ADOOpp

Automatic Differentation through Operator Overloading in C++

Dylan Bassi

November 8, 2020

1 Summary

For my first project of CSE 701, I chose to create a sparse implementation for a DAE structural analysis method referred to as the Sigma Matrix method. This project required me to utilize operator overloading quickly determine the order of potentially millions of independent variables. Seeing this new functionality of operator overloading, coupled with the fact that my research centers around the computation of solutions to differential algebraic equations, helped me arrive at the idea for my next project.

For this project I will be implementing forward mode Automatic Differentiation (AD) through operator overloading in C++. Condensing this gave me the acronym which I will refer to this package throughout the text: ADOOpp. This project aims to create an AD package through the use of Dual Numbers, an extension of the Real Number, system. A Dual number object will implemented which will overload the mathematic operators to track both the real and dual element analysis of operations. Due to the nature of dual numbers, this will automatically track the first order derivative of any function.

Analysis of the package will be performed through various tests that will be designed to produce verifiable results of the AD. The first test will cover scalar multivariate functions, the second test will cover vector multivariate functions and the third test will produce a nice comparison plot of a surface and its dual surface.

2 Code

The code is split into a couple of files that are easily grouped. There is the dual number class (in `dual.hpp` and `dual.cpp`), the main function (in `compute.cpp`) and our test functions (in `plottest.hpp` and `jactest.hpp`).

2.1 The Dual Number Class

Dual Numbers serve the basis of the forward mode automatic differentiation in this package. The dual numbers are implemented in a class named Dual, contained within the adoopp namespace. The main idea behind the implementation of this class, is that there is an object which tracks both the real and dual element for a given function or variable. By overloading the mathematic operations to properly track these changes, the derivative can be automatically computed along with the real value. This gives us a class implementation of Dual which requires a `real_` and `dual_` private value that is updated on arithmetic operations.

```
  Dual() : real_(0), dual_(0) {}
  Dual(double r) : real_(r), dual_(0) {}
  Dual(double r, double d) : real_(r), dual_(d) {}
  Dual(const Dual& t) : real_(t.real_), dual_(t.dual_) {}
#ifdef SPARSE_DAE
```

```
  Dual(const Dual&& t) : real_(t.real_), dual(t.dual_) {
    dual_map_ = std::move(t.dual_map_);
  }
#else
  Dual(const Dual&& t) : real_(t.real_), dual_(t.dual_) {}
```

Overloaded constructor operator, indicating how real and dual values can be assigned, or another `Dual` number can be used to construct a new `Dual` object.

2.1.1 Forward Mode

The forward mode is implemented simply by overloading the various mathematic operators to include operations on the `dual_` number as well as the `real_` value. This is performed by execution of the chain rule.

```
Dual operator*(const Dual& t1, const Dual& t2) {
  Dual temp(t1);
  temp.real_ *= t2.real_;
  temp.dual_ = t1.real_ * t2.dual_ + t1.dual_ * t2.real_;
  return temp;
}
```

An example is seen in the multiplication operator `*`.

A temporary `Dual` is created with the values held by `t1`. Then the real values of the temporary values are multiplied by the real values of `t2`. Next the chain rule is applied and we see that `d(x*y) = x*dy+y*dx`. Finally the temporary `Dual` is returned.

```
Dual sin(const Dual& t) {
  if (t.is_const()) {
    assert(t.dual_ == 0);
    return Dual(std::sin(t.real_), 0);
  }
  double real_out = std::sin(t.real_);
  double dual_out = std::cos(t.real_) * t.dual_;
  return Dual(real_out, dual_out);
}
```

We can see with how the same idea applies for other mathematic operators as well. The `sin` operator is overloaded to account for the chain rule's effect on the `dual_` portion of the number.

The overloaded operators for ADOOpp are:

- arithmetic operators (+,-,*,/)

- assignment operators (+=,-=,*=,/=,=)
- trig operators (sin,cos,tan and their inverses)
- exponential operators (exp(x),log(x))
- power operators (pow(x,d),sqrt(x),sqr(x))

2.2 Our Main Function (Compute)

Our main function just runs the tests of the AD with preassigned numbers.

```
int main(void) {
  const int sizeTest1 = 100000;  // This problem is O(n*cost(f)), can be
   higher
  const int sizeTest2 = 1000;  // This problem is O(n^2*cost(f)); takes
   longer
  const int sizeTest3 = 30;  // Too many points will make our plot crowded
   (also
                             // python is very slow)
  // Expected behavior:  "Jacobian Test 1 Completed:  N" printed to terminal
  runJacTest1(sizeTest1);
#define OMP_NUM_THREAS = 4;  // This test is poorly optimized, 4 is maximal
   gain
  // Expected behavior:  "Jacobian Test 2 Completed:  N N" printed to
   terminal
  runJacTest2(sizeTest2);
  std::string outputFile1 = "file1.out";
  // Expected bavior:  tsv of x*sin(y) and surface tangent to (sin(y)
   +x(cos*y))
  // Plot with python3 plot.py (need numpy and matplotlib)
  plotFunc(sizeTest3, outputFile1);
```

2.3 Test Functions

A series of test functions were created in order to showcase the AD abilities of the code. These were split into two header files: `jactest.hpp` and `plottest.hpp`.

2.3.1 Jacobian Tests

The first jacobian test is on a multivariate scalar function. We assign a function, $f = \Sigma_i x_i^2$. This gives us the result that $\partial_i f = 2x_i$

```
        if (i == wrt)
          check_deriv = 2 * vars[i].real();
      }
      if (func.dual() != check_deriv) {
        printf("Jacobian Test 1 Error: derivative wrong at: %d \n", wrt);
```

By assigning the values $x_i = i + 1$ we can simply check that our result holds that the derivative is equal to $2i$ at each spot in our check. When we run this test, we will know it executes properly because we will not see this print statement.

The second Jaboian Test is on a multivariate vector function. This will we require us to compute a jacobian matrix and check it similarly to the first test. This time, we will assign the function $f_j = x_i^2 x_j$. This gives us the result that our jacobian matrix will take the form of: $J_{ij} = 2x_i x_j + \delta_{ij} x_i x_j$

```
      // If (dual != deriv); i!=j
      if (func.dual() != 2 * vars[wrt].real() * vars[j].real())
        // If (dual != deriv); i==j
        if (func.dual() != 2 * vars[wrt].real() * vars[j].real() +
                               vars[wrt].real() * vars[wrt].real()) {
          printf("Jacobian Test 2 Error: derivative wrong at: %d %d\n", j,
   wrt);
```

Once again, we will know this test executes properly if we do not see this print statement appear on execution.

2.3.2 Plot Test

This test was designed out of the constraining factor of the jacobian tests, which were I wanted a large N which would produce non-trivial, varying results for each element of the Jacobian, which could be checked to guarantee it produced accurate results. This limited me to functions of which the behaviour was very easy to alogirthmically predict and compute for large values of the variable. In order to investigate that complex data was being accurately represented, I felt a plot of a surface, and the tangential surface would provide a visually intuitive way to confirm accuracy.

In this test, a 2D surface is generated according to $f(x, y) = xsin(y)$. This should produce a tangent surface with the function $f'(x, y) = sin(y) + xcos(y)$. By printing this data out to a file, and plotting with python, we will be able to visually inspect very quickly if the AD is producing the correct function.

```
  func = x * sin(y);
  outFile.precision(20);
```

```
outFile << x.real() << "\t" << y.real() << "\t" << func.real() << "\t"
        << func.dual() << "\n";
```

3 Results

My initial problem statement was to create a robust enough Automatic Differentiation package to allow me to create a relatively straightforward implementation to determine the Jacobian matrix for a set of multivariate equations. This was successfully accomplished and documented in the following tests tests. The first test investigates the scalar multivariate functon $f(x_i) = \Sigma_i x_i^2$. The second will investigate the vector multivariate function $f_j = x_i^2 x_j$. The third test will produce a plot of a $f(x, y) = xsin(y)$ and the surface tangent to it. This data will be plotted and compared to hard computed values in Python using numpy and matplotlib.

3.1 Building and Running

Code should be compile easily, just simply run the command:

```
g++ -fopenmp *.cpp -o program -O2
```

This project is written with OMP as well. Simply add the `-fopenmp` flag to the gcc compiler. Note, I never got around to optimizing the parallelism of the second test, but a noticeable improvement is still observed.

3.2 Jacobian Test 1 - Scalar Multivariate Function

The results of running the first jacobian test are seen in fig. 1. We can see, for the results printed out, that $J_i = 2i$. We also receive no errors indicating that our AD was not computed correctly.

3.3 Jacobian Test 2 - Vector Multivariate Function

The results of running the second jacobian test are seen in fig. 2. We can see, for the results printed out, that $J_{ij} = 2ij + \delta_{ij} ij$. We also receive no errors indicating that our AD was not computed correctly.

3.4 Plot Test - Surface of $xsin(y)$ and its Tangent

A plot of the function $f(x, y) = xsin(y)$ and it's derivative $f'(x, y) = sin(y) + xcos(y)$ are given in fig. 3. From this image we can discern no difference, so we can conclude that the data being computed is in agreement. This is just a rough visual test, since we technically

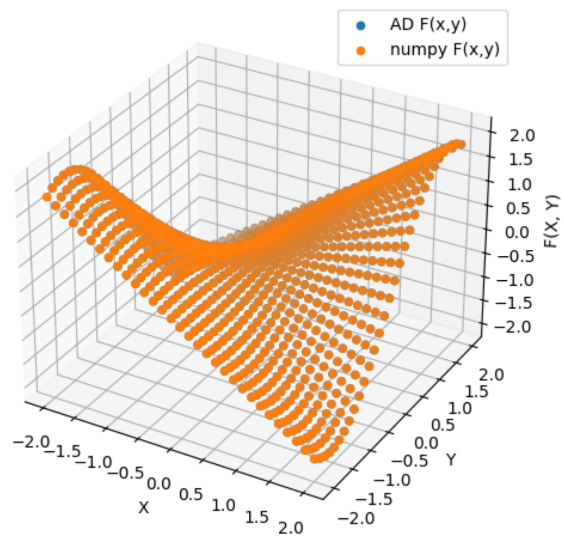
```
Jac(86000):172000
Jac(87000):174000
Jac(88000):176000
Jac(89000):178000
Jac(90000):180000
Jac(91000):182000
Jac(92000):184000
Jac(93000):186000
Jac(94000):188000
Jac(95000):190000
Jac(96000):192000
Jac(97000):194000
Jac(98000):196000
Jac(99000):198000
Jac(100000):200000
Jacobian Test 1 Completed with size: 100000
```

Figure 1: Output from Jacobian Test 1. We see that indeed $J_i = 2i$

```
Jac(900,600):1.08e+06
Jac(900,700):1.26e+06
Jac(900,800):1.44e+06
Jac(900,900):2.43e+06
Jac(900,1000):1.8e+06
Jac(1000,100):200000
Jac(1000,200):400000
Jac(1000,300):600000
Jac(1000,400):800000
Jac(1000,500):1e+06
Jac(1000,600):1.2e+06
Jac(1000,700):1.4e+06
Jac(1000,800):1.6e+06
Jac(1000,900):1.8e+06
Jac(1000,1000):3e+06
Jacobian Test 2 Completed with size: 1000 1000
```

Figure 2: Output from Jacobian Test 2. We see that indeed $J_{ij} = 2ij + \delta_{ij} ij$

already know it is computing correctly from the jacobian tests. This test really just gives a means for checking if a given function has not been implemented correctly.



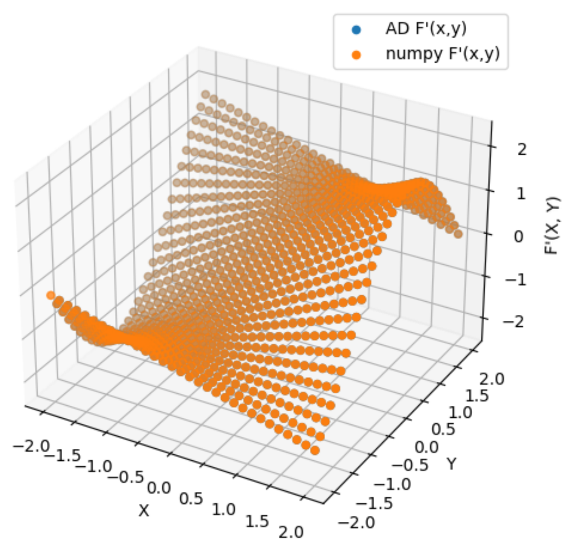


Figure 3: Plot of $xsin(y)$ and $sin(y) + xcose(y)$ as computed by both numpy and ADOOpp. Considering there is no discernible difference, the results are in agreement.

Index

compute.cpp
mainf, 4
dual.cpp
operator*, 3
sin, 3
dual.hpp
Dual, 3
jactest.hpp
check_deriv, 5
deriv_check2, 5
plottest.hpp
plottest, 6