The Analytics Template Library

A Developer's Guide

Matthew Supernaw
NOAA Fisheries

Version 1.0
August 2015

Contents

1 Introduction

The Analytics Template Library (ATL) is a generic scientific computing library that leverages the power of template metaprogramming for flexibility 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:

```
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 is 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.

```
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:

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

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)***
$x_2' = 0.0$
———
$g(x_1, x_2)' = x_1' x_2 + x_1 x_2' = 1.0 * 2.0 + 3.1459 * 0 = 2.0$
$f(g(x_1, x_2))' = \frac{g(x_1, x_2)'}{g(x_1, x_2)} = \frac{2.0}{2.0 * 3.1459} = 0.317874$

Compute: x_2'
Tape
$x_1' = 0.0$
$x_2' = 1.0$ ***(seed)***
———
$g(x_1, x_2)' = x_1' x_2 + x_1 x_2' = 0.0 * 2.0 + 3.1459 * 1.0 = 3.1459$
$f(g(x_1, x_2))' = \frac{g(x_1, x_2)'}{g(x_1, x_2)} = \frac{3.1459}{2.0 * 3.1459} = 0.5$

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: $\quad \hat{w}[i] = 1.0$
4: $\quad$ **for** $j = 1$ to n **do**
5: $\quad\quad Tape[j] \rightarrow Evaluate$
6: $\quad$ **end for**
7: $\quad \nabla f[i] = Tape[n] \rightarrow Value$
8: $\quad \hat{w}[i] = 0.0$
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: $\quad \frac{df}{dx_i} = \bar{w}[i]$
6: $\quad \bar{w}[i] = 0$
7: $\quad$ **for** $j = 1$ to i **do**
8: $\quad\quad \bar{w}[j] += \frac{\partial f}{\partial x_i} \bar{w}[j]$
9: $\quad$ **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*

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

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

$\bar{w}[m] = 1$

for $i = m$ to 1 **do**

  $\frac{df}{dx_i} = \bar{w}[i]$

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

  for $j = 1$ to i **do**

    $\bar{w}[j] += \frac{\partial f}{\partial x_i}\bar{w}[i]$

    for $k = 1$ to i **do**

      $\bar{h}[j][k] += \bar{h}[i][k]\frac{\partial f}{\partial x_j} + \bar{h}[i][j]\frac{\partial f}{\partial x_k} + \bar{h}[i][i]\frac{\partial f}{\partial x_k}\frac{\partial f}{\partial x_j} + w\frac{\partial^2 f}{\partial x_j \partial x_k}$

    end for

  end for

end for

Output:

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

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

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


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

    //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";

    //compute and extract the gradient and hessian
    variable::ComputeGradientAndHessian(
    variable::gradient_structure_g, //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:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]
```

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;

    //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::gradient_structure_g.HessianAndGradientAccumulate();

    std::cout <<std::scientific<< "gradient:\n" << x1.info->dvalue << "
" << x2.info->dvalue << "\n";
    std::cout << "hessian:\n" << x1.info->hessian_row[x1.info] << "
" << x1.info->hessian_row[x2.info] << "\n";
    std::cout << x2.info->hessian_row[x1.info] << " " << x2.info->hessian_row[x2.info] << "\n
 }
```

```
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 *GradientStructure*

After you have computed derivatives, you can reset the gradient structure (Tape) by calling **void** Reset() on the instance of GradientStructure. 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.

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

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::ComputeGradientAndHessian(
   variable::gradient_structure_g, //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 gradient structure
    variable_t::gradient_structure_g.Reset();

}
```

```
f = 1.83925

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]
```

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

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



/// 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::gradient_structure_g.SetRecording(false);
    } else {
        std::cout << "Recording\n";
        variable_t::gradient_structure_g.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::ComputeGradientAndHessian(
   variable::gradient_structure_g, //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::gradient_structure_g.Reset();

}
```

Output

```
Not Recording
f = 1.83925

gradient:
[0.00000e+00   0.00000e+00   ]

Hessian:
[0.00000e+00   0.00000e+00   ]
[0.00000e+00   0.00000e+00   ]


Recording
f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


Not Recording
f = 1.83925e+00

gradient:
[0.00000e+00   0.00000e+00   ]

Hessian:
[0.00000e+00   0.00000e+00   ]
[0.00000e+00   0.00000e+00   ]


Recording
f = 1.83925e+00

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]


Not Recording
f = 1.83925e+00

gradient:
[0.00000e+00   0.00000e+00   ]

Hessian:
[0.00000e+00   0.00000e+00   ]
[0.00000e+00   0.00000e+00   ]
```

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

3.4.3 Using *GradientStructure*'s Other Than The Default

To use an instance of *GradientStructure* 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 *GradientStructure* other than the default:

```
typedef double real_t;
typedef atl::Variable<real_t> variable_t;
//create an instance of GradientStructure
atl::GradientStructure<real_t> gradient_structure;



/// 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 GradientStructure
    f.Assign(gradient_structure, atl::log(x1 * x2));

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

    //extract the gradient and hessian from our instance of GradientStructure
     variable::ComputeGradientAndHessian(
    gradient_structure,    //other 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";
    gradient_structure.Reset();
```

Output

```
f = 1.83925

gradient:
[3.17874e-01   5.00000e-01   ]

Hessian:
[-1.01044e-01  0.00000e+00   ]
[0.00000e+00   -2.50000e-01  ]
```

Having this capability is particularly useful when one wishes to run multiple models concurrently. Also, even though entries in the *GradientStructure* are thread safe, they use atomic operations, which can result in cache contention between threads. Furthermore, threads sharing the same *GradientStructure* must not have shared dependency as this will result in errors in the accumulation of derivatives. Using a separate *GradientStructure* in each thread, computing the derivatives and adding them to the main instance of *GradientStructure* 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::vector<REAL_T> gradient(2);
    gradient[0] = std::cos(x.GetValue()) * cos(y.GetValue());
    gradient[1] = -1.0 * std::sin(x.GetValue()) * std::sin(y.GetValue());

    //set the Hessian values for this function
    std::vector<std::vector<REAL_T> > hessian(2, std::vector<REAL_T>(2));
    hessian[0][0] = -1.0 * std::sin(x.GetValue()) * std::cos(y.GetValue());
    hessian[0][1] = -1.0 * std::cos(x.GetValue()) * std::sin(y.GetValue());
    hessian[1][0] = -1.0 * std::cos(x.GetValue()) * std::sin(y.GetValue());
    hessian[1][1] = -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>::gradient_structure_g.NextEntry(),
           ret,
           independents,
           gradient,
           hessian);

    //return a Variable
    return ret;
}

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

    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>::gradient_structure_g,
           independents,
           gradient,
           hessian);
```

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

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

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

    atl::Variable<double>::ComputeGradientAndHessian(
           atl::Variable<double>::gradient_structure_g,
           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:
gradient[-7.07371e-02 -1.58866e-03 ]
Hessian:
[-1.12660e-04  9.97494e-01  ]
[9.97494e-01  -1.12660e-04 ]

Result from computed entry:
gradient[-7.07371e-02 -1.58866e-03 ]
Hessian:
[-1.12660e-04  9.97494e-01  ]
[9.97494e-01  -1.12660e-04 ]
```

In the above example, we computed all the necessary derivatives to supply our main *GradientStructure* with an adjoint entry. The result is a reduced amount of entries into our main *GradientStructure*. In the *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:

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

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:

```
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 the *FunctionMinimizer* class found in the *ATL/Optimization* package.

4 Containers

Currently there are three container types in ATL, Array, Vector, 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, however one must be mindful of stack overflows. Large expressions can cause a stack overflow and the user will not receive an error as to why the program terminated. Furthermore, if your program is compiled with $-DPROMOTE\_BINARY\_OPERATORS$, container types that have a *atl::Variable* for a base will have operators that return expression templates rather than a value, which will increase the chance for a stack overflow because you have an expression template operating on expression templates. That said, compiling with $-DPROMOTE\_BINARY\_OPERATORS$ can lead to faster container operations for containers operating on *atl::Variable* types.

4.1 Vectors

ATL provides one dense, dynamic vector type called *atl::Vector*. 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::Vector* is available in Appendix A. Here is an example of using a *atl::Vector*:

```
atl::Vector<float> a = {1, 2, 3, 4, 5};
atl::Vector<double> b = {6, 7, 8, 9, 10};
atl::Vector<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 *VectorMultiply* and *atl::log* returns and object called *VectorLog* that operates on *VectorMultiply*, which are used to initialize *atl::Vector c*. Also, notice the *atl::Vector a* has a base type of *float*, ATL container support mixed precision with the higher being promoted. In this case, a*b returns an instance of *VectorMultiply* that has a base type of *double*.

4.2 Matrices

ATL provides one dense, dynamic matrix type called *atl::Matrix*. Future version will allow for static and sparse matrix types. The *atl::Matrix* class supports all elementary operations (+,-.*,/), as well all common math functions. In addition, mixed types are also supported, with the large type being promoted. A full listing of functions involving *atl::Matrix* is available in Appendix A. Here is an example of using a *atl::Matrix*:

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

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

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

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

atl::Vector<atl::Variable<double> > row = atl::Row(D, 0);

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

atl::Vector<atl::Variable<double> > column = atl::Column(D, 0);

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

Output

```
A =
[      1      1      0 ]
[      0      0      2 ]
[      0      0     -1 ]

B = exp(A) =
[  2.718  2.718      1 ]
[      1      1  7.389 ]
[      1      1 0.3679 ]

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

D = A*B =
[  3.718  3.718  8.389 ]
[      2      2 0.7358 ]
[     -1     -1 -0.3679 ]

D(0: ) = [ 3.718 3.718 8.389 ]

D(:0)= [ 3.718 2 -1 ]
```

4.3 Arrays

Arrays are basic containers in ATL and can have up to seven dimensions. Simply put, *atl::Array* are basic multidimensional containers to hold data. Just like the *Vector* and *Matrix* classes, the *Array* class supports all elementary operations (+,-.*,/), as well as all common math functions. Here is an example of initializing an instance of *atl::Array* of four dimensions using an initializer list:

```
atl::Array<double> a = {
     {
         {
             {1, 2, 3},
             {4, 5, 6}
         },
         {
             {7, 8, 9},
             {10, 11, 12}
         }
     },
     {
         {
             {13, 14, 15},
             {16, 17, 18}
         },
         {
             {19, 20, 21},
             {22, 23, 24}
         }
     }
  };

  std::cout << "Array a has " << a.Dimensions() << " dimensions" << std::endl;
  std::cout << "Array a = " << a.Size(0) << " x " << a.Size(1)
              << " x " << a.Size(2) << " x " << a.Size(3) << std::endl;
  for (int i = 0; i < a.Size(0); i++) {
      for (int j = 0; j < a.Size(1); j++) {
          for (int k = 0; k < a.Size(2); k++) {
              for (int l = 0; l < a.Size(3); l++) {
                  std::cout << a(i, j, k, l) << " ";
              }
              std::cout << "\n";
          }
      }
  }
```

Output

```
Array a has 4 dimensions
Array a = 2 x 2 x 2 x 3
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
19 20 21
22 23 24
```

4.4 Type Promotion

ATL containers have the ability to handle type promotion. Consider the class *Vector*, it can be instantiated like:

```
atl::Vector<atl::Variable<double> > a; // Vector of AD types
atl::Vector<double> b;                 // Vector of double precision types
atl::Vector<int> c;                        // Vector of integer types
```

What would happen if a and b are multiplied together? Obviously the resulting *Vector* should be *atl::Vector<atl::Variable<double> >*. One way to ensure this is the case would be to code specialized versions of operator* for each possible case:

```
atl::Vector<atl::Variable<double> > operator*(
     const atl::Vector<atl::Variable<double> >& a, const atl::Vector<double>& b)
{
    ...
}
```

This would clearly become cumbersome to code every possible combination of operator*. To get around this, we turn to a template metaprogramming technique called type promotion. So instead of coding every possible combination of operator*, we simply create a class to handle the promotion:

```
template<typename T1, typename T2>
struct PromoteType {
     typedef T1 return_type;
};
```

Now we can create a simple specialized versions of *struct PromoteType* to handle the desired promotion:

```
template<atl::Vector<double>, double>
struct PromoteType {
     typedef atl::Vector<double> return_type;
};

template<double, int>
struct PromoteType {
     typedef double return_type;
};
...
```

Using this concept we can write one *operator** that will handle all types:

```
template<typename LHS, typename RHS>
atl::Vector<typename PromoteType<LHS,RHS>::return_type > operator*(
     const atl::Vector<LHS>& a, const atl::Vector<RHS>& b)
{
    ...
}
```

This can be particularly useful in cases where one has large amounts of data that is not given in high precision. While the modeling may require a high degree of precision, it is not necessary to store the user supplied data at this level of precision. Using a float rather than double may suffice and help to reduce the memory requirements of your program.

5 Optimization

5.1 Function Minimization

```
template<class T>
class Simple : public atl::FunctionMinimizer<T> {
    int nobs = 10;
    atl::Vector<T> x = {-1, 0, 1, 2, 3, 4, 5, 6, 7, 8};
    atl::Vector<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::Vector<atl::Variable<T> > pred_y;
public:

    void Initialize() {

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

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

    }

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


};

int main() {

    Simple<double> s;
    s.Initialize();
    s.Run();
}
```

Output:

```
Iteration: 0
Phase: 1 of 1
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 36.4936
Max Gradient:3.61269
Id      Name      Value         Gradient      |Id      Name      Value         Gradient
1       a         0             -3.61269      2       b         0             -0.727814

Iteration: 10
Phase: 1 of 1
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 15.6174
Max Gradient:10.2763
Id      Name      Value         Gradient      |Id      Name      Value         Gradient
1       a         1.99319       10.2763       2       b         4.25282       2.06375

Iteration: 13
Phase: 1 of 1
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 14.9642
Max Gradient:1.24121e-05
Id      Name      Value         Gradient      |Id      Name      Value         Gradient
1       a         1.90909       -1.24121e-05  2       b         4.07818       2.81961e-06
```

```
template<class T>
class VonBertalanffy : public atl::FunctionMinimizer<T> {
    int nobs = 20;
    atl::Matrix<float> data = { //use float for data
        {2, 1},
        {2, 3},
        {2, 4},
        {2, 4},
        {3, 3},
        {3, 4},
        {3, 5},
        {4, 6},
        {4, 9},
        {6, 10},
        {9, 14},
        {10, 13},
        {11, 16},
        {11, 17},
        {15, 18},
        {19, 19},
        {21, 20},
        {24, 21},
        {30, 20},
        {35, 21}
    };

    atl::Vector<T> a;
    atl::Vector<T> L;
    atl::Variable<T> t0;
    atl::Variable<T> Linf;
    atl::Variable<T> k;
    atl::Variable<T> sd;
    atl::Vector<atl::Variable<T> > Lpred;
public:
```

```
    void Initialize() {

        a = atl::Column(data,0);
        L = atl::Column(data,1);

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

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

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

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

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


};


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

    VonBertalanffy<double> s;
    s.Initialize();
    s.Run();
}
```

Output:

```
Phase: 1 of 2
Iteration: 0
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 1.24844
Max Gradient:16.923
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0             -0.889178     |2      Linf     21            0.071395
3       k         0.1           16.923

Phase: 1 of 2
Iteration: 10
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 1.00705
Max Gradient:15.0288
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.505368      0.214999      |2      Linf     21.045        -0.101968
3       k         0.0878456     -15.0288

Phase: 1 of 2
Iteration: 20
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 0.840932
Max Gradient:0.041383
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.976758      0.00873464    |2      Linf     21.095        -0.007477
3       k         0.123973      -0.041383

Phase: 1 of 2
Iteration: 30
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 0.837156
Max Gradient:0.00120471
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.92906       -2.157e-05    |2      Linf     22.1739       -5.0845e-06
3       k         0.11317       -0.00120471

Phase: 1 of 2
Iteration: 32
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 0.837156
Max Gradient:5.14737e-06
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.929196      -4.37591e-07  |2      Linf     22.1727       2.9163e-08
3       k         0.113188      5.14737e-06

Phase: 2 of 2
Iteration: 0
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: 0.837156
Max Gradient:18.3257
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.929196      -4.37591e-07  |2      Linf     22.1727       2.9163e-08
3       k         0.113188      5.14737e-06   4      sd       1             18.3257

Phase: 2 of 2
Iteration: 10
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: -14.8033
Max Gradient:0.000114764
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.929199      3.63803e-05   |2      Linf     22.1727       2.47461e-06
3       k         0.113188      0.000114764   4      sd       0.289336      2.37732e-06

Phase: 2 of 2
Iteration: 11
Convergence Criteria: 0.0001
Routine: L-BFGS
Function Value: -14.8033
Max Gradient:4.31945e-05
Id      Name      Value         Gradient      |Id     Name     Value         Gradient
1       t0        0.929199      4.31945e-05   |2      Linf     22.1727       1.72384e-06
3       k         0.113188      2.61195e-06   4      sd       0.289336      8.38737e-09
```

11 Future Work

11.1 Random Effects Modeling

11.2 MPI

12 References

13 Appendices