Notes On Implementing Differentiable Minimum And Maximum Functions In ATL and ADMB

Matthew R. Supernaw

National Oceanic Atmospheric Administration
National Marine Fisheries Service, Office of Science and Technology
matthew.supernaw@noaa.gov

Z. Teresa A'mar, PhD

National Oceanic Atmospheric Administration
National Marine Fisheries Service, Office of Science and Technology
teresa.amar@noaa.gov

Contents

1 Introduction

The "minimum" and "maximum" functions (usually written as *min(a,b)* and *max(a,b)*, respectively) are problematic for automatic differentiation systems. The issue is that existing *min()* and *max()* functions rely on conditional statements, or branches, to calculate the return value. This method results in a piecewise continuous function which is not generally differentiable. In this paper we describe a method so that the *min()* and *max()* functions with two automatic differentiation variable types as arguments can be calculated with no branching, thereby preserving the derivatives of the variables. In addition, we describe how these functions can be implemented in ATL, ADMB, and TMB.

2 Methods

Given two variables a and b, calculate the minimum or maximum value. Traditional methods 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;
}
```

These branchless versions are now differentiable everywhere except when a and b are both zero, as the absolute value function *fabs()* is not differentiable when the argument is zero.

2.1 ATL

The Analytics Template Library (ATL) is a general purpose C++ template metaprogramming library for scientific computing being developed by NOAA. ATL has a reverse mode automatic differentiation module that can compute higher-order mixed derivatives up to the 3rd order. In

ATL, the *fabs(x)* function is not differentiable when x is zero, thus the *min()* and *max()* functions are not differentiable when a minus b is zero. The code for *min()* and *max()* in ATL:

```
/**
 * Returns the minimum of a and b in a branchless manner using:
 *
 * (a + b - |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
template <typename T>
inline const atl::Variable<T> min(const atl::Variable<T>& a,
  const atl::Variable<T>& b) {
    return (a + b - atl::fabs(a - b)) / 2.0;
}

/**
 * Returns the maximum of a and b in a branchless manner using:
 *
 * (a + b + |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
template <typename T>
inline const atl::Variable<T> max(const atl::Variable<T>& a,
  const atl::Variable<T>& b) {
    return (a + b + atl::fabs(a - b)) / 2.0;
}
```

2.2 ADMB

Unlike ATL, the *fabs(x)* function in ADMB always returns a derivative value even if x is zero. In ADMB, if x is greater than or equal to zero, the resulting derivative is 1, otherwise it is -1. The ADMB code for *min()* and *max()*:

```
/**
 * Returns the maximum between a and b in a continuous manner using:
 *
 * (a + b + |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
inline prevariable& max(const dvariable& a, const dvariable& b) {
   if (++gradient_structure::RETURN_PTR > gradient_structure::MAX_RETURN)
      gradient_structure::RETURN_PTR = gradient_structure::MIN_RETURN;

   *gradient_structure::RETURN_PTR = (a + b + fabs(a - b)) / 2.0;
   return *gradient_structure::RETURN_PTR;
}


/**
 * Returns the minimum between a and b in a continuous manner using:
 *
 * (a + b - |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
inline prevariable& min(const dvariable& a, const dvariable& b) {
   if (++gradient_structure::RETURN_PTR > gradient_structure::MAX_RETURN)
      gradient_structure::RETURN_PTR = gradient_structure::MIN_RETURN;

   *gradient_structure::RETURN_PTR = (a + b - fabs(a - b)) / 2.0;
   return *gradient_structure::RETURN_PTR;
}
```

3 Discussion

The differences in the ATL and ADMB versions of the *min()* and *max()* functions arise from the implementation of *fabs(x)*. ATL will return a derivative value of *nan* when *a* minus *b* is zero. This is because the derivative of the absolute value function is undefined at zero. However, ADMB will return a derivative value of 1 when x is zero. We suggest a discussion on which method is more robust for fisheries science applications, and ask for contributions from the broader community for guidance.

4 References

5 Appendix A: ATL Example Source Code

```
/**
 * Returns the minimum of a and b in a branchless manner using:
 *
 * (a + b - |a - b)|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
template <typename T>
inline const atl::Variable<T> ad_min(const atl::Variable<T>& a,
const atl::Variable<T>& b) {
    return (a + b - atl::fabs((a - b))) / 2.0;
}

/**
 * Returns the maximum of a and b in a branchless manner using:
 *
 * (a + b + |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
template <typename T>
inline const atl::Variable<T> max(const atl::Variable<T>& a,
 const atl::Variable<T>& b) {
    return (a + b + atl::fabs(a - b)) / (2.0);
}


inline const atl::Variable<double> ad_normalize_and_sum(
std::vector<atl::Variable<double> >& v) {
    atl::Variable<double> maxv = v[0];
    for (int i = 1; i < v.size(); i++) {
        maxv = max(maxv, v[i]);
    }

    for (int i = 0; i < v.size(); i++) {
        v[i] /= maxv;
    }

    atl::Variable<double> sum;
    for (int i = 0; i < v.size(); i++) {
        sum += v[i];
    }

    return sum;
}
```

```cpp
inline const atl::Variable<double> ad_min_max_test(int nvar,
std::vector<atl::Variable<double> >& x) {
    std::vector<atl::Variable<double> > X(nvar, atl::Variable<double>());
    for (int i = 0; i < x.size(); i++) {
        X[i] = atl::Variable<double>(x[i] * x[i]);
    }
    return ad_normalize_and_sum(X);
}

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

    std::vector<atl::Variable<double> > x(10);
    for (int i = 0; i < x.size(); i++) {
        x[i] = (double) (i + 1);
    }

    atl::Variable<double> ret = ad_min_max_test(x.size(), x);

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

    std::cout << "Gradient:\n";
    for (int i = 0; i < x.size(); i++) {
        std::cout <<
        atl::Variable<double>::tape.first_order_derivatives[x[i].info->id] << " ";
    }
    return 0;
}
```

Output

```
Gradient:
0.02 0.04 0.06 0.08 0.1 0.12 0.14 0.16 0.18 -0.57
```

6 Appendix B: ADMB Example Source Code

```
/**
 * Returns the minimum of a and b in a branchless manner using:
 *
 * (a + b - |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
inline prevariable& min(const dvariable& a, const dvariable& b) {
   if (++gradient_structure::RETURN_PTR > gradient_structure::MAX_RETURN)
      gradient_structure::RETURN_PTR = gradient_structure::MIN_RETURN;

   *gradient_structure::RETURN_PTR = (a + b - fabs(a - b)) / (2.0);
   return *gradient_structure::RETURN_PTR;
}

/**
 * Returns the maximum of a and b in a branchless manner using:
 *
 * (a + b + |a - b|) / 2.0;
 *
 * @param a
 * @param b
 * @return
 */
inline prevariable& max(const dvariable& a, const dvariable& b) {
   if (++gradient_structure::RETURN_PTR > gradient_structure::MAX_RETURN)
      gradient_structure::RETURN_PTR = gradient_structure::MIN_RETURN;

   *gradient_structure::RETURN_PTR = (a + b + fabs(a - b)) / (2.0);
   return *gradient_structure::RETURN_PTR;
}

const dvariable ad_normalize_and_sum(std::vector<dvariable>& v) {
   dvariable maxd = v[0];
   for (int i = 1; i < v.size(); i++) {
      maxd = max(maxd, v[i]);
   }

   for (int i = 0; i < v.size(); i++) {
      v[i] /= maxd;
   }

   dvariable sum;
   for (int i = 0; i < v.size(); i++) {
      sum += v[i];
   }

   return sum;
}
```

```
inline const dvariable ad_min_max_test(int nvar, dvar_vector x) {
    std::vector<dvariable> X(nvar);
    for (int i = 0; i < X.size(); i++) {
        X[i] = x(i + 1) * x(i + 1);
    }
    return ad_normalize_and_sum(X);
}

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

    int nvar = 10;
    gradient_structure::set_MAX_NVAR_OFFSET(nvar);
    gradient_structure gs(800000000L);

    dvector g(1, nvar);
    independent_variables x(1, nvar);
    for (int i = 1; i <= nvar; i++) {
        x(i) = (double) i;
    }

    dvariable ret = ad_min_max_test(nvar, x);

    gradcalc(nvar, g); // The derivatives are calculated
    cout << "Gradient:\n" << g << "\n";
    return 0;
}
```

Output

```
Gradient:
0.02 0.04 0.06 0.08 0.1 0.12 0.14 0.16 0.18 -0.57
```