Adept automatic differentiation library for C++: User guide

Robin J. Hogan, University of Reading, UK

Document version 1.1 (May 2015) applicable to Adept version 1.1

Contents

If you have any queries about *Adept* that are not answered by this document or by the information on the *Adept* web site (http://www.met.reading.ac.uk/clouds/adept/) then please email me at r.j.hogan@reading.ac.uk.

1   Introduction

Adept (Automatic Differentiation using Expression Templates) is a software library that enables algorithms written in C and C++ to be automatically differentiated. It 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).

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 2 describes the functionality that the library provides. Section 3 outlines how to install it on your system and how to compile your own code to use it. Section 4 describes how to prepare your code for automatic differentiation, and section 5 describes how to perform forward- and reverse-mode automatic differentiation on this code. Section 6 describes how to compute Jacobian matrices. Section 7 provides a detailed description of how to interface an algorithm implemented using *Adept* with a third-party minimization library. Section 8 describes how to call a function both with and without automatic differentiation from within the same program. Section 9 describes how to interface to software modules that compute their own Jacobians. Section 11 provides some tips for getting the best performance from *Adept*. Section 12 describes the user-oriented member functions of the `Stack` class that contains the differential information and section 13 describes the member functions of the "active" double-precision type `adouble`. Section 14 describes the exceptions that can be thrown by some *Adept* functions. Section 15 describes how to configure the behaviour of *Adept* by defining certain preprocessor variables, and section 16 answers some frequently asked questions. Finally, section 17 describes the license terms.

2   What functionality does *Adept* provide?

Adept provides the following functionality:

Full Jacobian matrix   Given the non-linear function $\mathbf{y} = f(\mathbf{x})$ relating vector $\mathbf{y}$ to vector $\mathbf{x}$ coded in C or C++, after a little code modification *Adept* can compute the Jacobian matrix $\mathbf{H} = \partial\mathbf{y}/\mathbf{x}$, where the element at row i and column j of $\mathbf{H}$ is $H_{i,j} = \partial y_i/\partial x_j$. 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 $\mathbf{x}$ 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 $\mathbf{x}$ 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 nonlinear function $J(\mathbf{x})$ relating the scalar to be minimized J to vector $\mathbf{x}$, *Adept* will compute the vector of adjoints $\partial J/\partial\mathbf{x}$. Moreover, for a component of the code that may be expressed as a multi-dimensional non-linear function $\mathbf{y} = f(\mathbf{x})$, *Adept* can compute $\partial J/\partial\mathbf{x}$ if it is provided with the vector of input adjoints $\partial J/\partial\mathbf{y}$. In this case, $\partial J/\partial\mathbf{x}$ is equal to the matrix-vector product $\mathbf{H}^\mathrm{T}\partial J/\partial\mathbf{y}$, but it is computed here without computing the full Jacobian matrix $\mathbf{H}$. The vector $\partial J/\partial\mathbf{x}$ may then be used in a quasi-Newton minimization scheme (e.g., Liu and Nocedal, 1989).

Forward-mode differentiation   Given the non-linear function $\mathbf{y} = f(\mathbf{x})$ and a vector of perturbations $\delta\mathbf{x}$, *Adept* will compute the corresponding vector $\delta\mathbf{y}$ arising from a linearization of the function f. Formally, $\delta\mathbf{y}$ is equal to the matrix-vector product $\mathbf{H}\delta\mathbf{x}$, but it is computed here without computing the full Jacobian matrix $\mathbf{H}$. 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, 2013, showed that it was competitive).

Adept can currently automatically differentiate the standard mathematical operators +, −, * and /, as well as their assignment versions +=, −=, *= and /=. It supports the mathematical functions `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `abs` and `pow`, and since version 1.1 also the functions `asinh`, `acosh`, `atanh`, `expm1`, `log1p`, `cbrt`, `erf`, `erfc`, `exp2` and `log2`. The "active" 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.

- 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.

- All code to be differentiated in a single program must use the same precision. By default this is double precision, although the library may be recompiled to use single precision or quadruple precision (the latter only if supported by your compiler).

- Your code should operate on variables individually: they can be stored in C-style arrays or `std::vector` types, but if you use containers that allow operations on entire arrays, such as the `std::valarray` type, then some array-wise functionality (such as mathematical functions applied to the whole array and multiplying an array of active variables by an ordinary non-active scalar) will not work.

- 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).

3   Installing *Adept* and compiling your code to use it

Adept should be compatible with any C++98 compliant compiler, although most of the testing has been specifically on Linux with the GNU C++ 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 3.2.

3.1   Unix-like platforms

On a Unix-like system, do the following:

1. Unpack the package (`tar xvfz adept-1.1.tar.gz` on Linux) and `cd` to the directory `adept-1.1`.

2. Configure the build using the `configure` script (the use of a `configure` script generated by `autoconf` was introduced in *Adept* version 1.1). The most basic method is to just run

   ```
   ./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 `/opt/local`. These may be done by running instead

   ```
   ./configure "CXXFLAGS=-g -O3" --prefix=/opt/local
   ```

The `-g` option to `CXXFLAGS` ensures debugging information is stored. You can see the options available by running `./configure --help`. You can turn-off OpenMP parallelization in the computation of Jacobian matrices using `--disable-openmp`. After compiling the library you may optionally compile a test program that interfaces to the GNU Scientific Library, or a benchmarking program that compares the performance to other automatic differentiation tools ADOL-C, CppAD and Sacado. If these libraries are in a non-standard directory then further environment variables need to be specified in the same way as the example above. For example, you can specify an additional directory to search for header files with something like `CPPFLAGS=-I/opt/local/include` or an additional directory to search for static and shared libraries with something like `LDFLAGS=-L/opt/local/lib`. See also section 15 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.

3. Build *Adept* by running

```
make
```

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

4. Install the header file `include/adept.h` and the static and shared libraries by running

```
make install
```

If this is to be installed to a system directory, you will need to log in as the super-user first.

5. Build the example and benchmarking 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_adept`: 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.

In the `benchmark` directory it compiles `autodiff_benchmark` 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. Note that despite the implication, "`make check`" does not automatically run any of the programs it makes to check they function correctly.

To compile source files that use the *Adept* library, you need to make sure that `adept.h` is in your include path. If this file is 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` before the `-ladept` argument to specify its location. Section 8.3 provdes 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.

3.2 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 a fairly small library amounting to only around 3,400 lines of code, of which around 2,300 are in the `adept.h` header file and around 1,100 in the library source files. Therefore 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:

```
#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 the `adept.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. A demonstration of this 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 simply copy `adept.h` and `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 17 for details.

4 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 5 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. If you wish to use a different precision, or 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`. Section 15 describes how to configure these types to represent single, double or quadruple precision, but be aware that accumulation of round-off error can make the accuracy of derivatives computed using single precision insufficient for minimization algorithms. For now we consider only double precision.

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

```
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. It is hoped that a future version of *Adept* will include its own complex and vector types that overcome these limitations.

5   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:

```
#include "adept.h"
double algorithm_and_gradient(
                  const double x_val[2], // Input values
                  double dy_dx[2]) {     // Output gradients
  adept::Stack stack,                    // Where the derivative information is stored
  using adept::adouble;                  // Import adouble from adept
  adouble 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 objective 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.

5.1   Set-up stack to record derivative information

```
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, because such an instantiation registers the `adouble` object with the currently active stack. Otherwise the code will crash with a segmentation fault.

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 reallocated 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` in the main body of the program and pass a reference to it to the `algorithm_and_gradient` function, so that it does not go out of scope between calls.

- Put it in a class so that it is accessible to member functions; this approach is demonstrated in section 7.

5.2 Initialize independent variables and start recording

```
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, δ`x[0]`=0 and δ`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

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

or

```
adept::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.

5.3 Perform calculations to be differentiated

```
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.

5.4 Perform reverse-mode differentiation

```
y.set_gradient(1.0);
stack.compute_adjoint();
```

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 objective 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_ad using

```
adept::set_gradients(y, n, y_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.

5.5 Extract the final gradients

```
dy_dx[0] = x[0].get_gradient();
dy_dx[1] = x[1].get_gradient();
```

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

```
adept::get_gradients(x, 2, dy_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.

6 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

```
void algorithm2(int n, const adouble* x, int m, 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:

```
#include <vector>
#include "adept.h"
void algorithm2_jacobian(
                    int n,                  // Number of input values
                    const double* x_val,    // Input values
                    int m,                  // Number of output values
                    double* y_val,          // Output values
                    double* jac) {          // Output Jacobian matrix
  using adept::adouble;                     // Import Stack and adouble from adept
  adept::Stack stack;                       // Where the derivative information is stored
  std::vector<adouble> x(n);                // Vector of active input variables
  adept::set_values(&x[0], n, x_val);       // Initialize adouble inputs
  adept.new_recording();                    // Start recording
  std::vector<adouble> y(m);                // Create vector of active output variables
  algorithm2(n, &x[0], m, &y[0]);           // Run algorithm
  stack.independent(&x[0], n);              // Identify independent variables
  stack.dependent(&y[0], m);                // Identify dependent variables
  stack.jacobian(jac);                      // Compute & store Jacobian in jac
}
```

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×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.

7 Real-world usage: interfacing *Adept* to a minimization library

Suppose we want to find the vector $\mathbf{x}$ that minimizes an objective function $J(\mathbf{x})$ 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 4). 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 $\mathbf{x}$, and for each call may request not only the value of the function, but also its gradient with respect to $\mathbf{x}$. Thus the public interface provides `calc_function_value(const double*)` and `calc_function_value_and_gradient`, which could be implemented as follows:

```cpp
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 8 describes two ways to overcome this problem. The second function above implements reverse-mode automatic differentiation as described in section 5.

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

```cpp
#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;
  // 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:

```
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);
}

extern "C"
void my_function_gradient(const gsl_vector* x, void* params, gsl_vector* gradJ) {
  State* state = reinterpret_cast<State*>(params);
  state->calc_function_value_and_gradient(x->data, gradJ->data);
}

extern "C"
void my_function_value_and_gradient(const gsl_vector* x, void* params,
                                    double* J, gsl_vector* gradJ) {
  State* state = reinterpret_cast<State*>(params);
  *J = state->calc_function_value_and_gradient(x->data, gradJ->data);
}
```

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.

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

The `calc_function_value(const double*)` member function defined in section 7 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 trancendental functions) the difference between an algorithm coded in terms of `double`s and the same algorithm coded in terms of `adouble`s 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 `double`s and the other in terms of `adouble`s. The three ways to achieve this are now described.

8.1   Function templates

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

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

This takes the example from section 4 and replaces `adouble` by the template type `xdouble`. Thus, `calc_function_value` can be called with either `double` or `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.

8.2   Pausable recording

The second method involves compiling the entire code with the `ADEPT_RECORDING_PAUSABLE` preprocessor variable defined, which can be done by adding an argument `-DADEPT_RECORDING_PAUSABLE` to the compler command line. This modifies the behaviour of mathematical operations performed on `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 7:

```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 `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 `adouble` objects, generally adds a small overhead. However, in algorithms where most of the number crunching occurs in loops containing no trancendental functions, even if the stack is in the paused state, the presence of the check can prevent the compiler from agressively optimizing the loop. In that instance the third method may be preferable.

8.3   Multiple object files per source file

The third method involves compiling each source file containing functions with `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 `-DADEPT_NO_AUTOMATIC_DIFFERENTIATION` flag on the compiler command line. This instructs the `adept.h` header file to turn off automatic differentiation by defining the `adouble` type to be an alias of the `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 7, the `calc_function_value` function would be one that would be compiled twice in this way, once to provide the `calc_function_value(const adouble*)` version and the other to provide the `calc_function_value(const double*)` version. Note that any functions that do not include `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 (`algorithm1.cpp` and `algorithm2.cpp`) twice and a third (`main.cpp`) once:

```
# Specify compiler and flags
CPP = g++
CPPFLAGS = -Wall -O3 -g
# Normal object files to be created
OBJECTS = algorithm1.o algorithm2.o main.o
# Object files created with no automatic differentiation
NO_AD_OBJECTS = algorithm1_noad.o algorithm2_noad.o
# Program name
PROGRAM = my_program
# Include-file location
INCLUDES = -I/usr/local/include
# Library location and name, plus the math library
LIBS = -L/usr/local/lib -lm -ladept

# Rule to build the program (typing "make" will use this rule)
$(PROGRAM): $(OBJECTS) $(NO_AD_OBJECTS)
        $(CPP) $(CPPFLAGS) $(OBJECTS) $(NO_AD_OBJECTS) $(LIBS) -o $(PROGRAM)
# Rule to build a normal object file (used to compile all objects in OBJECTS)
%.o: %.cpp
        $(CPP) $(CPPFLAGS) $(INCLUDES) -c $<
# Rule to build a no-automatic-differentiation object (used to compile ones in NO_AD_OBJECTS)
%_noad.o: %.cpp
        $(CPP) $(CPPFLAGS) $(INCLUDES) -DADEPT_NO_AUTOMATIC_DIFFERENTIATION -c $< -o $@
```

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 7):

```
#include "adept.h"
class State {
 public:
    ...
    double calc_function_value(const double* x);
 protected:
```

```
#ifndef ADEPT_NO_AUTOMATIC_DIFFERENTIATION
   adouble 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.

9 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 4. 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:

```
void simulate_radiances(int n, // Size of temperature array
                        // Input variables:
                        double surface_temperature,
                        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*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 δI and δJ, to perturbations in the input variables, δT_s and δT_i (for all layers i). Mathematically, we wish the following relationship to be stored within the *Adept* stack:

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

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:

```
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 13, 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.

10   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 process spawns multiple threads (e.g. using OpenMP or Pthreads) then each thread can declare one `adept::Stack` object and all `adouble` operations will result in statements being stored on the stack object specific to that thread. The *Adept* package contains a test program `test_thread_safe` that demonstrates this approach in OpenMP.

If your problem is larger and you wish to use parallelism to speed-up the differentiation of a single large algorithm then the build-in support is more limited. Provided your program and the *Adept* library were compiled with OpenMP enabled (which is the default for the *Adept* library if your compiler supports OpenMP), the computation of Jacobian matrices will be parallelized. By default, the maximum number of concurrent threads will be equal to the number of available cores, but this can be overridden with the `set_max_jacobian_threads` member function of the `Stack` class. Note that the opportunity for speed-up depends on the size of your Jacobian matrix: for an $m \times n$ matrix, the number of independent passes through the stored data is $\min(m, n)$ and each thread treats `ADEPT_MULTIPASS_SIZE` of them (see section 15.2), so the maximum number of threads that can be exploited is $\min(m, n)/$`ADEPT_MULTIPASS_SIZE`. Again, the `test_thread_safe` program can demonstrate the parallelization of Jacobian calculations. Note, however, that if the `jacobian` function is called from within an OpenMP thread (e.g. if the program already uses OpenMP with each thread containing its own `adept::Stack` object), then the program is likely not to be able to spawn more threads to assist with the Jacobian calculation.

If you need Jacobian matrices then the ability to parallelize the calculation of them is useful since this tends to be more computationally costly than recording the original algorithm. If you only require the tangent-linear or adjoint calculations (equivalent to a Jacobian calculation with $n = 1$ or $m = 1$, respectively), then

unfortunately you are stuck with single threading. It is intended that a future version of *Adept* will enable all aspects of differentiating an algorithm to be parallelized with either or both of OpenMP and MPI.

11   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 -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. There appears to be no performance penalty associated with the debugging symbols, at least with the GNU C++ compiler.

- A high compiler optimization setting is recommended to inline the function calls associated with mathematical expressions. On the GNU C++ compiler, the -O3 setting is recommended.

- 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 *n* rows or columns at once, recompile the *Adept* library with -DADEPT_MULTIPASS_SIZE=*n*.

- If you suspect memory usage is a problem, you may investigate the memory used by *Adept* by simply sending your Stack object to a stream, e.g. "std::cout << stack". You may also use the memory() member function, which returns the total number of bytes used. Further details of similar functions is given in section 12.

12   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 (for example, if only one argument is supplied to the **jacobian** function below, then it will be executed as if called with a second argument **false**). Some of these functions throw *Adept* exceptions, defined in section 14.

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 8.2.

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 13) 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 he Jacobian, respectively. This function adds **x** to the list of independent variables. If it is the *n*th variable identified in this way, the *n*th 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 *m*th variable identified in this way, the *m*th 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 $m \times n$ values. The result is stored in column-major order, i.e., the *m* diemension 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 $n \leq m$ and **jacobian_reverse** if $n > m$.

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 $n \leq m$.

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 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.

std::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.)

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.

std::string version() Returns a string containing the version number of the *Adept* library (e.g. "1.1").

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.

13 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 14.

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 the 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(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 adouble& g) Add a differential statement to the currently active stack of the form $\delta\mathbf{l} = \mathbf{g} \times \delta\mathbf{r}$, 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 9; in this case **g** is the gradient $\partial\mathbf{l}/\partial\mathbf{r}$ obtained from such software.

void append_derivative_dependence(const adouble& r, const adouble& g) Assuming that the same **adouble** object has just had its **add_derivative_dependence** member function called, this function appends $+ \mathbf{g} \times \delta\mathbf{r}$ 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,
$\qquad\qquad\qquad\qquad$**size t n = 1, size t m stride = 1)**
$\qquad$ Add a differential statement to the currently active stack of the form $\delta \mathtt{l} = \sum_{i=0}^{\mathbf{n}-1} \mathtt{m[}i\mathtt{]} \times \delta \mathtt{r[}i\mathtt{]}$, where **l** is the **adouble** object from which this function is called. If the **g stride** argument is provided, then the index to the **g** array will be $i \times$ **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,
$\qquad\qquad\qquad\qquad$**size t n = 1, size t g stride = 1)**
$\qquad$ Assuming that the same **adouble** object has just called the **add derivative dependence** function, this function appends $+ \sum_{i=0}^{\mathbf{n}-1} \mathtt{m[}i\mathtt{]} \times \delta \mathtt{r[}i\mathtt{]}$ 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 8.3.

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.

14 Exceptions thrown by the *Adept* library

Some functions in the *Adept* library can throw exceptions, and all of the exceptions that can be thrown are derived from adept::autodiff_exception, which is itself derived from std::exception. All these exceptions indicate an error in the users code, usually associated with calling *Adept* functions in the wrong order.

$\qquad$ An exception-catching implementation that takes different actions depending on whether a specific *Adept* exception, a general *Adept* exception, or a non-*Adept* exception is thrown might have the following form:

```
try {
  adept::Stack stack;
  // ... Code using the Adept library goes here ...
}
catch (adept::stack_already_active& e) {
  // Catch a specific Adept exception
  std::cerr << "Error: " << e.what() << std::endl;
  // ... any further actions go here ...
}
catch (adept::autodiff_exception& e) {
  // Catch any 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. The following exceptions can be thrown, 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_NON_FINITE_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.

feature_not_available This exception is thrown by deprecated functions, such as `Stack::start()`.

15   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.h` file and uncommenting the relevant `#define` lines in sections 1 or 2 of the file, 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.

15.1   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 8.2.

ADEPT␣NO␣AUTOMATIC␣DIFFERENTIATION This option turns off automatic differentiation by treating `adouble` objects as `double`. 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 8.3.

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 If copy constructors for `adouble` objects are only used in the return values for functions, then defining this preprocessor variable will lead to slightly faster code, because it will be assumed that when a copy constructor is called, the index to its gradient can simply be copied because the object being copied will shortly be destructed (otherwise communication with the `Stack` object is required to unregister one and immediately register the other). You need to be sure that the code being compiled with this option does not invoke the copy constructor in any other circumstances. Specifically, it must not include either of these constructions: "`adouble x = y;`" or "`adouble x(y);`", where `y` is an `adouble` object. If it does, then strange errors will occur.

15.2 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.h` header file.

ADEPT␣FLOATING␣POINT␣TYPE If you want to compile *Adept* to use a precision other than double, define this preprocessor variable to be the floating-point type required, e.g. `float` or `long double`. To use from the compiler command-line, use the argument `-DADEPT_FLOATING_POINT_TYPE=float` or `-DADEPT_FLOATING_POINT_TYPE="long double"`.

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.h` it is set to `__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 GCC, Intel, Sun and IBM compilers). To override the default behaviour, define this variable yourself in section 2 of `adept.h`.

16   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 15.1). 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. Another approach is to add the following in a C++ source file:

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

This will cause a floating point exception to be thrown when a `NaN` or `Inf` is generated, which can again be located in a debugger.

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:

```cpp
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.

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

you will 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 numerical gradients. 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 5.1, 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 matrix libraries that use expression templates. However, I am working on a combined library that would combine array functionality with automatic differentiation, which will hopefully be released as version 2.0 of *Adept*.

Do you have plans to enable *Adept* to produce Hessian matrices? Not in the near future; adding array functionality is a higher priority at the moment. However, if your objective function $J(\mathbf{x})$ (also known as a cost function or penalty function) has a quadratic dependence on each of the elements of $\mathbf{y}(\mathbf{x})$, where $\mathbf{y}$ is a nonlinear function of the independent variables $\mathbf{x}$, then the Hessian matrix $\nabla_{\mathbf{x}}^2 J$ can be computed from the Jacobian matrix $\mathbf{H} = \partial \mathbf{y}/\partial \mathbf{x}$. This is the essence of the Gauss-Newton and Levenberg-Marquardt algorithms. Consider the optimization problem of finding the parameters $\mathbf{x}$ of nonlinear model $\mathbf{y}(\mathbf{x})$ that provides the closest match to a set of "observations" $\mathbf{y}^o$ in a least-squares sense. For maximum generality we add constraints that penalize differences between $\mathbf{x}$ and a set of *a priori* values $\mathbf{x}^a$, as well as a regularization term. In this case the objective function could be written as

$$J(\mathbf{x}) = [\mathbf{y}(\mathbf{x}) - \mathbf{y}^o]^{\mathrm{T}} \mathbf{R}^{-1} [\mathbf{y}(\mathbf{x}) - \mathbf{y}^o] + [\mathbf{x} - \mathbf{x}^a]^{\mathrm{T}} \mathbf{B}^{-1} [\mathbf{x} - \mathbf{x}^a] + \mathbf{x}^{\mathrm{T}} \mathbf{T} \mathbf{x}. \tag{2}$$

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

$$\nabla_{\mathbf{x}}^2 J = \mathbf{H}^{\mathrm{T}} \mathbf{R}^{-1} \mathbf{H} + \mathbf{B}^{-1} + \mathbf{T}, \tag{3}$$

which can be coded up using *Adept* to compute $\mathbf{H}$. Each term on the right-hand-side of (3) has its corresponding term in (2), so it is easy to work out what the Hessian would look like if only a subset of the terms in (2) were present.

17 License for *Adept* software

Version 1.1 of the *Adept* library is released under the Apache License, Version 2.0, which is available at `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 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 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*.

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*.

References

Bell, B., 2007: CppAD: A package for C++ algorithmic differentiation. `http://www.coin-or.org/CppAD`

Liu, D. C., and Nocedal, J., 1989: On the limited memory method for large scale optimization. *Math. Programming B,* **45,** 503–528.

Gay, D. M., 2005: Semiautomatic differentiation for efficient gradient computations. In *Automatic Differentiation: Applications, Theory, and Implementations*, H. M. Bücker, G. F. Corliss, P. Hovland, U. Naumann and B. Norris (eds.), Springer, 147–158.

Griewank, A., Juedes, D., and Utke, J., 1996: Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.,* **22,** 131–167.

Hogan, R. J., 2014: Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.,* **40,** 26:1-26:16.

[†]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.*