Chapter 1

Frequently Asked Questions

Here are some frequently asked questions (with assorted frequently given answers...). Please check these before contacting us with any problems.

- Compilation problems and run-time errors
  - Missing 'this->'
  - Warning about "discarded sections" during linking
  - My driver code compiles but dies with a segmentation fault
  - My driver code runs but it produces incorrect/non-sensical results
  - The Newton solver diverges
- Customisation and optimisation
  - I don't have tecplot. How do I change oomph-lib's output so it can be displayed by my own plotting package?
  - oomph-lib's implementation of the Navier-Stokes equations (say) is too general (and therefore too expensive) for my appl
- Finding your way around the distribution

  - There is so much information – how do I get started?
  - Where is this class/function/... defined?
- If all else fails...
  - How to report problems/bugs

1.1 Compilation problems and run-time errors

1.1.1 Missing 'this->'

Recent versions of the `gcc compilers` enforce the `C++ standard` much more rigorously than earlier versions. Unfortunately, the standard includes some rules that are so counter-intuitive that it is hard get into the habit of using them, especially if code is developed on a compiler that does not enforce the standard as rigorously. The most frequent problem arises in classes that are derived from a templated base class. `The C++ standard` insists that all references to member functions (or member data) that is defined in the templated base class must be preceded by "this->" when the reference is made in the derived class. Allegedly, this is necessary to avoid ambiguities, though it is not entirely clear what this ambiguity is supposed to be... Here is a driver code that illustrates the problem.

```
//LIC// ====================================================================
//LIC// This file forms part of oomph-lib, the object-oriented,
//LIC// multi-physics finite-element library, available
//LIC// at http://www.oomph-lib.org.
//LIC//
//LIC// Copyright (C) 2006-2025 Matthias Heil and Andrew Hazel
//LIC//
//LIC// This library is free software; you can redistribute it and/or
//LIC// modify it under the terms of the GNU Lesser General Public
//LIC// License as published by the Free Software Foundation; either
//LIC// version 2.1 of the License, or (at your option) any later version.
//LIC//
//LIC// This library is distributed in the hope that it will be useful,
//LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
//LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//LIC// Lesser General Public License for more details.
//LIC//
//LIC// You should have received a copy of the GNU Lesser General Public
//LIC// License along with this library; if not, write to the Free Software
//LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
//LIC// 02110-1301  USA.
//LIC//
//LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
//LIC//
//LIC//====================================================================
//Demo code to document problem with missing "this->"
#include<iostream>



//==========Templated_base_class=======================================
/// Some Non-templated base class
//=====================================================================
template<unsigned TEMPLATE_PARAMETER>
class TemplatedBaseClass
{
public:

 /// Empty constructor
 TemplatedBaseClass(){};
 /// Empty virtual constructor
 ~TemplatedBaseClass(){};

 /// Some member function
 void say_hello_world()
  {
   std::cout « "Hello world from base class " « std::endl;
  }

};



//======templated_derived_class========================================
/// Some templated derived class
//=====================================================================
template<unsigned TEMPLATE_PARAMETER>
class SomeDerivedClass : public virtual TemplatedBaseClass<TEMPLATE_PARAMETER>
{

public:

 // Empty constructor
 SomeDerivedClass(){};

 // Virtual empty constructor
 virtual ~SomeDerivedClass(){};

 /// Some member function
 void output_template_parameter()
  {
   std::cout « "My template parameter is: "
             « TEMPLATE_PARAMETER « std::endl;

   // Now call the function in the base class

#ifdef USE_BROKEN_VERSION

   // This is illegal according to the C++ standard
   say_hello_world();

#else

   // This is stupid but in line with the C++ standard
   this->say_hello_world();
#endif

  }

};


//======start_of_main==================================================
/// Driver
//=====================================================================
int main()
{

 // Build the templated object:
 SomeDerivedClass<2> object;

 // Get it to output its template parameter and say hello:
 object.output_template_parameter();

} // end of main
```

If you compile this with sufficiently recent versions of the gcc compilers, using the flag `-DUSE_BROKEN_VERSION`, the compilation will fail with the following error:

```
broken_this_demo.cc: In member function 'void
      SomeDerivedClass<TEMPLATE_PARAMETER>::output_template_parameter()':
broken_this_demo.cc:55: error: there are no arguments to 'say_hello_world' that depend on a template
      parameter, so a declaration of 'say_hello_world' must be available
broken_this_demo.cc:55: error: (if you use '-fpermissive', G++ will accept your code, but allowing the use
      of an undeclared name is deprecated)
```

You may not only stumble across this problem in one of your own codes but it is also possible that some code in the library itself still violates this rule. This is because templated classes are only built when needed and it is conceivable that `oomph-lib's` suite of self-tests do not instantiate all templated classes that exist in the library. If you encounter any such problems, check if putting a "this->" in front of the function call fixes the problem. If it does, let us know!

1.1.2 Warning about "discarded sections" during linking

When linking, some versions of the `gcc compiler` produce warnings about references to "discarded sections" being referenced. Here's an example:

```
/usr/bin/ld: `.gnu.linkonce.t._ZNK5oomph8QElementILj3ELj3EE14vertex_node_ptERKj' referenced in section
      `.rodata' of /home/mheil/version185/oomph/build/lib/libgeneric.a(Qelements.o): defined in discarded
      section `.gnu.linkonce.t._ZNK5oomph8QElementILj3ELj3EE14vertex_node_ptERKj' of
      /home/mheil/version185/oomph/build/lib/libgeneric.a(Qelements.o)
```

We admit to being slightly baffled by this. Other libraries seem to suffer from the same problem (google for `.rodata discarded`, say), but as far as we can tell no solution has ever been suggested, nor does one seem to be required. The executable works fine. Upgrade to a newer version of `gcc?`

1.1.3 My driver code compiles but dies with a segmentation fault

Suggestions:

- Run `Problem::self_test()` before solving the problem. This function performs a large number of sanity checks and reports any inconsistencies in the data structure.

- Recompile the relevant libraries with the PARANOID and RANGE_CHECKING flags set. Segmentation faults are often caused by out-of-bounds access to STL containers. Since $>$ 95% (?) of the containers used in `oomph-lib` are `Vectors` (`oomph-lib's` wrapper to the STL vector class, with optional range checking) they can easily be detected. Make sure to re-compile again with RANGE_CHECKING switched off before you start any production runs – the run-time overheads incurred by range-checking are significant!

- Recompile the relevant libraries and your driver code with the debugging flag ("-g" for the gnu compiler suite) switched on and all optimisation disabled. Re-run the code in a debugger (`gdb` or its GUI-based equivalent `ddd`) to (try to!) find out where the segmentation fault occurred. [**Careful:** If the segmentation fault is caused by a pointer problem, this naive inspection can be quite misleading – tell-tale signs are that the traceback displays a non-sensical call stack, e.g. a function being called "out of nowhere"; variables that have just been given values not existing; etc.]

1.1.4 My driver code runs but it produces incorrect/non-sensical results

Suggestions:

- Have you applied the boundary conditions correctly? Check this by looping over all nodes in your mesh and documenting the pinned-status of their nodal values, using the function `Node::is_pinned(...)`.

- Have you passed the required (function) pointers to the elements? Most element constructors assign default values for any physical parameters, e.g. the Reynolds number in the Navier-Stokes elements. Similarly, most source functions etc. default to zero. For instance `oomph-lib's` Poisson elements solve the Laplace equation unless a function pointer to the source function is specified.

- In time-dependent problems, have you assigned suitable initial conditions? Note that, if you use elements that are based on the ALE formulation (and all time-dependent elements should be!) you must initialise the history values for the nodal positions, even if the mesh is stationary! See the discussion of `oomph-lib's` timestepping procedures in the context of `the unsteady heat equation` for details.

- Have you implemented the relevant "action" functions, such as `Problem::actions_before_↩ implicit_timestep()` to update any time-dependent boundary conditions?

1.1.5 The Newton solver diverges

Suggestions:

- `oomph-lib's` default solver `Problem::newton_solve(...)` will converge quadratically, provided

  - a "good" initial guess for the solution has been assigned,

  or

  - the problem is linear.

  If the Newton solver fails to converge for a nonlinear problem, try to identify a related linear problem and use continuation to generate a sequence of good initial guesses. For instance, to solve the Navier-Stokes equations at a Reynolds number of 500, say, start by solving the problem for zero Reynolds number (in which case the problem becomes linear so that the Newton method converges in one iteration); increase the Reynolds number by 50, say, and re-solve. Repeat this procedure until the desired value of the Reynolds number is reached.

  Note: `oomph-lib` also provides automatic continuation methods, based on Keller's arclength continuation, but at the moment, no tutorials exist for these.

- If you have tried the above and the Newton method fails to converge even for a linear version of your problem, the most likely reasons are that

  1. You have developed a new element and made a mistake in the implementation of the element's Jacobian matrix. To check if this is the case, comment out the function that computes the element's Jacobian matrix, i.e. the element member functions `get_jacobian(...)` or `fill_in_↩ contribution_to_jacobian(...)`. `oomph-lib` will then use the default implementation of these functions in the `GeneralisedElement` base class to compute the Jacobian matrices by finite-differencing. The executable is likely to run more slowly since the finite-difference-based computation is unlikely to be as efficient as the customised implementation for your specific element, but if the Newton method then converges, you know where to look for your bug! You may also want to check for any un-initialised variables. They are the most likely culprits if your code behaves differently at different levels of optimisation as more aggressive optimisation may suppress any default initialisations of data – in fact, you should never rely on that anyway!

  2. Your problem contains "dependent" variables, such as the nodal positions in a free-boundary problem. If the node update in response to changes in the shape of the domain boundary is performed by an algebraic node update procedure (using `AlgebraicNodes`, `SpineNodes` or nodes whose position is updated by a `MacroElement/Domain` - based procedure), the position of the nodes in the "bulk" mesh must be updated whenever the Newton method updates the unknowns. This is most easily done by calling `Mesh::node_update()` in `Problem::actions_before_newton_↩ convergence_check()`.

1.2 Customisation and optimisation

1.2.1 I don't have tecplot. How do I change oomph-lib's output so it can be displayed by my own plotting package?

`oomph-lib's` high-level post-processing routines output the results of the computations in a form that is suitable for display with tecplot, a powerful commercial plotting package. Purists may find it odd that an open-source library should choose an output format that is customised for a commercial software package. We tend to agree... Our only excuse is that tecplot is very very good, and without it we would have found it extremely difficult to create many of the plots shown in the tutorials. [If you know of any open-source plotting package whose capabilities are comparable to those of tecplot, let us know!]
Angelo Simone has written a python script that converts `oomph-lib's` output to the vtu format that can be read by paraview, an open-source 3D plotting package. The conversion script can currently deal with output from meshes that are composed of 2D quad elements – the extension to 3D is work in progress. Use of the conversion script is documented in another tutorial.
It is possible to display `oomph-lib's` default output (in more elementary form, obviously) with gnuplot. The trick is to specify the `using` option in gnuplot's plot commands – in this mode gnuplot ignores tecplot's "ZONE" commands. For instance, trying to plot the x-y data created by the demo code for the solution of the 1D Poisson equation with

```
plot "RESLT/soln0.dat"
```

will fail because gnuplot gets confused by the `ZONE` specifications required by tecplot. However,

```
plot "RESLT/soln0.dat" using 1:2
```

works.
If the data is too complex to be displayed by gnuplot, you may wish to customise the output for your preferred plotting package. This is easily done as `oomph-lib` creates its output element-by-element. The elements' various `output(...)` functions are virtual functions that can easily be overloaded in a user-defined wrapper class.
Here is an example driver code that illustrates how to change the output from `oomph-lib's QPoisson↩ Element` family of 1D-line/2D-quad/3D-brick Poisson elements so that they output the string "Hello world".
We include `oomph-lib's generic` and `poisson` library headers:

```
//LIC// ====================================================================
//LIC// This file forms part of oomph-lib, the object-oriented,
//LIC// multi-physics finite-element library, available
//LIC// at http://www.oomph-lib.org.
//LIC//
//LIC// Copyright (C) 2006-2025 Matthias Heil and Andrew Hazel
//LIC//
//LIC// This library is free software; you can redistribute it and/or
//LIC// modify it under the terms of the GNU Lesser General Public
//LIC// License as published by the Free Software Foundation; either
//LIC// version 2.1 of the License, or (at your option) any later version.
//LIC//
//LIC// This library is distributed in the hope that it will be useful,
//LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
//LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//LIC// Lesser General Public License for more details.
//LIC//
//LIC// You should have received a copy of the GNU Lesser General Public
//LIC// License along with this library; if not, write to the Free Software
//LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
//LIC// 02110-1301  USA.
//LIC//
//LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
//LIC//
//LIC//====================================================================
//Demo code to document customisation of output

// oomph-lib includes
#include "generic.h"
#include "poisson.h"
```

and then create a customised version of the Poisson elements in which we overload the tecplot-based `QPoisson↩ Element<DIM,NNODE_1D>::output(...)` function, defined in the `poisson` library:

```
// The wrapper class for the element has to be included into
// the oomph-lib namespace
namespace oomph
{

//======customised_poisson===========================================
/// Customised Poisson element -- simply overloads the output function.
/// All other functionality is retained.
//===================================================================
template<unsigned DIM, unsigned NNODE_1D>
class CustomisedQPoissonElement : public virtual QPoissonElement<DIM,NNODE_1D>
{

public:

 /// Empty constructor
 CustomisedQPoissonElement(){};

 /// Empty virtual constructor
 ~CustomisedQPoissonElement(){};

 /// Overload output function
 void output(std::ostream& output_file)
  {
   output_file « "Hello world" « std::endl;
  }

};

} //end extension of oomph-lib namespace
```

If we now call the output function, the version defined in the customised element is used. The remaining implementation of the Poisson element remains unchanged.

```
//======start_of_main==================================================
/// Driver
//=====================================================================
int main()
{

 using namespace oomph;

 // Build the templated object:
 CustomisedQPoissonElement<2,2> element;

 // Call the element's (customised) output function and dump to screen
 element.output(std::cout);

} // end of main
```

1.2.2 oomph-lib's implementation of the Navier-Stokes equations (say) is too general (and therefore too expensive) for my application. How do I change this?

Many of `oomph-lib's` equations classes (or elements) are implemented in great generality. For instance, our discretisation of the Navier-Stokes equations includes a source term in the continuity equation, and body force terms in the momentum equations; it allows switching between the stress-divergence and simplified forms of the viscous terms; it includes the mesh velocity into the ALE formulation of the time-derivatives; etc. This makes the elements very versatile and robust. However, the generality/robustness comes at a price: Even though we provide default values for most functions (e.g. the body force terms default to zero), their evaluation requires a finite amount of CPU time. If you wish to use the elements in a simple application in which the Navier-Stokes equations are solved in a fixed domain, without any body forces or other source terms, say, you may wish to disable the additional functionality.

This is easily done: After all, `oomph-lib` is open-source software and you can therefore change anything you want! In principle, you could edit the source code in the `src/navier_stokes` directory and delete (or at least comment out) all the functionality that you do not require. However, this is probably a risky step as it will break all demo codes (used during `oomph-lib's` self-test procedure) that use some of the features that you are not interested in. We therefore recommend copying the content of the directory `src/navier_stokes` into a new directory, e.g. `src/my_navier_stokes` and to edit the copied sources. Follow the instructions in the `oomph-lib` `GitHub repository` to turn these sources into a separate library against which you can link.

1.3 Finding your way around the distribution

1.3.1 There is so much information – how do I get started?

Yes, `oomph-lib` does contain a lot of code and a lot of documentation. How to get started obviously depends on your background: Are you familiar with the finite element method? How good is your knowledge of C++? Etc.

Here are some possible "routemaps" around the library:

- **You are familiar with the finite element method and have a fairly good knowledge of C++**
  - Have a look through the `list of example codes` to get a feeling for `oomph-lib's` capabilities. Pick a problem that interests you and study the associated tutorial. Copy the driver code into your own directory and play with it.
  - Once you have played with a few example codes, you may wish to to learn more about `oomph-lib's` `overall data structure`, or find out `how to optimise the library` for your particular application.
- **You have never used finite element methods but have a fairly good knowledge of C++**
  - Study the `"Top Down" introduction`. This document includes a "low tech" overview of the mathematical/theoretical background and contrasts procedural implementations of the finite element method with the object-oriented approach adopted in `oomph-lib`.
  - Consult the `(Not-So-)Quick-Guide` to learn how to construct basic `oomph-lib` objects for your problem: `Problems`, `Meshes`, `FiniteElements`, etc.
  - Continue with the steps suggested above.
- **You have never used finite element methods and are a newcomer to C++**
  - Buy `Daoqi Yang` brilliant book `C++ and Object-Oriented Numeric Computing for Scientists and Engineers`. Read it! Pretty much everything in this book is relevant for some parts of `oomph-lib`. You should at least understand:
    - The procedural aspects of C++ (basic types, functions and control structures).
    - Namespaces.
    - Classes (private, protected and public members; inheritance and multiple inheritance; virtual and pure virtual functions; base classes and derived classes; static and dynamic casts).
    - Templates and template instantiations.
    - The standard template library (STL).

  - Continue with the steps suggested above.

1.3.2 Where is this class/function/... defined?

Assume you have studied one of the example codes and wish to find out more about the implementation of a particular class or function that is used there. How do you find its source code and/or its full documentation?
Generally, a class/function that is used in a demo code can only be defined in one of two places:

1. In the demo driver code itself.
2. In an included file and/or an associated library.

oomph-lib's tutorials tend to provide a fairly complete annotated listing of the relevant driver codes; if the function you are interested in is not mentioned explicitly in the tutorial, it is most likely to be defined in an include file. You can inspect the driver code in its entirety by following the link at the end of the tutorials. If you cannot find the class/function there, it must be defined in one of the include files listed at the beginning of the source code.
The included files themselves can either be located in the same directory as the demo driver (the directory also tends to be mentioned at the end of the tutorial) or in one of oomph-lib's sub-libraries. The source code for these is located in the sub-directories of the src directory. Often the class/function is defined in a source file with an "obvious" name; if not, use grep to find it. This can, of course, be done recursively. For instance, the command

```
find . \( -name '*.h' -o -name '*.cc' \) -exec grep -H FiniteElement {} \;
```

issued in oomph-lib's top-level directory will search through the entire distribution to locate files that contain the string "FiniteElement".
You can also use the html-based representation of oomph-lib's data structure, created by doxygen, in the "bottom-up" discussion of the data structure. (Note that the search menu may not work on your browser.)

1.4 If all else fails...

1.4.1 How to report problems/bugs

If all else fails and you think you have found a bug in the library, make sure you follow these steps:

1. Isolate the problem: Try to identify the shortest driver code that still produces the problem.

2. Double-check the relevant documentation, the installation instructions on oomph-lib's Git↩ Hub repository and the other FAQs listed here.

3. Does the problem persist when you compile the library and your test code without optimisation, and when the RANGE_CHECKING and PARANOID flags are set?

4. Does the problem occur with a sufficiently recent version of the gcc compiler suite?

5. If the above steps identify the problem, `let us know`, ideally with a bug fix!

6. If you can't fix the problem yourself, get in touch `either directly`, or via our GitHub-based bug tracking system, accessible online at

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

   and provide as much information as possible (clear description of the problem; the source code; the Makefile; details of the compiler and compilation flags used; any warning/error messages that are displayed during the compilation of the library or the driver code itself; etc.)

1.5 PDF file

A `pdf version` of this document is available. \