The Analytics Template Library

A Developer's Guide

Matthew Supernaw
National Oceanic and Atmospheric Administration
Office of Science and Technology
National Modeling and Assessment team

Version 1.0
November 2017

Contents

1 Introduction

The Analytics Template Library (ATL) is a scientific computing library with an emphasis on gradient based optimization. ATL leverages the power of template metaprogramming for flexibility, extensibility, and speed. This guide is intended to give the user a basic understanding of how to develop programs in ATL. The information in this document is intended for anyone interested in scientific computing in C++ and it is expected that the reader will have a basic understanding of the C++ programming language, as well as scientific computing.

2 Template Metaprogramming

Template metaprogramming is a technique in which templates are used by the compiler to generate source code. This allows the developer to focus on the architecture and flow of the program and delegate any implementation required to the compiler. This technique has the benefit of reducing source code and development time. Here is an example of how template metaprogramming works in C++.

```
/**
 * Generic Function to add two values.
 *
 * @param a
 * @param b
 * @return
 */
template<class T>
T add(T a, T b) {
    return a + b;
}

void main(){

    double d = add<double>(1.01, 1.01);
    std::cout << d << "\n";

    float f = add<float>(1.01f, 1.01f);
    std::cout << f << "\n";

    int i = add<int>(1, 1);
    std::cout << i << "\n";
}
```

Output

```
2.02
2.02
2
```

As you can see, we have one template function that successfully handled three different data types. So, true to the nature of template metaprogramming, we have reduce the amout of source code required and thus reduced development time. Templates can also be applied to classes as well, for example the following will give us the identical output:

```cpp
template <class T>
class Add{
    public:
    T Evaluate(T a, T b){
        return a+b;
    }
};

void main(){
    Add<double> d;
    std::cout << d.Evaluate(1.01, 1.01)<< "\n";

    Add<float> f;
    std::cout << f.Evaluate(1.01f, 1.01f) << "\n";

    Add<int> i;
    std::cout << i.Evaluate(1, 1) << "\n";
}
```

Output

```
2.02
2.02
2
```

2.1 Expression Templates

Expression templates are a template metaprogramming technique in which templates are used to represent part of an expression. The expression template can be evaluated at a later time, or even passed to a function. Expression templates are considered a source code optimization technique because their use reduces the amount of temporary variables created in a given calculation. Furthermore, expression templates are a special case of static polymorphism, this is a form of polymorphism that is handled at compile time, rather than runtime. To demonstrate how expression templates work, let's create a small vector library that can handle simple operators such as + and -. First, we'll need a base class.

```cpp
template<class T, class A>
struct VectorExpr {

    const A & Cast() const {
        return static_cast<const A&> (*this);
    }

    T operator()(int i) const {
        return Cast().operator()(i);
    }

    VectorExpr& operator=(const VectorExpr & exp) const {
        return *this;
    }
};
```

Ok, now we have a base class, that can take a template parameter T as the base data type, and template parameter A, which will be the class inheriting from *VectorExpr*. Now, lets inherit from *VectorExpr* and create another class to perform the addition operation:

```
template<class T, class LHS, class RHS>
struct VectorAdd : public VectorExpr<T, VectorAdd<T, LHS, RHS> > {
    const LHS& lhs;
    const RHS& rhs;

    VectorAdd(const VectorExpr<T, LHS>& l,
           const VectorExpr<T, RHS>& r)
    : lhs(l.Cast()), rhs(r.Cast()) {
    }

    T operator()(int i) const {
        return lhs(i) + rhs(i);
    }

};

template<class T, class LHS, class RHS>
inline const VectorAdd<T, LHS, RHS> operator+(const VectorExpr<T, LHS>& l,
const VectorExpr<T, RHS>& r) {
    return VectorAdd<T, LHS, RHS>(l.Cast(), r.Cast());
}
```

As you can see from the above code listing, we have a class called *VectorAdd*, which inherits from *VectorExpr*. We have overloaded the function *T operator()(int i) const* to provide the appropriate operation. In addition, we created the operator *operator+* to handle the actual operation. Notice that *operator+* returns an instance of *VectorAdd* rather than an instance of a *Vector*, which is defined below. We can also do this for the operation -:

```
template<class T, class LHS, class RHS>
struct VectorMinus : public VectorExpr<T, VectorMinus<T, LHS, RHS> > {
    const LHS& lhs;
    const RHS& rhs;

    VectorMinus(const VectorExpr<T, LHS>& l,
           const VectorExpr<T, RHS>& r)
    : lhs(l.Cast()), rhs(r.Cast()) {
    }

    T operator()(int i) const {
        return lhs(i) - rhs(i);
    }

};

template<class T, class LHS, class RHS>
inline const VectorMinus<T, LHS, RHS> operator-(const VectorExpr<T, LHS>& l,
const VectorExpr<T, RHS>& r) {
    return VectorMinus<T, LHS, RHS>(l.Cast(), r.Cast());
}
```

Lets create the actual *Vector* class:

```
template<class T, int SIZE>
class Vector : public VectorExpr<T, Vector<T, SIZE> > {
    T data[SIZE];

public:

    Vector() {
    }

    Vector& operator=(const T& value){
        for (int i = 0; i < SIZE; i++) {
            data[i] = value;
        }
        return *this;
    }

    template<class R, class A>
    Vector& operator=(const VectorExpr<R, A>& exp) {
        for (int i = 0; i < SIZE; i++) {
            data[i] = exp(i);
        }
        return *this;
    }

    T operator()(int i) const {
        return data[i];
    }

    size_t Size(){
        return SIZE;
    }
 };
```

In the above *Vector* definition, we have overloaded the operator *T operator()(int i) const* just as we did with *VectorAdd* and *VectorMinus*, the difference being that a stored value is returned rather than a computed one. Now we can use this code to do work:

```
int main() {

    Vector<double, 2> a;
    a = 1.0;

    Vector<double, 2> b;
    b = 2.0;

    Vector<double, 2> c;
    b = 3.0;

    Vector<double, 2> d;
    d = a - b + c;

    for(int i =0; i < d.Size(); i++){
        std::cout<<d(i)<<" ";
    }

    return 0;
}
```

Output

```
2 2
```

In this simple example, we've created a small library to do elementary operations on vectors using template metaprogramming techniques. Of course, the advantage of writing code like this is that we only had to write one vector library that can handle multiple data types and we've used the expression templates to eliminate the need to create and allocate memory for intermediate *Vector*'s at each operation(+ and -).

3 Automatic Differentiation

The term "Automatic Differentiation" refers to a set of methods to numerically evaluate the derivative of a function in a computer program. Automatic Differentiation (AD) exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.) ("Automatic differentiation.", 2015). The AD components in ATL are expression template based and are not only capable of computing exact gradients, but also exact Hessian matrices. This is particularly useful for gradient-based optimization problems when the Hessian is desired to improve search direction. In this section, we provide a very basic description of automatic differentiation. Compared to finite-difference techniques, AD has proven to be more efficient for accurately computing partial derivatives. At the heart of AD is the chain rule. Recall from calculus, the chain rule is a method for computing the derivative of two or more functions:

$$y = f(g(x)) \tag{1}$$

with *y=f(g(x))* as the outer function and *g(x)* as the inner function.

Chain Rule:

$$f(g(x))\text{'} = g\text{'}(x)\, f\text{'}(g(x)) \tag{2}$$

or

$$\frac{df}{dx} = \frac{dg}{dx}\frac{df}{dg} \tag{3}$$

with $\frac{df}{dg}$ as the outer derivative and $\frac{dg}{dx}$ as the inner derivative. In general, there are two modes of AD, forward and reverse.

3.1 Forward Mode

Forward mode (or tangent linear) AD traverses the chain rule from inside to outside. That is, from (3), $\frac{df}{dg}$ is computed before $\frac{dg}{dx}$. To demonstrate how the forward accumulation of the chain rule works, consider the expressions $f(x_1, x_2) = ln(x_1 x_2)$. Here $g(x_1, x_2) = x_1 x_2$ and $f(x_1, x_2) = ln(g(x_1, x_2))$. So, to find the gradient $\nabla f(x_1, x_2)$ we must evaluate $f(x_1, x_2)$ and record these operations to a "Tape" so we can evaluate the partial derivatives later.

Evaluation	**Tape**
$x_1 = 3.1459$	$x_1' = 0.0$
$x_2 = 2.0$	$x_2' = 0.0$
——————	——————
$g(x_1, x_2) = x_1 x_2$	$g(x_1, x_2)' = x_1' x_2 + x_1 x_2'$
$f(g(x_1, x_2)) = ln(g(x_1, x_2))$	$f(g(x_1, x_2))' = \frac{g(x_1, x_2)'}{g(x_1, x_2)}$

Now that we have a record of $f(x_1, x_2)$ on the "Tape", we can apply the forward mode accumulation of the chain rule and compute the gradient:

Let $\nabla f(x_1, x_2) = [x_1', x_2']$

Compute: x_1'

Tape

$x_1' = 1.0$***(seed)***

$x_2' = 0.0$

——————

$g(x_1, x_2)' = x_1'x_2 + x_1x_2' = 1.0 * 2.0 + 3.1459 * 0 = 2.0$

$f(g(x_1, x_2))' = \frac{g(x_1,x_2)'}{g(x_1,x_2)} = \frac{2.0}{2.0*3.1459} = 0.317874$

Compute: x_2'

Tape

$x_1' = 0.0$

$x_2' = 1.0$***(seed)***

——————

$g(x_1, x_2)' = x_1'x_2 + x_1x_2' = 0.0 * 2.0 + 3.1459 * 1.0 = 3.1459$

$f(g(x_1, x_2))' = \frac{g(x_1,x_2)'}{g(x_1,x_2)} = \frac{3.1459}{2.0*3.1459} = 0.5$

Gives:

$\nabla f(x_1, x_2) = [\frac{df}{dx_1}, \frac{df}{dx_2}] = [0.317874, 0.5]$

The above "Tape" evaluation can be generalized algorithmically by:

Algorithm 1 Forward Mode Accumulation

```
1: $\hat{w} = [x'_1, x'_2, x'_3 ... x'_m]$
2: for $i = 1$ to $m$ do
3:     $\hat{w}[i] = 1.0$
4:     for $j = 1$ to $n$ do
5:         $Tape[j] \rightarrow Evaluate$
6:     end for
7:     $\nabla f[i] = Tape[n] \rightarrow Value$
8:     $\hat{w}[i] = 0.0$
9: end for
```

As you can see from Algorithm 1, for a function $f(x_1, x_2 ..., x_m)$ the "Tape" must be evaluated m times to compute the gradient. For highly parameterized functions, it may be desirable to compute the gradient using reverse mode accumulation.

3.2 Reverse Mode

Reverse mode (or the adjoint method) AD traverses the chain rule from outside to inside. That is, from (3), $\frac{dg}{dx}$ is computed before $\frac{df}{dg}$. It works by accumulating a series of adjoints in the opposite direction of the forward mode method. This can be generalized by the following algorithm (Griewank, 1989):

Algorithm 2 Reverse Mode Accumulation

```
1: Input: $Tape$
2: $\bar{w} = [\bar{x_1}, \bar{x_2}, \bar{x_3} ... \bar{x_{m-1}}] = 0$
3: $\bar{w}[m] = 1$
4: for $i = m$ to $1$ do
5:     $\frac{df}{dx_i} = \bar{w}[i]$
6:     $\bar{w}[i] = 0$
7:     for $j = 1$ to $i$ do
8:         $\bar{w}[j] += \frac{\partial f}{\partial x_i} \bar{w}[j]$
9:     end for
10: end for
11: Output: $\nabla f = \bar{w} = [\bar{x_1}, \bar{x_2}, \bar{x_3} ... \bar{x_m}]$
```

To demonstrate the reverse mode method, lets revisit our example from the forward mode description in the previous section with slight modifications to the "Tape" in order to represent the partial derivative entries.

Evaluation	**Tape**
$x_1 = 3.1459$	$x'_1 = 0.0$
$x_2 = 2.0$	$x'_2 = 0.0$
————	————
$g(x_1, x_2) = x_1 x_2$	$x'_3 = g(x_1, x_2)' = [[x'_1, \frac{\partial g}{\partial x_1} = 1.0 * 2.0], [x'_2, \frac{\partial g}{\partial x_2} = 1.0 * 3.1459]]$
$f(g(x_1, x_2)) = ln(g(x_1, x_2))$	$x'_4 = f(g(x_1, x_2))' = [g(x_1, x_2)', \frac{\partial f}{\partial x_3} = (2.0 * 3.1459)^{-1}]$

Now we have a tape that contains entries that each have a list of adjoints to use in the reverse mode calculation. By traversing the tape from the bottom up, we can easily compute the full gradient using **Algorithm 2**:

$\nabla f = \bar{w} = [\bar{x_1}, \bar{x_2}, \bar{x_3}, \bar{x_4}]$

Tape **Trace**

m = 2 // last elementary operation

$\bar{w}[m] = 1$**(seed)**

---

i = m = 2

$w = \bar{w}[i]$

$\bar{w}[i] = 0$

$\bar{w}[3] += w\frac{\partial f}{\partial x_3} = 1 * (2.0 * 3.1459)^{-1} = 0.159$

$\nabla f = \bar{w} = [0, 0, 0.159, 0]$

---

i = m-1 = 1

$w = \bar{w}[i]$

$\bar{w}[i] = 0$

$\bar{w}[1] += w\frac{\partial g}{\partial x_1} = 0.159 * 2.0 = 0.317874$

$\bar{w}[2] += w\frac{\partial g}{\partial x_2} = 0.159 * 3.1459 = 0.5$

$\nabla f = \bar{w} = [0.317874, 0.5, 0, 0]$

Final Result: $\nabla f = [0.317874, 0.5, 0, 0]$

Reverse mode requires only one sweep to compute all the partials for a function $f(x_1, x_2..., x_m)$, therefore it is more efficient for functions having $m > 1$ variables.

3.3 Reverse Mode with Hessian Extension

The Hessian matrix is a square matrix of all the second order partial derivatives for a function. It describes the local curvature of a function of many variables.("Hessian matrix", 2015) It is often desirable to have the Hessian matrix for a function. Extending Algorithm 2 to compute the Hessian as well as the gradient is simple (Algorithm 3).

Algorithm 3 Reverse Mode With Hessian Accumulation

```
Input: Tape
$\bar{w} = [\bar{x_1}, \bar{x_2}, \bar{x_3} ... \bar{x_{m-1}}] = 0$

$\bar{h} = \begin{vmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_m} \\ \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_m} \\ \frac{\partial^2 f}{\partial x_3^2} & \frac{\partial^2 f}{\partial x_3 \partial x_2} & \frac{\partial^2 f}{\partial x_3 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_3 \partial x_m} \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ \frac{\partial^2 f}{\partial x_m^2} & \frac{\partial^2 f}{\partial x_m \partial x_2} & \frac{\partial^2 f}{\partial x_m \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_m \partial x_m} \end{vmatrix} = 0$

$S = \{w_m\}$
$\bar{w}[m] = 1$
for $i = m$ to 1 do
    $\frac{df}{dx_i} = \bar{w}[i]$
    $\bar{w}[i] = 0$
    $w_i \cup S_i$
    $S_i = \{\}$
    for $j = 1$ to $i$ do
        $\bar{w}[j] += \frac{\partial f}{\partial x_i} \bar{w}[i]$
        for $k = 1$ to $i$ do
            $temp = \bar{h}[i][k] \frac{\partial f}{\partial x_j} + \bar{h}[i][j] \frac{\partial f}{\partial x_k} + \bar{h}[i][i] \frac{\partial f}{\partial x_k} \frac{\partial f}{\partial x_j} + w \frac{\partial^2 f}{\partial x_j \partial x_k}$
            if $temp \neq 0$ then
                $\bar{h}[j][k] += temp$
                $j \in S_i$
                $k \in S_i$
            end if
        end for
    end for
end for
Output:

$\nabla f = \bar{w} = [\bar{x_1}, \bar{x_2}, \bar{x_3} ... \bar{x_m}]$

$\nabla f^2 = \bar{h} = \begin{vmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_m} \\ \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_m} \\ \frac{\partial^2 f}{\partial x_3^2} & \frac{\partial^2 f}{\partial x_3 \partial x_2} & \frac{\partial^2 f}{\partial x_3 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_3 \partial x_m} \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ . & . & . & \cdots & . \\ \frac{\partial^2 f}{\partial x_m^2} & \frac{\partial^2 f}{\partial x_m \partial x_2} & \frac{\partial^2 f}{\partial x_m \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_m \partial x_m} \end{vmatrix}$
```

Essentially all we've done is extend the level of recording on the tape and added an additional loop in the reverse mode procedure to accumulate hessian elements.

3.4 Examples

In this section we'll show how to use the ATL AutoDiff package. We'll start with simple examples to compute gradients and Hessian matrices, followed by an example of how we can use this information in optimization problems by implementing Newton's Method.

3.4.1 Computing Derivatives

We'll start with our simple example from the section on Automatic Differentiation. Recall we had the expression $f(x_1, x_2) = ln(g(x_1, x_2))$. This is how it is coded in ATL.

```
int main(int argc, char** argv) {

    //using 64-bit precision
    typedef atl::Variable<double> variable_t;
    variable_t::tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;

    std::vector<double> gradient;
    std::vector<std::vector<double> > hessian;

    //initialize x1 and x2 and add them to a list;
    std::vector<variable_t*> variables;
    variable_t x1 = 3.1459;
    variable_t x2 = 2.0;
    variables.push_back(&x1);
    variables.push_back(&x2);


    //evaluate our function
    variable_t f = atl::log(x1 * x2);
    std::cout<<"f = "<<f<<"\n\n";

    //compute and extract the gradient and hessian
    variable_t::ComputeGradientAndHessian(
    variable_t::tape, //default gradient structure
     variables,                  //list of independent variables
      gradient,                  //the gradient vector
      hessian                    //the hessian matrix
      );
    std::cout<<"gradient:\n" << gradient;
    std::cout << "\n\nHessian:\n" << hessian;

    return 0;
}
```

This simple program gives the following output:

```
f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25
```

It is not necessary to use vectors to hold gradient and Hessian information. An alternative method of the above example can be accomplished by:

```
int main() {
    //using 64-bit precision
    typedef atl::Variable<double> variable;

    //set the derivative trace level
    variable::tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;
    //initialize x1 and x2 and add them to a list;
    std::vector<variable*> variables;
    variable x1 = 3.1459;
    variable x2 = 2.0;
    variables.push_back(&x1);
    variables.push_back(&x2);


    //evaluate our function
    variable f = atl::log(x1 * x2);
    std::cout << "f = " << f << "\n\n";
    variable::tape.AccumulateSecondOrder();

    std::cout << std::scientific << "gradient:\n" << variable::tape.first_order_derivatives[x1.info->id]
    << "  " << variable::tape.first_order_derivatives[x2.info->id] << "\n";

    std::cout << std::scientific << "hessian:\n";
    for (int i = 0; i < variables.size(); i++) {
        for (int j = 0; j < variables.size(); j++) {
            std::cout << std::left << std::setw(10)
            << variable::tape.second_order_derivatives[variables[i]->info->id][variables[j]->info->id] << " ";
        }
        std::cout << std::endl;
    }

}
```

```
f = 1.83925

gradient:
3.178741e-01  5.000000e-01
hessian:
-1.010439e-01 0.000000e+00
0.000000e+00 -2.500000e-01
```

3.4.2 Controlling The *Tape*

After you have computed derivatives, you can reset the gradient structure (Tape) by calling **void** Reset() on the instance of Tape. Note, you should be mindful of this routine. Failing to call **void** Reset() will result in a polluted recording and give wrong derivatives as a result.

```cpp
int main(int argc, char** argv) {
    typedef double real_t;
    typedef atl::Variable<real_t> variable_t;

    variable_t::tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;
    std::vector<real_t> gradient;
    std::vector<std::vector<real_t> > hessian;

    /// initialize x1 and x2 and add them to a list;
    std::vector<variable_t*> variables;
    variable_t x1 = real_t(3.1459);
    variable_t x2 = real_t(2.0);
    variables.push_back(&x1);
    variables.push_back(&x2);

    //put the operations in a loop and reset after derivatives are computed
    for (int i = 0; i < 5; i++) {

        //evaluate our function
        variable_t f = atl::log(x1 * x2);

        std::cout << "f = " << f << "\n\n";

        //compute and extract the gradient and hessian
        variable_t::ComputeGradientAndHessian(
                variable_t::tape, //default gradient structure
                variables, //list of independent variables
                gradient, //the gradient vector
                hessian //the hessian matrix
                );
        std::cout << "gradient:\n" << gradient;
        std::cout << "\n\nHessian:\n" << hessian;
        std::cout << "\n\n";

        //reset the tape structure
        variable_t::tape.Reset();

    }
    return 0;
}
```

```
f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25
```

You can also pause recording. This is particular useful when line searching and you don't need derivatives.

```
int main() {
    typedef double real_t;
    typedef atl::Variable<real_t> variable_t;

    //set the derivative trace level
    variable_t::tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;


    /// initialize x1 and x2 and add them to a list;
    std::vector<variable_t*> variables;
    variable_t x1 = real_t(3.1459);
    variable_t x2 = real_t(2.0);
    variables.push_back(&x1);
    variables.push_back(&x2);

    std::vector<real_t> gradient;
    std::vector<std::vector<real_t> > hessian;

    //put the operations in a loop and reset after derivatives are computed
    for (int i = 0; i < 5; i++) {

        if ((i % 2) == 0) {
            std::cout << "Not Recording\n";
            variable_t::tape.SetRecording(false);
        } else {
            std::cout << "Recording\n";
            variable_t::tape.SetRecording(true);
        }
        //evaluate our function
        variable_t f = atl::log(x1 * x2);

        std::cout << "f = " << f << "\n\n";

        //extract the gradient and hessian
        //compute and extract the gradient and hessian
        variable_t::ComputeGradientAndHessian(
                variable_t::tape, //default gradient structure
                variables, //list of independent variables
                gradient, //the gradient vector
                hessian //the hessian matrix
                );
        std::cout << "gradient:\n" << gradient;
        std::cout << "\n\nHessian:\n" << hessian;
        std::cout << "\n\n";
        variable_t::tape.Reset();

    }
}
```

Output

```
Not Recording
f = 1.83925

gradient:
0         0

Hessian:
0         0
0         0


Recording
f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


Not Recording
f = 1.83925

gradient:
0         0

Hessian:
0         0
0         0


Recording
f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25


Not Recording
f = 1.83925

gradient:
0         0

Hessian:
0         0
0         0
```

Notice from the above output, when the instance of *Tape* is paused, derivatives are zero, assuming that it was reset before the evaluation. If the *Tape* was not reset, the previous derivatives will be recycled, but no new entries will be recorded.

3.4.3 Using *Tape*'s Other Than The Default

To use an instance of *Tape* other than the default, the *atl::Variable* class has an additional member function called **void** Assign. Here is an example of using an instance of *Tape* other than the default:

```
int main(int argc, char** argv) {

    typedef double real_t;
    typedef atl::Variable<real_t> variable_t;



    //create an instance of Tape
    atl::Tape<real_t> my_tape;
    my_tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;
    my_tape.recording = true;

    /// initialize x1 and x2 and add them to a list;
    std::vector<variable_t*> variables;
    variable_t x1 = real_t(3.1459);
    variable_t x2 = real_t(2.0);
    variables.push_back(&x1);
    variables.push_back(&x2);

    std::vector<real_t> gradient;
    std::vector<std::vector<real_t> > hessian;


    //evaluate our function
    variable_t f;
    //call the Assign function using our instance of Tape
    f.Assign(my_tape, atl::log(x1 * x2));

    std::cout << "f = " << f << "\n\n";

    //extract the gradient and hessian from our instance of Tape
    variable_t::ComputeGradientAndHessian(
            my_tape, //other tape
            variables, //list of independent variables
            gradient, //the gradient vector
            hessian //the hessian matrix
            );
    std::cout << "gradient:\n" << gradient;
    std::cout << "\n\nHessian:\n" << hessian;
    std::cout << "\n\n";
    my_tape.Reset();

}
```

Output

```
f = 1.83925

gradient:
0.317874  0.5

Hessian:
-0.101044 0
0         -0.25
```

Having this capability is particularly useful when one wishes to run multiple models concurrently. Also, even though entries in the *Tape* are thread safe, they use atomic operations, which can result in cache contention between threads. Furthermore, threads sharing the same *Tape* must not have shared dependency as this will result in errors in the accumulation of derivatives. Using a separate *Tape* in each thread, computing the derivatives and adding them to the main instance of *Tape* may help to alleviate these problem. An example of this technique will be covered in the next section **Adjoint Code/Entries**.

3.4.4 Adjoint Code/Entries

In this section we'll discuss the topic of adjoint code. Adjoint code is something that can be found in ADMB (Fournier et al) and we've just expanded the concept for ATL. This is a particularly convenient method for both reducing the stack size, as well as accomplishing concurrency for threads with shared dependency. The following code listing shows how to use adjoint code:

```
template<class REAL_T>
const atl::Variable<REAL_T> F(atl::Variable<REAL_T>& x, atl::Variable<REAL_T>& y) {
    //compute the value
    REAL_T f = std::sin(x.GetValue()) * std::cos(y.GetValue());
    atl::Variable<REAL_T> ret(f);

    //make a list of pointers to independent variables
    std::vector<atl::Variable<REAL_T>* > independents;
    independents.push_back(&x);
    independents.push_back(&y);

    //set the gradient values for this function
    std::map<uint32_t,REAL_T> gradient;
    gradient[x.GetId()] = std::cos(x.GetValue()) * cos(y.GetValue());
    gradient[y.GetId()] = -1.0 * std::sin(x.GetValue()) * std::sin(y.GetValue());

    //set the Hessian values for this function
    std::map<uint32_t,std::map<uint32_t,REAL_T> > hessian;
    hessian[x.GetId()][x.GetId()] = -1.0 * std::sin(x.GetValue()) * std::cos(y.GetValue());
    hessian[x.GetId()][y.GetId()] = -1.0 * std::cos(x.GetValue()) * std::sin(y.GetValue());
    hessian[y.GetId()][x.GetId()] = -1.0 * std::cos(x.GetValue()) * std::sin(y.GetValue());
    hessian[y.GetId()][y.GetId()] = -1.0 * std::sin(x.GetValue()) * std::cos(y.GetValue());

    //build the adjoint entry from the next one in the global gradient structure
    atl::Variable<REAL_T>::BuildAdjointEntry(
            atl::Variable<REAL_T>::tape.NextEntry(),
            ret,
            independents,
            gradient,
            hessian);

    //return a Variable
    return ret;
}

int main(int argc, char** argv) {

    atl::Variable<double>::tape.derivative_trace_level = atl::SECOND_ORDER_REVERSE;

    atl::Variable<double> x(3.14);
    atl::Variable<double> y(1.5);

    std::vector<atl::Variable<double>* > independents;
    independents.push_back(&x);
    independents.push_back(&y);

    atl::Variable<double> f = F(x, y);
    std::vector<double> gradient(2);
    std::vector<std::vector<double> > hessian(2, std::vector<double>(2));
    atl::Variable<double>::ComputeGradientAndHessian(
            atl::Variable<double>::tape,
            independents,
            gradient,
            hessian);

    std::cout<<"Result from adjoint entry:\n";
    std::cout <<"gradient"<< gradient << "\n";
    std::cout <<"Hessian:\n"<< hessian << "\n";

    atl::Variable<double>::tape.Reset();

    atl::Variable<double>f2 = atl::sin(x) * atl::cos(y);

    atl::Variable<double>::ComputeGradientAndHessian(
            atl::Variable<double>::tape,
            independents,
            gradient,
            hessian);

    std::cout<<"Result from computed entry:\n";
    std::cout <<"gradient"<< gradient << "\n";
    std::cout <<"Hessian:\n"<< hessian << "\n";
}
```

Output:

```
Result from adjoint entry:
Result from adjoint entry:
gradient-0.0707371 -0.00158866
Hessian:
-0.00011266 0.997494
0.997494  -0.00011266

Result from computed entry:
gradient-0.0707371 -0.00158866
Hessian:
-0.00011266 0.997494
0.997494  -0.00011266
```

In the above example, we computed all the necessary derivatives to supply our main *Tape* with an adjoint entry. The result is a reduced amount of entries into our main *Tape*. In the *Strategies for Concurrency* section, we'll expand on this concept and give an example of this technique can be useful for threads that have shared variable dependency.

3.4.5 Implementing Newton's Method

Now we'll see how we can apply this derivative information to an optimization problem. Newton's method in optimization is an iterative procedure for finding the roots of a objective function that is differentiable. In a single parameter objective function, Newton's method attempts to converge to a stationary point using:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{4}$$

For objective functions with more than one parameter:

$$x_{n+1} = x_n - [Hf(x_n)]^-1\nabla f(x_n) \tag{5}$$

where $[Hf(x_n)]^-1$ is the inverse Hessian matrix.

Lets start by creating a class called "MyFunctionMinimizer".

```
template<class T>
   class MyFunctionMinimizer {
       std::vector<T> gradient;
       std::vector<std::vector<T> > hessian;
       std::vector<atl::Variable<T>* > parameters;
   public:

       virtual void ObjectiveFunction(atl::Variable<T>& f) {

       }
   };
```

Recall from (5), we'll need the inverse of the Hessian matrix for functions with more than one parameter. Lets add a function to compute the inverse of the Hessian using the Gauss-Jordan elimination algorithm.

```
template<class T>
class MyFunctionMinimizer {
    ...
    /**
     * Invert the Hessian using Gauss-Jordan Elimination.
     */
    void InvertHessian() {
        int nrows = this->parameters.size();
        for (size_t i = 0; i < nrows; ++i) {
            for (size_t j = 0; j < nrows; ++j) {
                if(i == j){
                    inverse_hessian[i][j] = 1.0;
                }else{
                    inverse_hessian[i][j] = 0.0;
                }
            }
        }


        for (size_t dindex = 0; dindex < nrows; ++dindex) {

            if (hessian(dindex, dindex) == 0) {
                hessian[dindex].swap(hessian[dindex + 1]);
                inverse_hessian[dindex].swap(inverse_hessian[dindex + 1]);
            }


            T tempval = 1.0 / hessian[dindex][dindex];

            for (size_t col = 0; col < nrows; col++) {
                hessian[dindex][col] *= tempval;
                inverse_hessian[dindex][col] *= tempval;
            }

            for (size_t row = (dindex + 1); row < nrows; ++row) {
                T wval = hessian[row][dindex];
                for (size_t col = 0; col < nrows; col = col + 1) {
                    hessian[row][col] -= wval * hessian[dindex][col];
                    inverse_hessian[row][col] -= wval * inverse_hessian[dindex][col];
                }
            }
        }

        for (long dindex = nrows - 1; dindex >= 0; --dindex) {
            for (long row = dindex - 1; row >= 0; --row) {
                T wval = hessian[row][dindex];
                for (size_t col = 0; col < nrows; col = col + 1) {
                    hessian[row][col] -= wval * hessian[dindex][col];
                    inverse_hessian[row][col] -= wval * inverse_hessian[dindex][col];
                }
            }
        }

    }


};
```

Now adding the function to actually do the minimization, we have an exact Newton minimizer:

```cpp
template<class T>
class MyFunctionMinimizer {
    ...

   bool Minimize(int max_iterations, T tolerance = 1e-4) {
        bool found = false;

        atl::Variable<T> f;
        atl::Variable<T>::SetRecording(true);
        gradient.resize(parameters.size());
        hessian.resize(parameters.size());
        inverse_hessian.resize(parameters.size());
        for (int i = 0; i < parameters.size(); i++) {
            hessian[i].resize(parameters.size());
            inverse_hessian[i].resize(parameters.size());
        }

        for (int iteration = 0; iteration < max_iterations; iteration++) {

            T max_gradient = std::numeric_limits<T>::min();
            bool all_positive = true;
            f = 0.0;
            atl::Variable<T>::gradient_structure_g.Reset();
            ObjectiveFunction(f);
            atl::Variable<T>::ComputeGradientAndHessian(
            atl::Variable<T>::gradient_structure_g,
                    parameters,
                    gradient,
                    hessian);
            //
            if ((iteration % 10) == 0) {
                std::cout << "Iteration: " << iteration << std::endl;
                std::cout << "Function value = " << f << "\n";
                std::cout << "Number of Parameters: " << parameters.size() << "\n";
                if (parameters.size() <= 10) {
                    std::cout << "Parameters = [";
                    for (int i = 0; i < parameters.size(); i++) {
                        std::cout << parameters[i]->GetValue() << " ";
                    }
                    std::cout << "]\n";
                    std::cout << "Gradient = " << gradient << "\n";
                    std::cout << "Hessian:\n" << hessian << "\n\n\n";
                }
            }
            for (int i = 0; i < gradient.size(); i++) {
                if (std::fabs(gradient[i]) > max_gradient) {
                    max_gradient = std::fabs(gradient[i]);
                }
                if (hessian[i][i] < 0) {
                    all_positive = false;
                }
            }
            if (max_gradient < tolerance && all_positive) {

                std::cout << "Successful Convergence!\n";
                std::cout << "Iteration: " << iteration << std::endl;
                std::cout << "Function value = " << f << "\n";
                std::cout << "Number of Parameters: " << parameters.size() << "\n";
                if (parameters.size() <= 10) {
                    std::cout << "Parameters = [";
                    for (int i = 0; i < parameters.size(); i++) {
                        std::cout << parameters[i]->GetValue() << " ";
                    }
                    std::cout << "]\n";
                    std::cout << "Gradient = " << gradient << "\n";
                    std::cout << "Hessian:\n" << hessian << "\n\n\n";
                }
                found = true;
                break;
            }

            if (parameters.size() > 1) {
                this->InvertHessian();

                for (int j = 0; j < parameters.size(); j++) {
                    T val = 0;
                    for (int k = 0; k < parameters.size(); k++) {
                        val += inverse_hessian[j][k] * gradient[k];
                    }
                    parameters[j]->SetValue(parameters[j]->GetValue() - val);
                }

            } else {
                parameters[0]->SetValue(parameters[0]->GetValue()
                                       - (gradient[0]) / hessian[0][0]);
            }
        }
        return found;
    }
...
};
```

Ok, lets test our function minimizer on a real function. In the field of gradient-based optimization, the Rosenbrock function is a non-convex function used to test the performance minimization algorithms and is defined by:

$$f(x_1, x_2, ..., x_m) = \sum_{i=1}^{i=m-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \tag{6}$$

We'll start by inheriting from the *MyFunctionMinimizer* class and overriding the function **void** ObjectiveFunction(atl::Variable<T>& f). We'll also add an additional function to initialize our Rosenbrock class. The initialization method will randomly choose starting points for our parameters and add them to the list of parameters that our minimizer will attempt to estimate.

```
 template<class T>
   class Rosenbrock : public MyFunctionMinimizer<T> {
   public:
       typedef atl::Variable<T> variable;
       std::vector<variable > x;

       void Initialize() {

           //number of estimable parameters
           int nobs = 5;

           //random seed
           srand(2015);

           //set some random starting values
           for (int i = 0; i < nobs; i++) {
               double r;
               r = ((double) rand() / (RAND_MAX)) + 1;
               this->x.push_back(variable(0.0 + 2.0 * r));
           }

           //register the estimable parameters with the function minimizer
           for (int i = 0; i < x.size(); i++) {
               this->parameters.push_back(&x[i]);
           }
       }

       //Rosenbrock Function
       void ObjectiveFunction(atl::Variable<T>& f) {
           f = 0;
           for (int i = 0; i < x.size() - 1; i++) {
               f += 100.0 * ((x[i + 1] - x[i] * x[i])*
                       (x[i + 1] - x[i] * x[i])) + (x[i] - 1.0)*(x[i] - 1.0);
           }
       }
   };



int main(int argc, char** argv) {

    Rosenbrock<double> rosenbrock;
    rosenbrock.Initialize();
    rosenbrock.Minimize(1000);

    return 0;
 }
```

Output:

```
Iteration: 0
Function value = 6387.77
Number of Parameters: 5
Parameters = [2.03154 2.09729 3.08945 2.36003 3.05197 ]
Gradient = [1.65157e+03 6.94495e+02  8.62100e+03  9.42607e+02  -5.03557e+02 ]
Hessian:
[4.11567e+03  -8.12616e+02 0.00000e+00  0.00000e+00  0.00000e+00  ]
[-8.12616e+02 4.24455e+03  -8.38914e+02 0.00000e+00  0.00000e+00  ]
[0.00000e+00  -8.38914e+02 1.07116e+04  -1.23578e+03 0.00000e+00  ]
[0.00000e+00  0.00000e+00  -1.23578e+03 5.66491e+03  -9.44013e+02 ]
[0.00000e+00  0.00000e+00  0.00000e+00  -9.44013e+02 2.00000e+02  ]



Iteration: 10
Function value = 5.07479e-10
Number of Parameters: 5
Parameters = [1.00000e+00 1.00000e+00 1.00001e+00 1.00001e+00 1.00002e+00 ]
Gradient = [1.75074e-05 6.19819e-05  2.44997e-04  6.03291e-04  -3.58108e-04 ]
Hessian:
[8.02002e+02  -4.00001e+02 0.00000e+00  0.00000e+00  0.00000e+00  ]
[-4.00001e+02 1.00200e+03  -4.00001e+02 0.00000e+00  0.00000e+00  ]
[0.00000e+00  -4.00001e+02 1.00201e+03  -4.00002e+02 0.00000e+00  ]
[0.00000e+00  0.00000e+00  -4.00002e+02 1.00202e+03  -4.00004e+02 ]
[0.00000e+00  0.00000e+00  0.00000e+00  -4.00004e+02 2.00000e+02  ]



Successful Convergence!
Iteration: 11
Function value = 1.30504e-17
Number of Parameters: 5
Parameters = [1.00000e+00 1.00000e+00 1.00000e+00 1.00000e+00 1.00000e+00 ]
Gradient = [8.04885e-10 2.83979e-09  1.14278e-08  4.20190e-08  -2.02847e-08 ]
Hessian:
[8.02000e+02  -4.00000e+02 0.00000e+00  0.00000e+00  0.00000e+00  ]
[-4.00000e+02 1.00200e+03  -4.00000e+02 0.00000e+00  0.00000e+00  ]
[0.00000e+00  -4.00000e+02 1.00200e+03  -4.00000e+02 0.00000e+00  ]
[0.00000e+00  0.00000e+00  -4.00000e+02 1.00200e+03  -4.00000e+02 ]
[0.00000e+00  0.00000e+00  0.00000e+00  -4.00000e+02 2.00000e+02  ]
```

As you can see from the above output, our function minimizer was able to find a solution in just 11 iterations. For this simple test, it would appear that our Newton minimizer is all that is needed for real world problems, however this is not the case. Inverting the Hessian matrix is an expensive operation for highly parameterized problems. It's often more efficient to just solve $[H(f(x_n)]p_n = \nabla f(x_n)$ as a system of linear equations. Another option is to turn to Quasi-Newton methods for function minimization. These methods do not use the inverse of the Hessian, but rather estimates it to find a search direction. Some of these methods are implemented in ATL and can be found in the *ATL/Optimization* package.

4 Containers

Currently there are two container types in ATL , Vector and Matrix. All of which are dense, with plans to implement sparse variants in later versions. ATL containers implement expression templates for operations just as the AutoDiff package does, which help to reduce temporary variables.

4.1 Vectors

ATL provides two dense, dynamic vector types called *atl::RealVector* and *atl::VariableVector*. Both use the same expression templates as *atl::Variable*. As the names imply, *atl::RealVector* is a vector of primitive data types, usually a floating type, while *atl::VariableVector* contains *atl::Variable* types. Future version will allow for static and sparse vector types. Vectors support all elementary operations (+,-.*,/), as well as all common math functions. A full listing of functions involving *atl::RealVector* and *atl::VariableVector* is available in Appendix A. Here is an example of using a *atl::RealVector*:

```
atl::RealVector<double> a = {1, 2, 3, 4, 5};
atl::RealVector<double> b = {6, 7, 8, 9, 10};
atl::RealVector<double> c = atl::log(a * b); //element wise multiplication of a * b

std::cout << "c = log(a*b) = " << a << " * " << b << " = " << c << "\n";

double d = atl::Dot(a, b);
std::cout <<"a dot b = " << d << "\n";
```

Output

```
c = log(a*b) = [ 1 2 3 4 5 ] * [ 6 7 8 9 10 ] = [ 1.79176 2.63906 3.17805 3.58352 3.91202 ]
a dot b = 130
```

In the above example, the operation "*" does not return a new *atl::Vector*, but instead returns an object called *atl::Multiply* and *atl::log* returns and object called *atl::Log* that operates on *atl::Multiply*, which are used to initialize *atl::Vector c*.

4.2 Matrices

ATL provides two dense, dynamic matrix type called *atl::VariableMatrix* and *atl::RealMatrix*. Future version will allow for static and sparse matrix types. Just as with Vectors, both use the same expression templates as *atl::Variable*. In addition, mixed types are also supported, with the large type being promoted. A full listing of functions involving *atl::VariableMatrix* is available in Appendix A. Here is an example of using a *atl::Matrix*:

```
    atl::VariableMatrix<double> A = {
    {1.0, 1.0, 0.0},
    {0.0, 0.0, 2.0},
    { 0.0, 0.0f, -1.0}
};
std::cout << "A = \n" << A << "\n";

atl::VariableMatrix<double> B = atl::exp(A);
std::cout << "B = exp(A) = \n" << B << "\n";

atl::VariableMatrix<double> C = atl::log(B);
std::cout << "C = log(B) = \n" << C << "\n";

atl::VariableMatrix<double> D = A*B;
std::cout << "D = A*B = \n" << D << "\n";

atl::VariableMatrix<double> row = D.Row(0);//atl::Row(D, 0);

std::cout << "D(0: ) = " << row << "\n";

atl::VariableMatrix<double> column = D.Column(0);//atl::Column(D, 0);

std::cout << "D(:0)= " << column << "\n";
```

Output

```
A =
1         1         0
0         0         2
0         0         -1

B = exp(A) =
2.71828   2.71828   1
1         1         7.38906
1         1         0.367879

C = log(B) =
1         1         0
0         0         2
0         0         -1

D = A*B =
3.71828   3.71828   8.38906
2         2         0.735759
-1        -1        -0.367879

D(0: ) = 3.71828  3.71828   8.38906

D(:0)=
3.71828
2
-1
```

5 Function Minimization

In general, function minimization can be defined as an effort to find the arguments of a function f that returns the smallest possible value. The function f may have many arguments and can be reffered to as a multidimensional minimization problem. To find the maximum of a function *f*, we redefine f as *-f*. There are several algorithms to find the minimum of a function using gradient information, these are reffered to as Newton and Quasi-Newton Methods.

As mentioned above, Newton's Method in optimization is an iterative procedure for finding the roots of a objective function that is differentiable. In a single parameter objective function, Newton's method attempts to converge to a stationary point using:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{7}$$

For objective functions with more than one parameter:

$$x_{n+1} = x_n - [Hf(x_n)]^{-1}\nabla f(x_n) \tag{8}$$

Quasi-Newton methods are alternatives to the Newton Method and are used to find the minimum of a function. Unlike the traditional Newton Method, which requires the Hessian matrix to find extrema, Quasi-Newton methods rely on a estimation of the Hessian for updates at each iteration. ATL has several Quasi-Newton algorithms available to a achieve the minimization of *f*.

5.1 Objective Function Objects

In ATL, all function minimizers have a member object pointer to an objective function in the form of a *atl::ObjectiveFunction*. User defined objective function objects should inherit from the class *atl::ObjectiveFunction*. There are two virtual functions that the user must overload, *Initialize()* and *Objective_Function(atl::Variable<T> & f)*. The first initializes the objective function. Here is where estimable parameters are registered. To register a parameter, the user simply calls the *RegisterParameter* function like so:

```
this->RegisterParameter(myparamter);
```

The member function *RegisterParameter* takes two arguments, first a pointer to the the objective function, the second is optional integer phase in which the registered parameter will be estimated. The default value is 1, for the first phase.

The *Objective_Function(atl::Variable<T> & f)* is where the work is done and f is the resulting value of the objective function.

```
void Objective_Function(atl::Variable<T>& f){

}
```

Here is a simple example of defining an objective function in ATL:

```
template<class T>
class Simple : public atl::ObjectiveFunction<T> {
    int nobs = 10;
    atl::RealVector<T> x = {-1, 0, 1, 2, 3, 4, 5, 6, 7, 8};
    atl::RealVector<T> y = {1.4, 4.7, 5.1, 8.3, 9.0, 14.5, 14.0, 13.4, 19.2, 18};
    atl::Variable<T> a;
    atl::Variable<T> b;
    atl::VariableVector<T> pred_y;
public:

    void Initialize() {

        a.SetName("a");
        this->RegisterParameter(a);

        b.SetName("b");
        this->RegisterParameter(b);

    }

    void Objective_Function(atl::Variable<T>& f) {
        pred_y =((a * x + b) - y);
        f = nobs / 2.0 * atl::log(atl::Norm2(pred_y));
    }


};
```

The above code listing is a **ATL** port of an example from **ADMB** (Fournier, et al) and is a simple linear regression problem.

EXPAND ME

5.2 Available Quasi-Newton Methods

5.2.1 BFGS

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is a quasi-Newton iterative method for solving nonlinear optimization problems. Given an initial guess x_0 and an approximate *Hessian* B_0, we have the following:

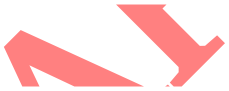
Algorithm 4 BFGS

```
1: while $|\nabla f(x)| <= \text{tol}$ do
2:     Find direction $p_0$ by solving $B_k P_k = -\nabla f$
3:     Use a line search to find step size $a_k$, $a_k = \arg\min f(x_k + ap_k)$
4:     $s_k = a_k p_k$,
5:     $x_{k+1} = x_k + s_k$
6:     $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$
7:     $B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$
8: end while
```

Convergence is obtained when $|\nabla f(x)|$ less than or equal to the user supplied tolerance, the default is 0.0001 for all function minimizers in **ATL**. To use the **BFGS** algorithm:

```
int main() {

    //create and initialize the objective function
    Simple<double> s;
    s.Initialize();

    //create an instance of bfgs minimizer
    atl::BFGS<double> minimizer;

    //set a pointer to the objective function
    minimizer.SetObjectiveFunction(&s);

    //run the minimizer
    minimizer.Run();
 }
```

Output



```
Iteration: 0
Phase: 1
Function Value = 21.3286
Max Gradient Component: 1.94738
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |             Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              a                |  2.5323e+00| 4.0349e-01|              *b               |  5.1015e-01| -1.9474e+00|
 ------------------------------------------------------------------------------------------------------------------------


Iteration: 1
Phase: 1
Function Value = 2.1083e+01
Max Gradient Component: 4.1613e+00
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |             Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              *a               |  2.6111e+00| 4.1613e+00|              b                |  7.7253e-01| -1.2516e+00|
 ------------------------------------------------------------------------------------------------------------------------


Iteration: 6
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 4.3262e-10
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |             Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              *a               |  1.9091e+00| -4.3262e-10|             b                |  4.0782e+00| -2.3870e-11|
 ------------------------------------------------------------------------------------------------------------------------
```

5.2.2 L-BFGS

The Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm is another quasi-Newton iterative method for solving nonlinear optimization problems. Given an initial guess x_0, we have the following:

Algorithm 5 L-BFGS

```
for $k = 0, 1, 2...$ do
    $g_k = -\nabla f(x_k)$
    if $|\nabla f(x)| <=$ tol then exit
    end if
    for $i = k-1, k-2, k-m$ do
        $\rho_i = \frac{1}{y_k^T s_k}$
        $\alpha_i = \rho_i s_i^T p_k$
        $p_k = p_k - \alpha_i y_i$
    end for
    Find direction $\alpha_k$ that minimizes $|f(x_k + \alpha_k p_k)|$
    $x_k + 1 = x_k + \alpha_k p_k$

    $s_k = \alpha_k p_k$

    $y_k = \nabla f_{k+1} - \nabla f_k$

end for
```

ATL offers two versions of the L-BFGS algorithm, *atl::LBFGS* and *atl::LBFGS2*. The Difference in the two is in the line search routine. To use *atl::LBFGS* :

```
int main() {

    //create and initialize the objective function
    Simple<double> s;
    s.Initialize();

    //create an instance of l-bfgs minimizer
    atl::LBFGS<double> minimizer;

    //set a pointer to the objective function
    minimizer.SetObjectiveFunction(&s);

    //run the minimizer
    minimizer.Run();
 }
```

```
Iteration 0
Phase = 1
Iteration: 0
Phase: 1
Function Value = 36.4936
Max Gradient Component: 3.61269
Floating-Point Type: Float64
Number of Parameters: 2
 -------------------------------------------------------------------------------------------------------------------------------
|             Name                |  Value    |  Gradient |             Name                |  Value    |  Gradient |
 -------------------------------------------------------------------------------------------------------------------------------
|              *a                 | 0.0000e+00| -3.6127e+00|              b                  | 0.0000e+00| -7.2781e-01|
 -------------------------------------------------------------------------------------------------------------------------------


Iteration 10
Phase = 1
Iteration: 10
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 1.3612e-01
Floating-Point Type: Float64
Number of Parameters: 2
 -------------------------------------------------------------------------------------------------------------------------------
|             Name                |  Value    |  Gradient |             Name                |  Value    |  Gradient |
 -------------------------------------------------------------------------------------------------------------------------------
|              *a                 | 1.9090e+00| 1.3612e-01|              b                  | 4.0864e+00| 3.9884e-02|
 -------------------------------------------------------------------------------------------------------------------------------


Iteration 12
Phase = 1
Iteration: 12
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 4.1663e-05
Floating-Point Type: Float64
Number of Parameters: 2
 -------------------------------------------------------------------------------------------------------------------------------
|             Name                |  Value    |  Gradient |             Name                |  Value    |  Gradient |
 -------------------------------------------------------------------------------------------------------------------------------
|              *a                 | 1.9091e+00| 4.1663e-05|              b                  | 4.0782e+00| 3.1493e-07|
 -------------------------------------------------------------------------------------------------------------------------------
```

To use *atl::LBFGS2* :

```cpp
int main() {

    //create and initialize the objective function
    Simple<double> s;
    s.Initialize();

    //create an instance of l-bfgs minimizer
    atl::LBFGS2<double> minimizer;

    //set a pointer to the objective function
    minimizer.SetObjectiveFunction(&s);

    //run the minimizer
    minimizer.Run();
 }
```

```
Iteration: 0
Phase: 1
Function Value = 21.3286
Max Gradient Component: 1.94645
Floating-Point Type: Float64
Number of Parameters: 2
 ----------------------------------------------------------------------------------------------------------------------------
|              Name               |  Value    |  Gradient |               Name               |  Value    |  Gradient |
 ----------------------------------------------------------------------------------------------------------------------------
|               a                 | 2.5325e+00| 4.0881e-01|               *b                 | 5.1019e-01| -1.9465e+00|
 ----------------------------------------------------------------------------------------------------------------------------


Iteration: 1
Phase: 1
Function Value = 2.1076e+01
Max Gradient Component: 4.2001e+00
Floating-Point Type: Float64
Number of Parameters: 2
 ----------------------------------------------------------------------------------------------------------------------------
|              Name               |  Value    |  Gradient |               Name               |  Value    |  Gradient |
 ----------------------------------------------------------------------------------------------------------------------------
|              *a                 | 2.6112e+00| 4.2001e+00|                b                 | 7.7816e-01| -1.2442e+00|
 ----------------------------------------------------------------------------------------------------------------------------


Iteration: 5
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 2.1320e-04
Floating-Point Type: Float64
Number of Parameters: 2
 ----------------------------------------------------------------------------------------------------------------------------
|              Name               |  Value    |  Gradient |               Name               |  Value    |  Gradient |
 ----------------------------------------------------------------------------------------------------------------------------
|              *a                 | 1.9091e+00| 2.1320e-04|                b                 | 4.0782e+00| 1.1964e-04|
 ----------------------------------------------------------------------------------------------------------------------------
```

5.2.3 Conjugate Gradient Descent

The Conjugate Gradient Descent routine uses the computation of conjugate directions in an attempt to minimize an objective function *f*. Given an initial guess x_0, we have:

Algorithm 6 CGD

```
1: for $k = 0, 1, 2...$ do
2:     $g = \nabla f(x_k)$
3:     $s = -g$
4:     if $|g| <=$ tol then exit
5:     end if
6:     if k>0 then
7:         $beta = \frac{g \cdot g}{g_{old} \cdot g_{old}}$
8:         $s+ = beta * s_{old}$
9:     end if
10:    Find direction $\alpha_k$ that minimizes $|f(x_k + \alpha_k s)|$
11:    $x_k + 1 = x_k + \alpha_k s$
12:    $g_{old} = g$
13:    $s_{old} = s$
14: end for
```

To use *atl::ConjugateGradientDescent*:

```
int main() {

   //create and initialize the objective function
   Simple<double> s;
   s.Initialize();

   //create an instance of cgd minimizer
   atl::ConjugateGradientDescent<double> minimizer;

   //set a pointer to the objective function
   minimizer.SetObjectiveFunction(&s);

   //run the minimizer
   minimizer.Run();
 }
```

Output

```
 Iteration: 0
Phase: 1
Function Value = 36.4936
Max Gradient Component: 3.61269
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name               |  Value   |  Gradient |             Name               |  Value   |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              *a                | 0.0000e+00| -3.6127e+00|             b                | 0.0000e+00| -7.2781e-01|
 ------------------------------------------------------------------------------------------------------------------------


Iteration: 1
Phase: 1
Function Value = 2.8959e+01
Max Gradient Component: 7.0807e+00
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name               |  Value   |  Gradient |             Name               |  Value   |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              *a                | 3.6127e+00| 7.0807e+00|             b                | 7.2781e-01| 7.9735e-01|
 ------------------------------------------------------------------------------------------------------------------------


Iteration: 10
Phase: 1
Function Value = 1.5436e+01
Max Gradient Component: 2.3743e+00
Floating-Point Type: Float64
Number of Parameters: 2
 ------------------------------------------------------------------------------------------------------------------------
|             Name               |  Value   |  Gradient |             Name               |  Value   |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              *a                | 1.7576e+00| 2.3743e+00|             b                | 4.6971e+00| 1.7437e+00|
 ------------------------------------------------------------------------------------------------------------------------
```

...

```
Iteration: 40
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 4.1455e-03
Floating-Point Type: Float64
Number of Parameters: 2
 ----------------------------------------------------------------------------------------------------------------------------
|             Name                |  Value    |  Gradient |             Name                |  Value    |  Gradient |
 ----------------------------------------------------------------------------------------------------------------------------
|              *a                 | 1.9090e+00| 4.1455e-03|               b                 | 4.0783e+00| 6.8011e-05|
 ----------------------------------------------------------------------------------------------------------------------------


Iteration: 50
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 7.8198e-05
Floating-Point Type: Float64
Number of Parameters: 2
 ----------------------------------------------------------------------------------------------------------------------------
|             Name                |  Value    |  Gradient |             Name                |  Value    |  Gradient |
 ----------------------------------------------------------------------------------------------------------------------------
|              *a                 | 1.9091e+00| 7.8198e-05|               b                 | 4.0782e+00| 5.6322e-05|
 ----------------------------------------------------------------------------------------------------------------------------
```

5.2.4 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function and steps are found by using the negative of the gradient. Given an initial guess x_0, we have:

Algorithm 7 GD

```
1: for $k = 0, 1, 2...$ do
2:     $g = \nabla f(x_k)$
3:     $s = -g$
4:     if $|g| <=$ tol then exit
5:     end if
6:     Find direction $\alpha_k$ that minimizes $|f(x_k + \alpha_k s)|$
7:     $x_{k+1} = x_k - \alpha_k g$
8: end for
```

To use *atl::GradientDescent*:

```
int main() {

    //create and initialize the objective function
    Simple<double> s;
    s.Initialize();

    //create an instance of gd minimizer
    atl::GradientDescent<double> minimizer;

    //set a pointer to the objective function
    minimizer.SetObjectiveFunction(&s);

    //run the minimizer
    minimizer.Run();
 }
```

Output:

```
Iteration: 0
Phase: 1
Function Value = 21.3286
Max Gradient Component: 1.94738
Floating-Point Type: Float64
Number of Parameters: 2
 --------------------------------------------------------------------------------------------------------------------------------
|             Name               |   Value    |  Gradient |              Name               |   Value    |  Gradient |
 --------------------------------------------------------------------------------------------------------------------------------
|              a                 |  2.5323e+00|  4.0349e-01|              *b                 |  5.1015e-01|  -1.9474e+00|
 --------------------------------------------------------------------------------------------------------------------------------


Iteration: 1
Phase: 1
Function Value = 1.5574e+01
Max Gradient Component: 9.8927e+00
Floating-Point Type: Float64
Number of Parameters: 2
 --------------------------------------------------------------------------------------------------------------------------------
|             Name               |   Value    |  Gradient |              Name               |   Value    |  Gradient |
 --------------------------------------------------------------------------------------------------------------------------------
|             *a                 |  1.8348e+00|  -9.8927e+00|              b                  |  3.8762e+00|  -2.0497e+00|
 --------------------------------------------------------------------------------------------------------------------------------


Iteration: 10
Phase: 1
Function Value = 1.4964e+01
Max Gradient Component: 1.3732e-05
Floating-Point Type: Float64
Number of Parameters: 2
 --------------------------------------------------------------------------------------------------------------------------------
|             Name               |   Value    |  Gradient |              Name               |   Value    |  Gradient |
 --------------------------------------------------------------------------------------------------------------------------------
|              a                 |  1.9091e+00|  1.8635e-06|              *b                 |  4.0782e+00|  -1.3732e-05|
 --------------------------------------------------------------------------------------------------------------------------------
```

6 Common Discontinuous Functions And Some Helpful Hacks

6.1 The Absolute Value Function

When using the function *atl::fabs*, ATL will return a derivative value of *nan* when the argument is zero. This is because the analytical derivative of the absolute value function is undefined at zero. As an alternative, we can use complex step differentiation and adjoint entries to get around this problem. Below is an example of how to accomplish this.

```
template<class REAL_T, class EXPR>
atl::Variable<REAL_T> ad_fabs(const atl::ExpressionBase<REAL_T, EXPR>& expr) {

    //set x
    atl::Variable<REAL_T> x = expr;

    //compute the absolute value of x
    atl::Variable<REAL_T> abs_x = std::fabs(x.GetValue());

    //use complex step automatic differentiation to estimate
    //the derivative
    REAL_T H = 1e-20;
    std::complex<REAL_T> cx(x.GetValue(), H);
    REAL_T dx;

    //if the value of x is zero, set dx to zero
    dx = x.GetValue() != static_cast<REAL_T>(0.0) ? std::sqrt(cx * cx).imag() / H : static_cast<REAL_T>(0.0) ;

    //Create the adjoint entry
    atl::StackEntry<REAL_T> entry = atl::Variable<REAL_T>::tape.NextEntry();
    entry.w = abs_x.info;
    entry.ids.insert(x.info);
    entry.first.push_back(dx);

    return abs_x;
}
```

The above code listing will return a derivative value of -1 for an argument less than 0 and 1 for an argument greater than 0. When the argument is zero, the derivative value is also zero.

6.2 Min and Max Functions

Given two variables a and b, calculate the minimum or maximum value. Existing implementations can involve conditional operations, i.e., "if" statements are used to determine the return value:

```
template<typename T>
T min(const T& a, const T& b){
   return a < b ? a : b;
}

template<typename T>
T max(const T& a, const T& b){
   return a > b ? a : b;
}
```

These functions do not work in an automatic differentiation system since the functions are not differentiable everywhere given the branching in the code.
A common branchless alternative can be used:

```
template<typename T>
T min(const T& a, const T& b){
   return (a + b - fabs(a - b)) / 2.0;
}

template<typename T>
T max(const T& a, const T& b){
   return (a + b + fabs(a - b)) / 2.0;
}
```

However, if the argument *a-b* for *fabs* is zero, the function is undefined. If this could be the case, we can redefine the above code listing to use the *ad_fabs* function for auto diff types as:

```
template<class REAL_T, class EXPR, class EXPR2>
atl::Variable<REAL_T> ad_min(const atl::ExpressionBase<REAL_T, EXPR>& a,
                                  const atl::ExpressionBase<REAL_T, EXPR2>& b){
   return (a + b - ad_fabs(a - b)) / 2.0;
}

template<class REAL_T, class EXPR, class EXPR2>
atl::Variable<REAL_T> ad_max(const atl::ExpressionBase<REAL_T, EXPR>& a,
                                   const atl::ExpressionBase<REAL_T, EXPR2>& b){
   return (a + b + ad_fabs(a - b)) / 2.0;
}
```

Now we have *min* and *max* functions that are not subject to undefined behavior.

7 Installation

7.1 Requirements

8 Future Work

8.1 More Efficient Random Effects Modeling

8.2 MPI and Multithreading

9 Appendices

9.1 ATL Examples

9.1.1 Von Bertalanffy Growth

```
template<class T>
class VonBertalanffy : public atl::ObjectiveFunction<T> {
    int nobs = 20;

    atl::RealVector<T> a = {1,
        2,
        3,
        4,
        5,
        6,
        7,
        8,
        9,
        10,
        11,
        12,
        13,
        14,
        15,
        16,
        17};

    atl::RealVector<T> L = {6.96,
        31.8,
        23.9,
        26.1,
        33.4,
        36.2,
        38.7,
        43.3,
        44.6,
        45.8,
        40.1,
        47.9,
        52.2,
        27.9,
        52.6,
        56.9,
        57.1};

    atl::Variable<T> t0;
    atl::Variable<T> Linf;
    atl::Variable<T> k;
    atl::Variable<T> sd;
    atl::VariableVector<T> Lpred;

public:
    void Initialize() {

        t0 = 0.0;
        t0.SetName("t0");
        this->RegisterParameter(t0);

        Linf = 21;
        Linf.SetName("Linf");
        this->RegisterParameter(Linf);

        k = 0.1;
        k.SetName("k");
        this->RegisterParameter(k);

        sd.SetName("sd");
        sd.SetBounds(0.1, 10.0);
        sd = 1.0;
        this->RegisterParameter(sd, 2);
    }

    void Objective_Function(atl::Variable<T>& f) {
        Lpred = Linf * (1.0 - atl::exp((-k)*(a - t0)));
        f = T(nobs) * atl::log(sd) +
                atl::Sum(atl::VariableVector<T>(0.5 * atl::pow((atl::log(L) - atl::log(Lpred)) / sd, 2.0)));
    }
};

int main() {

    VonBertalanffy<double> s;
    s.Initialize();

    atl::LBFGS<double> minimizer;
    minimizer.SetPrintWidth(2);
    minimizer.SetObjectiveFunction(&s);
    minimizer.Run();
}
```

Output:

```
Iteration 0
Phase = 1
Iteration: 0
Phase: 1
Function Value = 14.4968
Max Gradient Component: 142.749
Floating-Point Type: Float64
Number of Parameters: 3
 ------------------------------------------------------------------------------------------------------------------------------
|            Name              |  Value    | Gradient |             Name              |  Value    | Gradient |
 ------------------------------------------------------------------------------------------------------------------------------
|             t0               |  0.0000e+00| 3.8710e+00|           Linf              |  2.1000e+01| -1.0320e+00|
|             *k               |  1.0000e-01| -1.4275e+02|
 ------------------------------------------------------------------------------------------------------------------------------


Iteration 10
Phase = 1
Iteration: 10
Phase: 1
Function Value = 1.9993e+00
Max Gradient Component: 8.8555e-04
Floating-Point Type: Float64
Number of Parameters: 3
 ------------------------------------------------------------------------------------------------------------------------------
|            Name              |  Value    | Gradient |             Name              |  Value    | Gradient |
 ------------------------------------------------------------------------------------------------------------------------------
|             *t0              |  1.8287e+01| -8.8555e-04|           Linf              |  3.5863e+01| -4.8470e-04|
|              k               | -6.2980e+00| 1.8125e-04|
 ------------------------------------------------------------------------------------------------------------------------------


Iteration 14
Phase = 1
Iteration: 14
Phase: 1
Function Value = 1.9991e+00
Max Gradient Component: 8.2576e-05
Floating-Point Type: Float64
Number of Parameters: 3
 ------------------------------------------------------------------------------------------------------------------------------
|            Name              |  Value    | Gradient |             Name              |  Value    | Gradient |
 ------------------------------------------------------------------------------------------------------------------------------
|             t0               |  1.8683e+01| -6.1531e-05|           *Linf             |  3.5905e+01| 8.2576e-05|
|              k               | -6.4116e+00| 1.6163e-05|
 ------------------------------------------------------------------------------------------------------------------------------


Iteration 0
Phase = 2
Iteration: 0
Phase: 2
Function Value = 1.9991e+00
Max Gradient Component: 1.6002e+01
Floating-Point Type: Float64
Number of Parameters: 4
 ------------------------------------------------------------------------------------------------------------------------------
|            Name              |  Value    | Gradient |             Name              |  Value    | Gradient |
 ------------------------------------------------------------------------------------------------------------------------------
|             t0               |  1.8683e+01| -6.1531e-05|           Linf              |  3.5905e+01| 8.2576e-05|
|              k               | -6.4116e+00| 1.6163e-05|            *sd              |  1.0000e+00| 1.6002e+01|
 ------------------------------------------------------------------------------------------------------------------------------


Iteration 10
Phase = 2
Iteration: 10
Phase: 2
Function Value = -6.0988e+00
Max Gradient Component: 1.6164e-02
Floating-Point Type: Float64
Number of Parameters: 4
 ------------------------------------------------------------------------------------------------------------------------------
|            Name              |  Value    | Gradient |             Name              |  Value    | Gradient |
 ------------------------------------------------------------------------------------------------------------------------------
|             t0               |  1.8688e+01| -2.9745e-04|           Linf              |  3.5899e+01| -2.0980e-05|
|              k               | -6.4129e+00| 7.8349e-05|            *sd              |  4.4720e-01| 1.6164e-02|
 ------------------------------------------------------------------------------------------------------------------------------
```

```
Iteration 20
Phase = 2
Iteration: 20
Phase: 2
Function Value = -6.0988e+00
Max Gradient Component: 3.1933e-02
Floating-Point Type: Float64
Number of Parameters: 4
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |               Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              t0               |  1.8919e+01| -6.0707e-05|             Linf              |  3.5903e+01| 2.7580e-04|
|               k               | -6.4738e+00| 1.8007e-05|              *sd              |  4.4696e-01| -3.1933e-02|
 ------------------------------------------------------------------------------------------------------------------------


Iteration 30
Phase = 2
Iteration: 30
Phase: 2
Function Value = -6.0988e+00
Max Gradient Component: 9.1994e-05
Floating-Point Type: Float64
Number of Parameters: 4
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |               Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              t0               |  1.9795e+01| -1.1332e-07|             Linf              |  3.5899e+01| -8.7510e-06|
|               k               | -6.7047e+00| 4.7261e-08|              *sd              |  4.4711e-01| -9.1994e-05|
 ------------------------------------------------------------------------------------------------------------------------


Iteration 30
Phase = 2
Iteration: 30
Phase: 2
Function Value = -6.0988e+00
Max Gradient Component: 9.1994e-05
Floating-Point Type: Float64
Number of Parameters: 4
 ------------------------------------------------------------------------------------------------------------------------
|             Name              |   Value    |  Gradient |               Name              |   Value    |  Gradient |
 ------------------------------------------------------------------------------------------------------------------------
|              t0               |  1.9795e+01| -1.1332e-07|             Linf              |  3.5899e+01| -8.7510e-06|
|               k               | -6.7047e+00| 4.7261e-08|              *sd              |  4.4711e-01| -9.1994e-05|
 ------------------------------------------------------------------------------------------------------------------------
```

9.2 Software Requirements

9.3 Compiling ATL Applications

10 References