Nektar++: Spectral/hp Element Framework

Version 5.7.0

User Guide

October 30, 2024

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

Contents

Introduction

Nektar++ [6] 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 12.0 (bookworm)	`deb http://www.nektar.info/debian-bookworm bookworm contrib`
Debian 11.0 (bullseye)	`deb http://www.nektar.info/debian-bullseye bullseye contrib`
Ubuntu 24.04 (Noble Numbat)	`deb http://www.nektar.info/ubuntu-noble noble contrib`
Ubuntu 22.04 (Jammy Jellyfish)	`deb http://www.nektar.info/ubuntu-jammy jammy contrib`
Ubuntu 20.04 (Focal Fossa)	`deb http://www.nektar.info/ubuntu-focal focal 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 36	`http://www.nektar.info/fedora/36/$basearch`
Fedora 35	`http://www.nektar.info/fedora/35/$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 https://gitlab.nektar.info/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 by running
`git remote set-url origin git@gitlab.nektar.info:nektar/nektar.git`

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*. Typically, the development packages, with a *-dev* or *-devel* suffix, are required to compile codes against these libraries. 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, supporting C++11
CMake $\geq$ 3.5.1	✓	✓	✓		Ncurses GUI optional
BLAS	✓	✓	✓	✓	Or MKL, ACML, OpenBLAS
LAPACK	✓	✓	✓	✓	
Boost >= 1.56	✓	✓	✓	✓	Compile with iostreams
TinyXML	✓	✓	✓	✓	For reading XML input files
Scotch	✓	✓	✓	✓	Required for multi-level static condensation, highly recommended
METIS		✓	✓	✓	Alternative mesh partitioning
FFTW > 3.0		✓	✓	✓	For high-performance FFTs
ARPACK > 2.0		✓	✓	✓	For arnoldi algorithms
MPI		✓	✓		For parallel execution (OpenMPI, MPICH, Intel MPI, etc)
GSMPI				✓	For parallel execution
HDF5		✓	✓	✓	For large-scale parallel I/O (requires CMake >3.1)
OpenCascade CE		✓	✓	✓	For mesh generation and optimisation
PETSc		✓	✓	✓	Alternative linear solvers
PT-Scotch		✓	✓	✓	Required when MPI enabled
Tetgen		✓	✓	✓	For 3D mesh generation
Triangle		✓	✓	✓	For 2D mesh generation
VTK > 5.8		✓	✓		Not required to convert field output files to VTK, only mesh files

1.3.2.2 Quick Start

Open a terminal.

If you have downloaded the tarball, first unpack it:

```
tar -zxvf nektar++-5.7.0.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++-5.7.0.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_`). Some of these will be automatically enabled if not found on your system.
   - Choose the installation location by adjusting `CMAKE_INSTALL_PREFIX`. By default, this will be a `dist` subdirectory within the `build` directory, which is satisfactory for most users initially.

   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.

If you have installed additional system packages since running CMake, you may need to wipe your build directory and rerun CMake for them to be detected.

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).

Package	Req.	Installation			Note
		MacPorts	User	Auto.	
Xcode	✓				Provides developer tools
CMake $\geq$ 3.5.1	✓	`cmake`	✓		Ncurses GUI optional
BLAS	✓				Part of Xcode
LAPACK	✓				Part of Xcode
Boost >= 1.56	✓	`boost`	✓	✓	Compile with iostreams
TinyXML	✓	`tinyxml`	✓	✓	
Scotch	✓	`scotch`	✓	✓	Required for multi-level static condensation, highly recommended
METIS		`metis`	✓	✓	Alternative mesh partitioning
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
HDF5			✓	✓	For large-scale parallel I/O (requires CMake >3.1)
OpenCascade CE			✓	✓	For mesh generation and optimisation
PETSc		`petsc`	✓	✓	Alternative linear solvers
PT-Scotch			✓	✓	Required when MPI enabled
Tetgen			✓	✓	For 3D mesh generation
Triangle			✓	✓	For 2D mesh generation
VTK > 5.8		`vtk`	✓		Not required to convert field output files to VTK, only mesh files

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++-5.7.0.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++-5.7.0.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_`)
   - Choose the installation location by adjusting `CMAKE_INSTALL_PREFIX`. By default, this will be a `dist` subdirectory within the `build` directory, which is satisfactory for most users initially.

   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 there are some complexities with building additional features on this platform at present. As such, only builds with a minimal amount of additional build packages are currently supported. These can either be installed by the user, or automatically as part of the build process. Support has recently been added for building with MPI on Windows. This enables parallel computations to be carried out with *Nektar++* on Windows where only sequential computations were previously supported.

Package	Req.	Installation User	Installation Auto.	Note
MS Visual Studio	✓	✓		2015, 2017, 2019 and 2022 known working
CMake $\geq$ 3.5.1	✓	✓		3.16+ recommended, see info below
BLAS	✓	✓	✓	
LAPACK	✓	✓	✓	
Boost $\geq$ 1.61	✓	✓	✓	Recommend installing from binaries
Microsoft MPI $\geq$ 10.1.2		✓		Required for parallel execution. Install both runtime and SDK

Note

These instructions assume you are using a 64-bit version of Windows 10.

Note

There have been issues with automatically building Boost from source as a third party dependency during the *Nektar++* build when using MS Visual Studio 2015, 2017 and 2019. This should now be possible but it is, nonetheless, recommended you install a suitable version of Boost from binaries as detailed in the instructions below.

1.3.4.1 Detailed instructions

1. Install Microsoft Visual Studio 2022, 2019 (preferred), 2017 or 2015 (both known to work). This can be obtained from Microsoft free of charge by using their Community developer tools from https://visualstudio.microsoft.com/vs/community/.
2. Install CMake from http://www.cmake.org/download/. For building on Windows you are strongly recommended to use a recent version of CMake, e.g 3.16+. Minimum required CMake versions for building *Nektar++* on Windows with Visual Studio are CMake 3.5.1+ (VS2015), 3.7+ (VS2017), 3.15+ (VS2019) or 3.21+ (VS2022). When prompted, select the option to add CMake to the system PATH.
3. (Optional) As highlighted above, it is possible to have Boost built from source as a third-party library during the *Nektar++* build. However, it is currently recommended to install the Boost binaries that can be found at http://sourceforge.net/projects/boost/files/boost-binaries. By default these install into `C:\local\boost_<version>`. We recommend installing a specific version of the binaries depending on the version of Visual Studio you are using, these are known to be working with the *Nektar++* build:

- For Visual Studio 2015, install boost 1.61 using the package `boost_1_61_0-msvc-14.0-64.exe` from http://sourceforge.net/projects/boost/files/boost-binaries/1.61.0/
- For Visual Studio 2017, install boost 1.68 using the package `boost_1_68_0-msvc-14.1-64.exe` from http://sourceforge.net/projects/boost/files/boost-binaries/1.68.0/
- For Visual Studio 2019, install boost 1.72 using the package `boost_1_72_0-msvc-14.2-64.exe` from http://sourceforge.net/projects/boost/files/boost-binaries/1.72.0/
- For Visual Studio 2022, install boost 1.78 using the package `boost_1_78_0-msvc-14.3-64.exe` from http://sourceforge.net/projects/boost/files/boost-binaries/1.78.0/

If you use these libraries, you will need to:

- Add a `BOOST_HOME` environment variable. To do so, click the Start menu and type 'env', you should be presented with an *"Edit the system environment variables"* option. Alternatively, from the Start menu, navigate to *Settings* > *System* > *About* > *System info (under Related Settings on the right hand panel)*, select *Advanced System Settings*, and in the *Advanced* tab click the *Environment Variables* button. In the *System variables* box, click *New*. In the *New System Variable* window, type `BOOST_HOME` next to *Variable name* and `C:\local\<boost_dir>` next to *Variable value*, where *<boost_dir>* corresponds to the directory that boost has been installed to, based on the boost version you have installed. (e.g. *boost_1_61_0*, *boost_1_68_0*, *boost_1_72_0*, *boost_1_78_0*).

4. (Optional) Install Git for Windows from https://gitforwindows.org/ to use the development versions of *Nektar++*. You can accept the default set of components in the *Select Components* panel. When prompted, in the *"Adjusting your PATH environment"* panel, select the option *"Git from the command line and also from 3rd-party software"*. You do not need to select the option to add Unix tools to the PATH.

5. If you've downloaded the source code archive (as described in Section 1.3.1), unpack `nektar++-5.7.0.zip`.

Note

Some Windows versions do not recognise the path of a folder which has `++` in the name. If you are not using Windows 10 and think that your Windows version cannot handle paths containing special characters, you should rename `nektar++-5.7.0` to `nektar-5.7.0`.

6. Create a `build` directory within the `nektar++-5.7.0` subdirectory. *If you cloned the source code from the git repository, your* Nektar++ *subdirectory will be called* ***`nektar`*** *rather than* ***`nektar++-5.7.0`***

7. Open a Visual Studio terminal (*Developer Command Prompt for VS [2015/2017/2019/2022]* or *x64 Native Tools Command Prompt*. From the Start menu, this can be found under *Visual Studio [2015/2017/2019/2022]*.

8. Change directory into the `build` directory and run CMake to generate the build files. You need to set the *generator* to the correct Visual Studio version using the -G switch on the command line, e.g. for VS2019:

   ```
   cd C:\path\to\nektar\builds
   cmake -G "Visual Studio 16 2019" ..
   ```

   You can see the list of available generators using *cmake –help*. For VS2017 use *"Visual Studio 15 2017 Win64"* and for VS2015 use *"Visual Studio 14 2015 Win64"*. Alternatively, run:

   ```
   cmake-gui
   ```

   to see the available generators and compile options.

   If you want to build a parallel version of *Nektar++* with MPI support, you need to add the *-DNEKTAR_USE_MPI=ON* switch to the cmake command, e.g.:

   ```
   cmake -G "Visual Studio 16 2019" -D NEKTAR_USE_MPI=ON ..
   ```

   Note

   If you installed Boost binaries, as described above, you should ensure at this stage that the version of Boost that you installed has been correctly detected by CMake. You should see a number of lines of output from CMake saying *– – Found boost <library name> library:* followed by paths to one or more files which should be located in the directory where you installed your Boost binaries. If you do not see this output, CMake has failed to detect the installed Boost libraries and the build process will instead try to build Boost from source as part of building *Nektar++*.

   If you experience any issues with CMake finding pre-installed Boost, binaries ensure that you are working in a Visual Studio command prompt that was opened *after* you installed boost and set up the `BOOST_HOME` environment variable.

9. Assuming the configuration completes successfully and you see the message *Build files have been written to: ...*, you should now be ready to issue the build 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
```

10. After the build and installation process has completed, the executables will be available in `build\dist\bin`.

11. To use these executables, you need to modify your system `PATH` to include the `bin` directory and library directories where DLLs are stored. To do this, click the Start menu and type 'env', you should be presented with an *"Edit the system environment variables"* option. Alternatively, navigate to *Settings > System > About > System info (under Related Settings on the right hand panel)*, select *Advanced System Settings*, and in the *Advanced* tab click the *Environment Variables* button. In the *System Variables* box, select `Path` and click *Edit*. Add the **full paths** to the following directories to the end of the list of paths shown in the *"Edit environment variable"* window:

    - `nektar++-5.7.0\build\dist\lib\nektar++-5.7.0`
    - `nektar++-5.7.0\build\dist\bin`
    - `nektar++-5.7.0\ThirdParty`
    - `C:\local\boost_<boost_version>\<boost_lib_dir>` where *boost_<boost_version>* is the directory where the boost binaries were installed to and *<boost_lib_dir>* is the name of the library directory within this location, e.g. *lib64-msvc-14.2* or similar depending on the version of Boost binaries you installed.

12. To run the test suite, open a **new** command line window, change to the `build` 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_PYTHON`

  Installs the Python wrapper to *Nektar++*. Requires running the following command after installing *Nektar++* in order to install the Python package for the current user:

  ```
  make nekpy-install-user
  ```

  Alternatively, the Python package can be installed for all users by running the following command with appropriate priviledges:

  ```
  make nekpy-install-system
  ```

- `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, including the mesh conversion and generation tool `NekMesh` and the `FieldConvert` post-processing utility.

- `NEKTAR_SOLVER_X`

  Enable compilation of the 'X' solver.

- `NEKTAR_UTILITY_X`

  Enable compilation of the 'X' utility.

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_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_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_CWIPI`

  Use the CWIPI library for enabling inter-process communication between two solvers. Solvers may also interface with third-party solvers using this 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_HDF5`

  Build *Nektar++* with support for HDF5. This enables input/output in the HDF5 parallel file format, which can be very efficient for large numbers of processes. HDF5 output can be enabled by using a command-line option or in the `SOLVERINFO` section of the XML file. This option requires that *Nektar++* be built with MPI support with `NEKTAR_USE_MPI` enabled and that HDF5 is compiled with MPI support.

- `NEKTAR_USE_MESHGEN`

  Build the NekMesh utility with support for generating meshes from CAD geometries. This enables use of the OpenCascade Community Edition library, as well as Triangle and Tetgen.

- `NEKTAR_USE_METIS`

  Build *Nektar++* with support for the METIS graph partitioning library. This provides both an alternative mesh partitioning algorithm to SCOTCH for parallel simulations.

- `NEKTAR_USE_MPI` (Recommended)

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

- `NEKTAR_USE_PETSC`

  Build *Nektar++* with support for the PETSc package for solving linear systems. The user has the option to use either their own PETSc installations, or use the packaged PETSc version 3.19.3. To use your own PETSc install, specify the variables `PETSC_DIR` and `PETSC_ARCH` in your bash session or `.bashrc` before the *Nektar++* compilation step using the following commands:

  ```
  export PETSC_DIR=/path/to/petsc
  export PETSC_ARCH=name-of-petsc_arch
  ```

  and then set the `LD_LIBRARY_PATH` and `PKG_CONFIG_PATH`

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PETSC_DIR/$PETSC_ARCH/lib/
  export PKG_CONFIG_PATH=$PETSC_DIR/$PETSC_ARCH/lib/pkgconfig/:
      $PKG_CONFIG_PATH
  ```

- `NEKTAR_USE_PYTHON3` (Requires `NEKTAR_BUILD_PYTHON`)

  Enables the generation of Python3 interfaces.

- `NEKTAR_USE_SCOTCH` (Recommended)

  Build *Nektar++* with support for the SCOTCH graph partitioning library. This provides both a mesh partitioning algorithm for parallel simulations and enabled support for multi-level static condensation, so is highly recommended and enabled by default. However for systems that do not support SCOTCH build requirements (e.g. Windows), this can be disabled.

- `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_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:

- `ARPACK`

  Library of iterative Arnoldi algorithms.

- `BLAS_LAPACK`

  Library of linear algebra routines.

- `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.

- `CCMIO`

  I/O library for the Star-CCM+ format.

- `CWIPI`

  Library for inter-process exchange of data between different solvers.

- `FFTW`

  Fast-Fourier transform library.

- `GSMPI`

  (MPI-only) Parallel communication library.

- `HDF5`

  Hierarchical Data Format v5 library for structured data storage.

- `METIS`

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

- `OCE`

  OpenCascade Community Edition 3D modelling library.

- `PETSC`

  A package for the parallel solution of linear algebra systems. If this is switched on, then a packaged PETSc version 3.19.3 will be installed within the *Nektar++* installation.

- `SCOTCH`

  A graph partitioning library used for mesh partitioning when *Nektar++* is run in parallel, and reordering routines that are used in multi-level static condensation.

- `TETGEN`

  3D tetrahedral meshing library.

- `TINYXML`

  Library for reading and writing XML files.

- `TRIANGLE`

  2D triangular meshing library.

There are also a number of additional options to fine-tune the build:

- `NEKTAR_TEST_ALL`

  Enables an extra set of more substantial and long-running tests.

- `NEKTAR_TEST_USE_HOSTFILE`

  By default, MPI tests are run directly with the `mpiexec` command together with the number of cores. If your MPI installation requires a hostfile, enabling this option adds the command line argument `-hostfile hostfile` to the command line arguments when tests are run with `ctest` or the `Tester` executable.

We have recently added explicit support to SIMD (Single Instruction Multiple Data) x86 instruction set extensions (i.e. AVX2, AVX512). Selected operators (the matrix free operators) utilize the SIMD types, if none of them is enabled these operators default to scalar types. The various extensions available are marked as advanced options (to visualize them in the cmake gui you need to press the t-button):

- `NEKTAR_ENABLE_SIMD_AVX2`

  Enables 256 bit wide vector types and set the appropriate compiler flags (gcc only).

- `NEKTAR_ENABLE_SIMD_AVX512`

  Enables 512 bit wide vector types and set the appropriate compiler flags (gcc only).

Note that if you are not configuring cmake for the first time, you need to delete the cached variable `CMAKE_CXX_FLAGS` in order for the appropriate flags to be set. Alternatively you can manually set the flag to target the appropriate architecture.

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 [45] and more recently have gained greater popularity in the modelling of wave-based phenomena such as computational electromagnetics [19] and shallow water problems [5] - 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 [52] 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 [18] 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 [52].

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 [18], 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 [38] 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 first-level 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.

The contents of 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.

When generating mesh input files for *Nektar++* using `NekMesh`, the binary compressed form will be used by default. To convert a compressed XML file into human-readable ASCII format use, for example:

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

Note

The description in the remainder of this section explains how the `GEOMETRY` section is laid out in uncompressed ASCII format.

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

```
1 <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

```
1 <T ID="0"> 0 1 2 </T>
2 <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

```
1 <T ID="0"> 0 1 2 </T>
2 <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, using ranges, or a mixture of the two. Examples include

```
<C ID="0"> T[0-862] </C>
<C ID="1"> E[61-67,69,70,72-74] </C>
```

The composites can also optionally contain a name which is then used in the multi-block VTK output to label the block descriptively rather than by ID, for example

```
<C NAME="Main domain" ID="0"> T[0-862] </C>
<C NAME="Walls" ID="1"> E[61-67,69,70,72-74] </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 and variables. Expansion entries specify the number of modes and the expansion type, or a full list of data of basis type, number of modes, points type and number of points. 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" />
```

The expansions can be defined with a list of `<E>` elements (e.g., to represent different polynomial orders for different variables or to address different composites). The user can define a default expansion field by entering `<E>` tags without the `FIELDS` attribute. The default expansion is used to define any variables not explicitly listed in the `<E>`

entries. In the following example, the default expansion is used to define the expansions for the composites C[0], C[1] and C[2]:

```
<E COMPOSITE="C[0-2]" NUMMODES="5" TYPE="MODIFIED" />
<E COMPOSITE="C[3]"   NUMMODES="4" TYPE="MODIFIED" FIELDS="u,v"/>
<E COMPOSITE="C[3]"   NUMMODES="3" TYPE="MODIFIED" FIELDS="p"/>
```

The expansions of each field should be defined only once for each composite.

3.3 Refinements

This section explains how to define a local p-refinement in a specific region in a mesh. After the user defines the polynomial expansions to be used, one can also locally change the polynomial order in a specific region in a mesh. Firstly, we introduce the expansion entry `REFIDS` which specifies the reference ID for a local p-refinement. In order words, this is the ID that connects the composite to the local p-refinements to be performed. This expansion entry must be added in the list of `<E>` elements as follows

```
<E COMPOSITE="C[0]" NUMMODES="3" FIELDS="u" TYPE="MODIFIED" REFIDS="0" />
```

Then, in the `REFINEMENTS` section under `NEKTAR` tag as shown below, the local p-refinements are defined.

```
<REFINEMENTS>
   ...
</REFINEMENTS>
```

The refinements entries are the reference ID (`REF`) which must match the one determined in the list of elements, the type of the method (`TYPE`) to be used, the radius, the coordinates and the number of modes. The example below shows the entries when only the expansion type is provided. Note that the local p-refinements are set as a list of `<R>` refinement regions.

```
<R REF="0"
   TYPE="STANDARD"
   RADIUS="0.1"
   COORDINATE1="0.1,0.2,0.1"
   COORDINATE2="0.5,1.0,0.8"
   NUMMODES="5" />
```

There are currently two methods implemented. The methods are `STANDARD` and `SPHERE`. In these methods, the elements which the vertices lay within the surface (region) are refined based on the number of modes provided. Note that the example above is for the `STANDARD` method which creates a cylindrical surface for three-dimensional space problems. The `COORDINATE1` and `COORDINATE2` give the bottom and the top of the cylindrical surface and the `RADIUS` provides its radius. For two-dimensional meshes in the xy-plane, a parallelogram surface is defined based on the same entries. The radius entry, in two-dimension, gives the length (diameter of a circle) of one pair of parallel sides. In a one-dimensional problem (mesh in the x-axis only), a line is created and the

radius entry gives a extra length in both coordinates. Also, one important aspect to notice of this method is that the mesh dimension must match the space dimension. In order to set up a two- or one-dimensional surface, the user must provide the number of coordinates in `COORDINATE1` and `COORDINATE2` which match the space dimension.

One the other hand, the `SPHERE` method does not require that the space dimension matches the mesh dimension. In addition, two- and one-dimensional meshes can be refined even if they exist in a three-dimensional space. Thus, although the `STANDARD` method has additional geometrical flexibility to define the desired surface in the domain, the `SPHERE` method is more general so that the user can apply it in any mesh. Regarding the set up for the `SPHERE` method, the user must provide only `COORDINATE1` which defines the center of the sphere.

When the expansion basis is specified in detail as a combination of one-dimensional bases in the list of elements under the `EXPANSIONS` tag as shown below. The user also defines the number of quadrature points as explained in the previous section.

```
<E COMPOSITE="C[0]"
   BASISTYPE="Modified_A,Modified_A,Modified_A"
   NUMMODES="3,3,3"
   POINTSTYPE="GaussLobattoLegendre,GaussLobattoLegendre,GaussLobattoLegendre"
   NUMPOINTS="5,5,5"
   FIELDS="u"
   REFIDS="0" />
```

Thus, an additional entry must be provided in the list of refinement regions when a detailed description of the expansion basis is given. In this case, the number of quadrature points have to be also provided as follows

```
<R REF="0"
   RADIUS="0.1"
   TYPE="SPHERE"
   COORDINATE1="0.1,0.2,0.1"
   NUMMODES="5,5,5"
   NUMPOINTS="7,7,7" />
```

The *p*-refinement capability also allows the user to define multiple reference IDs (refinement regions) for each composite (see below). In other words, one can change the polynomial order in many locations in a mesh for a specific composite. It should be noted that if the user defines a region which is outside of the corresponding composite, the mesh is not going to be refined in the specified region.

```
<E COMPOSITE="C[0]" NUMMODES="3" FIELDS="u" TYPE="MODIFIED" REFIDS="0,1,2" />
<E COMPOSITE="C[1]" NUMMODES="5" FIELDS="u" TYPE="MODIFIED" REFIDS="3,4" />
```

The local p-refinement is only supported by CG discretisation at the moment.

3.4 Conditions

This 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.

3.4.1 Parameters

Numerical parameters may be required by a particular solver (for instance time-integration or physical parameters), or may be arbitrary and only used for the purpose of simplifying the problem specification in the session file (e.g. parameters which would otherwise be repeated in the definition of an initial condition and boundary conditions). 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>
```

A number of pre-defined constants may also be used in parameter expressions, for example `PI`. A full list of supported constants is provided in Section 3.8.1.2.

3.4.2 Time Integration Scheme

These specify properties to define the parameters specific to the time integration scheme to be used. The parameters are specified as XML elements and require a string corresponding to the time-stepping method and the order, and optionally the variant and free parameters. For example,

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> IMEX </METHOD>
  <VARIANT> DIRK </VARIANT>
  <ORDER> 2 </ORDER>
  <FREEPARAMETERS> 2 3 </FREEPARAMETERS>
</TIMEINTEGRATIONSCHEME>
```

For additional details on the different time integration schemes refer to the developer's guide.

3.4.3 Solver Information

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

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

Boolean-valued solver properties are specified using `True` or `False`. The list of available solvers in Nektar++ can be found in Part III.

3.4.3.1 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:

```
<TIMEINTEGRATIONSCHEME>
   <METHOD> IMEX </METHOD>
   <ORDER> 2 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
   <I PROPERTY="EQTYPE"               VALUE="UnsteadyNavierStokes"     />
   <I PROPERTY="SolverType"           VALUE="VelocityCorrectionScheme" />
   <I PROPERTY="Projection"           VALUE="Galerkin"                 />
   <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. **??** and 11.5.1, 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. 11.1.4) to obtain a steady-state solution of the Navier-Stokes equations (compressible or incompressible).

3.4.4 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>
```

3.4.5 Global System Solution Algorithm

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.

By default, iterative linear solvers uses a relative tolerance specified in the above exmaple. Setting the property `<I PROPERTY="AbsoluteTolerance" VALUE="True">` will override the default and force the solver to use the absolute tolerance. Note that the value of tolerance is still controlled by setting `<I PROPERTY=IterativeSolverTolerance" VALUE="1.e-8"/>` and if not set the default tolerance will be used.

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 specified there.

3.4.5.1 `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 3.4.5.5 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.

Warning

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`.

3.4.5.2 Iterative options

- **LinSysIterSolver** which specifies which iterative solver strategy to use and includes values of **ConjugateGradient** or **GMRES**. The default option is to solve this in a global degree of freedom array for continuous Galerkin discretisations but there are also options of **ConjugateGradientLoc** and **GMRESLoc** which solve the array in local element format. For DG discretisations these should be the same.

- **NekLinSysMaxIterations** specifies the maximum number of iterations and has a default value of 5000.

- **IterativeSolverTolerance** specifies the relative stopping tolerances for the iterative solver. Default is $1e-9$.

3.4.5.3 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
`Jacobi`	2/3D	All
`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.

3.4.5.4 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 or right-hand sides, dictated by the setting of the `SuccessiveRHS` option, to give a better initial guess for the iterative process. This method is better than any linear extrapolation method.

It can be activated by setting

```
<GLOBALSYSSOLNINFO>
    <V VAR="u,v,w">
        <I PROPERTY="GlobalSysSoln"          VALUE="IterativeStaticCond" />
        <I PROPERTY="Preconditioner"        VALUE="LowEnergyBlock"/>
        <I PROPERTY="SuccessiveRHS"        VALUE="8" />
        <I PROPERTY="IterativeSolverTolerance"    VALUE="1e-4"/>
    </V>
    <V VAR="p">
        <I PROPERTY="GlobalSysSoln"           VALUE="IterativeStaticCond" />
        <I PROPERTY="Preconditioner"         VALUE="LowEnergyBlock"/>
        <I PROPERTY="SuccessiveRHS"         VALUE="8" />
        <I PROPERTY="IterativeSolverTolerance"    VALUE="1e-4"/>
    </V>
</GLOBALSYSSOLNINFO>
```

or

```
<PARAMETERS>
    <P> SuccessiveRHS = 8 </P>
</PARAMETERS>
```

The typical value of `SuccessiveRHS` is $\leq$10.

The linear problem to be solved is

$$Ax = b, \tag{3.1}$$

here x and b are both column vectors. There are a sequence of already solved linear problems

$$Ax_n = b_n, n = 1, 2, ..., J. \tag{3.2}$$

Assume x_n are all linearly independent. In the successive right-hand method (see [15]), the best approximation to x is

$$\tilde{x} = \sum_{n=1}^{J} \alpha_n x_n \tag{3.3}$$

which is found by minimizing the object function

$$Q_1 = (A(\tilde{x} - x))^T A(\tilde{x} - x), \tag{3.4}$$

or

$$Q_2 = (\tilde{x} - x)^T A(\tilde{x} - x). \tag{3.5}$$

If Q_1 is used, the projection bases are $e_m = b_m, m = 1, 2, ..., J$. Using

$$(A(\tilde{x} - x))^T = \sum_{m=1}^{J} \alpha_m e_m^T - b^T, \tag{3.6}$$

there is

$$Q_1 = \sum_{m=1}^{J} \sum_{n=1}^{J} \alpha_m \alpha_n e_m^T b_n - 2 \sum_{m=1}^{J} \alpha_m e_m^T b + b^T b. \tag{3.7}$$

To minimize Q_1, there should be $\partial Q_1 / \partial \alpha_m = 0, m = 1, 2, ..., J$. The corresponding linear problem is

$$M(\alpha_1, \alpha_2, ..., \alpha_J)^T = (e_1^T b, e_2^T b, ..., e_J^T b)^T, \tag{3.8}$$

with symmetric positive definite coefficient matrix $M_{mn} = e_m^T b_n$. In this case, both the solutions x_m and the right-hand sides b_m need to be stored.

If Q_2 is used, A should be a symmetric positive definite matrix, as those encountered in the Poisson equation and the Helmholtz equation. Here, the projection bases are $\hat{e}_m = x_m, m = 1, 2, ..., J$. Using

$$(\tilde{x} - x)^T = \sum_{m=1}^{J} \alpha_n \hat{e}_m^T - x^T, \tag{3.9}$$

there is

$$Q_2 = \sum_{m=1}^{J} \sum_{n=1}^{J} \alpha_m \alpha_n \hat{e}_m^T b_n - 2 \sum_{m=1}^{J} \alpha_m \hat{e}_m^T b + x^T b. \tag{3.10}$$

To minimize Q_2, there should be $\partial Q_2 / \partial \alpha_m = 0, m = 1, 2, ..., J$. The corresponding linear problem is

$$M(\alpha_1, \alpha_2, ..., \alpha_J)^T = (\hat{e}_1^T b, \hat{e}_2^T b, ..., \hat{e}_J^T b)^T, \tag{3.11}$$

with symmetric positive definite coefficient matrix $M_{mn} = \hat{e}_m^T b_n$. In this case, only the solutions x_m need to be stored.

The formulations of Q_1 version and Q_2 version are the same, except the difference of projection bases. By default, Q_2 is used as the object function. If you want to use Q_1 instead, you can assign a negative value to `SuccessiveRHS`:

```
<I PROPERTY="SuccessiveRHS"          VALUE="-8" />
```

or

```
<P> SuccessiveRHS = -8 </P>
```

In the original paper of Fischer (1998) [15], the Gram-Schmidt orthogonal process is applied to the projection bases, this method is very stable and avoids the calculation of M^{-1}. However, when the memory space is full, this approach makes it hard to decide which basis should be overwritten. In our implementation, we just store the normalized right-hand sides or old solutions, i.e. $e_m^T b_m = 1$ or $\hat{e}_m^T b_m = 1$, and overwrite the oldest ones. The linear problem (3.8) or (3.11) is solved using a direct method.

To make sure M is positive definite, when a new basis e_{J+1} arrives, we test the following condition to decide whether or not to accept it,

$$r = (-y^T \tilde{M}^{-1}, 1) \begin{pmatrix} \tilde{M} & y \\ y^T & 1 \end{pmatrix} \begin{pmatrix} -\tilde{M}^{-1} y \\ 1 \end{pmatrix} = 1 - y^T \tilde{M}^{-1} y \geq \varepsilon > 0. \tag{3.12}$$

Assuming e_J is the oldest basis, there are $\tilde{M}_{mn} = M_{mn}, y_m = e_m^T b_{J+1}, (m, n = 1, 2, ..., J-1)$.

3.4.5.5 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.

3.4.6 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 boundary regions can also optionally contain a name which is then used in the multi-block VTK output to label the block descriptively rather than by ID, for example

```
<BOUNDARYREGIONS>
  <B ID="0" NAME="Wall"> C[2] </B>
  <B ID="1" NAME="Farfield"> 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.

3.4.6.1 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" />
```

3.4.6.2 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" />
```

3.4.6.3 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.
- It is possible specify periodic boundaries that are related by a rotation about a cartesian axis. In three-dimensions it is necessary to specify the rotational arguments to allow the orientation of each periodic face to be determined. This is not required in two-dimensions. An example of how two periodic boundaries are related by a rotation about the x-axis of $PI/6$ is shown below. The last number specifies an optional tolerance to which the rotation is considered as equivalent (default value is $1e-8$).

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

<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <P VAR="u" USERDEFINEDTYPE="Rotated:x:PI/6:1e-6"  VALUE="[1]" />
  </REGION>
  <REGION REF="1">
    <P VAR="u" USERDEFINEDTYPE="Rotated:x:-PI/6:1e-6"  VALUE="[0]" />
  </REGION>
</BOUNDARYCONDITIONS>
```

3.4.6.4 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))" />
```

3.4.6.5 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 expression (currently only implemented in 3D). For example, in the scenario where a spatial boundary condition is read from a file, but needs to be modulated by a time-dependent expression:

```
<REGION REF="1">
  <D VAR="u" 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.

3.4.7 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 5.6.20. 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 is 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>
```

3.4.7.1 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 filename, indicating the variable names in the target file that align with the desired function variable names. This gives a definition such as:

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

There are some caveats with this syntax:

- The same number of fields must be defined for both the `VAR` attribute and in the comma-separated list 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>
  ```

3.4.7.2 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 3.8 provides the list of acceptable mathematical functions and other related technical details.

3.4.8 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="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 spectral/hp element method.

```
<PARAMETERS>
  ...
  <P> HomModesZ     = 4       </P>
  <P> LZ            = 1.0     </P>
</PARAMETERS>
```

In case we want to create a Quasi-3D approach starting from 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="HOMOGENEOUS"          VALUE="2D"                     />
</SOLVERINFO>
<PARAMETERS>
  ...
  <P> HomModesY     = 10    </P>
  <P> LY            = 6.5   </P>
  <P> HomModesZ     = 6     </P>
  <P> LZ            = 2.0   </P>
</PARAMETERS>
```

By default the operations associated with the harmonic expansions are performed with the Matrix-Vector-Multiplication (MVM) defined inside the code. The Fast Fourier Transform (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="HOMOGENEOUS"          VALUE="2D"                    />
  <I PROPERTY="USEFFT"               VALUE="FFTW"                  />
</SOLVERINFO>
```

The number of homogeneous modes has to be even. The Quasi-3D approach 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.

3.5 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.

Some filters may perform a large number of operations, potentially taking up a significant percentage of the total simulation time. For this purpose, the parameter `IO_FiltersInfoSteps` is used to set the number of steps between successive total filter CPU time stats are printed. By default it is set to 10 times `IO_InfoSteps`. If the solver is run with the verbose `-v` flag, further information is printed, detailing the CPU time of each individual filter and percentage of time integration.

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.

3.5.1 Phase sampling

Note

This feature is currently only supported for filters derived from the FieldConvert filter: AverageFields, MovingAverage, ReynoldsStresses.

When analysing certain time-dependent problems, it might be of interest to activate a filter in a specific physical phase and with a certain period (for instance, to carry out phase averaging). The simulation time can be written as $t = m\mathcal{T} + n_\mathcal{T}\mathcal{T}$, where m is an integer representing the number of periods $\mathcal{T}$ elapsed, and $0 \leq n_\mathcal{T} \leq 1$ is the phase. This feature is not a filter in itself and it is activated by adding the parameters below to the filter of interest:

Option name	Required	Default	Description
`PhaseAverage`	✓		Feature activation
`PhaseAveragePeriod`	✓		Period $\mathcal{T}$
`PhaseAveragePhase`	✓		Phase $n_\mathcal{T}$.

For instance, to activate phase averaging with a period of $\mathcal{T} = 10$ at phase $n_\mathcal{T} = 0.5$:

```
<FILTER TYPE="FilterName">
    <PARAM NAME="Param1"> Value1 </PARAM>
    <PARAM NAME="Param2"> Value2 </PARAM>
    <PARAM NAME="PhaseAverage"> True </PARAM>
    <PARAM NAME="PhaseAveragePeriod"> 10 </PARAM>
    <PARAM NAME="PhaseAveragePhase"> 0.5 </PARAM>
</FILTER>
```

Since this feature monitors $n_\mathcal{T}$ every `SampleFrequency`, for best results it is recommended to set `SampleFrequency` $= 1$.
The maximum error in sampling phase is $n_{\mathcal{T},\text{tol}} = \frac{\Delta t}{2\mathcal{T}} \cdot$ `SampleFrequency`, which is displayed at the beginning of the simulation if the solver is run with the verbose `-v` option.
The number of periods elapsed is calculated based on simulation time. Caution is therefore recommended when modifying time information in the restart field, because if the new time does not correspond to the same phase, the feature will produce erroneous results.

3.5.2 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.
`PivotPoint`	✗	(0. 0. 0.)	Pivot Point around which the Moments are calculated. If Doesn't explicitly defined by user, the moments will be calculated around the point $\mathbf{X} = (0.\ 0.\ 0.)$

An example is given below:

```
<FILTER TYPE="AeroForces">
    <PARAM NAME="OutputFile">DragLift</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="Boundary"> B[1,2] </PARAM>
    <PARAM NAME="PivotPoint"> 1. 0.1 0. </PARAM>
</FILTER>
```

During the execution a file named `DragLift.fce` 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.

3.5.2.1 Extension for the SPM formulation

Note

This filter is only supported for the incompressible Navier-Stokes solver with the Smoothed Profile Method.

The lack of physical boundaries in the Smoothed Profile Method requires an alternative formulation to calculate the aerodynamic forces. Since the method imposes the boundary geometry by adding an impulse to the flow proportional to the difference between the flow velocity and the expected velocity of the immersed bodies, the forces in this filter are computed by integrating this difference where $\Phi \neq 0$ [27]:

$$\frac{\mathbf{F}^n}{\rho} = \frac{1}{\Delta t}\int_\Omega \Phi^{n+1}(\mathbf{u}^* - \mathbf{u^n}_p)\ \mathrm{d}\Omega$$

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 the output is written.
`StartTime`	✗	0	Forces before this instant are not written to the output.

For instance, a block like the following

```
<FILTER TYPE="AeroForcesSPM">
    <PARAM NAME="OutputFile"> Forces </PARAM>
    <PARAM NAME="OutputFrequency"> 10 </PARAM>
    <PARAM NAME="StartTime"> 50.0 </PARAM>
</FILTER>
```

will generate a file called `Forces.fce` with the values of the forces in the (X, Y, Z) directions, that must be scaled with the density of the fluid to get the real values. It is important to remark that the computed values are the sum of all the boundaries defined by Φ.

3.5.3 Benchmark

Note

This filter is only supported for the Cardiac Electrophysiology Solver.

Filter `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.

3.5.4 Cell history points

Note
This filter is only supported for the Cardiac Electrophysiology Solver.

Filter `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.

3.5.5 Checkpoint cell model

Note
This filter is only supported for the Cardiac Electrophysiology Solver.

Filter `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.

3.5.6 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`. Existing files are not overwritten, but renamed to e.g. `session_0.bak0.chk`. In case this file already exists, too, the `chk`-file is renamed to `session_0.bak*.chk` and so on.

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.
`OutputStartTime`	✗	0	Specifies the simulation time at which to start writing checkpoint files.

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>
    <PARAM NAME="OutputStartTime"> 50.0 </PARAM>
</FILTER>
```

It is also possible to specify the `OutputStartTime` option based on the simulation time step. For example, if checkpoint files should only be written after 500 time steps, one can write

```
<PARAM NAME="OutputStartTime"> TimeStep * 500 </PARAM>
```

3.5.7 Electrogram

Note

This filter is only supported for the Cardiac Electrophysiology Solver.

Filter `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.*

3.5.8 Offset Phase

Note

This filter is only supported for the Cardiac Electrophysiology Solver.

Filter `OffsetPhase` computes the instantaneous phase of the action potential using a time delay-embedding technique [20]:

$$\theta(t) = \text{atan2}[v(t-\tau) - \bar{v}, v(t) - \bar{v}].$$

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the phase are written.
`OutputFrequency`	✓	-	Number of timesteps after which output is written.
`MeanV`	✗	-57.2986	Specifies the time-averaged AP value ($\bar{v}$) in mV.

3.5.9 Hilbert Transform Phase based on FFTW

Note

This filter is only supported for the Cardiac Electrophysiology Solver. To use this filter, user must build Nektar++ with FFTW library.

Filter `HilbertFFTPhase` computes the instantaneous phase of the action potential by generating an analytic signal, $v(t) + i\mathcal{H}\{v\}(t)$.

$$\mathcal{H}\{v\}(t) = \frac{1}{\pi}\,\text{p.v.}\int_{-\infty}^{\infty}\frac{v(\tau)}{t-\tau}\,d\tau,$$

and

$$\theta(t) = \text{atan2}[\mathcal{H}\{v-\bar{v}\}(t), v(t)-\bar{v}].$$

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the phase are written.
`OutputFrequency`	✓	-	Number of timesteps after which an output is written. It is equal to the sampling frequency for Fourier Transform.
`WindowSize`	✓	-	Number of output steps taken in each FFT. *Note: output steps ≠ timesteps.*
`OverlapSize`	✓	-	Number of output steps overlapped between each segment.
`MeanV`	✗	Time-average	Specifies the time-averaged AP value in mV.
`LinearTransitionOverlap`	✗	`true`	`true`: Use linear transition in overlapping region(s). `false`: Use simple average in overlapping region(s). Input must be either `true` or `false`.

3.5.10 Error

This filter produces a file containing the time-evolution of the L_2 and L_{inf} errors. By default this file is called `session.err` where `session` is the session name.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the errors are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`ConsoleOutput`	✗	0	Also output error in the console when writing to file.

An example syntax is given below:

```
<FILTER TYPE="Error">
    <PARAM NAME="OutputFile">ErrorFile</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
```

```
    <PARAM NAME="ConsoleOutput">1</PARAM>
</FILTER>
```

3.5.11 Integral

This filter produces a file containing the time-evolution of the integral of the solver variables on user defined composites. By default this file is called `session.int` where `session` is the session name.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the integrals are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`OutputPrecision`	✗	7	Decimal precision with which the output is written.
`Composites`	✓	-	Composites on which to calculate the integral.

Multiple composites can be defined together by using ranges with each composite grouping producing a summed output for each variable in the `OutputFile`. For example `C[1,3,5-7]` would produce a single integral output for each variable whilst `C[1-3], C[4]` would produce two.

An example syntax is given below:

```
<FILTER TYPE="Integral">
    <PARAM NAME="OutputFile">IntegralFile</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="OutputPrecision"> 12 </PARAM>
    <PARAM NAME="Composites"> C[1], C[2-4,6]</PARAM>
</FILTER>
```

3.5.12 Integral

This filter produces a file containing the time-evolution of the integral of the solver variables on user defined composites. By default this file is called `session.int` where `session` is the session name.

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session`	Prefix of the output filename to which the integrals are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`OutputPrecision`	✗	7	Decimal precision with which the output is written.
`Composites`	✓	-	Composites on which to calculate the integral.

Multiple composites can be defined together by using ranges with each composite grouping producing a summed output for each variable in the `OutputFile`. For example `C[1,3,5-7]` would produce a single integral output for each variable whilst `C[1-3], C[4]` would produce two.

An example syntax is given below:

```
<FILTER TYPE="Integral">
    <PARAM NAME="OutputFile">IntegralFile</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="OutputPrecision"> 12 </PARAM>
    <PARAM NAME="Composites"> C[1], C[2-4,6]</PARAM>
</FILTER>
```

3.5.13 FieldConvert checkpoints

This filter applies a sequence of FieldConvert modules to the solution, writing an output file. An output is produced at the end of the simulation into `session_fc.fld`, or alternatively every M timesteps as defined by the user, into a sequence of files `session_*_fc.fld`, where `*` is replaced by a counter.

Module options are specified as a colon-separated list, following the same syntax as the `FieldConvert` command-line utility (see Section 5).

The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	`session.fld`	Output filename. If no extension is provided, it is assumed as .fld
`OutputFrequency`	✗	`NumSteps`	Number of timesteps after which output is written, M.
`Modules`	✗		FieldConvert modules to run, separated by a white space.
`OutputStartTime`	✗	0	Specifies the simulation time at which to start writing checkpoint files.

As an example, consider:

```
<FILTER TYPE="FieldConvert">
    <PARAM NAME="OutputFile">MyFile.vtu</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="OutputStartTime"> 50.0 </PARAM>
    <PARAM NAME="Modules"> vorticity isocontour:fieldid=0:fieldvalue=0.1 </PARAM>
</FILTER>
```

See the Checkpoint fields filter for more details about the `OutputStartTime` parameter.

This will create a sequence of files named `MyFile_*_fc.vtu` containing isocontours. The result will be output every 100 time steps.

3.5.14 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.
`OutputOneFile`	✗	`true`	If `OutputOneFile` is `true` or `yes`, only one file is generated for all points and the whole history. Otherwise, a separate file is created for each time snapshot.
`OutputPlane`	✗	-1	If the simulation is homogeneous, the plane on which to evaluate the history point. If set, all points will be reset to lie on this plane.
`WaveSpace`	✗	false	If the simulation is homogeneous and `WaveSpace` is `true` or `yes`, the Fourier coefficient on the nearest plane will be output.
`Points`	✗	-	A list of the history points.
`line`	✗	-	`npts,x0,y0,x1,y1` in 2D or `npts,x0,y0,z0,x1,y1,z1` in 3D. Set the history points as npts equispaced points between (x0,y0,z0) and (x1,y1,z1).
`plane`	✗	-	`npts1,npts2,x0,y0,z0,x1,y1,z1,x2,y2,z2,x3,y3,z3`. Set the history points as `npts1 * npts2` equispaced points on a plane. `npts1,npts2` is the number of equispaced points in each direction and (x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) define the plane of points specified in a clockwise or anticlockwise direction. Be careful about the size of history points file when using this option.
`box`	✗	-	`npts1,npts2,npts3,xmin,xmax,ymin,ymax,zmin,zmax`. Set the history points as `npts1 * npts2 * npts3` equispaced points in a box. `npts1,npts2,npts3` is the number of equispaced points in each direction and $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ define the limits of the box of points. Be careful about the size of history points file when using this option.

One of `Points`, `line`, `plane` or `box` must be specified.

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>
```

or

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

3.5.15 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>
```

3.5.16 Mean values

This filter calculates time-evolution of the averaged velocity components over the flow domain and can be used to track flow rates.

The following parameters are supported:

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

An example syntax is given below:

```
<FILTER TYPE="Mean">
    <PARAM NAME="OutputFile">mean</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
</FILTER>
```

3.5.17 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>
```

3.5.18 Moving body

Note

This filter is only supported for the Quasi-3D incompressible Navier-Stokes solver, in conjunction with the MovingBody forcing.

This filter `MovingBody` is encapsulated in the forcing module to evaluate the aerodynamic forces along the moving body surface. It is described in detail in section 11.3.8.1

3.5.19 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.

3.5.20 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>
```

3.5.21 Reynolds stresses

Note

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

This filter is an extended version of the time-average fields filter (see Section 3.5.22). 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="RestartFile">MyAverageRst.fld</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency"> 10 </PARAM>
</FILTER>
```

By default, this filter uses a simple average. Optionally, an exponential moving average can be used, in which case the output contains 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>
```

3.5.22 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.

This filter is derived from FieldConvert filter, and therefore support all parameters available in that case. The following additional parameter is supported:

Option name	Required	Default	Description
`SampleFrequency`	✗	1	Number of timesteps at which the average is calculated, N.
`RestartFile`	✗		Restart file used as initial average. If no extension is provided, it is assumed as .fld

As an example, consider:

```
<FILTER TYPE="AverageFields">
    <PARAM NAME="OutputFile">MyAverageField</PARAM>
```

```
    <PARAM NAME="RestartFile">MyRestartAvg.fld</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.

3.5.23 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`.

3.5.24 ThresholdMin value

Performs the same function as the `ThresholdMax` filter (see Section 3.5.23) but records the time at which the threshold variable drops below a prescribed value.

3.5.25 Maximun/minimun fields

This filter computes the local (pointwise) maximum or minimum field for each variable, including the addition variables for compressible flow solver, over the simulation.

The following parameters are supported:

Option name	Required	Default	Description
`MaxOrMin`	✓	max	Type of fields to compute. It can be either `max` or `min`.
`RestartFile`	✗	∓9999	Restart file used for initial comparison. If it is not provided, −9999 will be used for the `max` filter while 9999 will be used for the `min` filter. Output filename to which the threshold times are written.
`OutputFile`	✗	*session*_max.fld	Output filename to which the threshold times are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`SampleFrequency`	✗	1	Number of timesteps after which the fields are used to compute max/min.

An example is given below:

```
<FILTERS>
  <FILTER TYPE="MaxMinFields">
    <PARAM NAME="MaxOrMin">max</PARAM>
    <PARAM NAME="RestartFile">Baseflow.fld</PARAM>
    <PARAM NAME="OutputFile">DisturbedFields</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency">1</PARAM>
  </FILTER>
</FILTERS>
```

3.5.26 Python script

Note

This feature is only supported if Python support is enabled with the `NEKTAR_BUILD_PYTHON` flag enabled at configuration time.

The aim of this filter is to permit the execution of a Python script to implement more complex programmatic filters, but with the ease-of-use of the Python environment. Python filters mimic the action of C++ filters, and thus have three separate phases that should be implemented:

- an initialisation phase, which is called when the filter is constructed;
- an update phase, which is called typically at the end of a timestep;
- a finalise phase, which is called at the end of the simulation.

Given a Python file as input, the filter can operate in one of two ways:

- If only the Python file is supplied, then *Nektar++* will try to call three Python functions: `filter_initialise`, `filter_update` and `filter_finalise`. Each function is passed a list of the `ExpList` fields, and the value of the solver time t.
- If, in addition to the Python file, a filter name is supplied, then *Nektar++* assumes that your Python file creates a subclass of the `Filter` class which has been registered with the `Filter` factory.

The following parameters are supported:

Option name	Required	Default	Description
`PythonFile`	✓	-	Name of the Python file to load.
`FilterName`	✗	-	Optional name of filter if Python subclass is implemented.

An example is given below:

```
<FILTERS>
  <!-- Python filter with function approach -->
  <FILTER TYPE="Python">
    <PARAM NAME="PythonFile"> FilterPython_Function.py </PARAM>
  </FILTER>
  <!-- Python filter with class approach -->
  <FILTER TYPE="Python">
    <PARAM NAME="PythonFile"> FilterPython_Class.py </PARAM>
    <PARAM NAME="FilterName"> TestFilter </PARAM>
  </FILTER>
</FILTERS>
```

For the first example, a sample Python script may look something like:

```
def filter_initialise(fields, time):
    pass

def filter_update(fields, time):
    pass

def filter_finalise(fields, time):
    pass
```

For the second example, we might expect something like:

```
from NekPy.SolverUtils import Filter

class TestFilter(Filter):
    def __init__(self, session, eqsys, params):
        # Call super's constructor.
        super().__init__(session, eqsys)
        # Counter which is incremented at each timestep.
        self.num = 0

    def Initialise(self, fields, time):
        pass

    def Update(self, fields, time):
        self.num += 1

    def Finalise(self, fields, time):
        # Print for testing purposes
        print(self.num)

    def IsTimeDependent(self):
        return True

# Register filter with factory.
Filter.Register("TestFilter", TestFilter)
```

3.5.27 Body-fitted velocity fields

The `BodyFittedVelocity` filter projects the velocity fields in the Cartesian coordinate system into a body-fitted coordinate system. Meanwhile the user can chose to record the local (pointwise) maximum or minimum for the body-fitted velocity components, or simply output their original values. This filter is designed for both compressible flow solver and incompressible flow solver.

The following parameters are supported:

Option name	Required	Default	Description
`BodyFittedCoordinateFile`	✓	N/A	Body-fitted coordinate system generated by FieldConvert module. See 5.6.40 for reference.
`RestartFile`	✓	N/A	Restart file for the initial field for body-fitted velocity components u_{bfc} and v_{bfc} (and w_{bfc} for 3D cases). This restart file is necessary in the currtnt filter. The restart file can use the same output of the FieldConvert modul as well. See 5.6.40 for reference
`OriginalOrMaxOrMin`	✗	original	Type of fields to compute. It can be `original` or `max` or `min`.
`OutputFile`	✗	*session*_max.fld	Output filename to which the threshold times are written.
`OutputFrequency`	✗	1	Number of timesteps after which output is written.
`SampleFrequency`	✗	1	Number of timesteps after which the fields are used to compute original/max/min.

An example is given below:

```
<FILTERS>
  <FILTER TYPE="BodyFittedVelocity">
    <PARAM NAME="RestartFile">bfcFile.fld</PARAM>
    <PARAM NAME="BodyFittedCoordinateFile">bfcFile.fld</PARAM>
    <PARAM NAME="OriginalOrMaxOrMin">max</PARAM>
    <PARAM NAME="OutputFile">vel_bfc</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
    <PARAM NAME="SampleFrequency">1</PARAM>
  </FILTER>
</FILTERS>
```

3.5.28 Lagrangian points Tracking

The `FilterLagrangianPoints` filter tracks Lagrangian points in parallel. The following parameters are supported:

Option name	Required	Default	Description
`OutputFile`	✗	Session name	OutputFile_R%05d_T%010.6lf.plt is output file name for the Lagrangian points. R is followed by the thread number and T is followed by the instantaneous time.
`OutputFrequency`	✗	1	Output frequency of the Lagrangian points.
`Box_n`	✓	N/A	Box_n = Nx,Ny,Nz, xmin,xmax, ymin,ymax, zmin,zmax. Points defined by a box that initially owned by the thread n. At least one box needs to be assigned
`RootOutputL2Norm`	✗	0	Root thread outputs the L2 norm of the physics values over the points if RootOutputL2Norm is 1.
`FrameVelocity`	✗	0	The frame velocity which will be subtracted from the points' velocity.
`IntOrder`	✗	1	Time integration order for points tracking. The points will be set as stationary if IntOrder is 0.

An example is given below:

```
<FILTERS>
  <FILTER TYPE="LagrangianPoints">
    <PARAM NAME="OutputFile">Points</PARAM>
    <PARAM NAME="OutputFrequency">10</PARAM>
    <PARAM NAME="Box_0">0, 1,32,0.3,0.6,0.0,0.49</PARAM>
    <PARAM NAME="Box_1">32, 1,32,0.3,0.6,0.51,1.0</PARAM>
    <PARAM NAME="RootOutputL2Norm">1</PARAM>
  </FILTER>
</FILTERS>
```

3.6 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 one of the following.

3.6.1 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/>
    <BOUNDARYREGIONS> 1,4 <BOUNDARYREGIONS/>
</FORCE>
```

If a list of `BOUNDARYREGIONS` is specified, the distance to these regions is available as additional variable `r` in the definition of the `COEFF` function:

```
<FUNCTION NAME="AbsorptionCoefficient">
    <E VAR="p" EVARS="r" VALUE="-5000 * exp(-0.5 * (3*r / 0.4)^2)" />
    <E VAR="u" EVARS="r" VALUE="-5000 * exp(-0.5 * (3*r / 0.4)^2)" />
    <E VAR="v" EVARS="r" VALUE="-5000 * exp(-0.5 * (3*r / 0.4)^2)" />
</FUNCTION>
```

3.6.2 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>
```

3.6.3 Synthetic turbulence generator

This force type allows the user to apply synthetic turbulence generation in the flow field. The Synthetic Eddy Method is implemented. The approach developed here is based on a source term formulation. This formulation allows the user to apply synthetic turbulence generation in any specific area of the domain, not only in the boundary condition as most methodologies do. So that, after defining a synthetic eddy region (box of eddies),

the user can randomly release eddies inside this box which are going to be convected downstream and will produce turbulence depending on the flow conditions. Each eddy that leaves the synthetic eddy region is reintroduced in the inlet plane of the box, so this mechanism re-energise the system, roughly speaking.

Below it is shown how to define the Synthetic Eddy Method for a fully three-dimensional Navier-Stokes simulation. Note that this definition is under the `FORCING` tag. Firstly, in the `TYPE` entry, we define the force type as `IncNSSyntheticTurbulence` for the incompressible solver and `CFSSyntheticTurbulence` for the compressible solver. In the `BoxOfEddies` tag, under the `FORCE` tag, the center plane of the synthetic eddy region is defined. The coordinates of its center are given by `x0, y0, z0` and lengths of its sides are `lyref` and `lzref` in the y- and z-directions, respectively. Note that the length in the x-direction is defined in the characteristic length scale function (see below), so that `l00` defines the value of l_x. In the `Sigma` tag, we define the standard deviation (`sigma`) of the Gaussian function with zero mean, which is used to compute the stochastic signal. After that, the bulk velocity (`Ub`) of the flow must be provided in the `BulkVelocity` tag.

```
<FORCE TYPE="IncNSSyntheticTurbulence">
    <BoxOfEddies> x0 y0 z0 lyref lzref </BoxOfEddies>
    <Sigma> sigma </Sigma>
    <BulkVelocity> Ub </BulkVelocity>
    <ReynoldsStresses> [ReynoldsStresses FUNCTION NAME] </ReynoldsStresses>
    <CharLengthScales> [LenScales FUNCTION NAME] </CharLengthScales>
</FORCE>
```

In order to define the Reynolds stresses (`ReynoldsStresses` tag) and the characteristic length scales (`CharLengthScales` tag) of the eddies, the name of the functions which define them must be given. These functions must be placed under the `CONDITIONS` tag. Both functions are provided below. It is worthy mentioning that it is possible to define space-dependent functions for each Reynolds stress. In other words, the user can, for instance, provide the analytical solution of the Reynolds stresses close to the wall (boundary). This information is essential to calculate the velocity fluctuations.

```
<FUNCTION NAME="ReynoldsStresses">
    <E VAR="r00" VALUE="1e-3" />
    <E VAR="r10" VALUE="10*y+y^2+5*y^3" />
    <E VAR="r20" VALUE="0.0"  />
    <E VAR="r11" VALUE="1e-3" />
    <E VAR="r21" VALUE="0.0"  />
    <E VAR="r22" VALUE="1e-3" />
</FUNCTION>
```

Also, in the Synthetic Eddy Method implemented here, an isotropic or anisotropic turbulence can be described depending on the values provided in the characteristic length scale function. For an isotropic turbulence, all the values must be the same.

```
<FUNCTION NAME="LenScales">
    <E VAR="l00" VALUE="1.0"   />
    <E VAR="l10" VALUE="0.085" />
```

```
    <E VAR="120" VALUE="0.125" />
    <E VAR="101" VALUE="0.4"   />
    <E VAR="111" VALUE="0.085" />
    <E VAR="121" VALUE="0.125" />
    <E VAR="102" VALUE="0.4"   />
    <E VAR="112" VALUE="0.170" />
    <E VAR="122" VALUE="0.25"  />
</FUNCTION>
```

Note that the synthetic turbulence generator is only supported for fully three-dimensional simulations.

3.6.4 MovingReferenceFrame

This force type allows the solution of incompressilbe Navier-Stokes in moving frame of reference. The moving frame is attached the to body and can have translational, rotational or both motions. Although the Navier-Stokes equations are solved in a moving reference frame, our formulation is based on the absolute velocity and pressure (in inertial frame). However, note that these absolute velocities and any other vector quantities are expressed using the coordinate basis of the moving frame. Further, note that if you are using the FilterAeroForces, the force vector (F_x, F_y, F_z) is automatically converted and output in the inertial frame (ground reference frame).

To use this formulation the user need to specify the force type inside the `FORCING` tag as follwos:

```
<FORCE TYPE="MovingReferenceFrame">
    <FRAMEVELOCITY> [MRF FUNCTION NAME] </FRAMEVELOCITY>
    <PIVOTPOINT> x0,  y0,  z0  </PIVOTPOINT>
</FORCE>
```

Here we are required to provide one function for this force type which defines the linear velocity and the angular velocity of reference frame or both. In the case of rotating frame, i.e. when we are prescribing the angular velocity of reference frame, we can provide a coordinate of `PIVOTPOINT`, around which the frame is rotating. If no pivot point provided, the origin of coordinates in the moving reference frame will be used as the pivot point. Note that the frame velocities (both linear and angular velocities) must be defined in the inertial stationary frame of reference,i.e. ground reference frame (and expressed using the basis of inertial stationary frame), however, the Poivot point is in the moving reference frame.

Examples of linear and angular velocity funcitons together with their usage in the Forcing is shown below:

```
<CONDITIONS>

<FUNCTION NAME="VelMRF">
    <E VAR="u" VALUE="2*sin(PI*t)" />
    <E VAR="v" VALUE="0.1" />
```

```
    <E VAR="w" VALUE="0" />
    <E VAR="Omega_x" VALUE="0" />
    <E VAR="Omega_y" VALUE="0" />
    <E VAR="Omega_z" VALUE="0.3*cos(2*PI*t)" />
</FUNCTION>

</CONDITIONS>

<FORCING>

    <FORCE TYPE="MovingReferenceFrame">
        <FRAMEVELOCITY> VelMRF </FRAMEVELOCITY>
        <PIVOTPOINT> 0.2, 0.0, 0.0  </PIVOTPOINT>
    </FORCE>

</FORCING>
```

The moving frame function defines the velocity of the body frame observed in the inertial reference frame

$$\mathbf{u}_{frame} = \mathbf{u}_0 + \mathbf{\Omega} \times (\mathbf{x} - \mathbf{x}_0).$$

This means that these functions (such as the `VelMRF` in the above example) are defined and expressed in the stationary inertial frame (ground frame).

Here, $\mathbf{u}_0 =$ (u, v, w) is the translational velocity, $\mathbf{\Omega} =$ (Omega_x, Omega_y, Omega_z) is the angular velocity. $\mathbf{x}_0 = (0.2, 0.0, 0.0)$ is the rotation pivot and it is fixed in the body frame. Translational motion is allowed for all dimensions while rotational motion is currently restricted to z (omega_z) for 2D, 3DH1D and full 3D simulations.

Finally, note that when using `MovingReferenceFrame` force type, for any open part of the computational domain that the user specifies the velocity, such as inlet and free stream boundary conditions, the `USERDEFINEDTYPE="MovingFrameDomainVel"` tag can be used for velocity components. For example if boundary `ID=2` is the inlet with `Uinfx` and `Uinfy` the values of inlet velocities defined as parameters, the boundary condition for this boundary becomes:

```
<REGION REF="2">
   <D VAR="u" USERDEFINEDTYPE="MovingFrameDomainVel" VALUE="Uinfx" />
   <D VAR="v" USERDEFINEDTYPE="MovingFrameDomainVel" VALUE="Uinfy" />
   <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
</REGION>
```

for the wall boundary conditions on the surface of the body, we need to use `MovingFrameWall` tag as shown below:

```
<REGION REF="0">
    <D VAR="u" USERDEFINEDTYPE="MovingFrameWall" VALUE="Uinfx" />
    <D VAR="v" USERDEFINEDTYPE="MovingFrameWall" VALUE="Uinfy" />
    <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
</REGION>
```

The outlet and pressure boundary conditions are the same as before. In practical, It is not necessary to set the `MovingFrameDomainVel` or `MovingFrameWall` tag for all velocity components. In the circumstance when the analytical solution of a velocity component in the body frame is available, the value can still be prescribed as a normal Dirichlet-type boundary condition.

In addition to the prescribed motion, the frame velocity can also be determined from the fluid-body interaction. One example is as follows:

```
<CONDITIONS>
    <FUNCTION NAME="VelMRF">
        <E VAR="u" VALUE="0.0" />
        <E VAR="v" VALUE="0" />
        <E VAR="Omega_z" VALUE="cos(t)"/>
    </FUNCTION>

    <FUNCTION NAME="InitDisp">
        <E VAR="x" VALUE="0" />
        <E VAR="y" VALUE="0" />
        <E VAR="Theta_z" VALUE="0"/>
    </FUNCTION>

    <FUNCTION NAME="ExtForce">
        <E VAR="fx" VALUE="-1" />
        <E VAR="fy" VALUE="0" />
        <E VAR="Mz" VALUE="0"/>
    </FUNCTION>
</CONDITIONS>

<FORCING>
    <FORCE TYPE="MovingReferenceFrame">
        <FRAMEVELOCITY> VelMRF </FRAMEVELOCITY>
        <EXTERNALFORCE>  ExtForce </EXTERNALFORCE>
        <INITIALDISPLACEMENT>  InitDisp </INITIALDISPLACEMENT>
        <MASS>          1.0, 0., 0,  0,1,0,  0,0,1 </MASS>
        <MOTIONPRESCRIBED>  0, 1 ,1 </MOTIONPRESCRIBED>
        <PIVOTPOINT> 0. , 0,0 </PIVOTPOINT>
    </FORCE>
</FORCING>
```

The meanings of the parameters are:

Option name	Description	Default
`MOTIONPRESCRIBED`	A vector that denotes a certain degree of freedom is prescribed by the `FRAMEVELOCITY` (`1`) or determined from fluid force (`0`).	`1`
`EXTERNALFORCE`	External force exerted on the body, (fx, fy, fz, Mx, My, Mz).	`0`
`MASS`	Mass matrix of the body system. Its size is 3×3 in 2D and 4×4 in 3D.	`0`
`DAMPING`	Damper matrix of the body system. Its size is 3×3 in 2D and 4×4 in 3D.	`0`
`RIGIDITY`	Stiffness matrix of the body system. Its size is 3×3 in 2D and 4×4 in 3D.	`0`
`INITIALDISPLACEMENT`	A function to set the initial displacement of the body, (x, y, z, Theta_x, Theta_y, Theta_z).	`0`
`TRAVELINGWAVESPEED`	A traveling wave motion can be set to simulation problem such as a traveling wave propagating along an infinite geometry.	`0`
`OutputFile`	Output file name.	`SessionFileName.mrf`
`OutputFrequency`	Output frequency of the body's motion.	`1`

3.6.5 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>
```

3.6.6 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>
```

3.7 Coupling

Nektar++ Solvers can be run in parallel with third party applications and other Nektar++ solvers, where run-time data exchange is enabled by the coupling interface. The interface is configured in the `COUPLING` tag as

```
<COUPLING TYPE="[type]" NAME="[name]">
    <I PROPERTY="SendSteps"          VALUE="1" />
    <I PROPERTY="SendVariables"      VALUE="u0S,v0S" />
    <I PROPERTY="ReceiveSteps"       VALUE="1" />
    <I PROPERTY="ReceiveVariables"   VALUE="u0R,v0R" />
    ...
</COUPLING>
```

The coupling type may be any of the types described later in this section, while the name can be chosen arbitrarily. Inside each coupling block, the send and receive frequencies are defined by the `SendSteps` and `ReceiveSteps` parameters, respectively. Which variables are to be sent or received is specified by the `SendVariables` and `ReceiveVariables`. By default, the send and receive frequencies is set to zero, which disables the corresponding exchange in this coupling. An empty `SendVariables` or `ReceiveVariables` list has the same effect.

Option name	Required	Default	Description
`SendSteps`	✗	0	Frequency (in steps) at which fields are sent. Sending is disabled if set to zero.
`SendVariables`	✗	<empty>	Comma-separated list of sent variables. Sending is disabled if the list is empty.
`ReceiveSteps`	✗	0	Frequency (in steps) at which fields are received. Receiving is disabled if set to zero.
`ReceiveVariables`	✗	<empty>	Comma-separated list of received variables. Receiving is disabled if the list is empty.

3.7.1 File

This coupling type allows the user to exchange fields at run time by reading from and writing to files. Besides the basic parameters which define the exchanged variables and the exchange frequency, the file coupling type requires the `SendFileName` and `ReceiveFunction` parameters to be set. The Coupling name is not used for this type and can be ignored.

```
<COUPLING NAME="coupling1" TYPE="File">
    <I PROPERTY="SendSteps"          VALUE="1" />
```

```
    <I PROPERTY="SendVariables"     VALUE="u0S,v0S" />
    <I PROPERTY="SendFileName"      VALUE="Dummy0out_%14.8E.pts" />
    <I PROPERTY="ReceiveSteps"      VALUE="1" />
    <I PROPERTY="ReceiveVariables"  VALUE="u0R,v0R" />
    <I PROPERTY="ReceiveFunction"   VALUE="CouplingIn" />
</COUPLING>
```

`SendFileName` specifies a file name template to write the field data to. Currently, only `.pts` files are supported and the file is only created once fully written, avoiding race conditions between sender and receiver. Receiving is implemented by evaluating a session function specified in the `ReceiveFunction` parameter. The coupling waits for the file given in the receive function to appear.

Option name	Required	Default	Description
`SendFileName`	(✓)	-	File name where the sent fields should be written to. Required if sending is enabled. Time dependent file names are supported.
`ReceiveFunction`	(✓)	-	Function to evaluate to obtain the received fields.Required if receiving is enabled.

3.7.2 Cwipi

Note

The Cwipi coupling is only available when Nektar++ is compiled with OpenMPI and CWIPI

The Cwipi coupling uses CWIPI[1] to facilitate real time data exchange over MPI. See [25] for details. All data transfers are non-blocking to minimize the computational overhead. The interface must be enabled with the command line option `-cwipi` and a unique application name, e.g:

```
DummySolver --cwipi 'Dummy1' Dummy_3DCubeCwipi_1.xml
```

CWIPI uses the names of the current application and the coupling to identify two peers in cosimulation setups. The name of the remote application must be provided by the `RemoteName` parameter. Unlike the File-type coupling, a linear interpolation in time is applied to the received fields if non-unity values are set for `ReceiveSteps`.

```
<COUPLING NAME="coupling1" TYPE="Cwipi">
    <I PROPERTY="RemoteName"        VALUE="Dummy1" />
```

[1] http://sites.onera.fr/cwipi/

```
3     <I PROPERTY="SendSteps"          VALUE="1" />
4     <I PROPERTY="SendVariables"      VALUE="u0S,v0S" />
5     <I PROPERTY="SendMethod"         VALUE="Evaluate" />
6     <I PROPERTY="ReceiveSteps"       VALUE="1" />
7     <I PROPERTY="ReceiveVariables"   VALUE="u0R,v0R" />
8     <I PROPERTY="Oversample"         VALUE="5" />
9     <I PROPERTY="FilterWidth"        VALUE="10E-3" />
10    <I PROPERTY="NotLocMethod"       VALUE="Extrapolate" />
11 </COUPLING>
```

Additional options which define the coupling include `SendMethod`, the method used to retrieve the physical values at the locations requested by the remote application. Available options are `NearestNeighbour`, `Shepard` and the default `Evaluate`. The last option directly evaluates the expansions using a backward transform, giving superior accuracy at acceptable computational cost.

When using non-conforming domains, the current application might request values outside of the computational domain of the remote application. How to handle these not-located points is specified by the `NotLocMethod` parameter. When set to `keep`, the point value is not altered. With `Extrapolate`, the nearest neighbor value of the current application is used. Note that this can be very inefficient when using many MPI ranks.

Option name	Required	Default	Description
`RemoteName`	✓	-	Name of the remote application.
`SendMethod`	✗	`Evaluate`	Specifies how to evaluate fields before sending. Available options are `NearestNeighbour`, `Shepard` and `Evaluate`.
`Oversample`	✗	0	Receive fields at a higher (or lower) number of quadrature points before filtering to avoid aliasing.
`FilterWidth`	✗	0	Apply a spatial filter of a given filter width to the received fields. Disabled when set to zero.
`NotLocMethod`	✗	`keep`	Specifies how not located points in non-conformal domains are handled. Possible values are `keep` and `Extrapolate`.

3.8 Expressions

This section discusses particulars related to expressions appearing in Nektar++. 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.

Expressions appear as the content of `VALUE` attribute of

- parameter values;
- 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 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 expression. Currently one cannot directly link a boundary condition declaration with an expression uniquely specified somewhere else, e.g. in the `<FUNCTION>` subsection. However this duplication does not affect an overall computational performance.

3.8.1 Variables and coordinate systems

Declarations of 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 expressions depends on the problem dimension:

- For 1D problems, expressions must make use of variable `x` only;
- For 2D problems, expressions should make use of variables `x` and `y` only;
- For 3D problems, expressions may use any of variables `x`, `y` and `z`.

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 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.

3.8.1.1 Time dependence

Variable `t` represents time dependence within expressions. The boundary condition declarations need to add an additional property `USERDEFINEDTYPE="TimeDependent"` in order to flag time dependency to the library.

3.8.1.2 Syntax of 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 $\lfloor x \rfloor$
`fmax(x,y)`	maximum value (equivalent to `max`)
`fmin(x,y)`	minimum value (equivalent to `min`)
`fmod(x,y)`	floating point modulus operator
`log(x)`	logarithm base e, $\ln x = \log x$
`log10(x)`	logarithm base 10, $\log_{10} x$
`max(x,y)`	maximum value $\max(x, y)$
`min(x,y)`	minimum value $\min(x, y)$
`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)

3.8.1.3 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`

3.8.2 Performance considerations

Processing expressions is split into two stages:

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

Parsing of expressions with their partial evaluation take place at the time of setting the run up (reading an XML file). Each 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.

3.8.2.1 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.

3.8.2.2 Preparing 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.

3.8.2.3 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.

3.9 Movement

This section defines the movement of the mesh. Currently only static non-conformal interfaces are supported.

3.9.1 Non-conformal meshes

Non-conformal meshes are defined using `ZONES` and `INTERFACES`. Each zone is a domain as defined in the `GEOMETRY` section. For a mesh to be non-conformal it must consist of at least two zones with different domain tags. These two zones can then be split by an interface where every interface is defined by two composite entities, we use `LEFT` and `RIGHT` notation to distinguish between these. Each zone must contain either the left or the right interface edge. Zones can contain multiple edges across different interfaces but must not contain both edges for the same interface. These left and right interface edges have to be geometrically identical but topologically disconnected i.e. occupy the same space physically but consist of independent geometry objects.

Non-conformal interfaces are defined enclosed in the `NEKTAR` tag. An example showing two non-conformal interfaces on a single mesh is below:

```
<MOVEMENT>
    <ZONES>
        <FIXED ID="0" DOMAIN="D[0]" />
        <FIXED ID="1" DOMAIN="D[1]" />
        <FIXED ID="2" DOMAIN="D[2]" />
    </ZONES>
    <INTERFACES>
        <INTERFACE NAME="First">
            <LEFT  ID="0" BOUNDARY="C[0]" />
            <RIGHT ID="1" BOUNDARY="C[1]"  />
        </INTERFACE>
        <INTERFACE NAME="Second">
            <LEFT  ID="1" BOUNDARY="C[2]" />
            <RIGHT ID="2" BOUNDARY="C[3]"  />
        </INTERFACE>
    </INTERFACES>
</MOVEMENT>
```

Zones must have a type specified, at the moment only ‘FIXED’ interfaces are supported however in the future there are plans to implement rotating and sliding motion using the ALE method. It is important for the zone IDs to correspond with the relevant interface IDs present on the zone, that is if there is an interface with ID 0 there must also be a zone with ID 0 too. Zone IDs must be unique but interfaces can have the same ID, e.g. in the example above zone ID 1 has two interfaces attached to it. The inclusion of an `"INTERFACE NAME="..."` allows for specifying a name, which is used for the debug output when the verbose flag ‘-v’ is specified. This is for user reference to ensure the non-conformal interfaces are set up correctly, and shows zone/interface IDs, number of elements in each zone and interface, and connections between each zone/interface. An example debug output for the above XML is shown below:

```
Movement Info:
        Num zones: 3
        - 0 Fixed: 8  Quadrilaterals
        - 1 Fixed: 24 Quadrilaterals
        - 2 Fixed: 4  Quadrilaterals
        Num interfaces: 2
        - "First":  0 (4 Segments) <-> 1 (6 Segments)
        - "Second": 1 (6 Segments) <-> 2 (2 Segments)
```

3.9.2 Sliding Mesh

ALE method is implemented through `MOVEMENT` and `ALEHELPER` for sliding mesh with rotating and sliding motion. Zones type can be specified by 'ROTATE' and 'TRANSLATE' interface. Recently, the ALE method is implemented in `ADRSolver` and `CompressibleFlowSolver`, but only support explicit time integration shceme and `LDG` for diffusion.

An example for the rotationg movement is shown below:

```
<MOVEMENT>
<ZONES>
    <FIXED ID="0" DOMAIN="D[0]" />
    <ROTATE ID="1" DOMAIN="D[1]" ORIGIN="0,0.5,0" AXIS="0,0,1" ANGVEL="100"/>
</ZONES>
<INTERFACES>
    <INTERFACE NAME="Circle">
        <LEFT  ID="0" BOUNDARY="C[405]" />
        <RIGHT ID="1" BOUNDARY="C[406]"  />
    </INTERFACE>
</INTERFACES>
</MOVEMENT>
```

Here `ORIGIN` and `AXIS` are the origin and axis that the zone rotates about, respectively. `ANGVEL` is the the angular velocity of rotation with degree unit.

Another example for the translating movement is shown below:

```
<MOVEMENT>
<ZONES>
    <FIXED ID="0" DOMAIN="D[0]" />
    <TRANSLATE ID="1" DOMAIN="D[1]" VELOCITY="0,PI*cos(2*PI*t)" DISPLACEMENT="0,0.5*
    sin(2*PI*t)"/>
    <FIXED ID="2" DOMAIN="D[2]" />
</ZONES>
<INTERFACES>
    <INTERFACE NAME="Left interface">
        <LEFT  ID="0" BOUNDARY="C[405]" SKIPCHECK="TRUE"/>
        <RIGHT ID="1" BOUNDARY="C[406]" SKIPCHECK="TRUE"/>
    </INTERFACE>
    <INTERFACE NAME="Right interface">
        <LEFT  ID="1" BOUNDARY="C[407]" SKIPCHECK="TRUE"/>
        <RIGHT ID="2" BOUNDARY="C[408]" SKIPCHECK="TRUE"/>
```

```
15     </INTERFACE>
16 </INTERFACES>
17 </MOVEMENT>
```

Here `VELOCITY` and `DISPLACEMENT` are the translation velocity and displacement in x,y,z direction. `SKIPCHECK` is defined to ignore the missing coordinates found check for specific interface.

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 an early stage of development and as such is disabled by default. For the time being those not using the 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 January 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

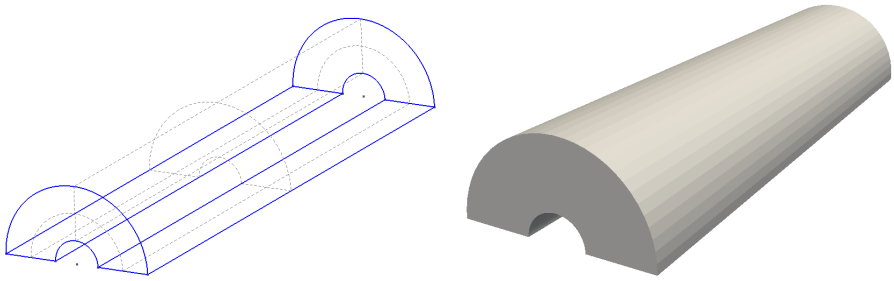


Figure 4.1 Geometry definition in Gmsh (left) and resulting high-order mesh visualised in ParaView (right).

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, we can use the `FieldConvert` command using:

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

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 in NekPy

The Python interface allows the user to instantiate input, output, and process modules by calling the static `Create` method of the `InputModule`, `ProcessModule`, and `OutputModule`, register configuration options, and process them. Consider the following example:

```
import sys
from NekPy.NekMesh import Mesh, ProcessModule, OutputModule

mesh = Mesh()
mesh.expDim = 3
mesh.spaceDim = 3
mesh.nummode = 5
mesh.verbose = True

# Load the CAD file
ProcessModule.Create("loadcad", mesh, \
                     filename="input.stp", verbose=True).Process()
# Load the octree
ProcessModule.Create("loadoctree", mesh, mindel=0.04,\
                      maxdel=0.2, eps=0.02).Process()
# Create a surface mesh
ProcessModule.Create("surfacemesh", mesh).Process()
# Output a 2D manifold mesh
mesh.expDim = 2
# Create a high-order surface
ProcessModule.Create("hosurface", mesh).Process()
# Dump out elemental Jacobians
ProcessModule.Create("jac", mesh, list=True).Process()
# Dump out the surface mesh.
OutputModule.Create("xml", mesh, test=True,\
                     outfile="output.xml").Process()
```

After importing the `Mesh`, `ProcessModule`, and `OutputModule` classes, first we create a `Mesh` object by calling the constructor. This object will be shared by the modules. Then we manipulate the `Mesh` object by creating different `ProcessModule`s, at the end we write out the result into a xml file using an `OutputModule`. The configuration options for a given module are passed to the static `Create` method of the `InputModule`, `ProcessModule`, and `OutputModule`. This creates the corresponding module and the modules can be processed immediately after instantiation. Note that the first parameter of the `Create` method has to be the key for a given module, the second is the previously created `Mesh` object. The remaining keyword arguments can specify additional parameters for a module.

The Python interface allows the user to create new modules by inheriting from one of the possible base classes (`InputModule`, `ProcessModule`, `OutputModule`).

The following is a simple example when we inherit from the `InputModule` and override the `Process` method:

```
import sys
from NekPy.LibUtilities import ShapeType
import NekPy.NekMesh as NekMesh
import numpy as np

# StructuredGrid creates a 2D structured grid of triangles.
class StructuredGrid(NekMesh.InputModule):
    def __init__(self, mesh):
        super(StructuredGrid, self).__init__(mesh)
        self.mesh.spaceDim = 2
        self.mesh.expDim = 2
        # Define some configuration options for this module.
        self.AddConfigOption("nx", "2", "Number of points in x direction")
        self.AddConfigOption("ny", "2", "Number of points in y direction")
        self.AddConfigOption("lx", "0", "Lower-left x-coordinate")
        self.AddConfigOption("rx", "0", "Upper-right x-coordinate")
        self.AddConfigOption("ly", "0", "Lower-left y-coordinate")
        self.AddConfigOption("ry", "0", "Upper-right y-coordinate")
        self.AddConfigOption("compid", "0", "Composite ID")

    def Process(self):
        # Get the input variables from our configuration options.
        coord_1x = self.GetFloatConfig("lx")
        coord_1y = self.GetFloatConfig("ly")
        coord_2x = self.GetFloatConfig("rx")
        coord_2y = self.GetFloatConfig("ry")
        nx = self.GetIntConfig("nx")
        ny = self.GetIntConfig("ny")
        compID = self.GetIntConfig("compid")
        x_points = np.linspace(coord_1x, coord_2x, nx)
        y_points = np.linspace(coord_1y, coord_2y, ny)

        nodes = []
        id_cnt = 0

```

```
36          for y in range(ny):
37              tmp = []
38              for x in range(nx):
39                  tmp.append(NekMesh.Node(id_cnt, x_points[x], y_points[y],
                        0.0))
40                  id_cnt += 1
41              nodes.append(tmp)
42          self._create_triangles(nodes, nx, ny, compID)
43          # Call the Module functions to create all of the edges, faces and
44          # composites.
45          self.ProcessVertices()
46          self.ProcessEdges()
47          self.ProcessFaces()
48          self.ProcessElements()
49          self.ProcessComposites()
50
51      def _create_triangles(self, nodes, nx, ny, compID):
52          ...
53
54  # Register our TestInput module with the factory.
55  NekMesh.Module.Register(
56      NekMesh.ModuleType.Input, "StructuredGrid", StructuredGrid)
57
58  if __name__ == '__main__':
59      # Create a 'pipeline' of the input and output modules.
60      mesh = NekMesh.Mesh()
61
62      # First, call our input module's create function from the NekMesh
            factory.
63      NekMesh.InputModule.Create(
64          "StructuredGrid", mesh,
65          nx = sys.argv[1], ny = sys.argv[2], lx = sys.argv[3],
66          ly = sys.argv[4], rx = sys.argv[5], ry = sys.argv[6],
67          compid = sys.argv[7], shape = sys.argv[8]).Process()
68
69      # Then ensure there's no negative Jacobians.
70      NekMesh.ProcessModule.Create("jac", mesh, list=True).Process()
71
72      # Finally, output the resulting file
73      NekMesh.OutputModule.Create(
74          "xml", mesh, test=True, outfile=sys.argv[9]).Process()
```

4.5 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

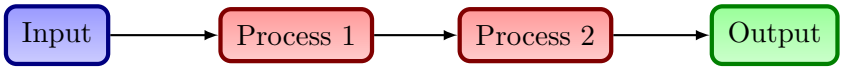


Figure 4.2 Illustrative pipeline of the `NekMesh` process.

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:`.

4.5.1 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
CGNS	`cgns`	✓	Based on the output currently provide by ANSA. Have support for high order elements supported. Requires NEKTAR_USE_CGNS option to be activated in cmake which then requires cgns library 4.4 to be available or compiled
Gmsh	`msh`	✓	Only reads nodes, elements and physical groups (which are mapped to composites). File format versions 2.x and 4.x currently supported.
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
```

The NekMesh input module also has an option to re-process all composites. By default

only composites are reprocessed. However when extracting a smaller mesh from a larger mesh definition by redefining the composite of volumetric elements you can force the edges and/or faces to be reprocessed removing the definition of any edges and/or faces not required in the smaller mesh. This option is called `processall` and has the syntax

```
NekMesh input.xml:xml:processall output.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. We note there is one exception to this convention in three-dimensional meshes where a face composite can contain both triangular and quadrilateral elements.

Typically a NekMesh call requires both an input and output module to be called, however, by specifying the output file name or file extension as `stdout` no output file will be created. This option is typically used for mesh statistics processing or inspecting composite values etc. An example call would be:

```
NekMesh input.xml stdout
```

or

```
NekMesh input.xml name.stdout
```

4.5.2 Output modules

The following output formats are supported:

Format	Extension	High-order	Notes
Gmsh	`msh`	✓	High-order hexes, quads, tetrahedra and triangles are supported up to arbitrary order. Prisms supported up to order 4, pyramids up to order 1.
Nektar++	`xml`	✓	Most functionality supported.
HDF5	`nekg`	✓	Most functionality supported.
VTK	`vtk`	✗	Outputs mesh only, supports line segments in 1D, triangles and quadrilaterals in 2D and tetrahedra, hexahedra, prisms and pyramids in 3D. The VTK legacy format and XML format, both compressed and uncompressed, are supported. Requires NEKTAR_USE_VTK option to be activated in cmake.

Note that for `Gmsh`, 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` and `vtk` 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`. For the uncompressed `xml` the user can execute:

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

If the user wishes to obtain the `vtk` output in the legacy format, the output module option `legacy` should be specified by executing:

```
NekMesh Mesh.xml output.vtk:vtk:legacy
```

Finally, both the Gmsh and Nektar++ output modules support an `order` parameter, which allows you to generate a mesh of a uniform polynomial order. This is used in the same manner as the above, so that the command

```
NekMesh Mesh.msh output.msh:msh:order=7
```

will generate an order 7 Gmsh mesh. In the rest of these subsections, we discuss the various processing modules available within `NekMesh`.

It is possible to use FieldConvert to extract a smaller region of a mesh from a larger mesh using the "-r xmin,xmax,ymin,ymax,zmin" range option, e.g.

```
FieldConvert -r xmin,xmax,ymin,ymax,zmin,zmax bigMesh.xml smallMesh.xml
```

However this will not provide a composite of he faces on the boundary if they were not part of the original boundary composites of the original `bigMesh.xml`. These can however be recovered by an output module option `chkbndcomp` in `NekMesh`. To do this one should specify

```
NekMesh smallMesh.xml newSmallMesh.xml:xml:chkbndcomp
```

this will then add a composite for any face that is on the boundary and not part of an existing boundary condition and put it into a composite with `id=9999`. This capability is currently only set up for 3D meshes

4.5.2.1 HDF5 format

NekMesh and all solvers within Nektar++ - along with subsequent FieldConvert modules - also support the HDF5 format. This allows for faster loading of geometries and meshes within each solver - and is a significant improvement over the XML format. HDF5 is recommended input format for any larger cases.

Converting from XML to HDF5 is a simple task that only requires the one NekMesh command:

```
NekMesh XMLMesh.xml HDF5Mesh.nekg
```

This will create two files `HDF5Mesh.xml` and `HDF5Mesh.nekg` which are both needed in the same directory to run the simulation. An additional flag in the session file is required, ensuring it is placed before the expansion list being:

```
1    <GEOMETRY DIM="3" SPACE="3" HDF5FILE="HDF5Mesh.nekg" />
```

HDF5 also has the additional advantage of ensuring the mesh and session file are split - which allows for easy ammending of the session file - whilst allowing for use of FieldCovnert modules that require only 1 XML input file - rather than having to concatenate the session and mesh XML files. Solvers and any FieldConvert modules can be run by referencing only the session file after the `GEOMETRY` tag is included.

4.5.3 Extract surfaces from a mesh

Often one wants to visualise surfaces of a 3D mesh, or extract the values of variables on the surface and visualise them. To support this, `NekMesh` can extract two-dimensional surfaces which can be visualised using `FieldConvert` in order to extract the value of a 3D field on a given surface.

As an example, we can extract composite surfaces 2 and 3-5 from a mesh using the `extract` module:

```
NekMesh -m extract:surf=2,3-5 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=2,3-5:detectbnd Mesh.xml output.xml
```

which will produce new composites for the extracted boundary.

4.5.4 Negative Jacobian detection and histogram visualisation

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
```

To extract the elements whose Jacobian is under a specific value for the purposes of visualisation within the domain, use the `extract` boolean parameter. The `value` should be a number below 1.0 and is 0.0 by default. This means if `extract` is set by user with no value, the output mesh file will only have invalid mesh elements:

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

To show the histogram of Jacobian on the screen, use the `histo` parameter.

```
NekMesh -m jac:histo=value1,value2,value3:quality:detail:histofile=filename
     Mesh.xml output.xml
```

The `histo` takes 3 input values: The maximum value shown on the histogram, the number of positive bins on the histogram and the number of negative bins. Each value is separated with a comma. By default the values are: 1.0,10,1. This means by default the histogram will have 10 positive bins between (0.0, 1.0) with an interval 0.1, and will have 1 negative bins which shows all the invalid elememnts. An example can be found below in figure 4.3 (a).

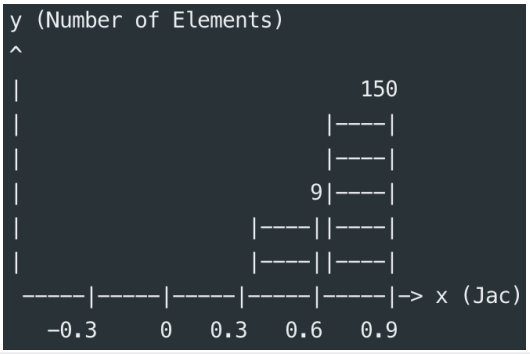
This command `histo` also generates a text file called "jacs.txt" which contains the information of each element in the histogram. The name of the file can be changed by using the command `histofile`. The information in this text file are: Element ID, Jacobian value, element type, if this element is a boundary element, the boundary edge/face ID, vertex coordinates and neighbor element ID will also be included. Note that only elements whose Jacobian is smaller than "value1" will be counted, and they will

be outputted in the histogram and text file. If command `detail` is added, the composite ID and name of each element will also be outputted into the text file. An example is shown in figure 4.3 (c).

The command `quality` is an option which show the percentage of each bin of the histogram. The results of `quality` always match the histogram. An example is shown in figure 4.3 (b).

An example with the following command is shown in figure 4.3. The results of `quality` and text file always correspond to the histogram.

```
NekMesh -m jac:histo=0.9,3,2:quality:detail Mesh.xml output.xml
```




```
Elements                 <  -0.3) =      0 (     0%)
Elements between (  -0.3,    -0) =      0 (     0%)
Elements between (     0,   0.3) =      0 (     0%)
Elements between (   0.3,   0.6) =      9 (  5.66%)
Elements between (   0.6,   0.9) =    150 ( 94.34%)
```

Element ID	Jac	type	Boundary face ID	Edge ID	Coordinates X	Coordinates Y	Coordinates Z	Composite ID	Composite Name	NeighborEl ID
26	0.876166	Tetrahedron								
102	0.791811	Tetrahedron	401	577	-0.158496	-0.162625	0.104556			
					-0.0484202	-0.220724	0.106942			
				578	-0.158496	-0.162625	0.104556			
					-0.0484202	-0.220724	0.106942			
				579	-0.158496	-0.162625	0.104556			
					-0.0484202	-0.220724	0.106942	8	NONE	2884 2014 1869
128	0.781195	Tetrahedron								
137	0.658822	Tetrahedron	535	760	-0.0204987	0.249158	0			
					-0.0728163	0.228541	0.0704746			
				761	-0.0204987	0.249158	0			
					-0.0728163	0.228541	0.0704746			
				762	-0.0204987	0.249158	0			
					-0.0728163	0.228541	0.0704746	1	NONE	1010 2209 2963
141	0.781368	Tetrahedron								
142	0.832657	Tetrahedron								
145	0.730934	Tetrahedron	567	806	0.25	0	0			
					0.215525	-0.126666	0			
				807	0.25	0	0			
					0.215525	-0.126666	0			
				808	0.25	0	0			
					0.215525	-0.126666	0			
			568	806	0.25	0	0			
					0.215525	-0.126666	0			
				810	0.25	0	0			
					0.215525	-0.126666	0			
				809	0.25	0	0			
					0.215525	-0.126666	0	2	NONE	2820 3063

Figure 4.3 (a) Results of option "histo": histogram shown on the screen, (b) Results of option "quality", (c) Part of the text file "jacs.txt"

To turn off curvature associated with negative jacobians one can try to use the `linearise` module:

```
NekMesh -m linerise:invalid  Mesh.xml output.xml
```

This option will remove all high order curvature on all element types with singular jacobians.

4.5.5 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:

4.5.6 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.

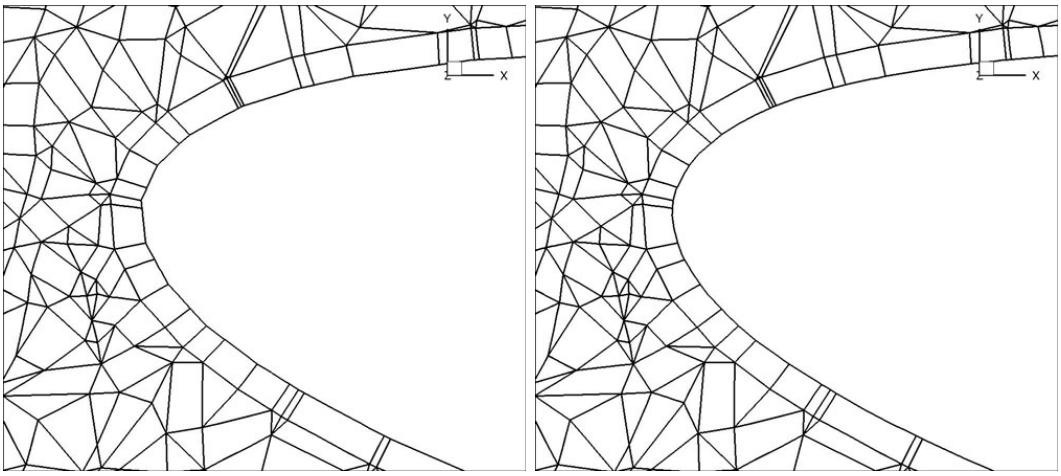


Figure 4.4 (a) Leading edge without spherigons, (b) Leading edge with spherigons

To facilitate this alignment, `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.

When using periodic boundary conditions that are rotationally aligned the following rotational options should be applied:

```
NekMesh -m peralign:surf1=11:surf2=12:dir=x:rot=PI/6 \
             Mesh.xml Mesh_aligned.xml
```

where `rot` specifies the rotation angle in radians from surf1 to surf2 about the axis specified by `dir` (i.e. the "x" axis in this example).

The rotation/translation is assumed to be exact within a relative tolerance. An optional factor, which is used to scale the tolerance, `tolfact` can also be specified. The default

tolerance factor is 4, and it needs to be $tolfact \geq 1$. For example:

```
NekMesh -m peralign:surf1=11:surf2=12:dir=x:rot=PI/6:tolfact=100 \
             Mesh.xml Mesh_aligned.xml
```

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.

4.5.7 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 [34] and [35]. 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 4.5.

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
```

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.

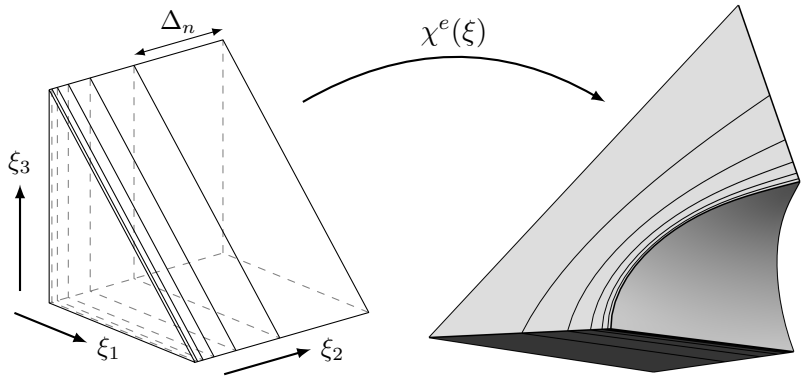


Figure 4.5 Splitting Ω_{st} and applying the mapping χ^e to obtain a high-order layer of prisms from the macro-element.

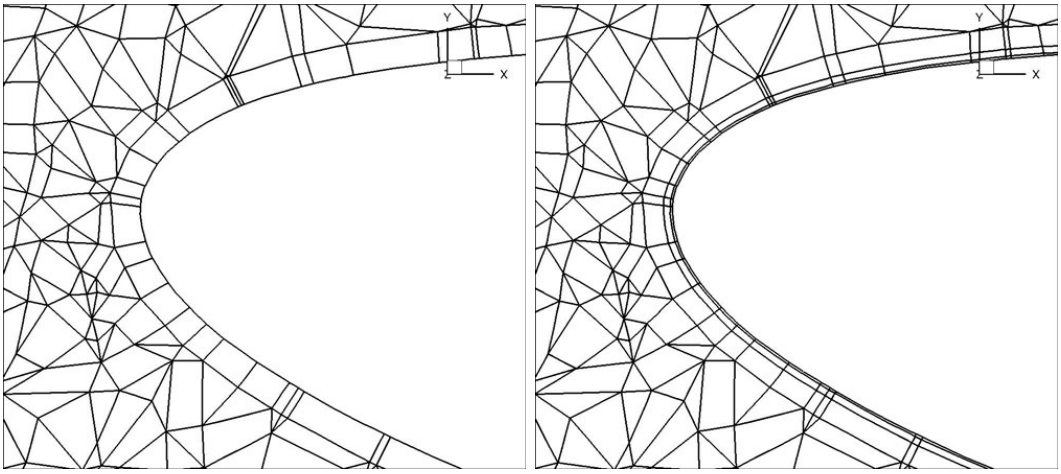


Figure 4.6 (a) LE with Spherigons but only one prism layer for resolving the boundary layer, (b) LE with Spherigons with 3 growing layers of prisms for better resolving the boundary layer.

4.5.8 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.

4.5.9 Linearisation

The ability to remove all the high-order information in a mesh can be useful at times when mesh generation is tricky or impossible in the presence of curvature

To do this in NekMesh use the command:

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

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

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

attempts to remove curvature from elements only where necessary. This is a simple algorithm that removes curvature from invalid elements and repeats until all elements are valid. Either `all` or `invalid` must be specified.

- `all`: remove curvature from all elements.

- `invalid`: remove curvature from invalid elements.
- `prismonly`: consider only prisms when removing curvature. This is useful in the presence of a prismatic boundary layer.

4.5.10 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.

4.5.11 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
```

4.5.12 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.

4.5.13 Link Checking

It is quite possible that a mesh contains some sort of hanging entity or element connectivity error. The check link module is a fast check that, a) elements are correctly connected and b) the boundary entities (composites) match the interior domain:

```
NekMesh -m linkcheck mesh.xml mesh2.xml
```

This module should be added to the module chain if the user suspects there may be a mesh issue. The module will print a warning if there is a connectivity error.

4.5.14 2D mesh extrusion

This module allows a 2D mesh, quads, triangles or both, to be extruded in the z direction to make a simple 3D mesh made of prisms and hexahedra. It is also capable of extruding the high-order curvature within the 2D mesh. The module requires two parameters:

```
NekMesh -m extrude:layers=n:length=l 2D.xml 3D.xml
```

length which determines how long the z extrusion will be and layers, the number of elements in the z direction.

4.5.15 2D mesh revolution

This module allows a 2D mesh, quads, triangles or both, to be revolved around the y axis to make a simple 3D mesh made of prisms and hexahedra. It is also capable of revolving the high-order curvature within the 2D mesh. The module requires two parameters:

```
NekMesh -m revolve:layers=n:angle=a 2D.xml 3D.xml
```

angle which determines the angle of revolution (θ) around the y axis and layers, the number of elements in the θ direction.

The angle can also be set to "full" for a full revolution of 2π, although this is the default setting:

```
NekMesh -m revolve:layers=n:angle=full 2D.xml 3D.xml
```

The angle must be between 0 and 2π. The 2D mesh must not cross or touch the y axis i.e. $x > 0$ for all points. By default the mesh is revolved through the full 2π.

4.5.16 Variational Optimisation

This module can correct invalid and improve the quality of elements in high-order meshes by applying curvilinear deformation to the interiors of domains. It achieves this by

solving a solid mechanics system which, using variational calculus has been cast is a non-linear energy optimsation problem. It is basis of the work in [48].

It works by considering the boundary (curved) mesh entities to be fixed and moving the interior nodes to a lower energy configuration. This new configuration in most scenarios is a higher quality mesh. The energy is evaluated depending on which functional is chosen. We find hyperleastic to be the most reliable but it can also model the mesh and a linearelastic solid as well as functionals based on the Winslow equation and the distortion method proposed by Roca et al. [16].

There are a large number of options which can be viewed using the help function but the basic usage is:

```
NekMesh -m varopti:type inital.xml optimised.xml
```

where type can be `hyperelastic`, `linearelastic`, `winslow` or `roca`.

4.5.17 *r*-adaptation: interpolation file

This module can deform an existing mesh by using the variational optimiser presented above. A file must be provided that contains a list of points and a scaling value for each of them. This scaling factor is then used to target an element size based on the initial size of the element. Scaling values are interpolated throughout the domain based on the interpolation method of the main library. The file should look like

```
0 0 0 2.0
0 1 0 2.0
1 0 0 0.5
1 1 0 0.5
```

where the first three columns are `x`, `y`, `z` and the last column is the scaling factor.

The call is identical to the variational optimisation module above:

```
NekMesh -m varopti:type:scalingfile=file.txt:subiter=x inital.xml adapted.
    xml
```

where `subiter` is an additional parameter to the variational optimiser that defines the frequency at which individual elements update their target scaling based on their latest location in the domain. `subiter` should be a scalar and is the number of steps between updates. It is often recommended to run *r*-adaptation on a linear mesh for stability and performance reasons. Note also that the mesh must have CAD information in order for nodes to slide on curves and surfaces.

4.5.18 *r*-adaptation: CAD curves

Another option for *r*-adaptation can be defined where instead of providing an interpolation file, the user can provide a list of CAD curves towards which the scaling takes place. With this option, the *r*-adaption uses a fixed scaling factor for all elements which are on or within a specified distance from the curve.

The call is similar to the *r*-adaption with interpolation file, with the difference of replacing the file with a list of CAD curve IDs, the scaling factor and an optional distance from the curve:

```
NekMesh -m varopti:type:radaptcurves=CurveIDs:radaptscale=scale:radaptrad=
    rad:subiter=x inital.xml adapted.xml
```

where `radaptcurves`, `radaptscale` and `radaptscale` are additional containing the list of CAD curve IDs (integers separated by commas, or with a dash for ranges, i.e. 1-3,6 would select curves with ID 1, 2, 3, and 6), the scaling factor as a float and the distance from the CAD curves as a float, respectively. As with the *r*-adaption with interpolation file, `subiter` is to define the frequency at which individual elements update their target scaling based on their current proximity to the selected CAD curves.

4.5.19 Mesh projection

This module can take any linear mesh, providing that it is a close representation of the CAD and project the boundary of the mesh onto the CAD. This will curve the surface of the mesh. The method has a number of failsafes ensuring that even bad CAD or poor linear meshes should be able to be curved to some degree. If the method encounters an issue, such as the linear mesh being a large distance from the CAD, it will simply leave that element straight sided. A well made CAD model and accurate linear mesh should be curved with little issue.

The module needs to be informed of the CAD file to project the mesh to and the order at which to curve the surface:

```
NekMesh -m projectcad:file=cadfile.step:order=x inital.xml optimised.xml
```

4.5.20 Mesh combine

This module can combine two mesh files into one output file. This will read all information from two mesh files and generate the combine meish file with all these information. The process will duplicate composite IDs from mesh 1 detected in mesh 2 and will remap into the output file.

```
NekMesh -m combine:file=mesh1.xml mesh2.xml combine.xml
```

4.6 Mesh generation

In addition to the functionality described previously, `NekMesh` is capable of generating high-order meshes directly from a CAD definition. By default this functionality is not activated, a user wishing to utilise the mesh generation capability of `NekMesh` must compile *Nektar++* with the `NEKTAR_USE_MESHGEN` option on. As well as compiling the relevant routines into `NekMesh` it will also download a number of other packages which are required.

The most critical dependancy of the mesh generation routines is `OpenCascade` which powers the CAD engine. `NekMesh` is capable of finding and using existing installations of `OpenCascade 6.8` or `OCE 0.17`. If either are not present on the installation machine `NekMesh` will install `OCE 0.17` from source. This is a very big installation and will take some time so it is advised that the user ensures `OpenCascade` is availble on the machine.

As with all tasks within `NekMesh` the mesh generation capability exists as its own separate module which is of type Input. Due to the vast amount of code associated with the generation of high-order meshes and the comparatively small nature of modules in the `NekMesh` program a new library has been created for *Nektar++* called *NekMeshUtils*, which contains all the core routines and classes for the `NekMesh` mesh format as well as a series of classes for the generation of meshes. This library also contains the CAD API for *Nektar++* which is used to generate the meshes.

4.6.1 Methodology

This section outlines the approach taken by `NekMesh` to generate high-order meshes. To simplify the sometimes very complicated high-order mesh generation processes in other programs, `NekMesh` executes all the stages required to produce a high-order mesh in one single pipeline which once started requires no interaction from the user. In broad terms these stages are:

- Specification of the element sizes in the mesh,
- Coarse linear mesh generation of the domain,
- Generation of optimised high-order surface on the geometric boundary,

and are outlined in more detail in the following sections.

4.6.1.1 CAD Interaction

At the core of all the ideas in the `NekMesh` generator is that the final mesh is a high quality representation of the underlying geometry. As such all of the entities in the mesh must know where they are located with respect to the CAD and the system to be able to query any geometric information at any point in the domain easily and with accuracy. To handle this `NekMesh` has been interfaced with the third-party suite of CAD

libraries called OpenCascade. In its normal state OpenCascade is a very large collection of libraries with tens of thousands of functions which are simply not needed for our purposes, because of this its installation is a very arduous and long process. Combine this with the fact that there are dozens of versions and types of OpenCascade, such as OpenCascade Community Edition, it is simply impossible for `NekMesh` to use already existing OpenCascade installations on a given machine. To solve these issues, when installing *Nektar++* with the mesh generator it will download pre-compiled binaries for the relevant OS and link against those, any previously installed versions of OpenCascade will not be searched for and therefore ignored. To reduce the massively complex libraries in OpenCascade down to a manageable set of functions to be used in `NekMesh` a set of interface classes have been created which act as buffer between it and *Nektar++*. These CAD classes mean that development of mesh generation routines is significantly easier and in the future *Nektar++* developers will be able to utilise CAD information in all aspects of the framework without having to learn OpenCascade. Another advantage with this approach is that adding support for other CAD engines, as well as OpenCascade, in the future should be relativity simple and will not require the rewriting of any of the `NekMesh` code.

4.6.1.2 Automatic specification of the mesh

One of the key challenges of generating a high-order mesh is the creation of a suitable coarse linear mesh. It is quite difficult for a user to define a full set spacings over a whole domain which will produce a good quality especially when aiming for coarseness. This is tackled in `NekMesh` with a system for automatically defining a set of smooth and coarse mesh spacings throughout the whole domain. This is achieved using an octree description of the domain. The domain is recursively subdivided into octants which each describe a small portion of the domain. The level to which the domain subdivides is based on the curvature of the geometric boundary. Higher curvature regions will subdivide to a finer level allowing for increased control on the mesh specification and smoothness. The geometric curvature is then related to a mesh sizing parameter and propagated throughout the domain ensuring a smooth mesh. For those unfamiliar with octrees, it is best to think of it as a non-conforming hexahedral mesh

4.6.1.3 Linear Mesh Generation

The first challenge mentioned in the previous section is addressed with the `NekMesh` approach to linear mesh generation. Primarily because of the difficulties in interfacing existing linear mesh generators for high-order applications the decision was made to include a bespoke linear mesh generator within the program. Compared with the mesh generators included in commercial packages this linear mesh generator takes the quite unconventional and more historic approach in building the mesh in a bottom up fashion from 0D to 3D. Using this approach means it is possible to guarantee a level of boundary conformity which direct to 3D approaches cannot at the desired level of coarseness. In this approach, first mesh nodes are placed on the vertices of the CAD model (0D), then the curves in the CAD are meshed in 1D using the vertex nodes as boundaries, then the

surfaces are meshed in their 2D parameter plane using the curve meshes as boundaries and finally the 3D volume is meshed using the surface mesh as the boundary to complete the linear mesh. In `NekMesh`, to achieve greater robustness, the 2D mesh generation library Triangle is used and the TetGen library for the 3D. Both of which are highly developed Delaunay based mesh generators. As with all additional libraies in *Nektar++* these are automatically downloaded and installed if needed.

4.6.1.4 High-order Surface Generation

Addition of the high-order nodes to and the curving of the mesh is very open problem, no high-order mesh generator has solved this and while the methods used in `NekMesh` are not 100% full-proof, the system currently in place can create good quality high-order curved meshes with a reasonable robustness. This area will receive the greatest level of development in the future. The most critical part of defining the high-order mesh is the addition of high-order nodes on the geometric surface. The mesh generator must achieve the greatest level of geometric accuracy as it can otherwise it will greatly affect the final flow solutions. If the linear surface triangulation is taken to be fixed during this process, the problem can be addressed in a element by element fashion. If the high-order nodes are placed by simply using an affine mapping to the CAD surface and back the resulting high-order triangle will inherit the same distortions as the CAD surface. To solve this `NekMesh` uses a system node location optimisation in the parameter plane of the CAD surface to ensure the high-order triangles have as little distortion as possible while remaining exactly on the geometric surface. To do this the system models the high-order edges and triangles as a network of springs with an associated spring energy which is minimised using a multidimensional Newton type optimisation procedure with a Gauss-Seidel matrix solver.

4.6.1.5 Mesh Correction

Due to the fact that, for the time being, no consideration is given to the curving of mesh interior entities explicitly in the mesh generation process, the curving the geometric surface can produce meshes with invalid elements, especially in the case of Euler type (Tetrahedra only) meshes. Three strategies exist within *Nektar++* to correct these elements. Firstly removing the curvature, by removing the curvature of invalid elements they become valid. However this has the massive downside of compromising the geometric accuracy of the mesh but is quick and effective, this can be enacted using the command:

```
NekMesh -m linearise:invalid invalidMesh.xml validMesh.xml
```

An alternative to this is to use the linear elastic solver within *Nektar++* to deform the mesh interior entities. Its use is very computationally expensive, as with all PDE solvers, and is also not particularly robust. It can be used with the set of commands outlined in the FieldConvert deform and displacement modules and the section on the Linear Elastic Solver.

The final and possibly most useful approach is to use the Variational Optimsation module to curve the interior of the domain. This is explained in 4.5.16.

4.6.2 Mesh generation manual

The mesh generation is executed with the command:

```
NekMesh session.mcf mesh.xml
```

where session.mcf is a mesh configuration file which contains all the options and parameters needed for mesh generation. Below is an example of a simple example which generates a 2D NACA wing.

```
<NEKTAR>
    <MESHING>

        <INFORMATION>
            <I PROPERTY="CADFile"   VALUE="6412"  />
            <I PROPERTY="MeshType"  VALUE="2D"    />
        </INFORMATION>

        <PARAMETERS>
            <P PARAM="MinDelta"     VALUE="0.01"  />
            <P PARAM="MaxDelta"     VALUE="1.0"   />
            <P PARAM="EPS"          VALUE="0.1"   />
            <P PARAM="Order"        VALUE="4"     />

            <!-- 2D Domain !-->
            <P PARAM="Xmin"         VALUE="-1.0"  />
            <P PARAM="Ymin"         VALUE="-2.0"  />
            <P PARAM="Xmax"         VALUE="3.0"   />
            <P PARAM="Ymax"         VALUE="2.0"   />
            <P PARAM="AOA"          VALUE="15.0"  />
        </PARAMETERS>

    </MESHING>
</NEKTAR>
```

In all cases the mesh generator needs two pieces of information and four parameters. It firstly needs to know the CAD file with which to work. In the example above this is listed as a 4 digit number, this is because the mesh generator is equiped with a NACA wing generator. In all other cases this parameter would be the name of a CAD file (in either STEP or GEO format). Secondly, what type of mesh to make, the options are `EULER` and `BndLayer` for 3D meshes and `2D` and `2DBndLayer` for 2D meshes. In the case of `EULER` the mesh will be made with only tetrahedra. For `BndLayer` the mesh generator will attempt to insert a single macro prism layer onto the geometry surface. This option requires additional parameters. This is similar for the 2D scenarios. The automatic mesh specification system requires three parameters to build the specification of a smooth, curvature refined mesh. Firstly `MinDelta` which is the size of the smallest element to be found in the final mesh. Secondly `MaxDelta` which is the maximum size

of an element in the mesh and lastly `EPS` which is a sensitivity to curvature parameter with a range $1 \geq \varepsilon > 0$ which heuristically controls the size of the elements for a given degree of curvature on the geometric surface. `Order` is the polynomial order of the mesh to be generated. When generating a boundary layer mesh a few additional parameters must be given. An example is shown.

```
<NEKTAR>
    <MESHING>

        <INFORMATION>
            <I PROPERTY="CADFile"            VALUE="6412"        />
            <I PROPERTY="MeshType"           VALUE="2DBndLayer"  />
        </INFORMATION>

        <PARAMETERS>
            <P PARAM="MinDelta"              VALUE="0.01"        />
            <P PARAM="MaxDelta"              VALUE="1.0"         />
            <P PARAM="EPS"                   VALUE="0.1"         />
            <P PARAM="Order"                 VALUE="4"           />

            <!-- Boundary layer !-->
            <P PARAM="BndLayerSurfaces"      VALUE="5,6"         />
            <P PARAM="BndLayerThickness"     VALUE="0.03"        />
            <P PARAM="BndLayerLayers"        VALUE="4"           />
            <P PARAM="BndLayerProgression"   VALUE="2.0"         />

            <!-- 2D Domain !-->
            <P PARAM="Xmin"                  VALUE="-1.0"        />
            <P PARAM="Ymin"                  VALUE="-2.0"        />
            <P PARAM="Xmax"                  VALUE="3.0"         />
            <P PARAM="Ymax"                  VALUE="2.0"         />
            <P PARAM="AOA"                   VALUE="15.0"        />
        </PARAMETERS>

    </MESHING>
</NEKTAR>
```

A list of the CAD surfaces which will have a prism generated on is described by `BndLayerSurfaces` and the thickness of the boundary to aim for is `BndLayerThickness`. The mesh generator has been created with a range of error messages to aid in debugging. If you encounter an error and the mesh generator fails, run `NekMesh` with the verbose `-v` flag and send the stdout with the .mcf and .step files to `d.moxey@exeter.ac.uk`. Without the feedback this functionality cannot improve.

4.6.2.1 Handling 3D geometries with voids

Although NekMesh supports the definition of 3D geometries that contain voids – for example, a sphere contained within a cube – at present it does require the definition of a point per-void that lies strictly on the interior of the void. This is so that tetrahedra on the interior of the void can be removed before the mesh is generated. For example, if one defines a geometry where two spheres of radius 1, centred at $(0, 0, 0)$ and $(2, 0, 0)$,

were contained within a larger domain, the void points can be specified through the `VOIDPOINTS` tag in the MCF as follows:

```
<NEKTAR>
    <MESHING>
        <!-- other parameters here... -->
        <VOIDPOINTS>
            <V> 0 0 0 </V>
            <V> 2 0 0 </V>
        </VOIDPOINTS>
   </MESHING>
</NEKTAR>
```

4.6.2.2 GEO format

Recent developments have been made to facilitate the generation of meshes from simple 2D and 3D geometries. The GEO file format, used by Gmsh, is a popular option that allows the user to script geometrical and meshing operations without the need of a GUI. A simplified reader has been implemented in NekMesh for 2D and 3D geometries. Although very basic this reader may be extended in the future to cover a wider range of geometrical features.

For a full description of the GEO format the user should refer to Gmsh's documentation. The following commands are currently supported:

- `//` and `/* */` (i.e. comments)
- `Point`
- `Line`
- `Spline` (through points)
- `BSpline` (i.e. a Bézier curve)
- `Ellipse` (arc): as defined in Gmsh's OpenCASCADE kernel, the first point defines the start of the arc, the second point the centre and the fourth point the end. The third point is not used. The start point along with the centre point form the major axis and the minor axis is then computed so that the end point falls onto the arc. The major axis must always be greater or equal to the minor axis.
- `Circle` (arc): the circle is a special case of the ellipse where the third point is skipped. The distances between the start and end points and the centre point must be equal or an error will be thrown.
- `Line Loop`
- `Plane Surface`
- `Ruled Surface` or, in newer versions of Gmsh, `Surface`

- `Surface Loop`
- `Volume`

At the present time, NekMesh does not support the full scripting capabilities of the GEO format, but the evaluation of simple variables is supported. The used GEO files should be a straightforward succession of entity creations (see list above). This should however allow for the creation of quite a wide range of 2D and 3D geometries by transformation of arbitrary curves into generic splines and arcs.

4.6.2.3 CCM format

For very complex 3D geometries with manifold bodies, *Nektar++* can take advantage of the meshing tool embedded in the Star-CCM package. `NekMesh` projects linear meshes generated in Star-CCM on the CAD definition to generate high-order meshes. *Nektar++* must be built `NEKTAR_USE_CCM` in conjunction with the `ccmio` package to use this functionality. Projection is performed by specifying the option `projectcad`.

The supported element types for the linear mesh are :

- `Tetrahedron`
- `Prism`
- `Pyramid`

Therefore it is recommended to use the `Surface Remesher`, the `Tetrahedral Mesher` and the `Prism Layer Mesher` in Star-CCM. The `CAD Projection` and `Generate Only Standard Prismatic Cells` options should be activated. The prismatic layer for the linear mesh in Star-CCM should not be splitted e.g. there should be only one layer with the total thickness of the boundary layer (see figure 4.7. Splitting will then be perfomed directly by `NekMesh` on surfaces specified by the argument `surf` and the number of layers (`layers` argument) and growth rate (`r` argument) are specified during the mesh conversion process with the `bl` option. Surfaces correspond to the regions created in Star-CCM and the reference index is displayed when running `NekMesh` with verbosity (`-v` option).

```
NekMesh -m projectcad:file=cadFile.STEP:order=meshOrder -m bl:surf=surf1,
    surf2,surf3:r=growthRate:layers=nLayers linearMesh.ccm outputMesh.xml
```

The command `linearise` can be added to correct bad elements.

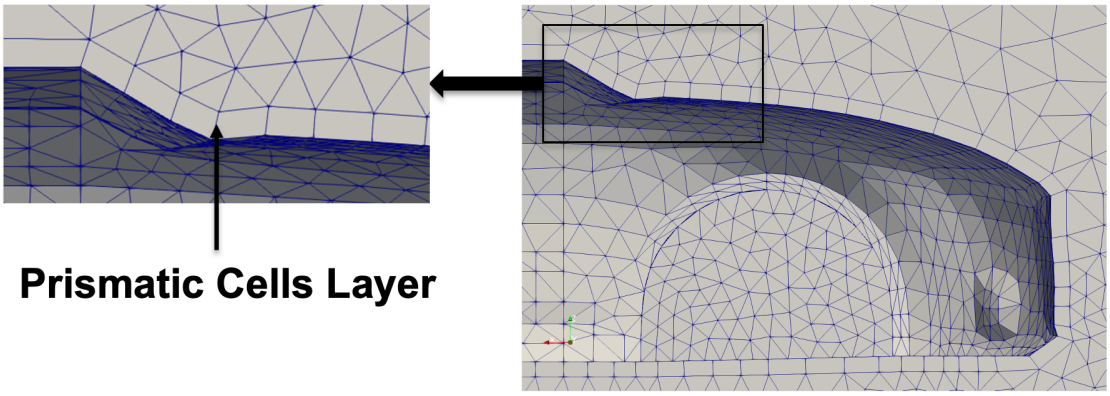


Figure 4.7 Linear mesh generated in Star-CCM (visualised with ParaView) showing the one-layer prismatic cells layer

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 formats which can be read by common visualisation and post-processing software, primarily Paraview/VisIt (in unstructured VTK `.vtu` format) or Tecplot/VisIt (in ASCII `.dat` or binary `.plt` formats). 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.

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 Basic usage

FieldConvert expects at least one input specification (such as a session file and its corresponding field file) and one output specification. These are specified on the command line as

```
FieldConvert in1.xml in2.fld out.dat
```

These can be combined with a processing module by adding the `-m` command line option. There can be more than one module specified, and they can appear anywhere in the command line arguments, although the order of execution is inferred from their order in the command line. For example, the command

```
FieldConvert in1.xml -m module1 in2.fld -m module2 out.dat
```

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

causes `in1.xml` and `in2.fld` to be read, followed by the `module1` processing module, the `module2` processing module, and finally output to the `out.dat` Tecplot file.

5.1.1 Input formats

FieldConvert supports XML and FLD-format files as produced by *Nektar++*. It also supports the reading of data files from two external spectral element codes: *Semtex*[2] and *Nek5000*[3]. These files can be directly converted to *Nektar++* format files by using the command

```
FieldConvert input.fld output.fld
```

Note that even though the `.fld` extension is typically associated with *Nektar++* files, FieldConvert can automatically identify *Semtex* and *Nek5000* input field files.

To use these files in a simulation, or to post-process the results of a simulation, an appropriate mesh must also be defined in the *Nektar++* XML format. `NekMesh` can be used to convert these input files to XML, as outlined in section 4.

5.2 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 or .plt 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
  ```

- Tecplot or VisIt(.plt format)

  ```
  FieldConvert test.xml test.fld test.plt
  ```

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

[2] http://users.monash.edu.au/ bburn/semtex.html
[3] https://nek5000.mcs.anl.gov

When converting to `.dat` or `.plt` format, it is possible to enable output with double precision, which is more accurate but requires larger disk space. For example, double precision output in plt. format can be produced with the command:

```
FieldConvert test.xml test.fld test.plt:plt:double
```

Tip

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

5.2.1 Using the VTK library for output

While the basic VTK output detailed above (.vtu unstructured grid files) works without the third party VTK library, *Nektar++* can also optionally be linked with the VTK library. By enabling VTK library support the creation of the VTK output files is offloaded to the VTK library rather than being manually written by *Nektar++*.

If you want to build a version of *Nektar++* with integrated VTK library support, you need to turn on the `NEKTAR_USE_VTK` switch in cmake. This will then look for a compatible VTK library on the system, VTK version 8.0.0 or higher is recommended for full functionality because this is the minimum version which allows for high-order VTK output. Versions older than this are supported but the high-order VTK output will be disabled.

Using the VTK library automatically enables file compression on the output .vtu file which can substantially reduce file sizes. This can be turned off with the command line option `uncompress` when using `FieldConvert`.

```
FieldConvert session.xml field.fld output.vtu:vtu:uncompress
```

If *Nektar++* has been compiled with `NEKTAR_USE_VTK` enabled then there are certain situations in which the legacy VTK output (manually writing rather than using the VTK library) must be used. The current VTK library output only supports 1D, 2D, 3D and 3DH1D (Quasi-3D) simulations. Any simulations with expansions other than those mentioned, such as 2DH2D will need to be converted using the `legacy` option to invoke the manual *Nektar++* VTK output.

```
FieldConvert session.xml field.fld output.vtu:vtu:legacy
```

The VTK library also adds two new output types that can be specified by the command line when choosing to output a .vtu format file, these are detailed below.

5.2.1.1 High-order output

High-order output can be enabled using the option `highorder` which first runs the mesh through the `equispacedoutput` module to get a truly equispaced mesh and then constructs a .vtu file using arbitrary-order Lagrange polynomials. This gives a more precise representation of the curvature of elements and allows for polynomial refinement using non-linear subdivisions. This should result in smaller output files as refinement of the mesh geometry can be computed on the fly. This allows for faster postprocessing in software such as Paraview due to the reduced memory requirements. High-order output is currently only implemented for non-homogenous expansions and does not support pyramid elements. A minimum VTK version of 8.0.0 is required for the high-order output, any VTK versions older than this will result in the high-order output being disabled.

```
FieldConvert session.xml field.fld output.vtu:vtu:highorder
```

5.2.1.2 Multi-block output

Multi-block output can be enabled using the option `multiblock` which splits the mesh in to separate blocks, one block for each domain, boundary, and any other composite. This allows each block to be toggled in the visualisation software independently.

Multi-block output uses the composite/boundary/domain names defined in the session file to label each block, so the ID numbers above will be replaced by the name if specified. It is also necessary to include the `CONDITIONS` section in the session file supplied to `FieldConvert` because this is where the boundary definitions are stored. Multi-block output is currently only implemented for non-homogenous expansions i.e. 1D, 2D, and 3D.

```
FieldConvert session.xml conditions.xml field.fld output.vtu:vtu:multiblock
```

An example multi-block tree structure is shown in figure 5.1, with the default output on the left and the output with the optional defined names on the right.

5.2.1.3 Improved FieldConvert filter

An additional advantage of using the VTK library is that when a filter is set up with `FieldConvert` to output a .vtu file, the mesh is now cached inbetween timesteps and only the new expansion data is written each filter call. This can provide significant speedups by keeping the geometry in memory. An example filter specifying a high-order and uncompressed .vtu output is shown below:

```
<FILTER TYPE="FieldConvert">
    <PARAM NAME="OutputFile">MyFile.vtu:vtu:highorder:uncompress</PARAM>
    <PARAM NAME="OutputFrequency">100</PARAM>
</FILTER>
```

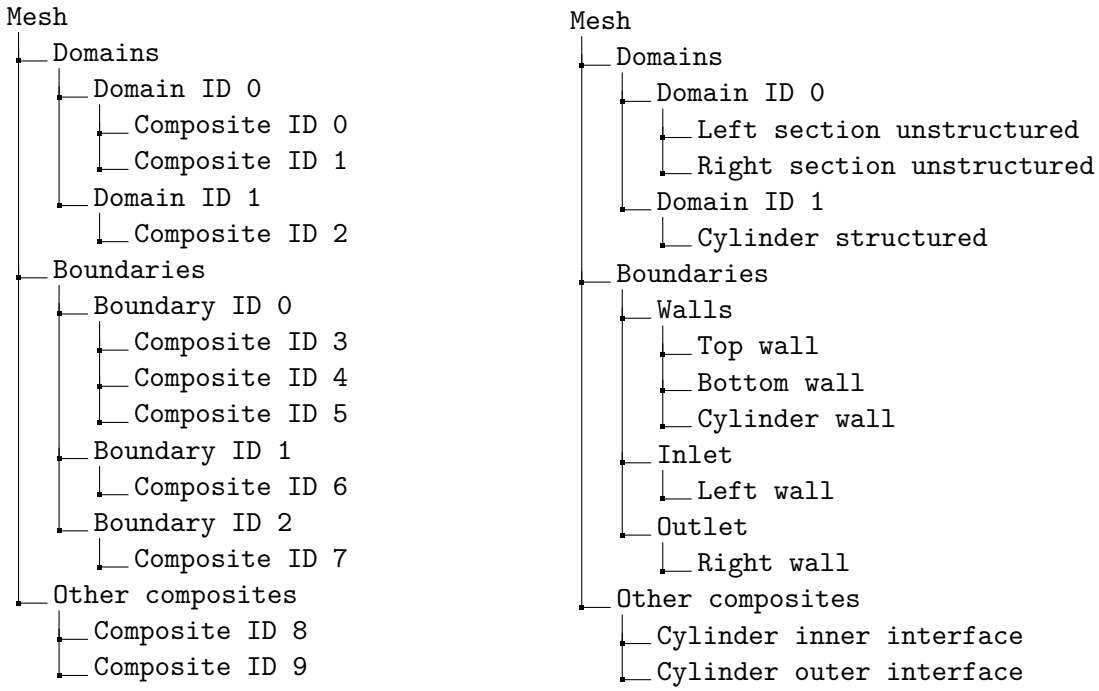


Figure 5.1 An example of the tree structure for a multi-block VTK output file. With the default IDs on the left and the optional names specified on the right.

5.3 Convert field files between XML and HDF5 format

When *Nektar++* is compiled with HDF5 support, solvers can select the format used for output of `.fld` files. FieldConvert can be used to convert between these formats using an option on the `.fld` output module. For example, if `in.fld` is stored in the default XML format, it can be converted to HDF5 format by issuing the command

```
FieldConvert in.fld out.fld:fld:format=Hdf5
```

5.4 Specifying a sub-range of the mesh

Fieldconvert allows for a sub-range of the mesh (computational domain) by using an xml input option 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 test.xml:xml:range=''-2,3,-1,2''  test.fld test.vtu
```

- Tecplot (.dat format)

```
FieldConvert  test.xml:xml:range=''2,3,-1,2'' test.fld test.dat
```

where the `range` information defines the range option of `xml` input module, 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.

5.5 FieldConvert in NekPy

The Python interface allows the user to instantiate input, output, and process modules by calling the static `Create` method of the `InputModule`, `ProcessModule`, and `OutputModule`, register configuration options, and run them. For example, consider the command:

```
FieldConvert -n 10 -m wss:bnd=0 session.xml field.fld  field_wss.fld
```

A Python script performing the same task is given below.

```
1 import sys
2 from NekPy.FieldUtils import *
3
4 field = Field(sys.argv, output_points=10)
5
6 InputModule.Create("xml", field, "session.xml").Run()
7 InputModule.Create("fld", field, "field.fld").Run()
8 ProcessModule.Create("wss", field, bnd="0").Run()
9 OutputModule.Create("fld" , field, "field_wss.fld").Run()
```

The key points are that the FieldConvert command line options, in this case `output-points`, are passed to the constructor of `Field`. The configuration options for a given module are passed to the static `Create` method of the `InputModule`, `ProcessModule`, and `OutputModule`. This creates the corresponding module and the modules can be run immediately after instantiation. Note that the first parameter of the `Create` method has to be the key for a given module, the second is the field variable and for input and output modules the remaining arguments may identify the input and output files for a given module. Optionally we can explicitly specify the file type of an input module using the "infile" keyword and the "outfile" keyword for output modules.

```
1 import sys
2 from NekPy.FieldUtils import *
3
4 field = Field(sys.argv, output_points=10)
5
6 InputModule.Create("xml", field, infile={"xml": "session.xml"}).Run()
7 InputModule.Create("fld", field, infile={"fld": "field.fld"}).Run()
```

```
ProcessModule.Create("wss", field, bnd="0").Run()
OutputModule.Create("fld" , field, outfile="field_wss.fld").Run()
```

It can also emulate the functionality of FieldConvert when using the `nparts` option in the following way. Here `session_xml` is a directory containing the mesh partitioned into 2.

```
import sys
from NekPy.FieldUtils import *

field = Field(sys.argv, nparts=2, force_output=True, error=True)

inputxml = InputModule.Create("xml", field, "session_xml")
inputfld = InputModule.Create("fld", field, "field.fld")
processwss = ProcessModule.Create("wss", field, bnd="2")
outputfld = OutputModule.Create("fld", field, "field_wss.fld")

for part in range(2):
       field.NewPartition(sys.argv, part)
       inputxml.Run()
       inputfld.Run()
       processwss.Run()
       outputfld.Run()

OutputModule.Create("info", field, "field_wss_b0.fld", nparts=2).Run()
```

The sub-range can also be specified by using the `xml` input module options in the following manner:

```
import sys
from NekPy.FieldUtils import *

field = Field(sys.argv)

inputxml = InputModule.Create("xml", field, , session.xml, range="-1,1,-1,1
    ").Run()
inputfld = InputModule.Create("fld", field, "field.fld").Run()
processwss = ProcessModule.Create("wss", field, bnd="2").Run()
outputfld = OutputModule.Create("fld", field, "field_wss.fld").Run()
```

5.6 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. `CFL`: Computes the CFL number over the solution domain for Incompressible flow
3. `QCriterion`: Computes the Q-Criterion for a given output file;

4. `L2Criterion`: Computes the Lambda 2 Criterion for a given output file;
5. `addcompositeid`: Adds the composite ID of an element as an additional field;
6. `fieldfromstring`: Modifies or adds a new field from an expression involving the existing fields;
7. `addFld`: Sum two .fld files;
8. `combineAvg`: Combine two *Nektar++* binary output (.chk or .fld) field file containing averages of fields (and possibly also Reynolds stresses) into single file;
9. `concatenate`: Concatenate a *Nektar++* binary output (.chk or .fld) field file into single file (deprecated);
10. `divergence`: Computes the divergence of the velocity field.
11. `dof`: Count the total number of DOF;
12. `equispacedoutput`: Write data as equi-spaced output using simplices to represent the data for connecting points;
13. `extract`: Extract a boundary field;
14. `gradient`: Computes gradient of fields;
15. `halfmodetofourier`: Convert `HalfMode` expansion to `SingleMode` for further processing;
16. `homplane`: Extract a plane from 3DH1D expansions;
17. `homstretch`: Stretch a 3DH1D expansion by an integer factor;
18. `innerproduct`: take the inner product between one or a series of fields with another field (or series of fields).
19. `interpfield`: Interpolates one field to another, requires fromxml, fromfld to be defined;
20. `interppointdatatofld`: Interpolates given discrete data using a finite difference approximation to a fld file given an xml file;
21. `interppoints`: Interpolates a field to a set of points. Requires fromfld, fromxml to be defined, and a topts, line, plane or box of target points;
22. `interpptstopts`: Interpolates a set of points to another. Requires a topts, line, plane or box of target points;
23. `isocontour`: Extract an isocontour of "fieldid" variable and at value "fieldvalue". Optionally "fieldstr" can be specified for a string definition or "smooth" for smoothing;

24. `jacobianenergy`: Shows high frequency energy of Jacobian;
25. `qualitymetric`: Evaluate a quality metric of the underlying mesh to show mesh quality;
26. `mean`: Evaluate the mean of variables on the domain;
27. `meanmode`: Extract mean mode (plane zero) of 3DH1D expansions;
28. `pointdatatofld`: Given discrete data at quadrature points project them onto an expansion basis and output fld file;
29. `printfldnorms`: Print L2 and LInf norms to stdout;
30. `removefield`: Removes one or more fields from .fld files;
31. `scalargrad`: Computes scalar gradient field;
32. `scaleinputfld`: Rescale input field by a constant factor;
33. `shear`: Computes time-averaged shear stress metrics: TAWSS, OSI, transWSS, TAAFI, TACFI, WSSG;
34. `streamfunction`: Calculates stream function of a 2D incompressible flow;
35. `surfdistance`: Computes height of a prismatic boundary layer mesh and projects onto the surface (for e.g. y^+ calculation);
36. `vorticity`: Computes the vorticity field;
37. `wss`: Computes wall shear stress field;
38. `phifile`: Computes the Φ function representing a body defined in an `.stl` file. Useful for the Smoothed Profile Method solver;
39. `wallNormalData`: Interpolate values for a set of points in the wall-normal direction (for e.g. extract boundary layer profiles);
40. `bodyFittedVelocity`: Generate a body-fitted coordinate system and then project the velocity components from Cartesian coordinate system into the new coordinate system;
41. `averagefld`: Compute the averaging of several field files;
42. `powerspectrum`: Compute power spectrum at given regions from a 3DH1D expansion;
43. `zero-homo-plane`: Zero a specfied homogeneous plane in wavespace.

The module list above can be seen by running the command

```
FieldConvert -l
```

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

5.6.1 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 5.2 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

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

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.

5.6.2 Calculate CFL number: *CFL* module

This module calculates the CFL number over the domain. It currently only supports the solution results from the Incompressible flow simulaitons, i.e. the outputs of `IncNavierStokesSolver`. To Estimate the *CFL* number, the user can run

```
FieldConvert -m CFL test.xml test.fld test-cfl.fld
```

where the file `test-cfl.fld` can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.3 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 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.4 Calculate λ_2: *L2Criterion* module

To perform the λ_2 vortex detection calculation and obtain an output data containing the values of the λ_2 eigenvalue, the user can run

```
FieldConvert -m L2Criterion test.xml test.fld test-L2Crit.fld
```

where the file `test-L2Crit.fld` can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.5 Add composite ID: *addcompositeid* module

When dealing with a geometry that has many surfaces, we need to identify the composites to assign boundary conditions. To assist in this, FieldConvert has a `addcompositeid` module, which adds the composite ID of every element as a new field. To use this we simply run

```
FieldConvert -m addcompositeid mesh.xml out.dat
```

In this case, we have produced a Tecplot file which contains the mesh and a variable that contains the composite ID. To assist in boundary identification, the input file `mesh.xml` should be a surface XML file that can be obtained through the `NekMesh` `extract` module (see section 4.5.3).

5.6.6 Add new field: *fieldfromstring* module

To modify or create a new field using an expression involving the existing fields, one can use the `fieldfromstring` module of FieldConvert

```
FieldConvert -m fieldfromstring:fieldstr="x+y+u":fieldname="result" \
file1.xml file2.fld file3.fld
```

In this case `fieldstr` is a required parameter describing a function of the coordinates and the existing variables, and `fieldname` is an optional parameter defining the name of the new or modified field (the default is newfield). `file3.fld` is the output containing both the original and the new fields, and can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.7 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 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.8 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 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.9 Concatenate two files: *concatenate* module

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

```
FieldConvert 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 5.2 to visualise the result either in Tecplot, Paraview or VisIt. The `concatenate` module previously used for this purpose is not required anymore, and will be removed in a future release.

5.6.10 Count the number of DOF: *dof* module

To count the number of DOF in a solution file, one can run the following command

```
FieldConvert -m dof file.xml file.fld out.stdout
```

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

This module interpolates the output data to a truly equispaced set of points (not equispaced along the collapsed coordinate system). Therefore a tetrahedron is represented by a tetrahedral number of points. 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 equispacedoutput test.xml test.fld test.vtu
```

Note

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

5.6.12 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 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 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.

5.6.13 Calculate divergence: *divergence* module

To perform the divergence calculation and obtain an output field containing the divergence solution, the user can run

```
FieldConvert -m divergence test.xml test.fld test-div.fld
```

where the file `test-div.fld` can be processed in a similar way as described in section 5.2.

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

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

```
FieldConvert -m gradient:vars=u,v,p:dirs=x,y test.xml  test.fld  test-grad.
    fld
```

or

```
FieldConvert -m gradient:vars=0,1,2:dirs=0,1 test.xml  test.fld  test-grad.
    fld
```

where `vars` specifies the field numbers (starting from 0) or variable names to process and `vars` specifies the spatial directions to compute derivatives. All fields will be processed if `vars` is not set. All spatial directions will be used if `vars` is not set. The output file `file-grad.fld` can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.15 Convert HalfMode expansion to SingleMode for further processing: *halfmodetofourier* module

To obtain full Fourier expansion form a `HalfMode` result, use the comand:

```
FieldConvert -m halfmodetofourier:realmodetoimag=value1,value2 file.xml
    half_mode_file.fld single_mode_file.fld
```

In linear stability analysis with halfmode expansion, the variable w, whose variable number is 2, is a imaginary mode but it is output as a real mode in the eigenvector field

file. By setting option `realmodetoimag=2`, w can be transformed to a imaginary mode correctly.

5.6.16 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 5.2 to visualise it either in Tecplot or in Paraview.

5.6.17 Stretch a 3DH1D expansion: *homstretch* module

To stretch a 3DH1D expansion in the z-direction, use the command:

```
FieldConvert -m homstretch:factor=value file.xml file.fld file-stretch.fld
```

The number of modes in the resulting field can be chosen using the command-line parameter `--output-points-hom-z`.

The output file `file-stretch.fld` can be processed in a similar way as described in section 5.2 to visualise it either in Tecplot or in Paraview.

5.6.18 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.

5.6.19 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.

If the fromfld is a 3DH1D field and uses HalfMode expansion, you can use `realmodetoimag=`$n_1, n_2, ..., n_m$ to transform the $n_1, n_2, ..., n_m$th variables from real modes to imaginary modes. Variable

index starts from 0.

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.

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

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

```
FieldConvert -m interppointdatatofld:frompts=file1.pts file1.xml file1.fld
```

or alternatively for csv data:

```
FieldConvert -m interppointdatatofld:frompts=file1.csv file1.xml file1.fld
```

This command will interpolate the data from `file1.pts` (`file1.csv`) to the mesh and expansions defined in `file1.xml` and output the field to `file1.fld`. The file `file.pts` must be 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. An equivalent csv file is:

```
# x, 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
```

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:frompts=1D-file1.pts:interpcoord=1 \
        3D-file1.xml 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.

5.6.21 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:topts=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>
```

Similar to the *interppointdatatofld* module, the `.pts` file can be interchanged with a `.csv` file (the output can also be written to `.csv`):

```
# x, y
0.0,0.0
0.5,0.0
1.0,0.0
```

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 file2.dat
```

can be specified where `npts` is the number of equispaced points between (x_0, y_0) to (x_1, y_1). This also works in 3D, by specifying (x_0, y_0, z_0) to (x_1, y_1, z_1).

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 file2.dat
```

where `npts1,npts2` is the number of equispaced points in each direction and (x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) 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 file2.dat
```

where `npts1,npts2,npts3` is the number of equispaced points in each direction and $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ define the limits of the box of points.

There is also 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.

If the fromfld is a 3DH1D field and uses HalfMode expansion, you can use `realmodetoimag=`$n_1, n_2, ..., n_m$ to transform the $n_1, n_2, ..., n_m$th variables from real modes to imaginary modes. Variable index starts from 0.

Note

This module runs in parallel for the line, plane and box extraction of points.

5.6.22 Interpolate a set of points to another: *interpptstopts* module

You can interpolate one set of points to another using the following command:

```
FieldConvert file1.pts -m interpptstopts:topts=file2.pts file2.dat
```

This command will interpolate the data in `file1.pts` to a new set of points defined in `file2.pts` and output it to `file2.dat`.

Similarly to the *interppoints* module, the target point distribution can also be specified using the `line`, `plane` or `box` options. The optional arguments `clamptolowervalue`, `clamptouppervalue`, `defaultvalue` and `cp` are also supported with the same meaning as in *interppoints*.

One useful application for this module is with 3DH1D expansions, for which currently the *interppoints* module does not work. In this case, we can use for example

```
FieldConvert file1.xml file1.fld -m interpptstopts:\
    plane=npts1,npts2,x0,y0,z0,x1,y1,z1,x2,y2,z2,x3,y3,z3 \
    file2.dat
```

With this usage, the *equispacedoutput* module will be automatically called to interpolate the field to a set of equispaced points in each element. The result is then interpolated to a plane by the *interpptstopts* module.

Note

This module does not work in parallel.

5.6.23 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 $u + v$ 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 op-

tion 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.

5.6.24 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 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.25 Calculate mesh quality: *qualitymetric* module

The `qualitymetric` module assesses the quality of the mesh by calculating a per-element quality metric and adding an additional field to any resulting output. This does not require any field input, therefore an example usage looks like

```
FieldConvert -m qualitymetric mesh.xml mesh-with-quality.dat
```

Two quality metrics are implemented that produce scalar fields Q:

- By default a metric outlined in [16] is produced, where all straight sided elements have quality $Q = 1$ and $Q < 1$ shows the deformation between the curved element and the straight-sided element. If $Q = 0$ then the element is invalid. Note that Q varies over the volume of the element but is not guaranteed to be continuous between elements.
- Alternatively, if the `scaled` option is passed through to the module, then the scaled Jacobian

$$J_s = \frac{\min_{\xi \in \Omega_{\text{st}}} J(\xi)}{\max_{\xi \in \Omega_{\text{st}}} J(\xi)}$$

(i.e. the ratio of the minimum to maximum Jacobian of each element) is calculated. Again $Q = 1$ denotes an ideal element, but now invalid elements are shown by $Q < 0$. Any elements with Q near zero are determined to be low quality.

5.6.26 Evaluate the mean of variables on the domain: *mean* module

To evaluate the mean of variables on the domain one can use the `mean` module of FieldConvert

```
FieldConvert -m mean file1.xml file2.fld out.stdout
```

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

5.6.27 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 5.2 to visualise the result either in Tecplot or in Paraview or VisIt.

5.6.28 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 -m pointdatatofld:frompts=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>
```

```
9 </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. Alternatively, the `file.pts` can be interchanged with a csv file.

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

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

In this case the pointdatatofld module should be used without the `-no-equispaced` 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 advisable to use the set of points given by

```
FieldConvert --no-equispaced file.xml file.dat
```

which produces a set of points at the gaussian collapsed coordinates.

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.

5.6.29 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.

5.6.30 Removes one or more fields from .fld files: *removefield* module

This module allows to remove one or more fields from a .fld file:

```
FieldConvert -m removefield:fieldname="u,v,p" test.xml test.fld test-
    removed.fld
```

where the file `test-removed.fld` can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt. The lighter resulting file speeds up the postprocessing of large files when not all fields are required.

5.6.31 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.

5.6.32 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-scal.fld` can be processed in a similar way as described in section 5.2 to visualise the result either in Tecplot, Paraview or VisIt.

5.6.33 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|.

5.6.34 Stream function of a 2D incompressible flow: *streamfunction* module

The streamfunction module calculates the stream function of a 2D incompressible flow, by solving the Poisson equation

$$\nabla^2 \psi = -\omega$$

where ω is the vorticity. Note that this module applies the same boundary conditions specified for the y-direction velocity component `v` to the stream function, what may not be the most appropriate choice.

To use this module, the user can run

```
FieldConvert -m streamfunction test.xml test.fld test-streamfunc.fld
```

where the file `test-streamfunc.fld` can be processed in a similar way as described in section 5.2.

5.6.35 Boundary layer height calculation: *surfdistance* module

The surface distance module computes the height of a boundary layer formed by quadrilaterals (in 2D) or prisms and hexahedrons (in 3D) 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`.

5.6.36 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 5.2.

5.6.37 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.
The output fields are the wall shear stress in the streamwise direction (`Shear_x`), the vertical direction (`Shear_y`) and its magnitude (`Shear_mag`) for 2D simulations. For 3D simulations, wall shear stress in the spanwise direction (`Shear_z`) is also included in the output fileds.

5.6.38 Calculating the shape function Φ for an SPM case: *phifile* module

Note

This module is in experimental phase and only runs in serial. When reading 3D geometries from `.stl` files, errors may occur if triangles are placed exactly at 90°.

This FieldConvert module converts a binary `.stl` CAD file into `.fld` or `.vtu`/`.dat` files that can be used as inputs for the Smoothed Profile Method solver. Running the command:

```
FieldConvert -m phifile:file=geom.stl:scale=value session.xml geom.fld
```

will generate an output file `geom.fld` with all the information required to define a shape function Φ representing the geometry specified in `geom.stl`. The option `scale` sets the value of ξ as it is described in the Synopsis chapter of the Incompressible N-S solver. If the output file gets the extension `.vtu` or `.dat`, the module will produce a graphical representation of the shape function; however, the recommended way to proceed is to

generate an `.fld` file and then, use FieldConvert to obtain the `.vtu` or `.dat` files if needed for visualisation purposes:

```
FieldConvert -m phifile:file=geom.stl:scale=value session.xml geom.fld
FieldConvert session.xml geom.fld geom.vtu
```

This module can also be used to produce a `.fld` and `.vtu` / `.dat` file of shapes defined directly in the session file through an analytical expression. In this case, the commands simplify to:

```
FieldConvert -m phifile session.xml geom.fld
FieldConvert session.xml geom.fld geom.vtu
```

The algorithm computes an octree for the triangles that define the 3D object in the `.stl` file, and then loops over all the nodes of the computational mesh looking for the shortest distance to the 3D object. Since this module is currently in experimental phase, it only runs in serial and therefore its performance in computing the shape function from the `.stl` file is limited, especially in 3D cases. In addition to this, it is recommended to make sure that the angles between triangles are not strictly equal to 90°, since the algorithm will probably fail to find the real Φ function.

5.6.39 Interpolate values for a point array: *wallNormalData* module

To obtain the values of an array of points in the wall-normal direction, such as boundary layer profiles, the user can run:

```
FieldConvert -m wallNormalData:bnd=0:xorig="0.5,0,0":projDir="0,1,0":
    maxDist=0.1:\
    distH=0.1:nptsH=0.01:d=0.2 test.xml test.fld test.pts
```

The option `bnd` specifies the target boundary region, to which the input point specified by `xorig` will be projected as the first point in the interpolation points array. The projection direction is specified by `projDir`, which does not have to be a unit vector but will be automatically normalized. The input point and projetion direction must be set carefully to make sure a projected point can be found on the target boundary. In cases with curved boundaries on which multiple projected points might exist, the user can use `maxDist` to limit the maximum projection distance between the input point to the projected point.

After the projected point is found, this module will compute the wall-normal direction at the point, and an array of points will be set in this direction. The number of points is specified by `nptsH`, and the parameter `d` controls the distribution of the points as follows

$$h(\xi) = H\left\{1 - \frac{\tanh\left[(1-\xi)atanh(\sqrt{1-\delta})\right]}{\sqrt{1-\delta}}\right\}$$

where ξ is equally-spaced parametric parameter varying from 0 to 1; H is the distance between the first point (on the booundary) and the last point, which is specified `distH`; δ is given by `d`. This distribution is employed for $0 < \delta \leq 0.95$ while $\delta > 0.95$ for evenly-spaced distribution.

Finally, this module will interpolate the values at the points from test.fld, and save the result in test.pts.

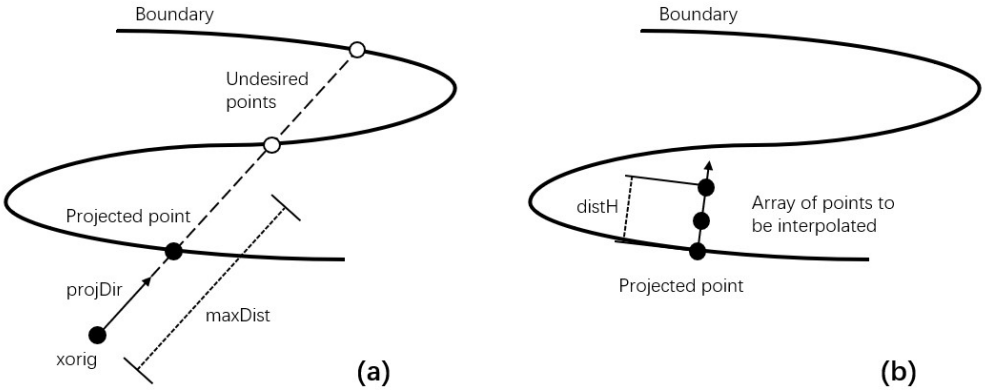


Figure 5.2 (a) Project the input point onto the boundary, (b) Array of points to be interpolated.

In practice, it is convenient to use this module interactively to find/visualize/adjust profile distributions. This can be achieved by using the NekPy interface. A sample Pyhton script is provieded in the following

```
import sys
from NekPy.FieldUtils import *

field = Field(sys.argv, forceoutput=True, error=True)
InputModule.Create("xml", field, "mesh.xml","session.xml").Run()
InputModule.Create("fld", field, "field.fld").Run() # e.g. x,y,u,v,p
ProcessModule.Create("wallNormalData", field,
                  bnd=str(bnd), xorig=str(xorig), projDir=str(projDir),
                  maxDist=str(maxDist), distH=str(distH), nptsH=str(nptsH),
                  d=str(d)).Run()

# Get data arrays from interpolation
y = field.GetPts(1)
u = field.GetPts(2)
```

5.6.40 Project velocity into body-fitted coordinate system: *bodyFittedVelocity* module

To obtain the velocity components in the body-fitted coordinate system, the user can run:

```
FieldConvert -m bodyFittedVelocity:bnd=0:assistDir="0,0,1":checkAngle=1:\
    distTol=1.0e-12:bfcOut=1 mesh.xml session.xml field.fld bfvFile.fld
```

The option `bnd` specifies the reference boundary composite to set up the body-fitted coordinate system with the help of vector specified by `assistDir`. For any point inside the domain, the the body-fitted coordinate system has three directions, i.e. main tangantial direction (0), normal direction (1) from the nearest point on the wall to the point , and minor tangential direction (2) specified by `assistDir`. Since the directions 1 and 2 are known, the main tangantial direction (0) can be computed by the cross product of them.

In general, when the surface is smooth, `checkAngle` and `distTol` is not needed to set up the body-fitted coordinate system since they are specially designed for geometries with surfarce irregularities, such a step or gap. As shown in the following sketch, the presence of a gap in 2D geometry introduces fours singular points at the cornors, where the body-fited velocity components will experience significant change, leading to jump in the final fields. Considering the surfaces irregularities are typically small in size, the user can choose to exclude the side wall(s) as reference to set up the body-fitted coordinate system by setting boundary id=0,2,4 into one composite while id=1,3 into another. Then only the first composte id is specified through `bnd`. In this way the finaly results will be comparable with the results using the clean geometry.

Finally, `bfcOut` is a bool parameter controls whether the fields for body-fitted coordinate system will be output. This file can be read by the `BodyFittedVelocity` filter to process data. See 3.5.27.

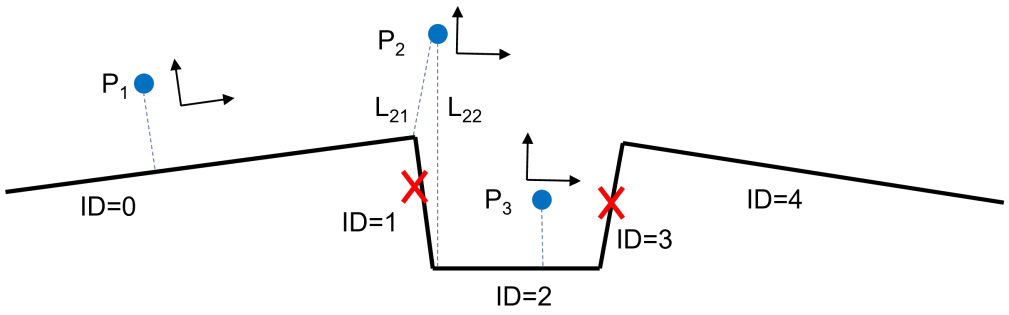


Figure 5.3 Body-fitted coordinate system when the geometry has a gap on the surace.

5.6.41 Zero a homogeneous plane in wavespace: *zero-homo-plane* module

In cases the users would like to visualize the fluctuations in a 3DH1D flow field, the mean mode is expected to be removed (in the Fourier space). This can be achieved by running the current module:

```
FieldConvert -m zero-homo-plane:planeid=0 mesh.xml session.xml field.fld \
    field-zeroMean.fld
```

In general, `planeid` can be any meaningful plane id, which is 0 by default.

5.6.42 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 12. 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
```

5.6.43 Field averaging module

To obtain the averaged field of several field files, the user can run:

```
FieldConvert -m averagefld:inputfld=chan3DH1D_%d.chk:range=0,1,4 chan3DH1D.
    xml phaseavg.fld
```

The option `inputfld` specifies the format of field file name. The option `range=start,skip,end` specifies the range of %d in the input file name.

5.6.44 Computing the spanwise power spectrum module

To obtain the spanwise power spectrum of a 3DH1D field in a specific domain, the user can run:

```
FieldConvert -m powerspectrum:vars=u,v,w:box=0,0.5,0,0.5 chan3DH1D.xml
    chan3DH1D_0.chk chan3DH1D.plt
```

The option `vars=u,v,w` specifies the variables used to calculate the power spectrum. Both the number and variable name can be used. The option `box=xmin,xmax,ymin,ymax` specifies the domain where the power spectrum is computed.

5.6.45 Computing the velocity field induced by vortex filaments

To obtain the vortex-induced velocity, the user can run:

```
FieldConvert -m vortexinducedvelocity:vortex=vortexfilament.dat  Helmholtz.
    xml Helmholtz.plt
```

The Lamb-Oseen vortex model is used to avoid the singularity at the vortex center. In the cylindrical coordinate, the vorticity and velocity are

$$\omega_z(r) = \frac{\Gamma}{2\pi\sigma^2}\exp\left(-\frac{r^2}{2\sigma^2}\right)$$

$$v_r(r) = \frac{\Gamma}{2\pi r}\left[1-\exp\left(-\frac{r^2}{2\sigma^2}\right)\right]$$

The option `vortex` specifies the filename that defines the vortex filaments. The format of one vortex segment is: xstart, ystart, zstart, is half infinity (true is 1); xend, yend, zend, is half infinity; σ, Γ. The format to define a uniform flow is: u, v, w. One example of `vortexfilament.dat` that defines two-counter rotating Lamb-Oseen vortices with a uniform background flow is

```
0 0.159154943091895 0
-0.5 0 -1 1; -0.5 0 1 1; 0.1 -1
0.5 0 -1 1; 0.5 0 1 1; 0.1 1
```

5.7 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
```

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

or

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

replacing `<nprocs>` with the number of processors. For the `.dat` and `.plt` outputs the current version will produce a single output file. However it is also sometimes useful to produce multiple output files, one for each partition, and this can be done by using the `writemultiplefiles` option, i.e.

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

```
mpirun -np <nprocs> FieldConvert test.xml test.fld \
                test.plt:plt:writemultiplefiles
```

For the `.vtu` format multiple files will by default be produced of the form `test_vtu/P0000000.vtu`, test_vtu/P0000001.vtu, test_vtu/P0000002.vtu. For this format an additional file called `test.pvtu` is written out which allows for parallel reading of the individual `.vtu` files.

FieldConvert functions that produce a `.fld` file output will also be created when running in parallel. In this case when producing a .fld file a directory called `test.fld` (or the specified output name) is created with the standard parallel field files placed within the directory.

5.8 FieldConvert for parallel-in-time

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

```
mpirun -np <nprocs> FieldConvert --npt <nprocs> test.xml \
```

```
                in/test_<n>.fld out/test_<n>.vtu
```

```
mpirun -np <nprocs> FieldConvert --npt <nprocs> test.xml \
                in/test_<n>.fld out/test_<n>.dat
```

replacing `<nprocs>` with the number of processors and `<n>` by the starting time index while the subdirectories `in` and `out` are optional. FieldConvert will then process in parallel time solutions `test_<n>`.dat to `test_<n>+<nprocs>`.dat.

5.9 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 or creating an output file for visualization.

5.9.1 Using the *part-only* and *part-only-overlapping* options

Loading full `file1.xml` can be expensive if the `file1.xml` is already large. So instead you can pre-partition the file using the using the `-part-only` option. So the command

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

will partition the mesh into 10 partitions 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.

5.9.2 Using the *nparts* options

If you have a partitioned directory either from a parallel run or using the `-part-only` option you can now run the `FieldConvert` option using the `nparts` command line

option, that is

```
FieldConvert --nparts 10  file1_xml:xml file1.fld file1.vtu
```

Note the form `file1_xml:xml` option tells the code it is a parallel partition which should be treated as an `xml` type file. the argument of `nparts` should correpsond to the number of partitions used in generating the file1_xml directory. This will create a parallel vtu file as it processes each partition.

Another example is to interpolate `file1.fld` from one mesh `file1.xml` to another `file2.xml`. If the mesh files are large we can do this by partitioning `file2.xml` into 10 (or more) partitions to generate the `file_xml` directory and interpolating each partition one by one using the command:

```
FieldConvert --nparts 10 -m interpfield:fromxml=file1.xml:fromfld=file1.fld
     \
file2_xml:xml file2.fld
```

Note that internally the routine uses the range option so that it only has to load the part of `file1.xml` that overlaps with each partition of `file2.xml`. 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`. In previous versions of FieldConvert it was necessary to generate an updated `Info.xml` file but in the current version it should automatically be updating this file.

5.9.3 Running in parallel with the *nparts* option

The examples above will process each partition serially which may now take a while for many partitions. You can however run this option in parallel using a smaller number of cores than the nparts.

For the example of creating a vtu file above you can use 4 processor concurrently with the command line:

```
mpirun -n 4 FieldConvert --nparts 10   file1_xml:xml file1.fld file1.vtu
```

Obviously the executable will have to have been compiled with the MPI option for this to work.

Part III

Solver Applications

Chapter 6

Acoustic Solver

6.1 Synopsis

The aim of the AcousticSolver is to predict acoustic wave propagation. Through the application of a splitting technique, the flow-induced acoustic field is totally decoupled from the underlying hydrodynamic field.

6.1.1 Linearized Euler Equations

The Linearized Euler Equations (LEE) are obtained by linearizing the Euler Equations about a mean flow state $\left(\overline{\rho}, \overline{c^2}, \overline{\boldsymbol{u}}\right)$. Hence, they describe the evolution of perturbations $(p^{\mathrm{a}}, \rho^{\mathrm{a}}, \overline{\rho}\boldsymbol{u}^{\mathrm{a}})$ around this state. In conservative form, the LEE are given as:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}_1}{\partial x_1} + \frac{\partial \boldsymbol{F}_2}{\partial x_2} + \frac{\partial \boldsymbol{F}_3}{\partial x_3} + \boldsymbol{C}\boldsymbol{U} = \boldsymbol{W} \tag{6.1}$$

with

$$\boldsymbol{U} = \begin{bmatrix} p^{\mathrm{a}} \\ \rho^{a} \\ \overline{\rho} u_1^a \\ \overline{\rho} u_2^a \\ \overline{\rho} u_3^a \end{bmatrix}, \tag{6.2}$$

$$\boldsymbol{F}_1 = \begin{bmatrix} \overline{\rho} u_1^a \overline{c^2} + \overline{u}_1 p^{\mathrm{a}} \\ \overline{\rho} u_1^a + \overline{u}_1 \rho^a \\ \overline{\rho} u_1^a \overline{u}_1 + p^{\mathrm{a}} \\ \overline{\rho} u_2^a \overline{u}_1 \\ \overline{\rho} u_3^a \overline{u}_1 \end{bmatrix}, \quad \boldsymbol{F}_2 = \begin{bmatrix} \overline{\rho} u_2^a \overline{c^2} + \overline{u}_2 p^{\mathrm{a}} \\ \overline{\rho} u_2^a + \overline{u}_2 \rho^a \\ \overline{\rho} u_1^a \overline{u}_2 \\ \overline{\rho} u_2^a \overline{u}_2 + p^{\mathrm{a}} \\ \overline{\rho} u_3^a \overline{u}_2 \end{bmatrix}, \quad \boldsymbol{F}_3 = \begin{bmatrix} \overline{\rho} u_3^a \overline{c^2} + \overline{u}_3 p^{\mathrm{a}} \\ \overline{\rho} u_3^a + \overline{u}_3 \rho^a \\ \overline{\rho} u_1^a \overline{u}_3 \\ \overline{\rho} u_2^a \overline{u}_3 \\ \overline{\rho} u_3^a \overline{u}_3 + p^{\mathrm{a}} \end{bmatrix}, \tag{6.3}$$

$$\boldsymbol{C} = \begin{bmatrix} (\gamma-1)\frac{\partial \overline{u}_k}{\partial x_k} & 0 & \frac{1}{\overline{\rho}}(1-\gamma)\frac{\partial \overline{p}}{\partial x_1} & \frac{1}{\overline{\rho}}(1-\gamma)\frac{\partial \overline{p}}{\partial x_2} & \frac{1}{\overline{\rho}}(1-\gamma)\frac{\partial \overline{p}}{\partial x_3} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \overline{u}_k \frac{\partial \overline{u}_1}{\partial x_k} & \frac{\partial \overline{u}_1}{\partial x_1} & \frac{\partial \overline{u}_1}{\partial x_2} & \frac{\partial \overline{u}_1}{\partial x_3} \\ 0 & \overline{u}_k \frac{\partial \overline{u}_2}{\partial x_k} & \frac{\partial \overline{u}_2}{\partial x_1} & \frac{\partial \overline{u}_2}{\partial x_2} & \frac{\partial \overline{u}_2}{\partial x_3} \\ 0 & \overline{u}_k \frac{\partial \overline{u}_3}{\partial x_k} & \frac{\partial \overline{u}_3}{\partial x_1} & \frac{\partial \overline{u}_3}{\partial x_2} & \frac{\partial \overline{u}_3}{\partial x_3} \end{bmatrix}. \tag{6.4}$$

By default, the source term vector $\boldsymbol{W}$ is zero and has to be specified by an appropriate forcing.

6.1.2 Acoustic Perturbation Equations

The acoustic perturbation equations (APE-1/APE-4) proposed by Ewert and Schroeder [14] assure stable aeroacoustic simulations. These equations are similar to the LEE, but account for acoustic perturbations exclusively. The AcousticSolver implements the APE-1/4 type operator:

$$\frac{\partial p^{\mathrm{a}}}{\partial t} + \overline{c^2} \nabla \cdot \left(\overline{\rho} \boldsymbol{u}^{\mathrm{a}} + \overline{\boldsymbol{u}} \frac{p^{\mathrm{a}}}{\overline{c^2}} \right) = \dot{\omega}_{\mathrm{c}} \tag{6.5a}$$

$$\frac{\partial \boldsymbol{u}^{\mathrm{a}}}{\partial t} + \nabla \left(\overline{\boldsymbol{u}} \cdot \boldsymbol{u}^{\mathrm{a}} \right) + \nabla \left(\frac{p^{\mathrm{a}}}{\overline{\rho}} \right) = \dot{\boldsymbol{\omega}}_{\mathrm{m}} \, , \tag{6.5b}$$

where $(\overline{\boldsymbol{u}}, \overline{c^2}, \overline{\rho})$ represents the base flow and $(\boldsymbol{u}^{\mathrm{a}}, p^{\mathrm{a}})$ the acoustic perturbations. Similar to the LEE, the acoustic source terms $\dot{\omega}_{\mathrm{c}}$ and $\dot{\boldsymbol{\omega}}_{\mathrm{m}}$ are by default zero and must be specified e.g. by an appropriate forcing. This way, e.g. the APE-1, APE-4 [14] or revised APE equations [17] can be obtained. Expressed as hyperbolic conservation law, the APE-1/4 operator reads:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}_1}{\partial x_1} + \frac{\partial \boldsymbol{F}_2}{\partial x_2} + \frac{\partial \boldsymbol{F}_3}{\partial x_3} = \boldsymbol{W} \tag{6.6}$$

with

$$\boldsymbol{U} = \begin{bmatrix} p^{\mathrm{a}} \\ u_1^{\mathrm{a}} \\ u_2^{\mathrm{a}} \\ u_3^{\mathrm{a}} \end{bmatrix}, \tag{6.7}$$

$$\boldsymbol{F_1} = \begin{bmatrix} \overline{\rho c^2} u_1^{\mathrm{a}} + p^{\mathrm{a}} \overline{u}_1 \\ \overline{u}_j u_j^{\mathrm{a}} + p^{\mathrm{a}}/\overline{\rho} \\ 0 \\ 0 \end{bmatrix}, \ \boldsymbol{F_2} = \begin{bmatrix} \overline{\rho c^2} u_2^{\mathrm{a}} + p^{\mathrm{a}} \overline{u}_2 \\ 0 \\ \overline{u}_j u_j^{\mathrm{a}} + p^{\mathrm{a}}/\overline{\rho} \\ 0 \end{bmatrix}, \ \boldsymbol{F_3} = \begin{bmatrix} \overline{\rho c^2} u_3^{\mathrm{a}} + p^{\mathrm{a}} \overline{u}_3 \\ 0 \\ 0 \\ \overline{u}_j u_j^{\mathrm{a}} + p^{\mathrm{a}}/\overline{\rho} \end{bmatrix}. \tag{6.8}$$

6.2 Usage

```
AcousticSolver session.xml
```

6.3 Session file configuration

Parameters

Under this section it is possible to set the parameters of the simulation.

```
<PARAMETERS>
  <P> TimeStep       = 1e-05  /P>
  <P> NumSteps       = 1000   /P>
  <P> FinTime        = 0.01   /P>
  <P> IO_CheckSteps  = 100    /P>
  <P> IO_InfoSteps   = 10     /P>
  <P> IO_CFLSteps    = 10     /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;
- `IO_CFLSteps` sets the number of steps between successive Courant number stats are printed to screen;

6.3.1 Time Integration Scheme

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> RungeKutta </METHOD>
  <VARIANT> SSP </VARIANT>
  <ORDER> 3 </ORDER>
</TIMEINTEGRATIONSCHEME>
```

- `Method` is the time-integration method. Note that only an explicit discretisation is supported.
- `Order` is the order of the time-integration method.
- `Variant` is the variant of the time-integration method (variables for Runga Kutta: `Blank, SSP`).

6.3.2 Solver Info

```
<SOLVERINFO>
  <I PROPERTY="EQType"              VALUE="APE"              />
  <I PROPERTY="Projection"          VALUE="DisContinuous"    />
  <I PROPERTY="UpwindType"          VALUE="LaxFriedrichs"    />
</SOLVERINFO>
```

- `EQType` is the tag which specify the equations we want solve:
  - `APE` Acoustic Perturbation Equations (variables: `p,u,v,w`);
  - `LEE` Linearized Euler Equations (variables: `p,rho,rhou,rhov,rhow`).
- `Projection` is the type of projection we want to use. Currently, only `DisContinuous` is supported.
- `AdvectionType` is the advection operator. Currently, only `WeakDG` (classical DG in weak form) is supported.
- `UpwindType` is the numerical interface flux (i.e. Riemann solver) we want to use for the advection operator (see [25] for the implemented formulations):
  - `Upwind`;
  - `LaxFriedrichs`;

6.3.3 Variables

For the APE operator, the acoustic pressure and velocity perturbations are solved, e.g.:

```
<VARIABLES>
  <V ID="0"> p </V>
  <V ID="1"> u </V>
  <V ID="2"> v </V>
  <V ID="3"> w </V>
</VARIABLES>
```

The LEE use a conservative formulation and introduce the additional density perturbation:

```
<VARIABLES>
  <V ID="0"> p    </V>
  <V ID="1"> rho  </V>
  <V ID="2"> rhou </V>
  <V ID="3"> rhov </V>
  <V ID="4"> rhow </V>
</VARIABLES>
```

6.3.4 Functions

- `BaseFlow` Baseflow ($\overline{\rho}$, $\overline{c^2}$, $\overline{\boldsymbol{u}}$) defined by the variables `rho0, c0sq, u0, v0, w0` for APE and ($\overline{\rho}$, $\overline{c^2}$, $\overline{\boldsymbol{u}}$, γ) defined by `rho0, c0sq, u0, v0, w0, gamma` for LEE.
- `InitialConditions`

6.3.5 Boundary Conditions

In addition to plain Dirichlet and Neumann boundary conditions, the AcousticSolver features a slip-wall boundary condition, a non-reflecting boundary and a white noise boundary condition.

- Rigid (Slip-) Wall Boundary Condition, e.g. for APE:

  ```
  <BOUNDARYCONDITIONS>
  <REGION REF="0">
      <D VAR="p" USERDEFINEDTYPE="Wall" VALUE="0" />
      <D VAR="u" USERDEFINEDTYPE="Wall" VALUE="0" />
      <D VAR="v" USERDEFINEDTYPE="Wall" VALUE="0" />
      <D VAR="w" USERDEFINEDTYPE="Wall" VALUE="0" />
  </REGION>
  </BOUNDARYCONDITIONS>
  ```

  This BC imposes zero wall-normal perturbation velocity in a way that is more robust than using a Dirichlet boundary condition directly.

- Non-Reflecting Boundary Condition, e.g. for APE:

  ```
  <BOUNDARYCONDITIONS>
  <REGION REF="0">
      <D VAR="p" USERDEFINEDTYPE="RiemannInvariantBC"/>
      <D VAR="u" USERDEFINEDTYPE="RiemannInvariantBC"/>
      <D VAR="v" USERDEFINEDTYPE="RiemannInvariantBC"/>
      <D VAR="w" USERDEFINEDTYPE="RiemannInvariantBC"/>
  </REGION>
  </BOUNDARYCONDITIONS>
  ```

  The Riemann-Invariant BC approximates a non-reflecting (r.g. Farfield) boundary condition by setting incoming invariants to zero.

- White Noise Boundary Condition, e.g. for APE:

```
<BOUNDARYCONDITIONS>
<REGION REF="0">
    <D VAR="p" USERDEFINEDTYPE="Wall" VALUE="10" />
    <D VAR="u" USERDEFINEDTYPE="Wall" VALUE="10" />
    <D VAR="v" USERDEFINEDTYPE="Wall" VALUE="10" />
    <D VAR="w" USERDEFINEDTYPE="Wall" VALUE="10" />
</REGION>
</BOUNDARYCONDITIONS>
```

  The white noise BC imposes a stochastic, uniform pressure at the boundary. The implementation uses a Mersenne-Twister pseudo random number generator to generate white Gaussian noise. The standard deviation σ of the pressure is specified by the `VALUE` attribute.

6.4 Examples

6.4.1 Wave Propagation in a Sheared Base Flow

In this section we explain how to set up a simple, 2D simulation of aeroacoustics in Nektar++. We will study the propagation of an acoustic wave in the simple case of a sheared base flow, i.e. $\overline{\boldsymbol{u}} = [300\tanh(20x_2),\, 0]^{\mathrm{T}}$, $\overline{c^2} = (341\,\mathrm{m/s})^2$, $\overline{\rho} = 1.204\,\mathrm{kg/m^3}$. The geometry consists of 64 quadrilateral elements.

6.4.1.1 Input file

We require a discontinuous Galerkin projection and use an explicit fourth-order Runge-Kutta time integration scheme. We therefore set the following time integration schem and solver information:

```
<TIMEINTEGRATIONSCHEME>
    <METHOD> RungeKutta </METHOD>
    <ORDER> 4 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="EQType"                VALUE="APE"/>
    <I PROPERTY="Projection"            VALUE="DisContinuous"/>
    <I PROPERTY="UpwindType"            VALUE="LaxFriedrichs"/>
</SOLVERINFO>
```

To maintain numerical stability we must use a small time-step. Finally, we set the density, heat ratio and ambient pressure.

```
<PARAMETERS>
    <P> TimeStep       = 1e-05              </P>
    <P> NumSteps       = 1000               </P>
    <P> FinTime        = TimeStep*NumSteps  </P>
    <P> IO_CheckSteps  = 10                 </P>
    <P> IO_InfoSteps   = 10                 </P>
</PARAMETERS>
```

The initial condition and the base flow field are specified by the `Baseflow` and `InitialConditions` functions, respectively:

```
<FUNCTION NAME="Baseflow">
  <E VAR="u0" VALUE="300 * tanh(2*y/0.1)"/>
  <E VAR="v0" VALUE="0"/>
  <E VAR="c0sq" VALUE="1.4 * Pinfinity / Rho0"/>
  <E VAR="rho0" VALUE="Rho0"/>
</FUNCTION>
<FUNCTION NAME="InitialConditions">
  <E VAR="p" VALUE="0"/>
  <E VAR="u" VALUE="0"/>
  <E VAR="v" VALUE="0"/>
</FUNCTION>
```

At all four boundaries the `RiemannInvariantBC` condition is used:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="p" USERDEFINEDTYPE="RiemannInvariantBC"/>
    <D VAR="u" USERDEFINEDTYPE="RiemannInvariantBC"/>
    <D VAR="v" USERDEFINEDTYPE="RiemannInvariantBC"/>
  </REGION>
</BOUNDARYCONDITIONS>
```

The system is excited via an acoustic source term $\dot{\omega}_c$, which is modeled by a field forcing as:

```
<FORCING>
  <FORCE TYPE="Field">
    <FIELDFORCE> Source <FIELDFORCE/>
  </FORCE>
</FORCING>
```

and the corresponding function

```
<FUNCTION NAME="Source">
  <E VAR="p" VALUE="100 * 2*PI*5E2 * cos(2*PI*5E2 * t) * exp(-32*(x^2+y^2))"/>
  <E VAR="u" VALUE="0"/>
  <E VAR="v" VALUE="0"/>
</FUNCTION>
```

6.4.1.2 Running the code

```
AcousticSolver Test_pulse.xml
```

6.4.1.3 Results

Fig. 6.1 shows the acoustic source term, the velocity and the acoustic pressure and velocity perturbations at a single time step.

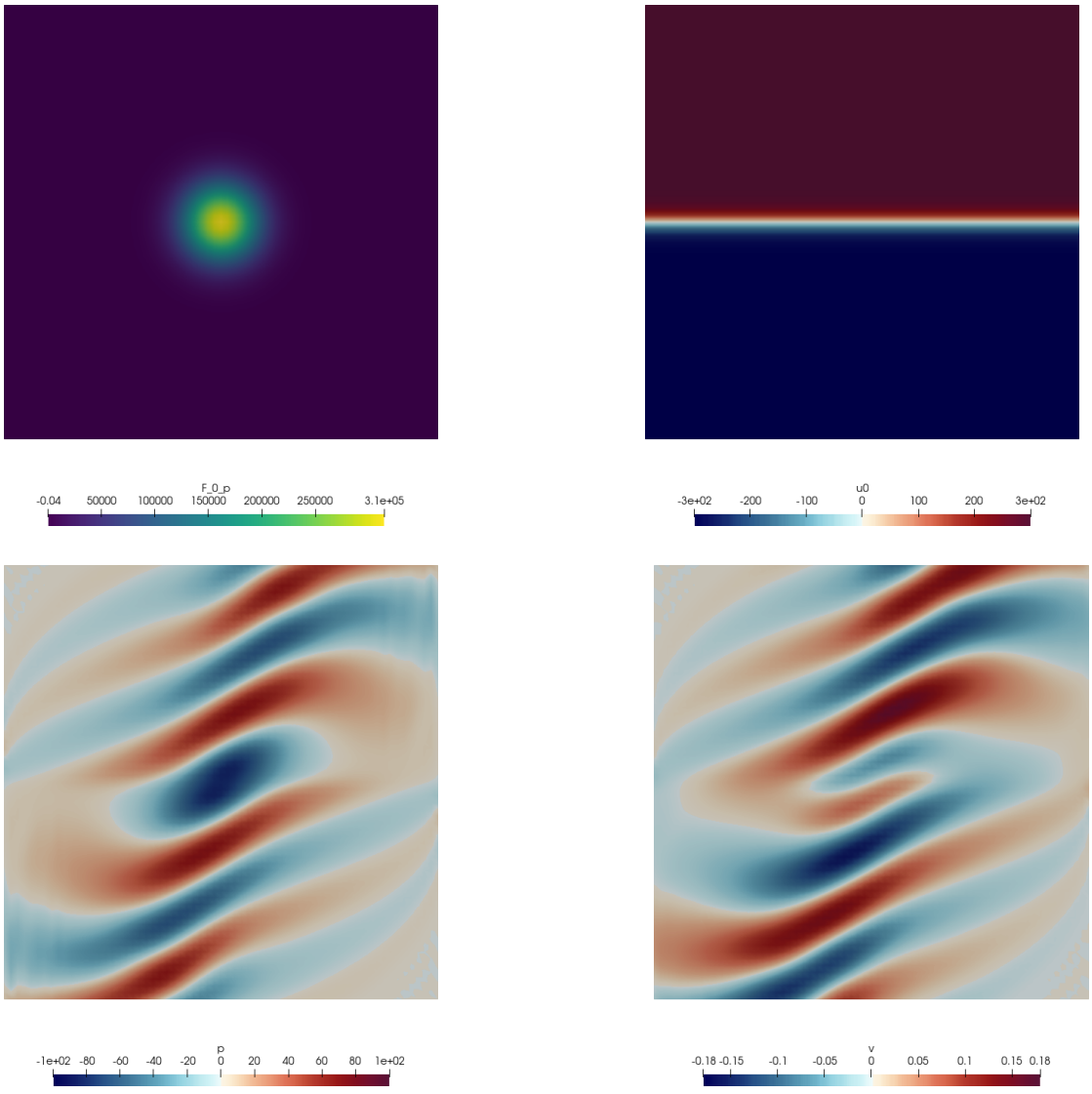


Figure 6.1 Acoustic source term, base flow velocity, acoustic pressure and acoustic velocity perturbations.

Chapter 7

Advection-Diffusion-Reaction Solver

7.1 Synopsis

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

$$\alpha\frac{\partial u}{\partial t} + \lambda u + \mathbf{V}\cdot\nabla u + f = \epsilon\nabla\cdot(D\nabla u) + R(u) \tag{7.1}$$

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
$u = f$	`Projection`	All	Continuous/Discontinuous
$\nabla^2 u = 0$	`Laplace`	All	Continuous/Discontinuous
$\nabla^2 u = f$	`Poisson`	All	Continuous/Discontinuous
$\nabla^2 u - \lambda u = f$	`Helmholtz`	All	Continuous/Discontinuous
$\epsilon\nabla^2 u - \mathbf{V}\cdot\nabla u = f$	`SteadyAdvectionDiffusion`	2D only	Continuous/Discontinuous
$\epsilon\nabla^2 u - \mathbf{V}\cdot\nabla u + \lambda u = f$	`SteadyAdvectionDiffusionReaction`	2D only	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + \mathbf{V}\cdot\nabla u = 0$	`UnsteadyAdvection`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + \mathbf{V}\cdot\nabla u = \epsilon\nabla^2 u$	`UnsteadyAdvectionDiffusion`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} = \epsilon\nabla\cdot(D\nabla u)$	`UnsteadyDiffusion`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} = \epsilon\nabla\cdot(D\nabla u) + R(u)$	`UnsteadyReactionDiffusion`	All	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + u\nabla u = 0$	`UnsteadyInviscidBurgers`	1D only	Continuous/Discontinuous
$\frac{\partial u}{\partial t} + u\nabla u = \epsilon\nabla^2 u$	`UnsteadyViscousBurgers`	1D only	Continuous/Discontinuous

Table 7.1 Equations supported by the ADRSolver with their capabilities.

7.2 Usage

```
ADRSolver session.xml
```

7.3 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 7.1. At present, the parallel run for steady solvers is not supported

7.3.1 Time Integration Scheme

- **TimeIntegrationScheme**: The following types of time integration schemes have been tested with each solver:

EqType	Explicit	Diagonally Implicit	IMEX	Implicit
`UnsteadyAdvection`	✓			
`UnsteadyAdvectionDiffusion`	✓		✓	
`UnsteadyDiffusion`	✓	✓		
`UnsteadyReactionDiffusion`	✓		✓	
`UnsteadyInviscidBurgers`	✓			
`UnsteadyViscousBurgers`	✓		✓	

7.3.2 Solver Info

The solver info are listed below:

- **Eqtype**: This sets the type of equation to solve, according to the table above.
- **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` (The penalty term is proportional to an optional parameter `LDGc11` which is by default set to one; proportionality to polynomial order can be manually imposed by setting the parameter `LDGc11` equal to p^2).

- **UpwindType**:
  - `Upwind`.

7.3.3 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: SteadyAdvectionDiffusion, SteadyAdvectionDiffusionReaction, UnsteadyDiffusion, UnsteadyReactionDiffusion, UnsteadyAdvectionDiffusion, and UnsteadyViscousBurgers.
  Default value: 1.

- `d00`, `d11`, `d22`: sets the diagonal entries of the diffusion tensor D.
  Can be used in: UnsteadyDiffusion and UnsteadyReactionDiffusion
  Default value: All set to 1 (i.e. identity matrix).

- `lambda`: sets the reaction coefficient λ.
  Can be used in: Helmholtz and SteadyAdvectionDiffusionReaction
  Default value: 0.

7.3.4 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.

7.4 Examples

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

7.4.1 Projection: 2D Problem

to be completed

7.4.2 Helmholtz: Two-dimensional octagonal plane

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

7.4.2.1 Helmholtz equation

We consider the elliptic partial differential equation:

$$\nabla^2 u - \lambda u = f \tag{7.2}$$

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

7.4.2.2 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>
```

7.4.2.3 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.

7.4.2.4 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. 7.1

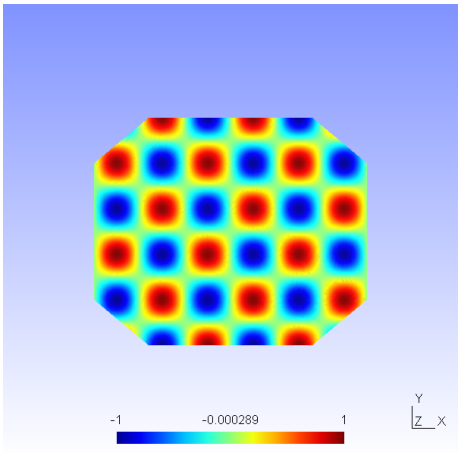


Figure 7.1 Solution of the 2D Helmholtz Problem.

7.4.3 SteadyAdvectionDiffusion: 2D Problem

to be completed

7.4.4 SteadyAdvectionDiffusionReaction: 2D Problem

to be completed

7.4.5 UnsteadyAdvection: 1D Advection equation

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

7.4.5.1 Advection equation

We consider the hyperbolic partial differential equation:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \tag{7.3}$$

where a is the advection velocity.

7.4.5.2 Input file

The input for this example is given in the example file `Advection1D_WeakDG_GLL_LAGRANGE.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.

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> RungeKutta </METHOD>
  <ORDER> 4 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvection"    />
    <I PROPERTY="Projection"            VALUE="DisContinuous"        />
    <I PROPERTY="AdvectionType"         VALUE="WeakDG"                 />
    <I PROPERTY="UpwindType"            VALUE="Upwind"               />
</SOLVERINFO>
```

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>
```

7.4.5.3 Running the code

```
ADRSolver Advection1D_WeakDG_GLL_LAGRANGE.xml
```

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

```
FieldConvert Advection1D_WeakDG_GLL_LAGRANGE.xml
    Advection1D_WeakDG_GLL_LAGRANGE.fld Advection1D_WeakDG_GLL_LAGRANGE.dat
FieldConvert Advection1D_WeakDG_GLL_LAGRANGE.xml
    Advection1D_WeakDG_GLL_LAGRANGE.fld Advection1D_WeakDG_GLL_LAGRANGE.vtu
```

7.4.6 UnsteadyAdvectionDiffusion: 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 $O(10^{-10})$.

7.4.6.1 Background

The governing equation for modelling mass transport is the unsteady advection diffusion equation:

$$\frac{\partial u}{\partial t} + a\nabla u + \epsilon\nabla^2 u = 0$$

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.

7.4.6.2 Input file

The geometry under consideration is a pipe of radius, $R = 0.5$ and length $l = 0.5$

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:

```
<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>
```

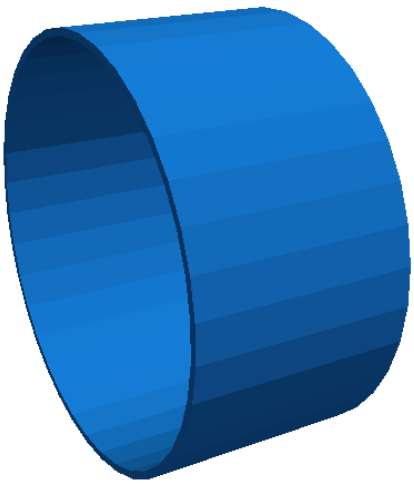


Figure 7.2 Pipe.

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.

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> IMEX </METHOD>
  <ORDER> 1 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="EQTYPE"                VALUE="UnsteadyAdvectionDiffusion" />
    <I PROPERTY="Projection"            VALUE="Continuous"                 />
    <I PROPERTY="DiffusionAdvancement"  VALUE="Implicit"                   />
    <I PROPERTY="AdvectionAdvancement"  VALUE="Explicit"                   />
    <I PROPERTY="GlobalSysSoln"         VALUE="IterativeStaticCond"        />
</SOLVERINFO>
```

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>
```

7.4.6.3 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. 7.3.

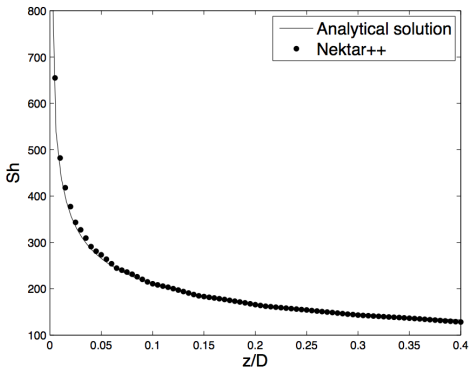


Figure 7.3 Concentration gradient at the surface of the pipe.

7.4.7 UnsteadyDiffusion: 2D Anisotropic diffusion problem

7.4.7.1 Unsteady diffusion equation

The ADRSolver supports the solution of a single-variable system

$$\frac{\partial u}{\partial t} = \epsilon \nabla \cdot (D \nabla u)$$

where the diffusion coefficient ϵ and the diagonal components of the anisotropic diffusion tensor D are defined using the session file.

7.4.7.2 Input file

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

The two-dimensional geometry for this problem contains 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"> T[0-3] </C>
    <C ID="1"> Q[4-5] </C>
    <C ID="2"> E[2,7,4,9] </C>
    <C ID="3"> E[0,1,10,11] </C>
</COMPOSITE>

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

For both the triangular and quadrilateral elements, we use the modified Legendre basis with 12 modes (maximum polynomial order is 11).

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="12" FIELDS="u" TYPE="MODIFIED" />
    <E COMPOSITE="C[1]" NUMMODES="12" FIELDS="u" TYPE="MODIFIED" />
</EXPANSIONS>
```

The Continuous Galerkin Method is used as projection scheme along with a third-order DIRK scheme to solve the diffusion equation, so we need to specify the following time-integration scheme and solver information.

```
<TIMEINTEGRATIONSCHEME>
    <METHOD> DIRK </METHOD>
    <ORDER> 3 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="EQTYPE" VALUE="UnsteadyDiffusion" />
    <I PROPERTY="Projection" VALUE="Continuous" />
    <I PROPERTY="DiffusionAdvancement" VALUE="Implicit" />
</SOLVERINFO>
```

For information, while the ADRSolver support both Continuous Galerkin and DisContinuous Galerkin projections with implicit time-integration, only the DisContinuous Galerkin projection is supported when using an explicit time-integration method for the unsteady diffusion problem.

We solve this unsteady problem by considering 200 time units with a time-step of 0.0001 along with an anisotropic diffusion coefficients `d00` and `d11` of 0.5 and 2.0, respectively. Please note that if one only desires to consider isotropic diffusion, the parameters `d00` and `d11` are not required.

```
<PARAMETERS>
    <P> TimeStep       = 0.0001           </P>
    <P> NumSteps       = 200              </P>
    <P> FinTime        = TimeStep*NumSteps</P>
    <P> wavefreq       = PI               </P>
    <P> epsilon        = 1.0              </P>
    <P> d00            = 0.5              </P>
    <P> d11            = 2.0              </P>
</PARAMETERS>
```

Both Dirichlet and Neumann boundary conditions are used to specify this problem.

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

<BOUNDARYCONDITIONS>
    <REGION REF="0">
        <D VAR="u" VALUE="0" />
    </REGION>
    <REGION REF="1">
```

```
        <N VAR="u" VALUE="0" />
    </REGION>
</BOUNDARYCONDITIONS
```

For this problem, we use the following initial condition

```
<FUNCTION NAME="InitialConditions">
    <E VAR="u" VALUE="sin(wavefreq*x)*cos(wavefreq*y)" />
</FUNCTION>
```

7.4.7.3 Running the code

```
ADRSolver ImDiffusion_VarCoeff.xml
```

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

```
FieldConvert ImDiffusion_VarCoeff.xml ImDiffusion_VarCoeff.fld
    ImDiffusion_VarCoeff.dat
FieldConvert ImDiffusion_VarCoeff.xml ImDiffusion_VarCoeff.fld
    ImDiffusion_VarCoeff.vtu
```

7.4.8 UnsteadyReactionDiffusion: Unsteady reaction-diffusion systems

7.4.8.1 Unsteady reaction-diffusion equation

Reaction-diffusion systems are prevalent in a number of areas relating to the modelling of various physical phenomena, and are particularly prevalent in the study of chemical interactions and pattern formation. The ADRSolver supports the solution of a single-variable system

$$\frac{\partial u}{\partial t} = \epsilon \nabla \cdot (D \nabla u) + R(u)$$

where the diffusion coefficient ϵ, the diagonal components of the anisotropic diffusion tensor D, and reaction term $R(u)$ are defined using the session file.

7.4.8.2 Input file

The input for this example is given in the example file `ReactionDiffusion2D.xml`.

The geometry section defines a 2D domain consisting of 64 quadrilateral elements.

```
<COMPOSITE>
    <C ID="0"> Q[0-63] </C>
    <C ID="1"> E[0,25,42,59,76,93,110,127] </C>
    <C ID="2"> E[128,130,132,134,136,138,140,142] </C>
    <C ID="3"> E[23,41,58,75,92,109,126,143] </C>
    <C ID="4"> E[3,6,9,12,15,18,21,24] </C>
</COMPOSITE>
<DOMAIN> C[0] </DOMAIN>
```

For this example, we use the modified Legendre basis with 6 modes.

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="6" TYPE="MODIFIED" FIELDS="u" />
</EXPANSIONS>
```

We choose to solve our solution for 100 time units with a time-step of 0.001 and a diffusion coefficient `epsilon` of 0.2

```
<P> TimeStep       = 0.001            </P>
<P> NumSteps       = 100              </P>
<P> epsilon        = 0.2              </P>
```

The following `TIMEINTEGRATIONSCHEME` and `SOLVERINFO` configuration is used:

```
<TIMEINTEGRATIONSCHEME>
    <METHOD> IMEX </METHOD>
    <ORDER> 3 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="EQTYPE"                VALUE="UnsteadyReactionDiffusion" />
    <I PROPERTY="Projection"            VALUE="Continuous"                />
    <I PROPERTY="DiffusionAdvancement"  VALUE="Implicit"                  />
</SOLVERINFO>
```

Periodic boundary conditions are used

```
<BOUNDARYCONDITIONS>
    <REGION REF="0">
         <P VAR="u" VALUE="[2]" />
    </REGION>
    <REGION REF="1">
         <P VAR="u" VALUE="[3]" />
    </REGION>
    <REGION REF="2">
         <P VAR="u" VALUE="[0]" />
    </REGION>
    <REGION REF="3">
         <P VAR="u" VALUE="[1]" />
    </REGION>
</BOUNDARYCONDITIONS>
```

Further to this, the reaction term $R(u)$ is imposed by the definition of a body forcing function by first specifying an appropriate `FORCING` block (as described in section 3.6):

```
<FORCING>
    <FORCE TYPE="Body">
        <BODYFORCE> BodyForce </BODYFORCE>
    </FORCE>
</FORCING>
```

The reaction term can then be defined using the function:

```
<!-- Body force to enforce reaction term -->
<FUNCTION NAME="BodyForce">
    <E VAR="u" EVARS="u" VALUE="0.1*u" />
</FUNCTION>
```

Note in particular the use of the `EVARS` (equation variables) attribute, which permits the definition of this function in terms of the scalar variable u.

7.4.8.3 Running the code

```
ADRSolver ReactionDiffusion2D.xml
```

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

```
FieldConvert ReactionDiffusion2D.xml ReactionDiffusion2D.fld
    ReactionDiffusion2D.dat
FieldConvert ReactionDiffusion2D.xml ReactionDiffusion2D.fld
    ReactionDiffusion2D.vtu
```

7.4.9 UnsteadyInviscidBurgers: 2D Problem

to be completed

7.4.10 UnsteadyViscousBurgers: 2D Problem

to be completed

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	[28]
`FitzHughNagumo`			
`Fox02`			
`LuoRudy91`	Mammalian ventricular	7	[26]
`PanditGilesDemir03`			
`TenTusscher06`	Human ventricular	18	[47]
`Winslow99`			

- `Projection` Specifies the Galerkin projection type to use. Only `Continuous` has been extensively tested.
- `TimeIntegrationScheme` Specifies the time integration scheme to use for advancing the PDE system. This must be an IMEX scheme. Suitable choices are: `IMEX Order 1,2,3`, `IMEX, Variant dirk, Order 3, Free Parameters 3 4`. 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 [13].

- `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 [13].

- `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

The following filters are supported exclusively for the cardiac EP solver. Further filters from section 3.5 are also available for this solver.

- `Benchmark` (section 3.5.3)
- `CellHistoryPoints` (section 3.5.4)
- `CheckpointCellModel` (section 3.5.5)
- `Electrogram` (section 3.5.7)

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 μA/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 can be achieved by using local discontinuous Galerkin (LDG) approach, interior penalty method or 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 [31].

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> GasConstant         = 287.058                   </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>
  <P> IO_Timer_Level      = 3                         </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. No checkpoint file is written if it is set to 0.
- `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.
- `GasConstant` universal gas contant. Default value = 287.058 $JKg^{-1}K^{-1}$.
- `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.
- `uInf` 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_∞). This can be set as an alternative to `Pr`, in which case the Prandtl number is calculated from κ_∞ (it is only possible to set one of them). By default, this is obtained from the Prandtl number.
- `CFL` is the CFL number (explicit and implicit solvers).
- `CFLGrowth` is the growing CFL (explicit and implicit solvers).
- `CFLEnd` is the maximum value of the CFL number (explicit and implicit solvers).
- `Timer_IO_Level` defines the amount of timer information that is printed after the solver is finished. The default value is -1, which disables output. By selecting a value between 0 and 2, more detailed timer information is printed.

Time Integration Scheme

Under this section it is possible to set the time integration scheme information.

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> RungeKutta </METHOD>
  <VARIANT> SSP </VARIANT>
  <ORDER> 3 </ORDER>
</TIMEINTEGRATIONSCHEME>
```

- `TimeIntegrationScheme` is the time-integration scheme we want to use. There are implicit and explicit schemes for the Compressible flow solver. For an explicit discretization, the time-integration schemes supported are as follows

Name	<METHOD>	<VARIANT>	<ORDER>
Forward Euler	ForwardEuler	-	1
Runge Kutta 2 - SSP	RungeKutta	SSP	2
Runge Kutta 3 - SSP	RungeKutta	SSP	3
Runge Kutta 4	ClassicalRungeKutta	-	4
Runge Kutta 5	RungeKutta	-	5

For an implicit discretization, the time-integration schemes available are

Name	<METHOD>	<VARIANT>	<ORDER>
Backward Euler	BackwardEuler	-	1
Backward Differentiation Formula Implicit	BDFImplicit	-	1
Backward Differentiation Formula Implicit	BDFImplicit	-	2
Singly Diagonally Implicit Runge Kutta	DIRK	-	2
Singly Diagonally Implicit Runge Kutta	DIRK	-	3
Singly Diagonally Implicit Runge Kutta	DIRK	ES5	3
Singly Diagonally Implicit Runge Kutta	DIRK	-	4
Singly Diagonally Implicit Runge Kutta	DIRK	ES5	4

Solver info

Under this section it is possible to set the solver information.

```
<SOLVERINFO>
  <I PROPERTY="EQTYPE"                    VALUE="NavierStokesImplicitCFE" />
  <I PROPERTY="Projection"                VALUE="DisContinuous"           />
  <I PROPERTY="TimeIntegrationMethod"     VALUE="DIRKOrder2"              />
  <I PROPERTY="AdvectioType"              VALUE="WeakDG"                  />
  <I PROPERTY="DiffusionType"             VALUE="InteriorPenalty"         />
  <I PROPERTY="UpwindType"                VALUE="Roe"                     />
  <I PROPERTY="ProblemType"               VALUE="General"                 />
  <I PROPERTY="ViscosityType"             VALUE="Constant"                />
  <I PROPERTY="EquationOfState"           VALUE="IdealGas"                />
  <I PROPERTY="Driver"                    VALUE="Standard"                />
</SOLVERINFO>
```

- `EQType` is the tag which specify the equations we want solve:
  Explicit discretization in time:
  - `NavierStokesCFE` (Compressible Navier-Stokes equations).
  - `EulerCFE` (Compressible Euler equations).

  Implicit discretization in time:
  - `NavierStokesImplicitCFE` (Compressible Navier-Stokes equations).
  - `EulerImplicitCFE` (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 $c = \infty$).

  Note that only `WeakDG` is fully supported, the other operators work only with quadrilateral elements ($2D$ or $2.5D$).

- `DiffusionType` is the diffusion operator we want to use for the Navier-Stokes equations:

  - `LDGNS` (LDG with primitive variables. The penalty term is inversely proportional to the element size, proportional to the local viscosity for the momentum equations and to the thermal conductivity for the energy equation, and proportional to an optional parameter `LDGNSc11` which is by default set to one; proportionality to polynomial order can be manually imposed by setting the parameter `LDGNSc11` equal to p^2).
  - `LFRDGNS` (Flux-Reconstruction recovering nodal DG scheme).
  - `LFRSDNS` (Flux-Reconstruction recovering a spectral difference (SD) scheme).
  - `LFRHUNS` (Flux-Reconstruction recovering Huynh (G2) scheme).
  - `LFRcminNS` (Flux-Reconstruction with $c = c_{min}$).
  - `LFRcinfNS` (Flux-Reconstruction with $c = \infty$).
  - `InteriorPenalty` (Symmetric interior penalty method).

  Note that only `LDGNS` and `InteriorPenalty` are fully supported, the other operators work only with quadrilateral elements ($2D$ or $2.5D$).

- `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`.

- `ViscosityType` is the viscosity type we want to use:
  - `Constant` (Constant viscosity).
  - `Variable` (Variable viscosity through the Sutherland's law).

- `EquationOfState` allows selecting an equation of state for accounting for non-ideal gas behaviour:
  - `IdealGas` (default option).
  - `VanDerWaals` (requires additional parameters `Tcrit` and `Pcrit`).
  - `RedlichKwong` (requires additional parameters `Tcrit` and `Pcrit`).
  - `PengRobinson` (requires additional parameters `Tcrit`, `Pcrit` and `AcentricFactor`).

- `Driver` specifies the type of problem to be solved:
  - `Standard` (default option to solve the unsteady equations).
  - `SteadyState` (uses the Selective Frequency Damping method (see Sec. 11.1.4) to obtain a steady-state solution of the Navier-Stokes equations (explicit or implicit)).

- `ShockCaptureType` specifies the type of operator to be used for shock capturing:
  - `NonSmooth` add a Laplacian operator to apply artificial diffusion (see Sec. 9.4.1.1). This option is only supported for explicit solvers (`EulerCFE` and `NavierStokesCFE`).
  - `Physical` add artificial viscosity to the physical viscosity. This option is only supported for Navier-Stokes solvers (`NavierStokesCFE` and `NavierStokesImplicitCFE`).

- `ShockSensorType` specifies the sensor type of shock capturing to be used:
  - `Modal` (default) use a modal sensor to identify where to add viscosity (see Sec. 9.4.1.2).
  - `Dilatation` use a dilatation sensor to identify where to add viscosity.

- `DucrosSensor` apply a Ducros [12] (anti-vorticity) filter to the shock sensor:
  - `On`
  - `Off`

- `Smoothing` apply a smoothing filter to the shock sensor:
  - `C0` smooth the artificial viscosity to be a continuous field.

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="2"> 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="3"> 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`. Note that the no-slip, isothermal, wall boundary condition requires Twall to be specified. In the following are 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>
```

  In some cases we need to excite perturbations inside the boundary layer. This can be achieved by setting part of the wall as a disturbance strip. In the no-slip/adiabatic wall boundary conditions, if the `VALUE` is not exact "0" but any expression, which can be time-dependent, the value of the expression will be added to the ghost state of what it should be in the input boundary conditions, and then generate a non-zero flux through the Riemann solver. The following is an example to set a disturbance strip of amplitude A, frequency f, and in the range of $[x_0, x_0 + len]$ on a flat plate.

```
  <BOUNDARYCONDITIONS>
    <REGION REF="0">
      <D VAR="rho"  USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
      <D VAR="rhou" USERDEFINEDTYPE="WallAdiabatic" VALUE="0" />
      <D VAR="rhov" USERDEFINEDTYPE="WallAdiabatic"
                    VALUE="A*sin(2*PI*f*t)*sin(2*PI*(x-x0)/len)
                           *(x>=x0)*(x<=(x0+len))"           />
      <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)"/>
  </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="pOut" />
  </REGION>
</BOUNDARYCONDITIONS>
```

where `pOut` is the target static pressure at the boundary.

- Stagnation inflow boundary condition:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="StagnationInflow" VALUE="rho0" />
    <D VAR="rhou" USERDEFINEDTYPE="StagnationInflow" VALUE="nx" />
    <D VAR="rhov" USERDEFINEDTYPE="StagnationInflow" VALUE="ny" />
    <D VAR="rhov" USERDEFINEDTYPE="StagnationInflow" VALUE="nz" />
    <D VAR="E"    USERDEFINEDTYPE="StagnationInflow" VALUE="p0/(Gamma-1)" />
  </REGION>
</BOUNDARYCONDITIONS>
```

where `rho0` and `p0` are the stagnation density and stagnation pressure, respectively. The values `nx`, `ny`, and `nz` further define the direction of the flow. Note that it is not necessary to normalize the vector `(nx, ny, nz)`, it will be done automatically by the solver. If the flow direction isn't specified, the flow will enter in the boundary-normal direction. The stagnation inflow boundary condition imposes a constant stagnation density and stagnation pressure on the boundary. These quantities are related to the stagnation temperature through the relation `T0 = p0/(rho0 * GasConstant)`. This boundary condition is useful in situations when stagnation quantities are known at the inlet, rather than the actual density and pressure. In addition to this, the standard boundary condition where 5 quantities (the conservative variables) are imposed on the boundary is not suitable in many situations because in a subsonic compressible flow, only 4 out of 5 variables can be imposed at an inlet boundary. In contrast, the stagnation inflow boundary condition only imposes 4 quantities (2 stagnation quantities, and the direction of the flow). The last quantitity at the inlet will instead be determined by the solution inside the domain. If you have a good estimate of one additional quantitity at the inlet, such as the magnitude of the velocity or the static pressure, you can estimate all remaining quantities at the inlet for the isentropic compressible flow relations. In fact, at each iteration, the solver uses the isentropic flow relations combined with the quantities imposed by the boundary + one quantity from the interior solution to compute all remaining quantities at the boundary. Once this has been done, the flux over the boundary at that iteration can be computed.

- Subsonic boundary condition enforcing entropy and pressure:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
```

```
    <D VAR="rho"  USERDEFINEDTYPE="EnforceEntropyPressure" VALUE="rhoInf" />
    <D VAR="rhou" USERDEFINEDTYPE="EnforceEntropyPressure" VALUE="rhoInf*uInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyPressure" VALUE="rhoInf*vInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyPressure" VALUE="rhoInf*wInf" />
    <D VAR="E"    USERDEFINEDTYPE="EnforceEntropyPressure" VALUE="pInf/(Gamma-1) + 0.5*rhoInf*(uInf*uInf + vInf*vInf)" />
  </REGION>
</BOUNDARYCONDITIONS>
```

where the input type can be either `VALUE` or `FILE` such that the specified entropy and pressure distribution (corresponding to the input `rhoInf`, `rhoInf*uInf`, `rhoInf*vInf`, `rhoInf*wInf`, and `E`) will be enforced on the subsonic inflow boundary. If the flow is supersonic, all the conditions are naturelly agreed with the input distribution. The following two conditions follows the same rule.

- Subsonic boundary condition enforcing entropy and velocity:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="EnforceEntropyVelocity" VALUE="rhoInf" />
    <D VAR="rhou" USERDEFINEDTYPE="EnforceEntropyVelocity" VALUE="rhoInf*uInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyVelocity" VALUE="rhoInf*vInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyVelocity" VALUE="rhoInf*wInf" />
    <D VAR="E"    USERDEFINEDTYPE="EnforceEntropyVelocity" VALUE="pInf/(Gamma-1) + 0.5*rhoInf*(uInf*uInf + vInf*vInf)" />
  </REGION>
</BOUNDARYCONDITIONS>
```

where both tangential and normal velocity components are enforced. Therefore the angle of attack is maintained.

- Subsonic boundary condition enforcing entropy and total enthalpy:

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="rho"  USERDEFINEDTYPE="EnforceEntropyTotalEnthalpy" VALUE="rhoInf" />
    <D VAR="rhou" USERDEFINEDTYPE="EnforceEntropyTotalEnthalpy" VALUE="rhoInf*uInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyTotalEnthalpy" VALUE="rhoInf*vInf" />
    <D VAR="rhov" USERDEFINEDTYPE="EnforceEntropyTotalEnthalpy" VALUE="rhoInf*wInf" />
    <D VAR="E"    USERDEFINEDTYPE="EnforceEntropyTotalEnthalpy" VALUE="pInf/(Gamma-1) + 0.5*rhoInf*(uInf*uInf + vInf*vInf)" />
  </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)"/>
</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)"/>
</FUNCTION>
```

9.4 Examples

9.4.1 Shock capturing

Compressible flows can be characterised by abrupt changes in flow variables within the flow domain often referred to as shocks. These discontinuities can lead to numerical instabilities (Gibbs phenomena). This problem is prevented by locally adding a diffusion term to the equations to damp the numerical oscillations.

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 [39]. The diffusivity of the system is controlled by a variable viscosity coefficient ε. For consistency ε is proportional to the element size and inversely proportional to the polynomial order. Finally, from physical considerations ε needs to be proportional to the maximum characteristic speed of the problem. The final form of the artificial viscosity is

$$\varepsilon = \varepsilon_0 \frac{h}{p} \lambda_{max} S, \tag{9.8}$$

where S is a sensor.

To enable the non-smooth viscosity model, the following line has to be added to the `SOLVERINFO` section:

```
<SOLVERINFO>
  <I PROPERTY="ShockCaptureType"      VALUE="NonSmooth" />
<SOLVERINFO>
```

9.4.1.2 Modal sensor

As shock sensor, a modal resolution-based indicator is used

$$s_e = log_{10}\left(\frac{\langle q-\tilde{q}, q-\tilde{q}\rangle}{\langle q, q\rangle}\right), \tag{9.9}$$

where $\langle\cdot,\cdot\rangle$ represents a L^2 inner product, q and $\tilde{q}$ are the full and truncated expansions of a state variable (in our case density)

$$q(x) = \sum_{i=1}^{N(P)} \hat{q}_i\phi_i, \quad \tilde{q}(x) = \sum_{i=1}^{N(P-1)} \hat{q}_i\phi_i, \tag{9.10}$$

then the constant element-wise sensor is computed as follows

$$S_\varepsilon = \begin{cases} 0 & \text{if} \quad s_e < s_0 - \kappa \\ \frac{1}{2}\left(1+\sin\frac{\pi(s_e-s_0)}{2\kappa}\right) & \text{if} \quad |s_e - s_0| \leq \kappa \\ 1 & \text{if} \quad s_e > s_0 + \kappa \end{cases}, \tag{9.11}$$

where $s_0 = s_\kappa - 4.25\ log_{10}(p)$.

The modal sensor is enabled by default and it can be explicilty set adding the following line to the `SOLVERINFO` section:

```
<SOLVERINFO>
  <I PROPERTY="ShockSensorType"       VALUE="Modal" />
<SOLVERINFO>
```

The diffusivity and the sensor can be controlled by the following parameters:

```
<PARAMETERS>
<P> Skappa              = -1.3                  </P>
<P> Kappa               = 0.2                   </P>
<P> mu0                 = 1.0                   </P>
</PARAMETERS>
```

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 coupling between different elements are determined by common numerical fluxes and there is no 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

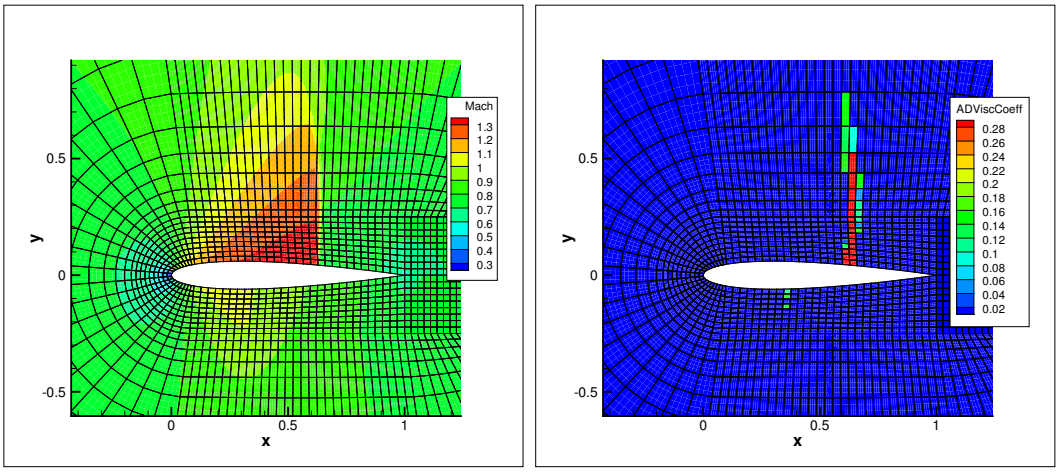


Figure 9.1 (a) Steady state solution for $M = 0.8$ flow at $\alpha = 1.25°$ past a NACA 0012 profile, (b) Artificial viscosity (ε) distribution

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.

$$p_e = \begin{cases} p_e - 1 & \text{if } \; s_e > s_{ds} \\ p_e + 1 & \text{if } \; s_{sm} < s_e < s_{ds} \\ p_e & \text{if } \; s_{fl} < s_e < s_{sm} \\ p_e - 1 & \text{if } \; s_e < s_{fl} \end{cases} \tag{9.12}$$

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.

9.4.3 De-Aliasing Techniques

Aliasing effects, arising as a consequence of the nonlinearity of the underlying problem, need to be address to stabilise the simulations. Aliasing appears when nonlinear quantities are calculated at an insufficient number of quadrature points. We can identify two types of nonlinearities:

- PDE nonlinearities, related to the nonlinear and quasi-linear fluxes.
- Geometrical nonlinearities, related to the deformed/curves meshes.

We consider two de-aliasing strategies based on the concept of consistent integration:

- Local dealiasing: It only targets the PDE-aliasing sources, applying a consistent integration of them locally.
- Global dealiasing: It targets both the PDE and the geometrical-aliasing sources. It requires a richer quadrature order to consistently integrate the nonlinear fluxes, the geometric factors, the mass matrix and the boundary term.

Since Nektar++ tackles separately the PDE and geometric aliasing during the projection and solution of the equations, to consistently integrate all the nonlinearities in the compressible NavierStokes equations, the quadrature points should be selected based on the maximum order of the nonlinearities:

$$Q_{min} = P_{exp} + \frac{max(2P_{exp}, P_{geom})}{2} + \frac{3}{2} \tag{9.13}$$

where Q_{min} is the minimum required number of quadrature points to exactly integrate the highest-degree of nonlinearity, P_{exp} being the order of the polynomial expansion and P_{geom} being the geometric order of the mesh. Bear in mind that we are using a discontinuous discretisation, meaning that aliasing effect are not fully controlled, since the boundary terms introduce non-polynomial functions into the problem.

To enable the global de-aliasing technique, modify the number of quadrature points by:

```
<E COMPOSITE="[101]"
   BASISTYPE="Modified_A,Modified_A"
   NUMMODES="7,7"
   POINTSTYPE="GaussLobattoLegendre,GaussLobattoLegendre"
   NUMPOINTS="14,14"
   FIELDS="rho,rhou,rhov,E"
/>
```

where `NUMMODES` corresponds to P+1, where P is the order of the polynomial used to approximate the solution. `NUMPOINTS` specifies the number of quadrature points.

9.4.4 Implicit solver

In this example, we solve a compressible flow past a circular cylinder using an implicit discontinuous Galerkin compressible flow solver as shown in figure 9.2. For the implicit time-integration schemes, `TimeStep` or `CFL` can be adopted to control the time step. For the case of using `CFL`, the CFL number can grow from `CFL` to `CFLEnd` by a ratio of `CFLGrowth` to adjust the time step at different stages of the simulation.

The CFL number, growing CFL number and the maximum value of the CFL number are controlled by the following parameters as also described in section 9.3

```
<PARAMETERS>
  <P> CFL           = 0.1                 </P>
  <P> CFLGrowth     = 1.1                 </P>
  <P> CFLEnd        = 2.0                 </P>
</PARAMETERS>
```

In this case, the numerical simulation starts from a CFL number of 0.1 and grows by a ratio of 1.1 up to a maximum value of CFL number of 2.0. Note that the `CFLEnd` parameter may assume higher values, depending on the strategy adopted. In addition, there is no need to define the `TimeStep` parameter, since the time step size is calculated based on the CFL number in each time step.

Since we are solving an implicit time-integration scheme, we must specify to the solver information the `EQType` which corresponds to an implicit solver. It should be noted that currently the CFS solver only supports `NavierStokesImplicitCFE` and `EulerImplicitCFE`.

```
<PARAMETERS>
  <I PROPERTY="EQTYPE"                    VALUE="NavierStokesImplicitCFE" />
  <I PROPERTY="TimeIntegrationMethod"     VALUE="DIRKOrder2"              />
</PARAMETERS>
```

There are some other parameters controlling the performance of the implicit solver. `NonlinIterTolRelativeL2` determines the convergence tolerance of the nonlinear system solver relative to the initial nonlinear system residual. `NekNonlinSysMaxIterations` is the maximum iteration number of the nonlinear system solver. `LinSysRelativeTolInNonlin` determines the convergence tolerance of linear system solver in each nonlinear iteration. `NekLinSysMaxIterations` is the maximum iteration number of the linear system solver in each nonlinear iteration. `LinSysMaxStorage` determines the maximum number of variable vector allowed to store in the linear system solver. Specifically for GMRES solver, the GMRES solver will be restarted if `NekLinSysMaxIterations` is larger than `LinSysMaxStorage` and thus `LinSysMaxStorage` determines the storage consumption of the GMRES solver. Regarding the parameters to control the preconditioners in GMRES, `PreconMatFreezNumb` specifies the number of time steps to freeze the preconditioning matrices, in other words, the preconditioning matrices will be updated based on the number of time steps provided by the user. `PreconItsStep` determines the number of preconditioning iterations to calculate the preconditioned vector. The default parameters are listed below.

```
<PARAMETERS>
  <P> NonlinIterTolRelativeL2   = 1E-3    </P>
  <P> NekNonlinSysMaxIterations = 10      </P>
  <P> LinSysRelativeTolInNonlin = 5.0E-2  </P>
  <P> NekLinSysMaxIterations    = 30      </P>
  <P> LinSysMaxStorage          = 30      </P>
  <P> PreconMatFreezNumb        = 200     </P>
  <P> PreconItsStep             = 7       </P>
</PARAMETERS>
```

Here, we choose to solve the compressible Navier-Stokes equations and use the 2nd order Singly Diagonally Implicit Runge–Kutta (SDIRK) method.

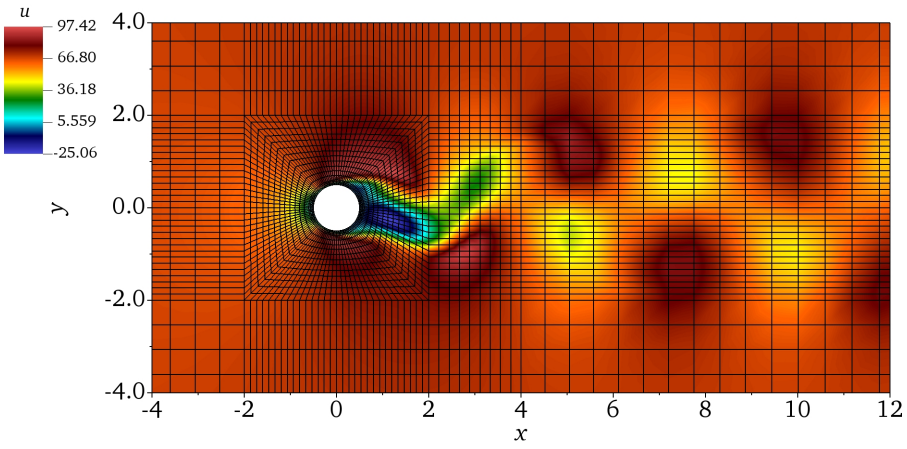


Figure 9.2 Laminar flow past a circular cylinder at $Re = 200$ and $M = 0.2$.

Chapter 10

Dummy Solver

10.1 Synopsis

The Dummy solver does not solve any equation systems but only serves to exchange fields with other solvers and applications. It is intended for demonstrating and testing the coupling implementations only.

Chapter 11

Incompressible Navier-Stokes Solver

11.1 Synopsis

A useful tool implemented in Nektar++ is the incompressible Navier Stokes solver that allows one to solve the governing equation for viscous Newtonians fluids governed by:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{11.1a}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{11.1b}$$

where $\mathbf{u}$ is the velocity, p is the specific pressure (including density) and ν the kinematic viscosity.

Various approaches to solve and analyse this set of equations are available in Nektar++ as shwon in table 11.1.

Table 11.1 An overview of the solvers available in Nektar++ for the incompressible Navier-Stokes equations.

Solver	Options
Velocity Correction Scheme	Semi-Implicit
"	Semi-Implicit w/ Weak Pressure
"	Implicit
"	Coordinate Transformation
Smoothed Profile Method	-
Linear Stability Analysis	Direct
"	Adjoint
"	Transient Growth
Steady-State solver	Selective Frequency Damping
Direct (Coupled) Solver	Stokes Problem

11.1.1 Velocity Correction Scheme

The first approach uses a splitting/projection method where the velocity system and the pressure are typically decoupled. Splitting schemes are typically favoured for their numerical efficiency since the velocity and pressure are handled independently, requiring the solution of four (in three dimensions) elliptic systems of rank N (opposed to a single system of rank $4N$ 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.

The scheme discretises the momentum equation Eq. (11.1a) with a backwards approximation of the time derivative to obtain

$$\frac{\partial \mathbf{u}}{\partial t}^{n+1} \simeq \frac{\gamma_0 \tilde{\mathbf{u}}^{n+1} - \hat{\mathbf{u}}}{\Delta t} \tag{11.2}$$

where $\tilde{\mathbf{u}}^{n+1}$ is an intermediate velocity and $\hat{\mathbf{u}}$ is the summation of previous solutions.

With the discrete time derivative, we initially have to solve a pressure Poisson equation of the form

$$\nabla^2 p^{n+1} = \nabla \cdot \left(\frac{\hat{\mathbf{u}}}{\Delta t} - \mathbf{N}^{*,n+1} \right) \tag{11.3}$$

and use consistent Neumann boundary conditions prescribed as

$$\frac{\partial p}{\partial n}^{n+1} = -\left[\frac{\partial \mathbf{u}}{\partial t}^{n+1} + \nu (\nabla \times \nabla \times \mathbf{u})^{*,n+1} + \mathbf{N}^{*,n+1} \right] \cdot \mathbf{n} \tag{11.4}$$

Here, the advection term is denoted as $\mathbf{N}^{*,n+1} = [\mathbf{u} \cdot \nabla \mathbf{u}]^{\star,n+1}$ where the superscript indicates extrapolation from previous solutions.

The second step solves a Helmholtz problem for each new velocity component $[u^{n+1}, v^{n+1}, w^{n+1}]$ which leads us to

$$\left(\Delta - \frac{\gamma_0}{\nu \Delta t} \right) \mathbf{u}^{n+1} = -\left(\frac{\gamma_0}{\nu \Delta t} \right) \hat{\mathbf{u}} + \frac{1}{\nu} \nabla p^{n+1}. \tag{11.5}$$

This form of the Velocity Correction scheme is generally referred to as Semi-Implicit scheme, because diffusion terms are treated implicit while advection is treated explicit.

The algorithm follows the general structure outlined in figure 11.1.1. Essentially, it solves for the new pressure p^{n+1} and velocity u^{n+1} based on initial conditions at $t^n = n\Delta t$ and boundary conditions. The three-step structure begins with evaluating the (explicit) advection terms. Next, a Poisson problem for the new pressure p^{n+1} is solved. Finally, a Helmholtz problem is solved for each velocity component ($\mathbf{u}^{n+1} = [u, v, w]^T$ in 3D).

11.1.1.1 Velocity Correction Scheme with a Weak Pressure formulation

One way to improve the Velocity Correction scheme is the weak pressure formulation. This form uses the same algorithm from figure 11.1.1. However, the scheme uses the

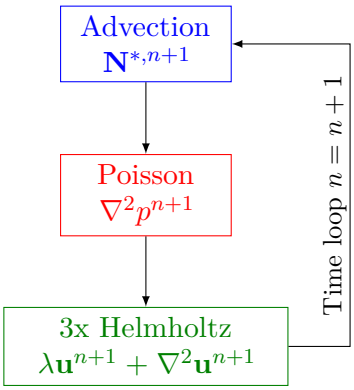


Figure 11.1 The algorithm of the Velocity Correction scheme to compute pressure and velocity at the new time $t^n = n\Delta t$.

extrapolated advection term in the pressure forcing instead of the boundary conditions. This changes the pressure Poisson boundary condition from equation Eq. (11.4) to become

$$\frac{\partial p}{\partial n}^{n+1} = -\Big[\frac{\partial \mathbf{u}}{\partial t}^{n+1} + \nu(\nabla \times \nabla \times \mathbf{u})^{*,n+1}\Big] \cdot \mathbf{n}. \tag{11.6}$$

Further details for this scheme can be found in the Developer-Guide.

11.1.1.2 Implicit Velocity Correction Scheme

The implicit solver handles the advection terms implicitly. Effectively, it uses the same formulation as the Velocity Correction scheme described above, but brings the advection operator on the left-hand side through a linearisation. The approach is similar to the work of [46, 11].

The new velocity equation with linearised advection term is

$$\Big(\Delta - \frac{\gamma_0}{\nu \Delta t} + \frac{1}{\nu}\mathbf{u} \tilde{\cdot} \nabla\Big)\mathbf{u}^{n+1} = -\Big(\frac{\gamma_0}{\nu \Delta t}\Big)\hat{\mathbf{u}} + \frac{1}{\nu}\nabla p^{n+1} \tag{11.7}$$

The algorithm follows a similar structure to the Semi-Implicit algorithm introduced above in section 11.1.1. The Poisson problem for the new pressure p^{n+1} is identical to the weak pressure formulation in section 11.1.1.1. The difference here is using an implicit advection operator that relaxes the Courant-Friedrichs-Levy (CFL) limitation for the time step size. This results in an Advection-Diffusion-Reaction (ADR) problem for the new velocity components.

The linearisation of the advection operator uses the approximation $[\mathbf{u} \cdot \nabla \mathbf{u}]^{n+1} \approx [\tilde{\mathbf{u}} \cdot \nabla \mathbf{u}]^{n+1}$ where $\tilde{\mathbf{u}}$ is an approximate advection velocity. The advection velocity $\tilde{\mathbf{u}}$ is implemented in two forms. The first form is referred to as `Extrapolated` as it is based on a simple extrapolation of previous velocity solutions [46]. The second form is called

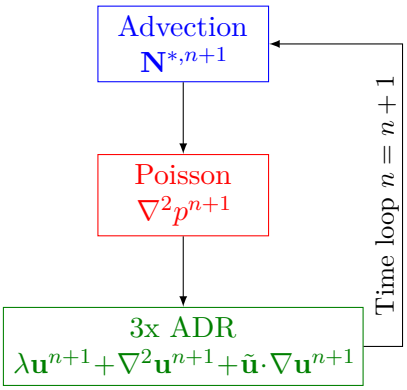


Figure 11.2 The algorithm of the Implicit Velocity Correction scheme using an ADR solve for the new velocity components $\mathbf{u}^{n+1}$.

`Updated` and it is based on a pressure-equation residual that uses the updated pressure gradient ∇p^{n+1} [11]. They are defined respectively as,

$$\texttt{Extrapolated} \qquad \tilde{\mathbf{u}}^{n+1} = \sum_q \frac{\alpha_q}{\gamma}\mathbf{u}^{n-q}, \tag{11.8}$$

$$\texttt{Updated} \qquad \tilde{\mathbf{u}}^{n+1} = \sum_q \frac{\alpha_q}{\gamma}\mathbf{u}^{n-q} - \frac{\Delta t}{\gamma}\left(\nabla p^{n+1} + [\mathbf{u}\cdot\nabla\mathbf{u}]^n + \nu\nabla\times\omega^n - \mathbf{f}^{n+1}\right). \tag{11.9}$$

The scheme defaults to `Extrapolated`, see the examples in section 11.11 for how to choose `Updated` instead.

Note that the advection matrix needs to be updated for every time-step, this potentially leads to increased computational cost when compared to the Semi-Implicit scheme. Also, consider that the solver is linearly-implicit instead of fully-implicit. This means that there are no sub iterations for each time step and thus no free parameters (e.g. tolerance) that need to be defined additionally.

11.1.1.3 Coordinate Transformation

For some problems the physical coordinate system is not the most computationally-efficient. In these cases, there is a coordinate transformation option for the Velocity Correction scheme that allows mapping from the physical coordinates to a numerical coordinate system. More details for this method can be found in the Developer Guide as well as examples on how to use it in section 11.11.

11.1.1.4 Substepping/subcycling the Velocity Correction Scheme

It is possible to use different time steps in the velocity correction scheme using a substepping (also known as subcycling) [42] or auxiliary semi-Lagrangian approach [51].

Originally the scheme was proposed by Maday, Patera and Ronquist who referred to as an operator-integration-factor splitting method [29]

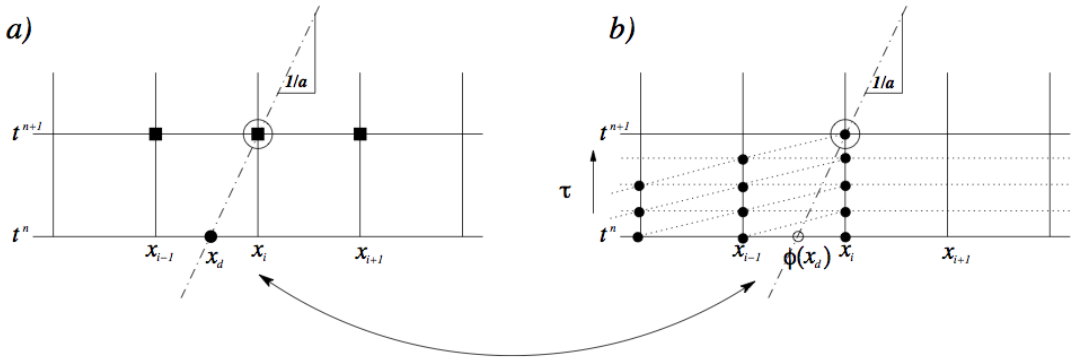


Figure 11.3 Schematic representation of the substepping approach. (a) Making an explicit time step the hyperbolic solution, travelling with a speed a, can be understood as being related to the solution at point x_d (the departure point). (b) Making smaller explicit time steps we can evaluate the solution $\phi(x_d)$ 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 11.1.1.4 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 [51] and [42]. 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

Some examples of the substepping scheme can be found in the regression tests directory under `$NEKHOME/Solver/IncNavierStokesSolver/Tests/` directory: `KovaFlow_SubStep_2order.xml`, `Hex_Kovasnay_SubStep.xml` and `Tet_Kovasnay_SubStep.xml`.

11.1.2 Immersed Boundary Methods: Smoothed Profile Method

The usual way to solve any PDE requires the definition of a well defined domain where the solution is to be determined. Thus, for complex geometries, the meshing process may get cumbersome and, in any case, the solver will have to the meshing process may get cumbersome, and likely struggle to avoid the presence of highly deformed and skewed elements. In addition, when solving cases with moving boundaries, the mesh has to be updated every time step to follow the shape of the boundaries, leading to a very resource and time-consuming simulation that limits the capabilities of the solver.

Immersed Boundary Methods may be very useful in these situations, where the definition of the boundaries requires a very complex mesh to reach convergence. The main idea behind them is the use of a forcing term in the incompressible Navier-Stokes equations in such a way that the mesh does not necessariliy follow the boundaries. The solution in the regions falling outside the boundaries is simply that of the boundaries, forcing the flow to behave as if there were a real object even if the mesh does not represent it. The method presented here is an adaptation of the Smoothed Profile Method [36] extended to a high-order semi-implicit splitting scheme [27, 49]. This method ensures that the no-slip, no-penetration and incompressibility constraints are mathematically enforced. Starting from the incompressible Navier-Stokes equations, the term $\mathbf{f_s}$ is added to the right hand side:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} + \mathbf{f_s} \tag{11.10a}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{11.10b}$$

The definition of this term depends on the method but, for the Smoothed Profile Method (SPM), it is related to a *shape function* $\Phi(\mathbf{x}, t)$ valued 0 in the fluid domain and 1 outside. It is usually defined as:

$$\Phi(\mathbf{x}, t) = -\frac{1}{2}\left[\tanh\left(\frac{d(\mathbf{x}, t)}{\xi}\right) - 1\right], \tag{11.11}$$

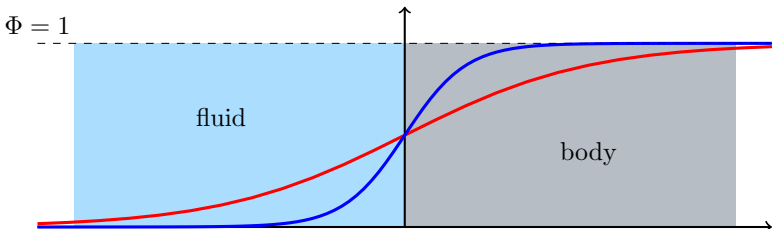


Figure 11.4 Definition of the shape function Φ close to the boundary of an immersed body.

being ξ a scaling factor [49] and $d(\mathbf{x}, t)$ a function representing the distance to the boundary (positive inside the body, negative inside the fluid). If the case to be simulated includes more than one immersed boundaries, the final shape function is calculated by adding the individual ones as long as they do not overlap:

$$\Phi = \sum_i \Phi_i \tag{11.12}$$

The approach followed during the implementation in *Nektar++* is an extension of the Velocity Correction Scheme, using the final velocity obtained with this method as an

intermediate velocity to determine the value of the forcing term. The initial equation Eq. (11.10a) is slightly modified and integrated in time by means of a *stiffly-stable* scheme and, then, split into different smaller parts that are solved separately:

$$\frac{\gamma_0 \mathbf{u}^{n+1} - \sum_{q=0}^{J-1} \alpha_q \, \mathbf{u}^{n-q}}{\Delta t} = -\sum_{q=0}^{J-1} \beta_q \, (\mathbf{u}\cdot\nabla\mathbf{u})^{n-q} - \nabla(p^* + p_p)^{n+1} + \nu\nabla^2\mathbf{u}^{n+1} + \mathbf{f}^{n+1} + {\mathbf{f_s}}^{n+1} \tag{11.13}$$

$$\frac{\tilde{\mathbf{u}} - \sum_{q=0}^{J-1} \alpha_q \, \mathbf{u}^{n-q}}{\Delta t} = -\sum_{q=0}^{J-1} \beta_q \, (\mathbf{u}\cdot\nabla\mathbf{u})^{n-q} + \mathbf{f}^{n+1}, \tag{11.14a}$$

$$\frac{\hat{\mathbf{u}} - \tilde{\mathbf{u}}}{\Delta t} = -\nabla {p^*}^{n+1}, \tag{11.14b}$$

$$\frac{\gamma_0 \mathbf{u}^* - \hat{\mathbf{u}}}{\Delta t} = \nu\nabla^2\mathbf{u}^{n+1}, \tag{11.14c}$$

$$\frac{\gamma_0 \mathbf{u}^{n+1} - \gamma_0 \mathbf{u}^*}{\Delta t} = -\nabla p_p^{n+1} + {\mathbf{f_s}}^{n+1} \tag{11.14d}$$

The new term $\mathbf{f_s}$ is defined as follows:

$${\mathbf{f_s}}^{n+1} = \frac{\gamma_0 \Phi^{n+1} ({\mathbf{u_p}}^{n+1} - \mathbf{u}^*)}{\Delta t}, \tag{11.15}$$

where α_q, β_q and γ_0 are coefficients of the stiffly-stable time integration method and $\mathbf{u_p}$ is the velocity of the points that lay outside the boundaries. Thus, the new term is just an acceleration proportional to the difference between the expected and the intermediate velocity, forcing the flow to follow the shapes defined by Φ and $\mathbf{u_p}$.

11.1.3 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 assumption 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;

11.1.3.1 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:

$$\frac{\partial \mathbf{u}'}{\partial t} + \mathbf{U} \cdot \nabla\mathbf{u}' + \mathbf{u}' \cdot \nabla\mathbf{U} = -\nabla p + \nu\nabla^2\mathbf{u}' + \mathbf{f} \tag{11.16a}$$

$$\nabla \cdot \mathbf{u}' = 0 \tag{11.16b}$$

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:

$$\mathbf{u}'(\mathbf{x}, t) = \mathcal{A}(\mathbf{U})\mathbf{u}'(\mathbf{x}, 0) \tag{11.17}$$

Let us assume that the base flow $\mathbf{U}$ is steady, then the perturbations are autonomous and we can assume that:

$$\mathbf{u}'(\mathbf{x},t) = \mathbf{q}'(\mathbf{x})\exp(\lambda t) \quad \text{where } \lambda = \sigma + i\omega \tag{11.18}$$

Then we obtain the associated eigenproblem:

$$\mathcal{A}(\mathbf{U})\mathbf{q}' = \lambda\mathbf{q}' \tag{11.19}$$

The dominant eigenvalue determines the behaviour of the flow. If the real part is positive then there exist exponentially growing solutions. Conversely, if all the eigenvalues have negative real part then the flow is linearly stable. If the real part of the eigenvalue is zero, it is a bifurcation point.

11.1.3.2 Adjoint Stability Analysis

The adjoint of a linear operator is one of the most important concepts in functional analysis and it plays an important role in understanding transition to turbulence. Let us write the linearised Navier-Stokes equation in a compact form:

$$\mathcal{H}\mathbf{q} = 0 \quad \text{where} \quad \mathcal{H} = \left(\begin{array}{c|c} -\partial_t - (\mathbf{U}\cdot\nabla) + (\nabla\mathbf{U})\cdot + \frac{1}{Re}\nabla^2 & -\nabla \\ \hline \nabla\cdot & 0 \end{array}\right) \tag{11.20}$$

The adjoint operator $\mathcal{H}^*$ is defined as:

$$\langle \mathcal{H}\mathbf{q}, \mathbf{q}\rangle = \langle \mathbf{q}, \mathcal{H}^*\mathbf{q}^*\rangle \tag{11.21}$$

Integrating by parts and employing the divergence theorem, it is possible to express the adjoint equations:

$$-\frac{\partial \mathbf{u}^*}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{u}^* + (\nabla\mathbf{U})^T\cdot\mathbf{u}^* = -\nabla p^* + \frac{1}{Re}\nabla^2\mathbf{u} \tag{11.22a}$$

$$\nabla\cdot\mathbf{u}^* = 0 \tag{11.22b}$$

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.

11.1.3.3 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:

$$\mathcal{A}(\tau)\mathbf{v} = \sigma\mathbf{u}, \quad \|\mathbf{u}\| = 1 \tag{11.23}$$

where $\sigma = \|\mathbf{u}'(\tau)\|$. This is no other than 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 11.1.3.3. 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.

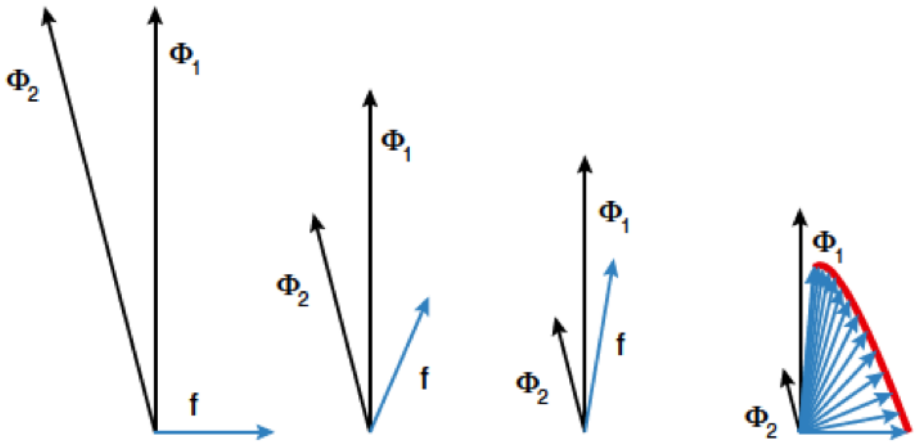


Figure 11.5 Geometric interpretation of the transient growth. Adapted from Schmid, 2007

11.1.4 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 [21]. 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

$$\begin{cases} \dot{q} = NS(q) - \chi(q - \bar{q}), \\ \dot{\bar{q}} = \frac{q-\bar{q}}{\Delta}. \end{cases} \tag{11.24}$$

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.

11.1.5 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.

Note

The coupled solver is only supported for two-dimensional or quasi-3D problems, and only using a direct solver (e.g. `DirectStaticCond`) which prevents its use in parallel.

We consider the weak form of the Stokes problem for the velocity field $\boldsymbol{u} = [u, v]^T$ and the pressure field p:

$$(\nabla\phi, \nu\nabla\boldsymbol{u}) - (\nabla\cdot\phi, p) = (\phi, \boldsymbol{f}) \tag{11.25a}$$

$$(q, \nabla\cdot\boldsymbol{u}) = 0 \tag{11.25b}$$

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:

$$\begin{bmatrix} A - BC^{-1}B^T & D_b^T - BC^{-1}D_i & 0 \\ D_b - D_i^T C^{-1}B^T & -D_i^T C^{-1}D_i & 0 \\ B^T & D_i & C \end{bmatrix} \begin{bmatrix} \boldsymbol{u_b} \\ p \\ \boldsymbol{u_i} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f_b} - BC^{-1}\boldsymbol{f_i} \\ -D_i^T C^{-1}\boldsymbol{f_i} \\ \boldsymbol{f_i} \end{bmatrix} \tag{11.26}$$

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, 43].

11.2 Usage

To run the incompressible solver in serial:

```
IncNavierStokesSolver mesh.xml session.xml -v
```

where `IncNavierStokesSolver` is the name of the executable, `mesh.xml` is the name of the file that inlcudes all the high-order mesh information, `session.xml` is the name of the file that describes the polynomial expansions for the pressure and velocity fields, the

boundary conditions and the numerical configuration of the problem. It is recommend to use the `-v`, or `-verbose` flag which activates additional information to be printed upon execution of the executable. It is possible to have the information from `mesh.xml` and `session.xml` in a single file, for example `meshAndSession.xml`. This compact format for the problem set-up is useful for small size problems and the previous command will look like:

```
IncNavierStokesSolver meshAndSession.xml -v
```

In *Nektar++*, it is possible to restart a simulation from an existing pressure and velocity field. In this scenario, the simulation will resume from the latest available time-step in the restart field and the numbering of the output checkpoint files will continue from the latest index. To avoid this and set the start time of the simulation and/or the index numbering of the checkpoint files one can use:

```
IncNavierStokesSolver meshAndSession.xml -v --set-start-time 0 --set-start-
    chknumber 0
```

where the flag `-set-start-time` receives float values that set the starting time of the simulation and the flag `-set-start-chknumber` receives int values that specify the starting index for the output checkpoint files regardless of the settings introduced by the initialization field.

Note

If you want to run the IncNavierStokesSolver process in the background, so that you continue using the existing terminal, then you can use the following:
`IncNavierStokesSolver meshAndSession.xml -v > logAndErr.IncNS &`
In this command, the `> logAndErr.IncNS` is specified to dump all the information that is printed from the executable inside the `logAndErr.IncNS` file and the error information in case an error shows up, the & symbol is responsible for launching the process in the background. This way, the current terminal can be used for different actions and the job will continue to run, even if the user loses connection from the terminal that the process was launched. To monitor the progress of the simulation and how far it is from completion when the process is running in the background:
`tail -f logAndErr.IncNS`

If *Nektar++* is compiled with HDF5 support, then the output fields can be exported in compressed HDF5 format using the follwing command:

```
IncNavierStokesSolver mesh.xml session.xml -v -i Hdf5
```

where the flag `-i` or `-io-format Hdf5` enables the `IncNavierStokesSolver` executable to write the velocity and the pressure fields in HDF5. For large scale problems where the number of elements is above one million, it is recommended to convert the mesh in compressed HDF5 format and use the flag `-use-hdf5-node-comm` to enable loading the mesh in parallel and avoid running out of memory, in case of serial partitioning, as shown below:

```
IncNavierStokesSolver mesh.xml session.xml -v -i Hdf5 --use-hdf5-node-comm
```

If *Nektar++* is configured to run in parallel as described in Section 17.1, then the following command is used:

```
mpirun -np 10 IncNavierStokesSolver mesh.xml session.xml -v
```

where `mpirun` can be replaced by `mpiexec` depending on the MPI protocol that is available in the system and 10 is the number of the requested processors. A detailed summary of all the available flags is written in Section 16.

Note

To list all the available flags for the `IncNavierStokesSolver` executable, run: `IncNavierStokesSolver -h`

11.3 Setting up the Simulation

In the following the possible options are shown for setting up a simulation using the incompressible Navier-Stokes.

11.3.1 Expansions and Approximation Spaces

The `Expansions` are introduced generally in section 3.2. 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 with all variables having a same polynomial order (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

For well resolved simulations it appears that often using the same polynomial space for the pressure and velocity does give suitable answer but this does not satisfy the so-called LBB or inf-sup condition. Therefore, it is potentially better to specify an equivalent of the **Taylor Hood** approximation and use one higher polynomial order for velocity than the pressure with a continuous expansion. To specify this type of expansion you can use an expansion section of the form:

```
<EXPANSIONS>
      <E COMPOSITE="C[0]" NUMMODES="8" FIELDS="u,v" TYPE="MODIFIED" />
      <E COMPOSITE="C[0]" NUMMODES="7" FIELDS="p"   TYPE="MODIFIEDQUADPLUS1" />
</EXPANSIONS>
```

In the above example the "u,v" fields are specified to have a polynomial order of 7 using a modified expansion. Implicitly this form of the expansion definition uses a quadrature order of 9. The above definition then also uses a modified expansion for pressure but of polynomial order 6. Since currently for this solver to run we need to use a consistent quadrature order for both the velocity and pressure fields we specify the `MODIFIEDQUADPLUS1` to tell the solver to use an additional quadrature point and therefore also use 9 quadrature points in each 1D direction for the pressure.

In other cases it is sometimes useful to run with an even higher quadrature order, for example to handle highly deformed elements where the Jacobian is represented by a polynomial expansion. This can be done by using a more detailed definition of the expansion of the form:

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" BASISTYPE="Modified_A,Modified_B" NUMMODES="8,8"
  POINTSTYPE="GaussLobattoLegendre,GaussRadauMAlpha1Beta0" NUMPOINTS="9,8"
  FIELDS="u,v" />
    <E COMPOSITE="C[0]" BASISTYPE="Modified_A,Modified_B" NUMMODES="7,7"
  POINTSTYPE="GaussLobattoLegendre,GaussRadauMAlpha1Beta0" NUMPOINTS="9,8"
  FIELDS="p" />
</EXPANSIONS>
```

In this example we have specified an 8th order expansion for "u,v" and a 7th order expansion for "p". The BasisType is given as "Modified_A, Modified_B" which is for a triangular expansion (note that for a quadrilateral expansion it would have been "Modified_A,Modified_A") and so the number of quadrature points in this case is 9 in the first direction which uses Gauss-Lobatto-Legendre points but only 8 in the second direction since this uses a Gauss-Radau formula with $\alpha = 1, \beta = 0$ weights (see [22] for details).

Further information is also available in Section 3.2.

11.3.2 Governing Equation

The first thing in setting up a simulation is to define what governing eqautions are to be solved. This can be done using the `EqType` tag in the `SOLVERINFO` section.

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

For example, for solving the unsteady Navier-Stokes equations, the following should be included inside the `SOLVERINFO` tags

```
<SOLVERINFO>
<I PROPERTY="EQTYPE" VALUE="UnsteadyNavierStokes"/>
</SOLVERINFO>
```

Please note that, we will have only one `SOLVERINFO` tags in the xml file which inclues all the settings. If more than one `SOLVERINFO` is defined in the xml session file, the last one will override the previous ones and the settings could be lost or be overwritten with the defaults.

11.3.3 Solution Algorithms

Depend on the type of the equations that will be solved, a consistent sovler scheme should be defined in the `SOLVERINFO` using `SolverType` tag.

Available options are as shown next:

Algorithm	SolverType	Dimensions	Projections
Velocity Correction Scheme (VCS)	`VelocityCorrectionScheme`	2D, Quasi-3D, 3D	CG, CG-DG
Implicit VCS	`VCSImplicit`	2D and 3D	CG
VCS with coordinate transformation	`VCSMapping`	2D, Quasi-3d and 3D	CG, CG-DG
VCS with weak pressure	`VCSWeakPressure`	2D, Quasi-3D, 3D	CG, CG-DG
Smoothed Profile Method (SPM)	`SmoothedProfileMethod`	2D, Quasi-3D, 3D	CG, CG-DG
Direct solver	`CoupledLinearisedNS`	2D, Quasi-3D	CG

For exmaple, for the unsteady incompressible Navier-Stokes, using the VCS scheme otherwise known as Velocity Correction Scheme, the follwoing should be added inside the `SOLVERINFO` tags.

```
<I PROPERTY="SolverType"  VALUE="VelocityCorrectionScheme"/>
```

or if we want to improve the timestepping stability and hence being able to run with higher CFL condition, this could be done using the implicit VCS as:

```
<I PROPERTY="SolverType"  VALUE="VCSImplicit"/>
```

Along with setting the solver scheme, we need to set the following tags in `SOLVERINFO` tags

- `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. 11.1.4)	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

11.3.4 TimeIntegrationScheme

- `TimeIntegrationScheme`: sets the time integration as

  ```
  <TIMEINTEGRATIONSCHEME>
      <METHOD> IMEX </METHOD>
      <ORDER> 2 </ORDER>
  </TIMEINTEGRATIONSCHEME>
  ```

  Possible values are

Time-Integration Scheme	`Method`	`Order`	Dimensions	Equations	Projections
IMEX Order 1	`IMEX`	`1`	all	US, UNS	CG
IMEX Order 2	`IMEX`	`2`	all	US, UNS	CG
IMEX Order 3	`IMEX`	`3`	all	US, UNS	CG
Backward Euler	`BackwardEuler`	`1`	all	US, UNS	CG-DG
BDF Order 1	`BDFImplicit`	`1`	all	US, UNS	CG-DG
BDF Order 2	`BDFImplicit`	`2`	all	US, UNS	CG-DG

- `Extrapolation`: Specify the extrapolation method (standard or substepping) 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 Scheme	`SubStepIntScheme`
ForwardEuler, Order 1	`ForwardEuler, Order 1`
RungeKutta, Order 2	`RungeKutta, Variant SSP, Order 2`

  This option is useful if you wish to use an overall scheme that is first order accurate for example with TimeIntegrationScheme as BDFImplicit Order 1 but using a second order RungeKutta, Variant SSP, Order 2 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, or the velocity Helmholtz equation in the splitting-scheme. It can be globally set in section `SOLVERINFO` as:

```
1 <I PROPERTY="GlobalSysSoln" VALUE="IterativeStaticCond"/>
```

  Possible values are:

System solution	`GlobalSysSoln`	Parallel
Direct Solver (DS)	`DirectFull`	quasi-3D
DS with Static Condensation	`DirectStaticCond`	quasi-3D
DS with Multilevel Static Condensation	`DirectMultiLevelStaticCond`	quasi-3D
Iterative Solver (IS)	`IterativeFull`	quasi-3D
IS with Static Condensation	`IterativeStaticCond`	quasi-3D, 3D
IS with Multilevel Static Condensation	`IterativeMultiLevelStaticCond`	quasi-3D, 3D
DS via Xxt	`XxtFull`	2D, quasi-3D, 3D
IS with Static Condensation via Xxt	`XxtStaticCond`	2D, quasi-3D, 3D
IS with Multilevel Static Condensation via Xxt	`XxtMultiLevelStaticCond`	2D, quasi-3D, 3D
DS via PETSc	`PETScFull`	2D, quasi-3D, 3D
IS with Static Condensation via PETSc	`PETScStaticCond`	2D, quasi-3D, 3D
IS with Multilevel Static Condensation via PETSc	`PETScMultiLevelStaticCond`	2D, quasi-3D, 3D

Default values are `DirectMultiLevelStaticCond` in serial and `IterativeStaticCond` in parallel. More information on each solver can found in Section 3.4.5. For 2D problems that run in parallel, it is recommended to use the `XxtMultiLevelStaticCond` solver for all the flow variables. It is possible to specify a different solver and preconditioner for each of the pressure and velocity components in a separate `GLOBALSYSSOLNINFO` section. If an Iterative Solver is used, then different tolerances can be specified for each variable and whether the relative, or absolute error is monitored. An efficient set-up for parallel simulations is presented below:

```
<GLOBALSYSSOLNINFO>
  <V VAR="u,v,w">
    <I PROPERTY="GlobalSysSoln"             VALUE="IterativeStaticCond" />
    <I PROPERTY="Preconditioner"            VALUE="LowEnergyBlock"/>
    <I PROPERTY="AbsoluteTolerance"         VALUE="True"/>
    <I PROPERTY="IterativeSolverTolerance"  VALUE="1e-2"/>
  </V>
  <V VAR="p">
    <I PROPERTY="GlobalSysSoln"             VALUE="IterativeStaticCond" />
    <I PROPERTY="Preconditioner"            VALUE="Diagonal"/>
    <I PROPERTY="AbsoluteTolerance"         VALUE="True"/>
    <I PROPERTY="IterativeSolverTolerance"  VALUE="1e-4"/>
  </V>
</GLOBALSYSSOLNINFO>
```

In case the relative error is tracked, where `PROPERTY="AbsoluteTolerance" VALUE="False"` (default setting), attention must be given on setting the `IterativeSolverTolerance`. As a guideline for setting the tolerances, some trial runs are necessary. In more detail, the flag `-v` needs to be used and read the information provided by the solver. When the Conjugate Gradient solver is used, the tolerance, the arithmetic error and the right-hand side magnitude(rhs_mag) is printed at every time-step. Then, the rhs_mag needs to be multiplied by the arithmetic error printed to express the stopping criteria for the Conjugate Gradient solver. If the order of this product is too large, then the tolerance needs to be reduced. The tolerance can reach values up to $1e-15$ depending on the problem size, mesh and configuration.

Note

To use the solvers and preconditioners that utilize the Xxt library, *Nektar++* needs to be compiled with MPI support. Similarly for PETSc, the setting `NEKTAR_USE_PETSC` needs to be activated upon configuring the compilation.

11.3.5 Stabilisation Techniques

There are various techniques and options available to stablise the simulation as explained in this section. Should any of these options to be used, they are needed to be included in the `SOLVERINFO` tags.

11.3.5.1 Dealiasing

One source of instablity in the simulation could be related to aliasing. This can be overcome using "Dealiasing" technique. The following options are available and if needed should be included in the `SOLVERINFO` section of the xml file.

- `DEALIASING`
- `SPECTRALDEALIASING`: activates the 3/2 padding rule on the advection term of a Quasi-3D simulation.

```
<I PROPERTY="DEALIASING" VALUE="True"/>
```

- `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" />
```

11.3.5.2 Spectral Vanishing Viscousity

Spectral Vanishing Viscousity (SVV) activates a stabilization technique which increases the viscosity on the modes with the highest frequencies. In the Nektar++ there are two kinds of SVV are supported, one acts on the spectral/hp expansions and the other on the Fourier expansions. Additionally, the SVV supports several different kernels as will be shortly explained next.

To activate SVV for the spectral/hp expansions, i.e. in 2D and 3D simulaitons as well as the xy domain in Quasi-3D simulations, the following should be included in the `SOLVERINFO` tag.

```
<I PROPERTY="SpectralVanishingViscosity" VALUE="True"/>
```

Using the `VALUE="True"` will activates the `Exponential Kernel` by default. There are indeed tree kernels available as summarized next:

SVV Kernel	`SpectralVanishingViscosity`
Exponential Kernel	`True`
Power Kernel	`PowerKernel`
DG Kernel	`DGKernel`

The Exponential kernel is based on the work of Maday et al. [30], its extension to 2D can be found in [23]. A diffusion coefficient can be specified which defines the base magnitude of the viscosity; this parameter is scaled by h/p. SVV viscosity is activated for expansion modes greater than the product of the cut-off ratio and the expansion order. The Power kernel is a smooth function with no cut-off frequency; it focusses on a narrower band of higher expansion modes as the polynomial order increases. The cut-off ratio parameter for the Power kernel corresponds to the power ratio, see Moura et al. [32].

It is recommended that for the spectral/hp SVV one uses `DGKernel` as follows:

```
<I PROPERTY="SpectralVanishingViscosity" VALUE="DGKernel"/>
```

also note that the `DGKernel` is only supported for the spectral/hp expansions.

Using any of the SVV Kernels, the cut-off ratio and diffusion coefficient will be set by default. However, as mentioned above these are could be modified by the user. For the `DGKernel`, the default diffusion coefficient is one and can be changed by setting the `SVVDiffCoeff` parameter in the `<PARAMETERS>` tag. It is recommended that the defalut values for the `DGKernel` won't be changed, however, should the user wanted to changed it for exmaple, to reduce the amount of the diffusion coefficient to "0.75", this can be done as shown below:

```
<P> SVVDiffCoeff  = 0.75 </P>
```

For the `Exponential Kernel` there are two parameters that can be adjusted. The first parameter, i.e. `SVVCutOffRatio`, controls the cut-off ratio and the second parameter is the `SVVDiffCoeff` that controls the diffusion coefficient.

A practical hint especially for high-Reynolds number flows is that if the simulation is not very well stable using the default values of exponential kernel, one could start with the cut-off ratio around "0.5" and the diffusion coefficient of "1" and gradually increases the cut-off ratio and/or reducing the diffusion coefficient to get the stable solution with the least amount of dissipation.

In a Quasi-3D simulation, in addition to the spectral/hp modes, one can activate the SVV for the Fourier expansions too. In this case, these two kinds of SVV can be independently activated using the `SpectralVanishingViscositySpectralHP` for spectral/hp and `SpectralVanishingViscosityHomo1D` for Forier expansions. Additionally, the cut-off ratio and diffusion coefficients are also can be set independently. For the spectral/hp this would be the same as explained before. For the Fourier expansions the cut-off ratio and diffusion coefficient can be set using `SVVCutoffRatioHomo1D` and `SVVDiffCoeffHomo1D` parameters respectively. These paramters need to be set in the `PARAMETERS` tag.

```
<P> SVVDiffCoeff  = 0.1 </P>
<P> SVVCutOffRatio= 0.5 </P>
<P> SVVDiffCoeffHomo1D  = 0.1 </P>
<P> SVVCutOffRatioHomo1D  = 0.5 </P>
```

11.3.5.3 GJPStabilisation

GJPStabilisation activates the gradient jump penalty[33] stabilization technique. The GJP is less dissipative than the spectral vanishing viscousity (SVV) stabilisation and gives a better turbulent structures specially when the simulaiton is under-resolved or in other words when the polynomial order is low. It is expected that with increasing

the polynomial order and approaching towards a resolved solution both SVV and GJP stabilisation solutions approaches each other. Since the GJP is less dissipative than the SVV, it means that it probably requires a smaller Δt for the stable solution. Further, GJP only affects the spectral/hp discretisations and doesn't have any stabilisation effect on the Fourier discritisation. This means that in the Quasi-3D simulations, if GJP is activated we still needs to use the SpectralVanishingViscousityHomo1D to stablise the solution in the spanwise direction which is discretised using the Fourier method. For the 2D and full 3D problems, GJP stabilisation should suffices though.

```
1 <I PROPERTY="GJPStabilisation" VALUE="SemiImplicit"/>
```

Another option for the GJP stabilisation is to use `Explicit` for its value instead of `SemiImplicit`. If the `Explicit` value is set, there is an additional option that can be set for the explicit GJP stabilisation in the solver info as follwos.

```
1 <I PROPERTY="GJPNormalVelocity" VALUE="True"/>
```

Using the `GJPNormalVelocity`, the GJP formulation will uses the average velocity normal to the edge/face as the velocity scaling of jump term.

11.3.6 Simulation Parameters

In Setting up the simulation, various parmeters can be used to set required parameters such as TimeStep or determine the controls such as number of I/O. The following parameters can be specified in the `PARAMETERS` section of the session file. Note that some of these have already been discussed in previous sections and only repeated here for the sake of completeness.

- `TimeStep`: sets the timestep for the integration in time formula.
- `NumSteps`: sets 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.
- `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 (Exponential and Power kernel), 1 (DG-Kernel)).
- `GJPJumpScale`: This is a parameter that scales all of the jump terms in the GJP stabilisation formualtion. The default value `GJPJumpScale=1`.
- `IO_CFLWriteFld`: sets a treshold value for the CFL number. If CFL exceeds this value, the flow field is written to file (only once). This is useful for debugging purposes, allowing to visually inspect a flow field that is becoming unstable.
- `IO_CFLWriteFldNumSteps`: sets the number of timesteps after which `IO_CFLWriteFld` becomes operational. This avoids writing the flow field at the beginning of a simulation when initialising a new geometry.

11.3.7 Boundary conditions

The process of setting up the incompressible solver requires setting apropriate boundary conditions. This includes consistent pressure boundary conditions, outflow boundary conditions for truncated domains. Additionally, for the biomechanical simulaitons such as pulsatile flows, e.g. flow in arteries a specific type of boundary condition such as Wormersley conditions may be required. These boundary conditions are explained in this section.

11.3.7.1 Pressure boundary conditions

In order to specify the pressure boundary conditions given by equation (11.4) or for the equivalent conditions in the VCSWeakPressure scheme the `USERDEFINEDTYPE` condition "H" can be used. Therefore a zero velocity wall boundary condition on boundary region 0 in two-dimensions can be specified as

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="u" VALUE="0" />
    <D VAR="v" VALUE="0" />
    <N VAR="p" USERDEFINEDTYPE="H" VALUE="0"  />
  </REGION>
</BOUNDARYCONDITIONS>
```

11.3.7.2 Outflow boundary conditions

The most straightforward outflow condition is to specify fully developed conditions of $\nabla \mathbf{u}^{n+1} \cdot \mathbf{n} = 0$ and $p = 0$ which can be specified as

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <N VAR="u" VALUE="0" />
    <N VAR="v" VALUE="0" />
```

```
      <D VAR="p" VALUE="0"  />
    </REGION>
  </BOUNDARYCONDITIONS>
```

However when energetic vortices pass through an outflow region one can experience instabilities as identified by the work of Dong, Karnidakis and Chryssostomidis [10]. In this paper they suggest to impose a pressure Dirichlet outflow condition of the form

$$p^{n+1} = \nu \mathbf{n} \cdot \nabla \mathbf{u}^{*,n+1} \cdot \mathbf{n} - \frac{1}{2} \mid \mathbf{u}^{*,n+1} \mid^2 S_o(\mathbf{n} \cdot \mathbf{u}^{*,n+1}) + \mathbf{f}_b^{n+1} \cdot \mathbf{n} \tag{11.27}$$

with a step function defined by $S_o(n \cdot \mathbf{u}) = \frac{1}{2}(1 - \tanh \frac{n \cdot \mathbf{u}}{\mathbf{u_0} \delta})$, where $\mathbf{u_0}$ 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. $\mathbf{f}_b = 0$. (see the test KovaFlow_m8_short_HOBC.xml for a non-zero example). Note that in the paper [10] they define this term as the negative of what is shown here so that it could be use used to impose a default pressure values. This does however mean that the forcing term is imposed through the velocity components u, v by specifying the entry `VALUE` (An example can be found in ChanFlow_m3_VCSWeakPress_ConOBC.xml). For the velocity component one can specify

$$\nabla \mathbf{u}^{n+1} \cdot \mathbf{n} = \frac{1}{\nu} \Big[p^{n+1} \mathbf{n} + \frac{1}{2} \mid \mathbf{u}^{*,n+1} \mid^2 S_o(\mathbf{n} \cdot \mathbf{u}^{*,n+1}) - \nu (\nabla \cdot \mathbf{u}^{*,n+1}) \mathbf{n} - \mathbf{f}_b^{n+1} \Big] \tag{11.28}$$

This condition can be enforced using the `USERDEFINEDTYPE` "HOutflow", i.e.

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <N VAR="u" USERDEFINEDTYPE="HOutflow" VALUE="0" />
    <N VAR="v" USERDEFINEDTYPE="HOutflow" VALUE="0" />
    <D VAR="p" USERDEFINEDTYPE="HOutflow" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

Note that in the moving body work of Bao et al. [4] some care must be made to identify when the flow over the boundary is incoming or outgoing and so a modification of the term

$$\frac{1}{2} \mid \mathbf{u}^{*,n+1} \mid^2 S_o(\mathbf{n} \cdot \mathbf{u}^{*,n+1})$$

is replaced with

$$\frac{1}{2} \Big((\theta + \alpha_2) \mid \mathbf{u}^{*,n+1} \mid^2 + (1 - \theta + \alpha_1)(\mathbf{u}^{*,n+1} \cdot \mathbf{n}) \mathbf{u}^{*,n+1} \Big) S_o(\mathbf{n} \cdot \mathbf{u}^{*,n+1})$$

where the default values are given by $\theta = 1, \alpha_1 = 0, \alpha_2 = 0$ and these values can be set through the parameters `OutflowBC_theta`, `OutflowBC_alpha1` and `OutflowBC_alpha2`.

Dong has also suggested convective like outflow conditions in [9] which can be enforced through a Robin type specification of the form

$$\frac{\partial \mathbf{u}^{n+1}}{\partial n} + \frac{\gamma_0 D_0}{\Delta t}\mathbf{u}^{n+1} = \frac{1}{\nu}\Big[\mathbf{f}^{n+1} + \mathbf{E}(\mathbf{n}, \mathbf{u}^{*,n+1}) + p^{n+1}\mathbf{n} - \nu(\nabla \cdot \mathbf{u}^{*,n+1})\mathbf{n}\Big] + \frac{D_0}{\Delta t}\hat{\mathbf{u}} \quad (11.29)$$

$$\begin{aligned}\frac{\partial p^{n+1}}{\partial n} + \frac{1}{\nu D_0}p^{n+1} &= -\Big(-\nu(\nabla \times \nabla \times \mathbf{u})^{*,n+1} + \mathbf{N}^{*,n+1}\Big) \cdot \mathbf{n} \\ &\quad - \frac{1}{\nu D_0}\Big[\mathbf{f}^{n+1} + \mathbf{E}(\mathbf{n}, \mathbf{u}^{*,n+1} + p^{n+1}\mathbf{n} - \nu(\nabla \cdot \mathbf{u}^{*,n+1})\mathbf{n}\Big] \quad (11.30)\end{aligned}$$

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <R VAR="u" USERDEFINEDTYPE="HOutflow" VALUE="0" PRIMCOEFF="D0/TimeStep"/>
    <R VAR="v" USERDEFINEDTYPE="HOutflow" VALUE="0" PRIMCOEFF="D0/TimeStep"/>
    <R VAR="p" USERDEFINEDTYPE="HOutflow" VALUE="0" PRIMCOEFF="1.0/(D0*Kinvis)"/>
  </REGION>
</BOUNDARYCONDITIONS>
```

11.3.7.3 Womersley Boundary Condition

It is possible to define the time-dependent Womersley velocity profile for pulsatile flow in a pipe. The modulation of the velocity profile is based on the desired peak or centerline velocity which can be represented by a Fourier expansion $U_{max} = A(\omega_n)e^{i\omega_n t}$ where A are the Fourier modes and ω the frequency. The womersely solution is then defined as:

$$w(r,t) = A_0(1-(r/R)^2) + \sum_{n=1}^{N} \tilde{A}_n[1 - \frac{J_0(i^{3/2}\alpha_n r/R)}{J_0(i^{3/2}\alpha)}]e^{i\omega_n t}$$

where the womersley number α is defined:

$$\alpha_n = R\sqrt{\frac{2\pi n}{T\nu}}$$

and $\tilde{A}_n$ $(n = 1 : N)$are the Fourier coefficients scaled in the following way:

$$\tilde{A}_n = 2A_n/[1 - \frac{1}{J_0(i^{3/2}\alpha)}]$$

The Womersley velocity profile is implemented in the following way:

```
<REGION REF="0">
    <D VAR="u" USERDEFINEDTYPE="Womersley:WomParams.xml" VALUE="0" />
    <D VAR="v" USERDEFINEDTYPE="Womersley:WomParams.xml" VALUE="0" />
    <D VAR="w" USERDEFINEDTYPE="Womersley:WomParams.xml" VALUE="0" />
    <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
</REGION>
```

A file containing the Fourier coefficients, $\tilde{A}$, must be in the directory where the solver is called from. The name of the file is defined by the string given in the attribute `USERDEFINEDTYPE` after the ":" and contains the real and imaginary coefficients. This file has the format

```
<NEKTAR>
  <WOMERSLEYBC>
    <WOMPARAMS>
      <W PROPERTY="Radius" VALUE="0.5" />
      <W PROPERTY="Period" VALUE="1.0" />
      <W PROPERTY="axisnormal" VALUE="0.0,0.0,1.0" />
      <W PROPERTY="axispoint" VALUE="0.0,0.0,0.0" />
    </WOMPARAMS>

    <FOURIERCOEFFS>
      <F ID="0"> 0.600393641193,    0.0               </F>
      <F ID="1"> -0.277707172935,   0.0767582715413   </F>
      <F ID="2"> -0.0229953131146,  0.0760936232478   </F>
      <F ID="3"> 0.00858135174058,  0.017089888642    </F>
      <F ID="4"> 0.0140332527651,   0.0171575122496   </F>
      <F ID="5"> 0.0156970122129,   -0.00547357750345 </F>
      <F ID="6"> 0.00473626554238,  -0.00498786519876 </F>
      <F ID="7"> 0.00204434981523,  -0.00614566561937 </F>
      <F ID="8"> -0.000274697215201, 0.000153571881197 </F>
      <F ID="9"> -0.000148037910774, 2.68919619581e-05 </F>
    </FOURIERCOEFFS>
  </WOMERSLEYBC>
</NEKTAR>
```

Each value of $\tilde{A}$ is provided in the `FOURIERCOEFFS` section and provided as separate entries containing the real and imaginary components, i.e. the mean component provided above is $0.600393641193, 0.0$.

Similarly in the `WOMPARAMS` section the key parameters of the boundary condition are also provided as:

- `RADIUS` is the radius of the boundary.
- `PERIOD` is the cycle time period,
- `AXISNORMAL` defines the normal direction to the boundary,
- `AXISPOINT` defines a coordinate in the boundary centre,

11.3.8 Forcing

11.3.8.1 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 [37]. 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 `TIMEINTEGRATIONSCHEME` and `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. For example:

```
<TIMEINTEGRATIONSCHEME>
    <METHOD> IMEX </METHOD>
    <ORDER> 2 </ORDER>
</TIMEINTEGRATIONSCHEME>

<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="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 called `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</PARAM>
      <PARAM NAME="OutputFrequency">10</PARAM>
      <PARAM NAME="Boundary"> B[0] </PARAM>
  </FORCE>
```

During the execution a file named `DragLift.fce` 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 11.7. 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.

11.3.9 Filters

The following filters are supported exclusively for the incompressible Navier-Stokes solver. Further filters from section 3.5 are also available for this solver.

- `Aerodynamic forces` (section 3.5.2)
- `Aerodynamic forces SPM` (section 3.5.2.1)
- `Kinetic energy and enstrophy` (section 3.5.15)
- `Mean value` (section 3.5.16)
- `Modal energy` (section 3.5.17)
- `Moving body` (section 3.5.18)
- `Reynolds stresses` (section 3.5.21)

11.4 Smoothed Profile Method: Session file configuration

As the Immersed boundary method, i.e. the smoothed profile method (see Sec. 11.1.2) is a derived class of the Velocity Correction Scheme (VCS) (see Sec. 11.1.1 and Sec. 11.3), hence most of the setting introduced for the VCS is applicable for the SPM and hence will not be repeated here and only the unique options to the SPM will be discussed here.

For the SPM simulations, the property `SolverType` must be set to `SmoothedProfileMethod`, while the immersed boundaries are defined in a function called `ShapeFunction`. Besides, the property `ForceBoundary` can be set to `True` for a more accurate velocity profile inside the body in expense of adding a slightly extra compressibility. Thus, the `TIMEINTEGRATIONSCHEME` and `SOLVERINFO` section could be similar to:

```
<TIMEINTEGRATIONSCHEME>
    <METHOD> IMEX </METHOD>
    <ORDER> 3 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
    <I PROPERTY="SolverType"            VALUE="SmoothedProfileMethod" />
    ...
    <I PROPERTY="ForceBoundary"         VALUE="True"                  />
</SOLVERINFO>
```

The `ShapeFunction` which defines the shape of immersed boundary must be defined in the `CONDITIONS` section:

```
<FUNCTION NAME="ShapeFunction">
    <E VAR="Phi" USERDEFINEDTYPE="TimeDependent" VALUE="..." />
    <E VAR="Up" VALUE="..." />
    <E VAR="Vp" VALUE="..." />
    ...
</FUNCTION>
```

As a brief guideline, to define a cylinder of radius 0.5 and center at the point (0,0) according to expression (11.11), the `"Phi"` field in the `ShapeFunction` function should be:

```
    <E VAR="Phi" USERDEFINEDTYPE="TimeDependent" VALUE="-0.5*(tanh((rad(x,y)-0.5)
    /0.04)-1.0)" />
```

where the scaling coefficient has been set to 0.04. The variable names are compulsory, being `Phi` the shape of the bodies, and `Up`, `Vp` and `Wp` functions representing the velocity field inside them. The attribute `USERDEFINEDTYPE` is compulsory only if the functions depend on time, when it must be set to `"TimeDependent"`.

For immersed boundaries with geometries that cannot be represented by means of analytical functions, an `.stl` binary file can be supplied as well. However, the geometry file must be first converted to `.fld` format with the `phifile` module of `FieldConvert`. The simplest way to proceed is by issuing the command:

```
FieldConvert -m phifile:file=geom.stl:scale=value session.xml geom.fld
```

where the value of `scale` corresponds to the coefficient ξ in equation Eq. (11.11). More details can be found in section 5.6.38. In any case, it is important to remark that this functionality is still under development.

In the `ShapeFunction` block of the session file, the line `<E VAR="Phi" ... />` indicates that the immersed bodies are defined by the function introduced in `VALUE`, while a line like the following:

```
    <F VAR="Phi" FILE="geometry.fld" />
```

must be used when the Φ field is defined in an `.stl` file previously converted to `.fld` format. In this case, the solver only supports non-moving geometries and the attribute `USERDEFINEDTYPE="TimeDependent"`, if specified, will not be used.

11.5 Session file configuration: Linear stability analysis

Stability analyses of incompressible flow involves solving the linearised Navier-Stokes equations

$$\frac{\partial \mathbf{u}'}{\partial t} + \mathcal{L}(\mathbf{U}, \mathbf{u}') = -\nabla p + \nu \nabla^2 \mathbf{u}',$$

where $\mathcal{L}$ is a linear operator, its adjoint form, or both. The evolution of the linearised Navier-Stokes operator, which evolves a solution from an initial state to a future time t, can be written as

$$u(t) = \mathcal{A}(t)u(0).$$

The adjoint evolution operator is denoted as $\mathcal{A}^*$. This section details the additional configuration options, in addition to the standard configuration options described earlier, relating to performing this task.

11.5.1 Solver Info

- `Eqtype`: sets the type of equation to solve. For linear stability analysis this must be set to

Equation Type	Dimensions	Projections	Algorithms
UnsteadyNavierStokes	2D, Quasi-3D	Continuous	VCS,DS

- `EvolutionOperator`: sets the choice of the evolution operator:
  - `Nonlinear` (standard non-linear Navier-Stokes equations).
  - `Direct` ($\mathcal{A}$ – linearised Navier-Stokes equations).
  - `Adjoint` ($\mathcal{A}^*$ – adjoint Navier-Stokes equations).
  - `TransientGrowth` ($\mathcal{A}^*\mathcal{A}$ – transient growth evolution operator).
- `Driver`: specifies the type of problem to be solved:
  - `Standard` (normal 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.
  - `MultipleModes` (stability analysis with multiple modes, `HomModesZ` sets number of modes).
  - `SingleMode` (BiGlobal Stability Analysis: full-complex mode. Overrides `HomModesZ` to 1.).
  - `HalfMode` (BiGlobal Stability Analysis: half-complex mode u.Re v.Re w.Im p.Re).

Note

For visualization of `Homogeneous` results with `FieldConvert` you can use `--output-points-hom-z` to set output number of modes to a desired value. To process results obtained with `HalfMode` you can convert to `SingleMode` using `FieldConvert` module `halfmodetofourier`.

11.5.2 Parameters

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

- `kdim`: sets the dimension of the Krylov subspace κ. Can be used with: `ModifiedArnoldi` and `Arpack`. Default value: 16.
- `evtol`: sets the tolerance of the iterative eigenvalue algorithm. Can be used with: `ModifiedArnoldi` and `Arpack`. Default value: 1×10^{-6}.
- `nvec`: sets the number of converged eigenvalues sought. Can be used with: `ModifiedArnoldi` and `Arpack`. Default value: 2.
- `nits`: sets the maximum number of Arnoldi iterations to attempt. Can be used with: `ModifiedArnoldi` and `Arpack`. Default value: 500.
- `realShift`: provide a real shift to the direct solver eigenvalue problem by the specified value to improve convergence. Can be used with: `Arpack` only.
- `imagShift`: provide an imaginary shift to the direct solver eigenvalue problem by the specified value to improve convergence. Can be used with: `Arpack` only.
- `LZ`: sets the length in the spanswise direction L_z. Can be used with `Homogeneous` set to `1D`. Default value: 1.
- `HomModesZ`: sets the number of planes in the homogeneous directions. Can be used with `Homogeneous` set to `1D` and `ModeType` set to `MultipleModes`.

- `N_start`: sets the start number of temporal slices for Floquet stability analysis. Default value: 0.
- `N_skip`: sets the number skip of temporal slices for Floquet stability analysis. Default value: 1.
- `N_slices`: sets the number of temporal slices for Floquet stability analysis. Files sequence `N_start, N_start + N_skip, ..., N_start + N_skip * (N_slices-1)` will be loaded.
- `BaseFlow_interporder`: sets the interpolation order of temporal slices for Floquet stability analysis. If `BaseFlow_interporder` < 2, the baseflow is taken as periodic and trigonometric functions are used for interpolation. It should be noted that the file `N_start + N_skip * N_slices` is at t = `period` and should not be loaded for the periodic case. If `BaseFlow_interporder` $>= 2$, the flow is taken as aperiodic and Lagrange interpolation is used. In this case, the file `N_start + N_skip * (N_slices-1)` is at t = `period` and should be loaded. Default value: 0.
- `period`: sets the time span (if `BaseFlow_interporder` $>= 2$) or period (if `BaseFlow_interporder` < 2) of the base flow. Default value: 1.

11.5.3 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.

If the `Driver` is `ModifiedArnoldi`, instead of using the whole flow field, a subdomain or a subgroup of variables can be selected to calculate eigenvalues.

The selected domains are defined by

```
<FUNCTION NAME="SelectEVCalcDomain0">
  <E VAR="C0" VALUE=" x" />
  <E VAR="C1" VALUE="-x+1." />
  <E VAR="C2" VALUE=" y+1.5" />
  <E VAR="C3" VALUE="-y+1.5" />
</FUNCTION>

<FUNCTION NAME="SelectEVCalcDomain1">
  <E VAR="C0" VALUE=" x" />
  <E VAR="C1" VALUE="-x+1." />
```

```
  <E VAR="C2" VALUE=" y+1.5" />
  <E VAR="C3" VALUE="-y+1.5" />
</FUNCTION>

...
```

The number in `SelectEVCalcDomain?` and `C?` should increase continuously from 0 to 9. Empty function will be ignored.

Each function `SelectEVCalcDomain?` selects elements whose center satisfies C0>0 and C1>0 and C2>0, and so on. The finally selected domain is the union of all `SelectEVCalcDomain?`s.

The selected variables are defined by

```
<FUNCTION NAME="SelectEVCalcVariables">
  <E VAR="u" VALUE="1" />
  <E VAR="v" VALUE="1" />
</FUNCTION>
```

Empty function will be ignored.

If `SelectEVCalcDomain?` or `SelectEVCalcVariables` is defined, both the original eigenvector and the masked eigenvector (with _masked) will be output.

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.

11.6 Session file configuration: Steady-state solver

In this section, we detail how to use the steady-state solver (that implements the selective frequency damping method, see Sec. 11.1.4). 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.

11.6.1 Execution of the classical steady-state solver

11.6.1.1 Solver Info

The definition of `Eqtype`, `TimeIntegrationScheme` and `Projection` is similar as what is explained in section 11.5.1. 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`).

11.6.1.2 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.

11.6.2 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. 11.5.

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
```

11.7 Session file configuration: Coordinate transformations

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.

11.7.1 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.

11.7.2 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.

11.7.3 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.

11.7.4 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.

11.7.5 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`.

11.8 Session file configuration: Adaptive polynomial order

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 Eq. (9.9) 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.

11.8.1 Solver Info

The use of the adaptive polynomial order procedure is enforced through the definition of the `Driver` which has to be `Adaptive`.

11.8.2 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.

11.8.3 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.

11.8.4 Restarting the simulation

The simulation can be restarted using the final distribution of polynomial orders obtained from the adaptive procedure by setting the expansions as

```
<EXPANSIONS>
  <F FILE="restartfile.fld" />
</EXPANSIONS>
```

note that this will only affect the polynomial order. The initial condition still needs to be set correctly, and does not need to come from the same file used for the expansions.

11.9 Advecting extra scalar fields

In some cases, it might be useful to advect scalar fields with the velocity obtained from the solution of the Navier-Stokes equation. For example, in study of mass transfer or heat transfer problems where getting analytical expression for advection velocity is not possible, the transport (advection-diffusion) equation needs to be solved along with the Navier-Stokes equation to get the scalar concentration or temperature distribution in the flow field.

In the input file, the extra field variables that are being advected need to be defined after the variables representing the velocity components. The pressure needs to be at the end of the list. For example, for a 2D simulation the expansions and variables would be defined as

```
<EXPANSIONS>
    <E COMPOSITE="C[0]" NUMMODES="5" FIELDS="u,v,c1,c2,p" TYPE="MODIFIED" />
</EXPANSIONS>

<VARIABLES>
    <V ID="0"> u  </V>
    <V ID="1"> v  </V>
    <V ID="2"> c1 </V>
    <V ID="3"> c2 </V>
    <V ID="4"> p  </V>
</VARIABLES>
```

where u, v are the velocity components, c1 and c2 are extra fields that are being advected and p is the pressure.

In addition, diffusion coefficients for each extra variable can be specified by adding a function `DiffusionCoefficient`

```
<FUNCTION NAME="DiffusionCoefficient">
    <E VAR="c1" VALUE="0.1" />
    <E VAR="c2" VALUE="0.01" />
</FUNCTION>
```

Boundary conditions for the extra fields are set up in the same way as the velocity and pressure

```
<BOUNDARYCONDITIONS>
  <REGION REF="0">
    <D VAR="u"  VALUE="0" />
    <D VAR="v"  VALUE="0" />
    <D VAR="c1" VALUE="1" />
    <D VAR="c2" VALUE="0" />
    <N VAR="p"  USERDEFINEDTYPE="H" VALUE="0" />
  </REGION>
</BOUNDARYCONDITIONS>
```

It should be noted that if the diffusion coefficient is too small, the transport equation becomes advection dominated. This leads to small grid spacing required to resolve all physical scales associated with the transport equation (the ratio of resolution required for transport to Navier Stokes equation scales with $Sc^{3/4}$, where Sc is the Schmidt number = kinematic viscosity/diffusion coefficient). Hence, small diffusion coefficient might lead to spurious oscillations if the mesh spacing is not small enough.

Next, we consider an example of an active scalar in the case of Rayleigh-Bénard convection (RBC) where the temperature (scalar) field is coupled with the momentum equations. RBC describes the motion of a layer of fluid sandwiched between two infinite parallel plates, separated by a distance, d, heated from the bottom and cooled from the top - a canonical fluid configuration of natural convection. The governing equations of RBC can be described by the non-dimensional Navier-Stokes equations with Boussinesq approximations given as,

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + Pr \nabla^2 \mathbf{u} + RaPr\theta \mathbf{j}, \tag{11.31a}$$

$$\frac{\partial \theta}{\partial t} + \mathbf{u} \cdot \nabla \theta = \nabla^2 \theta, \tag{11.31b}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{11.31c}$$

with the following boundary conditions,

$$\mathbf{u}|_{y=0,d} = 0, \quad \theta|_{y=0} = 1, \quad \theta|_{y=d} = 0, \tag{11.32}$$

where the $Ra, Pr, \theta, \mathbf{u}, p$ refers to the Rayleigh number, Prandtl number, non-dimensional temperature and velocities respectively. To setup a simulation for a 2D simulation of RBC, the expansions and variables are prescribed as,

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

<VARIABLES>
    <V ID="0"> u     </V>
    <V ID="1"> v     </V>
    <V ID="2"> theta </V>
    <V ID="3"> p     </V>
</VARIABLES>
```

The boundary conditions are set up in the same way as listing 11.9. Note that the boundary conditions on the left/right edges are periodic, and its setup is shown in section 3.4.6.3. Next, the diffusion coefficient for the momentum equations (equation 11.31a) is prescribed directly by setting `Kinvis` (within `PARAMETERS`) to the Prandtl number, i.e `Kinvis = Pr`. The diffusion coefficient for the non-dimensional temperature equation (equation 11.31b) in this case is 1, which is prescribed in `DiffusionCoefficient` as,

```
<FUNCTION NAME="DiffusionCoefficient">
    <E VAR="theta" VALUE="1" />
</FUNCTION>
```

The $Ra Pr \theta \mathbf{j}$ term refers to the temperature (scalar) field acting as a body force on the $v-$momentum equations (equation 11.31a), which is prescribed using the `BodyForce` function,

```
<FUNCTION NAME="BodyForce">
    <E VAR="u"     VALUE="0"                EVARS="u v theta" />
    <E VAR="v"     VALUE="Ra * Pr * theta"  EVARS="u v theta" />
    <E VAR="theta" VALUE="0"                EVARS="u v theta" />
</FUNCTION>
```

The onset of convection occurs above the critical Rayleigh number, $Ra_c = 1708$. The solution for $Ra = 2000, Pr = 1$, consisting of two pairs of counter-rotating convection rolls are shown below. Note that the wavelength of a pair of rolls is close to the critical wavelength i.e $\lambda_c \approx 2d$.

11.10 Imposing a constant flowrate

In some simulations, it may be desirable to drive the flow by fixing a value of the volumetric flux through a boundary surface. A common use case for this is a channel flow, where the inflow and outflow are treated using periodic boundary conditions, requiring a

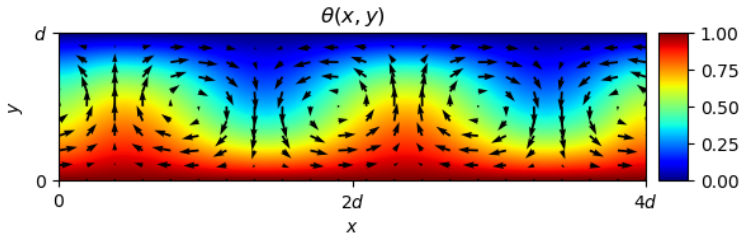


Figure 11.6 Convection rolls at $Ra = 2000$

use of either a body force or a flowrate condition to drive the flow. Often, the use of a body force is sufficient, but in some cases (e.g. transitional or turbulent simulations), it may be difficult to determine the correct body force to use in order to attain a specific Reynolds number. The incompressible solver supports the use of an alternative forcing whereby the volumetric flux,

$$Q(\mathbf{u}) = \frac{1}{\mu(R)} \int_R \mathbf{u} \cdot d\mathbf{s},$$

through a user-defined surface R of area $\mu(R)$ is kept constant. This is supported for standard two- and three-dimensional simulations, where R is a boundary region, as well as three-dimensional homogeneous simulations. In the latter case, the forcing can be imposed either in the homogeneous direction (in the $x - y$ plane) or perpendicular to it (in the z direction).

The flowrate correction works by taking each timestep's velocity field $\mathbf{u}^n$, and computing a scalar α so that the corrected flow

$$\tilde{\mathbf{u}}^n = \mathbf{u}^n + \alpha \mathbf{u}_s$$

has the desired flowrate. Here, $\mathbf{u}_s$ is a linear Stokes solution that is calculated once at the start of the simulation, so that the condition is not expensive to implement.

To enable flowrate corrections, three things must be defined in the session file:

- The `Flowrate` parameter in the parameters section, which defines the desired value of the volumetric flux $Q(\mathbf{u})$ through the reference region. To set a flux per unit surface of $Q = 1$ we would therefore define:

```
<PARAMETERS>
  <P> Flowrate = 1.0 </P>
</PARAMETERS>
```

- A boundary condition must be tagged with the `Flowrate` user-defined type to define the reference region R. For example, a 2D channel flow with periodic boundary conditions might use the following arrangement:

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

- a `FlowrateForce` function with components `ForceX`, `ForceY` and `ForceZ` that defines the direction in which the forcing will be applied. This should be a unit vector (i.e. of magnitude 1) and constant (i.e. not dependent on x, y, z or t). As an example, to impose a force in the x-direction we specify:

```
<FUNCTION NAME="FlowrateForce">
  <E VAR="ForceX" VALUE="1.0" />
  <E VAR="ForceY" VALUE="0.0" />
  <E VAR="ForceZ" VALUE="0.0" />
</FUNCTION>
```

Importantly, note that in homogeneous simulations where the forcing is in the z-direction only the `Flowrate` parameter should be specified, and the reference area R is taken to be the homogeneous plane.

Optionally, the `IO_FlowSteps` parameter can be defined. If set to a non-zero integer, this produces a file `session.prs` which records the value of α used in the flowrate correction, every `IO_FlowSteps` steps.

11.11 Examples

11.11.1 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.

11.11.1.1 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="8" FIELDS="u,v,p" TYPE="MODIFIED" />
</EXPANSIONS>
```

We next specify the time integration scheme and 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 time-integration scheme must be used and we choose one of second order.

```
<TIMEINTEGRATIONSCHEME>
   <METHOD> IMEX </METHOD>
   <ORDER> 2 </ORDER>
</TIMEINTEGRATIONSCHEME>

<SOLVERINFO>
   <I PROPERTY="EQTYPE"                VALUE="UnsteadyNavierStokes"     />
   <I PROPERTY="SolverType"            VALUE="VelocityCorrectionScheme" />
   <I PROPERTY="AdvectionForm"         VALUE="Convective"               />
   <I PROPERTY="Projection"            VALUE="Galerkin"                 />
</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>
```

11.11.1.2 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 11.7.

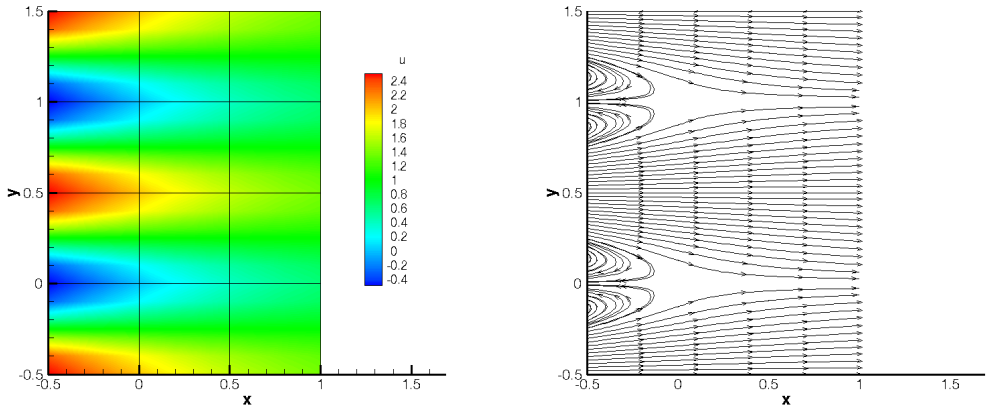


Figure 11.7 Velocity profiles for the Kovasznay Flow (2D).

11.11.2 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 [10]. 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>
```

11.11.2.1 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 11.8.

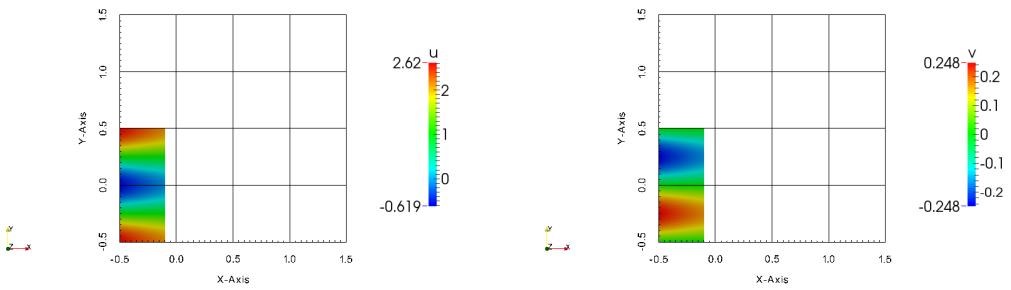


Figure 11.8 Velocity profiles for the Kovasznay Flow in truncated domain (2D).

11.11.3 Laminar Channel Flow 2D

In this example, we will simulate the flow through a channel at Reynolds number (Re) 1 with fixed boundary conditions.

11.11.3.1 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>
```

11.11.3.2 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 11.9.

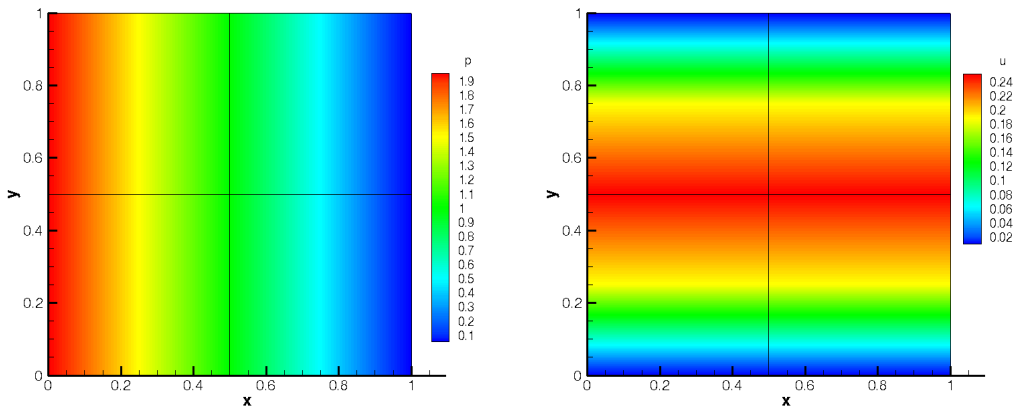


Figure 11.9 Pressure and velocity profiles for the laminar channel flow (2D).

11.11.4 Laminar Channel Flow 3D

We now solve the incompressible Navier-Stokes equations on a three-dimensional domain using all element types supported by *Nektar++* which include Hexahedrons, Prisms, Pyramids and Tetrahedrons. In particular, we solve the three-dimensional equivalent of the previous example. We will also run the IncNavierStokesolver solver in parallel.

Note

In order to run the example, you must have a version of *Nektar++* compiled with MPI and HDF5. This is the case for the packaged binary distributions.

Input file

The input file for this example is given in the directory

`$NEkHOME/solvers/IncNavierStokesolver/Examples/ChannelFlow3D`.

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 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>
```

```
35      <REGION REF="4">
36          <D VAR="u" VALUE="0" />
37          <D VAR="v" VALUE="0" />
38          <D VAR="w" VALUE="0" />
39          <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
40      </REGION>
41      <REGION REF="5">
42          <D VAR="u" VALUE="0" />
43          <D VAR="v" VALUE="0" />
44          <D VAR="w" VALUE="y*(1-y)" />
45          <N VAR="p" USERDEFINEDTYPE="H" VALUE="0" />
46      </REGION>
47  </BOUNDARYCONDITIONS>
```

Initial conditions and exact solutions are also prescribed.

Running the solver

To run the solver in parallel, we use the `mpirun` command.

```
mpirun -np 2 IncNaverStokesSolver ChannelFlow3D.xml
```

The expected results are shown in Figure 11.10.

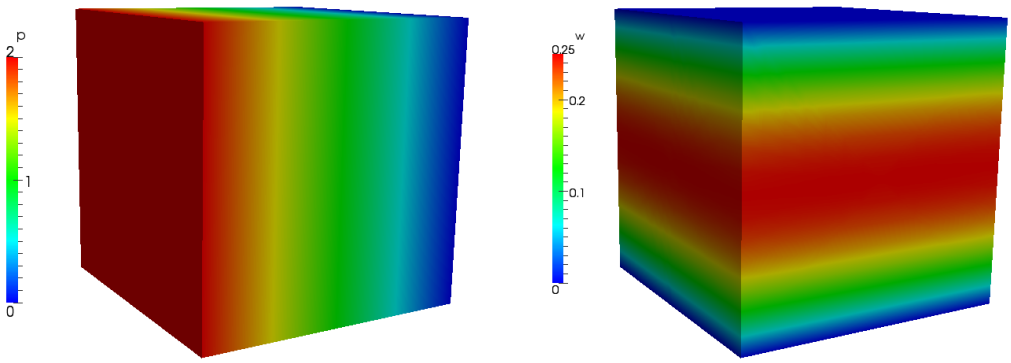


Figure 11.10 Pressure and velocity profiles for the laminar channel flow (full 3D).

11.11.5 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.

11.11.5.1 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

```
<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.

11.11.5.2 Running the solver

```
IncNaverStokesSolver ChannelFlowQuasi3D.xml
```

The results can be post-processed and should match those of the fully three-dimensional case as shown in Figure 11.10.

11.11.6 2D biglobal/direct stability analysis of the channel flow

In this example, we illustrate how to perform a biglobal or direct stability analysis using Nektar++ which determines the leading eigenvalues and eigenvectors of the the linearised Navier-Stokes equations:

$$\frac{\partial \mathbf{u}'}{\partial t} + \mathbf{U} \cdot \nabla \mathbf{u}' + \mathbf{u}' \cdot \nabla \mathbf{U} = -\nabla p + \nu \nabla^2 \mathbf{u}' + \mathbf{f} \tag{11.33a}$$

$$\nabla \cdot \mathbf{u}' = 0 \tag{11.33b}$$

In this appraoch we use a time stepping approached as part of an Arnoldi iterative scheme [7].

We 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 (Re) 7500. This problem is a particular case of the stability solver for the IncNavierStokesSolver.

An example input files can be found under:

`$NEkHOME/solvers/IncNavierStokesolver/Examples/ChannelStability`.

In this directory there are three files: the session and geometry file `ChannelStability.xml`, a base flow file `ChannelStbaility.bse` and a restart file `ChannelStability.rst`.

Solver Info

In this example the `EvolutionOperator` must be `Direct` to consider the linearised Navier-Stokes equations and the `Driver` is 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"         />
</SOLVERINFO>
```

Parameters

For a stability run we require the following parameters

```
<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>
```

Here the `TimeStep` and `NumSteps` define for how long we run the time stepping scheme to perform a forward/direct action of the linearised Navier-Stokes equation over a fixed time `TimeStep` x `NumSteps`. As previously we set the Reynolds number based on the kinematic viscosity, inflow velocity and channel height (the latter two are assumed to be one). The Arnoldi parameters are provide by `kdim` which defines the dimension of the Krylov space, `nvec` which defines the number of eigenvalues to converge to tolerance `evtol` and `nits` gives the maximum number of Arnoldi iterations.

Boundary Conditions

Similar to the Navier-Stokes equations we specify similar boundary conditions where typically the linearised problem is zero on most boundaries.

```
<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>

```

Base flow and initial conditions

For the linearised problem 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="ChannelStability.bse" />
        </FUNCTION>

```

As in previous examples we also can specify a restart file using the `InitialConditions` section

```
        <FUNCTION NAME="InitialConditions">
            <F VAR="u,v,p" FILE="ChannelStability.rst" />
        </FUNCTION>

```

Usage

As previously once we have set up the xml file appropriatel we can simply run:

```
IncNavierStokesSolver ChannelStability.xml
```

Output

The stability simulation takes about 250 iterations to converge and the dominant eigenvalues (together with the respective eigenvectors) will be visualised. In this case the files are given as

```
ChannelStability_eig_0.xml and ChannelStability_eig_1.xml
```

which can be post processed using `FieldConvert`.

The eigenvalues are provided in the file `ChannelStability.evl` and 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.

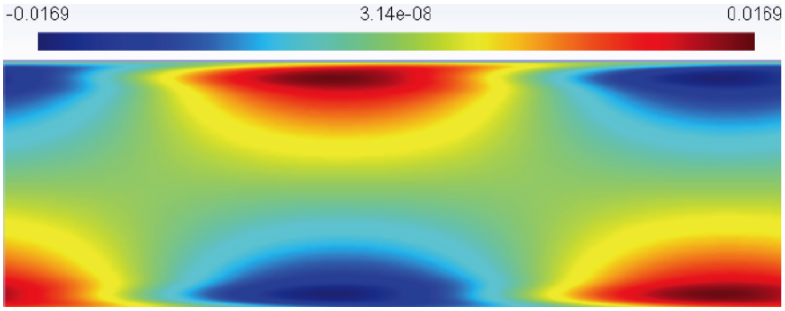


Figure 11.11 u-component of the leading eigenvalue of channel flow at $Re = 7500$

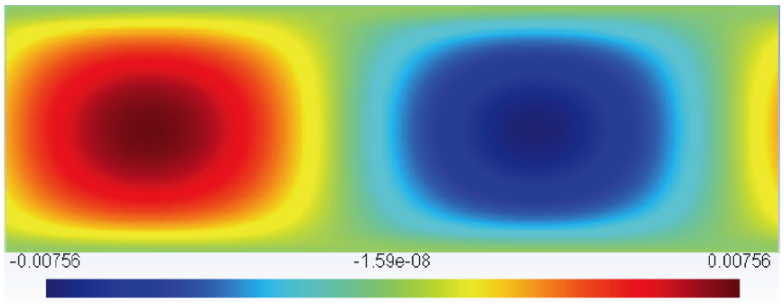


Figure 11.12 v-component of the leading eigenvalue of channel flow at $Re = 7500$

11.11.7 2D adjoint stability analysis of the channel flow

We consider in this example how to perform an adjoint stability analysis using Nektar++. We 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 (Re) 7500

The adjoint equations of the linearised Navier-Stokes equations can be written as

$$-\frac{\partial \mathbf{u}^*}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{u}^* + (\nabla\mathbf{U})^T\cdot\mathbf{u}^* = -\nabla p^* + \frac{1}{Re}\nabla^2\mathbf{u} \tag{11.34a}$$

$$\nabla\cdot\mathbf{u}^* = 0 \tag{11.34b}$$

We are interested in computing the leading eigenvalue of the system using the Arnoldi method [7].

An example input files can be found under:

`$NEkHOME/solvers/IncNavierStokesolver/Examples/ChannelStabilityAdjoint`.

In this directory there are three files: the session and geometry file `ChannelStabilityAdjoint.xml`, a base flow file `ChannelStbailityAdjoint.bse` and a restart file `ChannelStabilityAdjoint.rst`.

Solver Info

In this example the `EvolutionOperator` must be set to `Adjoint` to consider the adjoint Navier-Stokes equations defined above and the `Driver` was set up to `ModifiedArnoldi` for the solution of the eigenproblem, i.e.

```
<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"         />
</SOLVERINFO>
  \end{subequations}

\subsubsection*{Parameters}

Similar to the BiGlobal problem we specify the 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>

```

Here the `TimeStep` and `NumSteps` define for how long we run the time stepping scheme to the adjoint of the linearised Navier-Stokes equation over a fixed time `TimeStep` x `NumSteps`. We set the Reynolds number based on the kinematic viscosity, inflow velocity and channel height (the latter two are assumed to be one). The Arnoldi parameters are provide by `kdim` which defines the dimension of the Krylov space, `nvec` which defines the number of eigenvalues to converge to tolerance `evtol` and `nits` gives the maximum number of Arnoldi iterations.

Boundary Conditions

In this case we have a periodic domain and so we only have to worry about the adjoint variables on the walls which are set to zero.

```
<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>

```

Base flow and intiial conditions

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="ChanStabilityAdjoint.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="ChanStabilityAdjoint.rst" />
        </FUNCTION>

```

Usage

Execution is relatively straightforward once the input files are setup:

```
IncNavierStokesSolver ChannelStabilityAdjoint.xml
```

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. As in the biglobal stability problem the adjoint eigenvalues are given in the file `ChannelStabilityAjoint.evl` and the eigenvectors are provided in file `ChannelStabilityAdjoint_eig_0.fld` and `ChannelStabilityAdjoint_eig_1.fld`

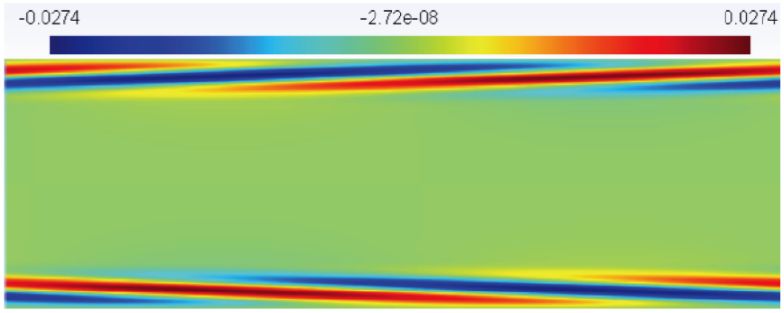


Figure 11.13 u-field of the leading adjoint mode in channel flow at $Re = 7500$

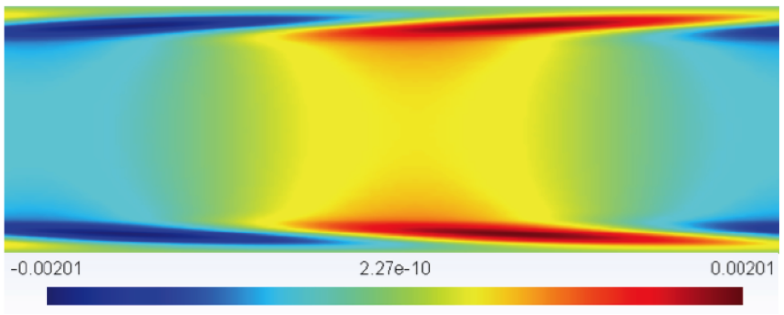


Figure 11.14 v-field of the leading adjoint mode in channel flow at $Re = 7500$

11.11.8 2D Transient Growth analysis of a flow past a backward-facing step

In this example we consider how to perform a transient growth stability analysis using Nektar++. Let us consider a flow past a backward-facing step (figure 11.15). 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.

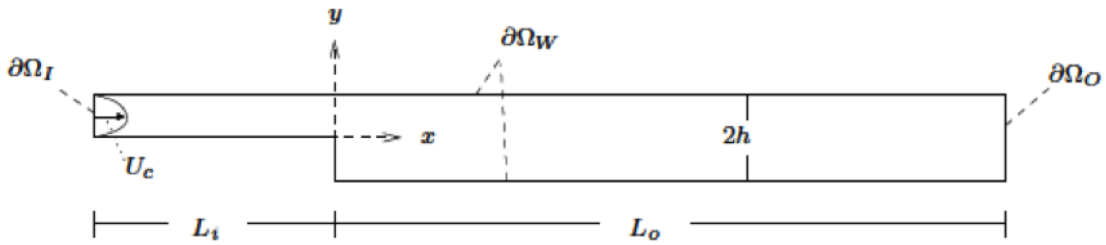


Figure 11.15

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 [7].

An example input files can be found under:

`$NEkHOME/solvers/IncNavierStokesolver/Examples/BackwardFacingStep_TG`.

In this directory there are three files: the session and geometry file `BackwardsFacingStep_TG.xml`, a base flow file `BackwardsFacingStep_TG.bse` and a restart file `BackwardsFacingStep_TG.rst`.

Solver Info

In this example the `EvolutionOperator` must be `TransientGrowth` and the `Driver` was set up to `Arpack` for the solution of the eigenproblem. This means the code msut be compiled wit the option NEKTAR_USE_ARPACK turned on. For this example the solver parameters are:

```
<SOLVERINFO>
    <I PROPERTY="EQTYPE"                VALUE="UnsteadyNavierStokes"     />
    <I PROPERTY="EvolutionOperator"     VALUE="TransientGrowth"          />
    <I PROPERTY="Projection"            VALUE="Galerkin"                 />
    <I PROPERTY="SOLVERTYPE"            VALUE="VelocityCorrectionScheme" />
    <I PROPERTY="Driver"                VALUE="Arpack"                   />
    <I PROPERTY="ArpackProblemType"     VALUE="LargestMag"               />
</SOLVERINFO>

```

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>

```

Here the `FinalTime` to define for how long we run the time stepping scheme to perform a forward/direct and backwards action of the linearised and adjoint Navier-Stokes equation over a fixed time `TimeStep` x `NumSteps`. As previously we set the Reynolds number based on the kinematic viscosity, inflow velocity and step height (the latter two are assumed to be one). The Arnoldi parameters are provide by `kdim` which defines the dimension of the Krylov space, `nvec` which defines the number of eigenvalues to converge to tolerance `evtol`.

Boundary Conditions

To satisy the boundary coditiosn for both the forward linearised and the backwards adjoint problems we set the otuflow and inflow conditiosn to zero Dirichlet. Similarly on the walls we have zeor Dirichlet 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>
```

```
24
```

Bsae flow and initial conditions

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.

```
1 <FUNCTION NAME="BaseFlow">
2             <F VAR="u,v,p" FILE="BackwardsFacingStep_TG.bse" />
3         </FUNCTION>
4
```

An initial guess can be specified in the `InitialConditions` functions and in this case is read from a file.

```
1 <FUNCTION NAME="InitialConditions">
2             <F VAR="u,v,p" FILE="BackwardsFacingStep_TG.rst" />
3 /FUNCTION>
4
```

Usage

Execution is relatively straightforward once the input files are setup:

```
IncNavierStokesSolver BackwardsFacingdtep_TG.xml
```

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.

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.

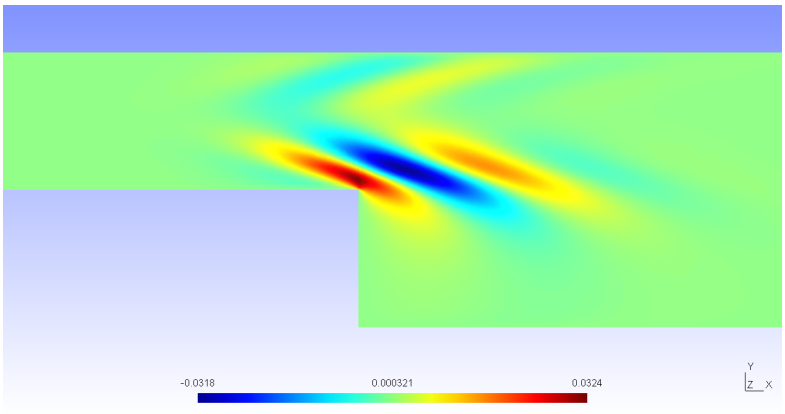


Figure 11.16

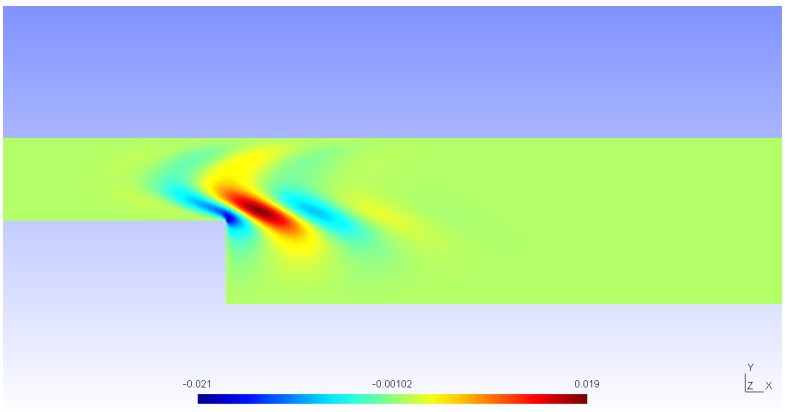


Figure 11.17

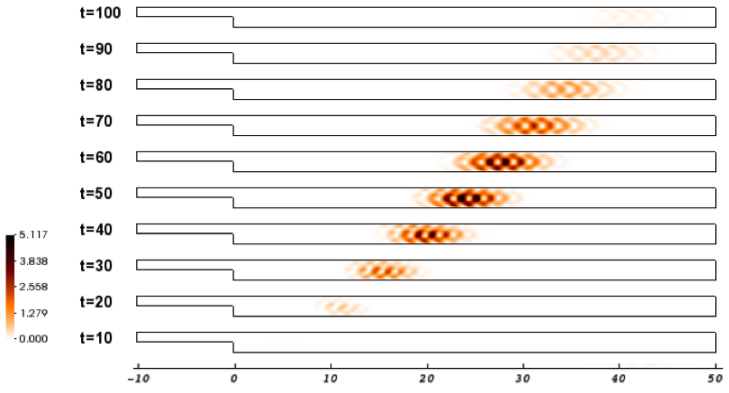


Figure 11.18

Chapter 12

Linear elasticity solver

12.1 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.

12.1.1 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

$$\nabla \cdot \mathbf{S} + \mathbf{f} = \mathbf{0} \quad \text{in} \quad \Omega \tag{12.1}$$

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

$$\mathbf{E} = \frac{1}{2}\left(\nabla \mathbf{u} + \nabla \mathbf{u}^t\right) \tag{12.2}$$

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.

$$\mathbf{u} = \hat{\mathbf{u}} \quad \text{in } \partial\Omega. \tag{12.3}$$

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.

12.2 Usage

```
LinearElasticSolver [arguments] session.xml [another.xml] ...
```

12.3 Session file configuration

12.3.1 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.

12.3.2 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.

12.4 Examples

12.4.1 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 [24]. 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.

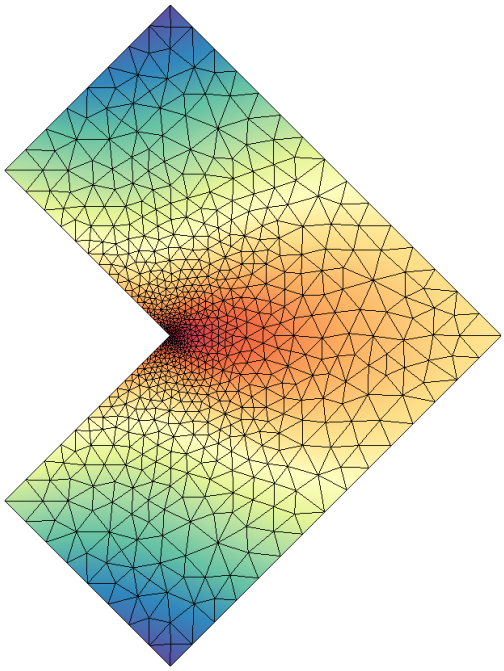


Figure 12.1 Solution of the u displacement field for the L-shaped domain.

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 12.1. 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
```

12.4.2 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 12.2.

The setup is very straightforward. The geometry can be found inside the file `Examples/bl-mesh.xml`

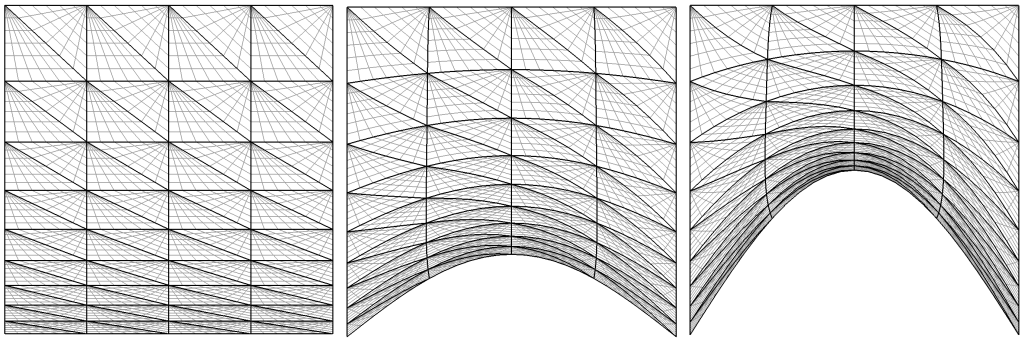


Figure 12.2 Figures that show the initial domain (left), after 50 steps (middle) and final deformation of the domain (right).

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 13

Pulse Wave Solver

13.1 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

$$\frac{\partial \mathbf{U}}{\partial t} + \mathbf{H}\frac{\partial \mathbf{U}}{\partial x} = \mathbf{S} \tag{13.1}$$

$$\mathbf{U} = \begin{bmatrix} U \\ A \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} U & A \\ \rho\frac{\partial P}{\partial A} & U \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} 0 \\ \frac{1}{\rho}\left(\frac{f}{A} - s\right) \end{bmatrix}$$

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 Eq. 13.1 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, viscoelastic tube this is given by

$$P = P_0 + \beta\left(\sqrt{A} - \sqrt{A_0}\right) + \frac{\Gamma}{\sqrt{A}}\frac{\partial A}{\partial t}, \quad \beta = \frac{\sqrt{\pi}Eh}{(1-\nu^2)A_0}, \quad \Gamma = \frac{2\sqrt{\pi}\varphi h}{3A_0} \tag{13.2}$$

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, ν is the Poisson's ratio, and φ is the wall viscosity. An empirical law has also been implemented that incorporates strain-stiffening through the parameter α [1]:

$$P = P_0 - \frac{\beta\sqrt{A_0}}{2\alpha} \ln\left[1 - \alpha \ln\left(\frac{A}{A_0}\right)\right] + \frac{\Gamma}{\sqrt{A}} \frac{\partial A}{\partial t}. \tag{13.3}$$

Application of Riemann's method of characteristics to Eqs. (13.1) and (13.2) 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.

13.2 Usage

To execute in serial one should type

```
PulseWaveSolver session.xml
```

the solver can also be run in parallel if compiled with MPI sing the command

```
mpirun -n 2 PulseWaveSolver session.xml
```

where in this example 2 processors would be used.

13.3 Session file configuration

13.3.1 Pulse Wave Solver 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.

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>
```

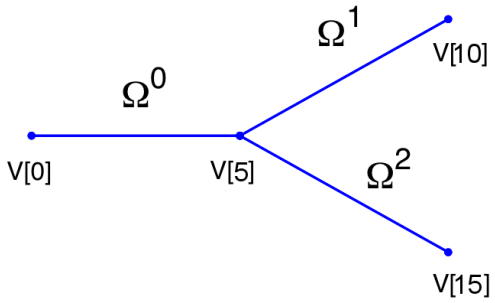


Figure 13.1 Model of bifurcating artery. The bifurcation is made of three domains and 15 vertices. Vertex V[0] is the inlet and vertices V[10] and V[15] the outlets.

```
  <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.

13.3.2 Time Integration Scheme

- `Method` the time-stepping method.
- `Variant` the variant to the method.
- `Order` the order of the method.
- `FreeParameters` any free parameters required.

13.3.3 Session Info

The PulseWaveSolver is specified 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.
- `UpwindTypePulse`:
  - `UpwindPulse`

- `OutputExtraFields`:
  - `True` returns the wave speed and both characteristics
- `PressureArea`:
  - `Beta` for Eq. (13.2)
  - `Empirical` for Eq. (13.3)

13.3.4 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 ;
- `pext` external pressure. Default value = 0;
- `pout` outflow pressure to the venous system for the terminal boundary conditions. Default value = 0;
- `h0` wall thickness. Default value = 1.0;

13.3.5 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 `BOUNDARYREGIONS`, 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.

13.3.5.0.1 Inlet boundary condition: The inlet condition may be specified algebraically in four different ways: as an area variation (`A-inflow`); a velocity profile (`U-inflow`); a volume flux (`Q-inflow`); or by prescribing the forward characteristic (`TimeDependent`). When prescribing a volume flux, it must be specified in the input file via the area, as illustrated below. Note that $u = 1.0$.

```
<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>
```

13.3.5.0.2 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

13.3.6 Functions

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

- `MaterialProperties`: specifies β for each domain.
- `A_0`: specifies A_0 for each domain as used in the tube law.

- `Viscoelasticity`: specifies Γ for each domain. Defaults to zero for every artery if not included.
- `StrainStiffening`: specifies α for each domain for Eq. (13.3). Defaults to 0.5 for every artery if not included.
- `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 (Eq. (13.2)).

13.4 Examples

13.4.1 Human Vascular Network

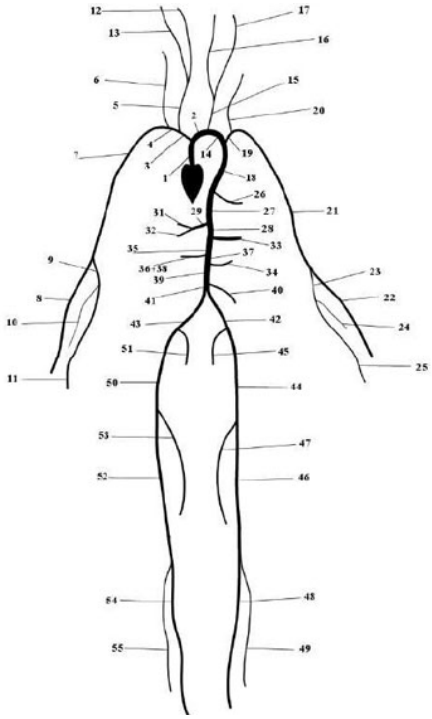
The PulseWaveSolver is also capable of handling more complex networks, such as a complete human arterial tree proposed by Westerhof et al. [50]. In this example, we will use the refined data from [44] and set up the network shown in the figure in the right. We will explain how bifurcations are set correctly 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.

```
<GEOMETRY DIM="1" SPACE="1">
    <VERTEX>
        <V ID="0"> 0.000e+00 0.000e+00 0.000e+00</V> <!-- 1 -->
```



#	Artery	Length (cm)	Area (cm^2)	β ($kg\,s^{-2}\,cm^{-3}$)	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

```
        <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>
          .
          .
```

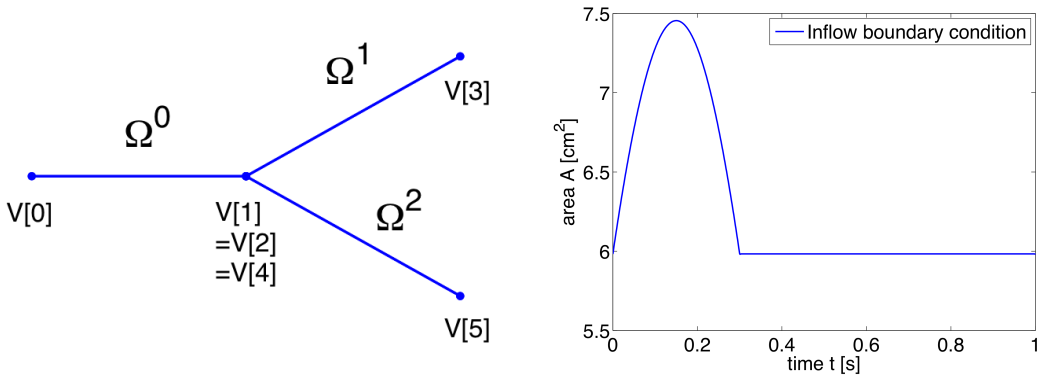
```
          .
        <S ID="106">   103    108 </S>
        <S ID="107">   108    109 </S>
        <S ID="108">   85    98 </S>
    <ELEMENT>
    <COMPOSITE>
        <C ID="0"> S[0] </C> <!-- 1 -->
        <C ID="1"> V[0] </C>
        <C ID="2"> V[1] </C>

        <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>

    <TIMEINTEGRATIONSCHEME>
        <METHOD> RungeKutta </METHOD>
        <VARIANT> SSP </VARIANT>
        <ORDER> 2 </ORDER>
```

```
    </TIMEINTEGRATIONSCHEME>

    <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 [44] 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" />
```

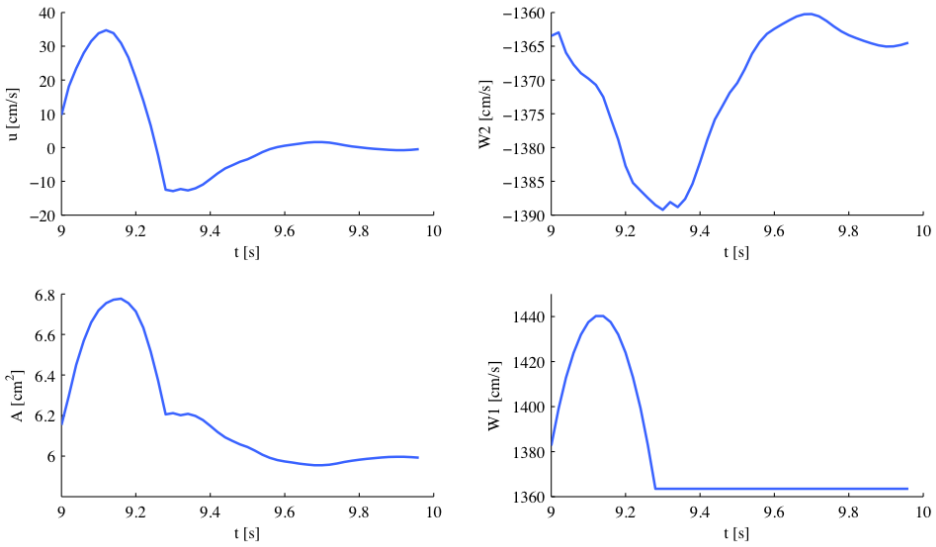
```
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
    <REGION REF="2">
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  VALUE="A8" />
        <D VAR="u" USERDEFINEDTYPE="TimeDependent"  VALUE="0.0" />
    </REGION>
    <REGION REF="3">
        <D VAR="A" USERDEFINEDTYPE="TimeDependent"  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.

13.4.2 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.

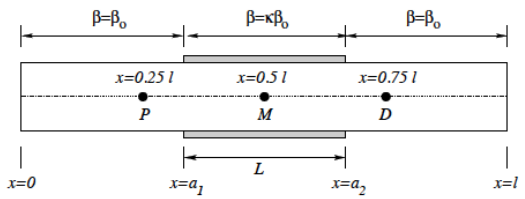


Figure 13.2 Model of straight artery with a stent in the middle.

13.4.2.0.1 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]).

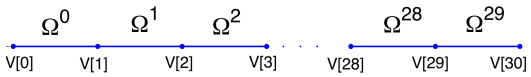


Figure 13.3 1D arterial domain consisting of 30 elements and 31 vertices.

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.

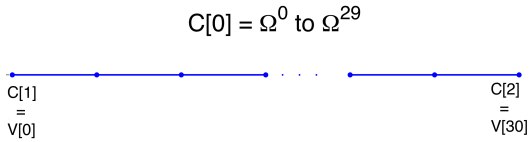


Figure 13.4 Three composites (C[0], C[1] and C[2]) for the stunted artery.

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>
```

13.4.2.0.2 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>
```

13.4.2.0.3 Time Integration Scheme: For the Time Integration Scheme we use a basic Forward Euler method.

```
<TIMEINTEGRATIONSCHEME>
  <METHOD> ForwardEuler </METHOD>
  <ORDER> 1 </ORDER>
</TIMEINTEGRATIONSCHEME>
```

13.4.2.0.4 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>
```

13.4.2.0.5 Parameters: Parameters used for the simulation are taken from [44]

```
<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>
```

13.4.2.0.6 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.

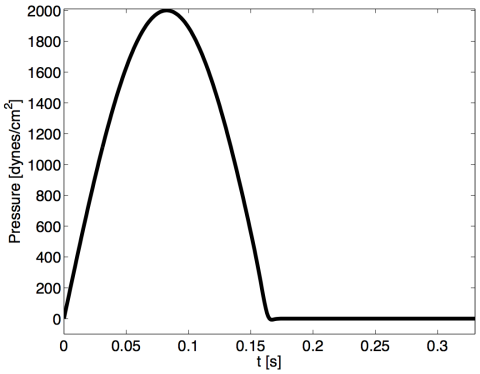


Figure 13.5 Pressure profile applied at the inlet of the artery

These are defined in the xml file as follows,

```
<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>
```

13.4.2.0.7 Functions: The stent is introduced by applying a variable material properties function (β - see Eq. (13.2)) along the vessel in the x direction, shown graphically below and defined in the xml file by

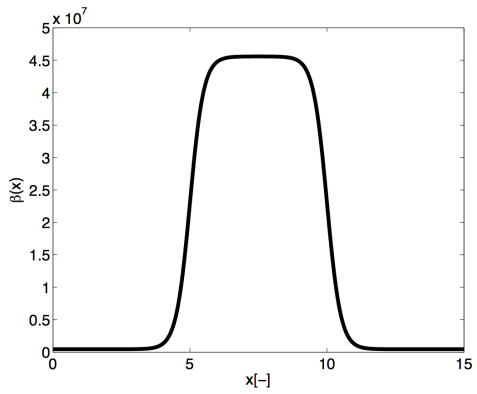


Figure 13.6 material property variation along the artery. The stiff region in the middle represents the stent.

```
<FUNCTION NAME="MaterialProperties">
        <E VAR="E0" DOMAIN="0" VALUE=
        "Y0*(1.0-kappa/(1+exp(-2*k*(a1-x)))+kappa/(1+exp(-2*k*(a2-x))))" />
</FUNCTION>
```

13.4.2.1 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.

13.4.2.2 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. 13.7).

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.

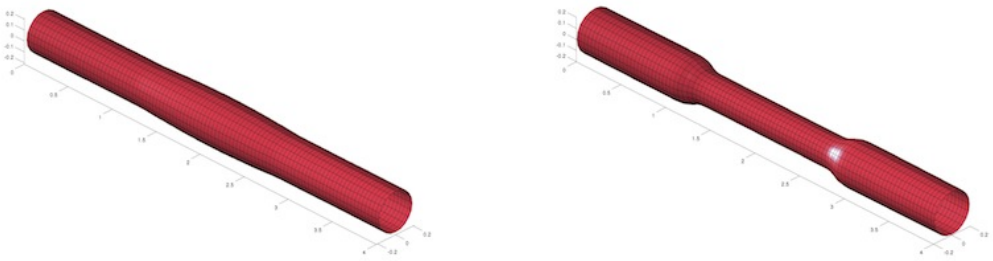
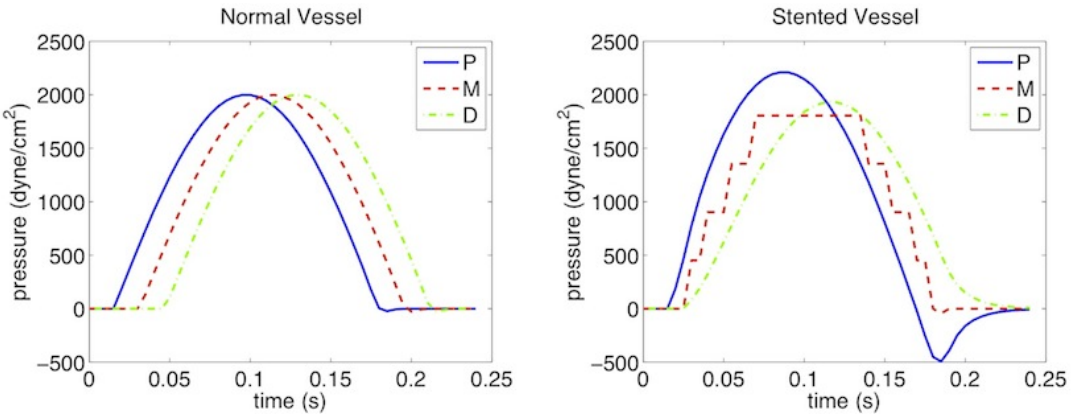


Figure 13.7



13.5 Further Information

The PulseWaveSolver has been developed with contributions by various students and researchers at the Department of Aeronautics, Imperial College London. Further information on the solver and its underlying mathematical framework can be found in [41, 40].

13.6 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.

13.7 References

[1] Reavette R, Sherwin SJ, Tang MX, Weinberg PW. Comparison of arterial wave intensity analysis by pressure-velocity and diameter-velocity methods in a virtual population of adult aubjects. *Journal of Engineering in Medicine*. 2020.

[2] Alastruey J. Numerical modelling of pulse wave propagation in the cardiovascular system: development, validation and clinical applications. *PhD thesis, Imperial College London*. 2006.

Chapter 14

Shallow Water Solver

14.1 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)

14.1.1 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.

14.2 Usage

```
ShallowWaterSolver session.xml
```

14.3 Session file configuration

14.3.1 Time Integration Scheme

- `Method` the time-stepping method.
- `Variant` the variant to the method.
- `Order` the order of the method.
- `FreeParameters` any free parameters required.

14.3.2 Solver Info

- `Eqtype`: Specifies the equation to solve. This should be set to `NonlinearSWE`.
- `UpwindType`
- `Projection`

14.3.3 Parameters

- `Gravity`

14.3.4 Functions

- `Coriolis`: Specifies the Coriolis force (variable name: ‘f‘)
- `WaterDepth`: Specifies the water depth (variable name: ‘d‘)

14.4 Examples

14.4.1 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.

14.4.1.1 Input Options

For what concern the ShallowWaterSolver the `<TIMEINTEGRATIONSCHEME>` and `<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:

```
<TIMEINTEGRATIONSCHEME>
   <METHOD> RungeKutta </METHOD>
   <ORDER> 4 </ORDER>
</TIMEINTEGRATIONSCHEME>
```

```
<SOLVERINFO>
  <I PROPERTY="EqType"                VALUE="NonlinearSWE"        />
  <I PROPERTY="Projection"            VALUE="DisContinuous"       />
  <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>
```

14.4.1.2 Running the code

After the input file has been copied to the build directory of the `ShallowWaterSolver` the code can be executed by:

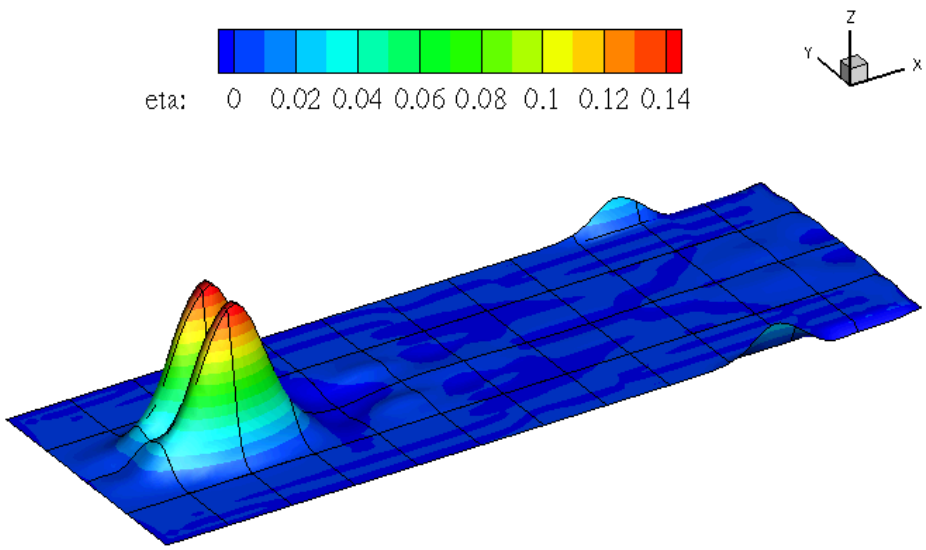
```
./ShallowWaterSolver Rossby_Nonlinear_DG.xml
```

14.4.1.3 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 15

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 section 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.

In the current version of the library we now attempt to turn on some of these optimisations automatically and so you will likely observe a `session.opt` file appear in your directory which can be viewed to see what settings are being selected.

15.1 Collections and MatrixFree operations

The Collections and associated MatrixFree libraries adds optimisations to perform certain elemental operations collectively by applying an operator using either matrix-matrix or unrolled matrix free operations, 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 (i.e. direction by direction) /operations.
- IterPerExp: Loop through elements, performing matrix-vector operation utilising StdRegions building blocks.

- MatrixFree: call matrix free implementations that can utilise vectorisation by performing SIMD (single instruction multiple data) operations over multiple elements concurrently.
- NoCollections: Use the original LocalRegions implementation to perform the operation which involves looping over the elements which may subsequently call the StdRegions implementations.

All configuration relating to Collections is given in the `COLLECTIONS` Xml element within the `NEKTAR` XML element.

15.1.1 Automatic tuning and the –write-opt-file command line option

By default we now try to select the optimal choice of implementation when you first run a solver. If you run the solver in verbose mode you will observe an output of the form:

```
Collection Implementation for Tetrahedron ( 4 4 4 ) for ngeoms = 428
           Op.    :    opt. Impl.       (IterLocExp,  IterStdExp,  StdMat,
    SumFac,      MatrixFree)
         BwdTrans:      MatFree          (0.000344303, 0.000336822, 0.000340444,
    0.000185503, 6.80494e-05)
         Helmholtz:     MatFree          (0.00227906, 0.00481378,      --    ,
    --    , 0.000374155)
         IPWrtBase:     MatFree          (0.000364424, 0.000318054, 0.000291705,
    0.000138584, 8.37257e-05)
         IPWrtDBase:    MatFree          (0.00378674, 0.00308545, 0.00100464,
    0.000653242, 0.000283372)
         PhysDeriv :    MatFree          (0.000881537, 0.000774604, 0.00407994,
    0.000540257, 0.000185529)
Collection Implemenation for Prism ( 4 4 4 ) for ngeoms = 136
           Op.    :    opt. Impl.       (IterLocExp,  IterStdExp,  StdMat,
    SumFac,      MatrixFree)
         BwdTrans:      MatFree          (0.000131559, 0.000130099, 0.000237854,
    8.40501e-05, 2.78436e-05)
         Helmholtz:     MatFree          (0.000988519, 0.00133484,      --    ,
     --    , 0.000166906)
         IPWrtBase:     MatFree          (0.000113946, 0.000105544, 0.00022007,
    5.74802e-05, 3.18842e-05)
         IPWrtDBase:    MatFree          (0.00148209, 0.000717362, 0.000885148,
    0.000257414, 0.00011241)
         PhysDeriv :    MatFree          (0.000295485, 0.000247841, 0.00186362,
    0.000219107, 7.38712e-05)
```

This shows the selected collection operation, in this case `MatrixFree`, for the different operators implmentations and the various approaches. Note that IterLocExp is equivalent to NoCollection and IterStdExp is directly related to the IterPerExp option.

This choice of optimisation is then written into a file called `Session.opt` where `Session` is name of the user defined xml file. We note that the optimal choice is currently based on the volumetric elements of the mesh (i.e. Tris and Quads in 2D and Tets, Pyramids, Prisms and Hexs in 3D) and not on the boundary conditions. In the case of a parallel

run the root process will write the file based on the optimisation on this processor. In the case one type of element is not on the root processor the output form the highest rank process with this element shape will be outputted. Once this file is present it will be read directly rather than re-running the auto-tuning. Even if an opt file already exist you can force a new one to be generated by using the `--write-opt-file` command line option.

15.1.2 Manually selecting the COLLECTIONS section

The `COLLECTIONS` section can be set manually within the `COLLECTIONS` tag as shown in the following example. Note this section can be added in either the input `Session.xml` file or the `Session.opt` file that is auto-generated.

Different implementations may be chosen for different element shapes and expansion orders. Specifying `*` for `ORDER` sets the default implementation for any expansion orders not explicitly 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" />
    </OPERATOR>
</COLLECTIONS>
```

15.1.2.1 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`, `NoCollection` or `Matrixfree`. The `StdMat` sets up a standard matrix for the element in the collection as the underlying operator. The following uses the collated matrix-matrix type elemental operation for all operators and expansion orders:

```
<COLLECTIONS DEFAULT="StdMat" />
```

The `NoCollection` option iterates over each expansion in the local region calling the local operator which is implemented in a sum factorization method within the element. The `IterPerExp` holds a standard expansion and then also holds an expanded copy of the geometric factors within the collection operator. `SumFac` is a sum factorization implementation which undertakes each direction of the method over multiple elements in the collection. Finally `MatrixFree` implements a vectorisation suitable version of the sum factorisation which has minimal memory movement but requires some initial data re-orientation when vectorising over multiple elements.

15.1.2.2 Auto-tuning

The choice of implementation for each operator, for the given mesh and expansion orders, can be selected selected automatically through an attribute in the `COLLECITON` section.

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.

15.1.3 Collection size

The maximum number of elements within a single collection can be enforced using the `MAXSIZE` attribute.

Chapter 16

Command-line Options

`--force-output, -f`
: Force checkpoint files and final field files to be written, even when they exist, and disable the creation of any backup files.

`--help`
: Displays help information about the available command-line options for the executable.

`--solverinfo [key]=[value]`
: Override a solverinfo (or define a new one) specified in the XML file.

`--parameter [key]=[value]`
: Override a parameter (or define a new one) specified in the XML file.

`--verbose`
: Displays extra info.

`--version`
: Displays software version, and source control information if applicable.

`--io-format [format]`
: Determines the output format for writing *Nektar++* field files that are used to store, for example, checkpoint and solution field files. The default for `format` is `Xml`, which is an XML-based format, which is written as one file per process. If *Nektar++* is compiled with HDF5 support, then an alternative option is `Hdf5`, which will write one file for all processes and can be more efficient for very large-scale parallel jobs.

`--no-exp-opt`
: This option will disable the optimisation of the expansion list ordering for collection type operations using the default definition of what is in the input

mesh file.

`--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.

`--npt [int]`
When using parallel-in-time integration (Parareal driver), specifies the number of chunks to use in the time 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.

`--write-opt-file`
This option forces the run to write a new optimsiation file by calling the auto-tuning option. Otherwise this is only done if a .opt file does not exist.

`--use-opt-file [file]`
Use the file provided with deifntiion of the collection optimisation for this run. By default the CI system runs using this option.

`--use-hdf5-node-comm`
Partition the `Hdf5`-format mesh in parallel to avoid one single thread runs out of memory in serial partitioning.

`--set-start-chknumber [int]`
Set the starting number of the checkpoint file. This overwrites the checkpoint number in the file-type initial condition.

`--set-start-time [float]`
Set the starting time of the simulation. This overwrites the time in the file-type initial condition.

`--use-metis`
: Forces the use of METIS for mesh partitioning. Requires the `NEKTAR_USE_METIS` option to be set.

`--use-scotch`
: Forces the use of Scotch for mesh partitioning. If *Nektar++* is compiled with METIS support, the default is to use METIS.

Chapter 17

Frequently Asked Questions

17.1 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 project, but they are
set to NOTFOUND. Please set them or make sure they are set and tested
correctly 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

17.2 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)

Q. When running my solver, I see files appearing with `.`bak extensions. What are these?

When running a Nektar++ solver, if a checkpoint or field file already exists with the same name as that to be written, by default a backup will be created where that directory is renamed. For example, if writing a file named `output.fld`, then the existing `output.fld` will be renamed to `output.bak0.fld`. If the solver were run again, then the `output.fld` this would be renamed to `output.bak1.fld` to avoid overwriting the `output.bak0.fld` file. This is a deliberate choice to avoid accidentally overwriting simulation data.

Both `NekMesh` and `FieldConvert` utilities will also prevent mesh and visualisation files being overwritten by default.

Q. How can I disable backup output?

Backup output is useful but can produce a large number of files, which can be an obstruction when developing with Nektar++ or debugging. There are two ways to disable backups:

- Pass the command line option `-force-output` or `-f`. This will turn off backup output for Nektar++ solvers, or the `NekMesh` and `FieldConvert` utilities.
- Set the environment variable `NEKTAR_DISABLE_BACKUPS`, for example by including the line:

  ```
  export NEKTAR_DISABLE_BACKUPS=1
  ```

  While this environment variable exists, it will prevent backups from being written.

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] Y. Bao, R. Palacios, M. Graham, and S.J. Sherwin. Generalized "thick" strip modelling for vortex-induced vibration of long flexible cylinders. *J. Comp. Phys*, 321:1079–1097, 2016.

[5] 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.

[6] 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.

[7] Barkley D, Blackburn HM, and Sherwin SJ. Direct optimal growth analysis for timesteppers. *International Journal for Numerical Methods in Fluids*, 57:1435–1458, 2008.

[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. A convective-like energy-stable open boundary condition for simulation of incompressible flows. *Journal of Computational Physics*, 302:300–328, 2015.

[10] 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.

[11] S. Dong and J. Shen. An unconditionally stable rotational velocity-correction scheme for incompressible flows. 229(19):7013–7029.

[12] F Ducros, V Ferrand, Franck Nicoud, C Weber, D Darracq, C Gacherieu, and Thierry Poinsot. Large-eddy simulation of the shock/turbulence interaction. *Journal of Computational Physics*, 152(2):517–549, 1999.

[13] 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.

[14] Roland Ewert and Wolfgang Schröder. Acoustic perturbation equations based on flow decomposition via source filtering. *Journal of Computational Physics*, 188(2):365–398, 7 2003.

[15] Paul F. Fischer. Projection techniques for iterative solution of ax = b with successive right-hand sides. *Computer Methods in Applied Mechanics and Engineering*, 163(1):193 – 204, 1998.

[16] Abel Gargallo-Peiró, Xevi Roca, Jaime Peraire, and Josep Sarrate. Distortion and quality measures for validating and generating high-order tetrahedral meshes. *Engineering with Computers*, 31(3):423–437, 2015.

[17] Georg Geiser, Holger Nawroth, Arash Hosseinzadeh, Feichi Zhang, Henning Bockhorn, Peter Habisreuther, Johannes Janicka, Christian O. Paschereit, and Wolfgang Schröder. Thermoacoustics of a turbulent premixed flame. In *20th AIAA/CEAS Aeroacoustics Conference*. American Institute of Aeronautics and Astronautics.

[18] David Gottlieb, Steven A Orszag, and CAMBRIDGE HYDRODYNAMICS INC MA. *Numerical analysis of spectral methods*. SIAM, 1977.

[19] 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.

[20] Anand N. Iyer and Richard A. Gray. An Experimentalist's Approach to Accurate Localization of Phase Singularities during Reentry. *Annals of Biomedical Engineering*, 29(1):47–59, January 2001.

[21] B. E. Jordi, C. J. Cotter, and S. J. Sherwin. Encapsulated formulation of the selective frequency damping method. *Phys. Fluids*, 2014.

[22] G. E. Karniadakis and S. J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford Science Publications, 2005.

[23] Robert M Kirby and Spencer J Sherwin. Stabilisation of spectral/hp element methods through spectral vanishing viscosity: Application to fluid mechanics modelling. *Computer methods in applied mechanics and engineering*, 195(23):3128–3144, 2006.

[24] Jonas Koko. Vectorized matlab codes for linear two-dimensional elasticity. *Scientific Programming*, 15(3):157–172, 2007.

[25] Kilian Lackhove. *Hybrid Noise Simulation for Enclosed Configurations*. Doctoral thesis, Technische Universität Darmstadt, 2018.

[26] 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.

[27] Xian Luo, Martin R. Maxey, and George Em Karniadakis. Smoothed profile method for particulate flows: Error analysis and simulations. *Journal of Computational Physics*, 228(5):1750–1769, 2009.

[28] 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.

[29] 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.

[30] Yvon Maday, Sidi M Ould Kaber, and Eitan Tadmor. Legendre pseudospectral viscosity method for nonlinear conservation laws. *SIAM Journal on Numerical Analysis*, 30(2):321–342, 1993.

[31] 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.

[32] RC Moura, SJ Sherwin, and Joaquim Peiró. Eigensolution analysis of spectral/hp continuous galerkin approximations to advection–diffusion problems: Insights into spectral vanishing viscosity. *Journal of Computational Physics*, 307:401–422, 2016.

[33] Rodrigo C Moura, Andrea Cassinelli, André FC da Silva, Erik Burman, and Spencer J Sherwin. Gradient jump penalty stabilisation of spectral/hp element discretisation for under-resolved turbulence simulations. *Computer Methods in Applied Mechanics and Engineering*, 388:114200, 2022.

[34] 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.

[35] 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.

[36] Yasuya Nakayama and Ryoichi Yamamoto. Simulation method to resolve hydrodynamic interactions in colloidal dispersions. *Phys. Rev. E*, 71:036707, Mar 2005.

[37] 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.

[38] 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.

[39] P.-O. Persson and J. Peraire. Sub-cell shock capturing for Discontinuous Galerkin methods. In *44th AIAA Aerospace Sciences Meeting and Exhibit*, page 112, 2006.

[40] N Pignier. One-dimensional modelling of blood flow in the cardiovascular system, 2012.

[41] CJ Roth. Pulse wave propagation in the human vascular system, 2012.

[42] S Sherwin. A substepping navier-stokes splitting scheme for spectral/hp element discretisations. pages 43–52. Elsevier Science, 2003.

[43] SJ Sherwin and M Ainsworth. Unsteady navier-stokes solvers using hybrid spectral/hp element methods. *APPLIED NUMERICAL MATHEMATICS*, 33:357–363, 2000.

[44] 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.

[45] 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.

[46] J. C. Simo and F. Armero. Unconditional stability and long-term behavior of transient algorithms for the incompressible navier-stokes and euler equations. 111(1):111–154.

[47] 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.

[48] M Turner, J Peiró, and D Moxey. A Variational Framework for High-Order Mesh Generation. In *25th International Meshing Roundtable*, volume 163, pages 340–352, 2016.

[49] Zhicheng Wang, Michael S Triantafyllou, Yiannis Constantinides, and George Em Karniadakis. A spectral-element/Fourier smoothed profile method for large-eddy simulations of complex VIV problems. *Computers & Fluids*, 172:84–96, 2018.

[50] N Westerhof. Anatomic studies of the human systemic arterial tree. *J. Biomech.*, 2:121–143, 1969.

[51] 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.

[52] Olgierd Cecil Zienkiewicz and Robert Leroy Taylor. *Basic formulation and linear problems*. McGraw-Hill, 1989.