FieldConvert KovaFlow.xml KovaFlow.fld KovaFlow.vtu
```

The result should look similar to that shown in Figure 10.3.

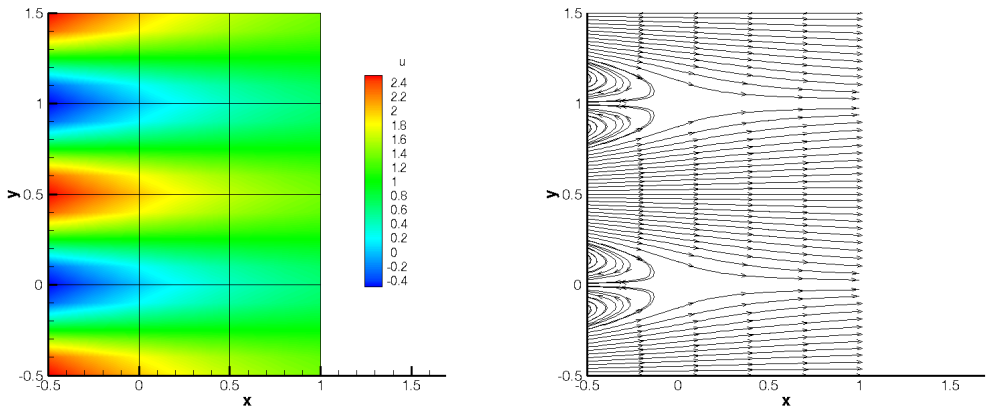


Figure 10.3 Velocity profiles for the Kovasznay Flow (2D).

10.8.2 Kovasznay Flow 2D using high-order outflow boundary conditions

In this example, we solve the same case of 2D Kovasznay flow on severely-truncated computational domain but using a high-order outflow boundary condition, which is much more accurate and robust for unbounded flows [9]. The solver information and parameters used here are similar to the previous one. What only we need to modify in the input file is just the boundary condition type upon the outlet region shown as below

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="1-exp(KovLam*x)*cos(2*PI*y)" />
        <D VAR="v" VALUE="(KovLam/2/PI)*exp(KovLam*x)*sin(2*PI*y)" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
```

```
    <REGION REF="1">
        <N VAR="u" USERDEFINEDTYPE="HOutflow"
              VALUE="-Kinvis*KovLam*exp(KovLam*x)*cos(2*PI*y)
                - 0.5*(1-exp(2*KovLam*x))-0.5*(((1-exp(KovLam*x)*cos(2*PI*y))
                *(1-exp(KovLam*x)*cos(2*PI*y))+(KovLam/(2*PI)*exp(KovLam*x)
                *sin(2*PI*y))*(KovLam/(2*PI)*exp(KovLam*x)*sin(2*PI*y))))
                *(0.5*(1.0-tanh((1-exp(KovLam*x)*cos(2*PI*y))*20)))" />
        <N VAR="v" USERDEFINEDTYPE="HOutflow"
              VALUE="Kinvis*KovLam*KovLam/(2*PI)*exp(KovLam*x)*sin(2*PI*y)" />
        <D VAR="p" USERDEFINEDTYPE="HOutflow"
              VALUE="-Kinvis*KovLam*exp(KovLam*x)*cos(2*PI*y)
                - 0.5*(1-exp(2*KovLam*x)) -0.5*(((1-exp(KovLam*x)*cos(2*PI*y))
                *(1-exp(KovLam*x)*cos(2*PI*y))+(KovLam/(2*PI)*exp(KovLam*x)
                *sin(2*PI*y))*(KovLam/(2*PI)*exp(KovLam*x)*sin(2*PI*y))))
                *(0.5*(1.0-tanh((1-exp(KovLam*x)*cos(2*PI*y))*20)))" />
    </REGION>
    <REGION REF="2">
        <N VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <N VAR="p" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

We note that in this example the "VALUE" property is set based on the analytic solution but this is not typically known and so often a VALUE of zero will be specified.

Instead of loading an initial condition from a specified file, we initialized the flow fields in this example by using following expressions

```
<FUNCTION NAME="InitialConditions">
    <E VAR="u" VALUE="(1-exp(KovLam*x)*cos(2*PI*y))" />
    <E VAR="v" VALUE="(KovLam/(2*PI))*exp(KovLam*x)*sin(2*PI*y)" />
    <E VAR="p" VALUE="0.5*(1-exp(2*KovLam*x))" />
</FUNCTION>
```

10.8.2.1 Running the simulation

We then launch the simulation by the same solver as that in the previous example

```
IncNavierStokesSolver KovaFlow_m8_short_HOBC.xml
```

The solution with errors displayed as below

```
L 2 error (variable u) : 2.51953e-08
L inf error (variable u) : 9.56014e-09
L 2 error (variable v) : 1.10694e-08
L inf error (variable v) : 9.47464e-08
L 2 error (variable p) : 5.59175e-08
L inf error (variable p) : 2.93085e-07
```

The physical solution visualized in velocity profiles is also illustrated in Figure 10.4.

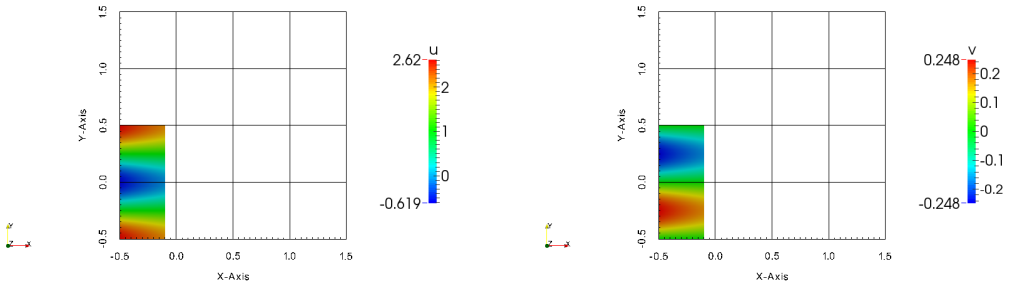


Figure 10.4 Velocity profiles for the Kovasznay Flow in truncated domain (2D).

10.8.3 Steady Kovasznay Oseen Flow using the direct solver

In this example, we instead compute the steadhy Kovasznay Oseen flow using the direct solver. In contrast to the velocity correction scheme in which we time-step the solution to the final time, the direct solver computes the solution with a single solve.

10.8.3.1 Input file

We can begin with the same input file as for the previous example, but with the following modifications. For reference, the modified version is provided in the example `Oseen_m8.xml`.

In the solver information, we must instead select the Steady-Oseen equation type and choose to use the coupled linearised Navier-Stokes

```
<I PROPERTY="EQTYPE" VALUE="SteadyOseen" />
<I PROPERTY="SolverType" VALUE="CoupledLinearisedNS" />
```

Note

Since we are using a coupled system, we are not solving for the pressure. We should therefore remove all references to the variable `p` in the session. In particular, it should be removed from the `EXPANSIONS`, `VARIABLES`, `BOUNDARYCONDITIONS` and `FUNCTIONS` sections of the file.

Instead of loading an initial condition from file, we can simply prescribe a zero field.

```
<FUNCTION NAME="InitialConditions">
    <E VAR="u" VALUE="0" />
    <E VAR="v" VALUE="0" />
</FUNCTION>
```

We must also provide an advection velocity.

```
<FUNCTION NAME="AdvectionVelocity">
    <E VAR="u" VALUE="(1-exp(-LAMBDA*x)*cos(2*PI*y))" />
    <E VAR="v" VALUE="(-LAMBDA/(2*PI))*exp(-LAMBDA*x)*sin(2*PI*y)" />
</FUNCTION>
```

10.8.3.2 Running the simulation

Run the simulation using

```
IncNavierStokesSolver Oseen_m8.xml
```

The resulting flow field should match the solution from the previous example.

10.8.4 Laminar Channel Flow 2D

In this example, we will simulate the flow through a channel at Reynolds number 1 with fixed boundary conditions.

10.8.4.1 Input file

The input file for this example is given in `ChanFlow_m3_SKS.xml`. The geometry is a square channel with height and length $D = 1$, discretised using four quadrilateral elements. We use a quadratic expansion order, which is sufficient to capture the quadratic flow profile. In this example, we choose to use the skew-symmetric form of the advection term. This is chosen in the solver information section:

```
<I PROPERTY="EvolutionOperator" VALUE="SkewSymmetric" />
```

A first-order time integration scheme is used and we set the time-step and number of time integration steps in the parameters section. We also prescribe the kinematic viscosity $\nu = 1/Re = 1$.

Boundary conditions are defined on the walls (region 0) and at the inflow (regions 1) as Dirichlet for the velocity field and as high-order for the pressure. At the outflow the velocity is left free using Neumann boundary conditions and the pressure is pinned to zero.

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="1">
        <D VAR="u" VALUE="y*(1-y)" />
        <D VAR="v" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="2">
```

```
        <N VAR="u" VALUE="0" />
        <N VAR="v" VALUE="0" />
        <D VAR="p" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Initial conditions are set to zero. The exact solution is a parabolic profile with a pressure gradient dependent on the Reynolds number. This is defined to allow verification of the calculation.

```
<FUNCTION NAME="ExactSolution">
    <E VAR="u" VALUE="y*(1-y)" />
    <E VAR="v" VALUE="0" />
    <E VAR="p" VALUE="-2*Kinvis*(x-1)" />
</FUNCTION>
```

10.8.4.2 Running the solver

```
IncNaverStokesSolver ChanFlow_m3_SKS.xml
```

The error in the solution should be displayed and be close to machine precision

```
L 2 error (variable u) : 4.75179e-16
L inf error (variable u) : 3.30291e-15
L 2 error (variable v) : 1.12523e-16
L inf error (variable v) : 3.32197e-16
L 2 error (variable p) : 1.12766e-14
L inf error (variable p) : 7.77156e-14
```

The solution should look similar to that shown in Figure 10.5.

10.8.5 Laminar Channel Flow 3D

We now solve the incompressible Navier-Stokes equations on a three-dimensional domain. In particular, we solver the three-dimensional equivalent of the previous example. We will also solve the problem in parallel.

Note

In order to run the example, you must have a version of *Nektar++* compiled with MPI. This is the case for the packaged binary distributions.

10.8.5.1 Input file

The input file for this example is given in `Tet_channel_m8_par.xml`. In this example we use tetrahedral elements, indicated by the `A` element tags in the geometry section. All dimensions have length $D = 1$. We will use a 7th-order polynomial expansion. Since

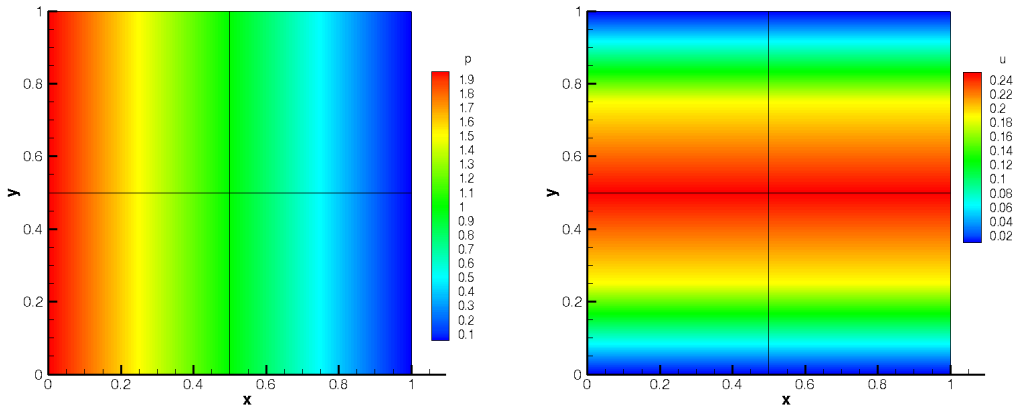


Figure 10.5 Pressure and velocity profiles for the laminar channel flow (2D).

we now have three dimensions, and therefore three velocity components, the expansions section is now

```
<EXPANSIONS>
  <E COMPOSITE="C[0]" NUMMODES="8" FIELDS="u,v,w,p" TYPE="MODIFIED" />
</EXPANSIONS>
```

The solver information and parameters are similar to the previous example. Boundary conditions must now be defined on the six faces of the domain. Flow is prescribed in the z-direction through imposing a Poiseulle profile on the inlet and side walls. The outlet is zero-Neumann and top and bottom faces impose zero-Dirichlet conditions.

```
<BOUNDARYREGIONS>
    <B ID="0"> C[1] </B>    <!-- Inlet -->
    <B ID="1"> C[6] </B>    <!-- Outlet -->
    <B ID="2"> C[2] </B>    <!-- Wall -->
    <B ID="3"> C[3] </B>    <!-- Wall left -->
    <B ID="4"> C[4] </B>    <!-- Wall -->
    <B ID="5"> C[5] </B>    <!-- Wall right -->
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="y*(1-y)" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="1">
        <N VAR="u" VALUE="0" />
        <N VAR="v" VALUE="0" />
        <N VAR="w" VALUE="0" />
        <D VAR="p" VALUE="0" />
```

```
    </REGION>
    <REGION REF="2">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="3">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="y*(1-y)" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="4">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="5">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="y*(1-y)" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Initial conditions and exact solutions are also prescribed.

10.8.5.2 Running the solver

To run the solver in parallel, we use the `mpirun` command.

```
mpirun -np 2 IncNaverStokesSolver Tet_channel_m8_par.xml
```

The expected results are shown in Figure 10.6.

10.8.6 Laminar Channel Flow Quasi-3D

For domains where at least one direction is geometrically homogeneous, a more efficient discretisation is to use a pure spectral discretisation, such as a Fourier expansion, in these directions. We use this approach to solve the same problem as in the previous example. We reuse the two-dimensional spectral/hp element mesh from the nd couple this with a Fourier expansion in the third component.

10.8.6.1 Input file

The input file for this example is `ChanFlow_3DH1D_MVM.xml`. We indicate that we are coupling the spectral/hp element domain with a pure spectral expansion using the following solver information

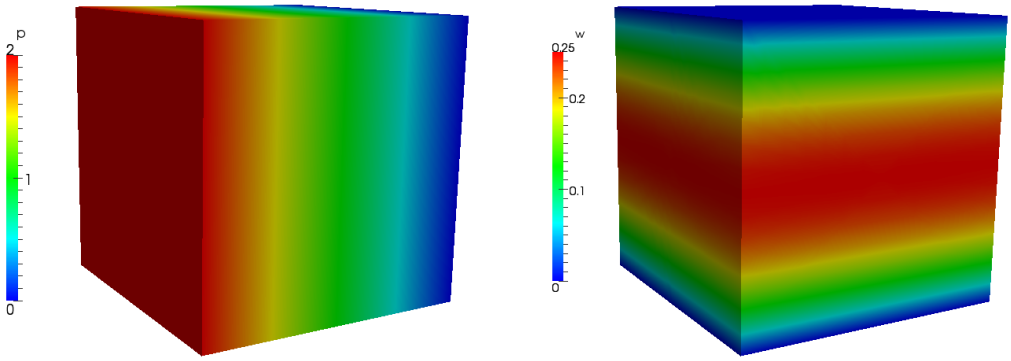


Figure 10.6 Pressure and velocity profiles for the laminar channel flow (full 3D).

```
<I PROPERTY="HOMOGENEOUS" VALUE="1D"/>
```

We must also specify parameters to describe the particular spectral expansion

```
<P> HomModesZ     = 20       </P>
<P> LZ            = 1.0      </P>
```

The parameter `HomModesZ` specifies the number of Fourier modes to use in the homogeneous direction. The `LZ` parameter specifies the physical length of the domain in that direction.

Note

This example uses an in-built Fourier transform routine. Alternatively, one can use the FFTW library to perform Fourier transforms which typically offers improved performance. This is enabled using the following solver information

```
<I PROPERTY="USEFFT" VALUE="FFTW"/>
```

As with the the spectral/hp element mesh consists of four quadrilateral elements with a second-order polynomial expansion. Since our domain is three-dimensional we have to now include the third velocity component

```
<E COMPOSITE="C[0]" NUMMODES="3" FIELDS="u,v,w,p" TYPE="MODIFIED" />
```

The remaining parameters and solver information is similar to previous examples.

Boundary conditions are specified as for the two-dimensional case (except with the addition of the third velocity component) since the side walls are now implicitly periodic. The initial conditions and exact solution are prescribed as for the fully three-dimensional case.

10.8.6.2 Running the solver

```
IncNaverStokesSolver ChanFlow_3DH1D_MVM.xml
```

The results can be post-processed and should match those of the fully three-dimensional case as shown in Figure 10.6.

10.8.7 Turbulent Channel Flow

In this example we model turbulence in a three-dimensional square channel at a Reynolds number of 2000.

Note

This example requires the FFTW Fast-Fourier transform library to be selected when compiling Nektar++.

10.8.7.1 Input file

The input file for this example is `TurbChFl_3DH1D.xml`. The geometry makes used of the homogeneous extension discussed in the previous example. The channel has height $D = 2$ and length $L = 4\pi$ and is discretised using quadratic quadrilateral elements in the spectral/hp element plane and a Fourier basis in the third coordinate direction. The elements are non-uniformly distributed so as to best capture the flow features with fewest degrees of freedom and is shown in Figure 10.7.

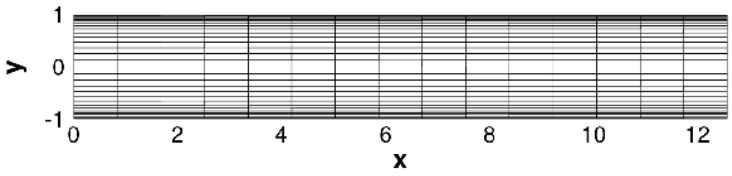


Figure 10.7 Mesh used for the turbulent channel flow.

The spanwise length of the channel is set using the `LZ` parameter and discretised with 32 Fourier modes by setting the value of `HomModesZ`.

```
<P> HomModesZ     = 32          </P>
<P> LZ            = 4*PI/3      </P>
```

A second-order IMEX scheme is used for time-integration scheme is used with a time-step of 0.0001. The length of the simulation is 1 time-unit (10,000 steps).

Periodicity is naturally enforced in the spanwise direction, so boundary conditions need only be provided for the upper and lower walls, inlet and outlet as denoted by the following `BOUNDARYREGIONS`.

```
<BOUNDARYREGIONS>
  <B ID="0"> C[1] </B> //walls
  <B ID="1"> C[2] </B> //inflow
  <B ID="2"> C[3] </B> //outflow
</BOUNDARYREGIONS>
```

In this example, we will use a body force to drive the flow and so, in addition to the spanwise periodicity, enforce periodicity in the streamwise direction of the spectral/hp element mesh. This is achieved by imposing the following boundary conditions

```
  <REGION REF="1">
    <P VAR="u" VALUE="[2]" />
    <P VAR="v" VALUE="[2]" />
    <P VAR="w" VALUE="[2]" />
    <P VAR="p" VALUE="[2]" />
  </REGION>
  <REGION REF="2">
    <P VAR="u" VALUE="[1]" />
    <P VAR="v" VALUE="[1]" />
    <P VAR="w" VALUE="[1]" />
    <P VAR="p" VALUE="[1]" />
  </REGION>
```

Here, we use `P` to denote the boundary type is periodic, and the value in square brackets denotes the boundary region to which the given boundary is periodic with. In this case regions 1 and 2 are denoted periodic with each other.

A streamwise plug-profile initial condition is prescribed such that $u = 1$ everywhere, except the wall boundaries. The body force requires two components in the XML file. The first specifies the type of forcing to apply and appears directly within the `NEKTAR` tag.

```
<FORCING>
    <FORCE TYPE="Body">
        <BODYFORCE> BodyForce </BODYFORCE>
    </FORCE>
</FORCING>
```

The second defines the `BodyForce` function which will be used and is located within the `CONDITIONS` section,

```
<FUNCTION NAME="BodyForce">
    <E VAR="u" VALUE="2*Kinvis" />
    <E VAR="v" VALUE="0" />
    <E VAR="w" VALUE="0" />
```

```
</FUNCTION>
```

To improve numerical stability, we also enable dealising of the advection term. This uses additional points to perform the quadrature and then truncates the higher-order terms when projecting back onto the polynomial space, thereby removing spurious oscillations. It is enabled by setting the solver information tag

```
<I PROPERTY="DEALIASING" VALUE="ON" />
```

This feature is only available when using the FFTW library is used, so we enable this using

```
<I PROPERTY="USEFFT" VALUE="FFTW" />
```

10.8.7.2 Running the solver

To run the solver, we use the following command

```
IncNaverStokesSolver TurbChFl_3DH1D.xml
```

The result after transition has occurred is illustrated in Figure 10.8.

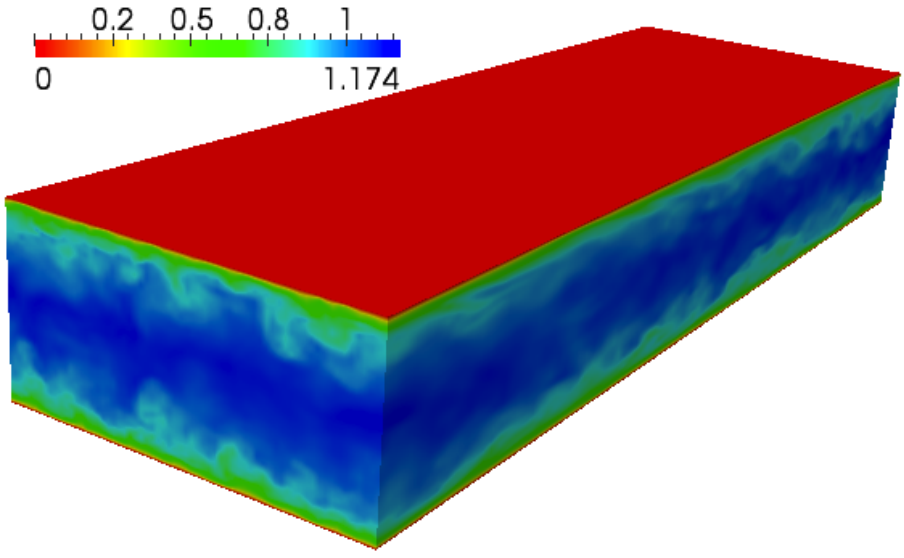


Figure 10.8 Velocity profile of the turbulent channel flow (quasi-3D).

10.8.8 Turbulent Pipe Flow

In this example we simulate flow in a pipe at Reynolds number 3000 using a mixed spectral/hp element and Fourier discretisation. The Fourier expansion is used in the

streamwise direction in this case and the spectral/hp elements are used to capture the circular cross-section.

10.8.8.1 Input File

The circular pipe has diameter $D = 1$, the 2D mesh is composed of 64 quadrilateral elements and the streamwise direction is discretised with 128 Fourier modes. An illustrative diagram of the discretisation is given in Figure 10.9.

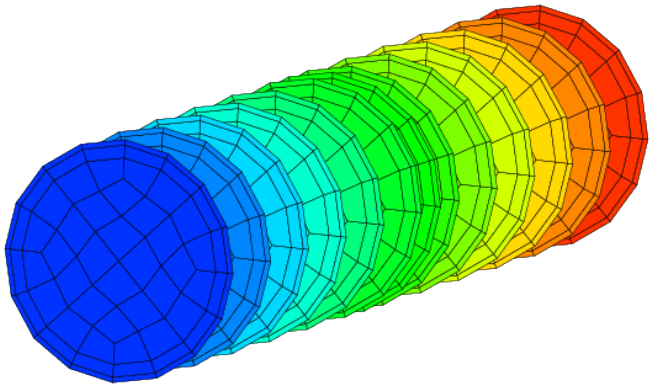


Figure 10.9 Domain for the turbulent pipe flow problem.

The input file for this example is `Pipe_turb.xml`. We use 7th-order lagrange polynomials through the Gauss-Lobatto-Legendre points for the quadrilateral expansions.

```
1 <E COMPOSITE="C[0]" NUMMODES="8" FIELDS="u,v,w,p" TYPE="GLL_LAGRANGE_SEM" />
```

We set the Fourier options, as in the previous example, except using 128 modes and a length of 5 non-dimensional units. A small amplitude noise is also added to the initial condition, which is a plug profile, to help stimulate transition. Since the streamwise direction is the Fourier direction, we must necessarily use a body force to drive the flow.

10.8.8.2 Running the solver

In this example we will run the solver in parallel. Due to the large number of Fourier modes and relatively few elements, it is more efficient to parallelise in the streamwise direction. We can specify this by providing an additional flag to the solver, `-npz`. This indicates the number of partitions in the z-coordinate. In this example, we will only run two processes. We therefore would specify `-npz 2` to ensure parallelisation only occurs in the Fourier direction.

To improve the efficiency of the solver further, we would prefer to solve the Helmholtz problems within the spectral/hp element planes using a direct solver (since no paralleli-

sation is necessary). The default when running in parallel is to use an iterative solver, so we explicitly specify the type of algorithm to use in the session file solver information:

```
1 <I PROPERTY="GlobalSysSoln" VALUE="DirectStaticCond" />
```

The solver can now be run as follows

```
mpirun -np 2 IncNavierStokesSolver --npz 2 Pipe_turb.xml
```

When the pipe transitions, the result should look similar to Figure 10.10.

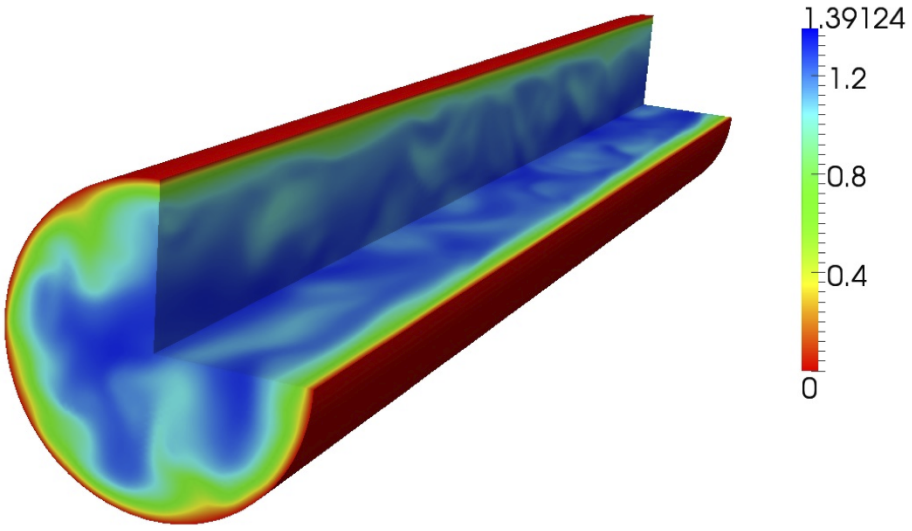


Figure 10.10 Velocity profile of the turbulent pipe flow (quasi-3D).

10.8.9 Aortic Blood Flow

The following example demonstrates the application of the incompressible Navier-Stokes solver using the Velocity Correction Scheme algorithm for modelling viscid Newtonian blood flow in a region of a rabbit descending thoracic aorta with intercostal branch pairs. Such studies are necessary to understand the effect local blood flow changes have on cardiovascular diseases such as atherosclerosis.

In the following we will numerically solve for the three dimensional velocity and pressure field for steady boundary conditions. The Reynolds number under consideration is 300, which is physiologically relevant.

Geometry

The geometry under consideration is a segment of a rabbit descending aorta with two pairs of intercostal arteries branching off. The inlet has a diameter $D = 3.32mm$.

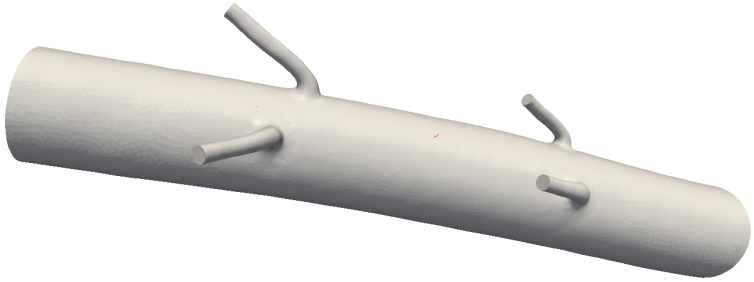


Figure 10.11 Reduced region of rabbit descending thoracic aorta.

In order to capture the physics of the flow in the boundary layer, a thin layer consisting of prismatic elements is created adjacent to the surface, and curved using spherigons. The interior consist of tetrahedral elements.

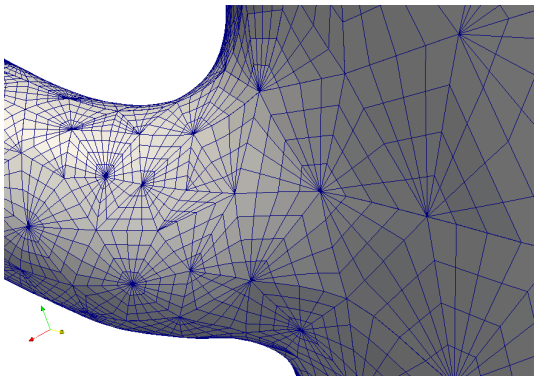


Figure 10.12 Surface mesh indicating curved surface elements at a branch location.

Input parameters

10.8.9.1 Expansion:

In this example we will use a fourth order polynomial expansion. There are two composites defined here since we have both prismatic and tetrahedral elements.

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="5" TYPE="MODIFIED" FIELDS="u,v,w,p" />
    <E COMPOSITE="C[1]" NUMMODES="5" TYPE="MODIFIED" FIELDS="u,v,w,p" />
</EXPANSIONS>
```

10.8.9.2 Solver information:

```
<SOLVERINFO>
    <I PROPERTY="SolverType" VALUE="VelocityCorrectionScheme" />
    <I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes" />
    <I PROPERTY="AdvectionForm" VALUE="Convective" />
```

```
    <I PROPERTY="Projection" VALUE="Galerkin" />
    <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder1" />
</SOLVERINFO>
```

10.8.9.3 Parameters:

Since we are prescribing a Reynolds number of 300, and to simplify the problem definition, we set the mean inlet velocity to 1, this allows us to define the kinematic viscosity as $\nu = \frac{UD}{Re} = \frac{3.32}{300} = 1/90.36$.

```
<PARAMETERS>
    <P> TimeStep       = 0.0005     </P>
    <P> NumSteps       = 1600       </P>
    <P> IO_CheckSteps  = 200        </P>
    <P> IO_InfoSteps   = 50         </P>
    <P> Kinvis         = 1.0/90.36  </P>
</PARAMETERS>
```

10.8.9.4 Boundary conditions:

For the purpose of this example a blunted inlet velocity profile has been prescribed. Ideally to obtain more significant results, the velocity profile at the inlet would be obtained from previous simulations on the complete rabbit aorta (including aortic root, aortic arch, and descending aorta with all 5 pairs of intercostal arteries), where a blunted profile at the aortic root is a better representation of reality.

Dirichlet boundary conditions are imposed for the velocity at the inlet, as well as on the wall to account for the no-slip condition. Neumann boundary conditions are imposed for the velocity field at the outlets where fully developed flow is imposed.

```
<BOUNDARYREGIONS>
    <B ID="0"> C[2] </B>          <!-- Inlet -->
    <B ID="1"> C[3,4,5,6] </B>    <!-- intercostal outlets -->
    <B ID="2"> C[7] </B>          <!-- outlet -->
    <B ID="3"> C[8] </B>          <!-- wall -->
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="0.024" />
        <D VAR="v" VALUE="-0.064" />
        <D VAR="w" VALUE="-0.998" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
    <REGION REF="1">
        <N VAR="u" VALUE="0" />
        <N VAR="v" VALUE="0" />
        <N VAR="w" VALUE="0" />
        <D VAR="p" VALUE="0" />
    </REGION>
    <REGION REF="2">
        <N VAR="u" VALUE="0" />
```

```
        <N VAR="v" VALUE="0" />
        <N VAR="w" VALUE="0" />
        <D VAR="p" VALUE="0" />
    </REGION>
    <REGION REF="3">
        <D VAR="u" VALUE="0" />
        <D VAR="v" VALUE="0" />
        <D VAR="w" VALUE="0" />
        <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

10.8.9.5 Functions:

```
<FUNCTION NAME="InitialConditions">
    <E VAR="u" VALUE="0" />
    <E VAR="v" VALUE="0" />
    <E VAR="w" VALUE="0" />
    <E VAR="p" VALUE="0" />
</FUNCTION>
```

10.8.9.6 Results

We can visualise the internal velocity field by applying a volume render filter in ParaView.

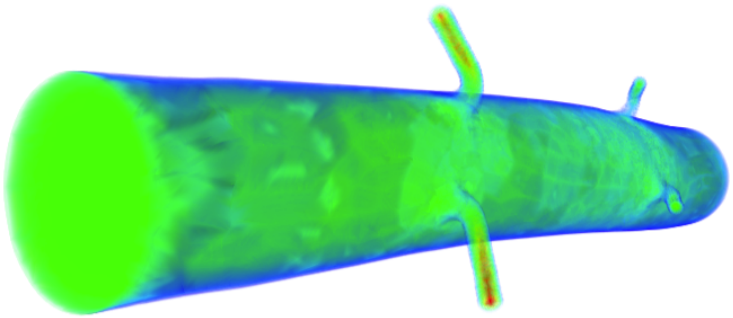


Figure 10.13 The solved-for velocity field.

It is possible to visualise the wall shear stress distribution by running the FldAddWSS utility.

10.8.10 finite-strip modeling of flow past flexible cables

As a computationally efficient model, strip theory-based modeling technique has been proposed previously to predict vortex-induced vibration (VIV) for higher Reynolds number flows. In the strip theory-based model, the fluid flow solution is obtained on a series of 2D computational planes (also called as “strips”) along the riser’s axis direction. These strips then are coupled with each other through structural dynamic model of the

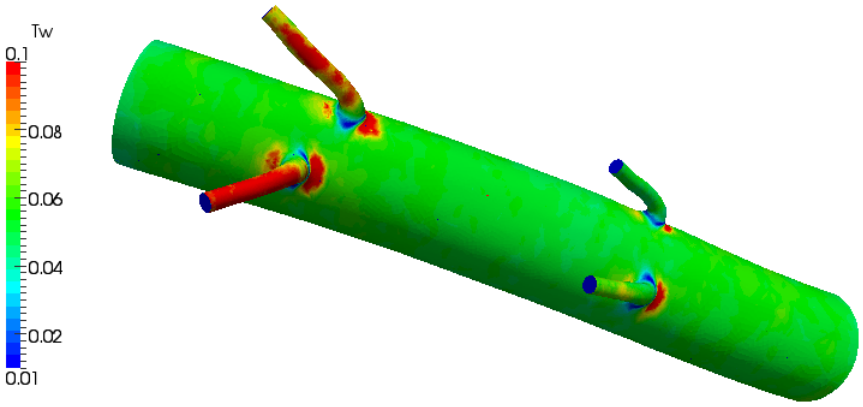


Figure 10.14 Non-dimensional wall shear stress distribution.

riser, and then VIV response prediction is achieved by the strip-structure interactions.In the 2D strip theory, it is assumed that the flow is purely two-dimensional without spanwise correlation, which allows the problems to be split into various 2D planes. A consequence of 2D strip solution under this assumption is that it is unable to reflect the influence of spanwise wake turbulence on the structural dynamics. In order to overcome this shortcoming, we proposed a new module in the framework of Nektar++, in which a spanwise scale is locally allocated to each one of the strips, so that the spanwise velocity correlation is reconstructed in the flow field within each strips. In particular, this model lets the fluid domain to be divided in N strips with thickness ratio of L_z/D and evenly distributed along the spanwise (z) direction. The gap between the neighboring strips, represented by L_g, satisfies relation $L_c = N(L_z + L_g)$. Since the strip in this model has finite scale in the z-direction, we named it as finite strip to distinguish from traditional 2D strip plane. Next, the flow dynamics within each individual strips are modeled by viscous incompressible Navier-Stokes equations, while a tensioned beam model is employed to govern the dynamics of the flexible structures. In this example, we will show how to perform a finite-strip model to predict the vortex-induced vibration responses of flexible cables. Let us consider a vortex-induced vibration of a slender cable with an aspect ratio of L_z/D=4π, which is immersed in uniform flows at Re=100.

10.8.10.1 Input File

The cable with a mass ratio (defined as the ratio of the total oscillating mass to the mass of displaced fluid) of 1 has diameter $D = 1$, the 2D mesh is composed of 284 quadrilateral elements. The spanwise direction is split in 16 strips with thickness ratio of L_c/D=$\pi/8$ and one pair of complex Fourier modes for each one of the strips. We will use a sixth order polynomial expansion for the spectral element and the input file for this example is `CylFlow_HomoStrip.xml`.

```
1 <E COMPOSITE="C[73]" NUMMODES="6" TYPE="MODIFIED" FIELDS="u,v,w,p" />
```

To use the finite strip routines we need just to insert a flag of `"HomoStrip"` in the solver information as below, in addition, we need to specify the types of vibration and support ends for the cables. In this case, the vibration type is specified as `VALUE="CONSTRAINED"`, which means that the cable's vibration is constrained only in the crossflow direction. Other options include `VALUE="FREE"` and `"FORCED"`, respectively corresponding to the free vibrations in both streamwise and crossflow directions and forced vibration by specified functions given in input file. For the support ends of the cable, another option of `VALUE="PINNED-PINNED"` is available for the simulations, which satisfies the condition of zero values of displacements on the support ends.

10.8.10.2 Solver information:

```
        <SOLVERINFO>
            <I PROPERTY="HomoStrip" VALUE="True"/>
            <I PROPERTY="VibrationType" VALUE="CONSTRAINED"/>
            <I PROPERTY="SupportType" VALUE="FREE-FREE"/>
        </SOLVERINFO>
```

10.8.10.3 Parameters

All the simulation parameters are specified in the section as follows.

```
        <PARAMETERS>
            <P> LZ            = PI/8            </P> <!--thickness ratio-->
            <P> LC            = 4*PI            </P> <!--aspect ratio-->
            <P> A             = 0.025           </P>
            <P> omega         = 1.0             </P>
            <P> PROC_Z        = 16              </P>
            <P> Strip_Z       = 16              </P> <!--number of the strips-->
            <P> DistStrip     = PI/4            </P> <!--distance of the strips-->
            <P> StructStiff   = 0.02            </P>
            <P> StructRho     = 2.0             </P>
            <P> CableTension  = 8.82            </P>
            <P> BendingStiff  = 0.0             </P>
            <P> FictDamp      = 0.0             </P>
            <P> FictMass      = 3.0             </P>
        </PARAMETERS>

```

10.8.10.4 Running the solver

In this example we will run the solver in parallel. We can specify the number of the strips by providing an additional flag to the solver, –nsz. In this example, we will run 16 strips, therefore it would be specified as –nsz 16. The solver can now be run as follows

```
mpirun -np 16 IncNavierStokesSolver  CylFlow_HomoStrip.xml --npz 16 --nsz 16
```

The simulation results are illustrated in spanwise vorticity contours in Figure 10.15. The wake response of the cable appears as standing wave patter in the earlier stage and then it transitions into traveling wave response, as shown in this figure.

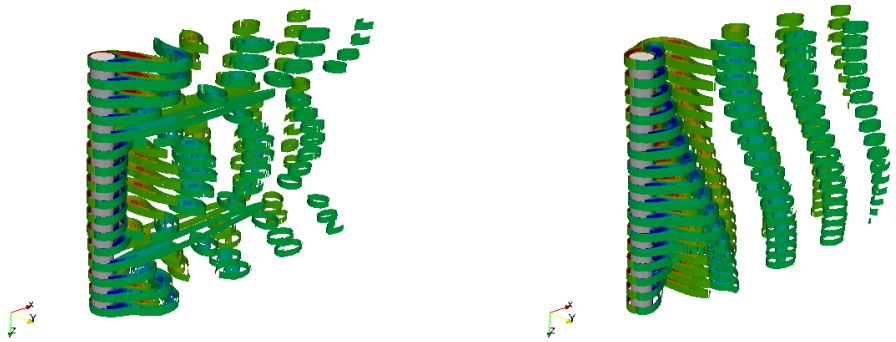


Figure 10.15 Spanwise vorticity contours in standing wave and traveling wave patterns predicted in finite strip modeling.

10.8.11 2D direct stability analysis of the channel flow

In this example, it will be illustrated how to perform a direct stability analysis using Nektar++. Let us consider a canonical stability problem, the flow in a channel which is confined in the wall-normal direction between two infinite plates (Poiseuille flow) at Reynolds number 7500. This problem is a particular case of the stability solver for the IncNavierStokesSolver.

10.8.11.1 Background

We consider the linearised Incompressible Navier-Stokes equations:

$$\frac{\partial \mathbf{u}'}{\partial t} + \mathbf{U} \cdot \nabla \mathbf{u}' + \mathbf{u}' \cdot \nabla \mathbf{U} = -\nabla p + \nu \nabla^2 \mathbf{u}' + \mathbf{f} \tag{10.21a}$$

$$\nabla \cdot \mathbf{u}' = 0 \tag{10.21b}$$

We are interested to compute the leading eigenvalue of the system using the Arnoldi method.

10.8.11.2 Geometry

The geometry under consideration is a 2D channel.

10.8.11.3 Mesh Definition

In the `GEOMETRY` section, the dimensions of the problem are defined. Then, the coordinates (`XSCALE`, `YSCALE`, `ZSCALE`) of each vertices of each element are specified. As this input

file defines a two-dimensional problem: `ZSCALE = 0`.

```
<GEOMETRY DIM="2" SPACE="2">
        <VERTEX>
            <V ID="0">3.142e+00 1.000e+00 0.000e+00</V>
            ...
            <V ID="62">-3.142e+00 -1.000e+00 0.000e+00</V>
        </VERTEX>
```

Edges can now be defined by two vertices.

```
<EDGE>
            <E ID="0">    0  1   </E>
            ...
            <E ID="109">   62  55   </E>
        </EDGE>
```

In the `ELEMENT` section, the tag `T` and `Q` define respectively triangular and quadrilateral element. Triangular elements are defined by a sequence of three edges and quadrilateral elements by a sequence of four edges.

```
        <ELEMENT>
            <Q ID="0">    0     1     2     3 </Q>
            ...
            <Q ID="47">  107   108   109    95 </Q>
        </ELEMENT>

```

Finally, collections of elements are listed in the `COMPOSITE` section and the `DOMAIN` section specifies that the mesh is composed by all the triangular and quadrilateral elements. The other composites will be used to enforce boundary conditions.

```
              <COMPOSITE>
            <C ID="0"> Q[0-47] </C>
            <C ID="1"> E[17,31,44,57,70,83,96,109,0,19,32,45,58,71,84,97] </C> //wall
            <C ID="2"> E[3,6,9,12,15,18] </C>//inflow
            <C ID="3"> E[98,100,102,104,106,108] </C> //outflow
        </COMPOSITE>
        <DOMAIN> C[0] </DOMAIN>
</GEOMETRY>

```

10.8.11.4 Expansion

This section defines the polynomial expansions used on each composites. For this example we will use a 10th order polynomial, i.e. $P = 11$.

```
<EXPANSIONS>
     <E COMPOSITE="C[0]" NUMMODES="11" FIELDS="u,v,p" TYPE="MODIFIED" />
</EXPANSIONS>

```

10.8.11.5 Solver Info

In this example the `EvolutionOperator` must be `Direct` to consider the linearised Navier-Stokes equations and the `Driver` was set up to `ModifiedArnoldi` for the solution of the eigenproblem.

```
      <SOLVERINFO>
      <I PROPERTY="SolverType"        VALUE="VelocityCorrectionScheme"/>
      <I PROPERTY="EQTYPE"            VALUE="UnsteadyNavierStokes"/>
      <I PROPERTY="EvolutionOperator" VALUE="Direct"/>
      <I PROPERTY="Projection"        VALUE="Galerkin"/>
      <I PROPERTY="Driver"            VALUE= "ModifiedArnoldi" />
      <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder1" />
    </SOLVERINFO>

```

10.8.11.6 Parameters

All the stability parameters are specified in this section.

```
<PARAMETERS>
      <P> TimeStep      = 0.002 </P>
      <P> NumSteps      = 500 </P>
      <P> IO_CheckSteps = 1000    </P>
      <P> IO_InfoSteps  = 10    </P>
      <P> Re            = 7500     </P>
      <P> Kinvis         =1./Re   </P>
      <P> kdim           =16   </P>
      <P> nvec           =2   </P>
      <P> evtol          =1e-5</P>
      <P> nits           =5000 </P>
   </PARAMETERS>

```

10.8.11.7 Boundary Conditions

```
<BOUNDARYREGIONS>
           <B ID="0"> C[1] </B>
           <B ID="1"> C[2] </B>
           <B ID="2"> C[3] </B>
       </BOUNDARYREGIONS>

       <BOUNDARYCONDITIONS>
           <REGION REF="0">
               <D VAR="u" VALUE="0" />
               <D VAR="v" VALUE="0" />
               <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
           </REGION>
           <REGION REF="1">
               <P VAR="u" VALUE="[2]" />
               <P VAR="v" VALUE="[2]" />
               <P VAR="p" VALUE="[2]" />
           </REGION>
           <REGION REF="2">
```

```
                <P VAR="u" VALUE="[1]" />
                <P VAR="v" VALUE="[1]" />
                <P VAR="p" VALUE="[1]" />
            </REGION>
        </BOUNDARYCONDITIONS>

```

10.8.11.8 Function

We need to set up the base flow that can be specified as a function `BaseFlow`. In case the base flow is not analytical, it can be generated by means of the `Nonlinear` evolution operator using the same mesh and polynomial expansion. The initial guess is specified in the `InitialConditions` functions and can be both analytical or a file. In this example it is read from a file.

```
<FUNCTION NAME="BaseFlow">
            <F VAR="u,v,p" FILE="ChanStability.bse" />
        </FUNCTION>

        <FUNCTION NAME="InitialConditions">
            <F VAR="u,v,p" FILE="ChanStability.rst" />
        </FUNCTION>

```

10.8.11.9 Usage

```
IncNavierStokesSolver ChanStability.xml
```

10.8.11.10 Results

The stability simulation takes about 250 iterations to converge and the dominant eigenvalues (together with the respective eigenvectors) will be printed. In this case it was found $\lambda_{1,2} = 1.000224 \times e^{\pm 0.24984i}$. Therefore, since the magnitude of the eigenvalue is larger than 1, the flow is absolutely unstable. It is possible to visualise the eigenvectors using the post-processing utilities. The figure shows the profile of the two eigenmode component, which shows the typical Tollmien - Schlichting waves that arise in viscous boundary layers.

10.8.12 2D adjoint stability analysis of the channel flow

In this example, it will be illustrated how to perform an adjoint stability analysis using Nektar++. Let us consider a canonical stability problem, the flow in a channel which is confined in the wall-normal direction between two infinite plates (Poiseuille flow) at Reynolds number 7500

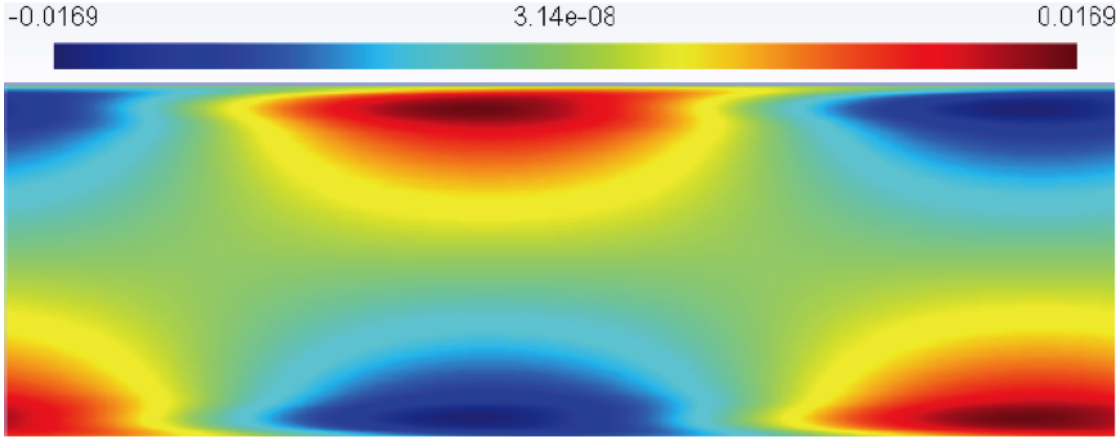


Figure 10.16

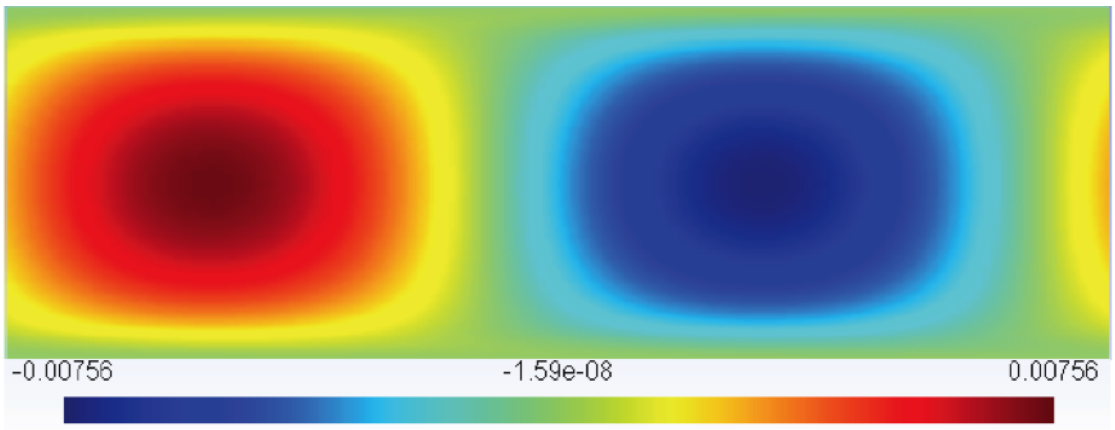


Figure 10.17

10.8.12.1 Background

We consider the equations:

$$-\frac{\partial \mathbf{u}^*}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{u}^* + (\nabla\mathbf{U})^T\cdot\mathbf{u}^* = -\nabla p^* + \frac{1}{Re}\nabla^2\mathbf{u} \tag{10.22a}$$

$$\nabla\cdot\mathbf{u}^* = 0 \tag{10.22b}$$

We are interested in computing the leading eigenvalue of the system using the Arnoldi method.

10.8.12.2 Geometry & Mesh

The geometry and mesh are the same ones used for the direct stability analysis in the previous example.

10.8.12.3 Solver Info

This sections defines the problem solved. In this example the `EvolutionOperator` must be `Adjoint` to consider the adjoint Navier-Stokes equations and the `Driver` was set up to `ModifiedArnoldi` for the solution of the eigenproblem.

```
<SOLVERINFO>
      <I PROPERTY="SolverType"        VALUE="VelocityCorrectionScheme"/>
      <I PROPERTY="EQTYPE"            VALUE="UnsteadyNavierStokes"/>
      <I PROPERTY="EvolutionOperator" VALUE="Adjoint"/>
      <I PROPERTY="Projection"        VALUE="Galerkin"/>
      <I PROPERTY="Driver"            VALUE= "ModifiedArnoldi" />
      <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder1" />
</SOLVERINFO>
  \end{subequations}

\textbf{Parameters}

    \begin{lstlisting}[style=XMLStyle]
<PARAMETERS>
      <P> TimeStep      = 0.002 </P>
      <P> NumSteps      = 500 </P>
      <P> IO_CheckSteps = 1000    </P>
      <P> IO_InfoSteps  = 10    </P>
      <P> Re            = 7500     </P>
      <P> Kinvis         =1./Re   </P>
      <P> kdim           =16   </P>
      <P> nvec           =2   </P>
      <P> evtol          =1e-5</P>
      <P> nits           =5000 </P>
</PARAMETERS>

```

10.8.12.4 Boundary Conditions

```
 <BOUNDARYREGIONS>
            <B ID="0"> C[1] </B>
            <B ID="1"> C[2] </B>
            <B ID="2"> C[3] </B>
        </BOUNDARYREGIONS>

        <BOUNDARYCONDITIONS>
            <REGION REF="0">
                <D VAR="u" VALUE="0" />
                <D VAR="v" VALUE="0" />
                <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
            </REGION>
            <REGION REF="1">
                <P VAR="u" VALUE="[2]" />
```

```
                <P VAR="v" VALUE="[2]" />
                <P VAR="p" VALUE="[2]" />
            </REGION>
            <REGION REF="2">
                <P VAR="u" VALUE="[1]" />
                <P VAR="v" VALUE="[1]" />
                <P VAR="p" VALUE="[1]" />
            </REGION>
        </BOUNDARYCONDITIONS>

```

10.8.12.5 Functions

We need to set up the base flow that can be specified as a function `BaseFlow`. In case the base flow is not analytical, it can be generated by means of the `Nonlinear` evolution operator using the same mesh and polynomial expansion.

```
        <FUNCTION NAME="BaseFlow">
            <F VAR="u,v,p" FILE="ChanStability.bse" />
        </FUNCTION>

```

The initial guess is specified in the `InitialConditions` functions and can be both analytical or a file. In this example it is read from a file.

```
        <FUNCTION NAME="InitialConditions">
            <F VAR="u,v,p" FILE="ChanStability.rst" />
        </FUNCTION>

```

10.8.12.6 Usage

```
IncNavierStokesSolver ChanStability_adj.xml
```

10.8.12.7 Results

The equations will then be evolved backwards in time (consistently with the negative sign in front of the time derivative) and the leading eigenvalues will be computed after about 300 iterations. It is interesting to note that their value is the same one computed for the direct problem, but the eigenmodes present a different shape.

10.8.13 2D Transient Growth analysis of a flow past a backward-facing step

In this section it will be described how to perform a transient growth stability analysis using Nektar++. Let us consider a flow past a backward-facing step (figure 10.20). This is an important case because it allows us to understand the effects of separation caused by abrupt changes in the geometry and it is a common geometry in several studies of flow control and turbulence of separated flow.

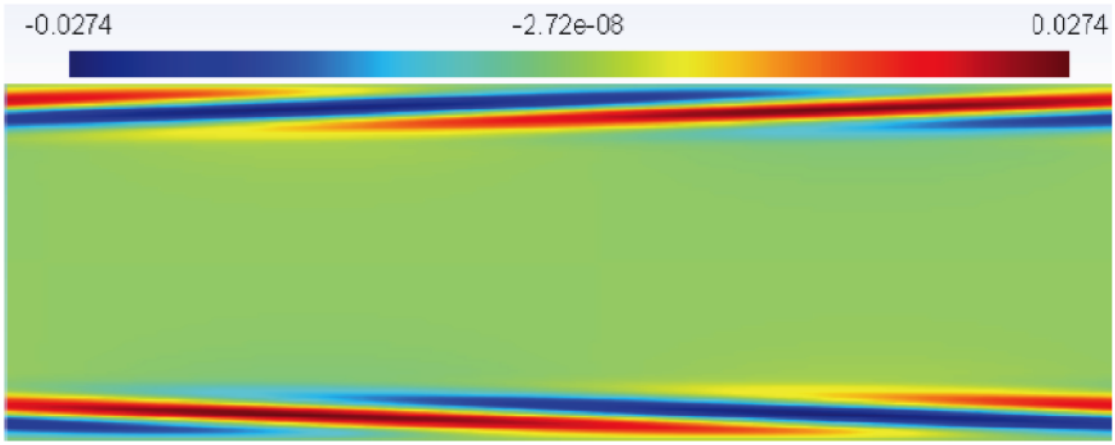


Figure 10.18

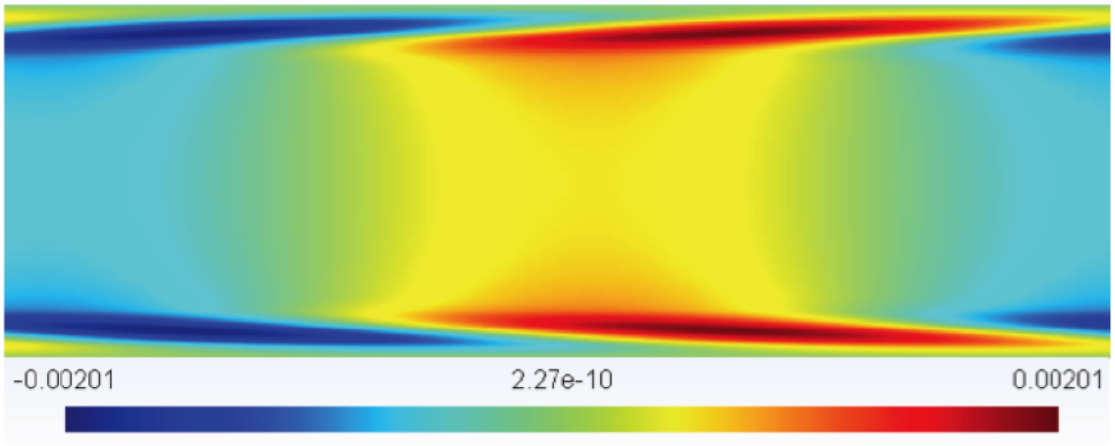


Figure 10.19

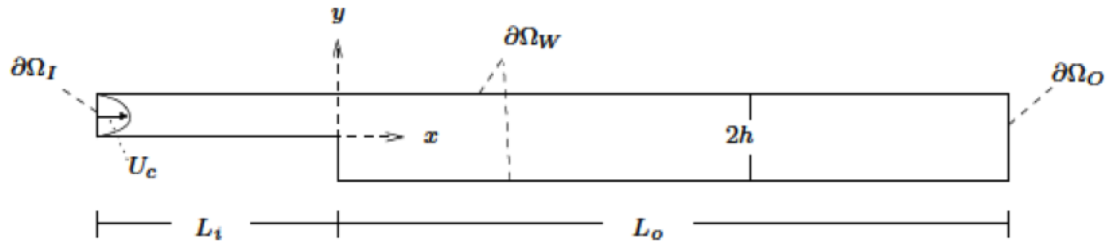


Figure 10.20

10.8.13.1 Background

Transient growth analysis allows us to study the presence of convective instabilities that can arise in stable flows. Despite the fact that these instabilities will decay for a long time (due to the stability of the flow), they can produce significant increases in the energy of perturbations. The phenomenon of transient growth is associated with the non-normality of the linearised Navier-Stokes equations and it consists in computing the perturbation that leads to the highest energy growth for a fixed time horizon.

10.8.13.2 Input Parameters

In the `GEOMETRY` section, the dimensions of the problem are defined. Then, the coordinates (`XSCALE`, `YSCALE`, `ZSCALE`) of each vertices are specified. As this input file defines a two-dimensional problem: `ZSCALE = 0`.

```
<GEOMETRY DIM="2" SPACE="2">
   <VERTEX>
       <V ID="0">3.000e+00 -1.000e+00 0.000e+00</V>
       ...
       <V ID="399">-1.000e+01 0.000e+00 0.000e+00</V>
   </VERTEX>

```

Edges can now be defined by two vertices.

```
<EDGE>
       <E ID="0">    0  1   </E>
       ...
       <E ID="828">  399  394   </E>
   </EDGE>

```

In the `ELEMENT` section, the tag `T` and `Q` define respectively triangular and quadrilateral element. Triangular elements are defined by a sequence of three edges and quadrilateral elements by a sequence of four edges.

```
<ELEMENT>
       <T ID="0">    0     1     2 </T>
       ...
       <T ID="209">  333   314   332 </T>
       <Q ID="210">  334   335   336     0 </Q>
       ...
       <Q ID="429">  826   827   828   818 </Q>
   </ELEMENT>

```

Finally, collections of elements are listed in the `COMPOSITE` section and the `DOMAIN` section specifies that the mesh is composed by all the triangular and quadrilateral elements. The other composites will be used to enforce boundary conditions.

```
<COMPOSITE>
       <C ID="0"> T[0-209] </C>
```

```
        <C ID="1"> Q[210-429] </C>
        <C ID="2"> E[2-3,7,10,16,21,2,...,828] </C>
        <C ID="3"> E[821,823,825,827] </C>
        <C ID="4"> E[722,724,726,728] </C>
    </COMPOSITE>

    <DOMAIN> C[0,1] </DOMAIN>
</GEOMETRY>

```

10.8.13.3 Expansion

For this example we will use a 6th order polynomial, i.e. $P = 7$:

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="7" FIELDS="u,v,p" TYPE="MODIFIED" />
    <E COMPOSITE="C[1]" NUMMODES="7" FIELDS="u,v,p" TYPE="MODIFIED" />
</EXPANSIONS>

```

10.8.13.4 Solver Information

This sections defines the problem solved. In this example the `EvolutionOperator` must be `TransientGrowth` and the `Driver` was set up to `Arpack` for the solution of the eigenproblem.

```
 <SOLVERINFO>
            <I PROPERTY="EQTYPE"                VALUE="UnsteadyNavierStokes"    />
            <I PROPERTY="EvolutionOperator"     VALUE="TransientGrowth"         />
            <I PROPERTY="Projection"            VALUE="Galerkin"                />
            <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2"               />
            <I PROPERTY="SOLVERTYPE"            VALUE="VelocityCorrectionScheme"/>
            <I PROPERTY="Driver"                VALUE="Arpack"                  />
            <I PROPERTY="ArpackProblemType"     VALUE="LargestMag"              />
        </SOLVERINFO>

```

10.8.13.5 Parameters

```
<PARAMETERS>
            <P> FinalTime       = 0.1                   </P>
            <P> TimeStep        = 0.005                 </P>
            <P> NumSteps        = FinalTime/TimeStep    </P>
            <P> IO_CheckSteps   = 1/TimeStep            </P>
            <P> IO_InfoSteps    = 1                     </P>
            <P> Re              = 500                   </P>
            <P> Kinvis          = 1.0/Re                </P>
            <P> kdim            = 4                     </P>
            <P> nvec            = 1                     </P>
            <P> evtol           = 1e-4                  </P>
        </PARAMETERS>

```

10.8.13.6 Boundary Conditions

```
<BOUNDARYREGIONS>
            <B ID="0"> C[2] </B>    <!-- Wall -->
            <B ID="1"> C[3] </B>    <!-- Inlet -->
            <B ID="2"> C[4] </B>    <!-- Outlet -->
        </BOUNDARYREGIONS>

        <BOUNDARYCONDITIONS>
            <REGION REF="0">
                <D VAR="u" VALUE="0" />
                <D VAR="v" VALUE="0" />
                <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
            </REGION>
            <REGION REF="1">
                <D VAR="u" VALUE="0" />
                <D VAR="v" VALUE="0" />
                <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
            </REGION>
            <REGION REF="2">
                <D VAR="u" VALUE="0" />
                <D VAR="v" VALUE="0" />
                <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
            </REGION>
        </BOUNDARYCONDITIONS>

```

10.8.13.7 Functions

We need to set up the base flow that can be specified as a function `BaseFlow`. In case the base flow is not analytical, it can be generated by means of the `Nonlinear` evolution operator using the same mesh and polynomial expansion.

```
<FUNCTION NAME="BaseFlow">
            <F VAR="u,v,p" FILE="bfs_tg-AR.bse" />
        </FUNCTION>

```

The initial guess is specified in the `InitialConditions` functions and in this case is read from a file.

```
<FUNCTION NAME="InitialConditions">
            <F VAR="u,v,p" FILE="bfs_tg-AR.rst" />
        </FUNCTION>

```

10.8.13.8 Usage

```
IncNavierStokesSolver bfs_tg-AR.xml
```

10.8.13.9 Results

The solution will be evolved forward in time using the operator $\mathcal{A}$, then backward in time through the adjoint operator $\mathcal{A}^*$. The leading eigenvalue is $\lambda = 3.236204$). This

corresponds to the largest possible transient growth at the time horizon $\tau = 1$. The leading eigenmode is shown below. This is the optimal initial condition which will lead to the greatest growth when evolved under the linearised Navier-Stokes equations.

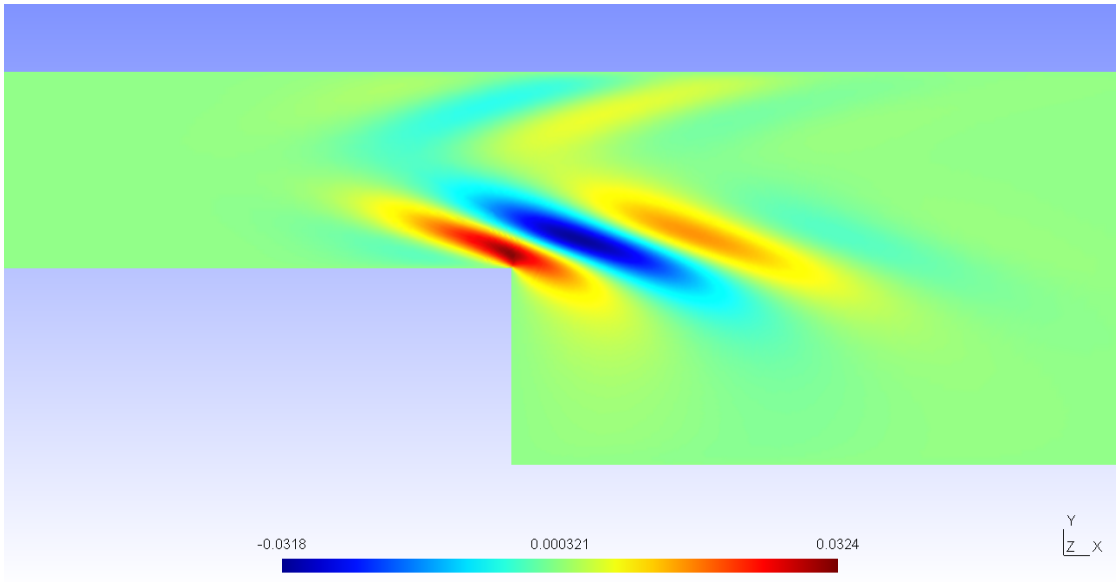


Figure 10.21

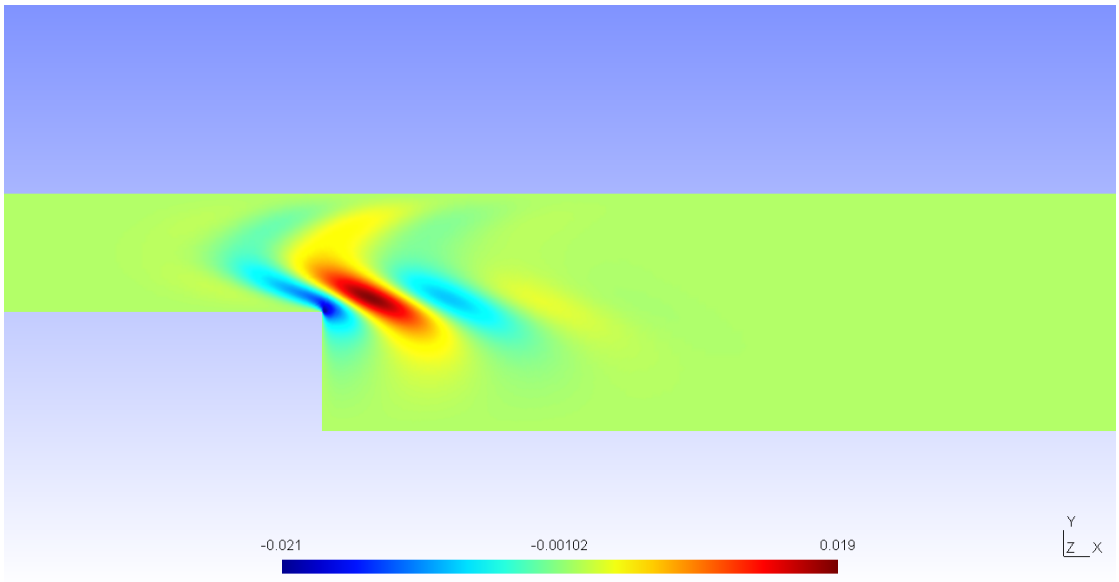


Figure 10.22

It is possible to visualise the transient growth plotting the energy evolution over time if the system is initially perturbed with the leading eigenvector. This analysis was performed for a time horizon $\tau = 60$. It can be seen that the energy grows in time

reaching its maximum value at x = 24 and then decays, almost disappearing after 100 temporal units.

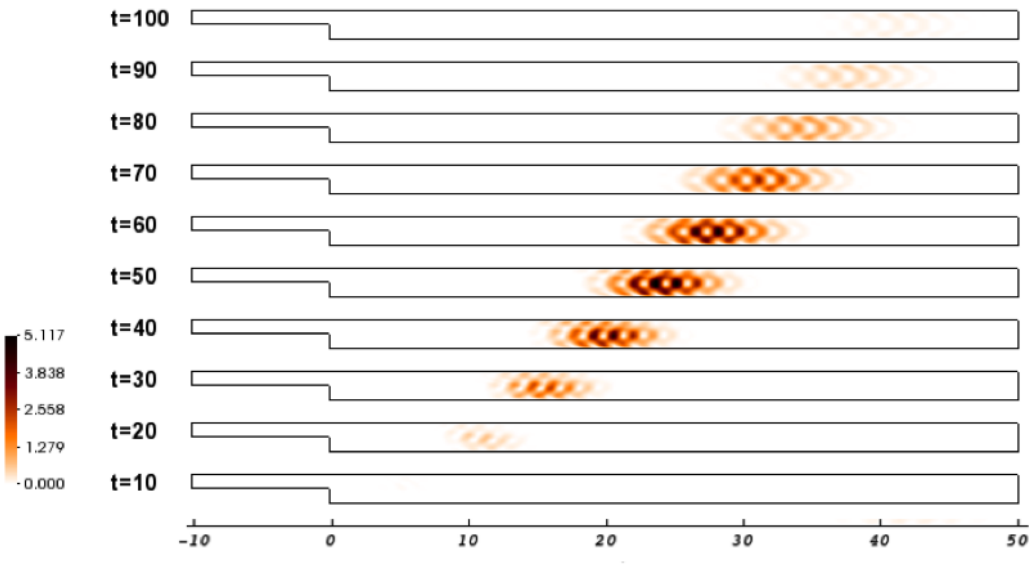


Figure 10.23

10.8.14 BiGlobal Floquet analysis of a of flow past a cylinder

In this example it will be described how to run a BiGlobal stability analysis for a time-periodic base flow using Nektar++. Let us consider a flow past a circular cylinder at $Re = 220$ has a 2D time-periodic wake that is unstable to a 3D synchronous "mode A" instability.

10.8.14.1 Background

The numerical solution of the fully three- dimensional linear eigenvalue problem is often computationally demanding and may not have significant advantages over performing a direct numerical simulation. Therefore, some simplifications are required; the most radical consist in considering that the base flow depends only on one spatial coordinate, assuming that the other two spatial coordinates are homogenous. While this method offers a good prediction for the instability of boundary layers, it is not able to predict the instability of Hagen-Poiseuille flow in a pipe at all Reynolds numbers. Between a flow that depends upon one and three-spatial directions, it is possible to consider a steady or time-periodic base flow depending upon two spatial directions and impose three-dimensional disturbances that are periodic in the the third homogeneous spatial direction. This approach is known as BiGlobal stability analysis and it represents the extension of the classic linear stability theory; let us consider a base flow **U** that is function of only two spatial coordinates: $mathbfU(x, y, t)$. The perturbation velocity can $\mathbf{u}'$ can be expressed in a similar form, but with the dependence on the third homogeneous direction

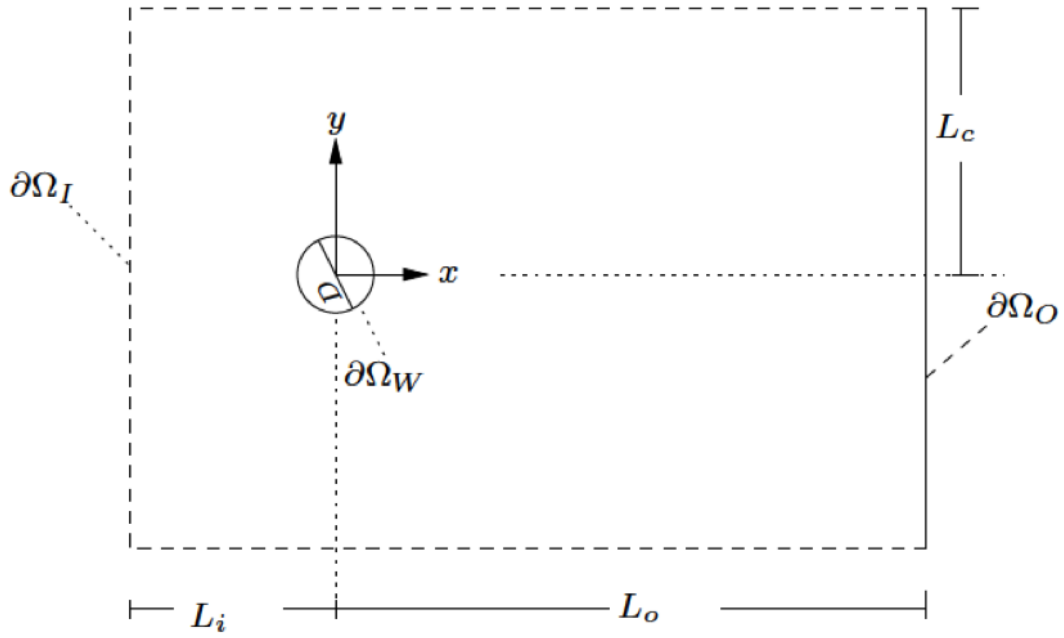


Figure 10.24

incorporated through the Fourier mode: $\mathbf{u}' = \hat{\mathbf{u}}'(x, y, t)e^{i\beta z}$, where $\beta = 2\pi/L$)and L is the length in the homogeneous direction.

10.8.14.2 Input parameters

In this example we use a mesh of 500 quadrilateral elements with a 6th order polynomial expansion. The base flow has been computed using the `Nonlinear` evolution operator with appropriate boundary conditions. From its profile, it was possible to determine the periodicity of the flow sampling the velocity profile over time. In order to reconstruct the temporal behaviour of the flow, 32 time slices were considered over one period. Using these data it is possible to set up the stability simulation for a specified β, for example $\beta = 1.7$. Let us note that while the base flow is 2D, the stability simulation that we are performing is 3D.

10.8.14.3 Expansion

In this example we will use a 6th order polynomial expansion, i.e. $P = 7$.

```
<EXPANSIONS>
        <E COMPOSITE="C[0]" NUMMODES="7" TYPE="GLL_LAGRANGE_SEM" FIELDS="u,v,w,p" />
    </EXPANSIONS>

```

10.8.14.4 Parameters

```
<SOLVERINFO>
      <I PROPERTY="SolverType"         VALUE="VelocityCorrectionScheme"/>
      <I PROPERTY="EQTYPE"             VALUE="UnsteadyNavierStokes"/>
      <I PROPERTY="EvolutionOperator" VALUE="Direct"/>
      <I PROPERTY="Projection"         VALUE="Galerkin"/>
      <I PROPERTY="ModeType"           VALUE="HalfMode"/>
      <I PROPERTY="Driver"             VALUE= "ModifiedArnoldi" />
      <I PROPERTY="HOMOGENEOUS"        VALUE="1D"/>
      <I PROPERTY="TimeIntegrationMethod" VALUE="IMEXOrder2" />
</SOLVERINFO>

```

10.8.14.5 Functions

```
<FUNCTION NAME="BaseFlow">
      <F VAR="u,v,p" FILE="cyinder_floq" />
</FUNCTION>

```

10.8.14.6 Usage

```
IncNavierStokesSolver session.xml
```

10.8.14.7 Results

The stability simulation takes about 20 cycles to converge and the leading eigenvalue is $\lambda = 1.2670$ with a growth rate $\sigma = 4.7694e-02$. The figure below shows the profile of the magnitude of the eigenmode at $z = 2$.

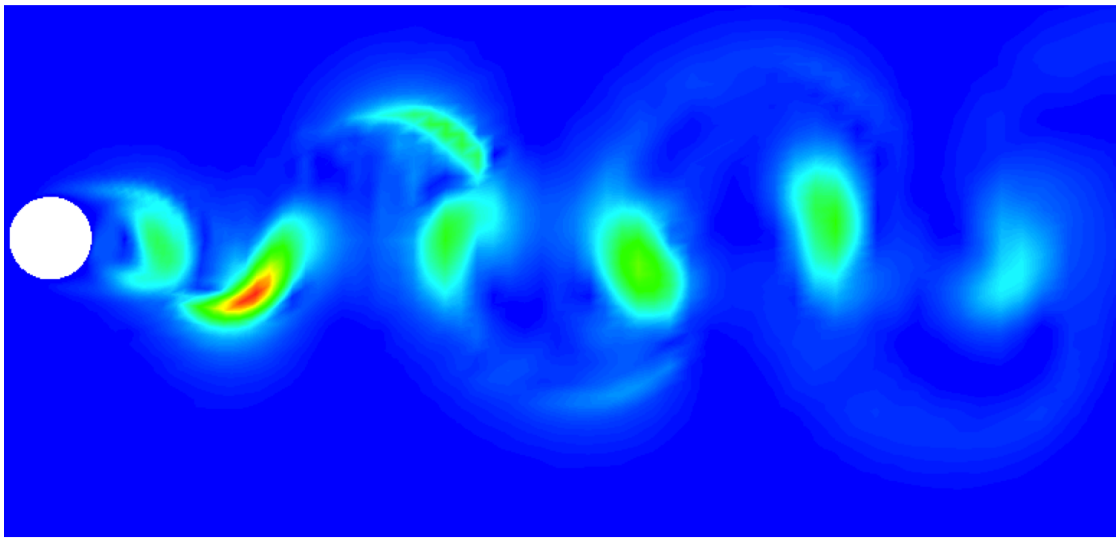


Figure 10.25

Chapter 11

Linear elasticity solver

11.1 Synopsis

The LinearElasticSolver is a solver for solving the linear elasticity equations in two and three dimensions. Whilst this may be suitable for simple solid mechanics problems, its main purpose is for use for mesh deformation and high-order mesh generation, whereby the finite element mesh is treated as a solid body, and the deformation is applied at the boundary in order to curve the interior of the mesh.

Currently the following equation systems are supported:

Value	Description
`LinearElasticSystem`	Solves the linear elastic equations.
`IterativeElasticSystem`	A multi-step variant of the elasticity solver, which breaks a given deformation into multiple steps, and applies the deformation to a mesh.

11.1.1 The linear elasticity equations

The linear elasticity equations model how a solid body deforms under the application of a 'small' deformation or strain. The formulation starts with the equilibrium of forces represented by the equation

$$\nabla \cdot \mathbf{S} + \mathbf{f} = \mathbf{0} \quad \text{in} \quad \Omega \tag{11.1}$$

where $\mathbf{S}$ is the stress tensor and $\mathbf{f}$ denotes a spatially-varying force. We further assume that the stress tensor $\mathbf{S}$ incorporates elastic and, optionally, thermal stresses that can be switched on to assist in mesh deformation applications. We assume these can be decomposed so that $\mathbf{S}$ is written as

$$\mathbf{S} = \mathbf{S}_e + \mathbf{S}_t,$$

where the subscripts e and t denote the elastic and thermal terms respectively. We adopt the usual linear form of the elastic stress tensor as

$$\mathbf{S}_e = \lambda \mathrm{Tr}(\mathbf{E})\,\mathbf{I} + \mu \mathbf{E},$$

where λ and μ are the Lamé constants, $\mathbf{E}$ represents the strain tensor, and $\mathbf{I}$ is the identity tensor. For small deformations, the strain tensor $\mathbf{E}$ is given as

$$\mathbf{E} = \frac{1}{2}\left(\nabla \mathbf{u} + \nabla \mathbf{u}^t\right) \tag{11.2}$$

where $\mathbf{u}$ is the two- or three-dimensional vector of displacements. The boundary conditions required to close the problem consist of prescribed displacements at the boundary $\partial\Omega$, i.e.

$$\mathbf{u} = \hat{\mathbf{u}} \quad \text{in } \partial\Omega. \tag{11.3}$$

We further express the Lamé constants in terms of the Young's modulus E and Poisson ratio ν as

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}.$$

The Poisson ratio, valid in the range $\nu < \frac{1}{2}$, is a measure of the compressibility of the body, and the Young's modulus $E > 0$ is a measure of its stiffness.

11.2 Usage

```
LinearElasticSolver [arguments] session.xml [another.xml] ...
```

11.3 Session file configuration

11.3.1 Solver Info

- `EqType` Specifies the PDE system to solve, based on the choices in the table above.
- `Temperature` Specifies the form of the thermal stresses to use. The choices are:
  - `None`: No stresses (default).
  - `Jacobian`: Sets $\mathbf{S}_t = \beta J I$, where β is a parameter defined in the parameters section, J is the elemental Jacobian determinant and I is the identity matrix.
  - `Metric`: A more complex term, based on the eigenvalues of the metric tensor. This can only be used for simplex elements (triangles and tetrahedra). Controlled again by the parameter β.
- `BCType` Specifies the type of boundary condition to apply when the `IterativeElasticSystem` is being used.

- `Normal`: The boundary conditions are split into `NumSteps` steps, as defined by a parameter in the session file (default).
- `Repeat`: As the geometry is updated, re-evaluate the boundary conditions. This enables, for example, a cirlce to be rotated continuously.

11.3.2 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file:

- `nu`: sets the Poisson ratio ν.
  Default value: 0.25.
- `E`: sets the Young's modulus E.
  Default value: 1.
- `beta`: sets the thermal stress coefficient β.
  Default value: 1.
- `NumSteps`: sets the number of steps to use in the case that the iterative elastic system is enabled. Should be greater than 0.
  Default value: 0.

11.4 Examples

11.4.1 L-shaped domain

The first example is the classic L-shaped domain, in which an exact solution is known, which makes it an ideal test case [17]. The domain is the polygon formed from the vertices

$$(-1,-1),(0,-2),(2,0),(0,2),(-1,-1),(0,0).$$

The exact solution for the displacements is known in polar co-ordinates (r,θ) as

$$u_r(r,\theta) = \frac{r^\alpha}{2\mu}\left[C_1(C_2-\alpha-1)\cos((\alpha-1)\theta) - (\alpha+1)\cos((\alpha+1)\theta)\right]$$
$$u_\theta(r,\theta) = \frac{r^\alpha}{2\mu}\left[(\alpha+1)\sin((\alpha+1)\theta) + C_1(C_2+\alpha-1)\sin((\alpha-1)\theta)\right]$$

where $\alpha \approx 0.544483737\ldots$ is the solution of the equation $\alpha\sin(2\omega)+\sin(2\omega\alpha)=0$,

$$C_1 = -\frac{\cos((\alpha+1)\omega)}{\cos((\alpha-1)\omega)}, \qquad C_2 = 2\frac{\lambda+2\mu}{\lambda+\mu}$$

with λ and μ being the Lamé constants and $\omega = 3\pi/4$. Boundary conditions are set to be the exact solution and $\mathbf{f}=\mathbf{0}$. The solution has a singularity at the origin, and so in order to test convergence h-refinement is required.

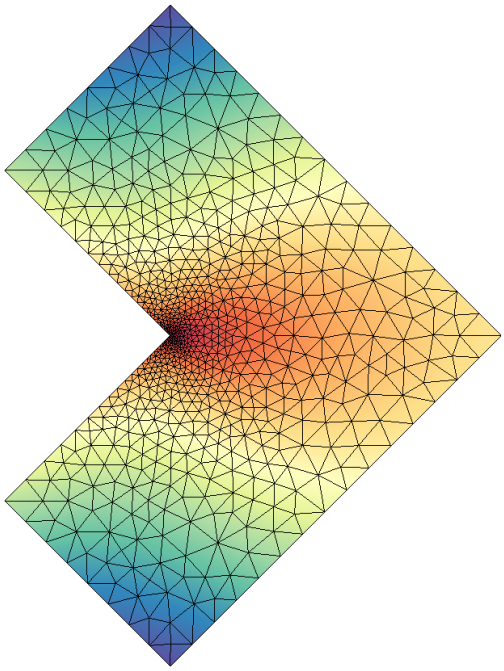


Figure 11.1 Solution of the u displacement field for the L-shaped domain.

A simple example of how the linear elastic solver can be set up can be found in the `Tests/L-shaped.xml` session file in the linear elastic solver directory. A more refined domain with the obtained u solution is shown in figure 11.1. The solver can be run using the command:

```
LinearElasticSolver L-domain.xml
```

The obtained solution `L-domain.fld` can be applied to the mesh to obtain a deformed XML file using the `deform` module in `FieldConvert`:

```
FieldConvert -m deform L-domain.xml L-domain.fld L-domain-deformed.xml
```

11.4.2 Boundary layer deformation

In this example we use the iterative elastic system to apply a large deformation to a triangular boundary layer mesh of a square mesh $\Omega = [0, 1]^2$. At the bottom edge, we apply a Dirichlet condition $g = \frac{1}{2}\sin(\pi x)$ that is enforced by splitting it into N substeps, so that at each step we solve the system with the boundary condition $g^n(x) = g(x)/N$. The process is depicted in figure 11.2.

The setup is very straightforward. The geometry can be found inside the file `Examples/bl-mesh.xml`

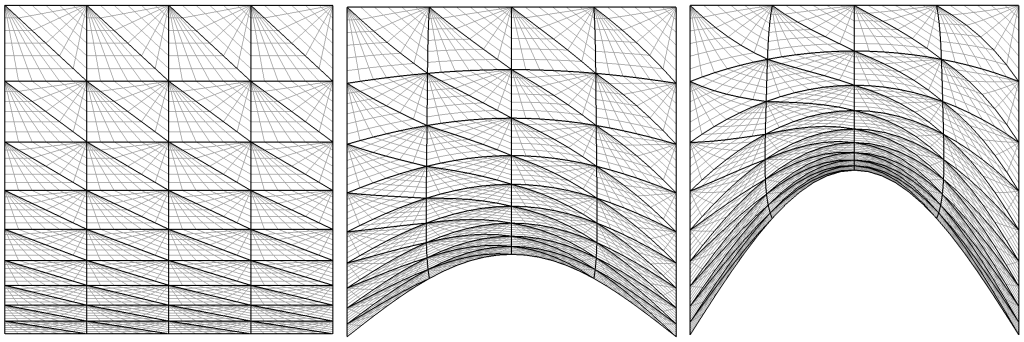


Figure 11.2 Figures that show the initial domain (left), after 50 steps (middle) and final deformation of the domain (right).

and the conditions inside `Examples/bl-conditions.xml`. The solver can be set up using the following parameters, with `NumSteps` denoting N:

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE" VALUE="IterativeElasticSystem" />
</SOLVERINFO>

<PARAMETERS>
  <P> nu       = 0.3 </P>
  <P> E        = 1.0 </P>
  <P> NumSteps = 100 </P>
</PARAMETERS>
```

To identify the boundary that we intend to split up into substeps, we must assign the `WALL` tag to our boundary regions:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="u" VALUE="0" USERDEFINEDTYPE="Wall" />
    <D VAR="v" VALUE="0.5*sin(PI*x)" USERDEFINEDTYPE="Wall" />
  </REGION>
  <REGION REF="1">
    <D VAR="u" VALUE="0" />
    <D VAR="v" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

The solver can then be run using the command:

```
LinearElasticSolver bl-mesh.xml bl-conditions.xml
```

This will produce a series of meshes `bl-mesh-%d.xml`, where %d is an integer running between 0 and 100. If at any point the mesh becomes invalid, that is, a negative Jacobian is detected, execution will cease.

Chapter 12

Pulse Wave Solver

12.1 Synopsis

1D modelling of the vasculature (arterial network) represents and insightful and efficient tool for tackling problems encountered in arterial biomechanics as well as other engineering problems. In particular, 3D modelling of the vasculature is relatively expensive. 1D modelling provides an alternative in which the modelling assumptions provide a good balance between physiological accuracy and computational efficiency.To describe the flow and pressure in this network we consider the conservation of mass and momentum applied to an impermeable, deformable tube filled with an incompressible fluid, the nonlinear system of partial differential equations presented in non-conservative form is given by

$$\frac{\partial \mathbf{U}}{\partial t} + \mathbf{H}\frac{\partial \mathbf{U}}{\partial x} = \mathbf{S} \tag{12.1}$$

$$\mathbf{U} = \begin{bmatrix} U \\ A \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} U & A \\ \rho\frac{\partial p}{\partial A} & U \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} 0 \\ \frac{1}{\rho}\left(\frac{f}{A} - s\right) \end{bmatrix}$$

in which A is the Area (related to pressure), x is the axial coordinate along the vessel, $U(x,t)$ the axial velocity, $P(x,t)$ is the pressure in the tube, ρ is the density and finally f the frictional force per unit length. The unknowns in equation 12.1 are u, A and p; hence we must provide an explicit algebraic relationship to close this system. Typically, closure is provided by an algebraic relationship between A and p. For a thin elastic tube this is given by

$$p = p_0 + \beta\left(\sqrt{A} - \sqrt{A_0}\right), \quad \beta = \frac{\sqrt{\pi h E}}{(1-\nu^2)A_0} \tag{12.2}$$

where p_0 is the external pressure, A_0 is the initial cross-sectional area, E is the Young's modulus, h is the vessel wall thickness and ν is the Poisson's ratio. Other more elaborate pressure - area relationships are currently being implemented into the framework. Application of Riemann's method of characteristics to equations 12.1 and 12.2 indicates that velocity and area are propagated through the system by forward and backward travelling waves. These waves are reflected and within the network by appropriate treatment of interfaces and boundaries. In the following, we will explain the usage of the blood flow solver on the basis of a single-artery problem and also on an arterial network consisting of 55 arteries.

12.2 Usage

```
PulseWaveSolver session.xml
```

12.3 Session file configuration

12.3.1 Pulse Wave Sovler mesh connectivity

Typically 1D arterial networks are made up of a connection of different base units: segments, bifurcations and merging junctions. The input format in the PulseWaveSolver means these connections are handle naturally from the mesh topology; hence care must be taken when designing the 1D domain. The figure below outlines the structure of a bifurcation, which is a common reoccurring structure in the vasculature.

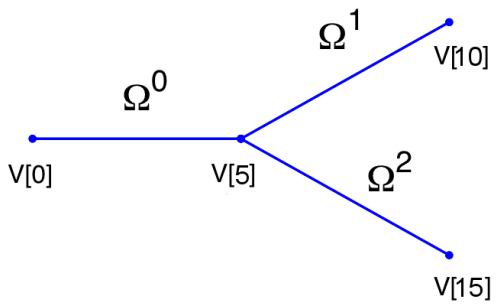


Figure 12.1 Model of bifurcating artery. The bifurcation is made of three domains and 15 vertices. Vertex V[0] is the inlet and vertices V[10] and V[15] the outlets.

To represent this topology in the xml file we specify the following vertices under the section `VERTEX` (the extents are: $-100 \geq x \leq 100$ and $-100 \geq y \leq 100$)

```
<VERTEX>
  <V ID="0">-1.000e+02 0.000e+00 0.000e+00</V>
  <V ID="1">-8.000e+01 0.000e+00 0.000e+00</V>
  <V ID="2">-6.000e+01 0.000e+00 0.000e+00</V>
  <V ID="3">-4.000e+01 0.000e+00 0.000e+00</V>
```

```
  <V ID="4">-2.000e+01 0.000e+00 0.000e+00</V>
  <V ID="5"> 0.000e+00 0.000e+00 0.000e+00</V>

  <V ID="6"> 2.000e+01 2.000e+01 0.000e+00</V>
  <V ID="7"> 4.000e+01 4.000e+01 0.000e+00</V>
  <V ID="8"> 6.000e+01 6.000e+01 0.000e+00</V>
  <V ID="9"> 8.000e+01 8.000e+01 0.000e+00</V>
  <V ID="10"> 1.000e+02 1.000e+02 0.000e+00</V>

  <V ID="11"> 2.000e+01 -2.000e+01 0.000e+00</V>
  <V ID="12"> 4.000e+01 -4.000e+01 0.000e+00</V>
  <V ID="13"> 6.000e+01 -6.000e+01 0.000e+00</V>
  <V ID="14"> 8.000e+01 -8.000e+01 0.000e+00</V>
  <V ID="15"> 1.000e+02 -1.000e+02 0.000e+00</V>
</VERTEX>
```

The elements from these vertices are then constructed under the section `ELEMENT` by defining

```
<ELEMENT>
  <!-- Parent artery -->
  <S ID="0">    0     1 </S>
  <S ID="1">    1     2 </S>
  <S ID="2">    2     3 </S>
  <S ID="3">    3     4 </S>
  <S ID="4">    4     5 </S>
  <!-- Daughter artery 1 -->
  <S ID="5">    5     6 </S>
  <S ID="6">    6     7 </S>
  <S ID="7">    7     8 </S>
  <S ID="8">    8     9 </S>
  <S ID="9">    9     10 </S>
  <!-- Daughter artery 2 -->
  <S ID="11">     5     11 </S>
  <S ID="12">    11    12 </S>
  <S ID="13">    12    13 </S>
  <S ID="14">    13    14 </S>
  <S ID="15">    14    15 </S>
</ELEMENT>
```

The composites, which represent groups of elements and boundary regions are defined under the section `COMPOSITE` by

```
<COMPOSITE>
  <C ID="0"> S[0-4] </C>        <!-- Parent artery -->
  <C ID="1"> V[0] </C>          <!-- Inlet to domain -->

  <C ID="3"> S[5-9] </C>        <!-- Daughter artery 1 -->
  <C ID="4"> V[10] </C>         <!-- Outlet of daughter artery 1 -->

  <C ID="6"> S[11-15] </C>      <!-- Daughter artery 2 -->
  <C ID="8"> V[15] </C>         <!-- Outlet of daughter artery 2 -->
</COMPOSITE>
```

Each of the segments can be then represented under the section `DOMAIN` by

```
<DOMAIN>
  <D ID="0"> C[0] </D>  <!-- Parent artery -->
  <D ID="1"> C[3] </D>  <!-- Daughter artery 1 -->
  <D ID="2"> C[6] </D>  <!-- Daughter artery 2 -->
</DOMAIN>
```

We will use the different domains later to define variable material properties and cross-sectional areas.

12.3.2 Session Info

The PulseWaveSolver is sqpecified through the `EquationType` option in the session file. This can be set as follows:

- `Projection`: Only a discontinuous projection can be specified using the following option:
  - `Discontinuous` for a discontinous Galerkin (DG) projection.
- `TimeIntegrationMethod`
- `UpwindTypePulse`:
  - `UpwindPulse`

12.3.3 Parameters

The following parameters can be specified in the `PARAMETERS` section of the session file.

- `TimeStep` is the time-step size;
- `FinTime` is the final physical time at which the simulation will stop;
- `NumSteps` is the equivalent of `FinTime` but instead of specifying the physical final time the number of time-steps is defined;
- `IO_CheckSteps` sets the number of steps between successive checkpoint files;
- `IO_InfoSteps` sets the number of steps between successive info stats are printed to screen;
- `rho` density of the fluid. Default value = 1.0;
- `nue` Poisson's ratio. Default value = 0.5 ;
- `pest` external pressure. Default value = 0;
- `h0` wall thickness Default value = 1.0;

12.3.4 Boundary conditions

In this section we can specify the boundary conditions for our problem. First we need to define the variables under the section `VARIABLES`.

```
<VARIABLES>
   <V ID="0"> A </V>
   <V ID="1"> u </V>
</VARIABLES>
```

The composites that we want to apply out boundary conditions then need to be defined in the `BOUNDARYREIONS`, for example if we had three composites (C[1], C[4] and C[8]) that correspond to three vertices of the computational mesh we would define:

```
<BOUNDARYREGIONS>
  <B ID="0"> C[1] </B>
  <B ID="1"> C[4] </B>
  <B ID="2"> C[8] </B>
 </BOUNDARYREGIONS>
```

Finally we can specify the boundary conditions on the regions specified under `BOUNDARYREGIONS`.

The Pulse Wave Solver comes with a number of boundary conditions that are unique to this solver. Boundary conditions must be provided for both the area and velocity at the inlets and outlets of the domain. Examples of the different boundary conditions will be provided in the following.

12.3.4.0.1 Inlet boundary condition: Typically at the inlet of the domain a flow profile (`Q-inflow`) is specified through a `USERDEFINEDTYPE` boundary conditioning . An example inlet condition for the parent artery of the previously bifurcation example is

```
<REGION REF="0">
        <D VAR="A" USERDEFINEDTYPE="Q-inflow" VALUE="(7.112e-4)*(sin(7.854*t)
-0.562)*(1/(1+exp(-400*(sin(7.854*t)-0.562))))" />
        <D VAR="u" USERDEFINEDTYPE="Q-inflow" VALUE="1.0" />
</REGION>
```

12.3.4.0.2 Terminal boundary conditions: At the outlets of the domain there are four possible boundary conditions: reflection (`Terminal`), terminal resistance `R-terminal`, Two element windkessel (CR) `CR-terminal`, and three element windkessel (RCR) `RCR-terminal`. An example of the outflow boundary condition of the RCR terminal is given below

```
<REGION REF="1">
        <D VAR="A" USERDEFINEDTYPE="RCR-terminal" VALUE="RT" />
        <D VAR="u" USERDEFINEDTYPE="RCR-terminal" VALUE="C" />
</REGION>
```

Where `RT` is the total peripheral resistance used in the the `R-terminal`, `CR-terminal` and `RCR-terminal` models

12.3.5 Functions

The following functions can be specified inside the `CONDITIONS` section of the session file:

- `MaterialProperties`: specifies the material properties for each domain.
- `A_0`: Initial area of each domain.
- `AdvectionVelocity`: specifies the advection velocity v.
- `InitialConditions`: specifies the initial condition for unsteady problems.
- `Forcing`: specifies the forcing function f

As an example to specify the material properties for each domain in the previous bifurcation example we would enter:

```
<FUNCTION NAME="MaterialProperties">
   <E VAR="beta" DOMAIN="0" VALUE="97" />
   <E VAR="beta" DOMAIN="1" VALUE="87" />
   <E VAR="beta" DOMAIN="2" VALUE="233" />
</FUNCTION>
```

The values of `beta` are used in the pressure-area relationship (equation 12.2).

12.4 Examples

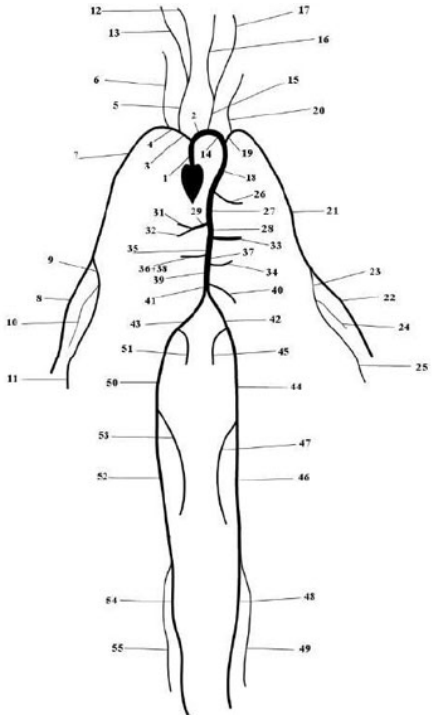
12.4.1 Human Vascular Network

The Pulse Wave Solver is also capable of handling more complex networks, such as a complete human arterial tree proposed by Westerhof et al. [34]. In this example, we will use the refined data from [30] and set up the network shown in the figure in the right. We will explain how bifurcations are set correctly and how each arterial segment gets its correct physiological data.

First, we will set up the mesh where each arterial segment is represented by one element and two vertices respectively. Then, we will subdivide the mesh into different subdomains by using the `<COMPOSITE>` section. Here, each arterial segment is described by the contained elements and its first and last vertex.

The mesh connectivity is specified during the creation of elements by indicating the starting vertex and ending vertex of each individual artery segment. Shared vertices are used to describe bifurcations, junctions and mergers between different artery segments in the network.

The composites are then used to specify the two adjoining segments of an artery, where the first segment merely allows for description of the connectivity.



#	Artery	Length (cm)	Area (cm^2)	β ($kg\,s^{-2}\,cm^{-2}$)	R_t
1	Ascending Aorta	4.0	5.983	97	-
2	Aortic Arch I	2.0	5.147	87	-
3	Brachiocephalic	3.4	1.219	233	-
4	R. Subclavian I	3.4	0.562	423	-
5	R. Carotid	17.7	0.432	516	-
6	R. Vertebral	14.8	0.123	2590	0.906
7	R. Subclavian II	42.2	0.510	466	-
8	R. Radial	23.5	0.106	2866	0.82
9	R. Ulnar I	6.7	0.145	2246	-
10	R. Interosseous	7.9	0.031	12894	0.956
11	R. Ulnar II	17.1	0.133	2446	0.893
12	R. Internal Carotid	17.6	0.121	2644	0.784
13	R. External Carotid	17.7	0.121	2467	0.79
14	Aortic Arch II	3.9	3.142	130	-
15	L. Carotid	20.8	0.430	519	-
16	L. Internal Carotid	17.6	0.121	2644	0.784
17	L. External Carotid	17.7	0.121	2467	0.791
18	Thoracic Aorta I	5.2	3.142	124	-
19	L. Subclavian I	3.4	0.562	416	-
20	Vertebral	14.8	0.123	2590	0.906
21	L. Subclavian II	42.2	0.510	466	-
22	L. Radial	23.5	0.106	2866	0.821
23	L. Ulnar I	6.7	0.145	2246	-
24	L. Interosseous	7.9	0.031	12894	0.956
25	L. Ulnar II	17.1	0.133	2446	0.893
26	Intercostals	8.0	0.196	885	0.627
27	Thoracic Aorta II	10.4	3.017	117	-
28	Abdominal I	5.3	1.911	167	-
29	Celiac I	2.0	0.478	475	-
30	Celiac II	1.0	0.126	1805	-
31	Hepatic	6.6	0.152	1142	0.925
32	Gastric	7.1	0.102	1567	0.921
33	Splenic	6.3	0.238	806	0.93
34	Superior Mesenteric	5.9	0.430	569	0.934
35	Abdominal II	1.0	1.247	227	-
36	L. Renal	3.2	0.332	566	0.861
37	Abdominal III	1.0	1.021	278	-
38	R. Renal	3.2	0.159	1181	0.861
39	Abdominal IV	10.6	0.697	381	-
40	Inferior Mesenteric	5.0	0.080	1895	0.918
41	Abdominal V	1.0	0.578	399	-
42	R. Common Iliac	5.9	0.328	649	-
43	L. Common Iliac	5.8	0.328	649	-
44	L. External Iliac	14.4	0.252	1493	-
45	L. Internal Iliac	5.0	0.181	3134	0.925
46	L. Femoral	44.3	0.139	2559	-
47	L. Deep Femoral	12.6	0.126	2652	0.885
48	L. Posterior Tibial	32.1	0.110	5808	0.724
49	L. Anterior Tibial	34.3	0.060	9243	0.716
50	R. External Iliac	14.5	0.252	1493	-
51	R. Internal Iliac	5.1	0.181	3134	0.925
52	R. Femoral	44.4	0.139	2559	-
53	R. Deep Femoral	12.7	0.126	2652	0.888
54	L. Posterior Tibial	32.2	0.110	5808	0.724
55	R. Anterior Tibial	34.4	0.060	9243	0.716

```
<GEOMETRY DIM="1" SPACE="1">
    <VERTEX>
        <V ID="0"> 0.000e+00 0.000e+00 0.000e+00</V> <!-- 1 -->
        <V ID="1"> 4.000e+00 0.000e+00 0.000e+00</V>

        <V ID="2"> 4.000e+00 0.000e+00 0.000e+00</V> <!-- 2 -->
        <V ID="3"> 6.000e+00 0.000e+00 0.000e+00</V>

        <V ID="4"> 4.000e+00 0.000e+00 0.000e+00</V> <!-- 3 -->
        <V ID="5"> 7.400e+00 0.000e+00 0.000e+00</V>
            .
            .
            .
        <V ID="108"> 109.100e+00 -45.000e+00 0.000e+00</V> <!-- 55 -->
        <V ID="109"> 143.500e+00 -45.000e+00 0.000e+00</V>
    </VERTEX>
    <ELEMENT>
        <S ID="0">    0     1 </S>
        <S ID="1">    1     2 </S>
        <S ID="2">    1     4 </S>
        <S ID="3">    2     3 </S>
        <S ID="4">    4     5 </S>
        <S ID="5">    5     6 </S>
        <S ID="6">    5     8 </S>
        <S ID="7">    6     7 </S>
```

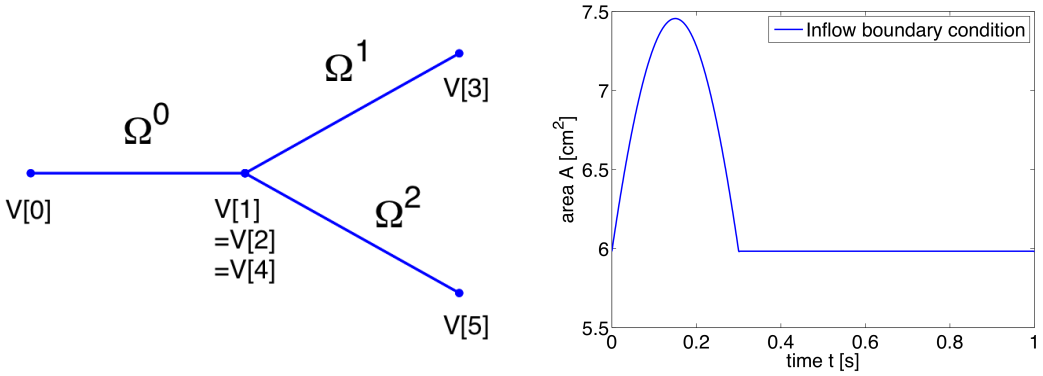
```
        <S ID="8">    8     9 </S>
          .
          .
          .
        <S ID="106">   103    108 </S>
        <S ID="107">   108    109 </S>
        <S ID="108">   85    98 </S>
    <ELEMENT>
    <COMPOSITE>
        <C ID="0"> S[0] </C> <!-- 1 -->
        <C ID="1"> V[0] </C>
        <C ID="2"> V[1] </C>

        <C ID="3"> S[1,3] </C> <!-- 2 -->
        <C ID="4"> V[2] </C>
        <C ID="5"> V[3] </C>

        <C ID="6"> S[2,4] </C> <!-- 3 -->
        <C ID="7"> V[4] </C>
        <C ID="8"> V[5] </C>
           .
           .
           .
        <C ID="162"> S[106,107] </C> <!-- 55 -->
        <C ID="163"> V[108] </C>
        <C ID="164"> V[109] </C>
    </COMPOSITE>
</GEOMETRY>
```

Then the choice of polynomial order, solver information, area of the arteries and other parameters are specified.

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="5" FIELDS="A,u" TYPE="MODIFIED" />
    <E COMPOSITE="C[3]" NUMMODES="5" FIELDS="A,u" TYPE="MODIFIED" />
       ...

    <E COMPOSITE="C[162]" NUMMODES="5" FIELDS="A,u" TYPE="MODIFIED" />
</EXPANSIONS>

<CONDITIONS>

    <PARAMETERS>

        <P> TimeStep       = 1e-4              </P>
        <P> FinTime        = 1.0             </P>
        <P> NumSteps       = FinTime/TimeStep   </P>
        <P> IO_CheckSteps  = NumSteps/50        </P>
           ...
        <P> A53            = 0.126             </P>
        <P> A54            = 0.110             </P>
        <P> A55            = 0.060             </P>
    </PARAMETERS>

    <SOLVERINFO>
```

```
        <I PROPERTY="EQTYPE" VALUE="PulseWavePropagation" />
        <I PROPERTY="Projection" VALUE="DisContinuous" />
        <I PROPERTY="TimeIntegrationMethod" VALUE="RungeKutta2_ImprovedEuler" />
        <I PROPERTY="UpwindTypePulse"  VALUE="UpwindPulse"/>
    </SOLVERINFO>

    <VARIABLES>
        <V ID="0"> A </V>
        <V ID="1"> u </V>
    </VARIABLES>
```

The vertices where the network terminates are specified as boundary regions based on their subsequent composite ids.




```
        <BOUNDARYREGIONS>
            <B ID="0"> C[1] </B> <B ID="1"> C[17] </B> <B ID="2"> C[23] </B>
               ...
            <B ID="28"> C[164] </B>
        </BOUNDARYREGIONS>
```

In the boundary conditions section the inflow and outflow conditions are set up. Here we use an inflow boundary condition for the area at the beginning of the ascending aorta taken from [30] and plotted on the right. Potential choices for inflow boundary conditions include Q-Inflow and Time-Dependent inflow. The outflow conditions for the terminal regions of the network could be specified by different models including eTerminal, R, CR, RCR and Time-Dependant outflow.

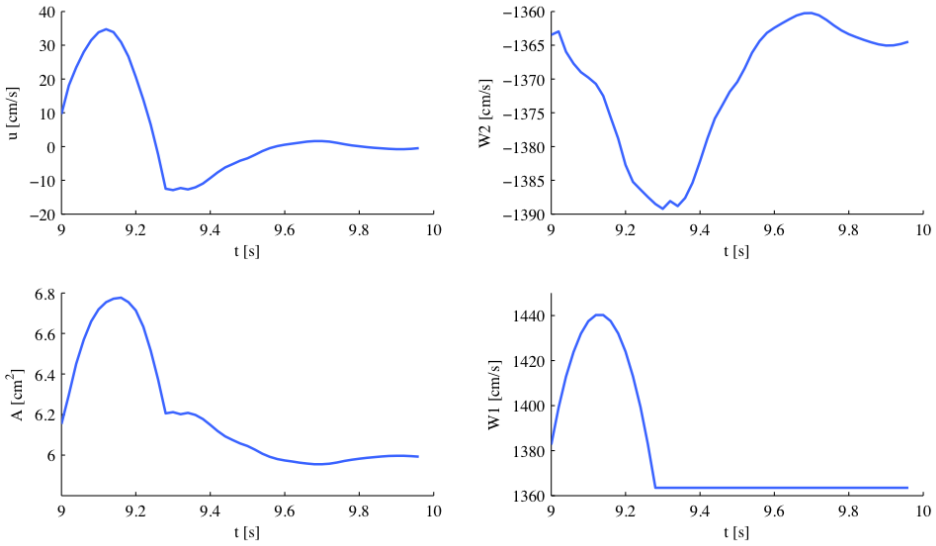
```
<BOUNDARYCONDITIONS>
    <REGION REF="0"> <!-- Inflow -->
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"
           VALUE="5.983*(1+0.597*(sin(6.28*t + 0.628) - 0.588)*
                (1./(1+exp(-2*200*(sin(6.28*t + 0.628) - 0.588)))))" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent" VALUE="0.0" />
    </REGION>
    <REGION REF="1">
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  VALUE="A6" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
    <REGION REF="2">
```

```
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  VALUE="A8" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
    <REGION REF="3">
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  VALUE="A10" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
    ....
    <REGION REF="28">
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  VALUE="A55" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Again, for the initial conditions we start our simulation from static equilibrium conditions $A = A_0$ and for u being initially at rest. The following lines show how we specify A_0 and β for different arterial segments.

```
<FUNCTION NAME="InitialConditions">
    <E VAR="A" DOMAIN="0" VALUE="5.983" />
    <E VAR="u" DOMAIN="0" VALUE="0.0" />
</FUNCTION>
   ...
<FUNCTION NAME="InitialConditions">
    <E VAR="A" DOMAIN="54" VALUE="A55" />
    <E VAR="u" DOMAIN="54" VALUE="0.0" />
</FUNCTION>

<FUNCTION NAME="A_0">
    <E VAR="A_0" DOMAIN="0" VALUE="A1" />
       ...
    <E VAR="A_0" DOMAIN="54" VALUE="A55" />
</FUNCTION>

<FUNCTION NAME="MaterialProperties">
    <E VAR="beta" DOMAIN="0" VALUE="97" />
      ...
    <E VAR="beta" DOMAIN="54" VALUE="9243" />
</FUNCTION>
```

Our simulation is started as described before and the results show the time history for the conservative variables A and u, as well as for the characteristic variables W1 and W2 at the beginning of the ascending aorta (Artery 1). We can see that physically correct the shape of the inflow boundary condition appears in the forward traveling characteristic W1. As we do not have a terminal resistance at the outflow, one would normally expect W2 to be constant. However this is not the case, as bifurcations cause reflections if the radii of parent and daughter vessels are not well matching, leading to changes in W2. The shapes of A and u result from this facts and show the values for the physiological variables during one cardiac cycle. We may annotate that this values slightly differ from in vivo measurements due to the missing terminal resistance, which will be added in future.



These short examples should give an insight to the functionality of our PulseWaveSolver and show that results such as luminal area and pressure within the artery can be simulated. These results can contribute to understanding the physiology of the human vascular system and they can be used for patient-specific planning of medical interventions.

12.4.2 Stented Artery

12.4.3 Stented Artery

In the following we will explain the usage of the Pulse Wave solver to model the flow and pressure variation through a stented artery - a cardiovascular procedure in which a small mesh tube is inserted into an artery to restore blood flow through a constricted region. Due to the implantation of the stent this region will have different material properties compared to the the surrounding unstented tissue; hence will influence the propagation of waves through this system. The stent scenario to be modelled is a straight arterial segment with a stent situated between $x = a_1$ and $x = a_2$ as shown below.

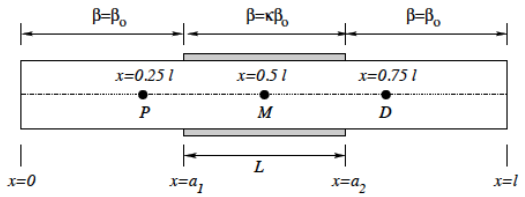


Figure 12.2 Model of straight artery with a stent in the middle.

12.4.3.0.1 Geometry: In the following we describe the geometry setup for modelling 1D flow in a stent. This is done by defining vertices, elements and composites. The vertices of the domain are shown below, consisting of 30 elements (Ω) and 31 vertices (V[n]).

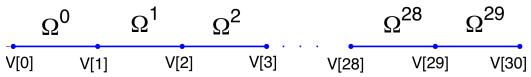


Figure 12.3 1D arterial domain consisting of 30 elements and 31 vertices.

To represent the above in the xml file, we define 31 vertices as follows:

```
<VERTEX>
  <V ID="0"> 0.000e+00 0.000e+00 0.000e+00</V>
                .
                .
                .
  <V ID="30">30.000e+00 0.000e+00 0.000e+00</V>
</VERTEX>
```

and the connectivity of these vertices to make up the 30 elements:

```
<ELEMENT>
  <S ID="0">    0     1 </S>
                .
                .
                .
  <S ID="29">   29    30 </S>
</ELEMENT>
```

These elements are combined to three different composites (shown below): composite 0 represents all the elements; composite 1 the inflow boundary and composite 2 the outflow boundary.

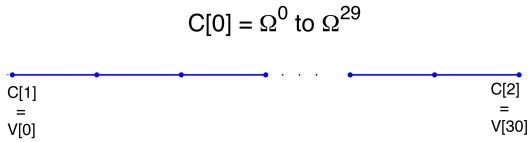


Figure 12.4 Three composites (C[0], C[1] and C[2]) for the stunted artery.

The above composites are specified as follows:

```
<COMPOSITE>
  <C ID="0"> S[0-29] </C>
  <C ID="1"> V[0] </C>
  <C ID="2"> V[30] </C>
</COMPOSITE>
```

Finally the domain is specified by the first composite by

```
<DOMAIN>
  <D ID="0"> C[0] </D>
</DOMAIN>
```

12.4.3.0.2 Expansion: For the expansions we use 4th-order polynomials which define our two variables A and u on the domain.

```
<EXPANSIONS>
  <E COMPOSITE="C[0]" NUMMODES="5" FIELDS="A,u" TYPE="MODIFIED" />
</EXPANSIONS>
```

12.4.3.0.3 Solver Information: The Discontinuous Galerkin Method is used as projection scheme and the time-integration is performed by a simple Forward Euler scheme. A full list of possible time integration scheme is given in the parameter section of the Pulse Wave Solver

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE" VALUE="PulseWavePropagation" />
  <I PROPERTY="Projection" VALUE="DisContinuous" />
  <I PROPERTY="TimeIntegrationMethod" VALUE="ForwardEuler" />
  <I PROPERTY="UpwindTypePulse"  VALUE="UpwindPulse"/>
</SOLVERINFO>
```

12.4.3.0.4 Parameters: Parameters used for the simulation are taken from [30]

```
<PARAMETERS>
   <P> TimeStep       = 2e-6              </P>
   <P> FinTime        = 0.25              </P>
   <P> NumSteps       = FinTime/TimeStep  </P>
   <P> IO_CheckSteps  = NumSteps/50       </P>
   <P> IO_InfoSteps   = 100               </P>
   <P> T              = 0.33              </P>
   <P> h0             = 1.0               </P>
   <P> rho            = 1.0               </P>
   <P> nue            = 0.5               </P>
   <P> pext           = 0.0               </P>
   <P> a1             = 10.0              </P>
   <P> a2             = 20.0              </P>
   <P> kappa          = 100.0             </P>
   <P> Y0             = 1.9099e+5         </P>
   <P> k              = 2                 </P>
   <P> k1             = 200               </P>
</PARAMETERS>
```

12.4.3.0.5 Boundary conditions: At the inflow we apply a pressure boundary condition as shown in the figure below. This condition models the pressure variation during one heartbeat. A simple absorbing outflow boundary condition is applied the right end of the tube.

These are defined in the xml file as follows,

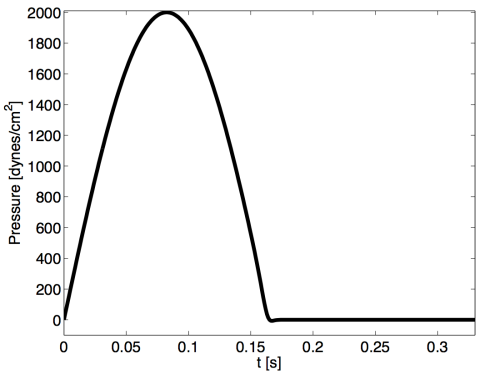


Figure 12.5 Pressure profile applied at the inlet of the artery

```
<BOUNDARYREGIONS>
   <B ID="0"> C[1] </B>
   <B ID="1"> C[2] </B>
</BOUNDARYREGIONS>

<BOUNDARYCONDITIONS>
   <REGION REF="0">
      <D VAR="A" USERDEFINEDTYPE="TimeDependent" VALUE=
      "(2000*sin(2*PI*t/T)*1./(1+exp(-2*k1*(T/2-t))-pext)/451352+1)^2" />
      <D VAR="u" USERDEFINEDTYPE="TimeDependent" VALUE="1.0" />
   </REGION>
   <REGION REF="1">
      <D VAR="A" VALUE="1.0" />
      <D VAR="u" VALUE="1.0" />
   </REGION>
</BOUNDARYCONDITIONS>
```

The simulation starts from the static equilibrium of the vessel with normalised area and velocity.

```
<FUNCTION NAME="InitialConditions">
        <E VAR="A" DOMAIN="0" VALUE="1.0" />
        <E VAR="u" DOMAIN="0" VALUE="1.0" />
</FUNCTION>

<FUNCTION NAME="A_0">
        <E VAR="A" DOMAIN="0" VALUE="1.0" />
</FUNCTION>
```

12.4.3.0.6 Functions: The stent is introduced by applying a variable material properties function (β - see equation 12.2) along the vessel in the x direction, shown graphically below and defined in the xml file by

```
<FUNCTION NAME="MaterialProperties">
        <E VAR="E0" DOMAIN="0" VALUE=
```

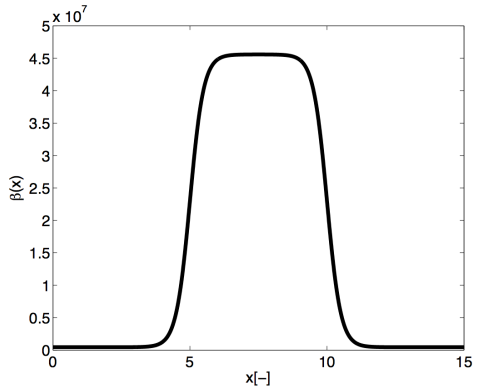


Figure 12.6 material property variation along the artery. The stiff region in the middle represents the stent.

```
        "Y0*(1.0-kappa/(1+exp(-2*k*(a1-x)))+kappa/(1+exp(-2*k*(a2-x))))" />
</FUNCTION>
```

12.4.3.1 Simulation

The simulation is started by running

```
PulseWaveSolver Test_1.xml
```

It will take about 60 seconds on a 2.4GHz Intel Core 2 Duo processor and therefore is computationally realisable at every clinical site.

12.4.3.2 Results

As a result we get a 3-dimensional interpretation of the aortic cross-sectional area varying in axial direction both for the stented and non-stented vessel. In case of the stent, the rigid metal mesh will restrict the deformation of the area in that specific part of the artery compared to the normal vessel (Fig. 12.7).

Also, if we look at the pressure at three points within the artery (P, M, D) we will recognize that there are major differences between the stented and normal vessel. While in the normal vessel (left) the pressure wave applied at the inflow is propagated without any losses, this does not hold for the stented artery (right). Here, the stiffening at the stent causes reflections and thus there are losses for total pressure at the medial (M) and distal (D) point.

12.5 Further Information

The PulseWaveSolver has beqen developed with contributions by various students and researchers at the Department of Aeronautics, Imperial College London. Further in-

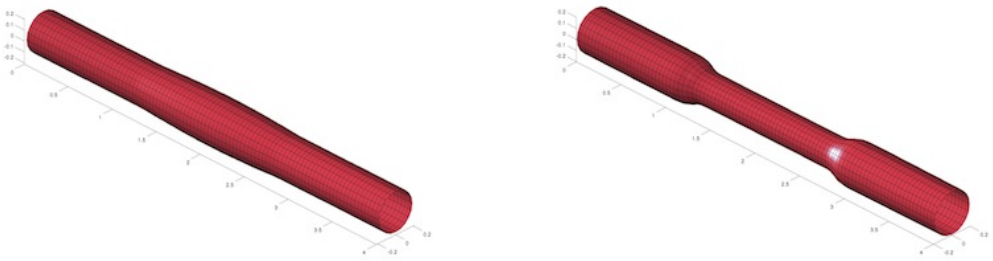
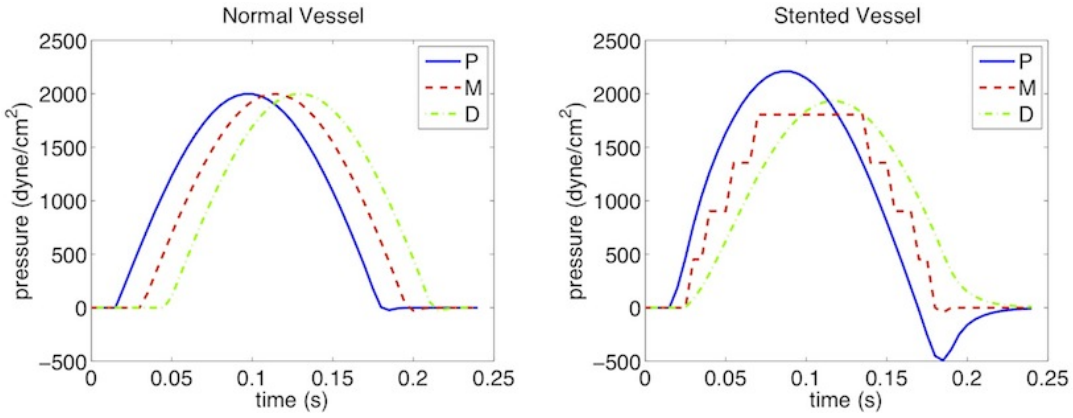


Figure 12.7



formation on the solver and its underlying mathematical framework can be found in [27, 26].

12.6 Future Development

The PulseWaveSolver is a useful tool for computational modelling of one-dimensional blood flow in the human body. However, there are several ideas for future development which include:

1. Inclusion of a pre-processor and post-processor.
2. Profiling the code to improve performance.
3. Cleaning up the input file to make the input format more user-friendly.
4. Modelling of valves and alternative pressure-area laws for models of venous flow.
5. Incorporating a model of the heart.

Chapter 13

Shallow Water Solver

13.1 Synopsis

The ShallowWaterSolver is a solver for depth-integrated wave equations of shallow water type. Presently the following equations are supported:

Value	Description
`LinearSWE`	Linearized SWE solver in primitive variables (constant still water depth)
`NonlinearSWE`	Nonlinear SWE solver in conservative variables (constant still water depth)

13.1.1 The Shallow Water Equations

The shallow water equations (SWE) is a two-dimensional system of nonlinear partial differential equations of hyperbolic type that are fundamental in hydraulic, coastal and environmental engineering. In deriving the SWE the vertical velocity is considered negligible and the horizontal velocities are assumed uniform with depth. The SWE are hence valid when the water depth can be considered small compared to the characteristic length scale of the problem, as typical for flows in rivers and shallow coastal areas. Despite the limiting restrictions the SWE can be used to describe many important phenomena, for example storm surges, tsunamis and river flooding.

The two-dimensional SWE is stated in conservation form as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}(U) = \mathbf{S}(U)$$

where $\mathbf{F}(U) = [\mathbf{E}(U), \mathbf{G}(U)]$ is the flux vector and the vector of conserved variables read $\mathbf{U} = [H, Hu, Hv]^{\mathrm{T}}$. Here $H(\mathbf{x}, t) = \zeta(\mathbf{x}, t) + d(\mathbf{x})$ is the total water depth, $\zeta(\mathbf{x}, t)$ is the free surface elevation and $d(\mathbf{x})$ is the still water depth. The depth-averaged velocity is

denoted by $\mathbf{u}(\mathbf{x}, t) = [u, v]^{\mathrm{T}}$, where u and v are the velocities in the x- and y-directions, respectively. The content of the flux vector is

$$\mathbf{E}(U) = \begin{bmatrix} Hu \\ Hu^2 + gH^2/2 \\ Huv \end{bmatrix}, \qquad \mathbf{G}(U) = \begin{bmatrix} Hv \\ Hvu \\ Hv^2 + gH^2/2 \end{bmatrix},$$

in which g is the acceleration due to gravity. The source term $\mathbf{S}(U)$ accounts for, e.g., forcing due to bed friction, bed slope, Coriolis force and higher-order dispersive effects (Boussinesq terms). In the distributed version of the ShallowWaterSolver only the Coriolis force is included.

13.2 Usage

```
ShallowWaterSolver session.xml
```

13.3 Session file configuration

13.3.1 Solver Info

- `Eqtype`: Specifies the equation to solve. This should be set to `NonlinearSWE`.
- `UpwindType`
- `Projection`
- `TimeIntegrationScheme`

13.3.2 Parameters

- `Gravity`

13.3.3 Functions

- `Coriolis`: Specifies the Coriolis force (variable name: ‘f‘)
- `WaterDepth`: Specifies the water depth (variable name: ‘d‘)

13.4 Examples

13.4.1 Rossby modon case

This example, provided in `RossbyModon_Nonlinear_DG.xml` is of a discontinuous Galerkin simulation of the westward propagation of an equatorial Rossby modon.

13.4.1.1 Input Options

For what concern the ShallowWaterSolver the `<SOLVERINFO>` section allows us to specify the solver, the type of projection (continuous or discontinuous), the explicit time integration scheme to use and (in the case the discontinuous Galerkin method is used) the choice of numerical flux. A typical example would be:

```
<SOLVERINFO>
  <I PROPERTY="EqType" VALUE="NonlinearSWE">
  <I PROPERTY="Projection" VALUE="DisContinuous">
  <I PROPERTY="TimeIntegrationMethod" VALUE="ClassicalRungeKutta4">
  <I PROPERTY="UpwindType" VALUE="HLLC">
</SOLVERINFO>
```

In the `<PARAMETERS>` section we, in addition to the normal setting of time step etc., also define the acceleration of gravity by setting the parameter "Gravity":

```
 <PARAMETERS>
    <P> TimeStep       = 0.04            </P>
    <P> NumSteps       = 1000            </P>
    <P> IO_CheckSteps  = 100             </P>
    <P> IO_InfoSteps   = 100             </P>
    <P> Gravity        = 1.0             </P>
 </PARAMETERS>
```

We specify f which is the Coriolis parameter and d denoting the still water depth as analytic functions:

```
<FUNCTION NAME="Coriolis">
    <E VAR="f" VALUE="0+1*y" />
</FUNCTION>

<FUNCTION NAME="WaterDepth">
    <E VAR="d" VALUE="1" />
</FUNCTION>
```

Initial values and boundary conditions are given in terms of primitive variables (please note that also the output files are given in terms of primitive variables). For the discontinuous Galerkin we typically enforce any slip wall boundaries weakly using symmetry technique. This is given by the `USERDEFINEDTYPE="Wall"` choice in the `<BOUNDARYCONDITIONS>` section:

```
 <BOUNDARYCONDITIONS>
   <REGION REF="0">
    <D VAR="eta"  USERDEFINEDTYPE="Wall"  VALUE="0" />
    <D VAR="u"    USERDEFINEDTYPE="Wall"  VALUE="0" />
    <D VAR="v"    USERDEFINEDTYPE="Wall"  VALUE="0" />
   </REGION>
 </BOUNDARYCONDITIONS>
```

13.4.1.2 Running the code

After the input file has been copied to the build directory of the `ShallowWaterSolver` the code can be executed by:

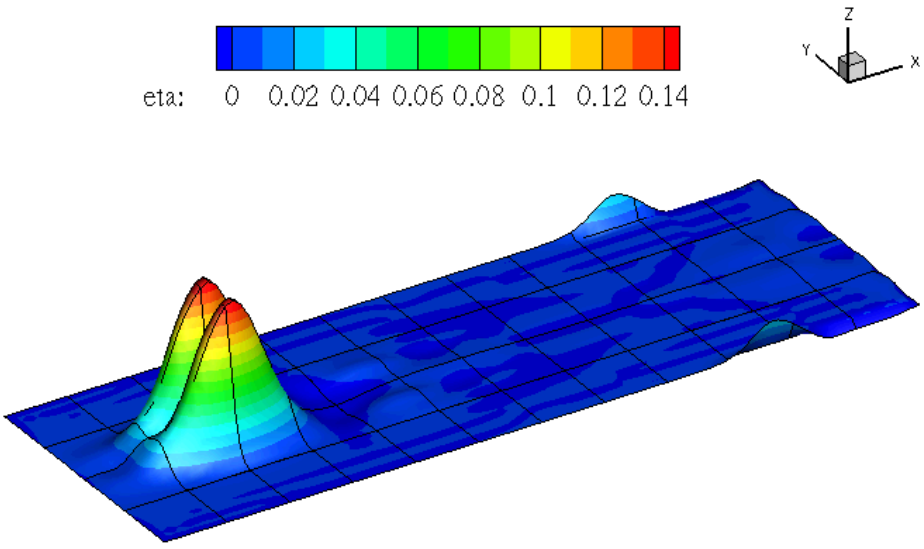
```
./ShallowWaterSolver Rossby_Nonlinear_DG.xml
```

13.4.1.3 Post-proceesing

After the final time step the solver will write an output file `RossbyModon_Nonlinear_DG.fld`. We can convert it to tecplot format by using the `FieldConvert` utility. Thus we execute the following command:

```
FieldConvert RossbyModon_Nonlinear_DG.xml RossbyModon_Nonlinear_DG.fld \
    RossbyModon_Nonlinear_DG.dat
```

This will generate a file called `RossbyModon_Nonlinear_DG.dat` that can be loaded directly into tecplot:



Part IV

Reference

Chapter 14

Optimisation

One of the most frequently asked questions when performing almost any scientific computation is: how do I make my simulation faster? Or, equivalently, why is my simulation running so slowly?

The spectral element method is no exception to this rule. The purpose of this chapter is to highlight some of the easiest parameters that can be tuned to attain optimum performance for a given simulation.

Details are kept as untechnical as possible, but some background information on the underlying numerical methods is necessary in order to understand the various options available and the implications that they can have on your simulation.

14.1 Operator evaluation strategies

When discretising a PDE using most variants of the spectral element method, the resulting problem is usually expressed as a matrix equation. In traditional linear finite element codes, the matrix is usually represented as a large sparse global matrix, which represents the action of a particular operator such as the Laplacian matrix across the whole domain.

However, when we consider spectral element methods, in which the polynomial order representing the expansion can be far higher, this method becomes far less optimal. We can instead consider the action of an operator locally on each element, and then perform an assembly operation. This is mathematically equivalent to the global matrix approach and gives exactly the same answer, but at high polynomial orders it is far more efficient on modern CPU architectures.

Furthermore, this local approach can be represented in one of two ways: either as a dense matrix for each element, which is typically more efficient at intermediate polynomial orders, or in the *hp* element case as a tensor product of smaller dense matrices via an approach deemed *sum-factorisation*, which is used at very high polynomial orders. Figure ?? gives an overview of these three different operator strategies.

A goal of *Nektar++* is to support not only high order expansions, but *all* orders from low (where element size h is the dominant factor) to high (where p dominates); a procedure we have dubbed "from h to p efficiently".

14.1.1 Selecting an operator strategy

An obvious question is: "which strategy should I select?" The most important factors in this decision are:

1. what the operator is;
2. polynomial order p;
3. element type and dimension of the problem;
4. underlying hardware architecture;
5. the number of operator calls in the solver;
6. BLAS implementation speed.

Generally you can use results from three publications [33, 6, 5] which outline results for two- and three-dimensional elements.

In general, the best approach is to perform some preliminary timings by changing the appropriate variables in the session file, which is outlined below. As a very rough guide, for $1 \leq p \leq 2$ you should use the global approach; for $3 \leq p \leq 7$ use the local approach; and for $p \geq 8$ use sum-factorisation. However, these guidelines will vary due to the parameters noted above. In future releases of *Nektar++* we hope to tune these variables automatically to make this decision easier to make.

14.1.2 XML syntax

Operator evaluation strategies can be configured in the `GLOBALOPTIMISATIONPARAMETERS` tag, which lies inside the root `NEKTAR` tag:

```
<NEKTAR>
  <GLOBALOPTIMIZATIONPARAMETERS>
    <BwdTrans>
      <DO_GLOBAL_MAT_OP VALUE="0" />
      <DO_BLOCK_MAT_OP TRI="1" QUAD="1"  TET="1"
                       PYR="1" PRISM="1" HEX="1" />
    </BwdTrans>
    <IProductWRTBase>
      <DO_GLOBAL_MAT_OP VALUE="0" />
      <DO_BLOCK_MAT_OP TRI="1" QUAD="1"  TET="1"
                       PYR="1" PRISM="1" HEX="1" />
    </IProductWRTBase>
    <HelmholtzMatrixOp>
```

```
      <DO_GLOBAL_MAT_OP VALUE="0" />
      <DO_BLOCK_MAT_OP TRI="1" QUAD="1"  TET="1"
                       PYR="1" PRISM="1" HEX="1" />
    </HelmholtzMatrixOp>
    <MassMatrixOp>
      <DO_GLOBAL_MAT_OP VALUE="0" />
      <DO_BLOCK_MAT_OP TRI="1" QUAD="1"  TET="1"
                       PYR="1" PRISM="1" HEX="1" />
    </MassMatrixOp>
  </GLOBALOPTIMIZATIONPARAMETERS>
</NEKTAR>
```

14.1.3 Selecting different operator strategies

Operator evaluation is supported for four operators: backward transform, inner product, Helmholtz and mass operators. It is possible to specify the following optimisation flags for different operators:

1. `DO_GLOBAL_MAT_OP`: If `VALUE` is `1`, the globally assembled system matrix will be used to evaluate the operator. If `VALUE` is `0`, the operator will be evaluated elementally.

2. `DO_BLOCK_MAT_OP`: If `VALUE` is `1`, the elemental evaluation will be done using the elemental/local matrices (which are all concatenated in a block matrix, hence the name). If `VALUE` is `0`, the elemental evaluation will be done using the sum-factorisation technique.

   Each element type (triangle, quadrilateral, etc) has its own `VALUE`, since break-even points for sum-factorisation and the local matrix approach will differ depending on element type. Note that due to a small shortcoming in the code, all element types must be defined; so three-dimensional elements must still be defined even if the simulation is two-dimensional.

Note that global takes precendence over block, so if `VALUE` is set to `1` for both then the operator will be global.

For very complex operators – in particular `HelmholtzMatrixOp` – always set `DO_BLOCK_MAT_OP` to `1` as sum-factorisation for these operator types can be costly.

14.2 Collections

The Collections library adds optimisations to perform certain elemental operations collectively by applying an operator using a matrix-matrix operation, rather than a sequence of matrix-vector multiplications. Certain operators benefit more than other from this treatment, so the following implementations are available:

- StdMat: Perform operations using collated matrix-matrix type elemental operation.
- SumFac: Perform operation using collated matrix-matrix type sum factorisation operations.
- IterPerExp: Loop through elements, performing matrix-vector operation.
- NoCollections: Use the original LocalRegions implementation to perform the operation.

All configuration relating to Collections is given in the `COLLECTIONS` XML element within the `NEKTAR` XML element.

14.2.1 Default implementation

The default implementation for all operators may be chosen through setting the `DEFAULT` attribute of the `COLLECTIONS` XML element to one of `StdMat`, `SumFac`, `IterPerExp` or `NoCollection`. For example, the following uses the collated matrix-matrix type elemental operation for all operators and expansion orders:

```
<COLLECTIONS DEFAULT="StdMat" />
```

14.2.2 Auto-tuning

The choice of implementation for each operator, for the given mesh and expansion orders, can be selected automatically through auto-tuning. To enable this, add the following to the *Nektar++* session file:

```
<COLLECTIONS DEFAULT="auto" />
```

This will collate elements from the given mesh and given expansion orders, run and time each implementation strategy in turn, and select the fastest performing case. Note that the selections will be mesh- and order- specific. The selections made via auto-tuning are output if the `-verbose` command-line switch is given.

14.2.3 Manual selection

The choice of implementation for each operator may be set manually within the `COLLECTIONS` tag as shown in the following example. Different implementations may be chosen for different element shapes and expansion orders. Specifying `*` for `ORDER` sets the default implementation for any expansion orders not explicity defined.

```
<COLLECTIONS>
    <OPERATOR TYPE="BwdTrans">
        <ELEMENT TYPE="T" ORDER="*"   IMPTYPE="IterPerExp" />
        <ELEMENT TYPE="T" ORDER="1-5" IMPTYPE="StdMat" />
    </OPERATOR>
    <OPERATOR TYPE="IProductWRTBase">
        <ELEMENT TYPE="Q" ORDER="*"   IMPTYPE="SumFac" />
```

```
8     </OPERATOR>
9 </COLLECTIONS>
```

Manual selection is intended to document the optimal selections on a given hardware platform after extensive prior testing, to avoid the need to run the auto-tuning for each run.

14.2.4 Collection size

The maximum number of elements within a single collection can be enforced using the `MAXSIZE` attribute.

Chapter 15

Command-line Options

`--verbose`
: Displays extra info.

`--version`
: Displays software version, and source control information if applicable.

`--help`
: Displays help information about the available command-line options for the executable.

`--parameter [key]=[value]`
: Override a parameter (or define a new one) specified in the XML file.

`--solverinfo [key]=[value]`
: Override a solverinfo (or define a new one) specified in the XML file.

`--shared-filesystem`
: By default when running in parallel the complete mesh is loaded by all processes, although partitioning is done uniquely on the root process only and communicated to the other processes. Each process then writes out its own partition to the local working directory. This is the most robust approach in accounting for systems where the distributed nodes do not share a common filesystem. In the case that there is a common filesystem, this option forces only the root process to load the complete mesh, perform partitioning and write out the session files for all partitions. This avoids potential memory issues when multiple processes attempt to load the complete mesh on a single node.

`--npx [int]`
: When using a fully-Fourier expansion, specifies the number of processes to use in the x-coordinate direction.

`--npy [int]`
: When using a fully-Fourier expansion or 3D expansion with two Fourier directions, specifies the number of processes to use in the y-coordinate direction.

`--npz [int]`
: When using Fourier expansions, specifies the number of processes to use in the z-coordinate direction.

`--part-info`
: Prints detailed information about the generated partitioning, such as number of elements, number of local degrees of freedom and the number of boundary degrees of freedom.

`--part-only [int]`
: Partition the mesh only into the specified number of partitions, write to file and exit. This can be used to pre-partition a very large mesh on a single high-memory node, prior to being executed on a multi-node cluster.

`--use-metis`
: Forces the use of METIS for mesh partitioning. If *Nektar++* is compiled with Scotch support, the default is to use Scotch.

`--use-scotch`
: Forces the use of Scotch for mesh partitioning.

Chapter 16

Frequently Asked Questions

16.1 Compilation and Testing

Q. I compile Nektar++ successfully but, when I run ctest, all the tests fail. What might be wrong?

On Linux or Mac, if you compile the ThirdParty version of Boost, rather than using version supplied with your operating system (or MacPorts on a Mac), the libraries will be installed in the `ThirdParty/dist/lib` subdirectory of your Nektar++ directory. When Nektar++ executables are run, the Boost libraries will not be found as this path is not searched by default. To allow the Boost libraries to be found set the following environmental variable, substituting `$NEKTAR_HOME` with the absolute path of your Nektar++ directory:

- On Linux (sh, bash, etc)

  ```
  export LD_LIBRARY_PATH=${NEKTAR_HOME}/ThirdParty/dist/lib
  ```

  or (csh, etc)

  ```
  setenv LD_LIBRARY_PATH ${NEKTAR_HOME}/ThirdParty/dist/lib
  ```

- On Mac

  ```
  export DYLD_LIBRARY_PATH=${NEKTAR_HOME}/ThirdParty/dist/lib
  ```

Q. How to I compile Nektar++ to run in parallel?

Parallel execution of all Nektar++ solvers is available using MPI. To compile using MPI, enable the `NEKTAR_USE_MPI` option in the CMake configuration. On recent versions of

MPI, the solvers can still be run in serial when compiled with MPI. More information on Nektar++ compilation options is available in Section 1.3.5.

Q. When compiling Nektar++, I receive the following error:

```
CMake Error: The following variables are used in this
oject, but they are set  to NOTFOUND.
Please set them or make sure they are set and tested
rrectly in the CMake files:
NATIVE_BLAS (ADVANCED)
    linked by target "LibUtilities" in directory
    /path/to/nektar++/library/LibUtilities
NATIVE_LAPACK (ADVANCED)
    linked by target "LibUtilities" in directory
    /path/to/nektar++/library/LibUtilities
```

This is caused by one of two problems:

- The BLAS and LAPACK libraries and development files are not installed. On Linux systems, both the LAPACK library package (usually called liblapack3 or lapack) and the development package (usually called liblapack-dev or lapack-devel) must be installed. Often the latter is missing.
- An alternative BLAS/LAPACK library should be used. HPC systems frequently use the Intel compilers (icc, icpc) and the Intel Math Kernel Library (MKL). This software should be made available (if using the modules environment) and the option `NEKTAR_USE_MKL` should be enabled.

Q. When I compile Nektar++ I receive an error

```
error: #error "SEEK_SET is #defined but must not be for
the C++ binding of MPI. Include mpi.h before stdio.h"
```

This can be fixed by including the flags

```
-DMPICH_IGNORE_CXX_SEEK -DMPICH_SKIP_MPICXX
```

in the `CMAKE_CXX_FLAGS` option within the `ccmake` configuration.

Q. After installing Nektar++ on my local HPC cluster, when I run the 'ctest' command, all the parallel tests fail. Why is this?

The parallel tests are those which include the word `parallel` or `par`. On many HPC systems, the MPI binaries used to execute jobs are not available on the login nodes, to prevent inadvertent parallel runs outside of the queuing system. Consequently, these

tests will not execute. To fully test the code, you can submit a job to the queuing system using a minimum of two cores, to run the `ctest` command.

Q. When running any Nektar++ executable on Windows, I receive an error that zlib.dll cannot be found. How do I fix this?

Windows searches for DLL files in directories specified in the PATH environmental variable. You should add the location of the ThirdParty files to your path. To fix this (example for Windows XP):

- As an administrator, open "System Properties" in control panel, select the "Advanced" tab, and select "Environment Variables".
- Edit the system variable 'path' and append

  `C:\path\to\nektar++\ThirdParty\dist\bin`

  to the end, replacing `path\to\nektar++` appropriately.

Q. When compiling Nektar++ Thirdparty libraries I get an error "CMake Error: Problem extracting tar"

Nektar++ tries to download the appropriate ThirdParty libraries. However if the download protocols are restricted on your computer this may fail leading to the error "'CMake Error: Problem extracting tar". These libraries are available from

`http://www.nektar.info/thirdparty/`

and can be downloaded directly into the `$NEKTAR_HOME/ThirdParty` directory

16.2 Usage

Q. How do I run a solver in parallel?

In a desktop environment, simply prefix the solver executable with the `mpirun` helper. For example, to run the Incompressible Navier-Stokes solver on a 4-core desktop computer, you would run

```
mpirun -np 4 IncNavierStokesSolver Cyl.xml
```

In a cluster environment, using PBS for example, the `mpiexec` command should be used.

Q. How can I generate a mesh for use with Nektar++?

Nektar++ supports a number of mesh input formats. These are converted to the Nektar++ native XML format (see Section 3) using the NekMesh utility (see Section 4. Supported formats include:

- Gmsh (.msh)
- Polygon (.ply)
- Nektar (.rea)
- Semtex (.sem)

Bibliography

[1] M Ainsworth and S Sherwin. Domain decomposition preconditioners for p and hp finite element approximation of stokes equations. *COMPUTER METHODS IN APPLIED MECHANICS AND ENGINEERING*, 175:243–266, 1999.

[2] R. R. Aliev and A. V. Panfilov. A simple two-variable model of cardiac excitation. *Chaos, Solitons & Fractals*, 7:293–301, 1996.

[3] Ivo Babuška and Manil Suri. The p and h-p versions of the finite element method, basic principles and properties. *SIAM review*, 36(4):578–632, 1994.

[4] P-E Bernard, J-F Remacle, Richard Comblen, Vincent Legat, and Koen Hillewaert. High-order discontinuous galerkin schemes on general 2d manifolds applied to the shallow water equations. *Journal of Computational Physics*, 228(17):6514–6535, 2009.

[5] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p efficiently: selecting the optimal spectral/hp discretisation in three dimensions. *Math. Mod. Nat. Phenom.*, 6:84–96, 2011.

[6] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p efficiently: strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids*, 43:23–28, 2011.

[7] CD Cantwell, D Moxey, A Comerford, A Bolis, G Rocco, G Mengaldo, D De Grazia, S Yakovlev, J-E Lombard, D Ekelschot, et al. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[8] D. De Grazia, G. Mengaldo, D. Moxey, P. E. Vincent, and S. J. Sherwin. Connections between the discontinuous galerkin method and high-order flux reconstruction schemes. *International Journal for Numerical Methods in Fluids*, 75(12):860–877, 2014.

[9] S. Dong, G. E. Karniadakis, and C. Chryssostomidis. A robust and accurate outflow boundary condition for incompressible flow simulations on severely-truncated unbounded domains. *Journal of Computational Physics*, 261:95–136, 2014.

[10] AP Dowling and Ffowcs-Williams JE. Sound and sources of sound. *Ellis Horwood series in engineering science*, 1983.

[11] Niederer "et al.". Verification of cardiac tissue electrophysiology simulators using an n-version benchmark. *Philos Transact A Math Phys Eng Sci*, 369:4331–51, 2011.

[12] David Gottlieb, Steven A Orszag, and CAMBRIDGE HYDRODYNAMICS INC MA. *Numerical analysis of spectral methods*. SIAM, 1977.

[13] J.L. Guermond and J. Shen. Velocity-correction projection methods for incompressible flows. *SIAM J. Numer. Anal.*, 41:112–134, 2003.

[14] Jan S Hesthaven and Tim Warburton. Nodal high-order methods on unstructured grids: I. time-domain solution of maxwell's equations. *Journal of Computational Physics*, 181(1):186–221, 2002.

[15] B. E. Jordi, C. J. Cotter, and S. J. Sherwin. Encapsulated formulation of the selective frequency damping method. *Phys. Fluids*, 2014.

[16] G. E. Karniadakis, M. Israeli, and S. A. Orszag. High-order splitting methods for the incompressible Navier–Stokes equations. 97(2):414–443, 1991.

[17] Jonas Koko. Vectorized matlab codes for linear two-dimensional elasticity. *Scientific Programming*, 15(3):157–172, 2007.

[18] C. H. Luo and Y. Rudy. A model of the ventricular cardiac action potential. depolarization repolarization and their interaction. *Circulation research*, 68:1501–1526, 1991.

[19] R. J. Ramirez M. Courtemanche and S. Nattel. Ionic mechanisms underlying human atrial action potential properties: insights from a mathematical model. *American Journal of Physiology-Heart and Circulatory Physiology*, 275:H301–H321, 1998.

[20] Y. Maday, A. T. Patera, and E.M. Ronquist. An operator-integration-factor splitting method for time-dependent problems: Application to incompressible fludi flow. *J. Sci. Comp.*, 4:263–292, 1990.

[21] Gianmarco Mengaldo, Daniele De Grazia, Freddie Witherden, Antony Farrington, Peter Vincent, Spencer Sherwin, and Joaquim Peiro. *A Guide to the Implementation of Boundary Conditions in Compact High-Order Methods for Compressible Aerodynamics*. American Institute of Aeronautics and Astronautics, 2014/08/10 2014.

[22] D. Moxey, M. Hazan, J. Peiró, and S. J. Sherwin. An isoparametric approach to high-order curvilinear boundary-layer meshing. *Comp. Meth. Appl. Mech. Eng.*, 2014.

[23] D. Moxey, M. Hazan, J. Peiró, and S. J. Sherwin. On the generation of curvilinear meshes through subdivision of isoparametric elements. to appear in proceedings of Tetrahedron IV, 2014.

[24] David J Newman and George Em Karniadakis. A direct numerical simulation study of flow past a freely vibrating cable. *Journal of Fluid Mechanics*, 344:95–136, 1997.

[25] Anthony T Patera. A spectral element method for fluid dynamics: laminar flow in a channel expansion. *Journal of computational Physics*, 54(3):468–488, 1984.

[26] N Pignier. One-dimensional modelling of blood flow in the cardiovascular system, 2012.

[27] CJ Roth. Pulse wave propagation in the human vascular system, 2012.

[28] S Sherwin. A substepping navier-stokes splitting scheme for spectral/hp element discretisations. pages 43–52. Elsevier Science, 2003.

[29] SJ Sherwin and M Ainsworth. Unsteady navier-stokes solvers using hybrid spectral/hp element methods. *APPLIED NUMERICAL MATHEMATICS*, 33:357–363, 2000.

[30] SJ Sherwin, L Formaggia, J Peiró, and V Franke. Computational modelling of 1d blood flow with variable mechanical properties and its application to the simulation of wave propagation in the human arterial system. *Int. J. Numer. Meth. Fluids*, 43:673–700, 2003.

[31] SJ Sherwin and G Em Karniadakis. Tetrahedral< i> hp</i> finite elements: Algorithms and flow simulations. *Journal of Computational Physics*, 124(1):14–45, 1996.

[32] K. H. W. J. ten Tusscher and A. V. Panfilov. Alternans and spiral breakup in a human ventricular tissue model. *American Journal of Physiology-Heart and Circulatory Physiology*, 291:H1088–H1100, 2006.

[33] Peter EJ Vos, Spencer J Sherwin, and Robert M Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low-and high-order discretisations. *Journal of Computational Physics*, 229(13):5161–5181, 2010.

[34] N Westerhof. Anatomic studies of the human systemic arterial tree. *J. Biomech.*, 2:121–143, 1969.

[35] D Xiu, SJ Sherwin, S Dong, and GE Karniadakis. Strong and auxiliary forms of the semi-lagrangian method for incompressible flows. *J. Sci. Comp.*, 25:323–346, 2005.

[36] Olgierd Cecil Zienkiewicz and Robert Leroy Taylor. *Basic formulation and linear problems*. McGraw-Hill, 1989.