Chapter 1

Download/installation instructions

1.1 The download/install process, step by step

1. Get the distribution
2. Install the distribution
   (a) Installation with `autogen.sh`
   (b) Configuration options

1. Finding your way through the distribution
2. External (third-party) libraries
   (a) External (third-party) libraries that are distributed with oomph-lib
      i. Specifying an existing, local copy of the blas library
      ii. Specifying an existing, local copy of the lapack library
   (b) External (third-party) libraries whose tar files are distributed with oomph-lib
   (c) External (third-party) libraries that are not distributed with oomph-lib
      i. Hypre
      ii. Trilinos
      iii. MUMPS and ScaLAPACK

1. How to write your own code and link it against oomph-lib's library/libraries
   (a) Writing/linking your own driver codes under autotools control
      i. Adding your own driver codes
      ii. Adding new libraries and linking against them from driver codes

1. Writing/linking user code without autotools: How do I treat `oomph-lib` simply as a library?

1.2 Get the distribution

oomph-lib is hosted on and distributed via GitHub at

https://github.com/oomph-lib/oomph-lib.

You should clone the repository onto your own computer, using

```
git clone git@github.com:oomph-lib/oomph-lib.git
```

This will check out all the sources into a new directory oomph-lib.
If you anticipate making contributions to oomph-lib, follow the instructions in our guide for contributors.

1.3 Install the distribution

1.3.1 Installation with \c autogen.sh

Change into the oomph-lib directory and run the autogen.sh build script:

```
cd oomph-lib
./autogen.sh
```

This build script will ask a few questions, e.g. to verify that the default build directory (build , relative to the oomph-lib home directory) is appropriate before starting the build.
By default, autogen.sh will build the library and the demo codes with certain default settings (using the gcc compilers with full optimisation, no debugging, no PARANOIA, and fully installed header files rather than symbolic links). These are appropriate if you wish to install the library once-and-for-all and do not anticipate any changes to their sources. The section Configuration options provides more details on the various options). autogen.sh will display the configure options and ask for confirmation that they are appropriate. If you are unsure if they are, simply hit return – the default will be fine.
The following flags for autogen.sh may be of interest:

- --jobs=n: Run the build process using n cores. This can greatly speed up build times and is strongly recommended if you have a multicore machine.
- --rebuild: Rebuild the configuration files from scratch. This is useful if you believe you may have somehow broken the build process, but should not be necessary normally.

Under the hood autogen.sh simply collects settings in a question and answer manner then calls a non-interactive script non_interactive_autogen.sh (yes, really!) with the appropriate flags. If you prefer you can simply call non_interactive_autogen.sh directly.
The self-tests can be initiated outside of autogen.sh using make check -k (to run on a single core) or ./bin/parallel_self_test.py (to run on all available cores). These commands will compile and run all the demo codes and verify their output. This is an optional step and can be very time consuming, especially if run on a small number of cores.

1.3.2 Configuration options

The build scripts allow you to specify a file of configure options. For instance, you may wish to specify another compiler, change the optimisation level, allow for debugging or range checking, etc. Previously used sets of options are stored in various files in the sub-directory config/configure_options/. The default settings are in default; the currently used ones are in current.
What options are there? You can get a complete list by typing

```
./configure --help
```

in oomph-lib's top level directory. Here are some options that we use frequently:

1.3.2.1 Suppressing the build of the documentation:

Building the online documentation locally is time consuming and requires a significant amount of disk space. Since the documentation is also available from the oomph-lib homepage you may wish to suppress this step. To achieve this, specify the configure option

```
--enable-suppress-doc
```

1.3.2.2 Suppressing the build of the pdf version of the documentation:

By default the tutorials are built as html files (which are best accessed by starting from the local copy of the `oomph-lib` homepage, which is in `doc/html/index.html`) and as pdf files (which are accessible via a link at the bottom of the relevant html-based tutorial). In the past we have sometimes had problems with `doxygen` (and hence the entire build process!) hanging while the pdf files are generated. To avoid this (by not even attempting to create the pdf files) specify the configure option

```
--enable-suppress-pdf-doc
```

(or better: update to a more recent version of `doxygen` – for instance version 1.8.6 works).

1.3.2.3 Replace library headers by symbolic links to the sources

During the build process `oomph-lib's` various libraries are installed in the subdirectory `build/lib` (or in whatever other directory you may have specified when asked to confirm their location) and the associated header files are copied to `build/include`. This is a sensible default for libraries that are only installed once and then never again tinkered with. Here the situation is slightly different: If you ever decide to add your own "user libraries" to `oomph-lib` (and you are encouraged to do so!), their header files will also be copied to `build/include`. If during code development, any of your header files contain syntax errors, the compiler will complain about the syntax errors in the copied file in `build/include` rather than the one in your source directory. This will encourage you to edit the copied file rather than the original – clearly a recipe for disaster! To avoid it we provide the configure option

```
 --enable-symbolic-links-for-headers
```

In this mode, the copies of the header files in `build/include` are replaced by symbolic links to the actual sources in `src` or `user_src`.

1.3.2.4 PARANOIA

`oomph-lib` provides an extensive range of optional run-time self-tests. The self-tests issue diagnostic error messages if any inconsistencies are detected at run-time and then terminate the code execution (semi-)gracefully by throwing an exception which (if not caught) aborts. This allows backtracking of the call sequence in a debugger during code development. Obviously, the self-tests introduce a slight run-time overhead and are therefore only performed if the C++ code is compiled with a special compiler flag, `PARANOID`. For gcc (and most other compilers we know) this is done by passing the flag `-DPARANOID` to the C++ compiler. This is achieved by adding

```
CXXFLAGS="-DPARANOID"
```

to the configure options. As discussed, during code development, this is most useful if debugging is also enabled, so the combination

```
CXXFLAGS="-g -DPARANOID"
```

is common.

1.3.2.5 Range checking

Most of the containers used in `oomph-lib` allow for optional (and very costly!) range checking which is enabled by specifying the C++ compiler flag `RANGE_CHECKING`. You are advised to recompile the code (yes, all of it!) if (and only if) you encounter some mysterious segmentation fault. Again, this is most useful if used together with debugging,

```
CXXFLAGS="-g -DRANGE_CHECKING"
```

(Without the `-g` flag you will only find out that an illegal index has been specified, not where this happened...)

1.4 Finding your way through the distribution

The `oomph-lib` distribution has several main sub-directories:

1.4.1 The src directory

The `src` sub-directory contains the source code for the various sub-libraries that make up `oomph-lib`. The most important one is the `generic` library which is built from the sources in the sub-directory `src/generic`. This library defines the fundamental `oomph-lib` objects (nodes, elements, meshes, timesteppers, linear and nonlinear solvers, mesh-adaptation routines, etc.)
The other libraries (`poisson`, `navier-stokes`,...) define elements for the solution of specific systems of PDEs. Finally, the `src/meshes` subdirectory contains several fully functional `Meshes` (and, where appropriate, the associated `Domains`) that are used in the demo codes. All `Meshes` in this directory are templated by the type

of element they contain. Since the element type can only be specified in the driver codes, the meshes cannot be compiled into libraries – the sources are always included as header-like files. Our list of example codes contains an example that explains this in more detail.

1.4.2 The external_src directory

The external_src directory contains "frozen" versions of various external libraries (e.g. SuperLU). Inclusion of these libraries into the distribution facilitates the overall build process: You only have to download and install a single distribution. This is much easier than finding out (typically halfway through the install process), that library A depends on library B which invariably turns out to depend on library C, etc.) Often the sources in the external_src subdirectories are sub-sets of the full libraries. For instance, we only include the double precision sources for SuperLU as neither the complex nor the single-precision versions are required within oomph-lib.

1.4.3 The external_distributions directory

oomph-lib provides interfaces to various third-party libraries which have their own build machinery. Some of these libraries are built by default in the course of the oomph-lib installation, using tar files that are distributed with oomph-lib; others will only be built if the user places the relevant tar files into the approproriate location within the oomph-lib directory structure.

1.4.3.1 External distributions that are built by default

By default oomph-lib builds CGAL, the Computational Geometry Algorithms Library, http://www.cgal.org. This library requires three other libraries which we also install:

- The GNU Multiple Precision Arithmetic Library (GMP), https://gmplib.org.
- The GNU MPFR Library https://gmplib.org.
- The Boost library, http://www.boost.org.

Note that the installation of these libraries is not quick. We therefore provide the option to (i) suppress their installation (in which case oomph-lib will employ a sub-optimal "locate_zeta" algorithm in its multi-domain algorithms) or to (ii) link against already existing installations of the libraries; see External (third-party) libraries whose tar files are distributed with oomph-lib for details.

1.4.3.2 External distributions that are not built by default

oomph-lib provides interfaces to various optional third-party libraries whose sources we deemed to be too big to be included in the oomph-lib distribution. If you wish to use these we expect you to install them yourself. To facilitate this task, we provide the option to let the oomph-lib build machinery perform the installation for you. If you place a copy of the tar file into the appropriate sub-directory in external_distributions, oomph-lib will build and install the library for you; see External (third-party) libraries that are not distributed with oomph-lib for details. (Note that oomph-lib is fully functional without these libraries – if the libraries are not available the build process ignores any oomph-lib code that depends on them.)

1.4.4 The demo_drivers directory

The demo_drivers directory contains a large number of demo codes. They are arranged in sub-directories, based on the type of the problem that is being solved. For instance, the demo_drivers/poisson subdirectory contains a number of demo problems involving the Poisson equation.
All sub-directories in demo_drivers contain shell scripts that validate the output from the demo codes by comparing the computed results against the reference results stored in the validata sub-directories. The comparison is performed with the python script bin/fpdiff.py which tolerates slight differences due to the unavoidable variations in roundoff error on different platforms and/or at different optimisation levels. The validation scripts can either be executed individually in each sub-directory or for all sub-directories by issuing the command make check. If the self-test is run at the top-level, a summary of the self-tests is stored in self_test/analyse_self_tests/validation.log

1.4.5 The doc directory

The structure of the doc directory (approximately) mirrors that of demo_drivers and contains the source code for the doxygen - based detailed explanation of the demo codes. If configure locates a sufficiently up-to-date version of doxygen on your system, the entire oomph-lib documentation will be built locally and can be navigated from the homepage in doc/html/index.html – a copy of the oomph-lib homepage.

1.4.6 The user_src and user_drivers directories

The configure script and the associated Makefiles that build and install the oomph-lib libraries and demo codes are generated by autoconf and automake. If these powerful tools are installed on your machine, you can include your own libraries and driver codes into the fully-automated oomph-lib build process. Store your code in suitably named sub-directories in user_src and user_drivers. See How to write your own code and link it against oomph-lib's library/libraries for more details.

1.5 External (third-party) libraries

oomph-lib provides interfaces to a number of third-party libraries. Those libraries that are essential for oomph-lib are distributed with the library to ensure that the user does not have to install these separately. We also provide interfaces to a number of third-party libraries that are not distributed with oomph-lib, typically because they are too big and/or take (too?) long to build. oomph-lib will only build the interfaces to these libraries if they are available and their location is specified during the configuration stage (or if the appropriate tar file is dropped into the required directory in which case oomph-lib's build process will build and install the library for you). In the latter case, the compiler flags used to build oomph-lib will be passed directly to the third-party libraries, so you may wish to compile the libraries separately if you wish to specify different compiler flags, e.g. no debugging information.

1.5.1 External (third-party) libraries that are distributed with oomph-lib

oomph-lib provides local copies of the following third-party libraries:

- BLAS
- LAPACK
- SuperLU
- METIS

By default oomph-lib automatically builds and links against these.

1.5.1.1 Specifying an existing, local copy of the blas library

If a local, possibly optimised version of the blas library already exists on your machine you can force oomph-lib to link against it and avoid the compilation of oomph-lib's own copy.
If your local copy of the blas library is located at
```
/home/mheil/local/lib/blas/blas.a
```
say, you can link against it by specifying the configure option
```
--with-blas=/home/mheil/local/lib/blas/blas.a
```

1.5.1.2 Specifying an existing, local copy of the lapack library

If a local, possibly optimised version of the lapack library already exists on your machine you can force oomph-lib to link against it and avoid the compilation of oomph-lib's own copy.
For instance, if your local copy of the lapack library is located at
```
/home/mheil/local/lib/lapack/lapack.a
```
you can link against it by specifying the configure option
```
--with-lapack=/home/mheil/local/lib/lapack/lapack.a
```

1.5.2 External (third-party) libraries whose tar files are distributed with oomph-lib

By default oomph-lib builds CGAL, the Computational Geometry Algorithms Library, http://www.cgal.org. This library requires three other libraries which we also install:

- The GNU Multiple Precision Arithmetic Library (GMP), https://gmplib.org.
- The GNU MPFR Library https://gmplib.org.
- The Boost library, http://www.boost.org.

These four libraries are built from tar files that we downloaded from the relevant webpages and then included into the oomph-lib distribution. We adopted this procedure to ensure that the versions of the libraries are consistent with each other.

1.5.2.1 Default CGAL installation.

By default the four libraries are installed within oomph-lib's external_distributions directory, and the paths to the relevant lib and include directories are propagated to oomph-lib's Makefile s. The installations are deleted by "make clean" or "make distclean", i.e. they are treated like any other oomph-lib code.

1.5.2.2 Installing CGAL in a permanent location

Given that the installation of the libraries takes a fair amount of time, we also provide the option to install them in a permanent location outside the oomph-lib directory structure. This is done by specifying the configure flag:

```
--with-cgal-permanent-installation-dir=ABSOLUTE_PATH_TO_PERMAMENT_INSTALL_DIRECTORY
```

where ABSOLUTE_PATH_TO_PERMAMENT_INSTALL_DIRECTORY specifies what it says. So, for instance, specifying

```
--with-cgal-permanent-installation-dir=/home/mheil/junk_default_installation
```

installs the libraries in /home/mheil/junk_default_installation. In subsequent rebuilds of oomph-lib it is then possible to specify the location of these libraries using configure options. Following an oomph-lib installation with -with-cgal-permanent-installation-dir these configure options are displayed at the end of the oomph-lib build procedure. (They are also contained in the file external_distributions/cgal_configure_flags.txt.) For instance, if the libraries have been installed in /home/mheil/junk_default_installation the relevant, the configure options are:

```
--with-boost=/home/mheil/junk_default_installation/boost_default_installation
--with-gmp=/home/mheil/junk_default_installation/gmp_default_installation
--with-mpfr=/home/mheil/junk_default_installation/mpfr_default_installation
--with-cgal=/home/mheil/junk_default_installation/cgal_default_installation
```

hierher auto?

1.5.2.3 Suppressing the CGAL installation

Finally, it is possible to suppress the installation of CGAL (and the related libraries) using the configure option

```
--enable-suppress-cgal-build
```

In this case oomph-lib will employ a sub-optimal "locate_zeta" algorithm in its multi-domain algorithms.

1.5.3 External (third-party) libraries that are not distributed with oomph-lib

Note: The third-party libraries discussed here are not installed by default but are built on demand if suitable tar files are placed in the relevant directories in the oomph-lib directory tree. You can download the tar files using the script

bin/get_external_distribution_tar_files.bash

or download them one-by-one using the links provided below.

1.5.3.1 Hypre

oomph-lib provides wrappers to the powerful solvers and preconditioners from the Scalable Linear Solvers Project. The wrappers are only built if Hypre is available on your machine. If your local copy of the Hypre library installed in

```
/home/mheil/local/hypre
```

i.e. if this directory contains Hypre's lib and include directories:

```
biowulf:~ 10:44:22$ ll /home/mheil/local/hypre
total 8
```

```
drwxr-xr-x    2 mheil    users        4096 Nov  3  2007 include
drwxr-xr-x    2 mheil    users        4096 Nov  3  2007 lib
```

you can get oomph-lib to link against it (and to compile oomph-lib's wrappers to Hypre's solvers and preconditioners) by specifying the configure option

```
--with-hypre=/home/mheil/local/hypre
```

Note: oomph-lib works with version 2.0.0 of the library. If this version of Hypre is not available on your machine download the tar file from the our own website:

hypre-2.0.0.tar.gz

You can either build the library yourself or get oomph-lib to build it for you. To do this simply place a copy of the tar file into the directory

```
external_distributions/hypre
```

and (re-)run autogen.sh. The installation procedure will detect the tar file, unpack it, and install the library in

```
external_distributions/hypre/hypre_default_installation
```

Unless you explicitly specified a library location using the -with-hypre flag, oomph-lib will then link against this newly created version of the library. However, we strongly recommend moving the newly created library to another place (outside the oomph-lib distribution) to preserve it for future use. Once this is done you simply specify the (new) location of the library with the -with-hypre flag, as discussed above. (Also make sure to delete the tar file from external_distributions/hypre, otherwise the library will be re-built.) Note that make clean will delete the unpacked Hypre sources but not the tar file and the library itself.

1.5.3.2 Trilinos

oomph-lib provides wrappers to the powerful solvers and preconditioners from the Trilinos Project. The wrappers are only built if Trilinos is available on your machine. If your local copy of the Trilinos library installed in

```
/home/mheil/local/trilinos
```

i.e. if this directory contains Trilinos's lib and include directories:

```
biowulf:~ 10:44:31$ ll /home/mheil/local/trilinos
total 24
drwxr-xr-x    2 mheil    users        4096 Dec 20 15:34 bin
drwxr-xr-x    2 mheil    users       16384 Dec 20 15:35 include
drwxr-xr-x    2 mheil    users        4096 Dec 20 15:35 lib
```

you can get oomph-lib to link against it (and to compile oomph-lib's wrappers to Trilinos's solvers and preconditioners) by specifying the configure option

```
--with-trilinos=/home/mheil/local/trilinos
```

Note: oomph-lib should work with major version numbers 9, 10 and 11 of the Trilinos library, and for revision numbers from 11 onwards you will need to have cmake installed on your machine. If these versions of Trilinos are not available on your machine you can get the latest version from the Trilinos web site or download a copy of the relevant tar file from our own website:

trilinos-11.8.1-Source.tar.gz

You can either build the library yourself or get oomph-lib to build it for you. To do this simply place a copy of the tar file into the directory

```
external_distributions/trilinos
```

and (re-)run autogen.sh. The build process is somewhat different for major version numbers 9 and 10, but oomph-lib will detect this automatically, provided that the source file is called trilinos-N.*.tar.gz, where N is the major version number. The installation procedure will then detect the tar file, unpack it, and install the library in

```
external_distributions/trilinos/trilinos_default_installation
```

Unless you explicitly specified a library location using the -with-trilinos flag, oomph-lib will then link against this newly created version of the library. However, we strongly recommend moving the newly created library to another place (outside the oomph-lib distribution) to preserve it for future use. Once this is done you simply specify the (new) location of the library with the -with-trilinos flag, as discussed above. (Also make sure to delete the tar file from external_distributions/trilinos, otherwise the library will be re-built.) Note that make clean will delete the unpacked Trilinos sources but not the tar file and the library itself.

1.5.3.3 MUMPS and ScaLAPACK

oomph-lib also provides wrappers to the MUMPS multifrontal solver, if it is available on your system. MUMPS needs the linear algebra library ScaLAPACK which must also be installed on your system. The configure options

```
--with-mumps=/opt/mumps
--with-scalapack=/opt/scalapack
```

will compile oomph-lib's wrappers and link against the MUMPS solver provided that MUMPS and ScaLAPACK are installed in the directories
```
/opt/mumps
/opt/scalapack
```
i.e. these directories contain the lib and include directories that result from successful installations of MUMPS and ScaLAPACK, respectively.
If you do not have MUMPS available you can download the latest version here. You can build and install the library yourself, or get oomph-lib to build it during part of its own build process. Simply place a copy of the tar file MUMPS_4.10.0.tar.gz in the directory
```
external_distributions/mumps_and_scalapack
```
You will also need to download the scalapack_installer.tgz from here, and place a copy in the same directory
```
external_distributions/mumps_and_scalapack
```
You can also download both files from our own website:

MUMPS_4.10.0.tar.gz

scalapack_installer.tgz

Note that the configure option
```
--with-mpi-include-directory=/usr/lib/openmpi/include
```
must be specified in order to build MUMPS, where /usr/lib/openmpi/include is the directory that contains the file mpi.h. [You can use locate mpi.h, to, well, locate that directory.]
Once the files have been placed in the external_distributions/mumps_and_scalapack directory, simply (re-)run autogen.sh, which will detect and build the libraries and install then in directory
```
external_distributions/mumps_and_scalapack/mumps_and_scalapack_default_installation
```
Note that the installation of ScaLAPACK requires an active Internet connection because it automatically downloads additional files.
Unless you explicitly specified a library location using the -with-mumps and -with-scalapack flags, oomph-lib will then link against the newly created versions of the libraries. However, we strongly recommend moving the libraries outside the oomph-lib distribution to preserve them for future use. Once this is done you simply specify the (new) location of the library with the -with-mumps and -with-scalapack flags, as discussed above. (Also make sure to delete the tar files from external_distributions/mumps_and_scalapack, otherwise the libraries will be built again.) Note that make clean will delete the unpacked sources but not the tar files nor the installed libraries.

1.6 How to write your own code and link it against oomph-lib's library/libraries

If you followed the instructions so far, you will be able to install oomph-lib and run the demo codes that are provided in the demo_drivers directory. Great! Now on to the next step: How do you write your own codes and link them against oomph-lib? There are two options, depending on whether you have (or are willing to install) the gnu autotools autoconf, automake and libtool on your machine.

1.6.1 Writing/linking your own driver codes under autotools control

1.6.1.1 Adding your own driver codes

Let's start with the straightforward case: You want to use oomph-lib to solve one of your own problems. To do this within oomph-lib's autotools framework, simply create a new directory in user_drivers and write your driver code. To facilitate these steps, the user_drivers directory already contains a sample directory joe_cool for which all these steps have been performed. If you don't object to the directory name (or if your name is Joe Cool) you can simply work in that directory. If not, we suggest the following sequence of steps:

1. Go to the user_drivers directory and create a new directory, e.g.
   ```
   cd user_drivers
   mkdir josephine_cool
   ```
2. Copy the Makefile.am and the driver code joes_poisson_code.cc from user_drivers/joe_cool to user_drivers/josephine_cool.
3. Return to oomph-lib's top-level directory and re-run ./autogen.sh to generate the required Makefile etc. in your own directory.

4. You may now return to your own directory in `user_drivers` and make your own driver code:
```
cd user_drivers/josephine_cool
make
```
This will create the required executable.

5. Unfortunately, the driver code (copied from Joe Cool's directory!) is unlikely to be the one you want but you can now rename it, edit it, or add further driver codes to your directory. In general we suggest that you have a look at the list of example codes and try to identify a problem that is similar to the one you want to solve. The associated driver code will be a good starting point for your own. Note that whenever you add new driver codes or rename existing ones you will have to update the local `Makefile.am`, though it is not necessary to re-run `autogen.sh`. The sample `Makefile.am` copied from Joe Cool's directory is well annotated and gives clear instructions how to adapt its contents:

```
# Name of executables: The prefix "noinst" means the executables don't
# have to be installed anywhere.
noinst_PROGRAMS= joes_poisson_code
#------------------------------------------------------------------------
# Local sources that Joe's Poisson code depends on:
joes_poisson_code_SOURCES = joes_poisson_code.cc
# Required libraries: Only the "generic" and "poisson" libraries,
# which are accessible via the general library directory which
# we specify with -L. The generic sources also require the "external" libraries
# that are shipped with oomph-lib. The Fortran libraries, $(FLIBS), get
# included just in case we decide to use a solver that involves Fortran
# sources.
# NOTE: The order in which libraries are specified is important!
#       The most specific ones (that might involve references to other
#       libraries) need to be listed first, the more basic ones
#       later. In this example we have (from right to left, i.e. from
#       general to specific):
#       -- The fortran libraries: They  are compiler specific and
#          obviously can't depend on any code that we (or others)
#          have written. $(FLIBS) is a variable that automake will translate
#          to the actual fortran libraries.
#       -- The external (third party) libraries: They cannot depend on
#          any of our code. The variable $(EXTERNAL_LIBS) is defined
#          in the machine-generated file configure.ac in oomph-lib's
#          home directory.
#       -- Oomph-lib's generic library contains oomph-lib's fundamental
#          objects which do not depend on any specific system of PDEs
#          or element types.
#       -- Finally, oomph-lib's poisson library contains oomph-lib's
#          Poisson elements which refer to objects from the generic
#          library.
joes_poisson_code_LDADD = -L@libdir@ -lpoisson -lgeneric $(EXTERNAL_LIBS) $(FLIBS)
#------------------------------------------------------------------------
# Include path for library headers: All library headers live in
# the include directory which we specify with -I
# Automake will replace the variable @includedir@ with the actual
# include directory.
AM_CPPFLAGS += -I@includedir@
```

Note the following points:

- Lines that start with a "#" are comments.
- The first (non-comment) line in the above file specifies the name(s) of the executable(s) that will be created by make. These names must be the same as those in the `*_SOURCES` and `*_LDADD` variables.
- The `joes_poisson_code_SOURCES` variable declares which (local) sources your executable depends on. In the current example there is only a single file, the driver code, `joes_poisson_code.cc`, itself.

- The joes_poisson_code_LDADD variable declares:
  - the location of the library directory (automake will convert the macro -L@libdir@ into the actual directory – you don't have to change this!).
  - the libraries (oomph-lib or otherwise) that you wish to link against. This is done with the usual -l flag that you will be familiar with from your compiler. Have a look at the comments regarding the order of the libraries!

- The INCLUDES variable specifies where to find the include header files. This line is again completely generic – automake will convert the macro -I@includedir@ into the actual location.

- If you have multiple driver codes, add the name of all executables to the noinst_PROGRAMS variable, and specify the *_SOURCES and *_LDADD variables for each one.

- The INCLUDES variable should only be specified once.

It makes sense to create a separate GitHub repository for your user driver directory. This won't interfere with the forked/cloned oomph-lib repository that you're working within. As far as the oomph-lib repository is concerned your user driver directory is simply one of possibly many directories that it's not tracking. Similarly, from within your user driver directory, the files outside it are not tracked by your repository, so the two can happily co-exist.

1.6.1.2 Adding new libraries and linking against them from driver codes

The above instructions should be sufficient to get you started. You can create multiple sub-directories for different projects and each sub-directory may, of course, contain multiple files, separated into header and source files. automake will ensure that only those files that have been changed will be recompiled when you issue the make command. However, at some point you may wish to package some of your sources into your own library and maybe even offer it for permanent inclusion into oomph-lib. For this purpose the oomph-lib distribution provides the sub-directory user_src which closely mirrors that of the src and external_src directories discussed earlier. During the build process, each sub-directory in user_src is compiled into its own library and installed in the standard location.
The steps required to include your own library into the oomph-lib build process are very similar to those required to add additional user drivers. As before, the user_src directory already contains a sample directory jack_↩cool, to facilitate the procedure. We therefore suggest the following sequence of steps:

1. Go to the user_src directory and create a new directory, e.g.
   ```
   cd user_src
   mkdir jacqueline_cool
   ```

2. Copy the Makefile.am and the codes hello_world.cc and hello_world.h from user_↩src/jack_cool to user_src/jacqueline_cool.

3. Return to the top-level oomph-lib directory and re-run autogen.sh.

4. You may now return to your own directory in user_src and make and install our own library
   ```
   cd user_src/jacqueline_cool
   make
   make install
   ```
   This will create the library and install it in build/lib

   The Makefile.am for libraries is slightly more complicated (though reasonably well documented) so – for now – we'll just list it here and hope that the changes required to include additional sources are obvious. If you really can't figure it out, send us an email and prompt us to complete this bit of the documentation....

```
# A few file definitions
#----------------------
# Define the sources
sources =  \
hello_world.cc
# Define the headers
headers =  \
hello_world.h
# Define name of library
libname = jack_cool
# Combine headers and sources
headers_and_sources = $(headers) $(sources)
# Define the library that gets placed in lib directory
#-----------------------------------------------------
lib_LTLIBRARIES = libjack_cool.la
# Sources that the library depends on:
#-------------------------------------
libjack_cool_la_SOURCES = $(headers_and_sources)
# The library's include headers:
#-------------------------------
# Headers that are to be included in the $(includedir) directory:
# This is the combined header which contains "#include<...>" commands
# for the real headers in the subdirectory below $(includedir)
include_HEADERS = $(libname).h
#Here's the subdirectory where the actual header files are placed
library_includedir=$(includedir)/jack_cool
#These are the header files that are to be placed in subdirectory
library_include_HEADERS=$(headers)
# Required libraries -- [assuming that we want to link against stuff in generic
#-------------------     add other oomph-lib libraries if you need them....]
# Include path for library headers -- need to refer to include files
# in their respective source directories as they will not have been
# installed yet!
AM_CPPFLAGS += -I$(top_builddir)/src/generic
# Combined header file
#---------------------
# Rule for building combined header (dummy target never gets made so
# the combined header is remade every time)
$(libname).h: dummy_$(libname).h
dummy_$(libname).h:  $(headers)
        echo  $(libname) $(headers) > all_$(libname).aux
        $(AWK) -f $(top_builddir)/bin/headers.awk < \
               all_$(libname).aux > $(libname).h
        rm all_$(libname).aux
# Extra hook for install: Optionally replace headers by symbolic links
#---------------------------------------------------------------------
if SYMBOLIC_LINKS_FOR_HEADERS
install-data-hook:
        (cd $(library_includedir) && rm -f $(headers) )
        (echo "$(headers)" > include_files.list )
        ($(top_builddir)/bin/change_headers_to_links.sh `pwd`)
        ($(LN_S) `cat include_files.list.aux` $(library_includedir) )
        (rm -r include_files.list.aux include_files.list )
else
install-data-hook:
endif
# Tidy up
#--------
clean-local:
        rm -f $(libname).h
```

Note that the directory `user_drivers/jack_cool` contains an example of a user driver code (`jacks_own_code` .cc) that uses a user library.

1.6.2 Writing/linking user code without autotools: How do I treat \c oomph-lib simply as a library?

Linking directly against `oomph-lib's` (sub-)libraries is slightly complicated by cross-compilation issues arising from the fact that the `oomph-lib` distribution includes a few C and Fortran sources. When linking is done (by the C++ compiler) one usually has to explicitly specify a few compiler-specific Fortran libraries. The beauty of the `autotools` approach described above is that these libraries (and any other flags that need to be passed to the compiler/linker) are determined and specified automatically. Doing this manually is no fun! Have a look at Mike Gerdts's excellent document `"How gcc really works"` for details.
The good news is that `oomph-lib's` installation procedure automatically generates a sample `Makefile` that contains all the relevant information. Once the installation is complete, the sample makefile is located at
`demo_drivers/linking/makefile.sample`
Here is the version that was generated one of our machines:

```
############################################################
# Automatically-generated sample makefile to illustrate how to
# link against oomph-lib from outside the automake/autoconf
# framework. Do not edit this -- make a copy first
#
# When customising this makefile, you should only have to change
#
# - the variable OOMPH-LIB_LIBS:
#         Add any additional oomph-lib sub-libraries that
#         you may wish to use in your code.
#
# - the specific dependencies for your driver code:
#         Include any additional local dependencies such as
#         your own header files etc.
#
############################################################


# Installation-specific information -- don't change any of this!
#-----------------------------------------------------------

# Flags for C pre-processor
AM_CPPFLAGS=-DHAVE_CONFIG_H -I. -I../..  -DOOMPH_HAS_MPI -I/home/mheil/version_for_release/build/include

# Library include directory: This is where all the header files live
OOMPH-LIB_INCLUDE_DIR=/home/mheil/version_for_release/build/include

# Library directory: This is where all of oomph-lib's sub-libraries live
OOMPH-LIB_LIB_DIR=/home/mheil/version_for_release/build/lib

# These are the external (3rd party) libraries that are distributed
# with oomph-lib and that we always link against
OOMPH-LIB_EXTERNAL_LIBS=-loomph_hsl -loomph_superlu_3.0 -loomph_metis_4.0 -loomph_arpack
       -loomph_superlu_dist_2.0 /home/mheil/local/lib/lapack/lapack.a /home/mheil/local/lib/blas/blas.a

# This specifies where libraries built from third party
# distributions can be found
EXTERNAL_DIST_LIBRARIES=

# This is additional machine-specific linking information that
# allows mixed-language compilation/linking
FLIBS=-L/usr/lib/lam/lib -L/usr/lib/gcc/i486-linux-gnu/4.3.3
       -L/usr/lib/gcc/i486-linux-gnu/4.3.3/../../../../lib -L/lib/../lib -L/usr/lib/../lib
       -L/usr/lib/gcc/i486-linux-gnu/4.3.3/../../.. -llammpio -llamf77mpi -lmpi -llam -lutil -ldl
       -lgfortranbegin -lgfortran -lm -lpthread

# Flags required for the use of shared libraries
SHARED_LIBRARY_FLAGS=-Wl,--rpath -Wl,/home/mheil/version_for_release/build/lib

#Mac OSX: Replace the above line with the following
#SHARED_LIBRARY_FLAGS= --rpath=/home/mheil/version_for_release/build/lib
# Problem-specific information -- edit this for your driver code

#-----------------------------------------------------------
# These are the specific oomph-lib sub-libraries that we have to link against
# for this driver code -- edit this according to your requirements
# but remember that the order of the libraries matters: List the
# the more specific ones before the more general ones!
OOMPH-LIB_LIBS=-lpoisson -lgeneric

# Dependencies for this driver code and compile instructions:
# Which local source (usually *.cc or *.h) files does the
# driver code depend on?
my_demo_code.o: demo_code.cc
         mpic++ $(AM_CPPFLAGS) -c demo_code.cc -o my_demo_code.o \
               -I$(OOMPH-LIB_INCLUDE_DIR)

# Linking instructions: Just declare the target (i.e. the name of the executable)
# and the dependencies (i.e. the object files created above). The rest
# should not have to be changed.
my_demo_code: my_demo_code.o
         mpic++ $(SHARED_LIBRARY_FLAGS) $< -o $@ \
               -L$(OOMPH-LIB_LIB_DIR) $(EXTERNAL_DIST_LIBRARIES) $(OOMPH-LIB_LIBS) \
                $(OOMPH-LIB_EXTERNAL_LIBS) $(FLIBS)
```

The version that is generated during the build process on your machine provides template for your own customised Makefiles. When modifying the sample to different driver codes, you should not (have to) edit any of the "installation specific" variables. Simply specify the `oomph-lib` (sub-)libraries that you wish to link against in the `OOMPH_↩ LIB-LIBS` variable (in the example shown above, we are linking against the `generic` and `poisson` libraries), and specify the dependencies for your own driver code, following the usual `Makefile` syntax. The executable may then be created by the usual

```
make -f makefile.sample my_demo_code
```

Note/Disclaimer: The sample `Makefile` generated during `oomph-lib's` installation should work for most (if not all) linux machines, though it may require slight tweaks for Darwin (the BSD-derived UNIX core of Apple's

OSX operating system). Problems are most likely to arise from the SHARED_LIBRARY_FLAGS variable. As mentioned in the comment in the sample Makefile, on such machines the fragment -Wl,-rpath -Wl, should be deleted from the SHARED_LIBRARY_FLAGS variable.

1.7 PDF file

A pdf version of this document is available.