OOMMF
Programming Manual

September 27, 2023

This manual documents release 2.1a0.

WARNING: This document is under construction.

This manual provides source code level information on OOMMF (Object Oriented Micromagnetic Framework), a public domain micromagnetics program developed at the National Institute of Standards and Technology. Refer to the OOMMF User's Guide for an overview of the project and end-user details.

Table of Contents

Disclaimer

Chapter 1

Programming Overview of OOMMF

The OOMMF[1] (Object Oriented Micromagnetic Framework) project in the Information Technology Laboratory[2] (ITL) at the National Institute of Standards and Technology[3] (NIST) is intended to develop a portable, extensible public domain micromagnetic program and associated tools. This manual aims to document the programming interfaces to OOMMF at the source code level. The main developers of this code are Mike Donahue and Don Porter.

The underlying numerical engine for OOMMF is written in C++, which provides a reasonable compromise with respect to efficiency, functionality, availability and portability. The interface and glue code is written primarily in Tcl/Tk, which hides most platform specific issues. Tcl and Tk are available for free download[4] from the Tcl Developer Xchange[5].

The code may actually be modified at 3 distinct levels. At the top level, individual programs interact via well-defined protocols across network sockets. One may connect these modules together in various ways from the user interface, and new modules speaking the same protocol can be transparently added. The second level of modification is at the Tcl/Tk script level. Some modules allow Tcl/Tk scripts to be imported and executed at run time, and the top level scripts are relatively easy to modify or replace. The lowest level is the C++ source code. The OOMMF extensible solver, OXS, is designed with modification at this level in mind.

If you want to receive e-mail notification of updates to this project, register your e-mail address with the "μMAG Announcement" mailing list:

`https://www.ctcms.nist.gov/~rdm/email-list.html`.

The OOMMF developers are always interested in your comments about OOMMF. See the Credits (Ch. 5.6.3) for instructions on how to contact them.

[1] https://math.nist.gov/oommf/

[2] https://www.nist.gov/itl/

[3] https://www.nist.gov/

[4] http://purl.org/tcl/home/software/tcltk/choose.html

[5] http://purl.org/tcl/home/

Chapter 2

Platform-Independent Make Operational Details

The OOMMF **pimake** application compares file timestamps to determine which libraries and executables are out-of-date with respect to their source code, and then compiles and links those files as necessary to make everything up to date. The design and behavior of **pimake** is based on the Unix **make** program, but **pimake** is written in Tcl and so can run on any platform where Tcl is installed. Analogous to the `Makefile` or `makefile` of **make**, **pimake** uses `makerules.tcl` files that specify *rules* (actions) for creating or updating *targets* when the targets are older than their corresponding *dependencies*. The `makerules.tcl` files are Tcl scripts augmented by a handful of commands introduced by the **pimake** application.

The `makerules.tcl` files in the **Oxs** application area include rules to automatically compile and link all C++ code found under the `oommf/app/oxs/local/` directory, so programmers who are developing `Oxs_Ext` extension modules generally do not need to be concerned with the intricacies of **pimake** beyond the instructions on running **pimake** presented in the *OOMMF User's Guide*[1].

This chapter is intended instead for programmers who are debugging, extending, or creating new OOMMF modules outside of `oommf/app/oxs/local/`. The following sections provide an overview of the structure of `makerules.tcl` files and how they control the behavior of **pimake**. Further details may be gleaned from the **pimake** sources in `oommf/app/pimake/`.

2.1 Anatomy of `makerules.tcl` files

As may be deduced from the file extension, `makerules.tcl` files are Tcl scripts and so can make use of the usual Tcl commands. However, `makerules.tcl` files are run inside a Tcl interpreter that has been augmented by **pimake** with a number of additional commands. We discuss both types of commands here, beginning with some of the standard Tcl commands commonly found in `makerules.tcl` files:

[1] https://math.nist.gov/oommf/doc/

list, llength, lappend, lsort, lindex, lsearch, concat Tcl list formation and access commands.

file Provides platform independent access to the file system, including subcommands to split and join file names by path component.

glob Returns a list of filenames matching a wildcard pattern.

format, subst Construct strings with variable substitutions.

Refer to the Tcl documentation[2] for full details.

Notice that all the Tcl command names are lowercase. In contrast, commands added by **pimake** have mixed-case names. The most common OOMMF commands you'll find in `makerules.tcl` files are

MakeRule Defines dependency rules, which is the principle goal of `makerules.tcl` files. This command is documented in detail below (Sec. 2.2).

Platform Platform independent methods for common operations, with these subcommands:

- **Name** Identifier for current platform, e.g., `windows-x86_64`, `linux-x86_64`, `darwin`.
- **Executables** Given a file stem returns the name for the corresponding executable on the current platform by prepending the platform directory and appending an execution suffix, if any. For example, `Platform Executables varinfo` would return `windows-x86_64/varinfo.exe` on Windows, and `linux-x86_64/varinfo` on Linux.
- **Objects** Similar to Platform Executables, but returns object file names; the object file suffix is `.obj` on Windows and `.o` on Linux and macOS.
- **Compile** Uses the compiler specified in the `config/platform/<platform>.tcl` to compile the specified source code file (`-src` option) into the named object file (`-out` option).
- **Link** Uses the linker specified in `config/platform/<platform>.tcl` to link together the specified object files (`-obj` option) into the named executable (`-out` option).

CSourceFile New Creates an instance of the `CSourceFile` class. The `-inc` option to `New` specifies directories to add to the search path for header files. `CSourceFile` instances support these subcommands:

- **Dependencies** Dependency list for specified C++ source file consisting of the source file itself, header files included by `#include` statements in the source code files, and also any header files found by a recursive tracing of `#include` statements. The header file search excludes system header files requested using angle-brackets,

[2] https://www.tcl-lang.org/man

e.g., #include <stdio.h>. A source code file can speed the tracing process by placing a /* End includes */ comment following the last #include statement, as in this example from oommf/app/mmdisp/mmdispsh.cc:

```
/* FILE: mmdispsh.cc                 -*-Mode: c++-*-
 *
 * A shell program which includes Tcl commands needed to support a
 * vector display application.
 *
 */

#include "oc.h"
#include "vf.h"
#include "mmdispcmds.h"

/* End includes */
...
```

The /* End includes */ statement terminates the search for further #include statements in that file.

DepPath List of directories containing files on which the specified C++ source file depends.

Recursive Given a target, loads the makerules.tcl file in each child directory of the current directory and executes the rule found there for the target. Primarily used with the default targets all, configure, clean, mostlyclean, objclean, maintainer-clean, distclean, and upgrade. The default targets have an implicit rule to do nothing except recurse the action into the new child directories. If a makerules.tcl file found in this manner has an explicit rule defined for the given target, then that rule is invoked instead of the implicit rule, and, unless the explicit rule makes a Recursive call itself, the recursion on that directory branch will stop. As an example, the makerules.tcl file in the OOMMF root directory has the rule

```
MakeRule Define {
  -targets   all
  -script    {Recursive all}
}
```

All of makerules.tcl files one level below oommf/pkg and oommf/app have "all" targets that compile and link their corresponding libraries or executables. So

```
tclsh oommf.tcl pimake all
```

run in the root OOMMF directory will build all of those libraries and applications. In contrast, `makerules.tcl` files under `oommf/doc` do **not** have explicit `all` targets, so the `tclsh oommf.tcl pimake all` call has no effect in the `oommf/doc/` subtree.

On the other hand, the `makerules.tcl` in directories under `oommf/pkg/`, `oommf/app/`, and `oommf/doc/` **do** have explicit rules for the various `clean` targets, so

```
tclsh oommf.tcl pimake maintainer-clean
```

run from the OOMMF root directory will be active throughout all three subtrees. The `maintainer-clean` rules delete all files that can be regenerated from source, meaning object files, libraries, executables, and notably all the documentation files under `oommf/doc/`. Building the OOMMF documentation requires a working installation of LaTeX[3] and either LaTeX2HTML[4] or LaTeXML[5], so don't run the `maintainer-clean` target unless you are prepared to rebuild the OOMMF documentation!

The Tcl source defining the `MakeRule`, `Platform`, `CSourceFile`, and `Recursive` commands can be found in the `oommf/app/pimake/` directory. Example use of these commands can be found in the following section.

2.2 The MakeRule command

The `makerules.tcl` files consist primarily of a collection of `MakeRule` commands surrounded by a sprinkling of Tcl support code. The order of the `MakeRule` commands doesn't matter, except that the first target in the file, usually `all`, becomes the default target. (The "default" target is the effective target if **pimake** is run without specifying a target.)

The `MakeRule` command supports a number of subcommands, but the principle subcommand appearing in `makerules.tcl` files is `Define`. This takes a single argument, which is a list of option+value pairs, with valid options being `-targets`, `-dependencies`, and `-script`. The value string for the `-targets` option is a list of one or more build targets. The targets are usually files, in which case they must lie in the same directory or a directory below the `makerules.tcl` file. The `-dependencies` option is a list of one or more files or targets that the target depends upon. The value to the `-script` option is a Tcl script that is run if a target does not exist or if any of the file dependencies have a newer timestamp than any of the targets. The dependency checking is done recursively, that is, each dependency is checked to see if it up to date with its own dependencies, and so on. A target is out of date if it is older than any of its dependencies, or the dependencies of the dependencies, etc. If any of the dependencies is out of date with respect to its own dependencies, then its script will be run during the dependency resolution. The script associated with the original target is only run after its dependency resolution is fully completed.

[3]https://www.latex-project.org
[4]https://www.latex2html.org
[5]http://dlmf.nist.gov/LaTeXML/

The following examples from `oommf/app/omfsh/makerules.tcl` should help flesh out the above description:

```
MakeRule Define {
  -targets        [Platform Name]
  -dependencies   {}
  -script         {MakeDirectory [Platform Name]}
}
```

Here the target is the platform name, e.g., `windows-x86_64`, which is a directory under the current working directory `oommf/app/omfsh/`. There are no dependencies to check, so the rule script is run if and only if the directory `windows-x86_64` does not exist. In that case the OOMMF `MakeDirectory` routine is called to create it. This is an important rule because the compilation and linking commands place their output into this directory, so it must exist before those commands are run.

Next we look at a more complex rule that is really the bread and butter of `makerules.tcl`, a rule for compiling a C++ file:

```
MakeRule Define {
  -targets        [Platform Objects omfsh]
  -dependencies   [concat [list [Platform Name]] \
                          [[CSourceFile New _ omfsh.cc] Dependencies]]
  -script         {Platform Compile C++ -opt 1 \
                           -inc [[CSourceFile New _ omfsh.cc] DepPath] \
                           -out omfsh -src omfsh.cc
                  }
}
```

In this example the target is the object file associated with the stem `omfsh`. On Windows this would be `windows-x86_64/omfsh.obj`. The dependencies are the platform directory (e.g., `windows-x86_64/`), the file `omfsh.cc`, and any (non-system) files included by `omfsh.cc`. Directory timestamps do not affect the out-of-date computation, but directories will be constructed by their `MakeRule` if they don't exist.

Note that part of the `-dependencies` list is

```
[CSourceFile New _ omfsh.cc] Dependencies]
```

As discussed in Sec. 2.1, this command resolves to a list of all non-system `#include` header files from `omfsh.cc`, or header files found recursively from those header files. The first part of `omfsh.cc` is

```
/* FILE: omfsh.cc                 -*-Mode: c++-*-
 *
 *      A Tcl shell extended by the OOMMF core (Oc) extension
 ...
 */

/* Header files for system libraries */
#include <cstring>

/* Header files for the OOMMF extensions */
#include "oc.h"
#include "nb.h"
#include "if.h"

/* End includes */
...
```

The header file `cstring` is ignored by the dependency search because it is specified inside angle brackets rather than double quotes. But the `oc.h`, `nb.h`, and `if.h` files are all considered. These files are part of the `Oc`, `Nb`, and `If` package libraries, respectively, living in subdirectories under `oommf/pkg/`. The file `oommf/pkg/oc/oc.h`, for example, will be checked for included files in the same way, and so on. The full dependency tree can be quite extensive. The **pimake** application supports a `-d` option to print out the dependency tree, e.g.,

```
tclsh oommf.tcl pimake -cwd app/omfsh -d windows-x86_64/omfsh.obj
```

This output can be helpful is diagnosing dependency issues.

The `/* End includes */` line terminates the `#include` file search inside this file. It is optional but recommended as it will speed-up dependency resolution.

If `omfsh.obj` is older than any of its dependent files, then the Tcl script specified by the `-script` option will be triggered. In this case the script runs `Platform Compile C++`, which is the C++ compiler as specified by the `oommf/config/platforms/<platform>.tcl` file. In this command `-opt` enables compiler optimizations, `-inc` supplements the include search path for the compiler, `-out omfsh` is the output object file with name adjusted appropriately for the platform, and `-src omfsh.cc` specifies the C++ file to be compiled.

The rules for building executables and libraries from collections of object modules are of a similar nature. See the various `makerules.tcl` files across the OOMMF directory tree for examples.

In a normal rule, the target is a file and if the script is run it will create or update the file. Thus, if **pimake** is run twice in succession on the same target, the second run will not trigger the script because the target will be up to date. In contrast, a pseudo-target does not exist as a file on the file system, and the associated script does not create the pseudo-target. Since the pseudo-target never exists as a file, repeated runs of **pimake** on the target will result in repeated runs of the pseudo-target script.

Common pseudo-targets include `all`, `configure`, `help`, and several `clean` variants. This last example illustrates the chaining of `clean` pseudo-targets to remove constructed files.

```
MakeRule Define {
    -targets         clean
    -dependencies    mostlyclean
  }

  MakeRule Define {
    -targets         mostlyclean
    -dependencies    objclean
    -script          {eval DeleteFiles [Platform Executables omfsh] \
                         [Platform Executables filtersh] \
                         [Platform Specific appindex.tcl]}
  }

  MakeRule Define {
    -targets         objclean
    -dependencies    {}
    -script          {
                      DeleteFiles [Platform Objects omfsh]
                      eval DeleteFiles \
                             [Platform Intermediate {omfsh filtersh}]
                     }
  }
```

All three of these rules have targets that are non-existent files, so all three are pseudo-targets. The first rule, for target `clean`, has no script so the script execution is a no-op. However, the dependencies are still evaluated, which in this case means the rule for the target `mostlyclean` is checked. This rule has both a dependency and a script. The dependencies are evaluated first, so the `objclean` script is called to delete the `omfsh` object file and also any intermediate files created as side effects of building the **omfsh** and **filtersh** executables. Next the `mostlyclean` script is run, which deletes the **omfsh** and **filtersh** executables and also the platform-specific `appindex.tcl` file. Note that none of the scripts create their target, so the targets will all remain pseudo-targets.

Chapter 3

OOMMF Variable Types and Macros

The following typedefs are defined in the `oommf/pkg/oc/`*platform*`/ocport.h` header file; this file is created by the **pimake** build process (see `oommf/pkg/oc/procs.tcl`), and contains platform and machine specific information.

- `OC_BOOL` Boolean type, unspecified width.
- `OC_BYTE` Unsigned integer type exactly one byte wide.
- `OC_CHAR` Character type, may be signed or unsigned.
- `OC_UCHAR` Unsigned character type.
- `OC_SCHAR` Signed character type. If `signed char` is not supported by a given compiler, then this falls back to a plain `char`, so use with caution.
- `OC_INT2, OC_INT4` Signed integer with width of exactly 2, respectively 4, bytes.
- `OC_INT2m, OC_INT4m` Signed integer with width of at least 2, respectively 4, bytes. A type wider than the minimum may be specified if the wider type is handled faster by the particular machine.
- `OC_UINT2, OC_UINT4, OC_UINT2m, OC_UINT4m` Unsigned integer versions of the preceding.
- `OC_REAL4, OC_REAL8` Four byte, respectively eight byte, floating point variable. Typically corresponds to C++ "float" and "double" types.
- `OC_REAL4m, OC_REAL8m` Floating point variable with width of at least 4, respectively 8, bytes. A type wider than the minimum may be specified if the wider type is handled faster by the particular machine.
- `OC_REALWIDE` Widest type natively supported by the underlying hardware. This is usually the C++ "long double" type, but may be overridden by the

```
program_compiler_c++_typedef_realwide
```

option in the `oommf/config/platform/`*platform*`.tcl` file.

The `oommf/pkg/oc/`*platform*`/ocport.h` header file also defines the following macros for use with the floating point variable types:

- `OC_REAL8m_IS_DOUBLE` True if `OC_REAL8m` type corresponds to the C++ "double" type.
- `OC_REAL8m_IS_REAL8` True if `OC_REAL8m` and `OC_REAL8` refer to the same type.
- `OC_REAL4_EPSILON` The smallest value that can be added to a `OC_REAL4` value of "1.0" and yield a value different from "1.0". For IEEE 754 compatible floating point, this should be `1.1920929e-007`.
- `OC_SQRT_REAL4_EPSILON` Square root of the preceding.
- `OC_REAL8_EPSILON` The smallest value that can be added to a `OC_REAL8` value of "1.0" and yield a value different from "1.0". For IEEE 754 compatible floating point, this should be `2.2204460492503131e-016`.
- `OC_SQRT_REAL8_EPSILON, OC_CUBE_ROOT_REAL8_EPSILON` Square and cube roots of the preceding.
- `OC_FP_REGISTER_EXTRA_PRECISION` True if intermediate floating point operations use a wider precision than the floating point variable type; notably, this occurs with some compilers on x86 hardware.

Note that all of the above macros have a leading "`OC_`" prefix. The prefix is intended to protect against possible name collisions with system header files. Versions of some of these macros are also defined without the prefix; these definitions represent backward support for existing OOMMF extensions. All new code should use the versions with the "`OC_`" prefix, and old code should be updated where possible. The complete list of deprecated macros is:

```
BOOL, UINT2m, INT4m, UINT4m, REAL4, REAL4m, REAL8, REAL8m,
REALWIDE, REAL4_EPSILON, REAL8_EPSILON, SQRT_REAL8_EPSILON,
CUBE_ROOT_REAL8_EPSILON, FP_REGISTER_EXTRA_PRECISION
```

Macros for system identification:

- `OC_SYSTEM_TYPE` One of `OC_UNIX` or `OC_WINDOWS`.
- `OC_SYSTEM_SUBTYPE` For unix systems, this is either `OC_VANILLA` (general unix) or `OC_DARWIN` (Mac OS X). For Windows systems, this is generally `OC_WINNT`, unless one is running out of a Cygwin shell, in which case the value is `OC_CYGWIN`.

Additional macros and typedefs:

- `OC_POINTERWIDTH` Width of pointer type, in bytes.
- `OC_INDEX` Typedef for signed array index type; typically the width of this (integer) type matches the width of the pointer type, but is in any event at least four bytes wide and not narrower than the pointer type.
- `OC_UINDEX` Typedef for unsigned version of OC_INDEX. It is intended for special-purpose use only. In general, use OC_INDEX where possible.
- `OC_INDEX_WIDTH` Width of `OC_INDEX` type.
- `OC_BYTEORDER` Either “4321” for little endian machines, or “1234” for big endian.
- `OC_THROW(x)` Throws a C++ exception with value “x”.
- `OOMMF_THREADS` True for threaded (multi-processing) builds.
- `OC_USE_NUMA` If true, then NUMA (non-uniform memory access) libraries are available.

Chapter 4

OOMMF eXtensible Solver

The OOMMF eXtensible Solver (OXS) top level architecture is shown in Fig. 4.1. The "Tcl Control Script" block represents the user interface and associated control code, which is written in Tcl. The micromagnetic problem input file is the content of the "Problem Specification" block. The input file should be a valid MIF 2.0 file (see the OOMMF User's Guide for details on the MIF file formats), which also happens to be a valid Tcl script. The rest of the architecture diagram represents C++ classes.

All interactions between the Tcl script level and the core solver are routed through the Director object. Aside from the Director, all other classes in this diagram are examples of `Oxs_Ext` objects—technically, C++ child classes of the abstract `Oxs_Ext` class. OXS is designed to be extended primarily by the addition of new `Oxs_Ext` child classes.

The general steps involved in adding an `Oxs_Ext` child class to OXS are:

1. Create a subdirectory under `oommf/app/oxs/local`, and add source code files with class definitions into this subdirectory. The C++ non-header source code file(s) must be given the `.cc` or `.cpp` extension. (Header files are typically denoted with the `.h` extension, but this is not mandatory.)

2. Run **pimake** to compile your new code and link it in to the OXS executable.

3. Add the appropriate `Specify` blocks to your input MIF 2.0 files.

The source code can usually be modeled after an existing `Oxs_Ext` object. Refer to the Oxsii section of the OOMMF User's Guide for a description of the standard `Oxs_Ext` classes, or Sec. 4.1 for an annotated example of an `Oxs_Energy` class. Base details on adding a new energy term are presented in Sec. 4.2.

The **pimake** application automatically detects all files in the `oommf/app/oxs/local` directory with the `.cc` or `.cpp` extensions, and searches them for `#include` requests to construct a build dependency tree. Then **pimake** compiles and links them together with the rest of the OXS files into the **oxs** executable. Because of the automatic file detection, no modifications are required to any files of the standard OOMMF distribution in order to add local extensions.

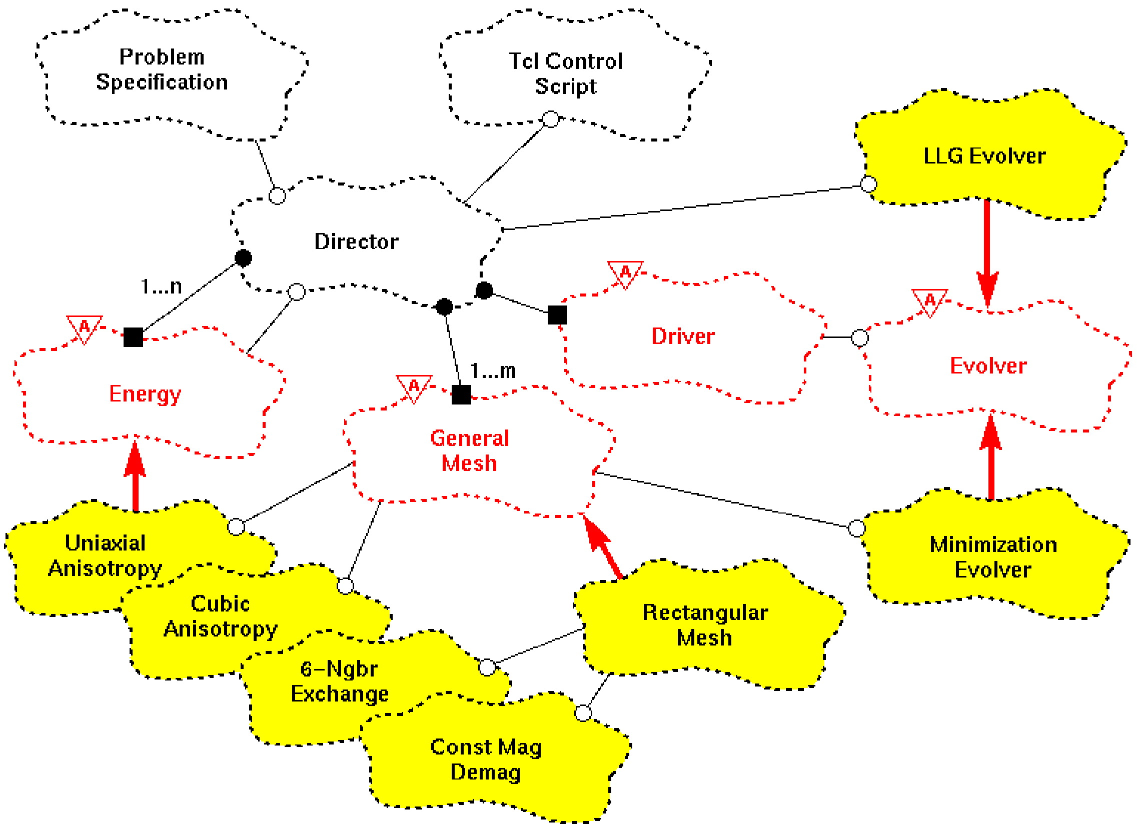


Figure 4.1: OXS top-level class diagram.

Local extensions are then activated by `Specify` requests in the input MIF 2.0 files. The object name prefix in the `Specify` block is the same as the C++ class name. All `Oxs_Ext` classes in the standard distribution are distinguished by an `Oxs_` prefix. It is recommended that local extensions use a local prefix to avoid name collisions with standard OXS objects. (C++ namespaces are not currently used in OOMMF for compatibility with some older C++ compilers.) The `Specify` block initialization string format is defined by the `Oxs_Ext` child class itself; therefore, as the extension writer, you may choose any format that is convenient. However, it is recommended that you follow the conventions laid out in the MIF 2.0 file format section of the OOMMF User's Guide.

4.1 Sample `Oxs_Energy` Class

This sections provides an extended dissection of a simple `Oxs_Energy` child class. The computational details are kept as simple as possible, so the discussion can focus on the C++ class structural details. Although the calculation details will vary between energy terms, the class structure issues discussed here apply across the board to all energy terms.

The particular example presented here is for simulating uniaxial magneto-crystalline energy, with a single anisotropy constant, `K1`, and a single axis, `axis`, which are uniform across the sample. The class definition (.h) and code (.cc) are displayed in Fig. 4.2 and 4.3, respectively.

```
/* FILE: exampleanisotropy.h
 *
 * Example anisotropy class definition.
 * This class is derived from the Oxs_Energy class.
 *
 */

#ifndef _OXS_EXAMPLEANISOTROPY
#define _OXS_EXAMPLEANISOTROPY

#include "energy.h"
#include "threevector.h"
#include "meshvalue.h"

/* End includes */

class Oxs_ExampleAnisotropy:public Oxs_Energy {
private:
  double K1;        // Primary anisotropy coeficient
  ThreeVector axis; // Anisotropy direction
public:
  virtual const char* ClassName() const; // ClassName() is
  /// automatically generated by the OXS_EXT_REGISTER macro.
  virtual BOOL Init();
  Oxs_ExampleAnisotropy(const char* name,  // Child instance id
                        Oxs_Director* newdtr, // App director
                        Tcl_Interp* safe_interp, // Safe interpreter
                        const char* argstr);  // MIF input block parameters

  virtual ~Oxs_ExampleAnisotropy() {}

  virtual void GetEnergyAndField(const Oxs_SimState& state,
                                 Oxs_MeshValue<REAL8m>& energy,
                                 Oxs_MeshValue<ThreeVector>& field
                                 ) const;
};


#endif // _OXS_EXAMPLEANISOTROPY
```

Figure 4.2: Example energy class definition. (description)

```
/* FILE: exampleanisotropy.cc            -*-Mode: c++-*-
 *
 * Example anisotropy class implementation.
 * This class is derived from the Oxs_Energy class.
 *
 */

#include "exampleanisotropy.h"

// Oxs_Ext registration support
OXS_EXT_REGISTER(Oxs_ExampleAnisotropy);

/* End includes */

#define MU0           12.56637061435917295385e-7   /* 4 PI 10^7 */

// Constructor
Oxs_ExampleAnisotropy::Oxs_ExampleAnisotropy(
  const char* name,     // Child instance id
  Oxs_Director* newdtr, // App director
  Tcl_Interp* safe_interp, // Safe interpreter
  const char* argstr)   // MIF input block parameters
  : Oxs_Energy(name,newdtr,safe_interp,argstr)
{
  // Process arguments
  K1=GetRealInitValue("K1");
  axis=GetThreeVectorInitValue("axis");
  VerifyAllInitArgsUsed();
}

BOOL Oxs_ExampleAnisotropy::Init()
{ return 1; }

void Oxs_ExampleAnisotropy::GetEnergyAndField
(const Oxs_SimState& state,
 Oxs_MeshValue<REAL8m>& energy,
 Oxs_MeshValue<ThreeVector>& field
 ) const
{
  const Oxs_MeshValue<REAL8m>& Ms_inverse = *(state.Ms_inverse);
  const Oxs_MeshValue<ThreeVector>& spin = state.spin;
  UINT4m size = state.mesh->Size();

  for(UINT4m i=0;i<size;++i) {
    REAL8m field_mult = (2.0/MU0)*K1*Ms_inverse[i];
    if(field_mult==0.0) {
      energy[i]=0.0;
      field[i].Set(0.,0.,0.);
      continue;
    }
    REAL8m dot = axis*spin[i];
    field[i] = (field_mult*dot) * axis;
    if(K1>0) {
      energy[i] = -K1*(dot*dot-1.0); // Make easy axis zero energy
    } else {
      energy[i] = -K1*dot*dot; // Easy plane is zero energy
    }
  }
}
```

Figure 4.3: Example energy class code. (description)

4.2 Writing a New `Oxs_Energy` Extension

Under construction.

4.3 Writing a New `Oxs_Evolver` Extension

Using the templated Runge-Kutta class. Under construction.

Chapter 5

Debugging OOMMF

This chapter provides an introduction to debugging OOMMF and OOMMF extension source code, providing background to the OOMMF build architecture and detailing some tools and techniques for uncovering programming errors. It begins with a look at the OOMMF **pimake** application used for compiling and linking OOMMF programs, followed by some considerations involving the **oommf.tcl** bootstrap wrapper. Then configuration files governing build and runtime behavior are detailed. After this methods for identifying and locating runtime errors are presented, including a brief introduction on using debugger applications with OOMMF. Although the primary focus of this chapter is on errors in C++ code, the interface and glue code linking the various OOMMF applications rely on Tcl script. An example of working with Tcl in OOMMF is provided in Fig. 5.1

Throughout this chapter, unless otherwise stated, commands are implicitly assumed to be run from the OOMMF root directory (i.e. the directory containing the file `oommf.tcl`), and directory paths are taken relative to this directory (e.g., `app/oxs/` refers to the directory `<oommf_root>/app/oxs/`).

In text blocks containing command statements and program output, command statements are indicated with a leading character representing the shell command prompt. On Windows this character is typically "`>`", whereas the Unix and macOS shells more commonly use "`$`" with `bash` shells or "`%`" with `zsh`. All three are used below, but "`%`" is limited to macOS specific examples to minimize confusion with the Tcl command prompt, which is also "`%`". For additional visibility shell commands are colored cyan and program commands (Tcl and debugger) are colored red. (Computer responses remain in black text.)

Some details in what follows may vary depending on the particular operating system and application version, but hopefully the differences are sufficiently small that this description remains a useful guide.

5.1 Configuration Files

There are several OOMMF configuration settings that impact debug operations. The controlling files are `config/options.tcl` and `config/platforms/<platform>.tcl`, where the

`<platform>` is `windows-x86_64`, `linux-x86_64`, or `darwin` for Windows, Linux, or macOS operating systems respectively. In practice, rather than modifying the default distribution files directly, you should place your modifications in local files `config/local/options.tcl` and `config/platforms/local/<platform>.tcl`. The `local/` directories and files are not part of the OOMMF distribution; you will need to create them manually. The files can be empty initially, and then populated as desired.

The `options.tcl` file contains platform-agnostic settings that are stored in the `Oc_Option` database. Some of these settings affect the build process, while others control post-build run-time behavior. All are set using the `Oc_Option` command, which takes name + value pairs. The `cflags` and `optlevel` settings control compiler options. The default setting for `cflags` is

```
Oc_Option Add * Platform cflags {-def NDEBUG}
```

which causes the C macro "`NDEBUG`" to be defined. If this is not set then various run-time checks such as `assert` statements and some array index checks are activated. These checks slow execution but may be helpful in diagnosing errors. Other `cflag` options include `-warn`, which enables compiler warning messages, and `-debug`, which tells the compiler to generate debugging symbols. A good `cflags` setting for debugging is

```
Oc_Option Add * Platform cflags {-warn 1 -debug 1}
```

There is also an `lflags` option, similar to `cflags`, that controls options to the linker. The default is an empty string (no options), and you generally don't need to change this.

The `optlevel` option sets the compiler optimization level, with an integer value between 0 and 3. The default value is 2, which selects for a high but reliable level of optimizations. Some optimizations may reorder and combine source code statements, making it harder to debug code, so you may want to use

```
Oc_Option Add * Platform optlevel 0
```

to disable all optimizations.

The `config/platforms/<platform>.tcl` files set default platform and compiler specific options. For example, `config/platforms/windows-x86_64.tcl` is the base platform file for 64-bit Windows. There are separate sections inside this file for the various supported compilers. You can make local changes to the default settings by creating a subdirectory of `config/platforms/` named `local/`, and creating there an initially empty file with the same name as the base platform file. Inside the base platform file is a code block labeled `LOCAL CONFIGURATION`, which lists all the available local modifications. You can copy some or all of this Tcl code block to your new `config/platforms/local/` file, and then uncomment and modify options as desired. For example, if you are using the Visual C++ compiler on Windows, you may want to include the `/RTCus` compiler flag to enable some run-time error checks. You can do that with these lines in your `local/windows-x86_64.tcl` file:

```
$config SetValue program_compiler_c++_remove_flags {/O2}
$config SetValue program_compiler_c++_remove_valuesafeflags {/O2}
$config SetValue program_compiler_c++_add_flags {/RTCus}
$config SetValue program_compiler_c++_add_valuesafeflags {/RTCus}
```

The `*_valuesafeflags` options are for code with sensitive floating-point operations that must be evaluated exactly as specified. This pertains primarily to the double-double routines in `pkg/xp/`. The `*_flags` options are for everything else. The `*_remove_*` controls remove options from the default compile command. This can be a (Tcl) list, with each element matching as a regular expression. (Refer to the Tcl documentation[1] on the `regexp` command for details.) The `*_add_*` controls append options. OOMMF sets `/O2` optimization by default, but `/O2` is incompatible with `/RTCus`, so in this example `/O2` is removed to allow `/RTCus` to be added. (Setting `optlevel 0` in the `config/local/options.tcl` file, as explained above, replaces `/O2` with `/Od`. So strictly speaking it is not necessary to remove `/O2` in that case, but it doesn't hurt either.)

You can run the command "`oommf.tcl +platform +v`" to see the effects of your current `options.tcl` and `<platform>.tcl` settings. For example,

```
$ tclsh oommf.tcl +platform +v
[...]
--- Local config options ---
[...]
   Oc_Option Add * Platform cflags -debug 1 -warn 1
   Oc_Option Add * Platform optlevel 0
[...]
--- Local platform options ---
   $config SetValue program_compiler_c++_remove_flags /O2
   $config SetValue program_compiler_c++_remove_valuesafeflags /O2
   $config SetValue program_compiler_c++_add_flags /RTCus
   $config SetValue program_compiler_c++_add_valuesafeflags /RTCus

--- Compiler options ---
     Standard options: /Od /D_CRT_SECURE_NO_DEPRECATE /RTCus
   Value-safe options: /Od /fp:precise /D_CRT_SECURE_NO_DEPRECATE /RTCus
```

To see the exact, full platform-specific compile and link commands, you can delete and rebuild individual executables in the OOMMF package. Two examples, one using the standard build options (`pkg/oc/<platform>/varinfo`) and one using the value-safe options (`pkg/xp/<platform>/build_port`) are presented below. (The response lines have been edited for clarity.)

```
% cd pkg/oc
```

[1]https://www.tcl-lang.org/man/

```
% tclsh ../../oommf.tcl pimake clean
% tclsh ../../oommf.tcl pimake darwin/varinfo
clang++ -c -DNDEBUG -m64 -std=c++11 -Ofast -o darwin/varinfo.o varinfo.cc
clang++ -m64 darwin/varinfo.o -o darwin/varinfo

% cd ../..
% cd pkg/xp
% tclsh ../../oommf.tcl pimake clean
% tclsh ../../oommf.tcl pimake darwin/build_port
clang++ -c -DNDEBUG -m64 -std=c++11 -O3 -DXP_USE_MPFR=0
   -o darwin/build_port.o build_port.cc
clang++ -m64 darwin/build_port.o -o darwin/build_port
```

The above is for macOS. Adjust the `<platform>` field as appropriate, and on Windows append `.exe` to the executable targets (`varinfo` and `build_port`).

You can also use this method to manually compile and/or link individual files: (1) Change to the relevant build directory (always one level below either `pkg` or `app`), (2) delete the file you want to rebuild from the `<platform>` directory, (3) run `pimake` as above to build the file, (4) copy and paste the compile/link command to the shell prompt, edit as desired, and rerun.

The `varinfo` and `build_port` executables are used to construct the platform-specific header files `pkg/oc/<platform>/ocport.h` and `pkg/xp/<platform>/xpport.h`. These files contain C++ macro definitions, typedefs, and function wrappers, and are an important adjunct when reading the OOMMF source code.

For in-depth investigations Tcl can be used to directly query and debug OOMMF initialization scripts. Start a Tcl shell, and from inside the shell append the OOMMF `pkg/oc` directory to the Tcl global `auto_path` variable. Next run `package require Oc` to load the Tcl-only portion of the OOMMF `Oc` library into the shell. Then you can check any and all `Oc_Option` values from `config/options.tcl`, platform configuration settings from `config/platforms/<platform>.tcl`, and perform various other types of introspection from the Tcl shell. See Fig. 5.1 for a sample session.

5.2 Understanding pimake

The OOMMF **pimake** application controls the compiling and linking of OOMMF's C++ components. Based broadly on the Unix make utility, **pimake** is a platform independent tool written in Tcl. Each of the source code directories in the OOMMF distribution tree has a `makerules.tcl` file that specifies build targets and dependencies. A dependency tree is build from this information augmented with recursive tracking of `#include` statements inside the referenced source code files. Each time **pimake** is run it compares file timestamps against the dependency tree, and compiles and links any object and executable files that are older than any of their dependencies.

```
$ pwd
/Users/barney/oommf
$ tclsh
% set env(OOMMF_BUILD_ENVIRONMENT_NEEDED) 1
% lappend auto_path [file join [pwd] pkg oc]
% package require Oc

% # Miscellaneous utilities from Oc_Main (oommf/pkg/oc/main.tcl)
% Oc_Main GetOOMMFRootDir    ;# OOMMF root directory
/Users/barney/oommf
% Oc_Main GetPid             ;# Process id
17423

% # Oc_Option database (oommf/config/options.tcl)
% # Code details in oommf/pkg/oc/option.tcl
% Oc_Option Get *            ;# Registered Option classes (glob-match)
Net_Link Oc_Url Platform Menu Nb_InputFilter Net_Server Oc_Class Color
Net_Host MIFinterp OxsLogs
% Oc_Option Get Platform *   ;# All options for class Platform (glob-match)
cflags lflags optlevel
% Oc_Option GetValue Platform cflags  ;# Platform,cflags value
-def NDEBUG

% # Configuration values (oommf/config/platforms/<platform>.tcl)
% # Code details in oommf/pkg/oc/config.tcl
% set config [Oc_Config RunPlatform]
% $config GetValue platform_name                           ;# Platform name
darwin
% $config GetValue program_compiler_c++_name               ;# C++ compiler
clang++
% $config GetValue program_compiler_c++_typedef_realwide   ;# realwide typedef
long double
% $config Features program_linker*             ;# GetValue names (glob-match)
program_linker_option_lib program_linker program_linker_rpath
program_linker_uses_-L-l program_linker_option_out program_linker_option_obj

% exit                                ;# Exit Tcl shell
```

Figure 5.1: Sample Tcl-level OOMMF introspection session. Shell commands are colored cyan (with $ prompt) and Tcl commands are colored red (with % prompt). (description)

After editing `*.h` or `*.cc` files in OOMMF, you should run **pimake** to propagate your changes to the associated OOMMF executable(s). If you run `tclsh oommf.tcl pimake` in a directory below the OOMMF root directory, then only changes at that directory and lower are affected. You can use the `-cwd` option to **pimake** to change the effective starting directory. Changes to the OOMMF configuration files (Sec. 5.1) do **not** trigger dependency updates, so if you make changes affecting the build process in these files you should manually run

```
$ tclsh oommf.tcl pimake distclean
$ tclsh oommf.tcl pimake
```

from the OOMMF root directory to delete and then rebuild the full OOMMF project.

5.3 Bypassing the `oommf.tcl` bootstrap

When an application is launched by clicking a button in **mmLaunch** or from the command shell like

```
> tclsh oommf.tcl mmdisp
```

the application (here **mmDisp**) is not executed directly but rather through the "bootstrap" program `oommf.tcl`. The bootstrap constructs a list linking application names to commands using the `appindex.tcl` files in the various application (`oommf/app/`) directories, and then runs the command associated with the given name. This is convenient for normal use, but the additional execution layer can obfuscate the debugging process. You can obtain the direct command from the bootstrap program itself with the `+command` option

```
> tclsh oommf.tcl mmdisp +command
app/mmdisp/windows-x86_64/mmdispsh.exe app/mmdisp/scripts/mmdisp.tcl &
```

The response is the command as used inside a Tcl shell to launch the application. You may need to make minor edits to run the application at your shell command prompt. For example, the trailing ampersand runs the program in the background, which is not what one usually wants when debugging, so you would omit this. On Windows you may want to change the forward slash path separators to backslashes. Another Windows-specific modification involves the first component of this command, `app/mmdisp/windows-x86_64/mmdispsh.exe`. This is an executable containing an embedded Tcl interpreter that processes the Tcl script specified as the second command component. If you examine the `app/mmdisp/windows-x86_64/` directory you'll find two executables, `mmdispsh.exe` and `condispsh.exe`. On Unix and macOS these two programs are the same, but on Windows the first is linked as a native Windows application and the second as a console application. The importance of this is that only the second provides the usual C++ standard channels `stdin`, `stdout`, and `stderr`. In case of abnormal operation programs will sometimes write error messages to `stdout` or `stderr`, which will be lost if the program is not running as a console application. The upshot is that for debugging purposes you would probably want to run **mmDisp** (for example) from a Windows command console as

```
> app\mmdisp\windows-x86_64\condispsh.exe app/mmdisp/scripts/mmdisp.tcl
```

It is worth noting that on the bootstrap command line, arguments starting with '`+`' (for example, "`+command`") are options to `oommf.tcl` itself. Run "`tclsh oommf.tcl +h`" to see the bootstrap help message. Options to the OOMMF application follow the application name and start with '`-`'. For example, to see the help message for a particular application, run "`tclsh oommf.tcl <appName> -h`".

5.4 Segfaults and other asynchronous termination

If an OOMMF application suddenly aborts without displaying an error message, the most likely culprit is a segfault caused by attempted access to memory outside the program's purview. If this occurs while running **oxsii** or **boxsi**, the first thing to check is the `oxsii.log` and `boxsi.log` log files in the OOMMF root directory. If there are no hints there, and the error is repeatable, then you can enable core dumps and re-run the program until the crash repeats. You can then obtain a stack trace from the core dump to determine the origin of the failure.

On Linux, enable core dumps with the shell command `ulimit -Sc unlimited`, and then run `ulimit -Sc` to check that the request was honored. If not, then ask your sysadmin about enabling core dumps. (Core dumps can be rather large, so after analysis is complete you should disable core dumps by running `ulimit -Sc 0` in the affected shell, or else exit that shell altogether.) Once core dumps are enabled, run the offending application from the core-dumped enabled shell prompt. When the application aborts an image of the program state at the time of termination is written to disk. The name and location of the core dump varies between Linux distributions. On older systems the core file will be written to the current working directory with a name of the form `core.<pid>`, where `<pid>` is the pid of the process. (If the process is **oxsii** or **boxsi** then the working directory will be the directory containing the `.mif` file.) Otherwise, use the command `sysctl kernel.core_pattern` to determine the pattern used to create core files. If the pattern begins with a `|` "pipe" symbol, then the core is piped through the indicated program, and you will have to check the system documentation for that program to figure out where the core went!

If the core was piped through **systemd-coredump**, then you can use the **coredumpctl** utility to gain information about the process. (More on this below.) Some Linux variants, for example Ubuntu, use **apport**, but may configure it to effectively disable core dumps for executables outside the system package management system. In this case you might want to install the `systemd-coredump` package to replace **apport**, or else use `sysctl` to change `kernel.core_pattern` to a simple file pattern (e.g., `/tmp/core-%e.%p.%h.%t`).

If you have a core dump, you can run the GNU debugger **gdb** on the executable and core dump to determine where the fault occurred:

```
$ cd app/oxs
$ gdb linux-x86_64/oxs /tmp/core.12345
Program terminated with signal 11, Segmentation fault.
#0  0x00000000005a40da in Oxs_UniaxialAnisotropy::RectIntegEnergy
  (Oxs_SimState const&, Oxs_ComputeEnergyDataThreaded&,
  Oxs_ComputeEnergyDataThreadedAux&, long, long) const ()
(gdb) bt
#0  0x00000000005a40da in Oxs_UniaxialAnisotropy::RectIntegEnergy
  (Oxs_SimState const&, Oxs_ComputeEnergyDataThreaded&,
  Oxs_ComputeEnergyDataThreadedAux&, long, long) const ()
#1  0x00000000005a6fed in Oxs_UniaxialAnisotropy::ComputeEnergyChunk
  (Oxs_SimState const&, Oxs_ComputeEnergyDataThreaded&,
  Oxs_ComputeEnergyDataThreadedAux&, long, long, int) const ()
#2  0x000000000040ce44 in Oxs_ComputeEnergiesChunkThread::Cmd(int,
   void*) ()
#3  0x00000000004697bd in _Oxs_Thread_threadmain(Oxs_Thread*) ()
#4  0x00007f90ea7fb330 in ?? () from /lib64/libstdc++.so.6
#5  0x00007f90ea019ea5 in start_thread () from /lib64/libpthread.so.0
#6  0x00007f90e9d42b0d in clone () from /lib64/libc.so.6
(gdb) quit
```

(For visibility, shell commands are colored cyan, and **gdb** commands are red. The **gdb** commands are also prefixed with the (gdb) prompt. For example, “bt” above invokes the **gdb** “backtrace” command.) We see that the segmentation fault occurred in the member routine `RectIntegEnergy` of class `Oxs_UniaxialAnisotropy`, called by `ComputeEnergyChunk`, and so on. If **oxs** had been built with debugging symbols (cf. configuration files, Sec. 5.1), then the stack trace would include the corresponding source code files and line numbers.

If the core dump was journaled by **systemd-coredump**, then the command `coredumpctl list` will list all available core dumps, including a timestamp, the pid, and the name of the executable. You can get a stack trace with `coredumpctl info <pid>`, or load the core dump directly into **gdb** with `coredumpctl gdb <pid>`. (Some versions of **coredumpctl** want “debug” in place of “gdb” in that command; check your system documentation for details.)

On macOS, crash reports are automatically generated and can be viewed from the macOS **Console** app. Select “User Reports” or “Crash Reports” from the left hand sidebar, and select the crashed process. The report provides details about the run, including a stack trace.

You can also create core files on macOS in a very similar way as on Linux. Set `ulimit -Sc unlimited` and run the application. Core files are written to the directory `/cores/`, with naming convention `core.<pid>`. If you built OOMMF with **g++**, then you can obtain a stack trace with **gdb** as above. (Note that in MacPorts the **gdb** executable is named `ggdb`.) If you built with **clang++** then you may want to use the LLVM **lldb** debugger, which should be included with the **clang++** package. Here is an example **lldb** session, for an **oxs** executable built with debugging symbols:

```
% cd app/oxs
% lldb -c /cores/core.54416 darwin/oxs
(lldb) target create "darwin/oxs" --core "/cores/core.54416"
Core file '/cores/core.54416' (x86_64) was loaded.
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
 * frame #0: 0x0000000103cfc188 oxs`Oxs_UniaxialAnisotropy::RectIntegEnergy
 (this=0x00007ff0f4801000, state=0x00007ff0f350e830, ocedt=0x00007ffeec35a9a8,
 ocedtaux=0x00007ff0f350e6a0, node_start=16384, node_stop=20000) const at
 uniaxialanisotropy.cc:246
   frame #1: 0x0000000103cfd864 oxs`Oxs_UniaxialAnisotropy::ComputeEnergyChunk
 (this=0x00007ff0f4801000, state=0x00007ff0f350e830, ocedt=0x00007ffeec35a9a8,
 ocedtaux=0x00007ff0f350e6a0, node_start=16384, node_stop=20000, (null)=0)
 const at uniaxialanisotropy.cc:454
   frame #2: 0x00000001038a1739 oxs`Oxs_ComputeEnergiesChunkThread::Cmd
 (this=0x00007ffeec35b440, threadnumber=0, (null)=0x0000000000000000) at
 chunkenergy.cc:199
   frame #3: 0x00000001039eabaf oxs`Oxs_ThreadTree::LaunchTree
 (this=0x0000000103ef3860, runobj=0x00007ffeec35b440, data=0x0000000000000000)
 at oxsthread.cc:856
[...]
(lldb) quit
```

Similar to the **gdb** example, the debugger prompt is "(lldb)", and "bt" requests a stack trace.

To create and examine core dumps on Windows, download and install **ProcDump** and either **WinDbg** or **Visual Studio** applications from Microsoft. To get symbols in the process dump file you will need to build OOMMF with symbols, i.e., include

```
Oc_Option Add * Platform cflags {-debug 1}
```

in the `config/local/options.tcl`. Also, since `-def NDEBUG` is not included on this line, the C macro `NDEBUG` will not be defined, which enables code `assert` statements and other consistency checks, including in particular array bound checks for `Oxs_MeshValue` arrays.

You can create an `oxs` process dump by

```
> cd app\oxs
> procdump -ma -t -e -x . windows-x86_64\oxs.exe boxsi.tcl foo.mif
```

On program exit (termination, `-t`) or unhandled exception (`-e`) `procdump` will write a full dump file (`-ma`) to `oxs.exe_YYMMDD_HHMMSS.dmp` in the `app/oxs` directory.

Follow this procedure to examine the dump file in **WinDbg**:

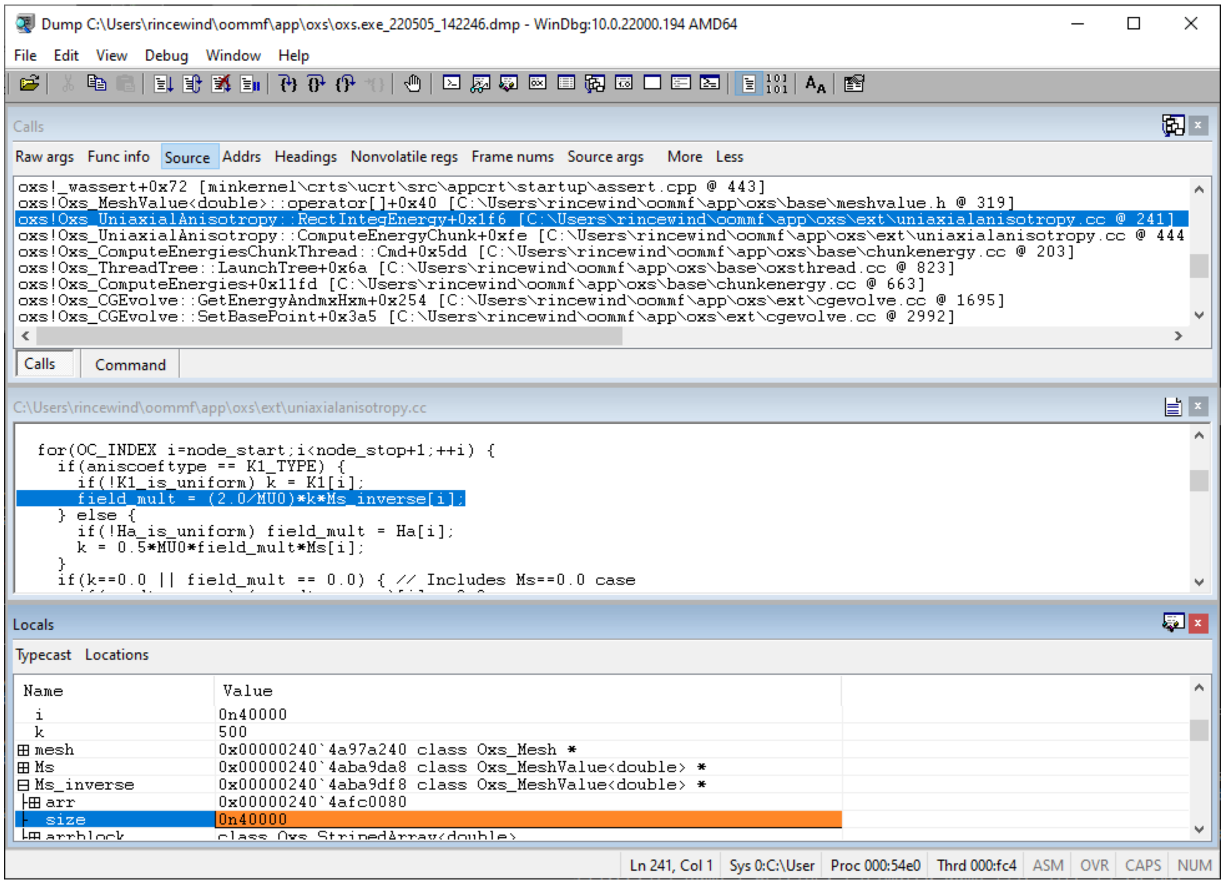
1. Launch **WinDbg**.
2. Use the menu item `File|Open Crash Dump...` to load the `.dmp` file.
3. Then `View|Call Stack` will open a call stack window.
4. Double-clicking on a call stack frame will highlight the corresponding line of code in the C++ source. By default only the upper portion of the call stack is displayed, which may be just system exit handling code. You may need to click the "More" control in the toolbar one or more times and scroll down to reach OOMMF routines. Enable the "Source" toolbar option to include filenames and line references in the stack list.
5. You can examine variable values at the time of the crash by opening the `View|Locals` window. Referring to the the source code and local variable windows in Fig. 5.2, we see that the index variable `i` has value 40000, but the size of the `Ms_inverse` array only has size 40000. Thus the access into `Ms_inverse` on line 241 (highlighted) is one element beyond the end of the array.

An alternative to **WinDbg** is to use the debugger built into Visual Microsoft's Visual Studio:

1. Launch Visual Studio.
2. Select the `Continue without code` option (below the "Get started" column).
3. Select `File|Open|File ...`, and load the `*.dmp` file.
4. Under "Actions" in the "Minidump File Summary" window, select `Debug with Native Only`.
5. If not automatically displayed, bring up `Debug|Windows|Call Stack`.
6. Double-clicking in the call stack will bring up and highlight the corresponding line of code in the C++ source.
7. Use the `Debug|Windows|Autos` and `Debug|Windows|Locals` menu items to display variable values.

5.5 Out-of-bounds memory access

One of the more common coding errors is allowing array access outside the allocated range of an array. This error can be insidious because the program may continue to run past the point of invalid access, but plant a seed that grows into a seemingly unrelated fatal error later on. There are a number of tools designed to uncover this problem, but an especially easy one to use that is common on Linux systems is the venerable Electric Fence, original written by Bruce Perens in 1987. If the `libefence.so` shared library is installed, then from the `bash` prompt in the `oommf/app/oxs` directory you can run


Dump C:\Users\rincewind\oommf\app\oxs\oxs.exe_220505_142246.dmp - WinDbg:10.0.22000.194 AMD64
File Edit View Debug Window Help
Calls
Raw args Func info Source Addrs Headings Nonvolatile regs Frame nums Source args More Less
Calls Command
C:\Users\rincewind\oommf\app\oxs\ext\uniaxialanisotropy.cc
Locals
Typecast Locations
Ln 241, Col 1 Sys 0:C:\User Proc 000:54e0 Thrd 000:fc4 ASM OVR CAPS NUM

```
oxs!_wassert+0x72 [minkernel\crts\ucrt\src\appcrt\startup\assert.cpp @ 443]
oxs!Oxs_MeshValue<double>::operator[]+0x40 [C:\Users\rincewind\oommf\app\oxs\base\meshvalue.h @ 319]
oxs!Oxs_UniaxialAnisotropy::RectIntegEnergy+0x1f6 [C:\Users\rincewind\oommf\app\oxs\ext\uniaxialanisotropy.cc @ 241]
oxs!Oxs_UniaxialAnisotropy::ComputeEnergyChunk+0xfe [C:\Users\rincewind\oommf\app\oxs\ext\uniaxialanisotropy.cc @ 444
oxs!Oxs_ComputeEnergiesChunkThread::Cmd+0x5dd [C:\Users\rincewind\oommf\app\oxs\base\chunkenergy.cc @ 203]
oxs!Oxs_ThreadTree::LaunchTree+0x6a [C:\Users\rincewind\oommf\app\oxs\base\oxsthread.cc @ 823]
oxs!Oxs_ComputeEnergies+0x11fd [C:\Users\rincewind\oommf\app\oxs\base\chunkenergy.cc @ 663]
oxs!Oxs_CGEvolve::GetEnergyAndmxHxm+0x254 [C:\Users\rincewind\oommf\app\oxs\ext\cgevolve.cc @ 1695]
oxs!Oxs_CGEvolve::SetBasePoint+0x3a5 [C:\Users\rincewind\oommf\app\oxs\ext\cgevolve.cc @ 2992]
```

```
  for(OC_INDEX i=node_start;i<node_stop+1;++i) {
    if(aniscoeftype == K1_TYPE) {
      if(!K1_is_uniform) k = K1[i];
      field_mult = (2.0/MU0)*k*Ms_inverse[i];
    } else {
      if(!Ha_is_uniform) field_mult = Ha[i];
      k = 0.5*MU0*field_mult*Ms[i];
    }
    if(k==0.0 || field_mult == 0.0) { // Includes Ms==0.0 case
```

```
Name            Value
 i              0n40000
 k              500
⊞ mesh          0x00000240`4a97a240 class Oxs_Mesh *
⊞ Ms            0x00000240`4aba9da8 class Oxs_MeshValue<double> *
⊟ Ms_inverse    0x00000240`4aba9df8 class Oxs_MeshValue<double> *
 ├⊞ arr         0x00000240`4afc0080
 ├ size         0n40000
```

Figure 5.2: **WinDbg** screenshot displaying call stack, source code, and local variables read from a crash dump generated by **procdump**.

```
$ LD_PRELOAD=libefence.so linux-x86_64/oxs boxsi.tcl foo.mif
```

(On some installations there may also be an equivalent shell wrapper `ef`.) This will abort with a segfault if an invalid memory reference (read or write) is detected. One nice feature is that you don't have to rebuild OOMMF to use this debugger—the `efence` shared library transparently replaces the standard system memory allocator with the instrumented Electric Fence version at runtime. If you enable core dumps as explained above, then on Linux systems even without debug symbols a stack trace on the core dump will provide the function call list. If you build OOMMF with debugging symbols (`Oc_Option cflags` option `-debug` in `config/local/options.tcl`), then the core stack trace will give the source file and line number where the invalid memory access occurred. Also, OOMMF runs at normal speed with Electric Fence enabled, so you can use it to check for errors in large simulations.

One caveat is that for performance reasons, OOMMF sometimes allocates larger memory blocks than needed. Electric Fence detects memory accesses outside the requested memory range, so OOMMF accesses of memory outside its proper range but inside the requested range will not be flagged. You can have OOMMF request tight blocks by putting these lines in your `local/<platform>.tcl` file:

```
$config SetValue program_compiler_c++_property_cache_linesize 1
$config SetValue program_compiler_c++_property_pagesize 1
$config SetValue sse_no_aligned_access 1
```

and rebuilding OOMMF (`pimake distclean` plus `pimake`).

Normally Electric Fence detects accesses to memory locations above the allocated range (index too high), but you can have it check instead for memory accesses preceding the allocated range (index too low) by setting the environment variable `EF_PROTECT_BELOW` to 1.

The Electric Fence documentation warns that core dumps of Electric Fence enabled runs can be significantly larger than core dumps without Electric Fence, and so recommends running Electric Fence with the selected executable (here `oxs`) from inside a debugger rather than creating a core dump. This does not appear to be a problem when used with OOMMF however, as the core dumps with Electric Fence tend to be only modestly larger than those without.

A similar tool on macOS is the gmalloc (Guard Malloc) package, which is included with Xcode. Run it from the `oommf/app/oxs` bash or zsh command line with

```
% DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib darwin/oxs boxsi.tcl foo.mif
```

See the documentation from Apple for full details.

5.6 C++ source code debuggers

If you know roughly where a bug is occurring in the code, you can often debug it by temporarily inserting `printf` or `std::cout <<` statements in the code. But for more complex problems it can be more informative and quicker in the long run to create a debugging build (i.e., one with debugging symbols and perhaps with compiler optimizations disabled) and run the program in a debugger. This section provides general information on running OOMMF in a debugger, including short examples in three common debuggers: **gdb**, **lldb**, and **Visual Studio Debugger**.

First edit the configuration files for debugging, as explained in Sec. 5.1. Then run

```
$ tclsh oommf.tcl pimake distclean
$ tclsh oommf.tcl pimake
```

to create a build of OOMMF with debugging symbols. After this you can load an OOMMF executable into a debugger, run the program, and examine its execution. (Remember to bypass the `oommf.tcl` bootstrap as explained in Sec. 5.3.) There are many debuggers available, some with multiple front-ends. But one overriding criterion in selecting a debugger is to choose one that supports the debugging symbol format output by your C++ compiler. To provide a brief taste of this subject, we will look at three debuggers: GNU's venerable **gdb** for use with `g++`, the **lldb** debugger packaged with Xcode/**clang++** on macOS, and the debugger built into Microsoft's **Visual Studio** for use with Visual C++ `cl` binaries.

5.6.1 Introduction to the GNU gdb debugger

This section provides a brief overview on using **gdb** for debugging OOMMF programs. For a more thorough background you can refer to the extensive documentation available from the GNU Project or the many online tutorials.

In the following examples, the (**bash**) shell prompt is indicated by `$`, and the **gdb** prompt with `(gdb)`. You launch **gdb** from the command line with the name of the executable file. You can provide arguments to the executable when you `run` the program inside **gdb**. For example, to debug a problem with an `Oxs` extension, we would run **Boxsi** with a sample troublesome `.mif` file, say

```
$ cd oommf/app/oxs
$ gdb linux-x86_64/oxs
(gdb) run boxsi.tcl local/foo/foo.mif -threads 1
```

Subsequent `run` commands will reuse the same arguments unless you specify new ones. In this example the `-threads 1` option to **Boxsi** is used to simplify the debugging process. If you need or want to debug with multiple threads, then read up on the "thread" command in the **gdb** documentation.

The program run will automatically terminate and return to the `(gdb)` prompt if the program exits or aborts. Alternately you can `Ctrl-C` at any time to manually halt. To exit **gdb** type `quit` at the `(gdb)` prompt.

gdb has a large collection of commands that you can use to control program flow and inspect program data. An example we saw before is `backtrace`, which can be abbreviated as `bt`. Fig. 5.3 lists a few of the more common commands, and Figs. 5.4 and 5.5 provide an example debugging session illustrating their use.

Shellcommand: `gdb linux-x86_64/oxs [corefile (opt)]`		
Command	**Abbr.**	**Description**
Process control		
run [*args*]		run executable with *args*
run		run executable with last *args*
show args		display current *args*
set env FOO bar		set envr. variable FOO to "bar"
unset env FOO		unset environment variable FOO
Ctrl-C		stop and return to (gdb) prompt
kill		terminate current run
quit		exit gdb
Introspection		
backtrace	bt	stack trace
frame 7	f 7	change to stack frame 7
list 123	l 123	list source about line 123
list foo.cc:50		list source about line 50 of foo.cc
list -	l -	list preceding ten lines
list foo::bar		list first ten lines of function foo::bar()
set listsize 20		change list output length to 20 lines
info locals	i lo	print local variables
info args		print function arguments
print foo	p foo	write info on variable foo
printf "%g", foo		print foo with format %g (note comma)
Flow control		
break bar.cc:13	b bar.cc:13	set breakpoint at line 13 of bar.cc
break foo::bar	b foo::bar	break on entry to C++ routine foo::bar()
info breakpoints	i b	list breakpoints
delete 4	d 4	delete breakpoint 4
delete	d	delete all breakpoints
ignore 3 100		skip breakpoint 3 100 times
watch -location foo		break when foo changes value
condition 2 foo>10		break if foo>10 at breakpoint 2
continue	c	continue running
step [#]	s [#]	take # steps, follow into subroutines
next [#]	n [#]	take # steps, step over subroutines
finish		run to end of current subroutine (step out)
Threads		
info threads	i th	list threads
thread 4	t 4	switch context to thread 4

Figure 5.3: **gdb** Debugger Cheatsheet (description)

```
$ cd app/oxs
$ gdb linux-x86_64/oxs
(gdb) run boxsi.tcl examples/stdprob1.mif -threads 1
Starting program: oommf/app/oxs/linux-x86_64/oxs boxsi.tcl examples/stdp...
oxs: oommf/app/oxs/base/meshvalue.h:319: const T& Oxs_MeshValue<T>::oper...
  Assertion `0<=index && index<size' failed.

Thread 1 "oxs" received signal SIGABRT, Aborted.
0x00007ffff65d837f in raise () from /lib64/libc.so.6
(gdb) bt
#0  0x00007ffff65d837f in raise () from /lib64/libc.so.6
[...]
#4  0x000000000041012a in Oxs_MeshValue<double>::operator[]
  (this=0xcbeb58, index=40000) at oommf/app/oxs/base/meshvalue.h:319
#5  0x000000000061e88a in Oxs_UniaxialAnisotropy::RectIntegEnergy
  (this=0x1307d60, state=..., ocedt=..., ocedtaux=..., node_start=36864,
  node_stop=40000) at oommf/app/oxs/ext/uniaxialanisotropy.cc:241
[...]
(gdb) frame 5
#5  0x000000000061e88a in Oxs_UniaxialAnisotropy::RectIntegEnergy...
241           field_mult = (2.0/MU0)*k*Ms_inverse[i];
(gdb) set listsize 5
(gdb) list
239         if(aniscoeftype == K1_TYPE) {
240           if(!K1_is_uniform) k = K1[i];
241           field_mult = (2.0/MU0)*k*Ms_inverse[i];
242         } else {
243           if(!Ha_is_uniform) field_mult = Ha[i];
(gdb) print i
$1 = 40000
(gdb) print Ms_inverse
$2 = (const Oxs_MeshValue<double> &) @0xcbeb58: {arr = 0x7ffff7ebf000,
  size = 40000, arrblock = {datablock = 0x7ffff7ebe010 "",
  arr = 0x7ffff7ebf000, arr_size = 40000, strip_count = 1,
  strip_size = 320000, strip_pos = std::vector of length 2,
  capacity 2 = {0, 320000}}, static MIN_THREADING_SIZE = 10000}
(gdb) kill
Kill the program being debugged? (y or n) y
[Inferior 1 (process 1309854) killed]
```

Figure 5.4: Sample **gdb** session, part 1: Locating the error (description)

Two notes concerning **gdb** on macOS: First, as mentioned earlier, if you install **gdb** through MacPorts, the executable name is ggdb. Second, debuggers operate outside the

```
(gdb) break uniaxialanisotropy.cc:239
Breakpoint 1 at 0x61e811: file ext/uniaxialanisotropy.cc, line 239.
(gdb) run
Starting program: oommf/app/oxs/linux-x86_64/oxs boxsi.tcl examples/s...
[...]
Thread 1 "oxs" hit Breakpoint 1, Oxs_UniaxialAnisotropy::RectIntegEne...
239         if(aniscoeftype == K1_TYPE) {
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000061e811 in Oxs_UniaxialAni...
        breakpoint already hit 1 time
(gdb) ignore 1 39999
Will ignore next 39999 crossings of breakpoint 1.
(gdb) continue

Thread 1 "oxs" hit Breakpoint 1, Oxs_UniaxialAnisotropy::RectIntegEne...
239         if(aniscoeftype == K1_TYPE) {
(gdb) print i
$3 = 39991
(gdb) condition 1 i>=40000
(gdb) c

Thread 1 "oxs" hit Breakpoint 1, Oxs_UniaxialAnisotropy::RectIntegEne...
239         if(aniscoeftype == K1_TYPE) {
(gdb) l
237
238       for(OC_INDEX i=node_start;i<=node_stop;++i) {
239         if(aniscoeftype == K1_TYPE) {
240           if(!K1_is_uniform) k = K1[i];
241           field_mult = (2.0/MU0)*k*Ms_inverse[i];
(gdb) next
240           if(!K1_is_uniform) k = K1[i];
(gdb) n
241           field_mult = (2.0/MU0)*k*Ms_inverse[i];
(gdb) step
Oxs_MeshValue<double>::operator[] (this=0xcbeb58, index=40000)
  at oommf/app/oxs/base/meshvalue.h:319
319       assert(0<=index && index<size);
(gdb) printf "%d,%d\n", index, size
40000,40000
(gdb) quit
```

Figure 5.5: Sample **gdb** session, part 2: Bug details (description)

normal end-user program envelope and may run afoul of the OS security system. In particular to use **gdb** you may need to set up a certificate in the macOS System Keychain for it; details on this process can be found online. This issue might be resolved for **lldb** (next section) as part of the installation process if it and **clang++** were installed as part of the Xcode package.

This introduction only scratches the surface of **gdb** commands and capabilities. You can find tutorials and additional information online, or else refer to the **gdb** documentation from GNU for full details.

5.6.2 Introduction to the LLVM lldb

If you are working on macOS, you may be building OOMMF with **g++** or **clang++**. The native debugger for **clang++** is **lldb**, which is included as part of the Xcode package. Both **g++** and **clang++** use the same debugging symbol format, so in principle you should be able to use either debugger with either compiler, but if you have problems with one try the other.

The **lldb** debugger is a command-line debugger very similar in concept to **gdb**, and although the command syntax is somewhat different, **lldb** provides a fair number of aliases to ease the transition for veteran **gdb** users. Fig. 5.6 lists a few of the more common **lldb** commands, and Figs. 5.7 and 5.8 illustrate an **lldb** debugging session analogous to the **gdb** session presented in Figs. 5.4 and 5.5.

5.6.3 Debugging OOMMF in Visual Studio

The debugger built into Microsoft's Visual Studio provides largely similar functionality to **gdb** and **lldb**, but with a GUI interface. It understands the debugging symbol files produced by the Visual C++ `cl` compiler, namely "Program DataBase" files having the `.pdb` extension. Other debugger options for this symbol file format include the GUI **WinDbg** mentioned earlier, and the related command line tool **CDB**.

Visual Studio is an integrated development environment, and normal usage involves building "projects" that specify all the source code files and rules for building them into an executable program. OOMMF does not follow this paradigm, but rather maintains similar information in a collection of Tcl `makerules.tcl` files distributed across the development tree. Thus there is no OOMMF project file to load into Visual Studio. Instead, to debug an OOMMF application in Visual Studio you need to load the application executable directly, along with some supplemental run information. The following details the process for Visual Studio 2022; specifics may differ somewhat for other releases.

1. Launch Visual Studio
2. Select `Open a project or solution` from the `Getting started` pane and then navigate to and select the executable.
3. In the `Solution Explorer` pane, right click on the executable and select `Properties`.

Shell command: `lldb [-c corefile (opt)] darwin/oxs`

Command	**Abbr.**	**Description**
Process control		
process launch -- [*args*]	r [*args*]	run executable with *args*
process launch	r	run executable with last *args*
settings show target.run-args		display current *args*
settings set target.env-vars FOO=bar	env FOO=bar	set envr. variable FOO to “bar”
Ctrl-C		stop and return to (lldb) prompt
process kill	kill	terminate current run
quit		exit lldb
Introspection		
thread backtrace	bt	stack trace of current thread
frame select 5	f 5	change to stack frame 5
frame variable		print args & vars for current frame
frame variable foo	p foo	print value of variable foo
source list -f foo.cc -l 50	l foo.cc:50	list source after line 50 of foo.cc
source list	l	list next ten lines
source list -r	l -	list preceding ten lines
source list -c 20		list 20 lines
Flow control		
breakpoint set --file foo.cc --line 99		set breakpoint at line 99 of foo.cc
breakpoint set --name foo::bar		break at C++ routine foo::bar()
breakpoint list	br l	list breakpoints
breakpoint delete 4	br del 4	delete breakpoint 4
breakpoint delete	br del	delete all breakpoints
breakpoint modify -i 100 3		skip breakpoint 3 100 times
breakpoint modify -c i>7 3		break if i>7 at breakpoint 3
watchpoint set variable foo		break when foo changes value
thread continue	c	continue running
thread step-in	s	take one step, into subroutines
thread step-over	n	take one step, over subroutines
thread step-out	finish	run to end of current subroutine
Threads		
thread list		list all threads
thread select 2		switch context to thread 2

Figure 5.6: **lldb** Debugger Cheatsheet (description)

```
% cd app/oxs
% lldb darwin/oxs
(lldb) target create "darwin/oxs"
Current executable set to 'oommf/app/oxs/darwin/oxs' (x86_64).
(lldb) process launch -- boxsi.tcl examples/stdprob1.mif -threads 1
Process 36662 launched: 'oommf/app/oxs/darwin/oxs' (x86_64)
Assertion failed: (0<=index && index<size) [...] file meshvalue.h, line 319.
Process 36662 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = hit program assert
    frame #4: 0x00000001000065cc oxs [...] at meshvalue.h:319:3
   316  template<class T>
   317  const T& Oxs_MeshValue<T>::operator[](OC_INDEX index) const
   318  {
-> 319    assert(0<=index && index<size);
   320    return arr[index];
   321  }
   322
Target 0: (oxs) stopped.
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = hit program assert
    frame #0: 0x00007fff207ba91e libsystem_kernel.dylib`__pthread_kill + 10
[...]
  * frame #4: 0x00000001000065cc oxs`Oxs_MeshValue<double>::operator[](th...
    frame #5: 0x0000000100350fa8 oxs`Oxs_UniaxialAnisotropy::RectIntegEne...
[...]
(lldb) frame select 5
frame #5: 0x0000000100350fa8 oxs`Oxs_UniaxialAnisotropy::RectIntegEnergy(...
   238    for(OC_INDEX i=node_start;i<=node_stop;++i) {
   239      if(aniscoeftype == K1_TYPE) {
   240        if(!K1_is_uniform) k = K1[i];
-> 241        field_mult = (2.0/MU0)*k*Ms_inverse[i];
   242      } else {
   243        if(!Ha_is_uniform) field_mult = Ha[i];
   244        k = 0.5*MU0*field_mult*Ms[i];
(lldb) frame variable i
(OC_INDEX) i = 40000
(lldb) frame variable Ms_inverse
(const Oxs_MeshValue<double> &) Ms_inverse = 0x0000000102b77928: {
  arr = 0x0000000101da4000
  size = 40000
[...]
(lldb) process kill
Process 36662 exited with status = 9 (0x00000009)
```

Figure 5.7: Sample **lldb** session, part 1: Locating the error (description)

```
(lldb) breakpoint set --file uniaxialanisotropy.cc --line 239
Breakpoint 1: where = oxs`Oxs_UniaxialAnisotropy::RectIntegEnergy(Oxs_Sim...
(lldb) process launch
Process 36718 launched: 'oommf/app/oxs/darwin/oxs' (x86_64)
[...]
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
   238    for(OC_INDEX i=node_start;i<=node_stop;++i) {
-> 239      if(aniscoeftype == K1_TYPE) {
   240        if(!K1_is_uniform) k = K1[i];
   241        field_mult = (2.0/MU0)*k*Ms_inverse[i];
(lldb) breakpoint list
Current breakpoints:
1: file = 'uniaxialanisotropy.cc', line = 239, exact_match = 0, locations...
  1.1: where = oxs`Oxs_UniaxialAnisotropy::RectIntegEnergy(Oxs_SimState c...
(lldb) breakpoint modify -i 39999 1
(lldb) thread continue
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
-> 239      if(aniscoeftype == K1_TYPE) {
(lldb) p i
(OC_INDEX) $0 = 39991
(lldb) breakpoint modify -c i>=40000
(lldb) c
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
-> 239      if(aniscoeftype == K1_TYPE) {
(lldb) thread step-over
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
-> 240        if(!K1_is_uniform) k = K1[i];
(lldb) n
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
-> 241        field_mult = (2.0/MU0)*k*Ms_inverse[i];
(lldb) thread step-in
* thread #1, queue = 'com.apple.main-thread', stop reason = step in
   317   const T& Oxs_MeshValue<T>::operator[](OC_INDEX index) const
   318   {
-> 319     assert(0<=index && index<size);
(lldb) print (void) printf("%d,%d\n", index, size)
40000,40000
(lldb) quit
```

Figure 5.8: Sample **lldb** session, part 2: Bug details (lldb output edited for space) (description)

4. Under `Parameters`, fill in the `Arguments` and `Working Directory` fields as appropriate. You may also have to modify the `Environment` setting, in particular if the Tcl and Tk `.dll`'s are not on the default path used by Visual Studio. In this case click on the ellipsis at the right of the `Environment` row, and then click the `Fetch` button at the bottom of the `Environment` pop-up to load the current environment. Scroll down to variable `path` and edit as necessary. Close when complete.

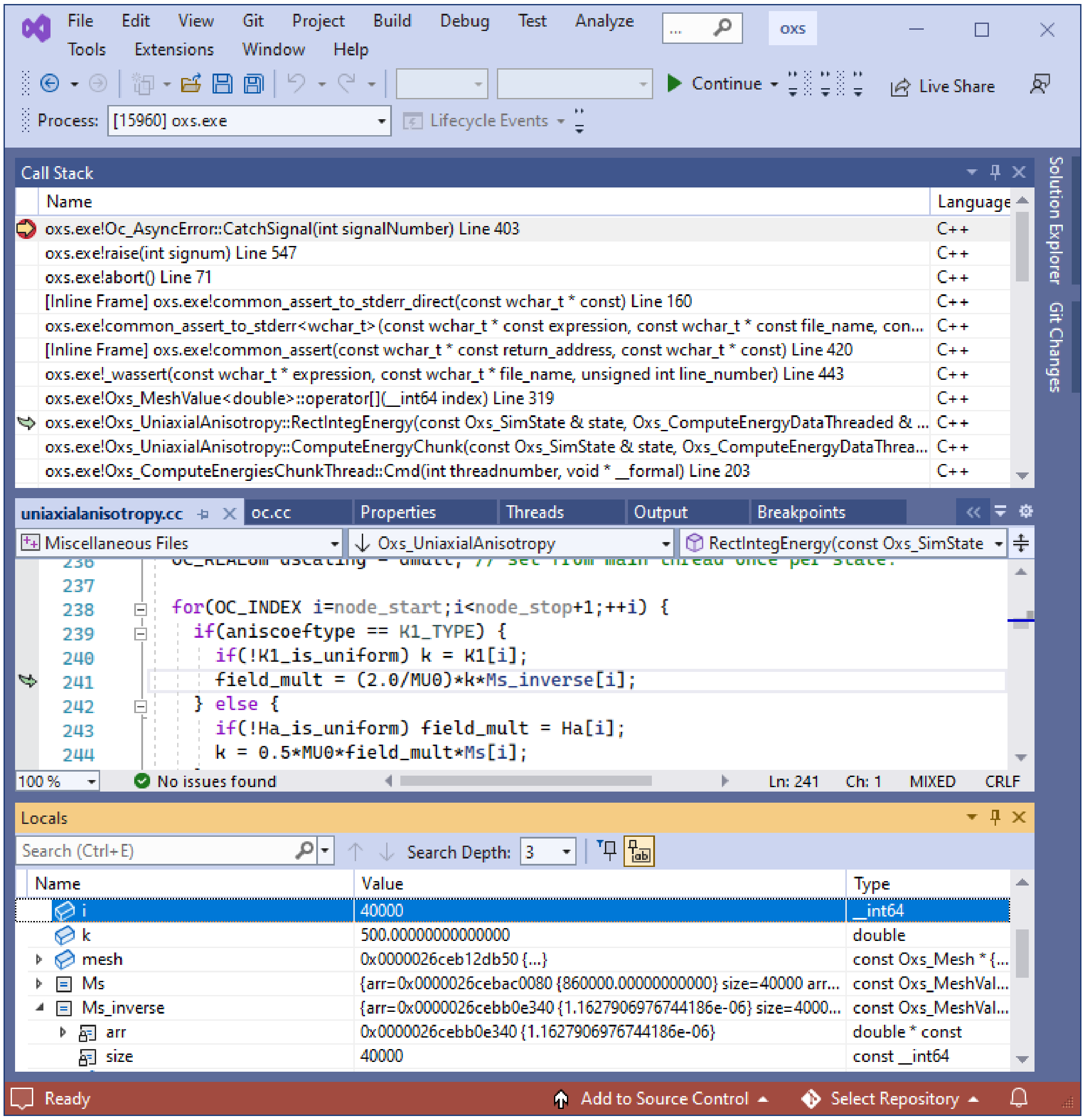
5. Select `Start` from the toolbar or `Debug|Start Debugging` from the top-level menu bar.

6. Debug! You can use the drop-down menus to perform actions analogous to those described above for the **gdb** and **lldb** debuggers. If you get a message that no symbols were loaded, then most likely either the `/Zi` switch was missing from the compile command or else the `/DEBUG` option was missing from the link command. In this case review the OOMMF configuration file settings (Sec. 5.1)) and rebuild OOMMF. The symbols for the executable should be stored in a `*.pdb` file next to the executable file.

7. The call stack should automatically appear when you start debugging. If not, you can manually call it up through the menu option `Debug|Windows|Call Stack`. A curious feature of Visual Studio is that the call stack window disappears when execution exits. This happens even when the exit is caused by an abnormal event, for example via an assertion failure. In default OOMMF builds many types of fatal errors are routed through the `Oc_AsyncError::CatchSignal(int)` routine in `pkg/oc/ocexcept.cc`. If you set a breakpoint in this function then the debugger will stop if it hits this function, but will not exit the debugger, so you can still examine the call stack. Do this before you start the debugging run by pulling up the `Debug|New Breakpoint|Function Breakpoint...` dialog, enter `Oc_AsyncError::CatchSignal(int)` in the "Function Name" box, and click "OK".

8. Double-clicking on a row in the Call Stack window will bring up the relevant line of source code. Menu option `Debug|Windows|Locals` will open a window showing the variable values accessible at this point in the code. An example is shown in Fig. 5.9, where we see that the index variable `i` at line 241 of `uniaxialanisotropy.cc` has value 40000, but the size of `Ms_inverse` is 40000, meaning the maximum valid index into `Ms_inverse` is only 39999.

9. When you exit the debugger you will be asked if you want to save the `.sln` (solution) file. If you do, it will be written in the same directory as the executable and `.pdb` files. In later debugging sessions you can load the solution file in step 2 above and bypass steps 3 and 4.



Process: [15960] oxs.exe

Call Stack

Name	Language
oxs.exe!Oc_AsyncError::CatchSignal(int signalNumber) Line 403	C++
oxs.exe!raise(int signum) Line 547	C++
oxs.exe!abort() Line 71	C++
[Inline Frame] oxs.exe!common_assert_to_stderr_direct(const wchar_t * const) Line 160	C++
oxs.exe!common_assert_to_stderr<wchar_t>(const wchar_t * const expression, const wchar_t * const file_name, con...	C++
[Inline Frame] oxs.exe!common_assert(const wchar_t * const return_address, const wchar_t * const) Line 420	C++
oxs.exe!_wassert(const wchar_t * expression, const wchar_t * file_name, unsigned int line_number) Line 443	C++
oxs.exe!Oxs_MeshValue<double>::operator[](__int64 index) Line 319	C++
oxs.exe!Oxs_UniaxialAnisotropy::RectIntegEnergy(const Oxs_SimState & state, Oxs_ComputeEnergyDataThreaded & ...	C++
oxs.exe!Oxs_UniaxialAnisotropy::ComputeEnergyChunk(const Oxs_SimState & state, Oxs_ComputeEnergyDataThrea...	C++
oxs.exe!Oxs_ComputeEnergiesChunkThread::Cmd(int threadnumber, void * __formal) Line 203	C++

```
  OC_REALWIDE dscaling = dmult; // set from main thread once per state.

  for(OC_INDEX i=node_start;i<node_stop+1;++i) {
    if(aniscoeftype == K1_TYPE) {
      if(!K1_is_uniform) k = K1[i];
      field_mult = (2.0/MU0)*k*Ms_inverse[i];
    } else {
      if(!Ha_is_uniform) field_mult = Ha[i];
      k = 0.5*MU0*field_mult*Ms[i];
```

100 %   No issues found   Ln: 241   Ch: 1   MIXED   CRLF

Locals

Search Depth: 3

Name	Value	Type
i	40000	__int64
k	500.00000000000000	double
mesh	0x0000026ceb12db50 {...}	const Oxs_Mesh * {...
Ms	{arr=0x0000026cebac0080 {860000.00000000000} size=40000 arr...	const Oxs_MeshVal...
Ms_inverse	{arr=0x0000026cebb0e340 {1.1627906976744186e-06} size=4000...	const Oxs_MeshVal...
arr	0x0000026cebb0e340 {1.1627906976744186e-06}	double * const
size	40000	const __int64

Figure 5.9: **Visual Studio Debugger** screenshot displaying call stack, source code, and local variables from a debugging session.

Credits

The main contributors to this document are Michael J. Donahue (michael.donahue@nist.gov) and Donald G. Porter (donald.porter@nist.gov), both of ITL/NIST.

If you have bug reports, contributed code, feature requests, or other comments for the OOMMF developers, please send them in an e-mail message to `<michael.donahue@nist.gov>` or `<donald.porter@nist.gov>`.

Bibliography

[1] W. F. Brown, Jr., *Micromagnetics* (J. Wiley, New York, 1963).

[2] M. J. Donahue and D. G. Porter, *OOMMF User's Guide, Version 1.0*, Tech. Rep. NISTIR 6376, National Institute of Standards and Technology, Gaithersburg, MD (1999).

Index