ReverseAD Library Documentation

Mu Wang
Jawad Ahmed Raheel (jraheel@purdue.edu)
Alex Pothen (apothen@purdue.edu)

Contents

1 Installation **2**

2 Preliminaries **3**

3 Declaring a Function **3**

4 Computing the Gradient of a Function **4**

5 Computing the Hessian **6**

6 Jacobian **7**

7 Hessian-Vector Product **8**

8 Higher Order Derivatives **10**

9 Reusing the Trace **11**

We assume the user is familiar with automatic differentiation. The C++ ReverseAD library was written by Mu Wang during his Ph.D. studies in the Computer Science department at Purdue University. It implements several methods to evaluate higher order derivate tensors in reverse (and other) modes using operator overloading for automatic differentiation. It is open source under GNU General Public License Version 3 and is available here.

1 Installation

Following are instructions to install reverseAD library on a Linux or MacOS system.
Obtain the reverseAD library here.
The following are the requirements for installation:
1. autoreconf
2. automake
3. libtool
4. a C++ complier

All of the above can be installed via apt-get on Linux, or homebrew on MacOS. Once these dependencies have been installed, the user should download the source code of ReverseAD from the git repository link above. Then please do the following:

Change directory to the location containing the source code of ReverseAD. We shall refer to this directory as $ReverseADHOME. Also the directory where the user wishes to install the compiled files (libraries, header files, etcl) as $install_location. This for example could be /usr/local/. Enter the following commands to install ReverseAD on your machine:

1. cd $ReverseADHOME
2. autoreconf -fi
3. ./configure −−prefix=$install_location
4. make
5. make install

2 Preliminaries

reversead.hpp is usually the only header file required.
`#include "reversead/reversead.hpp"`

The most widely used functions are written in the namespace ReverseAD.
`using namespace ReverseAD`

The libary implements the nonprimitive data type `adouble` (ReverseAD::adouble) to define all variables which are currently active, i.e. involved in the derivative computations.

An *active region* in ReverseAD is a section of the code which defines the function whose derivatives are required. This involves the dependent variables, independent variable, and the function definition. The following function defines a beginning of an active region:
`void ReverseAD::trace_on<double>();`
And the following function ends an active region:
`std::shared_ptr<TrivialTrace<double>> ReverseAD::trace_off<double>();`

The return value of type `std::shared_ptr<TrivialTrace<double>>` is the *trace*, capturing the active region. The *trace* is:
1. a precise step-by-step record of the function evaluation;
2. the record is with respect to the initial values of the input variables; and
3. the type of the variable is "double", as stated in the template.

3 Declaring a Function

We declare a function by making it a template. E.g. to declare a function $f : \mathbb{R}^2 \to \mathbb{R}, f(x, y) = sin(x * y)$, we need to declare the input variables x, y, the output variable z, and the definition of the function. All of this needs to be in the active region, defined by encapsulating it in between the two function mentioned in the section above.

The operator `<<=` signifies an independent variable. Please make use of this instead of the usual assignment operator, =. For our example, we use the following code:

```
adouble x, y;
x <<= 2.0; y <<= 3.0;
```
This signifies that `x` and `y` are assigned values 2.0 and 3.0 respectively, and are to be treated as independent variables in the active region.

To specify a dependent variable, the library overloads the `>>=` operator, e.g.
```
adouble z;
double vz;
z >>= vz;
```
denotes `z` to be a dependent variable of type adouble, and is assigned the value held by double `vz`.
The variables can also be in arrays or other containers like std::vector<adouble>, e.g. refer to the following code snippet:

```
// declare a vector of adoubles of size 2
        to hold the input variables //
std::vector<adouble> input_vars(2);
for(int i = 0; i < 2; ++i)
        input_vars[i] <<= (double)(i+2);
double vz = sin(input_vars[0] * input_vars[1]);
adouble z >>= vz; // output variable
```

The function is defined by making use of templates to make it independent of the data type of its inputs and output(s).

```
template <typename T>
T foo(T x, T y) {
   return sin(x*y);
}
```

4 Computing the Gradient of a Function

We shall use the prototype function $f = sin(x * y)$, with x, y being double variables. Assume the name of the variable resulting from the active section is appropriately named `trace` (it is the std::shared_ptr returned by the `trace_off<double>` function at the end of the active region). For the derivatives, the library implements several classes for different orders of derivates. Kindly refer to the implementation details here for more information. Each

class requires the *trace* in its constructor. Further, the *trace* assigns numeric labels to each of the independent (and dependent, in case of vector valued functions) variables. For our case, as described in the section above, we have x, y as independent variables. They shall be assigned labels 0 and 1 respectively, while the dependent variable z shall be assigned 0. With all this information, now we are ready to compute the first order derivatives of f, and hence its gradient using the following code:

```
std::unique_ptr<BaseReverseAdjoint<double>>
        grad(new BaseReverseAdjoint<double>(trace));
```

to create an instance, grad, of the class BaseReverseAdjoint.

```
std::shared_ptr<DerivativeTensor<size_t, double>> tensor =
        grad.compute(2, 1);
```

the compute(n,m) function has two arguments, $n :=$ number of independent variables (2 here), and $m :=$ number of dependent variables (1 for this case).

Finally to get the values, use the function
`get_internal_coordinate_list(n, m, &size, &tind, &values)`,
where:
$n =$ index of the dependent variable (should be 0 for our example as we only have a single dependent variable).
$m =$ the order of derivative (should be 1 here, as gradient has first order derivatives).
size := a variable of type *size_t*. It stores size of the gradient array (will be 2 for our case, as gradient array consists $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ evaluated at the given values of x and y).
tind := a variable of type $size\_t **$; and
values := a variable of type double*, representing the type for the independent variable. It points to an array of doubles that stores the actual values of the derivaties computed.

For our example, we use the following code (we set x, y to be respectively 1.0, and π):

```
size_t size;
size_t** tind;
double* values;
// adjoints : dep[0].order[1]
```

```
tensor->get_internal_coordinate_list(0, 1, &size,
                                     &tind, &values);
std::cout << "size of gradient = " << size << std::endl;
for (size_t i = 0; i < size; i++) {
        std::cout << "G["<< tind[i][0] << "]
                = " << values[i] << std::endl;
}
```

This should result in the follwing output:
size of gradient = 2
G[0] = -3.14159
G[1] = -1

5 Computing the Hessian

Computing Hessian for a scalar function is quite simple following the previous section. We use the BaseReverseHessian class instead of the BaseReverseAdjoint. Further as Hessian has second order derivatives, we also need to use 2 for m in the `get_internal_coordinate_list()` function. Please note the library assumes f, the first and mixed order partial derivatives are all continuous, leading to the Hessian matrix being symmetric. Hence it only computes and stores the lower triangular portion in a sparse structure.

Example code for computing the Hessian for $sin(x * y)$ at $x = 0.5, y = \pi$:

```
std::unique_ptr<BaseReverseHessian<double>>
        hessian(new BaseReverseHessian<double>(trace));
std::shared_ptr<DerivativeTensor<size_t, double>>
        tensor = hessian->compute(2,1);

// retrieve results
size_t size;
size_t** tind;
double* values;
// hessian : dep[0].order[2]
tensor->get_internal_coordinate_list(0, 2,
                        &size, &tind, &values);
std::cout << "size of hessian = " << size << std::endl;
for (size_t i = 0; i < size; i++) {
```

```
        std::cout << "H[" << tind[i][0] <<
        ", " << tind[i][1] << "] = " <<
        values[i] << std::endl;
}
```

Should produce the following output:
size of hessian = 3
H[0, 0] = -9.8696
H[1, 0] = -1.5708
H[1, 1] = -0.25

6 Jacobian

Consider a vector field, say $F = \{f_1, f_2\}$, and f_1, f_2 are functions of say x, y, i.e. $f_1 = f_1(x, y), f_2 = f_2(x, y)$. This should be defined the same way as defining the function as explained above. Then using the class BaseReverseAdjoint, the user can compute the Jacobian of F with respect to x, y at the (current) values of x, y. Example code is given below:
Assuming the user has defined some function, $F, (ay_1, ay_2) = F(ax_1, ax_2)$

```
adouble ax1,ax2,ay1,ay2;
double x1,x2,y1,y2;
ReverseAD::trace_on<double>(); // begin active region
// ax1 is the first independent variable (index 0 in trace)
ax1 <<= x1;
// ax2 is second independent variable (index 1 in trace)
ax2 <<= x2;
(y1, y2) = F(ax1, ax2); // using implementation of F
ay1 >>= y1; // ay1 is the 0-th index dependent variable
ay2 >>= y2; // ay2 is the 1-st index dependent variable
std::shared_ptr<TrivialTrace<double>> trace =
                        ReverseAD::trace_off<double>();
ReverseAD::BaseReverseAdjoint<double> Adjoint(trace);
std::shared_ptr<DerivativeTensor<size_t, double>> tensor =
                        Adjoint.compute(2,2);

// retrieve results
size_t size;
size_t** tind;
```

```
double* values;
// For the first dependent variable (index 0 in trace)
tensor->get_internal_coordinate_list(0, 1,
                                     &size, &tind, &values);
std::cout << "size of adjoints = " << size << std::endl;
for (size_t i = 0; i < size; i++) {
        std::cout << "A["<< tind[i][0] << "] = " <<
                          values[i] << std::endl;
}

// For the second dependent variable (index 1 in trace)
tensor->get_internal_coordinate_list(1, 1,
                                     &size, &tind, &values);
std::cout << "size of adjoints = " << size << std::endl;
for (size_t i = 0; i < size; i++) {
        std::cout << "A["<< tind[i][0] << "] = " <<
                          values[i] << std::endl;
}
```

Notice this is similar to computing the gradient of each of the dependent variables, and stacking them on top of each other to get the Jacobian. That is given a vector field $F = \{f_i\}_{i=1}^{s}$, with $f_i = f_i(x_1, x_2, \ldots, x_t)$, the k^{th} row of the Jacobian matrix of F is composed of the derivative of f_k.

7 Hessian-Vector Product

We assume the user is familiar with the cross-country modes to compute derivatives in automatic differentiation. The library implements it in the file single_forward.hpp, by defining a type *SingleForward*. Each *SingleForward* variable has two parts: a value part; and a derivative part. These can individually be accessed via using the methods getVal() and getDer() respectively. A Hessian-vector product then is the result of a forward-over-reverse mode in automatic differentiation. In the ReverseAD library, we need to direct the computation type, from *double* as we have been doing so far in the manual to *SingleForward*. Refer to the example code below for a demonstration:

```
// x1, x2 are the initial values, dx1, dx2 are the intial
// directional derivatives, or the vector to be multiplied
// with the Hessian.
SingleForward sf_x1(x1, dx1);
```

```
SingleForward sf_x2(x2, dx2);
BaseActive<SingleForward> sf_ax1;
BaseActive<SingleForward> sf_ax2;
BaseActive<SingleForward> sf_ay1;
BaseActive<SingleForward> sf_ay2;

// Notice trace will not be pointing to doubles now
ReverseAD::trace_on<SingleForward>();
sf_ax1 <<= sf_x1;
sf_ax2 <<= sf_x2;
// assume you have the function implementation
(sf_y1, sf_y2) = F(sf_ax1, sf_ax2);
sf_ay1 >>= sf_y1;
sf_ay2 >>= sf_y2;
std::shared_ptr<TrivialTrace<SingleForward>> trace =
                ReverseAD::trace_off<SingleForward>();

// This is almost the same as before except that the
        // type is SingleForward instead of double
// Now for Hessian-vector product, we can use:
BaseReverseAdjoint<SingleForward> Adjoint(trace);
std::shared_ptr<DerivativeTensor<size_t, SingleForward>>
        tensor = Adjoint.compute(2,2);

// Notice that the result "tensor" is also of SingleForward type
// Now for y1, the 0-th dependent variable.
size_t j_size;
size_t** j_ind;
SingleForward* j_values;
tensor->get_internal_coordinate_list(0, 1,
                        &j_size, &j_tind, &j_value);

// we shall expect j_size == 2, for non-sparse cases
SingleForward j_x1 = j_values[0];
SingleForward j_x2 = j_values[1];

// The adjoint of y1 w.r.t (x1, x2) is:
{[j_x1.getVal(), j_x2.getVal()]}

// The Hessian-vector product of y1 w.r.t (x1, x2),
```

```
        // with the vector (dx1, dx2) is:
{ [j_x1.getDer(), j_x2.getDer()] }
```

The key idea here is to use the SingleForward type to handle the forward mode part. Create the trace with SingleForward type, and run the reverse mode for Adjoint, thus getting us the forward-over-reverse mode, i.e, the Hessian vector product.

8 Higher Order Derivatives

For higher order derivatives, other functions have been implemented. For example if one is looking to compute the third order derivative, then one should use BaseReverseThird. The code required to compute after one has the trace accumulated is very similar to second or first order derivatives. A snippet has been produced below:

```
// n = the number of independent variables //
std::unique_ptr<ReverseAD::BaseReverseThird<double>>
        thirdDer (new ReverseAD::
                BaseReverseThird<double>(trace));
std::shared_ptr<DerivativeTensor<size_t, double>>
        tensor = thirdDer->compute(n,1);

// retrieve results
size_t size;
size_t** tind;
double* values;
// hessian : dep[0].order[3]
tensor->get_internal_coordinate_list(0,
                3, &size, &tind, &values);
std::cout << "size of tensor = " << size << std::endl;
for (size_t i = 0; i < size; i++) {
        std::cout << "T[" << tind[i][0] << ", "
        << tind[i][1] << ", " << tind[i][2] << "] = "
        << values[i]
        << std::endl;
}
```

Notice that all derivatives are symmetric, so only the *lower part* of the derivative tensor is stored. One can get the full tensor, e.g. for a third order one,

T, by noticing that $T_{i,j,k} = T_{\pi(i,j,k)}$, where $\pi(i,j,k)$ is any permutation of the indices.

For derivatives of order 4-6, please use the BaseReverseTensor class. The method to compute is similar after populating the trace. To retrieve the values for the $k-th$ order derivative, notice that the variable `tind`: tind[i][0] to tind[i][k-1] would be used.

9 Reusing the Trace

In a scenario where the derivative tensors (gradient/Jacobian/Hessian etc.) are to be computed multiple times, at different points, for example in an iterative scheme like Newton's method - it is likely that the trace itself will not change. Recomputing the trace each time the derivatives are needed is highly redundant. In this case, (it is stressed that the trace itself does not change, but only the point at which derivatives are needed varies), the library provides for using the same trace in BaseFunctionReplay. A code snippet is provided below:

Compute the trace once only, perhaps in the constructor of your class, or any initializing routine. We assume it is called init_trace_f.

```
std::shared_ptr<TrivialTrace<double>> new_trace =
        BaseFunctionReplay::replay_forward(init_trace_f, x, n);
BaseReverseAdjoint<double> adjoint(new_trace);
```

Here x is (or pointer to) an array of the (new) values of independent variables the derivatives are required at. Then the derivatives could be obtained using the same aforementioned code. The user can use the corresponding class for the derivative of appropriate order.