Adept C++ Software Library: User Guide

Robin J. Hogan

European Centre for Medium Range Weather Forecasts, Reading, UK
and School of Mathematical, Physical and Computational Sciences, University of Reading, UK,

Document version 2.0.8 (August 2020) applicable to Adept version 2.0.8
# Contents

1 Introduction 3
   1.1 What is Adept? ................................................................. 3
   1.2 Installing Adept and compiling your code to use it ................................................................. 3
      1.2.1 Unix-like platforms ................................................. 4
      1.2.2 Non-Unix platforms, and compiling Adept applications without linking to an external library 6

2 Using Adept for automatic differentiation 7
   2.1 Introduction ................................................................. 7
   2.2 Code preparation .......................................................... 8
   2.3 Applying reverse-mode differentiation .................................................. 9
      2.3.1 Set-up stack to record derivative information .................................................. 9
      2.3.2 Initialize independent variables and start recording ............................................ 10
      2.3.3 Perform calculations to be differentiated ......................................................... 10
      2.3.4 Perform reverse-mode differentiation ............................................................ 10
      2.3.5 Extract the final gradients ............................................................................. 11
   2.4 Computing Jacobian matrices ................................................. 11
   2.5 Real-world usage: interfacing Adept to a minimization library ........................................... 12
   2.6 Calling an algorithm with and without automatic differentiation from the same program 15
      2.6.1 Function templates .................................................. 15
      2.6.2 Pausable recording .................................................. 16
      2.6.3 Multiple object files per source file ......................................................... 16
   2.7 Interfacing with software containing hand-coded Jacobians ................................. 17
   2.8 Member functions of the Stack class ............................................ 19
   2.9 Member functions of the adouble object .............................................. 22

3 Using Adept's array functionality 24
   3.1 Introduction ................................................................. 24
   3.2 The Array class ............................................................... 25
   3.3 Operators and mathematical functions ................................................ 28
   3.4 Array slicing ........................................................................ 28
   3.5 Passing arrays to and from functions ................................................ 30
   3.6 Array reduction operations .................................................... 32
   3.7 Array expansion operations .................................................... 32
   3.8 Conditional operations ......................................................... 33
   3.9 Fixed-size arrays .................................................................... 33
   3.10 Special square matrices ....................................................... 34
   3.11 Matrix multiplication ........................................................ 35
   3.12 Linear algebra ...................................................................... 36
   3.13 Bounds and alias checking .................................................... 37
   3.14 Automatic differentiation capabilities specific to arrays .................................................. 38
   3.15 Array thread safety .......................................................... 38
   3.16 Writing an array to a stream .................................................. 38
   3.17 Fortran interoperability ......................................................... 39
      3.17.1 Passing arrays from C++/Adept to Fortran .................................................... 39
      3.17.2 Passing arrays from Fortran to C++/Adept ..................................................... 40

4 Using Adept's optimization functionality 42
   4.1 Background ................................................................. 42
   4.2 Adept interface................................................................. 43
   4.3 Other member functions of the Minimizer class ................................................ 44
4.4 Return status for minimization .............................................................................................................. 46

5 General considerations 47
  5.1 Setting and checking the global configuration .................................................................................... 47
  5.2 Parallelizing Adept programs .............................................................................................................. 47
  5.3 The fast exponential function .............................................................................................................. 48
  5.4 Tips for the best performance .............................................................................................................. 48
  5.5 Exceptions thrown by the Adept library .............................................................................................. 49
    5.5.1 General exceptions ..................................................................................................................... 50
    5.5.2 Automatic-differentiation exceptions ........................................................................................... 50
    5.5.3 Array exceptions ........................................................................................................................ 51
  5.6 Configuring the behaviour of Adept .................................................................................................... 51
    5.6.1 Modifications not requiring a library recompile ........................................................................... 51
    5.6.2 Modifications requiring a library recompile .................................................................................. 53
  5.7 Frequently asked questions .................................................................................................................. 53
  5.8 Copyright and license for Adept software ............................................................................................ 55
Chapter 1

Introduction

1.1 What is Adept?

Adept (Automatic Differentiation using Expression Templates) is a C++ software library that enables algorithms to be automatically differentiated. Since version 2.0* it also provides array classes that can be used in array expressions. These two capabilities are fully integrated such that array expressions can be differentiated efficiently, but the array capability may also be used on its own.

The automatic-differentiation capability uses an operator overloading approach, so very little code modification is required. Differentiation can be performed in forward mode (the “tangent-linear” computation), reverse mode (the “adjoint” computation), or the full Jacobian matrix can be computed. This behaviour is common to several other libraries, namely ADOL-C (Griewank et al., 1996), CppAD (Bell, 2007) and Sacado (Gay, 2005), but the use of expression templates, an efficient way to store the differential information and several other optimizations mean that reverse-mode differentiation tends to be significantly faster and use less memory. In fact, Adept is also usually only a little slower than an adjoint code you might write by hand, but immeasurably faster in terms of user time; adjoint coding is very time consuming and error-prone. For technical details of how it works, benchmark results and further discussion of the factors affecting its speed when applied to a particular code, see Hogan (2014).

Expression templates also underpin a number of libraries that provide the capability to perform mathematical operations on entire arrays (Veldhuizen, 1995). Unfortunately, if Adept version 1.x and such an array library are used together, then the speed advantages of expression templates are lost, if indeed the libraries work together at all. Since version 2.0, Adept provides array classes that overcome this problem: its automatic differentiation and array capabilities are underpinned by a single unified expression template framework so that array expressions may be differentiated very efficiently. However, it should be stressed that Adept is useful as a fully functional array library even if you don’t wish to use its automatic differentiation capability. Adept uses BLAS and LAPACK for matrix operations.

This user guide describes how to apply the Adept software library to your code, and many of the examples map on to those in the test directory of the Adept software package. Section 1.2 outlines how to install Adept on your system and how to compile your own code to use it. Chapter 2 describes how to use the automatic differentiation capability of the library, chapter 3 its array capability and chapter 4 its optimization capability. Chapter 5 then describes general aspects such as exception handling, configuration options and license terms.

1.2 Installing Adept and compiling your code to use it

Adept should work with any C++98 compliant compiler, but uses some C++11 features if compiled with support for this later standard. Most of the testing has been on Linux with the GNU C++ compiler, but it also compiles on Linux with the Clang and Intel compilers and on Windows with the Microsoft compiler. The code is built with the help of a configure shell script generated by GNU autotools. If you are on a non-Unix system (e.g. Windows) and cannot use shell scripts, see section 1.2.2.

*Note that the version 1.9.x series served as beta releases for version 2.0 of Adept.

3
1.2 Installing Adept and compiling your code to use it

1.2.1 Unix-like platforms

On a Unix-like system, do the following:

1. Install the BLAS library to enable matrix multiplication. For the best performance in matrix operations it is recommended that you install an optimized package such as OpenBLAS† or ATLAS‡. If you have multiple BLAS libraries available on your system you can specify the one you want by calling the configure script below with `--with-blas=openblas` or similar. If Adept is compiled without BLAS support then matrix multiplication will fail at run time.

2. Optionally install the LAPACK library, necessary for matrix inversion and solving linear systems of equations. If you do not install this then Adept will still compile but the functions `inv` and `solve` will fail at run time. Note that LAPACK relies on the underlying BLAS library for its speed.

3. The test and benchmarking programs can make use of additional libraries if available. If you also install any of the automatic differentiation tools ADOL-C, CppAD and/or Sacado then the benchmarking test program can compare them to Adept. One of the test programs uses the minimization algorithm from the GNU Scientific Library, if available, so you may wish to install that too.

4. Unpack the package (`tar xvfz adept-2.x.tar.gz` on Linux) and `cd` to the directory `adept-2.x`.

5. Configure the build using the `configure` script. The most basic method is to just run

   ```bash
   ./configure
   ```

   More likely you will wish to compile with a higher level of optimization than the default (which is `-O2`), achieved by setting the environment variable `CXXFLAGS`. You may also wish to specify the root directory of the installation, say to `/foo`. These may be done by running instead

   ```bash
   ./configure CXXFLAGS="-g -O3" --prefix=/foo
   ```

   The `-g` option to `CXXFLAGS` ensures debugging information is stored. If you use the GNU compiler then consider the `-g1` option instead to reduce the amount of debugging information stored. The GNU `-march=native` option will also enable the fastest instruction set for the machine on which the code is being compiled. Adept can vectorize certain floating-point array expressions making use of the SSE2, AVX and AVX512 instruction sets on Intel hardware and the NEON instruction set on ARM. If a library you wish to use is installed in a non-system directory, say under `/foo`, then specify the locations as follows:

   ```bash
   ./configure CPPFLAGS="-I/foo/include" LDFLAGS="-L/foo/lib -Wl,-rpath,/foo/lib"
   ```

   where the `-rpath` business is needed in order that the Adept shared library knows where to look for the libraries it is dependent on. If you have them then for the benchmarking program you can also add the non-system location of ADOL-C, CppAD and Sacado libraries with additional `-I` and `-L` arguments, but note that the `-rpath` argument is not needed in that case. You can see the more general options available by running `./configure --help`; for example, you can turn-off OpenMP parallelization in the computation of Jacobian matrices using `--disable-openmp`. See also section 5.6 for ways to make more fundamental changes to the configuration of Adept. The output from the `configure` script provides information on aspects of how Adept and the test programs will be built.

6. Build Adept by running

   ```bash
   make
   ```

   This will create the static and shared libraries in `adept/.libs`.

7. Install the header files and the static and shared libraries by running

   ```bash
   make install
   ```

†OpenBLAS is available from `http://www.openblas.net/`.
‡ATLAS is available from `http://math-atlas.sourceforge.net/`.
1.2. Installing Adept and compiling your code to use it

If this is to be installed to a system directory, you will need to log in as the super-user first, or run `sudo make install` on depending on your system.

8. Build and run the test programs by running

```
make check
```

Note that this may be done without first installing the Adept library to a system directory. This compiles the following programs in the `test` directory:

- `test_misc`: the trivial example from Hogan (2014);
- `test_addept`: compares the results of numerical and automatic differentiation;
- `test_with_without_ad`: does the same but compiling the same source code both with and without automatic differentiation (see `test/Makefile` for how this is done);
- `test_radiances`: demonstrates the interfacing of Adept with code that provides its own Jacobian;
- `test_gsl_interface`: implementation of a simple minimization problem using the L-BFGS minimizer in the GSL library;
- `test_checkpoint`: demonstration of checkpointing, a useful technique for large codes;
- `test_thread_safe`: demonstration of the use of multiple OpenMP threads, each with its own instance of an Adept stack;
- `test_no_lib`: demonstrates the use of the `adept_source.h` header file that means there is no need to link to the Adept library in order to create an executable.
- `test_arrays`: tests many of the array capabilities described in chapter 3.
- `test_arrays_active`: tests the same array capabilities but with active arrays.
- `test_arrays_active_pausage`: tests that active array operations can be paused as described in section 2.6.2.
- `test_fixed_arrays`: tests the similar capabilities of arrays with fixed dimensions (known at compile time).
- `test_fixed_arrays_active`: as `test_fixed_arrays` but with active arrays.
- `test_constructors`: test the different ways of constructing, assigning and linking arrays, and passing them to and from functions.
- `test_derivatives`: tests that all mathematical functions supported by Adept differentiate correctly.
- `test_array_derivatives`: tests that selected array operations differentiate correctly.
- `test_thread_safe_arrays`: tests two ways to ensure arrays may be accessed and subsetted safely in a multi-threaded environment.

After compiling these programs, they are run one by one; if any fail due to an incorrect result then `make check` will fail.

The `make check` operation also compiles `autodiff_benchmark` in the `benchmark` directory for comparing the speed of the differentiation of two advection algorithms using Adept, ADOL-C, CppAD and Sacado (or whichever subset of these tools you have on your system). It also compiles `animate` for visualizing at a terminal what the algorithms are doing. Further information on running these programs can be found in the `README` files in the relevant directories.

To compile source files that use the Adept library, you need to make sure that `adept.h` and `adept_arrays.h` are in your include path. If they are located in a directory that is not in the default include path, add something like `-I/home/fred/include` to the compiler command line. At the linking stage, add `-ladept` to the command line to tell the linker to look for the `libadept.a` static library, or equivalent shared library. If this file is in a non-standard location, also add something like `-L/home/fred/lib -Wl,-rpath,/home/fred/lib` before the `-ladept` argument to specify its location. Section 2.6.3 provides an example Makefile for compiling code that uses the Adept library. Read on to see how you can compile an Adept application without needing to link to a library.
1.2. Non-Unix platforms, and compiling Adept applications without linking to an external library

Most of the difficulty in maintaining software that can compile on multiple platforms arises from the different ways of compiling software libraries, and the need to test on compilers that may be proprietary. Unfortunately I don’t have the time to maintain versions of Adept that build specifically on Microsoft Windows or other non-Unix platforms. However, Adept is not a large library, so I have provided a very simple way to build an Adept application without the need to link to a pre-compiled Adept library. In one of your source files and one only, add this near the top:

```c
#include <adept_source.h>
```

Typically you would include this in the source file containing the main function. This header file is simply a concatenation of the Adept library source files, so when you compile a file that includes it, you compile in all the functionally of the Adept library. All other source files in your application should include only the adept.h or adept_arrays.h header file as normal. When you link all your object files together to make an executable, the Adept functionality will be built in, even though you did not link to an external Adept library.

By default, adept_arrays.h does not enable BLAS (needed for matrix multiplication) or LAPACK (needed for matrix inversion and solving linear systems of equations); to enable either BLAS alone, or both BLAS and LAPACK, uncomment the lines near the top of adept_source.h defining HAVE_BLAS and HAVE_LAPACK, and link against functioning BLAS and LAPACK library. A demonstration of the use of adept_source.h is in the test/test_no_lib.cpp source file, which needs to be compiled together with test/algorithm.cpp to make an executable. It is hoped that this feature will make it easy to use Adept on non-Unix platforms, although of course this feature works just as well on Unix-like platforms as well.

A further point to note is that, under the terms of the license, it is permitted to copy all the Adept include files, including adept_source.h, into an include directory in your software package and use it from there in both binary and source-code releases of your software. This means that users do not need to install Adept separately before they use your software. However, if you do this then remember that your use of these files must comply with the terms of the Apache License, Version 2.0; see section 5.8 for details.
Chapter 2

Using Adept for automatic differentiation

2.1 Introduction

This chapter describes how to use Adept to differentiate your code. For simplicity, none of the examples use array functionality described in the next chapter. Adept provides the following automatic-differentiation functionality:

**Full Jacobian matrix** Given the non-linear function \( y = f(x) \) relating vector \( y \) to vector \( x \) coded in C or C++, after a little code modification Adept can compute the Jacobian matrix \( H = \frac{\partial y}{\partial x} \), where the element at row \( i \) and column \( j \) of \( H \) is \( H_{ij} = \frac{\partial y_i}{\partial x_j} \). This matrix will be computed much more rapidly and accurately than if you simply recompute the function multiple times, each time perturbing a different element of \( x \) by a small amount. The Jacobian matrix is used in the Gauss-Newton and Levenberg-Marquardt minimization algorithms.

**Reverse-mode differentiation** This is a key component in optimization problems where a non-linear function needs to be minimized but the state vector \( x \) is too large for it to make sense to compute the full Jacobian matrix. Atmospheric data assimilation is the canonical example in the field of meteorology. Given a non-linear function \( J(x) \) relating the scalar to be minimized \( J \) to vector \( x \), Adept will compute the vector of adjoints \( \frac{\partial J}{\partial x} \). Moreover, for a component of the code that may be expressed as a multi-dimensional non-linear function \( y = f(x) \), Adept can compute \( \frac{\partial J}{\partial x} \) if it is provided with the vector of input adjoints \( \frac{\partial J}{\partial y} \). In this case, \( \frac{\partial J}{\partial x} \) is equal to the matrix-vector product \( H^T \frac{\partial J}{\partial y} \), but it is computed here without computing the full Jacobian matrix \( H \). The vector \( \frac{\partial J}{\partial x} \) may then be used in a quasi-Newton minimization scheme (e.g., Liu and Nocedal, 1989).

**Forward-mode differentiation** Given the non-linear function \( y = f(x) \) and a vector of perturbations \( \delta x \), Adept will compute the corresponding vector \( \delta y \) arising from a linearization of the function \( f \). Formally, \( \delta y \) is equal to the matrix-vector product \( H \delta x \), but it is computed here without computing the full Jacobian matrix \( H \). Note that Adept is designed for the reverse case, so might not be as fast or economical in memory in the forward mode as libraries written especially for that purpose (although Hogan, 2014, showed that it was competitive).

Adept can automatically differentiate the following operators and functions:

- The standard binary mathematical operators +, -, *, and /.
- The assignment versions of these operators: +=, -=, *= and /=.
- The unary mathematical functions sqrt, exp, log, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, abs, asinh, acosh, atanh, expm1, log1p, cbrt, erf, erfc, exp2, log2, round, trunc, rint and nearbyint.
- The binary functions pow, atan2, min, max, fmin and fmax.
2.2 Code preparation

Variables to take part in expressions to be differentiated have a special “active” type; such variables can take part in comparison operations ==, !=, >, <, >= and <=, as well as the diagnostic functions isfinite, isinf and isnan.

Note that at present Adept is missing some functionality that you may require:

- Differentiation is first-order only: it cannot directly compute higher-order derivatives such as the Hessian matrix, although section ?? describes how Adept can help compute the Hessian if the cost function (also known as the penalty function or objective function) is in a particular commonly used form.
- It has limited support for complex numbers; no support for mathematical functions of complex numbers, and expressions involving operations (addition, subtraction, multiplication and division) on complex numbers are not optimized.
- It can be applied to C and C++ only; Adept could not be written in Fortran since the language provides no template capability.

It is hoped that future versions will remedy these limitations (and maybe even a future version of Fortran will support templates).

Section 2.2 describes how to prepare your code for automatic differentiation, and section 2.3 describes how to perform forward- and reverse-mode automatic differentiation on this code. Section 2.4 describes how to compute Jacobian matrices. Section 2.5 provides a detailed description of how to interface an algorithm implemented using Adept with a third-party minimization library. Section 2.6 describes how to call a function both with and without automatic differentiation from within the same program. Section 2.7 describes how to interface to software modules that compute their own Jacobians. Section 2.8 describes the user-oriented member functions of the Stack class that contains the differential information and section 2.9 describes the member functions of the “active” double-precision type adouble.

2.2 Code preparation

If you have used ADOL-C, CppAD or Sacado then you will already be familiar with what is involved in applying an operator-overloading automatic differentiation package to your code. The user interface to Adept differs from these only in the detail. It is assumed that you have an algorithm written in C or C++ that you wish to differentiate. This section deals with the modifications needed to your code, while section 2.3 describes the small additional amount of code you need to write to differentiate it.

In all source files containing code to be differentiated, you need to include the adept.h header file and import the adouble type from the adept namespace. Assuming your code uses double precision, you then search and replace double with the “active” equivalent adouble, but doing this only for those variables whose values depend on the independent input variables. Under the hood this type is an alias for Active<double>. The single-precision equivalent is afloat, an alias for Active<float>. Active and passive variables of single and double precision may be used together in the same expressions, but note that by default all differentiation is done in double precision.

If you wish to enable your code to be easily recompiled to use different precisions, then you may alternatively use the generic Real type from the adept namespace with its active equivalent aReal (an alias for Active<Real>). Section 5.6 describes how to redefine Real to represent single, double or quadruple precision. Automatic differentiation will be performed using the same precision as Real, but be aware that if this is defined to be the same as a single-precision float, accumulation of round-off error can make the accuracy of derivatives insufficient for minimization algorithms. The examples in the remainder of this chapter use only double precision.

Consider the following contrived algorithm from Hogan (2014) that takes two inputs and returns one output:

```c
double algorithm(const double x[2]) {
    double y = 4.0;
    double s = 2.0*x[0] + 3.0*x[1]*x[1];
    y *= sin(s);
    return y;
}
```

The modified code would look like this:
#include <adept.h>
using adept::adouble;

adouble algorithm(const adouble x[2]) {
    adouble y = 4.0;
    adouble s = 2.0*x[0] + 3.0*x[1]*x[1];
y *= sin(s);
    return y;
}

Changes like this need to be done in all source files that form part of an algorithm to be differentiated.

If you need to access the real number underlying an adouble variable a, for example in order to use it as an argument to the fprintf function, then use a.value() or adept::value(a). Any mathematical operations performed on this real number will not be differentiated.

You may use adouble as the template argument of a Standard Template Library (STL) vector type (i.e. std::vector<adouble>), or indeed any container where you access individual elements one by one. For types allowing mathematical operations on the whole object, such as the STL complex and valarray types, you will find that although you can multiply one std::complex<adouble> or std::valarray<adouble> object by another, mathematical functions (exp, sin etc.) will not work when applied to whole objects, and neither will some simple operations such as multiplying these types by an ordinary (non-active) double variable. Moreover, the performance is not great because expressions cannot be fully optimized when in these containers. Therefore if you need array functionality then you should use the features described in chapter 3. It is hoped that a future version of Adept will include its own complex type.

## 2.3 Applying reverse-mode differentiation

Suppose you wanted to create a version of algorithm that returned not only the result but also the gradient of the result with respect to its inputs, you would do this:

```cpp
#include <adept.h>

double algorithm_and_gradient(const double x_val[2], double dy_dx[2]) {
    // Input values
    adept::Stack stack; // Where the derivative information is stored
    using adept::adouble;
    double x[2] = {x_val[0], x_val[1]}; // Initialize active input variables
    stack.new_recording(); // Start recording
    adouble y = algorithm(x); // Call version overloaded for adouble args
    y.set_gradient(1.0); // Defines y as the cost function
    stack.compute_adjoint(); // Run the adjoint algorithm
    dy_dx[0] = x[0].get_gradient(); // Store the first gradient
    dy_dx[1] = x[1].get_gradient(); // Store the second gradient
    return y.value(); // Return the result of the simple computation
}
```

The component parts of this function are in a specific order, and if this order is violated then the code will not run correctly. The steps are now described.

### 2.3.1 Set-up stack to record derivative information

```cpp
adept::Stack stack;
```

The Stack object is where the differential version of the algorithm will be stored. When initialized, it makes itself accessible to subsequent statements via a global variable, but using thread-local storage to ensure thread safety. It must be initialized before the first adouble object is instantiated and it must not go out of scope until the last adouble object is destructed. This is because adouble objects register themselves with the currently active stack, and deregister themselves when they are destroyed; if the same stack is not active throughout the lifetime of such adouble objects then the code will crash with a segmentation fault.
2.3. Applying reverse-mode differentiation

In the example here, the Stack object is local to the scope of the function. If another Stack object had been initialized by the calling function and so was active at the point of entry to the function, then the local Stack object would throw an adept::stack_already_active exception (see Test 3 described at test/README in the Adept package if you want to use multiple Stack objects in the same program). A disadvantage of local Stack objects is that the memory it uses must be deallocated each time the function is called. This can be overcome in several ways:

- Declare the Stack object to be static, which means that it will persist between function calls. This has the disadvantage that you won’t be able to use other Stack objects in the program without deactivating this one first (see Test 3 in the Adept package, referred to above, for how to do this).

- Initialize Stack at a higher level in the program. If you need access to the stack, you may either pass a reference to it to functions such as algorithm_and_gradient, or alternatively you can use the adept::active_stack() function to return a pointer to the currently active stack object.

- Put it in a class so that it is accessible to member functions; this approach is demonstrated in section 2.5.

2.3.2 Initialize independent variables and start recording

```cpp
adouble x[2] = {x_val[0], x_val[1]};
stack.new_recording();
```

The first line here simply copies the input values to the algorithm into adouble variables. These are the independent variables, but note that there is no obligation for these to be stored as one array (as in CppAD), and for forward- and reverse-mode automatic differentiation you do not need to tell Adept explicitly via a function call which variables are the independent ones. The next line clears all differential statements from the stack so that it is ready for a new recording of differential information. Note that the first line here actually stores two differential statements, $\delta x[0]=0$ and $\delta x[1]=0$, which are immediately cleared by the new_recording function call. To avoid the small overhead of storing redundant information on the stack, we could replace the first line with

```cpp
x[0].set_value(x_val[0]);
x[1].set_value(x_val[1]);
```

or

```cpp
adep::set_values(x, 2, x_val);
```

which have the effect of setting the values of $x$ without storing the equivalent differential statements.

Previous users of Adept version 0.9 should note that since version 1.0, the new_recording function replaces the start function call, which had to be put before the independent variables were initialized. The problem with this was that the independent variables had to be initialized with the set_value or set_values functions, otherwise the gradients coming out of the automatic differentiation would all be zero. Since it was easy to forget this, new_recording was introduced to allow the independent variables to be assigned in the normal way using the assignment operator (=). But don’t just replace start in your version-0.9-compatible code with new_recording; the latter must appear after the independent variables have been initialized.

2.3.3 Perform calculations to be differentiated

```cpp
adouble y = algorithm(x);;
```

The algorithm is called, and behind the scenes the equivalent differential statement for every mathematical statement is stored in the stack. The result of the forward calculation is stored in $y$, known as a dependent variable. This example has one dependent variable, but any number is allowed, and they could be returned in another way, e.g. by passing a non-constant array to algorithm that is filled with the final values when the function returns.

2.3.4 Perform reverse-mode differentiation

```cpp
y.set_gradient(1.0);
stack.compute_adjoint();
```
2.4. Computing Jacobian matrices

The first line sets the initial gradient (or adjoint) of \( y \). In this example, we want the output gradients to be the derivatives of \( y \) with respect to each of the independent variables; to achieve this, the initial gradient of \( y \) must be unity.

More generally, if \( y \) was only an intermediate value in the computation of cost function \( J \), then for the outputs of the function to be the derivatives of \( J \) with respect to each of the independent variables, we would need to set the gradient of \( y \) to \( \partial J / \partial y \). In the case of multiple intermediate values, a separate call to `set_gradient` is needed for each intermediate value. If \( y \) was an array of length \( n \) then the gradient of each element could be set to the values in a double array \( y_{\text{ad}} \) using

\[
\texttt{adept::set_gradients}(y, n, y_{\text{ad}});
\]

The `compute_adjoint()` member function of stack performs the adjoint calculation, sweeping in reverse through the differential statements stored on the stack. Note that this must be preceded by at least one `set_gradient` or `set_gradients` call, since the first such call initializes the list of gradients for `compute_adjoint()` to act on. Otherwise, `compute_adjoint()` will throw a `gradients_not_initialized` exception.

2.3.5 Extract the final gradients

\[
dx_0 \leftarrow x[0].\text{get\_gradient}();
\]
\[
dx_1 \leftarrow x[1].\text{get\_gradient}();
\]

These lines simply extract the gradients of the cost function with respect to the two independent variables. Alternatively we could have extracted them simultaneously using

\[
\texttt{adept::get\_gradients}(x, 2, dx);
\]

To do forward-mode differentiation in this example would involve setting the initial gradients of \( x \) instead of \( y \), calling the member function `compute_tangent_linear()` instead of `compute_adjoint()`, and extracting the final gradients from \( y \) instead of \( x \).

2.4 Computing Jacobian matrices

Until now we have considered a function with two inputs and one output. Consider the following more general function whose declaration is

\[
\textbf{void} \ \text{algorithm2}(\textbf{int} \ n, \ \textbf{const} \ \textbf{adouble*} \ x, \ \textbf{int} \ m, \ \textbf{adouble*} \ y);
\]

where \( x \) points to the \( n \) independent (input) variables and \( y \) points to the \( m \) dependent (output) variables. The following function would return the full Jacobian matrix:

\[
\texttt{#include <vector>}
\]
\[
\texttt{#include <adept.h>}
\]
\[
\texttt{void} \ \text{algorithm2\_jacobian}(\textbf{int} \ n, \ \textbf{const} \ \textbf{adouble*} \ x\_\text{val}, \ \textbf{int} \ m, \ \textbf{adouble*} \ y\_\text{val}, \ \textbf{double*} \ jac) \ { \ 
\begin{align*}
\texttt{using} \ & \texttt{adept::adouble;} \\
\texttt{adept::Stack stack;} \\
\texttt{std::vector<adouble>} & x(n); \\
\texttt{adept::set\_values}\{x[0], n, x\_val\}; \\
\texttt{stack.\text{new\_recording}();} \\
\texttt{std::vector<adouble>} & y(m); \\
\texttt{algorithm2}(n, 4x[0], m, 6y[0]); \\
\texttt{stack.\text{independent}\{4x[0], n\};} \\
\texttt{stack.\text{dependent}\{6y[0], m\};} \\
\texttt{stack.\text{jacobian}\{jac\};} \\
\texttt{for} \ (\textbf{int} \ iy = 0; \ iy < m; ++iy) \\
\texttt{\quad y\_val[iy] \leftarrow y[iy].\text{value}();} \\
\end{align*}
\}
\]

Note that:
• The independent member function of stack is used to identify the independent variables, i.e. the variables that the derivatives in the Jacobian matrix will be with respect to. In this example there are \( n \) independent variables located together in memory and so can be identified all at once. Multiple calls are possible to identify further independent variables. To identify a single independent variable, call independent with just one argument, the independent variable (not as a pointer).

• The dependent member function of stack identifies the dependent variables, and its usage is identical to independent.

• The memory provided to store the Jacobian matrix (pointed to by jac) must be a one-dimensional array of size \( m \times n \), where \( m \) is the number of dependent variables and \( n \) is the number of independent variables.

• The resulting matrix is stored in the sense of the index representing the dependent variables varying fastest (column-major order).

• Internally, the Jacobian calculation is performed by multiple forward or reverse passes, whichever would be faster (dependent on the numbers of independent and dependent variables).

• The use of `std::vector<adouble>` rather than `new adouble[n]` ensures no memory leaks in the case of an exception being thrown, since the memory associated with \( x \) and \( y \) will be automatically deallocated when they go out of scope.

As described in chapter 3, Adept version 2.0 introduced built-in multi-dimensional arrays of both active (e.g. aVector) and passive (e.g. Vector) variables. It therefore seems more natural to express the algorithm above in terms of these objects, which could be done as follows:

```cpp
#include <adept_arrays.h>

// Adept vectors know their own length, so lengths do not need to be // passed in as well
adept::aVector algorithm2(const adept::aVector& x);

void algorithm2_jacobian{
    const adept::Vector& x_val, // Input values
    adept::Vector& y_val, // Output values (correctly sized or empty)
    adept::Matrix& jac} {
    // Output Jacobian matrix (correctly sized)
    adept::Stack stack; // Where the derivative information is stored
    adept::aVector x = x_val; // Active vector of inputs
    stack.new_recording(); // Start recording
    adept::aVector y = algorithm2(x); // Run algorithm and store outputs
    stack.independent(x); // Identify independent variables
    stack.dependent(y); // Identify dependent variables
    stack.jacobian(jac); // Compute & store Jacobian (since Adept 2.0.8)
    // If jac is empty we can automatically resize it using this instead (since Adept 2.0.8):
    //jac = stack.jacobian();
    y_val = value(y); // Extract the values from the active array
}
```

### 2.5 Real-world usage: interfacing Adept to a minimization library

Suppose we want to find the vector \( x \) that minimizes an cost function \( J(x) \) that consists of a large algorithm coded using the Adept library and encapsulated within a C++ class. In this section we illustrate how it may be interfaced to a third-party minimization algorithm with a C-style interface, specifically the free one in the GNU Scientific Library. The full working version of this example, using the N-dimensional Rosenbrock banana function as the function to be minimized, is “Test 4” in the test directory of the Adept software package. The interface to the algorithm is as follows:

```cpp
#include <vector>
#include <adept.h>
```
using adept::adouble;
class State {
public:
    // Construct a state with n state variables
    State(int n) { active_x_.resize(n); x_.resize(n); }
    // Minimize the function, returning true if minimization successful, false otherwise
    bool minimize();
    // Get copy of state variables after minimization
    void x(std::vector<double>& x_out) const;
    // For input state variables x, compute the function J(x) and return it
    double calc_function_value(const double* x);
    // For input state variables x, compute function and put its gradient in dJ_dx
    double calc_function_value_and_gradient(const double* x, double* dJ_dx);
    // Return the size of the state vector
    unsigned int nx() const { return active_x_.size(); }
protected:
    // Active version: the algorithm is contained in the definition of this function
    adouble calc_function_value(const adouble* x);
    // DATA
    adept::Stack stack_; // Adept stack object
    std::vector<adouble> active_x_; // Active state variables
};

The algorithm itself is contained in the definition of calc_function_value(const adouble*), which is implemented using adouble variables (following the rules in section 2.2). However, the public interface to the class uses only standard double types, so the use of Adept is hidden to users of the class. Of course, a complicated algorithm may be implemented in terms of multiple classes that do exchange data via adouble objects. We will be using a quasi-Newton minimization algorithm that calls the algorithm many times with trial vectors x, and for each call may request not only the value of the function, but also its gradient with respect to x. Thus the public interface provides calc_function_value(const double*) and calc_function_value_and_gradient, which could be implemented as follows:

double State::calc_function_value(const double* x) {
    for (unsigned int i = 0; i < nx(); ++i) active_x_[i] = x[i];
    stack_.new_recording();
    return value(calc_function_value(&active_x_[0]));
}

double State::calc_function_value_and_gradient(const double* x, double* dJ_dx) {
    for (unsigned int i = 0; i < nx(); ++i) active_x_[i] = x[i];
    stack_.new_recording();
    adouble J = calc_function_value(&active_x_[0]);
    J.set_gradient(1.0);
    stack_.compute_adjoint();
    adept::get_gradients(&active_x_[0], nx(), dJ_dx);
    return value(J);
}

The first function simply copies the double inputs into an adouble vector and runs the version of calc_function_value for adouble arguments. Obviously there is an inefficiency here in that gradients are recorded that are then not used, and this function would be typically 2.5–3 times slower than an implementation of the algorithm that did not store gradients. Section 2.6 describes three ways to overcome this problem. The second function above implements reverse-mode automatic differentiation as described in section 2.3.

The minimize member function could be implemented using GSL as follows:

#include <iostream>
#include <gsl/gsl_multimin.h>

bool State::minimize() {
    // Minimizer settings
    const double initial_step_size = 0.01;
    const double line_search_tolerance = 1.0e-4;
    const double converged_gradient_norm = 1.0e-3;
2.5. Real-world usage: interfacing Adept to a minimization library

```c
// Use the "limited-memory BFGS" quasi-Newton minimizer
const gsl_multimin_fdfminimizer_type* minimizer_type
  = gsl_multimin_fdfminimizer_vector_bfgs2;

// Declare and populate structure containing function pointers
gsl_multimin_function_fdf my_function;
my_function.n = nx();
my_function.f = my_function_value;
my_function.df = my_function_gradient;
my_function.fdf = my_function_value_and_gradient;
my_function.params = reinterpret_cast<void*>(this);

// Set initial state variables using GSL's vector type
gsl_vector *x;
x = gsl_vector_alloc(nx());
for (unsigned int i = 0; i < nx(); ++i) gsl_vector_set(x, i, 1.0);

// Configure the minimizer
gsl_multimin_fdfminimizer* minimizer
  = gsl_multimin_fdfminimizer_alloc(minimizer_type, nx());
gsl_multimin_fdfminimizer_set(minimizer, &my_function, x,
  initial_step_size, line_search_tolerance);

// Begin loop
size_t iter = 0;
int status;
do {
  ++iter;
  // Perform one iteration
  status = gsl_multimin_fdfminimizer_iterate(minimizer);
  // Quit loop if iteration failed
  if (status != GSL_SUCCESS) break;
  // Test for convergence
  status = gsl_multimin_test_gradient(minimizer->gradient,
    converged_gradient_norm);
} while (status == GSL_CONTINUE && iter < 100);

// Free memory
gsl_multimin_fdfminimizer_free(minimizer);
gsl_vector_free(x);

// Return true if successfully minimized function, false otherwise
if (status == GSL_SUCCESS) {
  std::cout << "Minimum found after " << iter << " iterations\n";
  return true;
}
else {
  std::cout << "Minimizer failed after " << iter << " iterations: "
    << gsl_strerror(status) << "\n";
  return false;
}
```

The GSL interface requires three functions to be defined, each of which takes a vector of state variables `x` as input: `my_function_value`, which returns the value of the function; `my_function_gradient`, which returns the gradient of the function with respect to `x`; and `my_function_value_and_gradient`, which returns the value and the gradient of the function. These functions are provided to GSL as function pointers (see above), but since GSL is a C library, we need to use the `extern "C"` specifier in their definition. Thus the function definitions would be:

```c
extern "C"
double my_function_value(const gsl_vector* x, void* params) {
  State* state = reinterpret_cast<State*>(params);
  return state->calc_function_value(x->data);
}
```
2.6 Calling an algorithm with and without automatic differentiation from the same program

When the gsl_multimin_fdfminimizer_iterate function is called, it chooses a search direction and performs several calls of these functions to approximately minimize the function along this search direction. The this pointer (i.e. the pointer to the State object), which was provided to the my_function structure in the definition of the minimize function above, is provided as the second argument to each of the three functions above. Unlike in C, in C++ this pointer needs to be cast back to a pointer to a State type, hence the use of reinterpret_cast. That’s it! A call to minimize should successfully minimize well behaved differentiable multi-dimensional functions. It should be straightforward to adapt the above to work with other minimization libraries.

2.6 Calling an algorithm with and without automatic differentiation from the same program

The calc_function_value(const double*) member function defined in section 2.5 is sub-optimal in that it simply calls the calc_function_value(const adouble*) member function, which not only computes the value of the function, it also records the derivative information of all the operations involved. This information is then ignored. This overhead makes the function typically 2.5–3 times slower than it needs to be, although sometimes (specifically for loops containing no transcendental functions) the difference between an algorithm coded in terms of doubles and the same algorithm coded in terms of adoubles can exceed a factor of 10 (Hogan, 2014).

The impact on the computational speed of the entire minimization process depends on how many requests are made for the function value only as opposed to the gradient of the function, and can be significant. We require a way to avoid the overhead of Adept computing the derivative information for calls to calc_function_value(const double*), without having to maintain two versions of the algorithm, one coded in terms of doubles and the other in terms of adoubles. The three ways to achieve this are now described.

2.6.1 Function templates

The simplest approach is to use a function template for those functions that take active arguments, as demonstrated in the following example:

```c++
#include <adept.h>

class State {
  public:
  ...
  template <typename xdouble>
  xdouble calc_function_value(const xdouble* x);
  ...
};

// Example function definition that must be in a header file included
// by any source file that calls calc_function_value
template <typename xdouble>
inline
xdouble State::calc_function_value(const xdouble* x) {
  xdouble y = 4.0;
}
```
2.6. Calling an algorithm with and without automatic differentiation from the same program

```c
xdouble s = 2.0*x[0] + 3.0*x[1]*x[1];
y *= sin(s);
return y;
```

This takes the example from section 2.2 and replaces \texttt{adouble} by the template type \texttt{xdouble}. Thus, \texttt{calc\_function\_value} can be called with either \texttt{double} or \texttt{adouble} arguments, and the compiler will compile inline the inactive or active version accordingly. Note that the function template need not be a member function of a class.

This technique is good if only a small amount of code needs to be differentiated, but for large models the use of inlining is likely to lead to duplication of compiled code leading to large executables and long compile times. The following two approaches do not have this drawback and are suitable for large codes.

### 2.6.2 Pausable recording

The second method involves compiling the entire code with the \texttt{ADEPT\_RECORDING\_PAUSABLE} preprocessor variable defined, which can be done by adding an argument \texttt{-DADEPT\_RECORDING\_PAUSABLE} to the compiler command line. This modifies the behaviour of mathematical operations performed on \texttt{adouble} variables: instead of performing the operation and then storing the derivative information, it performs the operation and then only stores the derivative information if the Adept stack is not in the “paused” state. We then use the following member function definition instead of the one in section 2.5:

```cpp
double State::calc_function_value(const double* x) {
    stack_.pause_recording();
    for (unsigned int i = 0; i < nx(); ++i) active_x_[i] = x[i];
    double J = value(calc_function_value(&active_x_[0]));
    stack_.continue_recording();
    return J;
}
```

By pausing the recording for all operations on \texttt{adouble} objects, most of the overhead of storing derivative information is removed. The extra run-time check to see whether the stack is in the paused state, which is carried out by mathematical operations involving \texttt{adouble} variables, generally adds a small overhead. However, in algorithms where most of the number crunching occurs in loops containing no transcendental functions, even if the stack is in the paused state, the presence of the check can prevent the compiler from aggressively optimizing the loop. In that instance the third method may be preferable.

### 2.6.3 Multiple object files per source file

The third method involves compiling each source file containing functions with \texttt{adouble} arguments twice. The first time, the code is compiled normally to produce an object file containing compiled functions including automatic differentiation. The second time, the code is compiled with the \texttt{-DADEPT\_NO\_AUTOMATIC\_DIFFERENTIATION} flag on the compiler command line. This instructs the \texttt{adept.h} header file to turn off automatic differentiation by defining the \texttt{adouble} type to be an alias of the \texttt{double} type. This way, a second set of object files are created containing overloaded versions of the same functions as the first set but this time without automatic differentiation. These object files can be compiled together to form one executable. In the example presented in section 2.5, the \texttt{calc\_function\_value} function would be one that would be compiled twice in this way, once to provide the \texttt{calc\_function\_value(const adouble*)} version and the other to provide the \texttt{calc\_function\_value(const double*)} version. Note that any functions that do not include \texttt{adouble} arguments must be compiled only once, because otherwise the linker will complain about multiple versions of the same function.

The following shows a Makefile from a hypothetical project that compiles two source files (\texttt{algorithm1.cpp} and \texttt{algorithm2.cpp}) twice and a third (\texttt{main.cpp}) once:

```makefile
# Specify compiler and flags
CXX = g++
CXXFLAGS = -Wall -O3 -g
```
There is a further modification required with this approach, which arises because if a header file declares both the `double` and `adouble` versions of a function, then when compiled with `-DADEPT_NO_AUTOMATIC_DIFFERENTIATION` it appears to the compiler that the same function is declared twice, leading to a compile-time error. This can be overcome by using the preprocessor to hide the `adouble` version if the code is compiled with this flag, as follows (using the example from section 2.5):

```c
#include <adept.h>
class State {
  public:
    ...
    double calc_function_value(const double* x);
  protected:
    ifndef ADEPT_NO_AUTOMATIC_DIFFERENTIATION
      double calc_function_value(const adouble* x);
    endif
    ...
};
```

A final nuance is that if the code contains an `adouble` object `x`, then `x.value()` will work fine in the compilation when `x` is indeed of type `adouble`, but in the compilation when it is set to a simple `double` variable, the `value()` member function will not be found. Hence it is better to use `adept::value(x)`, which returns a `double` regardless of the type of `x`, and works regardless of whether the code was compiled with or without the `-DADEPT_NO_AUTOMATIC_DIFFERENTIATION` flag.

### 2.7 Interfacing with software containing hand-coded Jacobians

Often a complicated algorithm will include multiple components. Components of the code written in C or C++ for which the source is available are straightforward to convert to using Adept, following the rules in section 2.2. For components written in Fortran, this is not possible, but if such components have their own hand-coded Jacobian then it is possible to interface Adept to them. More generally, in certain situations automatic differentiation is much slower than hand-coding (see the Lax-Wendroff example in Hogan, 2014) and we may wish to hand-code certain critical parts. In general the Jacobian matrix is quite expensive to compute, so this interfacing strategy makes most sense if the component of the algorithm has a small number of inputs or a small number of outputs. A full working version of the following example is given as “Test 3” in the `test` directory of the Adept package.

Consider the example of a radiative transfer model for simulating satellite microwave radiances at two wavelengths, \( I \) and \( J \), which takes as input the surface temperature \( T_s \) and the vertical profile of atmospheric temperature \( T \) from a numerical weather forecast model. Such a model would be used in a data assimilation system to assimilate the temperature information from the satellite observations into the weather forecast model. In addition
to returning the radiances, the model returns the gradient $\partial I / \partial T_s$ and the gradients $\partial I / \partial T_i$ for all height layers $i$ between 1 and $n$, and likewise for radiance $J$. The interface to the radiative transfer model is the following:

```c
void simulate_radiances(int n, // Size of temperature array
                      double surface_temperature, // Input variables:
                      const double* temperature,  // Output variables:
                      double radiance[2],
                      // Output Jacobians:
                      double dradiance_dsurface_temperature[2],
                      double* dradiance_dtemperature);
```

The calling function needs to allocate $2 \times n$ elements for the temperature Jacobian $dradiance_dtemperature$ to be stored, and the stored Jacobian will be oriented such that the radiance index varies fastest.

**Adept** needs to be told how to relate the radiance perturbations $\delta I$ and $\delta J$ to perturbations in the input variables, $\delta T_i$ and $\delta T_i$ (for all layers $i$). Mathematically, we wish the following relationship to be stored within the **Adept** stack:

$$
\delta I = \frac{\partial I}{\partial T_s} \delta T_s + \sum_{i=1}^{n} \frac{\partial I}{\partial T_i} \delta T_i.
$$

(2.1)

This is achieved with the following wrapper function, which has `adouble` inputs and outputs and therefore can be called from within other parts of the algorithm that are coded in terms of `adouble` objects:

```c
void simulate_radiances_wrapper(int n,
                                const adouble& surface_temperature,
                                const adouble* temperature,
                                adouble radiance[2]) {
    // Create inactive (double) versions of the active (adouble) inputs
    double st = value(surface_temperature);
    std::vector<double> t(n);
    for (int i = 0; i < n; ++i) t[i] = value(temperature[i]);

    // Declare variables to hold the inactive outputs and their Jacobians
    double r[2];
    double dr_dst[2];
    std::vector<double> dr_dt(2*n);

    // Call the non-Adept function
    simulate_radiances(n, st, &t[0], &r[0], dr_dst, &dr_dt[0]);

    // Copy the results into the active variables, but use set_value in order
    // not to write any equivalent differential statement to the Adept stack
    radiance[0].set_value(r[0]);
    radiance[1].set_value(r[1]);

    // Loop over the two radiances and add the differential statements to the Adept stack
    for (int i = 0; i < 2; ++i) {
        // Add the first term on the right-hand-side of Equation 1 in the text
        radiance[i].add_derivative_dependence(surface_temperature, dr_dst[i]);
        // Now append the second term on the right-hand-side of Equation 1. The third argument
        // "n" of the following function says that there are n terms to be summed, and the fourth
        // argument "2" says to take only every second element of the Jacobian dr_dt, since the
        // derivatives with respect to the two radiances have been interlaced. If the fourth
        // argument is omitted then relevant Jacobian elements will be assumed to be contiguous
        // in memory.
        radiance[i].append_derivative_dependence(temperature, &dr_dt[i], n, 2);
    }
}
```

In this example, the form of `add_derivative_dependence` for one variable on the right-hand-side of the derivative expression has been used, and the form of `append_derivative_dependence` for an array of variables on the right-hand-side has been used. As described in section 2.9, both functions have forms that take single variables
and arrays as arguments. Note also that the use of std::vector<double> rather than new double[n] ensures that if simulate_radiances throws an exception, the memory allocated to hold dr_dt will be freed correctly.

2.8 Member functions of the Stack class

This section describes the user-oriented member functions of the Stack class. Some functions have arguments with default values; if these arguments are omitted then the default values will be used. Some of these functions throw Adept exceptions, defined in section 5.5.

Stack(bool activate_immediately = true) The constructor for the Stack class. Normally Stack objects are constructed with no arguments, which means that the object will attempt to make itself the currently active stack by placing a pointer to itself into a global variable. If another Stack object is currently active, then the present one will be fully constructed, left in the unactivated state, and an stack_already_active exception will be thrown. If a Stack object is constructed with an argument “false”, it will be started in an unactivated state, and a subsequent call to its member function activate will be needed to use it.

void new_recording() Clears all the information on the stack in order that a new recording can be started. Specifically this function clears all the differential statements, the list of independent and dependent variables (used in computing Jacobian matrices) and the list of gradients used by the compute_tangent_linear and compute_adjoint functions. Note that this function leaves the memory allocated to reduce the overhead of reallocation in the new recordings.

bool pause_recording() Stops recording differential information every time an adouble statement is executed. This is useful if within a single program an algorithm needs to be run both with and without automatic differentiation. This option is only effective within compilation units compiled with ADEPT_RECORDING_PAUSABLE defined; if it is, the function returns true, otherwise it returns false. Further information on using this and the following function are provided in section 2.6.2.

bool continue_recording() Instruct a stack that may have previously been put in a paused state to now continue recording differential information as normal. This option is only effective within compilation units compiled with ADEPT_RECORDING_PAUSABLE defined; if it is, the function returns true, otherwise it returns false.

bool is_recording() Returns false if recording has been paused with pause_recording() and the code has been compiled with ADEPT_RECORDING_PAUSABLE defined. Otherwise returns true.

void compute_tangent_linear() Perform a tangent-linear calculation (forward-mode differentiation) using the stored differential statements. Before calling this function you need call the adouble::set_gradient or set_gradients function (see section 2.9) on the independent variables to set the initial gradients, otherwise the function will throw a gradients_not_initialized exception. This function is synonymous with forward().

void compute_adjoint() Perform an adjoint calculation (reverse-mode differentiation) using the stored differential statements. Before calling this function you need call the adouble::set_gradient or set_gradients function on the dependent variables to set the initial gradients, otherwise the function will throw a gradients_not_initialized exception. This function is synonymous with reverse().

void independent(const adouble& x) Before computing Jacobian matrices, you need to identify the independent and dependent variables, which correspond to the columns and rows of the Jacobian, respectively. This function adds x to the list of independent variables. If it is the nth variable identified in this way, the nth column of the Jacobian will correspond to derivatives with respect to x.

void dependent(const adouble& y) Add y to the list of dependent variables. If it is the mth variable identified in this way, the mth row of the Jacobian will correspond to derivatives of y with respect to each of the independent variables.
void independent(const adouble* x_ptr, size_t n) Add n independent variables to the list, which must be stored consecutively in memory starting at the memory pointed to by x_ptr.

void dependent(const adouble* y_ptr, size_t n) Add n dependent variables to the list, which must be stored consecutively in memory starting at the memory pointed to by y_ptr.

void jacobian(double* jacobian_out) Compute the Jacobian matrix, i.e., the gradient of the m dependent variables (identified with the dependent(...) function) with respect to the n independent variables (identified with independent(...). The result is returned in the memory pointed to by jacobian_out, which must have been allocated to hold m × n values. The result is stored in column-major order, i.e., the m dimension of the matrix varies fastest. If no dependents or independents have been identified, then the function will throw a dependents_or_independents_not_identified exception. In practice, this function calls jacobian_forward if n ≤ m and jacobian_reverse if n > m.

void jacobian(Matrix jac) Compute Jacobian matrix and store in a correctly sized Adept Matrix object jac, which may be a subset of an larger matrix. See chapter 3 for a full description of Adept array objects.

Matrix jacobian() As above but the Jacobian matrix is returned from the function.

void jacobian_forward(double* jacobian_out) Compute the Jacobian matrix by executing n forward passes through the stored list of differential statements; this is typically faster than jacobian_reverse for n ≤ m.

void jacobian_forward(Matrix jac) As above but store in a correctly sized Adept Matrix object jac.

Matrix jacobian_forward() As above but the Jacobian matrix is returned from the function.

void jacobian_reverse(double* jacobian_out) Compute the Jacobian matrix by executing m reverse passes through the stored list of differential statements; this is typically faster than jacobian_forward for n > m.

void jacobian_reverse(Matrix jac) As above but store in a correctly sized Adept Matrix object jac.

Matrix jacobian_reverse() As above but the Jacobian matrix is returned from the function.

void clear_gradients() Clear the gradients set with the set_gradient member function of the adouble class. This enables multiple adjoint and/or tangent-linear calculations to be performed with the same recording.

void clear_independents() Clear the list of independent variables, enabling a new Jacobian matrix to be computed from the same recording but for a different set of independent variables.

void clear_dependents() Clear the list of dependent variables, enabling a new Jacobian matrix to be computed from the same recording but for a different set of dependent variables.

size_t n_independents() Return the number of independent variables that have been identified.

size_t n_dependents() Return the number of dependent variables that have been identified.

size_t n_statements() Return the number of differential statements in the recording.

size_t n_operations() Return the total number of operations in the recording, i.e the total number of terms on the right-hand-side of all the differential statements.

size_t max_gradients() Return the number of working gradients that need to be stored in order to perform a forward or reverse pass.

size_t memory() Return the number of bytes currently used to store the differential statements and the working gradients. Note that this does not include memory allocated but not currently used.
size_t n_gradients_registered() Each time an adouble object is created, it is allocated a unique index that is used to identify its gradient in the recorded differential statements. When the object is destructed, its index is freed for reuse. This function returns the number of gradients currently registered, equal to the number of adouble objects currently created.

void print_status(std::ostream& os = std::cout) Print the current status of the Stack object, such as number of statements and operations stored and allocated, to the stream specified by os, or standard output if this function is called with no arguments. Sending the Stack object to the stream using the “<<” operator results in the same behaviour.

void print_statements(std::ostream& os = std::cout) Print the list of differential statements to the specified stream (or standard output if not specified). Each line corresponds to a separate statement, for example “d[3] = 1.2*d[1] + 3.4*d[2]”.

bool print_gradients(std::ostream& os = std::cout) Print the vector of gradients to the specified stream (or standard output if not specified). This function returns false if no set_gradient function has been called to set the first gradient and initialize the vector, and true otherwise. To diagnose what compute_tangent_linear and compute_adjoint are doing, it can be useful to call print_gradients immediately before and after.

void activate() Activate the Stack object by copying its this pointer to a global variable that will be accessed by subsequent operations involving adouble objects. If another Stack is already active, a stack already active exception will be thrown. To check whether this is the case before calling activate(), check that the active_stack() function (described below) returns 0.

void deactivate() Deactivate the Stack object by checking whether the global variable holding the pointer to the currently active Stack is equal to this, and if it is, setting it to 0.

bool is_active() Returns true if the Stack object is the currently active one, false otherwise.

void start() This function was present in version 0.9 to activate a Stack object, since in that version they were not constructed in an activated state. This function has now been deprecated and will always throw a feature not available exception.

int max_jacobian_threads() Return the maximum number of OpenMP threads available for Jacobian calculations. The number will be 1 if either the library was or the current source code is compiled without OpenMP support (i.e. without the -fopenmp compiler and linker flag). (Introduced in Adept version 1.1.)

int set_max_jacobian_threads(int n) Set the maximum number of threads to be used in Jacobian calculations to n, if possible. A value of 1 indicates that OpenMP will not be used, while a value of 0 indicates that the maximum available will be used. Returns the maximum that will be used, which may be fewer than requested, e.g. 1 if the Adept library was compiled without OpenMP support. (Introduced in Adept version 1.1.)

void preallocate_statements(int n) If you know in advance roughly how many differential statements will be stored by an algorithm then you may be able to speed-up the first use of the stack by preallocating the memory needed to store them. More memory will still be allocated if needed, but this should reduce the number of allocations and copies.

void preallocate_operations(int n) Likewise, if you know in advance roughly how many operations will be stored then you can speed-up the first use of the stack with this member function.

The following non-member functions are provided in the adept namespace:

adept::Stack* active_stack() Returns a pointer to the currently active Stack object, or 0 if there is none.

bool is_thread_unsafe() Returns true if your code has been compiled with ADEPT_STACK_THREAD_UNSAFE, false otherwise.
2.9 Member functions of the adouble object

This section describes the user-oriented member functions of the adouble class. Some functions have arguments with default values; if these arguments are omitted then the default values will be used. Some of these functions throw Adept exceptions, defined in section 5.5.

**double value()** Return the underlying double value.

**void set_value(double x)** Set the value of the adouble object to x, without storing the equivalent differential statement in the currently active stack.

**void set_gradient(const double& gradient)** Set gradient corresponding to this adouble variable. The first call of this function (for any adouble variable) after a new recording is made also initializes the vector of working gradients. This function should be called for one or more adouble objects after a recording has been made but before a call to Stack::compute_tangent_linear() or Stack::compute_adjoint().

**void get_gradient(const double& gradient)** Set gradient to the value of the gradient corresponding to this adouble object. This function is used to extract the result after a call to Stack::compute_tangent_linear() or Stack::compute_adjoint(). If the set_gradient function was not called since the last recording was made, this function will throw a gradients_not_initialized exception. The function can also throw a gradient_out_of_range exception if new adouble objects were created since the first set_gradient function was called.

**void add_derivative_dependence(const adouble& r, const double& g)** Add a differential statement to the currently active stack of the form \( \delta l = g \times \delta r \), where \( l \) is the adouble object from which this function is called. This function is needed to interface to software containing hand-coded Jacobians, as described in section 2.7; in this case \( g \) is the gradient \( \partial l / \partial r \) obtained from such software.

**void append_derivative_dependence(const adouble& r, const double& g)** Assuming that the same adouble object has just had its add_derivative_dependence member function called, this function appends \( + g \times \delta r \) to the most recent differential statement on the stack. If the calling adouble object is different, then a wrong_gradient exception will be thrown. Note that multiple append_derivative_dependence calls can be made in succession.

**void add_derivative_dependence(const adouble* r, const double* g, size_t n = 1, size_t m_stride = 1)**
Add a differential statement to the currently active stack of the form \( \delta l = \sum_{i=0}^{n-1} m[i] \times \delta r[i] \), where \( l \) is the adouble object from which this function is called. If the gstride argument is provided, then the index to the g array will be \( i \times \text{g.stride} \) rather than \( i \). This is useful if the Jacobian provided is oriented such that the relevant gradients for \( l \) are not spaced consecutively.

**void append_derivative_dependence(const adouble* rhs, const double* g, size_t n = 1, size_t g_stride = 1)**
Assuming that the same adouble object has just called the add_derivative_dependence function, this function appends \( + \sum_{i=0}^{n-1} m[i] \times \delta r[i] \) to the most recent differential statement on the stack. If the calling adouble object is different, then a wrong_gradient exception will be thrown. The g_stride argument behaves the same way as in the previous function described.

The following non-member functions are provided in the adept namespace:

**double value(const adouble& x)** Returns the underlying value of \( x \) as a double. This is useful to enable \( x \) to be used in fprintf function calls. It is generally better to use adept::value(x) rather than \( x.value() \), because the former also works if you compile the code with the ADEPT_NO_AUTOMATIC_DIFFERENTIATION flag set, as discussed in section 2.6.3.

**void set_values(adouble* x, size_t n, const double* x.val)** Set the value of the n adouble objects starting at \( x \) to the values in \( x.val \), without storing the equivalent differential statement in the currently active stack.
void set_gradients(adouble* x, size_t n, const double* gradients) Set the gradients corresponding to the n adouble objects starting at x to the n doubles starting at gradients. This has the same effect as calling the set.gradient member function of each adouble object in turn, but is more concise.

void get_gradients(const adouble* y, size_t n, double* gradients) Copy the gradient of the n adouble objects starting at y into the n doubles starting at gradients. This has the same effect as calling the get.gradient member function of each adouble object in turn, but is more concise. This function can throw a gradient.out.of.range exception if new adouble objects were created since the first set.gradients function or set.gradient member function was called.
Chapter 3

Using *Adept*’s array functionality

3.1 Introduction

The design of *Adept*’s array capability and many of the functions is inspired to a significant extent by the built-in array support in Fortran 90 (and later), and a lesser extent by Matlab, although implemented in the “C++ way”, e.g. default row-major order with all array indices starting from zero. Future additions to the array capability in *Adept* will attempt to reproduce built-in Fortran array functions if available*. This design makes *Adept* a good choice if you have Fortran code that you wish to convert to C++. *Adept* provides the following array functionality:

**Multi-dimensional arrays.** Standard dynamically sized arrays can have an arbitrary number of dimensions (although indexing and slicing is supported only up to 7), and may refer to non-contiguous areas of memory. See section 3.2.

**Mathematical operators and functions.** *Adept* supports array expressions containing the standard mathematical operators $+$, $-$, $*$ and $/$, as well as their assignment versions $+=,-=,*=$ and $/=$. When applied to arrays, they work “element-wise”, applying the same operation to every element of the arrays. *Adept* also supports array operations on all the mathematical functions listed in section 2.1. The following operators and functions return boolean array expressions: $==$, $!=$, $>$, $<$, $>$= and $<=$, isfinite, isninf and isnan. See section 3.3.

**Array slicing.** There are many ways to produce an array that references a subset of another array, and therefore can be used as an lvalue in a statement. Arrays can be indexed with scalar integers, a contiguous range of integers, a strided range of integers or an arbitrary list of integers. This is facilitated with “...” (a double underscore) and “end”, such that $A(\ldots,end-1)$ returns a vector pointing to the penultimate column of matrix $A$. The member function subset produces an array pointing to a contiguous subset of the original array, while diag_vector and diag_matrix produce arrays pointing to the diagonal of the original array. $T$ produces an array pointing to the transpose of the original array. See section 3.4.

**Passing arrays to and from functions.** *Adept* uses a reference-counting approach to implement the storage of array data, enabling multiple array objects to point to the same data, or parts of it in the case of array slices. This makes it straightforward to pass arrays to and from functions without having to perform a deep copy. See section 3.5.

**Array reduction operations.** The functions sum, mean, product, minval, maxval and norm2 perform reduction operations that return an array of lower rank to the expression they are applied to. The functions all and any do the same but for boolean expressions. count returns the number of true elements in a boolean expression. See section 3.6.

---

*This decision may puzzle some readers, since Fortran is a dirty word to many C++ users due to the limitations of the FORTRAN 77 language. Many of these limitations were overcome in Fortran 90, whose array functionality in particular is rather well designed. Indeed, the pioneering “Blitz++” C++ array library (Veldhuizen, 1995) also reproduces many Fortran array functions. All references to Fortran in this document imply the 1990 (or later) standard.*
3.2 The Array class

Array expansion operations. The functions `outer_product` and `spread` return an expression of a higher rank than the expression they are applied to. See section 3.7.

Conditional operations. Two convenient ways are provided to perform an operation on an array depending on the result of a boolean expression: `where` and `find`. The statement `A.where(B>0)=C` assigns elements of `C` to elements of `A` whenever the corresponding element of `B` is greater than zero. For vectors only, the same result could be obtained with `A(find(B>0))=C(find(B>0))`. See section 3.8.

Fixed-size arrays. Adept provides a fixed-size array class with dimensions (up to seven) that are known at compile time. The functionality is very similar to standard dynamic arrays.

Special square matrices. Adept uses specific classes for symmetric, triangular and band-diagonal matrices, the latter of which use compressed storage and include diagonal and tridiagonal matrices. Certain operations such as matrix multiplication and solving linear equations are optimized especially for these objects. See section 3.10.

Matrix multiplication. Matrix multiplication can be applied to one- and two-dimensional arrays using the `matmul` function, of for extra syntactic sugar, the “*” pseudo-operator. Adept uses whatever BLAS (Basic Linear Algebra Subroutines) support is available on your system, including optimized versions for symmetric and band-diagonal matrices. See section 3.11.

Linear algebra. Adept uses the LAPACK library to invert matrices and solve linear systems of equations. See section 3.12.

Array bounds and alias checking. Adept checks at compile time that terms in an array expression accord in rank, and at run time that they accord in the size of each dimension. Run-time alias checking is performed to determine if any objects on the right-hand-side of a statement overlap in memory with the left-hand-side of the statement, making a temporary copy of the right-hand-side if they do. This can be overridden with the `noalias` function. See section 3.13.

Interoperability with Fortran arrays. The Fortran 2018 standard enables Fortran’s assumed-shape arrays to be passed to and from C/C++. Section 3.17 describes how they can be treated as Adept arrays within C++.

3.2 The Array class

The bread and butter of array operations is provided by the `Array` class template (in the `adept` namespace along with all other public types and classes), which has the following declaration:

```cpp
namespace adept {
  template <int Rank, typename Type = Real, bool IsActive = false>
  class Array;
}
```

The first template argument provides the number of dimensions of the array and may be 1 or greater, although indexing and slicing is only supported up to 7 dimensions. The second argument is the numerical type being stored and can be any simple integer or real number, including `bool`. The default type is `adept::Real`, which is the default floating-point type the Adept library has been compiled to use for computing derivatives, and is usually `double`. The final argument states whether the array is “active”, i.e. whether it participates in the differentiation of an algorithm.

A number of typedefs are provided for the most common types of array: `Vector`, `Matrix`, `Array3D` and so on up to `Array7D` provide inactive arrays of type `Real` and rank 1–7. The corresponding active types are `aVector`, `aMatrix`, `aArray3D` etc. Arrays of other numeric types have the pattern `boolVector`, `intVector`, `floatVector`, `afloatVector`, and similarly for matrices and higher dimensional arrays. If you wanted shortcuts for other types you could do the following:

```cpp
typedef adept::Array<4, unsigned int> uintArray4D;
typedef adept::Array<2, long double, true> alongdoubleMatrix; // Active
```
An Array with uninitialized elements can be constructed in numerous ways:

```cpp
using namespace adept;
Vector v; // Initialize an empty vector
Array3D A(3, 4, 5); // Initialize a 3x4x5 array (up to 7 arguments possible)
Matrix M(dimensions(3, 4)); // The "dimensions" function takes up to 7 arguments
Matrix N(M.dimensions()); // Make N the same size as M
```

In the remaining code examples it will be assumed that `using namespace adept` has already been called. When new memory is needed, the Array object creates a Storage object that contains the memory needed, and stores pointers to both the Storage object and the start of the data. By default the data are accessed in C-style row-major order (i.e. the final index corresponds to the array dimension that varies most rapidly in memory). However, this is flexible since in addition to storing the length of each of its \( n \) dimensions, a rank-\( n \) Array also stores \( n \) “offsets” that define the separation of elements in memory in each dimension. Thus, a 3-by-4 matrix with row-major storage would store offsets of (4,1). The same size matrix would use column-major storage simply by storing offsets of (1,3). To make new arrays use column-major storage, call the following function:

```cpp
set_array_row_major_order(false);
```

Note that this does not change the storage of any existing objects. Note also that when array expressions are evaluated, the data are requested in row-major order, so the use of column-major arrays will incur a performance penalty.

An Array may also be constructed such that it immediately contains data:

```cpp
Vector v = M(__, 0); // Link to a existing array, in this case the first column of M
Vector v(M(__, 0)); // Has exactly the same effect as the previous example
Matrix N = log(M); // Initialize with the size and values of a mathematical expression
```

It can be seen from the constructors involving Vectors that an Array can be configured to “link” to part of an existing Array, and modifications to the numbers in one will be seen by the other. This is a very useful feature as it allows slices of an array to be passed to functions and modified; see section 3.4. Note that the array or sub-array being linked to must be of the same rank, type and activeness as the linking array. Internally, linking is achieved by both the arrays pointing to the same Storage object, which itself contains a reference count of the number of arrays pointing to it. When an Array is destructed the reference count is reduced by one and only if it falls to zero will the data get deallocated. This ensures that if the Array being linked to goes out of scope, the linking Array will “steal” the data.

You can also make an Array point to data not held in a Storage object, for example in a function whose interface is only in terms of intrinsic C types:

```cpp
double my_norm2(int n, double* ptr) {
    Vector x(ptr, dimensions(n)); // Create a Vector pointing to existing data
    return norm2(x); // Use Adept’s L2-norm function
}
```

The Vector in this example can be used in the same way as any other array, but relies on the existing data not being deallocated for the lifetime of the Vector.

After it has been constructed, an Array can be resized, relinked or cleared completely as follows:

```cpp
M.resize(5, 2); // Works up to 7 dimensions
M.resize(dimensions(5, 2)); // As above
N.resize(M.dimensions()); // Resize N to be the same size as M
v.link(M(end-1, __)); // Size of v set to that of the argument and link to data
v >>= M(end-1, __); // Convenient syntax for linking, similar to Fortran’s "->"
M.clear(); // Returns array to original empty state
```

The member functions `resize` and `clear` unlink from any existing data, which involves deallocation if no other array is pointing to the same data. If the `link` function, or the alternative “\( \triangleright \triangleright \triangleright \)” syntax, is applied with a non-empty array on the left-hand-side then the existing data will be quietly cleared before linking to the new data. Note that if you assign one array to another (e.g. \( N=M \)), then they must be of the same size; if they are not then you should clear the left-hand-side first. By default, resized arrays are row-major, unless `set_array_row_major_order(false)` has been called. To explicitly specify the ordering, you may use the `resize_row_major` or `resize_column_major` member functions in place of `resize`. 
The `Array` class implements a number of member functions for inquiring about its properties:

- **size()** Returns the total number of elements, i.e. the product of the lengths of each of the dimensions.
- **dimension(i)** Returns the length of dimension i.
- **offset(i)** Returns the separation in memory of elements along dimension i.
- **gradient_index()** For active arrays, returns the gradient index of the first element of the array, which is always positive; for inactive arrays it returns a negative number.
- **empty()** Returns `true` if the array is in the empty state, or `false` otherwise.
- **dimensions()** Returns a object listing the extent of each dimension in the `Array`, useful for resizing other arrays. The object is actually of type `ExpressionSize<int Rank>` (where `Rank` is the rank of the array), a thin wrapper for a simple `int[Rank]` C-array, although it is rare to need to use it explicitly.
- **offset()** Returns an object (also of type `ExpressionSize<int Rank>`) describing how array indices are translated into memory offsets.

An `Array` may be filled using the `<<` operator for the first element followed by either the `<<` or `,` operators for subsequent elements:

```cpp
Vector v(4);
v << 1 << 2 << 3 << 4; // Fill the four elements of v
v << 1, 2, 3, 4; // Same behaviour but easier on the eye
v << 1, 2, 3, 4, 5; // Error: v has been overfilled
Matrix M(2,4);
M << 1, 2, 3, 4; // Filling of multi-dimensional arrays
5, 6, 7, 8; // automatically moves on to next dimension
M << 1, 2, 3, 4,
v; // v treated as a row vector here
```

For multidimensional arrays, elements are filled such that the final dimension ticks over fastest (regardless of whether the array uses row-major storage internally), and new rows are started when a row is complete. Moreover, other arrays can be part of the list of elements, provided that they fit in. In this context, a rank-1 array is treated as a row vector. An `index_out_of_bounds` exception is thrown if an array is overfilled, while an `empty_array` exception is thrown if an attempt is made to fill an empty array.

If you compile your code with C++11 features enabled then you can use the “initializer list” feature to fill arrays using the C-like curly bracket syntax:

```cpp
Vector v; // Construct an empty vector
v = {1, 2, 3}; // Resize to length 3 and fill
Vector w = {1, 2, 3}; // Construct a vector of length 3 and fill
w = {4.4, 5.5}; // Underfill leads to remaining elements set to zero (as in C)
w = {6, 7, 8, 9}; // Overfill leads to size_mismatch exception being thrown
Matrix M = {{1, 2, 3}, // Multi-dimensional arrays use nested curly brackets;
            {4, 5}}; // ...underfill again leads to remaining elements set to zero
```

Another convenient property of this syntax is that temporary arrays with explicit values can be used in expressions:

```cpp
v = w * Vector{3.0, 4.2, 5.1};
```

When interfacing with other libraries, direct access to the data is often required. The `Array` class provides the following member functions:

- **data()** Returns a pointer to the first element in the array, i.e. the element found by indexing all the dimensions of the array with zero. It is up to the caller to understand the layout of the data in memory and not to stray outside. Remember that an array may be strided and the stride may even be negative so that the data returned from increasing indices are actually from earlier memory addresses. Note that a double-precision active
array is not stored as an array of double objects, but as an array of double data and a single gradient index for the first element. Thus the pointer returned by data() will point to the underlying inactive data. In contexts where the Array object is const, a const pointer will be returned. Note that in a multi-dimensional array, successive array dimensions are not guaranteed to be contiguous in memory since it is sometimes advantageous for vectorization for Adept to pad the rows to an alignment boundary. You can use the output of the offset() member function to determine the spacing of the elements in each dimension.

const_data() It is sometimes convenient to specify explicitly that read-only access is required, in which case you can use const_data() to return a const pointer to the first element in the array.

### 3.3 Operators and mathematical functions

The operators and mathematical functions listed in section 2.1 have been overloaded so that they work exactly as you would expect. Consider this example:

```cpp
floatVector a(5);    // Inactive single-precision vector
aVector b(5), c(5); // Active vectors
aReal d;            // An active scalar
// ... other code manipulating a-d ...
b = 2.0;            // Set all elements of b to a scalar value
c += 5.0*a + sin(b)/d; // Add the right-hand-side to c
```

The penultimate illustrates that all elements of an Array can be set to the same value, although note that this will only work if the array is not in the empty state. The final line illustrates how terms with different rank, type and activeness can participate in the same expression. Scalars and arrays can participate in the same expression on the right-hand-side of a statement provided that the arrays have the same size as the array on the left-hand-side. Objects of different type (in this case single and double precision) can be combined in a mathematical operation, and the type of that operation will be the larger (higher precision) of the two types. If active and inactive objects participate in an expression then the left-hand-side must also be active. Expression templates ensure that no temporary arrays need to be created to store the output of intermediate parts of the expression. The functions max and min behave just like binary operators (such as + and *) in this regard, as shown by the following:

```cpp
c = max(a,b);        // Element-wise comparison of a and b
c = min(a,3.0);      // Return minimum of each element of a and 3
```

The examples so far have floating-point results, but some operators (e.g. “==”) and some functions (e.g. isninf) take floating-point arguments and return a boolean. The Adept versions take floating-point array expressions as arguments and return bool expressions of the same rank and size. Finally, the Adept versions of the operators !, || and && take a bool expression as arguments and return a bool expression of the same size and rank.

### 3.4 Array slicing

This section concerns the many ways that sub-parts of an Array can be extracted to produce an object that can be used as an lvalue; that is, if the object is modified then it will modify part of the original Array. It should be stressed that none of these methods results in any rearrangement of data in memory, so they should be efficient.

The first way this can be done is via the function-call and member-access operators (i.e. `operator()` and `operator[]`, respectively) of the Array. In the case of the function-call operator, the same number of arguments as the rank of the array must be provided, where each argument states how its corresponding dimension should be treated. The nature of the resulting object depends on the type of all of the arguments in a way that is similar to how Fortran arrays behave, although note that array indices always start at 0. The four different behaviours are as follows:

**Extract single value.** If every argument is an integer scalar or scalar expression, then a reference to a single element of the array will be extracted. If an argument is an integer expression containing end, then end will be interpreted to be the index to the final element of that dimension (a feature borrowed from Matlab). If
the array is active then the returned object will be of a special “active reference” type that can be used as an lvalue and ensures that any expressions making use of this element can be differentiated. Now for some examples:

```
AMatrix A(4,3);
aReal  x = A(1,1);  // Copy element at second row and second column into x
A(end-1,1) *= 2.0;  // Double the element in the penultimate column and 2nd row of A
A(3) = 4.0;  // Error: number of indices does not match number of dimensions
```

**Extract regular subarray.** If every argument is either (i) an integer scalar or scalar expression, or (ii) a regular range of indices, and there is at least one of (ii), then an `Array` object will be returned of the same type and activeness as the original. However, for each argument of type (i), the rank of the returned array will be one less than that of the original. There are three ways to express a regular range of indices: “_” represents all indices of a particular dimension, `range(a,b)` represents a contiguous range of indices between `a` and `b` (equivalent to `a:b` in Fortran and Matlab), and `stride(a,b,c)` represents a regular range of indices between `a` and `b` with spacing `c` (equivalent to `a:b:c` in Fortran and `a:c:b` in Matlab). Note that `a`, `b`, and `c` may be scalar expressions containing `end`, but `c` must not be zero although it can be negative to indicate a reversed ordering. The rank of the returned array is known at compile time; thus if range arguments are found at run-time to contain only one element (e.g. `range(1,1)`) then the dimension being referred to will not be removed in the returned array but will remain as a singleton dimension. This behaviour is the same as indexing an array dimension with `1:1` in Fortran. Now for some examples:

```
v(range(1,end-1))  // Subset of vector v that excludes 1st & last points
A(0,stride(end,0,-1))  // First row of A as a vector treated in reverse order
A(range(0,0),stride(0,0,1))  // A 1-by-1 matrix containing the first element of A
```

**Extract irregular subarray.** If an array is indexed as in either of the two methods above, except that one or more dimensions is instead indexed using a rank-1 `Array` of integers, then the result is a special “indexed-array” type that stores how each dimension is indexed. If it then participates either on the left- or right-hand-side of a mathematical expression then when an element is requested, the indices will be queried to map the request to obtain the correct element from the original array. This is much less efficient than using regular ranges of indices as above. It also means that if an indexed array is passed to a function expecting an object of type `Array`, then it will first be converted to an `Array` and any modifications performed within the function will not be passed back to the original array. For example:

```
intVector  index(3);
index << 2, 3, 5;
Array A(4,4);
A(0,index) = 2.0;  // Set irregularly spaced elements of the first row of A
```

**Slice leading dimension.** In C, an element is extracted from a two-dimensional array using `A[i][j]`, and `A[i]` returns a pointer to a single row of `A`, where `i` and `j` are integers. To enable similar functionality, if `A` is an `Adept` matrix then `A[i]` indexes the leading dimension by integer `i` returning an array of rank one less than the original. This is equivalent to `A(i,_,_)`. Furthermore, `A[i][j]` will return an individual element as in C, but it should be stressed that `A(i,j)` is more efficient since it does not involve the creation of intermediate arrays.

There are a few other ways to produce lvalues that consist of a subset or a reordering of an array. They are implemented as member functions of the `Array` class, in order to distinguish from non-member functions that produce a copy of the data and therefore cannot be usefully used as lvalues. For example, `A.T()` and `transpose(A)` both return the transpose of matrix `A`, but the former is faster since it does not make a copy of the original data, while the latter is more flexible since it can be applied to array expressions (e.g. `transpose(A*B)`). The member functions available are:

```
subset(int ibegin0, int iend0, ...)  This function returns a contiguous subset of an array as an array of the same rank that points to the original data. It takes twice as many arguments as the array has dimensions,
```
with each pair of arguments representing the indices to the first and last element to include from a particular
dimension. Exactly the same result can be obtained using `range` but the `subset` form is more concise. For
example, for a matrix `M`, `M.subset(1,5,3,10)` is equivalent to `M(range(1,5),range(3,10))`.

`T()` This function returns the transpose of a rank-2 array (a matrix). The returned array points to the same data
but with its dimensions reversed. A compile-time error occurs if this function is used on an array with rank
other than 2. Currently `Adept` doesn’t allow the transpose of a rank-1 array (a vector), since vectors are not
intended to have an intrinsic orientation. When orientation matters, such as in matrix multiplication, the
intended orientation may be inferred from the context or specified explicitly.

`permute(int i0, int i1, ...)` This function is the generalization of the transpose for multi-dimensional
arrays: it returns an array of the same rank as the original but with the dimensions rearranged according to
the arguments. There must be the same number of arguments as there are dimensions, and each dimension
(starting at 0) must be provided once only. The returned array is linked to the original; the permutation is
achieved simply by rearranging the list of dimensions and the list of “offsets” (the separation in memory of
elements along each dimension individually).

`diag(matrix)` When this function is applied to a rank-1 `Array` of length `n`, it returns an `n`-by-`n` diagonal matrix
(specifically a `DiagMatrix`; see section 3.10) that points to the data from the rank-1 array along its diagonal.

`diag_vector()` When this function is applied to a rank-2 `Array` with equally sized dimensions, it returns a rank-
1 array pointing to the data along its diagonals. An `invalid_operation` exception is thrown if applied to
a non-square matrix, and a compile-time error if applied to an array of rank other than 2.

`diag_vector(int i)` When applied to a square rank-2 `n`-by-`n` `Array`, this returns a rank-1 array of length
`n` − `abs(i)` pointing to the `i`th superdiagonal of the square matrix, or the `−i`th subdiagonal if `i` is negative.
An `invalid_exception` exception occurs if applied to a non-square matrix, and a compile-time error if
applied to an array of rank other than 2.

`submatrix_on_diagonal(int ibegin,int iend)` When applied to a square rank-2 array, this function returns a square matrix that shares part of the diagonal of the original
matrix. Thus `A.submatrix_on_diagonal(int ibegin,int iend)` is equivalent to `A(range(ibegin,iend),range(ibegin,iend))`. Its purpose is to provide a subsetting facility
for symmetric, triangular and band-diagonal matrices (see section 3.10) for which general array indexing is
not available. If applied to a non-square matrix, an `invalid_operation` exception will be thrown.

`reshape(int i0, int i1...)` Only applicable to an `Array` of rank 1, this returns a multi-dimensional array
whose dimensions are given by the arguments to the function. Between 2 and 7 dimensions are possible. If
the arguments are such that the total size of the returned array would not match the length of the vector, an
`invalid_dimension` exception is thrown.

### 3.5 Passing arrays to and from functions

When writing functions taking array arguments, there are three different ways to do it depending on the extent
to which the function needs to be able to modify the array. In the case of constant array arguments, a constant
reference should be used; for example:

```cpp
Real l3norm(const Vector& v) { // Function returning the L3-norm of a vector
    return cbrt(sum(v*v*v));
}

Vector w(3); w << 1.0, 2.0, 3.0; // Create a test vector
Real ans1 = l3norm(w); // Named vector argument
Real ans2 = l3norm(w(range(0,1))); // Temporary vector argument
Real ans3 = l3norm(2.0*w); // Expression implicitly converted to temporary vector
```
This function works with all three types of argument. The last example illustrates that when an inactive rank-1 expression is passed to the function, it is evaluated and the result placed in a temporary vector that is passed to the function.

At the other extreme, we may wish to create a function that modifies an array argument, including the possibility of changing its size; for example:

```cpp
void resize_and_zero(int n, Vector& v) { // A rather pointless function...
    v.resize(n); v = 0.0;
}
Vector w(4);
resize_and_zero(2,w); // Results in w=[0.0, 0.0]
resize_and_zero(2,w(range(0,2))); // Compile error: argument is temporary
resize_and_zero(2,2.0*w); // Compile error: argument is not an lvalue
```

In this case, due to the C++ rule that a non-constant reference cannot bind to a temporary object, the function can only take a non-temporary Vector as an argument. This is fair enough; it would not make sense to resize the subset of an array, or an expression. However, it is very common to want to pass a subset of an array to a function and for the function to modify the values of the array, but not to resize it. In Adept this is achieved as follows:

```cpp
void square_in_place(Vector v) {
    v *= v;
}
Vector w(3); w << 2.0, 3.0, 5.0;
square_in_place(w); // Results in w=[4.0, 9.0, 25.0]
square_in_place(w(range(0,1))); // Results in w=[4.0, 9.0, 5.0]
square_in_place(2.0*w); // No effect on w
```

Even though the Vector has been passed by value, the Vector copy constructor performs a “shallow copy”, which means that little more than the array dimensions and a pointer to the data are copied. Therefore, in the first two examples above the vector v inside the function points to data in w, and can therefore modify w. By contrast, when an expression is passed to the function, a new Vector is created to hold the result of the expression, and when this is modified inside the function it does not affect the data in the calling routine.

The fact that Array copy constructors perform shallow copies also improves the efficiency of functions that return arrays such as the following:

```cpp
Matrix square(const Matrix& in) {
    Matrix out = in*in; // Create an matrix containing the result of in*in
    return out;
}
Matrix A(100,100); // Allocate memory for "A"
Matrix B = square(A); // Copy constructor: shallow copy of "out" into "B"
```

At the return statement, matrix out is received by the copy constructor of matrix B, so a shallow copy is performed. This means that the description of matrix out is copied to B, including a pointer to Storage object containing both the data and a count of the number of references to it; this counter is increased by one. Matrix out is then destructed, and the counter is immediately reduced by one. The net result is that B has “stolen” the data in the matrix from out without it having been copied, thus avoiding unnecessary allocation of memory on the heap followed by copying and deallocation.

The shallow-copy implementation leads to behaviour that users may not be expecting. If an array is initialized from another array in either of the following two ways:

```cpp
Matrix M(3,4);
Matrix A(M); // Call copy constructor
Matrix B = M; // Call copy constructor
```

then the result is that A, B and M share the same data, rather than a copy being made. To make a deep copy, it is necessary to do the following:

```cpp
Matrix M(3,4);
Matrix A; // Create empty matrix
A = M; // Call assignment operator for deep copy
```
This is annoying, but the alternative is that there would be no clean way to pass a subset of an array to a function that then modifies its values. The same behaviour is implemented in the Blitz++ array class (Veldhuizen, 1995).

It should be noted that with the introduction of “move semantics” in the C++11 standard, the it is possible to detect when an array returned from a function is about to be destructed, and therefore invoke a move constructor that implements a shallow copy. This negates one of the two reasons from making the copy constructor execute only a shallow copy. But it does not help in passing array subsets to functions, unless two versions of every function were created, one accepting an lvalue reference (Array&) and the other accepting an rvalue reference (Array&&), which is hardly practical.

If you compile your code with C++11 features enabled then move semantics can sometimes make assignment more efficient. Consider code calling the square function above:

```cpp
Matrix A(10,10), B(10,10);
B = square(A); // Move assignment operator performs shallow copy
Matrix C(B); // B and C now share the same data
B = square(A); // Move assignment operator performs deep copy
```

Both assignments are to temporary objects about to be destructed, so the move assignment operator is called. This operator checks how many references there are to the data in B. In the first case there is only one reference, so the data in B can safely be discarded and a shallow copy (a “move”) of the data in the temporary is performed. In the second case there are two references, so a deep copy must be performed in order that C sees the change in B.

### 3.6 Array reduction operations

A family of functions return a result that is reduced in rank compared to their argument, and operate in the same way as Fortran functions of the same name. Consider the sum function, which can be used either to sum all the elements in an array expression and return a scalar, or to sum elements along the dimension specified in the second argument and return an array whose rank is one less than the first argument:

```cpp
Array A(3, 4);
Real x = sum(A); // Sum all elements of matrix A
Vector v = sum(A, 1); // Sum along the row dimension returning a vector of length 3
```

Functions that are used in the same way are mean, product, minval, maxval and norm2 (the square-root of the sum of the squares of each element). Note the difference between maxval and max: the behaviour of max is outlined in section 3.3. Three further functions operate in the same way but on boolean arrays: all returns true only if all elements are true, any returns true if any element is true (and false otherwise), while count returns the number of true elements. Each of these can work on an individual dimension as with sum and friends.

A further function, dot_product(a, b), takes two arguments that must be rank-1 arrays of the same length and returns the dot product. This is essentially the same as sum(a*b).

### 3.7 Array expansion operations

The function outer_product(x, y) returns the outer product of two rank-1 expressions; if x and y are interpreted as column vectors then xy^T is returned. If outer_product is used in an expression then an intermediate matrix object is not created to store it.

The function spread<dim>(A, n) returns an array that replicates the A array n times along dimension dim. The returned array has a rank one larger than a whose dimension dim is n and the remaining dimensions are the same as those of A. It is essentially the same as the Fortran function of the same name, but dim is provided as a template argument since performance is improved if this is known at compile time. The following illustrates spread for an argument of rank 1:

```cpp
Vector v(3); v << 1, 2, 3;
Matrix M0 = spread<0>(v, 2); // M1 contains {{1, 2, 3},
// {1, 2, 3}}
```
3.8 Conditional operations

There are two main ways to perform an operation on an array depending on the result of a boolean expression. The
first is similar to the Fortran where construct:

```cpp
Array A(3,4);
Array B(3,4);
A.where(B > 0.0) = 2.0 * B; // Only assign to A if B > 0
A.where(!isnan(B)) = either_or(-B, 0.0); // Read from either one expression or the other
```

In the first example, \( A \) is only assigned if a condition is met, and therefore \( A \) must be of the same size and rank of
the boolean expression. In the second example \( A \) is filled with elements from the first argument of either_or if the
boolean expression is true, or from the second argument otherwise; if \( A \) is empty then it will be resized to the size
of the boolean expression. In both cases, the expressions on the right-hand-side may be scalars or array expressions
of the same size as the boolean expression. Equivalent expressions are possible replacing the assignment operator
with the +=, -=, *= and /= operators, in which case \( A \) must already be the same size as the boolean expression.

An alternative approach that works with only vectors uses the find function. This is similar to the equiva-
lent Matlab function and returns an indexVector (a vector of integers of sufficient precision to index an array)
containing indices to the true elements of the vector:

```cpp
Vector v(10), w(10);
v(find(v > 5.0)) = 3.0;
IndexVector index = find(v > 5.0);
v(index) = 2.0 * u(index);
```

This will work if no true elements are found: find will return an empty array, and when \( v \) is indexed by an empty
vector, no action will be taken. In general, find is less efficient than where.

3.9 Fixed-size arrays

The size of the Array class is dynamic, which is somewhat sub-optimal for small arrays whose dimensions are
known at compile time. Adept provides an alternative class template for an array whose size is known at compile
time and whose data are stored on the stack. It has the following declaration:

```cpp
namespace adept {
    template <typename Type, bool IsActive, int Dim0, int Dim1 = 0, ...>
    class FixedArray;
}
```

The type (e.g. double) and activeness are specified by the first two template arguments, while the remaining
template arguments provide the size of the dimensions, up to 7. Only as many sizes need to be specified as there
are dimensions. A user working with arrays of a particular size could use typedef to provide convenient names; for
example:

```cpp
typedef FixedArray<double, false, 4> Vector4;
typedef FixedArray<double, false, 4, 4> Matrix44;
typedef FixedArray<double, true, 4> aVector4;
typedef FixedArray<double, true, 4, 4> aMatrix44;
```

In the adept namespace, Adept defines Vector2, Vector3, Matrix22, Matrix33 and their active counterparts.
Fixed arrays have all the same capabilities as dynamic arrays, with a few exceptions:

- Since their size is fixed, there are no member functions resize, clear or in_place_transpose.
3.10 Special square matrices

Adept offers several special types of square matrix that can participate in array expressions. They are more efficient than Arrays in certain operations such as matrix multiplication and assignment, but less efficient in operations such as accessing individual elements. All use an internal storage scheme compatible with BLAS (Basic Linear Algebra Subprograms). All are specializations of the `SpecialMatrix` class template, which has the following declaration:

```cpp
namespace adept {
    template <typename Type, class Engine, bool IsActive = false>
    class SpecialMatrix;
}
```

The first template argument is the numerical type, the second provides the functionality specific to the type of matrix being simulated, and the third states whether the matrix participates in the differentiation of an algorithm. The specific types of special matrix are as follows:

**Square matrices.** `SquareMatrix` provides a dense square matrix of type `Real` with `aSquareMatrix` its active counterpart. Its functionality is similar to a rank-2 `Array`, except that its dimensions are always equal and the data along its fastest varying dimension are always contiguous in memory, which may make it faster than `Array` in some instances.

**Symmetric matrices.** `SymmMatrix` provides a symmetric matrix of type `Real`, and `aSymmMatrix` is its active equivalent. Internally this type uses row-major unpacked storage with the data held in the lower triangle of the array and zeros in the upper triangle (equivalent to column-major storage with data in the upper triangle). If the opposite configuration is required then it is available by specifying different template arguments to the `SpecialMatrix` class template. Note that with normal access methods, the storage scheme is opaque to the user; for example, `S(1,2)=2.0` and `S(2,1)=2.0` have the same effect.

**Triangular matrices.** `LowerMatrix` and `UpperMatrix` (and their active equivalents prefixed by “a”) provide triangular matrices of type `Real`. Internally they use row-major unpacked storage, although column-major storage is available by specifying different template arguments to the `SpecialMatrix` class template.

**Band diagonal matrices.** `DiagMatrix`, `TridiagMatrix` and `PentadiagMatrix` provide diagonal, tridiagonal and pentadiagonal `Real` matrices, respectively (with their active equivalents prefixed by “a”). Internally, row-major BLAS-type band storage is used such that an `n`-by-`n` tridiagonal matrix stores `3n` rather than `n^2` elements. Adept supports arbitrary numbers of sub-diagonals and super-diagonals, accessible by specifying different template arguments to the `SpecialMatrix` class template.

A `SpecialMatrix` can be constructed and resized as for `Arrays` (see section 3.2), with the following additions:

```cpp
SymmMatrix S(4); // Initialize a 4-by-4 symmetric matrix
S.resize(5);     // Resize to a 5-by-5 matrix
```

These are applicable to all types of `SpecialMatrix`.

In terms of array indexing and slicing, the member functions `t`, `diag` and `diag_submatrix` described in section 3.4 are all available, but if you index a `SpecialMatrix` with `S(a,b)` then `a` and `b` must be scalars or scalar expressions. For triangular or band-diagonal matrices, if the requested element is one of the zero parts of the matrix then it can only be used as an rvalue in an expression. If you wish to extract arbitrary subarrays from a `SpecialMatrix` then it must first be converted to a `Matrix`:
3.11 Matrix multiplication

Matrix multiplication may be invoked in two equivalent ways: using the matmul function or the ‘‘★’’ pseudo-operator. Following Fortran, the two arguments may be either rank-1 or rank-2, but at least one argument must be of rank-2. The orientation of any rank-1 argument is inferred from whether it is the first or second argument, as shown here:

\[
\text{Matrix } A(3,5), B(5,3), C;
\]
\[
\text{Vector } v(5), w;
\]
\[
C = \text{matmul}(A,B); \quad \text{// Matrix-matrix multiplication: return a 3x3 matrix}
\]
\[
w = \text{matmul}(v,B); \quad \text{// Interpret v as a row vector: return a vector of length 3}
\]
\[
w = \text{matmul}(A,v); \quad \text{// Interpret v as a column vector: return a vector of length 3}
\]

In this way it is never necessary to transpose a vector; the appropriate orientation to use is inferred from the context.

You may find it clearer to use ‘‘★’’ for matrix multiplication as illustrated here:

\[
\text{Matrix } A(3,5), B;
\]
\[
\text{SymmMatrix } S(5);
\]
\[
\text{Vector } c, x(5);
\]
\[
c = A \star \star \log(S) \star \star x; \quad \text{// Returns a vector of length 3}
\]
\[
c = \text{matmul}(\text{matmul}(A,\log(S)),x); \quad \text{// Equivalent to the previous line but using matmul}
\]
\[
c = A \star \star (\log(S) \star \star x); \quad \text{// As the previous example but more efficient}
\]
\[
B = 2.0 \star S \star A.T(); \quad \text{// Returns a 5-by-3 matrix}
\]
\[
B = 2.0 \star S \star A; \quad \text{// Run-time error: inner dimensions don’t match}
\]

The ‘‘★’’ pseudo-operator has been implemented in Adept by overloading the dereference operator such that ‘‘★λ’’ returns a special type when applied to array expressions, and overloading the multiply operator to perform matrix multiplication when one of these types is on the right-hand-side. This means that ‘‘★’’ has the same precedence as ordinary multiplication, and both will be applied in order of left to right. Thus, in the first example above, matrix-matrix multiplication is performed followed by matrix-vector multiplication. The second example shows how to make this more efficient with parentheses to specify that the rightmost matrix multiplication should be applied first, leading to two matrix-vector multiplications. The final example shows an expression that would fail at runtime with an inner_dimension.mismatch exception due to the matrix multiplication being applied to matrices whose inner dimensions do not match.

You cannot use matmul or ‘‘★’’ for vector-vector multiplication, since it is ambiguous whether you require the inner product (dot product) or the outer product. Therefore you must explicitly call the function dot_product (section 3.6) or outer_product (section 3.7).

In order to get the best performance, Adept does not use expression templates for matrix multiplication but rather calls the appropriate level-2 BLAS function for matrix-vector multiplication and level-3 BLAS function for matrix-matrix multiplication. For matrix multiplication involving active vectors and matrices, Adept first uses BLAS to perform the matrix multiplication and then stores the equivalent differential statements. There are therefore a few factors that users should be aware of in order to get the best performance:

- If an array expression rather than an array is provided as an argument to matrix multiplication, it will first be converted to an Array of the same rank. Therefore, if the same expression is used more than once in a sequence of matrix multiplications, better performance will be obtained by precomputing the array expression and storing it in a temporary matrix:

\[
\begin{align*}
\text{SymmMatrix } S(6); \\
\text{intVector } \text{index}(3); \\
\text{index} << 2, 3, 5; \\
\text{Matrix } M = \text{Matrix}(S)(\text{index, stride}(0,4,2));
\end{align*}
\]

\[
\text{A drawback of the ‘‘★’’ interface with the orientation of vector arguments being inferred is that in an expression like } \lambda \star \star v \star \star B \text{ (where } \lambda \text{ and } B \text{ are matrices and } v \text{ is a vector), } v \text{ is interpreted as a column vector in } \lambda \star \star v, \text{ which returns a column vector result, but this result is then implicitly transposed when it is used as the left-hand argument of the matrix multiplication with } B. \text{ Moreover, the order of precedence affects the result, since this expression will not give the same answer as } \lambda \star \star (v \star \star B).\]
3.12 Linear algebra

Matrix $A(5,5), B(5,5), C(5,5), D(5,5)$

// Slow implementation:
$C = \text{transpose}(2.0A*B) ** (2.0A*B);$  
$D = (2.0A*B) ** C;$

// Faster implementation:
{
  Matrix tmp = 2.0*A*B;
  C = tmp.T() ** tmp;
  D = tmp ** C;
} // "tmp" goes out of scope here

- If the left-hand argument of a matrix multiplication is a symmetric, triangular or band matrix then a specialist BLAS function will be used that is faster than the one for general dense matrices. *Adept* may not be able to tell if the result of an array expression is symmetric, triangular or has a band structure, and so may not call the most efficient BLAS function. The user can help as follows:

SymmetricMatrix $S(5,5)$

Matrix $A(5,5), B(5,5)$

$B = (2.0*\exp(S)) ** A; // Slower$

$B = \text{SymmMatrix}(2.0*\exp(S)) \ast A; // Faster$

- BLAS requires that the fastest-varying dimension of input matrices are contiguous and increasing. This is always the case for the special square matrices described in section 3.10, but not necessarily for a Matrix or an aMatrix, which are particular cases of the general Array type. If the fastest-varying dimension of such a matrix is not contiguous and increasing then *Adept* will copy it to a temporary matrix before invoking matrix multiplications, as in the following example:

Matrix $A(5,5), B, C(5,5)$

$B.<\text{link}(A(__, \text{stride}(\text{end,1,-1})); // Fastest varying dim is contiguous but decreasing$

$C = A \ast A; // Matrix multiplication applied directly with A$

$C = B \ast B; // Adept will copy B to a temporary matrix before multiplication$

An additional member function to mention in this section is *inplace_transpose()*, which is only applicable to matrices. It transposes the matrix by swapping the dimensions and the offsets to each dimension, but leaving the actual data untouched. This means that a matrix with row-major storage will be changed to column-major, and vice versa.

*Adept* can differentiate expressions involving matrix multiplication, but this is far from optimal in *Adept* version 2.0, for two reasons. Firstly, only differentiation of dense matrices has been implemented, so when matrix multiplication is applied to active “special matrices” (symmetric, band, upper-triangular and lower-triangular matrices), they are first copied to a dense matrix. Secondly, the *Adept* stack format can currently only store differential states for scalar expressions, which for matrix multiplication leads to lots of repeated values on the stack. A future version of *Adept* will redesign the stack to allow matrices to be stored in it; this will be much faster and much less memory-hungry.

### 3.12 Linear algebra

*Adept* provides the functions *solve* and *inv* to solve systems of linear equations and to invert a matrix, respectively, which themselves call the most appropriate function from LAPACK.

Matrix $A(5,5), Ainv(5,5), X(5,5), B(5,5)$

SymmetricMatrix $S(5,5), Sinv(5);$  

Vector $x(5), b(5);$  

$\text{Ainv} = \text{inv}(A); // Invert general square matrices using LU decomposition$

$\text{Sinv} = \text{inv}(S); // Invert symmetric matrices using Cholesky decomposition$

$x = \text{solve}(A,b); // Solve general system of linear equations$

$X = \text{solve}(S,B); // Solve symmetric system of linear equations with matrix right-hand-side$

Statements involving *solve* and *inv* cannot yet be automatically differentiated. When the *Adept* stack is redesigned to hold matrices, this capability will be added.
3.13 Bounds and alias checking

When encountering an array or active expression, Adept performs several checks to test the validity of the expression both at compile time and at runtime:

**Activeness check.** An expression in which an active expression is assigned to an inactive array will fail to compile.

**Rank check.** An expression will fail to compile if the rank of the array on the left-hand-side of the “=” operator (or the operators “+=”, “-=”, etc.) does not match the rank of the array expression on the right-hand-side. However, a scalar (rank-0) expression can be assigned to an array of any rank; its value will be assigned to all elements of the array. Compile-time rank checks are also performed for each binary operation (binary operators such as “+” and binary functions such as `pow`) making up an array expression: compilation will fail if the two arguments do not have the same rank and neither is of rank 0.

**Dimension check.** When a binary operation is applied to two array expressions of rank \( n \) then Adept checks at run-time that each of the \( n \) dimensions has the same length. Otherwise, a size_mismatch exception is thrown.

**Alias check.** By default, Adept checks to see whether the memory referenced in the array object on the left-hand-side of a statement overlaps with the memory referenced by any of the objects on the right-hand-side, as in this example of a shift-right operation:

```cpp
Vector v(6);
v(range(1,end)) = v(range(0,end-1));
```

In order to prevent the right-hand-side changing during the operation, Adept copies the expression on the right-hand-side to a temporary array and then assigns the left-hand-side array to this temporary, which is equivalent to the following:

```cpp
{
    Vector tmp;
    tmp = v(range(0,end-1));
    v(range(1,end)) = tmp;
} // tmp goes out of scope here
```

However, for speed Adept does not check to see whether individual memory locations are shared; rather the start and end memory locations are checked to see if they overlap. This means that for certain strided operations, copying to a temporary array is unnecessary. Nor is it necessary if elements of an array will be accessed in exactly the same order on the left-hand-side as the right-hand-side. If the user is sure that alias checking is not necessary then he or she can override alias checking for part or all of an array expression using the `noalias` function, as follows:

```cpp
v(stride(1,end,2)) = noalias(v(stride(0,end-1,2))); // No overlap between RHS and LHS
v = 1.0 + noalias(exp(v)); // LHS & RHS accessed in same order
```

Note that for speed, alias checking is not performed if the left-hand-side is a `FixedArray`, since such arrays can never point to another location and therefore aliasing is less likely to arise. Aliasing is still possible if one of the terms on the right-hand-side points to the data in the `FixedArray` on the left. In this case, you can use the `eval` function, which takes a non-scalar expression as an argument, and returns an array containing a copy of the data. For example:

```cpp
FixedArray<Real,false,3> v = {1.0, 2.0, 3.0}; // C++11 initialization of inactive vector
v = v(stride(end,0,-1)); // Aliasing leads to v = {3.0, 2.0, 3.0}
v = eval(v(stride(end,0,-1))); // Expected result: v = {3.0, 2.0, 1.0}
```

To avoid the overhead of alias checking, you can define the preprocessor variable `ADEPT_NO_ALIAS_CHECKING`, but then it is up to the user to identify the statements where aliasing will occur and use the `eval` function to ensure the correct behaviour.
3.14 Automatic differentiation capabilities specific to arrays

Section 2.3 described how the get_gradient() member function could be used to extract the gradients from a scalar adouble object after applying forward- or reverse-mode differentiation. In the same way, gradients may be extracted from active Array and FixedArray objects, returning an inactive Array of the same rank and size. For example, to compute the derivative of a norm2 operation, we could do the following:

```cpp
Stack stack;  // Stack to store differential statements
aVector x = {1.0, 2.0, 3.0};  // C++11 initialization
stack.new_recording();  // Clear any stored differential statements
aReal y = norm2(x);  // Perform operation to be differentiated
y.set_gradient(1.0);  // Seed the independent variable
stack.reverse();  // Reverse-mode differentiation
Vector dy_dx = x.get_gradient();  // Extract vector of derivatives
```

3.15 Array thread safety

There are numerous ways of obtaining an Array that links to data in another Array object; not only the “>>=” link operator described in section 3.2, but also the various subsetting member functions described in section 3.4, and even just passing arrays to and from functions. This avoids deep copying and so improves efficiency. In addition to the new Array pointing to the same data, it also points to the same Storage object, and when a new link is created, the counter in this object indicating the number of objects pointing to it is incremented. This ensures that the data will remain provided there is at least one object linking to it. A downside of this model is that if multiple threads access an array simultaneously, even if just to read it, then the reference counter can become corrupted. There are two solutions to this problem.

If you are using C++11 then you can define the ADEPT_STORAGE_THREAD_SAFE preprocessor variable, which makes the reference counter in Storage objects of type std::atomic<int> and thereby protects all operations on them by a mutex. This may degrade the efficiency of your code since the mutex will be redundant in single-threaded code.

Alternatively, we use the capability of arrays to access data not held in a Storage object. The Array and SpecialMatrix classes have a soft_link() member function that returns an object of the same type, size and activeness, which points to the same data but does not contain a link to the Storage object:

```cpp
Matrix M(2,2);  // ...enter multi-threaded environment
Matrix N;
N >>= M.soft_link();  // N links to same data as M but without Storage object
Vector v = M.soft_link()(__,0);  // v links to subset of M but without Storage object
// (recall that the copy constructor is called here)
```

The linked objects may be used in the same way as any other Array. This is demonstrated in the test_thread_safe_arrays test program.

3.16 Writing an array to a stream

As you would expect, an array can be written to a stream with the “<<” operator:

```cpp
Vector v = {1, 2};  // Using C++11 initializer lists
Matrix M = {{3, 4}, {5, 6}};  // for convenience
std::cout << v << "\n";
std::cout << M << "\n";
```
which by default produces

```
1 2
3 4
5 6
```

You can change the output to use curly brackets to indicate the dimensions of the array as follows:

```
Vector v = {1, 2};
Matrix M = {{3, 4}, {5, 6}};
adept::set_array_print_style(PRINT_STYLE_CURLY);
std::cout << "v = " << v << ";
std::cout << "M = " << M << ";
```

which produces output that looks like C/C++ code:

```
v = {1, 2};
M = {{3, 4}, {5, 6}};
```

The available print styles for use by `set_array_print_style` are `PRINT_STYLE_PLAIN` (default), `PRINT_STYLE_CURLY`, `PRINT_STYLE_CSV` (comma-separated values) and `PRINT_STYLE_MATLAB` (matrix ordering indicated by Matlab-style semi-colons and square brackets).

### 3.17 Fortran interoperability

The traditional way to pass arrays between Fortran and C/C++ makes use of the fact that Fortran passes its “explicit-shape” arrays (the type used since Fortran-77) to and from routines simply as a pointer to the first element of the array. It is then up to the receiving routine to declare the size of the array correctly. *Adept* arrays can therefore be passed to Fortran routines using their `data()` and `const data()` member functions, which return pointers to the first element of the array.

Since Fortran-90, the language also supports “assumed-shape” arrays, which are very much like *Adept’s Array* objects: they contain within them the extent of each array dimension, and may refer to data that are strided (non-contiguous) in memory. Fortran passes an assumed-shape array to subroutines and functions in the form of a pointer to its *array descriptor* (sometimes known as a *dope vector*), which contains a pointer to the first element of the array and information on the rank, type, and the extent and stride-in-memory of each dimension.

The Fortran 2018 standard defines an interface to allow assumed-shape arrays to be passed to and from C or C++ functions. Fortran compilers supporting this standard provide a C/C++ header file `ISO_Fortran_binding.h` that defines the array descriptor as a structure `CFI_cdesc_t`. The *Adept* header file `adept_fortran.h` provides a class `adept::FortranArray`, a thin wrapper to this structure, that enables an *Adept* Array object to share its data with a Fortran array. This is very efficient as only the array descriptor information is copied, not the actual data in the array. At the time of writing, support for this capability in Fortran compilers is limited.

An crucial point to be aware of in all the examples that follow is that *Adept* indexes its arrays in row-major order starting at 0, while Fortran indexes its arrays in column-major order starting (by default) at 1. When arrays are passed between the two languages, the native array convention is adopted. Therefore, matrix element \( A(0,10) \) in *Adept* would be indexed as \( A(11,1) \) in Fortran.

#### 3.17.1 Passing arrays from C++/Adept to Fortran

Suppose we have a Fortran subroutine that takes an integer array and a single-precision array as arguments:

```fortran
! Define a routine callable with same name in C/C++
subroutine fortran_routine(int_array, flt_array) bind(c)
    implicit none
    integer(kind=4), intent(inout) :: int_array(:, :) ! Matrix of 4-byte integers
    real(kind=4), intent(inout) :: flt_array(:, :) ! Matrix of 4-byte real numbers
! --- Body of routine here ---
end subroutine fortran_routine
```
3.17. Fortran interoperability

The following C++ program demonstrates how Adept arrays can be passed this routine:

```cpp
#include <adept_fortran.h>
// Declare interface to the routine, turning off C++ name mangling so that it can be linked
// to Fortran
extern "C" void fortran_routine(adept::FortranArray* int_array,
                                      adept::FortranArray* flt_array);

int main() {
  // Initialize Adept matrices, using shortcuts to the types Array<2,int> and Array<2,float>
  adept::intMatrix int_arr = {{2, 3, 5}, {7, 11, 13}};
  adept::floatMatrix flt_arr = {{2.0, 3.0, 5.0}, {7.0, 11.0, 13.0}};
  // Convert Adept arrays to Fortran arrays pointing to the same data, and call the routine;
  // the conversion to FortranArray pointers is done automatically
  fortran_routine(adept::FortranArray(int_arr),
                  adept::FortranArray(flt_arr));
  return 0;
}
```

This will fail to compile if ISO_Fortran_binding.h file is not found. To link the two object files into an executable you will need to use your C++ compiler, but include the relevant Fortran library on the command line (e.g. -lgfortran if you compiled fortran_routine with the GNU Fortran compiler, or -lifcore if you used the Intel Fortran compiler).

3.17.2 Passing arrays from Fortran to C++/Adept

We can also pass arrays the other way. Consider the following Fortran program:

```fortran
program test_interoperability
  implicit none

  ! Define interface to a function implemented in C++
  interface
    subroutine adept_routine(int_array, flt_array) bind(c)
      integer(kind=4), intent(inout) :: int_array(:,:) ! Matrix of 4-byte integers
      real(kind=4), intent(inout) :: flt_array(:,:) ! Matrix of 4-byte real numbers
    end subroutine
  end interface

  ! Body of program starts here
  integer(kind=4), allocatable :: imat(:,:)
  real(kind=4), allocatable :: fmat(:,:)

  ! --- Code to allocate and populate imat and fmat here ---
  ! Now call the C++ function
  call adept_routine(imat, fmat)
end program test_interoperability
```

The routine could be implemented in C++ as follows:

```cpp
#include <adept_fortran.h>
extern "C" void adept_routine(adept::FortranArray* int_array,
                              adept::FortranArray* flt_array) {
  // Declare Adept arrays
  adept::intMatrix int_arr;
  adept::floatMatrix flt_arr;

  // Associate Adept arrays with Fortran data, or throw a fortran_interoperability_error
  // exception if the rank or type do not match
  int_arr >>= int_array;
  flt_arr >>= flt_array;

  // --- Operations on int_array and flt_array now modify the Fortran arrays ---
}
```

Since the executable now contains a Fortran source file with a program statement, rather than a C++ source file defining a main function, the linking step of the compilation must be carried out using the Fortran compiler, but passing it the C++ standard library, i.e. -lstdc++.

In the example above, the Adept arrays int_arr and flt_arr behave in the same way as “linked” arrays described in section 3.2: they know that they do not “own” the original data, so if the user then calls their clear
or resize member functions, they will unlink themselves from the Fortran arrays. The FortranArray class provides no array features itself, so must be linked to an Array object before any work can be done on it, but it does provide a handful of member functions for querying its properties:

```cpp
int rank() Return the number of dimensions of the array.
int dimension(int i) Return the extent of dimension i in memory, counting dimensions from 0 but using the Fortran ordering.
int offset(int i) Return the stride in memory of dimension i.
bool is_type<Type>() Return true if the element type of the array is the same as Type (which must be a known type at compile time).
Type* data<Type>() Return a pointer to the first element of the data, cast to the specified type.
```
Chapter 4

Using Adept’s optimization functionality

Note that this is a recently added capability in Adept, and so only a small number of minimizations algorithms are currently available.

4.1 Background

Since version 2.0.8, Adept provides functionality for solving non-linear optimization problems, specifically finding the state vector \( x \) that minimizes the scalar cost function \( J(x) \) (also known as a penalty function or objective function). A gradient-free minimization algorithm (e.g. Nelder-Mead) requires simply a user-supplied function for computing \( J \), calling it multiple times for different \( x \) to find the minimum \( J \). A first-order minimization algorithm requires also a user-supplied function returning the gradient of the cost function \( \partial J / \partial x \) (a vector). Examples are the Conjugate Gradient method and the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method. Knowing the gradient enables such algorithms to find the minimum with far fewer function calls, although a function call returning \( \partial J / \partial x \) is slower than one returning only \( J \). Adept’s optimization interface is in terms of passive array types, so the user is not obliged to use Adept’s automatic differentiation capability to compute these gradients, although the examples in this chapter assume that they do. A second-order minimization algorithm makes use of not only \( J \) and \( \partial J / \partial x \), but also a user-supplied function for the Hessian \( A = \partial^2 J(x) / \partial x^2 = \nabla_x^2 J \) (a symmetric matrix). Examples are the Gauss-Newton and Levenberg-Marquardt methods. Knowing the second derivative means that even fewer iterations should be required to find the minimum of \( J \), but \( A \) is more expensive to compute than \( \partial J / \partial x \).

Adept does not have the ability to automatically compute Hessian matrices for an arbitrary cost function, but frequently the cost function has a specific form that makes it possible to compute the Hessian from the Jacobian matrix. Consider the optimization problem of finding the parameters \( x \) of non-linear model \( y(x) \) that provides the closest match to a set of “observations” \( y_o \) in a least-squares sense. For maximum generality we add constraints that penalize differences between \( x \) and a set of a priori values \( x^a \), as well as a regularization term. In this case the cost function could be written as

\[
J(x) = \frac{1}{2} [y(x) - y_o]^T R^{-1} [y(x) - y_o] + \frac{1}{2} [x - x^a]^T B^{-1} [x - x^a] + \frac{1}{2} x^T T x. \tag{4.1}
\]

Here, all vectors are treated as column vectors, \( R \) is the error covariance matrix of the observations, \( B \) is the error covariance matrix of the a priori values, and \( T \) is a Twomey-Tikhonov matrix that penalizes either spatial gradients or curvature in \( x \). The Hessian matrix is then given by

\[
A = H^T R^{-1} H + B^{-1} + T, \tag{4.2}
\]

which can be coded up using Adept to compute the Jacobian matrix \( H = \partial y / \partial x \). Each term on the right-hand-side of (4.2) has its corresponding term in (4.1), so it is easy to work out what the Hessian would look like if only a subset of the terms in (4.1) were present.
4.2  Adept interface

For the purposes of demonstrating how this would be implemented in Adept we simplify (4.1) down to the case of minimizing a quadratic function, in which case \( J = y^T y / 2 \) and \( y = x \). The former of these two equations means that the Hessian matrix is simply \( A = H^T H \). The latter we implement using active variables:

```cpp
adep::aVector calc_y(const adept::aVector& x) { return x; }
```

The test program `test/test_minimizer.cpp` uses a \( y(x) \) function for the much more interesting case of the \( N \)-dimensional Rosenbrock function. To set up the problem ready for minimizing, we use create a class that derives from Adept's Optimizable class and overrides five of its virtual functions:

```cpp
#include <adep_optimize.h>

class SimpleOptimizable : public adept::Optimizable {
public:
  // Return the cost function for a given state vector x
  virtual adept::Real calc_cost_function(const adept::Vector& x) {
    adept::Vector y = value(calc_y(x)); // "value" throws away the activeness
    return 0.5*sum(y*y);
  }
  // Calculate the cost function and its gradient from x
  virtual adept::Real calc_cost_function_gradient(const adept::Vector& x, adept::Vector gradient) {
    adept::aVector xactive = x; // Copy x to an active variable
    stack.new_recording();
    adept::aVector y = calc_y(xactive); // Calculate y from x
    adept::aReal cost = 0.5*sum(y*y); // Calculate cost function as an active variable
    cost.set_gradient(1.0); // Use reverse-mode differentiation to compute the gradient
    gradient = xactive.get_gradient(); // Return cost function as passive variable
    return value(cost);
  }
  // Calculate the cost function, its gradient and the Hessian matrix
  virtual adept::Real calc_cost_function_gradient_hessian(const adept::Vector& x, adept::Vector gradient, adept::SymmMatrix& hessian) {
    adept::aVector xactive = x; // Copy x to an active variable
    stack.new_recording();
    adept::aVector y = calc_y(xactive); // Calculate y from x
    adept::aReal cost = 0.5*sum(y*y); // Calculate cost function as an active variable
    stack.independent(xactive); // Define independent variables
    stack.dependent(y); // Define dependent variables
    adept::Matrix jac = stack.jacobian(); // Compute Jacobian matrix dy/dx
    hessian = jac.T() ** jac; // Hessian is a simple matrix product of Jacobian
    gradient = jac.T() ** value(y); // Gradient is a matrix-vector product
    return value(cost); // Return cost function as passive variable
  }
  // Every iteration this function is called: here simply report progress to standard output
  virtual void report_progress(int niter, const adept::Vector& x, adept::Real cost, adept::Real gnorm) {
    std::cout << "Iteration " << niter << " cost function = " << cost << " \n"
  }
  // Minimization algorithm may want to check what derivatives are available: here we provide 0th (cost function alone) 1st (gradient) and 2nd (Hessian), so return true for 0, 1 or 2, false otherwise
  virtual bool provides_derivative(int order) { return (order >= 0 && order <= 2); }
}
```

// Keep an instance of the Adept stack within the class: avoids the initialization costs
// incurred each iteration if it was inside calc_cost_function_gradient and
calc_cost_function_gradient_hessian
Note that if you plan to use a first-order minimization algorithm, you do not need to provide a `calc_cost_function`, `gradient`, or `hessian` function.

Adept’s `Minimizer` class can minimize the cost function held in an `Optimizable` object by calling the user-supplied virtual functions, as follows:

```cpp
SimpleOptimizable quadratic_function;
adept::Minimizer minimizer(MINIMIZER_ALGORITHM_LEVENBERG); // Select minimization algorithm
int nx = 10; // Number of state variables
adept::Vector x(nx); // Declare state vector
x = 3.0; // Initialize state vector to first guess values, all 3.0
// Minimize the cost function:
adept::MinimizerStatus status = minimizer.minimize(quadratic_function, x);
// Report the convergence status:
std::cout << "Convergence status: " << adept::minimizer_status_string(status) << "\n";
```

After the `minimize` member function is called, `x` contains the state vector that minimizes the cost function.

The available minimization algorithms are:

**MINIMIZER_ALGORITHM_LEVENBERG** The second-order Levenberg algorithm tries to perform a Gauss-Newton step using the Hessian matrix and assuming that the curvature of `J` is locally constant. If `J` at the new `x` is not reduced by this step then a damping parameter `\lambda` is used to scale between Gauss-Newton and a steepest-descent algorithm.

**MINIMIZER_ALGORITHM_LEVENBERG_MARQUARDT** The second-order Levenberg-Marquardt algorithm is similar to the Levenberg algorithm, but scales such that the step sizes are changed in each dimension according to the curvature of the cost function in that dimension (i.e. the diagonal of the Hessian matrix). This generally, but not always, results in faster convergence than the Levenberg algorithm.

Later versions of Adept will include more minimization algorithms, particularly first-order. The minimizer can be configured in detail by calling its member functions listed in section 4.3. The possible values for the return status are given in section 4.4.

The case above is an example of unconstrained minimization: the minimizer is free to try any values of `x`. This can lead to it trying unphysical values, such as negative values for a quantity that cannot be negative. To prevent this, all the minimization algorithms allow the user to specify simple box constraints on the elements of the state vector. Suppose we wanted to constrain element 0 to be positive and element 1 to lie in the range 10–20, we would add these lines:

```cpp
// Declare vectors containing the lower and upper bounds on x
adept::Vector x_lower, x_upper;
// Set them to the minimum and maximum possible values for their element type (e.g. double)
adept::minimizer_initialize_bounds(nx, x_lower, x_upper);
// Set a lower bound on element 0 and both bounds on element 1
x_lower(0) = 0.0;
x_lower(1) = 10.0;
x_upper(1) = 20.0;
// Call the minimize function with two extra arguments specifying the bounds
status = minimizer.minimize(quadratic_function, x, x_lower, x_upper);
```

## 4.3 Other member functions of the Minimizer class

In addition to the `minimize` member function described in section 4.2, the following `adept::Minimizer` member functions may be called to configure the behaviour of the minimization algorithm:

**void set_algorithm(MinimizerAlgorithm algo)** Set the algorithm to one of the available minimization algorithm, e.g. `MINIMIZER_ALGORITHM_LEVENBERG`. This is an alternative to providing it as an argument to the `Minimizer` constructor.
void set_algorithm(const std::string& algo) Set the algorithm using a string, currently either “Levenberg” or “Levenberg-Marquardt”.

void set_max_iterations(int max_it) Set the maximum number of iterations (default 100).

void set_converged_gradient_norm(Real cgn) The L2-norm of the $\partial J/\partial x$ vector is computed each iteration, and convergence is deemed to have been achieved when it falls below the value specified here (default 0.1).

void set_max_step_size(Real mss) Set the maximum step size each iteration (default: no maximum). A negative or zero value indicates that no maximum step size is to be used.

void ensure_updated_state(int order = 2) Often the user will require the Hessian matrix to compute errors in the solution, and will store the Hessian matrix each time the calc_cost_function.gradient_hessian function is called. However, by default there is no guarantee that when the minimization has completed this function will have been called with the final version of the state vector. Calling the member function here requests that after minimization is complete, the derivatives of at least the specified order are consistent with the final state vector (e.g. 2 for both the Hessian and the gradient vector).

void set_levenberg_damping_limits(Real damp_min, Real damp_max) Set the minimum and maximum positive values of the damping parameter $\lambda$ used by both the Levenberg and Levenberg-Marquardt algorithms (default 1/128 and 10^5). Internally the algorithm can still use zero when each iteration is reducing the cost function.

void set_levenberg_damping_start(Real damp_start) Set the initial damping factor for the Levenberg and Levenberg-Marquardt algorithms (default 0).

void set_levenberg_damping_restart(Real damp_restart) Set the value of the damping factor $\lambda$ in the Levenberg and Levenberg-Marquardt algorithms that is used when a value of $\lambda = 0$ does not result in the cost function being reduced (default 1/4).

void set_levenberg_damping_multiplier(Real damp_multiply, Real damp_divide) Set the multiplier and divider that will be used to scale the damping factor when an iteration does not and does reduce the cost function, respectively (default 2.0 and 5.0).

The following member functions return the minimizer algorithm that the Minimizer is currently configured to use:

MinimizerAlgorithm algorithm() Return the enumeration representing the minimization algorithm.

std::string algorithm_name() Return a string representing the minimization algorithm.

The following member functions extract information about the progress of the minimization after it has completed:

int n_iterations() Return the number of iterations performed. Only iterations that successfully reduced the cost function are counted.

int n_samples() Return the number of times the cost function was computed, including times when this did not reduce the cost function.

Real cost_function() Return the final value of the cost function.

Real gradient_norm() Return the final value of the norm of the $\partial J/\partial x$ vector.

Real start_cost_function() Return the cost function for the first guess of the state vector provided by the user.

MinimizerStatus status() Return the convergence status.
4.4 Return status for minimization

The following enumerations may be returned by Minimizer’s `minimize` member function representing the status of the minimization. The `adept::minimizer_status_string` function converts a status to a user-readable string, as demonstrated in one of the examples in section 4.2.

MINIMIZER_STATUS_SUCCESS Minimization was successful.

MINIMIZER_STATUS_EMPTY_STATE The state vector provided is empty.

MINIMIZER_STATUS_MAX_ITERATIONS_REACHED The maximum number of iterations was reached.

MINIMIZER_STATUS_FAILED_TO_CONVERGE Convergence was not achieved, even though some progress may have been made in minimizing the cost function. This usually occurs when, in the vicinity of the minimum, the $J(x)$ terrain is quite flat and numerical errors mean that the gradient returned from the user-supplied function does not point uphill as it should. This means that when the algorithm uses the gradient to try to go downhill it finds the cost function increasing.

MINIMIZER_STATUS_INVALID_COST_FUNCTION The cost function returned is NaN or infinity. This is usually solved by using bounded minimization to ensure that the values of $x$ are kept within physically reasonable bounds.

MINIMIZER_STATUS_INVALID_GRADIENT The gradient vector returned contains NaN or infinity values. Use bounded minimization.

MINIMIZER_STATUS_INVALID_BOUNDS The bounds requested are not valid, for instance a maximum bound was requested that is less than the minimum bound.
Chapter 5

General considerations

5.1 Setting and checking the global configuration

The following non-member functions are provided in the adept namespace:

\begin{verbatim}
std::string version()  // Returns a string containing the version number of the Adept library (e.g. "2.0.8").
std::string compiler_version()  // Returns a string containing the compiler name and version used to compile the Adept library.
std::string compiler_flags()  // Returns a string containing the compiler flags used when compiling the Adept library.
std::string configuration()  // Returns a multi-line string listing numerous aspects of the way Adept has been configured.
bool have_matrix_multiplication()  // Returns true if the Adept library has been compiled with BLAS support, false otherwise.
bool have_linear_algebra()  // Returns true if the Adept library has been compiled with LAPACK support, false otherwise.
int set_max_blas_threads(int n)  // Set the maximum number of threads used for matrix operations by the BLAS library, or zero to use the upper limit on your system. The number returned is the number actually used.
int max_blas_threads()  // Return the maximum number of threads available for matrix operations by the BLAS library.
\end{verbatim}

The preprocessor can detect the Adept version at run-time via the ADEPT_VERSION preprocessor variable, which is an integer variable with the digits \texttt{abbcc} corresponding to Adept version \texttt{a.bb.cc}. This could be used to activate a different compile path dependent on the version, or even to fail to compile if the version is not recent enough:

\begin{verbatim}
#if ADEPT_VERSION < 10910
#error "Adept >= 1.9.10 is required by this program"
#endif
\end{verbatim}

5.2 Parallelizing Adept programs

Adept currently has limited built-in support for parallelization. If the algorithms that you wish to differentiate are individually small enough to be treated by a single processor core, and you wish to differentiate multiple algorithms independently (or the same algorithm but with multiple sets of inputs) then parallelization is straightforward. This is because the global variable containing a pointer to the Adept stack uses thread-local storage. This means that if a
5.3. The fast exponential function

Adept was originally developed for algorithms that make frequent calls to the exponential function exp, but unfortunately most compilers do not vectorize exp. Therefore, adept provides the function fastexp in the adept namespace, which can operate on active and passive scalars and array arguments (including the simple float and double) just like exp. It uses an adapted form of an algorithm from Agner Fog’s Vector Class Library (VCL) that is around a third faster for scalar arguments, but can be vectorized making it as much as 10 times faster when applied to Adept arrays depending on the instruction set available. It is accurate but not bit-reproducible with exp and produces finite results for a slightly smaller range of input values: from $-708.39$ to $+709.70$ for double arguments.

If you have an existing code that calls exp with Adept types as arguments, and wish to use the faster algorithm for all of them, then simply compile your code with -DADEPT_FAST_EXPONENTIAL. This will not change the behaviour of exp for other types of arguments, which would typically use the version from the C++ standard library. If you compile your code with -DADEPT_FAST_SCALAR_EXPONENTIAL then a fast exponential function adept::exp will be defined that works on arguments of type float and double. However, this can cause a namespace clash as some C header files import exp from the standard library outside of any namespace.

5.4 Tips for the best performance

- If you are working with single-threaded code, or in a multi-threaded program but with only one thread using a Stack object, then you can get slightly faster code by compiling all of your code with -DADEPT_STACK_THREAD_UNSAFE. This uses a standard (i.e. non-thread-local) global variable to point to the currently active stack object, which is slightly faster to access.
• If you compile with the \texttt{-g} option to store debugging symbols, your object files and executable will be much larger because every mathematical statement in the file will have the name of its associated templated type stored in the file, and these names can be long. Once you have debugged your code, you may wish to omit debugging symbols from production versions of the executable, or reduce the level of detail with \texttt{-g1} (on the GNU C++ compiler). There is typically no performance penalty associated with including debugging symbols.

• A high compiler optimization setting is recommended to inline the function calls associated with mathematical expressions. On the GNU C++ compiler, the \texttt{-O3 -march=native} setting is recommended.

• As outlined in the previous section, if you use the \texttt{exp} function then you can replace them with the faster \texttt{fastexp} function in or compile your code with \texttt{-DADEPT\_FAST\_EXPONENTIAL}.

• On Intel and ARM architectures, \texttt{Adept} will use the SSE2, AVX, AVX512 or NEON instruction sets (depending on availability) to vectorize array expressions that satisfy a number of requirements: (1) they contain only elementary mathematical operators (including the functions \texttt{sqrt, max, min and fastexp}), (2) the arrays in the expression are either all of type \texttt{float} or all of type \texttt{double}, (3) all the arrays in the expression must have their final dimension increasing in memory with no stride, and (4) none of the arrays are active. On the GNU compiler the \texttt{-march=native} selects the best available instruction set, but you can select a specific set with \texttt{-msse2, -mavx or -mavx512f}. With the SSE2 and NEON instruction sets, 2 doubles or 4 \texttt{floats} are operated on at once, for AVX these rise to 4 and 8 respectively, and for AVX512 they rise to 8 and 16 respectively.

• By default the Jacobian functions are compiled to process a strip of rows or columns of the Jacobian matrix at once. The optimum width of the strip depends on your platform, and you may wish to change it. To make the Jacobian functions process \textit{n} rows or columns at once, recompile the \texttt{Adept} library with \texttt{-DADEPT\_MULTIPASS\_SIZE=n}.

• If you suspect memory usage is a problem, you may investigate the memory used by \texttt{Adept} by simply sending your \texttt{Stack} object to a stream, e.g. \texttt{"std::cout \ll stack"}. You may also use the \texttt{memory()} member function, which returns the total number of bytes used. Further details of similar functions is given in section \ref{2.8}.

\section{5.5 Exceptions thrown by the \texttt{Adept} library}

Some functions in the \texttt{Adept} library can throw exceptions, and the exceptions that can be thrown are typically derived from either \texttt{adept::autodiff\_exception} or \texttt{adept::array\_exception}. These classes are derived from \texttt{adept::exception}, which is itself derived from \texttt{std::exception}. Most indicate an error in the users code, usually associated with calling \texttt{Adept} functions in the wrong order.

An overly comprehensive exception-catching implementation that takes different actions depending on whether a specific \texttt{Adept} exception, an exception related to automatic differentiation, a general \texttt{Adept} exception, or a non-\texttt{Adept} exception is thrown, could have the following form:

\begin{verbatim}
try {
    adept::Stack stack;
    // ... Code using the Adept library goes here ...
}

catch (adept::stack\_already\_active& e) {
    // Catch a specific Adept exception
    std::cerr \ll "Error: " \ll e.what() \ll std::endl;
    // ... any further actions go here ...
}

catch (adept::autodiff\_exception& e) {
    // Catch any Adept exception related to automatic differentiation not yet caught
    std::cerr \ll "Error: " \ll e.what() \ll std::endl;
    // ... any further actions go here ...
}
\end{verbatim}
catch (adept::exception& e) {
    // Catch any other Adept exception not yet caught
    std::cerr << "Error: " << e.what() << std::endl;
    // ... any further actions go here ...
}
}  
catch (...) {  
    // Catch any exceptions not yet caught
    std::cerr << "An error occurred" << std::endl;
    // ... any further actions go here ...
}

All exceptions implement the what() member function, which returns a const char* containing an error message.

5.5.1 General exceptions

The following exceptions are not specific to arrays or automatic differentiation and inherit directly from adept::exception::

**feature_not_available** This exception is thrown by deprecated functions, such as Stack::start(). It is also thrown by functions that are not available because a certain library is not being used, such as inv if Adept was compiled without LAPACK support, or matrix multiplication via the ‘*’ pseudo-operator if Adept was compiled without BLAS support.

5.5.2 Automatic-differentiation exceptions

The following exceptions relate to automatic differentiation (the functionality described in chapter 2), and all are in the adept namespace:

**gradient_out_of_range** This exception can be thrown by the adouble::get_gradient member function if the index to its gradient is larger than the number of gradients stored. This can happen if the adouble object was created after the first adouble::set_gradient call since the last Stack::new_recording call. The first adouble::set_gradient call signals to the Adept stack that the main algorithm has completed and so memory can be allocated to store the gradients ready for a forward or reverse pass through the differential statements. If further adouble objects are created then they may have a gradient index that is out of range of the memory allocated.

**gradients_not_initialized** This exception can be thrown by functions that require the list of working gradients to have been initialized (particularly the functions Stack::compute_tangent_linear and Stack::compute_adjoint). This initialization occurs when adouble::set_gradient is called.

**stack_already_active** This exception is thrown when an attempt is made to make a particular Stack object "active", but there already is an active stack in this thread. This can be thrown by the Stack constructor or the Stack::activate member function.

**dependents_or_independents_not_identified** This exception is thrown when an attempt is made to compute a Jacobian but the independents and/or dependents have not been identified.

**wrong_gradient** This exception is thrown by the adouble::append_derivative_dependence if the adouble object that it is called from is not the same as that of the most recent adouble::add_derivative_dependence.

**non_finite_gradient** This exception is thrown if the users code is compiled with the preprocessor variable ADEPT_TRACK_NONFINITE_GRADIENTS defined, and a mathematical operation is carried out for which the derivative is not finite. This is useful to locate the source of non-finite derivatives coming out of an algorithm.
5.5.3 Array exceptions

The following exceptions relate to arrays (the functionality described in chapter 3), and all are in the adept namespace:

**size_mismatch** A mathematical operation taking two arguments has been applied to array expressions that are not of the same size. The same exception is thrown if an array expression is applied to an array of a different size.

**inner_dimension_mismatch** Matrix multiplication has been attempted with arrays whose inner dimensions don’t agree.

**empty_array** An empty array has been used in an operation when a non-empty array is required; for example, if an attempt is made to link an array to an empty array (see section 3.2 for more information on linking).

**invalid_dimension** Attempt to create an array with a negative dimension.

**index_out_of_bounds** An element or range of elements has been requested from an array but one of the indices provided is out of range; for a dimension of length \( n \), the index is not in the range 0 to \( n - 1 \). Note that bounds checking is only applied if the preprocessor variable ADEPT_BOUNDS_CHECKING is defined.

**invalid_operation** An invalid operation has been performed that can only be detected at run-time, for example, calling the diag_submatrix member function of a non-square rank-2 Array.

**matrix_ill_conditioned** An attempt has been made to factorize an ill-conditioned matrix (either via solve or inv).

**fortran interoperability_error** An attempt has been made to associate an Adept Array with a FortranArray of the wrong rank or type.

5.6 Configuring the behaviour of Adept

The behaviour of the Adept library can be changed by defining one or more of the Adept preprocessor variables. This can be done either by editing the adept/base.h file and uncommenting the relevant #define lines, or by compiling your code with -Dxxx compiler options (replacing xxx by the relevant preprocessor variable. There are two types of preprocessor variable: the first types only apply to the compilation of user code, while the second types require the Adept library to be recompiled.

5.6.1 Modifications not requiring a library recompile

The preprocessor variables that apply only to user code and do not require the Adept library to be recompiled are as follows:

**ADEPT_STACK_THREAD_UNSAFE** If this variable is defined, the currently active stack is stored as a global variable but is not defined to be “thread-local”. This is slightly faster, but means that you cannot use multi-threaded code with separate threads holding their own active Stack object. Note that although defining this variable does not require a library recompile, all source files that make up a single executable must be compiled with this option (or all not be).

**ADEPT_RECORDING_PAUSABLE** This option enables an algorithm to be run both with and without automatic differentiation from within the same program via the functions Stack::pause_recording() and Stack::continue_recording(). Note that although defining this variable does not require a library recompile, all source files that make up a single executable must be compiled with this option (or all not be). Further details on this option are provided in section 2.6.2.
5.6. Configuring the behaviour of Adept

**ADEPT_NO_AUTOMATIC_DIFFERENTIATION** This option turns off automatic differentiation by treating `double` objects as `float`. It is useful if you want to compile one source file twice to produce versions with and without automatic differentiation. Further details on this option are provided in section 2.6.3.

**ADEPT_TRACK_NON_FINITE_GRADIENTS** Often when an algorithm is first converted to use an operator-overloading automatic differentiation library, the gradients come out as Not-a-Number or Infinity. The reason is often that the algorithm contains operations for which the derivative is not finite (e.g. $\sqrt{a}$ for $a = 0$), or constructions where a non-finite value is produced but subsequently made finite (e.g. `exp(-1.0/a)` for $a = 0$). Usually the algorithm can be recoded to avoid these problems, if the location of the problematic operations can be identified. By defining this preprocessor variable, a `non_finite_gradient` exception will be thrown if any operation results in a non-finite derivative. Running the program within a debugger (and ensuring that the exception is not caught within the program) enables the offending line to be identified.

**ADEPT_INITIAL_STACK_LENGTH** This preprocessor variable is set to an integer, and is used as the default initial amount of memory allocated for the recording, in terms of the number of statements and operations.

**ADEPT_REMOVE_NULL_STATEMENTS** If many variables in your code are likely to be zero then redundant operations will be added to the list of differential statements. For example, the assignment $a = b \times c$ with active variables $b$ and $c$ both being zero results in the differential statement $\delta a = 0 \times \delta b + 0 \times \delta c$. This preprocessor variable checks for zeros and removes terms on the right-hand-side of differential statements if it finds them. In this case it would put $\delta a = 0$ on the stack instead. This option slows down the recording stage, but speeds up the subsequent use of the recorded stack for adjoint and Jacobian calculations. The speed up of the latter is only likely to exceed the slow down of the former if your code contains many zeros. For most codes, this option causes a net slow down.

**ADEPT_COPY_CONSTRUCTOR_ONLY_ON_RETURN_FROM_FUNCTION** In Adept 1.1 this enabled a small but unsafe optimization. It now has no effect.

**ADEPT_BOUNDS_CHECKING** If this variable is defined, check that all array indices are within the bounds of the array throwing an `index_out_of_bounds` exception if necessary. If this variable is not defined then these checks are not performed, which is faster but means that attempts to access arrays out of bounds will result either of corruption of other memory used by the process, or a segmentation fault.

**ADEPT_NO_ALIAS_CHECKING** This variable turns off alias checking, which results in faster code, but may lead to unexpected results if the right-hand-side of an array statement shares data with the left-hand-side of the expression. If this is likely for a particular statement then use the `eval` function, described in section 3.13.

**ADEPT_NO_DIMENSION_CHECKING** This variable turns off checking the dimensions match when an array expression is assigned to another array.

**ADEPT_STORAGE_THREAD_SAFE** This variable ensures that accesses to the reference counter in `Storage` objects are atomic, enabling the `Array` and `SpecialMatrix` objects that use them to be accessed safely in a multi-threaded environment. Note that this may incur a performance penalty, and is only available in C++11. See section 3.15.

**ADEPT_INIT_REAL_SNAN** To detect errors caused by use of uninitialized data, initialize floating point arrays and active scalars with signaling NaNs. This is typically accompanied by directing the program to fail with a floating-point exception if a NaN is used in an expression, achieved by adding the following to one of the program source files:

```c
#include <fenv.h>
int _feenableexcept_status = feenableexcept(FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);
```

It should then be possible to use a debugger to identify the array that was read before being initialized with real numbers.
5.6.2 Modifications requiring a library recompile

The preprocessor variables that require the Adept library to be recompiled are as follows. Note that if these variables are used they must be the same when compiling both the library and the user code. This is safest to implement by editing section 2 of the adept/base.h header file.

**ADEPT_REAL_TYPE_SIZE** If you want to compile Adept to use a precision other than double for the Real type, and hence for automatic differentiation, then define this preprocessor variable to be 4 (for float), 8 (for double) or 16 (for long double). This will also change the default floating-point type for arrays, including shortcuts such as Vector, Matrix, SymmMatrix. Note that if you specify 16 but your compiler cannot support it (i.e. sizeof(long double)==8) then Adept would produce sub-optimal code so will fail to compile.

**ADEPT_STACK_STORAGE_STL** Use the C++ standard template library vector or valarray classes for storing the recording and the list of gradients, rather than dynamically allocated arrays. In practice, this tends to slow down the code.

**ADEPT_MULTIPASS_SIZE** This is set to an integer, invariably a power of two, specifying the number of rows or columns of a Jacobian that are calculated at once. The optimum value depends on the platform and the capability of the compiler to optimize loops whose length is known at compile time.

**ADEPT_MULTIPASS_SIZE_ZERO_CHECK** This is also set to an integer; if it is greater than ADEPT_MULTIPASS_SIZE, then the Stack::jacobian_reverse function checks gradients are non-zero before using them in a multiplication.

**ADEPT_THREAD_LOCAL** This can be used to specify the way that thread-local storage is declared by your compiler. Thread-local storage is used to ensure that the Adept library is thread-safe. By default this variable is not defined initially, and then later in adept/base.h it is set to an appropriate value on your system: thread_local if you compile with the C++11 standard, otherwise _declspec(thread) on Microsoft Visual C++, an empty declaration on Mac (since thread-local storage is not available on many Mac platforms) and _thread otherwise (appropriate for at least the GNU, Intel, Sun and IBM compilers). To override the default behaviour, define this variable yourself in adept/base.h.

5.7 Frequently asked questions

**Why are all the gradients coming out of the automatic differentiation zero?** You have almost certainly omitted or misplaced the call of the adept::Stack member function “new_recording()”. It should be placed after the independent variables in the algorithm have been initialized, but before any subsequent calculations are performed on these variables. If it is omitted or placed before the point where the independent variables are initialized, the differential statements corresponding to this initialization (which are all of the form \( \delta x = 0 \)), will be placed in the list of differential statements and will unhelpfully set to zero all your gradients right at the start of a forward pass (resulting from a call to forward()) or set them to zero right at the end of a reverse pass (resulting from a call to reverse()).

**Why are the gradients coming out of the automatic differentiation NaN or Inf (even though the value is correct)?** This can occur if the algorithm contains operations for which the derivative is not finite (e.g. \( \sqrt{a} \) for \( a = 0 \)), or constructions where a non-finite value is produced but subsequently made finite (e.g. \( \exp(-1.0/a) \) for \( a = 0 \)). Usually the algorithm can be recoded to avoid these problems, if the location of the problematic operations can be identified. The simplest way to locate the offending statement is to recompile your code with the -g option and the ADEPT_TRACK_NON_FINITE_GRADIENTS preprocessor variable set (see section 5.6.1). Run the program within a debugger and a non_finite_gradient exception will be thrown, which if not caught within the program will enable you to locate the line in your code where the problem originated. You may need to turn optimizations off (compile with -O0) for the line identification to be accurate. You can also turn on trapping of floating-point exceptions, as explained in the description of the ADEPT_INIT_REAL_SNAN option in section 5.6.1.
Why are the gradients coming out of the automatic differentiation wrong? Before suspecting a bug in Adept, note that round-off error can lead to incorrect gradients even in hand-coded differential code. Consider the following:

```c
int main() {
    Stack stack;
    adouble a = 1.0e-26, b;
    stack.new_recording();
    b = sin(a) / a;
    b.set_gradient(1.0);
    stack.compute_adjoint();
    std::cout << "a=" << a << " b=" << b << " db/da=" << a.get_gradient() << "\n";
}
```

We know that near \( a=0 \) we should have \( b=1 \) and the gradient should be 0. But running the program above will give a gradient of \( 1.71799e+10 \). If you hand-code the gradient, i.e.

```c
double A = 1.0e-26;
double dB_dA = cos(A)/A - sin(A) / (A*A);
```

you will also get the wrong gradient. You can see that the answer is the difference of two very large numbers and so subject to round-off error. This example is therefore not a bug of Adept, but a limitation of finite-precision machines. To check this, try compiling your code using either the ADOL-C or CppAD automatic differentiation tools; I have always found these tools to give exactly the same gradient as Adept. Unfortunately, round-off error can build up over many operations to give the wrong result, so there may not be a simple solution in your case.

Can Adept reuse a stored tape for multiple runs of the same algorithm but with different inputs? No. Adept does not store the full algorithm in its stack (as ADOL-C does in its tapes, for example), only the derivative information. So from the stack alone you cannot rerun the function with different inputs. However, rerunning the algorithm including recomputing the derivative information is fast using Adept, and is still faster than libraries that store enough information in their tapes to enable a tape to be reused with different inputs. It should be stressed that for any algorithm that includes different paths of execution (“if” statements) based on the values of the inputs, such a tape would need to be rerecorded anyway. This includes any algorithm containing a look-up table.

Why does my code crash with a segmentation fault? This means it is trying to access a memory address not belonging to your program, and the first thing to do is to run your program in a debugger to find out at what point in your code this occurs. If it is in the `adept::aReal` constructor (note that `aReal` is synonymous with `adouble`), then it is very likely that you have tried to initiate an `adept::adouble` object before initiating an `adept::Stack` object. As described in section 2.3.1, there are good reasons why you need to initialize the `adept::Stack` object first.

How can I interface Adept with a matrix library such as Eigen? Unfortunately the use of expression templates in Adept means that it does not work optimally (if it works at all) with third-party matrix libraries that use expression templates. This is the reason why Adept 2.0 combines array functionality with automatic differentiation in a single expression-template framework.

Do you have plans to enable Adept to produce Hessian matrices? Not in the near future as this is a huge change. However, if your cost function \( J(x) \) (also known as a cost function or penalty function) has a specific form then the Hessian matrix can be computed from the Jacobian matrix, as described in chapter 4.

Why doesn’t the ternary operator work? Some compilers will fail to compile the following function:

```c
adept::adouble piecewise(adept::adouble x) {
    return x < 1.0 ? x*x : 2.0*x-1.0;
}
```
The reason is that these compilers require that the two possible outcomes of the ternary operator have the same type, but due to the use of expression templates, the types of these mathematical expressions are actually different. The ternary operator cannot be overloaded to allow such arguments. The solution is to explicitly convert the outcomes to `adouble`:

```cpp
adep::adouble piecewise(adep::adouble x) {
    return x < 1.0 ? adept::adouble(x*x) : adept::adouble(2.0*x-1.0);
}
```

**Why is my executable so huge?** Probably you are including debugging symbols by compiling with the `-g` option. Expression templates need long strings to describe them, so this extra content can increase the size of object files and executables by a factor of ten. This does not slow down execution, but for production code you may wish to compile without debugging symbols, or if you use the GNU compiler use instead the `-g1` option which stores a reduced amount of debugging information.

**Why do I get incorrect behaviour when I use the “auto” keyword?** Since C++11, many programmers make widespread use of `auto` as the type of a local object that can be inferred from its initializer. This is problematic for most expression-template libraries, including Adept, because expressions are not evaluated immediately. For example, dividing one `aReal` object by another returns an `adep::internal::BinaryOperation<Real,aReal,Divide,aReal>` object, and the division is only performed (and differentiated) when this object is assigned to an `aReal` object. The `auto` keyword will be interpreted as the type of the internal object, but this internal object may contain references to temporary objects that make up the other parts of the expression, and which go out of scope after the semi-colon at the end of the `auto` statement. In this example the correct behaviour is obtained by replacing `auto` with `aReal`. Never to use the `auto` keyword when initializing an object from an Adept expression.

### 5.8 Copyright and license for Adept software

Versions 1.9 of Adept and later are owned and copyrighted jointly by the University of Reading and the European Centre for Medium Range Weather Forecasts. The copyright to versions 1.1 and earlier is held solely by the University of Reading.

Since version 1.1, the Adept library is released under the Apache License, Version 2.0, which is available at [http://www.apache.org/licenses/LICENSE-2.0](http://www.apache.org/licenses/LICENSE-2.0). In short, this free-software license permits you to use the library for any purpose, and to modify it and combine it with other software to form a larger work. If you choose, you may release the modified software in either source code or object code form, so may use Adept in both open-source software and non-free proprietary software. However, distributed versions must retain copyright notices and also distribute both the information in the NOTICES file and a copy of the Apache License. Different license terms may be applied to your distributed software, although they must include the conditions on redistribution provided in the Apache License. This is a just short summary; if in doubt, consult the text of the license.

In addition to the legally binding terms of the license, it is requested that:

- You cite Hogan (2014) in publications describing algorithms and software that make use of the Adept library. While not a condition of the license, this is good honest practice in science and engineering.

- If you make modifications to the Adept library that might be useful to others, you release your modifications under the terms of the Apache License, Version 2.0, so that they are available to others and could also be merged into a future official version of Adept. If you do not state the license applied to your modifications then by default they will be under the terms of the Apache License. You will retain copyright of your modifications, but if your modifications are written in the course of employment then under almost all circumstances (including employment by a University) it is your employer who holds the copyright. Therefore you should obtain permission from them to release your modifications under the Apache License.
Note that other source files in the Adept package used for demonstrating and benchmarking Adept are released under the GNU all-permissive license*, which is specified at the top of all files it applies to.

Adept version 1.0 was released under the terms of the GNU General Public License (GPL) and so could not be released as part of a larger work unless the entire work was released under the conditions of the GPL. It is hoped that the switch to the Apache License will facilitate wider use of Adept.

Acknowledgments

Adept 1.0 was developed by Robin Hogan at the University of Reading with funding from European Space Agency contract 40001041528/11/NL/CT. Some of the modifications to produce version 1.1 were funded by a National Centre for Earth Observation Mission Support grant (Natural Environment Research Council grant NE/H003894/1). Dr Brian Tse is thanked for his work exploring different parallelization strategies during this period. Subsequent development has been carried out under employment at the European Centre for Medium Range Weather Forecasts.

*The GNU all-permissive license reads: Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved. This file is offered as-is, without any warranty.


