Finite Difference Method on Nonuniform Grid

Stephen Face

June 24, 2017

The goal of these calculations is to describe a method by which one can use finite difference methods of order N with a nonuniform grid. The method takes $N+1$ distinct points near the point of interest, fits an Nth order polynomial, and uses that polynomial for the derivative. Boundary conditions such as even or odd symmetry are not addressed, but they could be computed from a simple extension of this method.

As input to the method, we take the location of $N+1$ grid points x_i and the value of the quantity at the grid points y_i. First, we find an Nth order polynomial which passes through all (x_i, y_i). This can be done by first constructing a collection of Nth order polynomials ℓ_j such that $\ell_j(x_i) = \delta_{ij}$ (see `https://en.wikipedia.org/wiki/Lagrange_polynomial`).

$$\ell_j(x) = \prod_{a \neq j} \frac{x - x_a}{x_j - x_a} \tag{1}$$

$$\ell_j(x_i) = \prod_{a \neq j} \frac{x_i - x_a}{x_j - x_a} = \delta_{ij} \tag{2}$$

The order of $\ell_i(x)$ is N, so $p(x) = \sum_i y_j \ell_j(x)$ is an Nth order polynomial such that $p(x_i) = y_i$. Let $dy_i = p'(x_i)$ be the derivative of $p(x)$ at our grid points.

$$dy_i = \sum_j y_j \ell_j'(x_i) \tag{3}$$

This is an inner product between the values y_j and a matrix of coefficients $\ell_j'(x_i)$. Note that the coefficients depend on at which x_i we want the derivative, but they are independent of the values y, so one can compute them once and take derivatives of many functions with less computational cost. Our task now is just to compute these coefficients $\ell_j'(x_i)$. We split the computation in two cases, $i = j$ and $i \neq j$.

First, $i \neq j$:

$$\ell_j'(x_i) = \frac{\partial}{\partial x}\left(\prod_{a\neq j}\frac{x - x_a}{x_j - x_a}\right)\Bigg|_{x=x_i} \tag{4}$$

$$= \frac{1}{\prod_{a\neq j}(x_j - x_a)}\frac{\partial}{\partial x}\left(\prod_{b\neq j}(x - x_b)\right)\Bigg|_{x=x_i} \tag{5}$$

Since $i \neq j$, one of the x_a in the product is x_i, thus only one term contributes to the derivative.

$$\ell_j'(x_i) = \frac{1}{\prod_{a\neq j}(x_j - x_a)}\prod_{b\neq i,j}(x_i - x_b) \tag{6}$$

Next, $i = j$:

$$\ell_j'(x_j) = \frac{1}{\prod_{a\neq j}(x_j - x_a)}\frac{\partial}{\partial x}\left(\prod_{b\neq j}(x - x_b)\right)\Bigg|_{x=x_j} \tag{7}$$

Now all of the terms contribute to the derivative.

$$\ell_j'(x_j) = \frac{1}{\prod_{a\neq j}(x_j - x_a)}\sum_{b\neq j}\left(\prod_{c\neq b,j}(x_j - x_c)\right) \tag{8}$$

As a final note, the explicit computation for $i = j$ can be skipped if the values for all the other j are to be computed as well since $\sum_j \ell_j'(x_i) = 0$. On the following pages an implementation of these calculations in C++ is included for completeness.

N.B. The code makes use of templates and C++14 features to allow the compiler to automatically generate code at different orders. I make no claims that other implementations should mimic it.

C++ implementation of nonuniform finite difference

```cpp
#ifndef DX_H_
#define DX_H_

#include <algorithm>
#include <array>
#include <cassert>
#include <iterator>
#include <numeric>
#include <utility>
#include <vector>

//
// A class for differentiation with unevenly spaced x.
//
// To use, first call set_x(x) to compute coefficient weights
//
template <class T, int Order>
class DX {
 public:
  constexpr static int const N = Order + 1;  // Number of points in the stencil
  constexpr static int const R = Order / 2;  // Number of points to one side
  constexpr DX(size_t n) : wx_(n) {}
  constexpr DX() {}

  // No copy
  DX(DX const&) = delete;
  DX& operator=(DX const&) = delete;

  size_t size() const { return wx_.size(); }
  void resize(size_t size) {
    assert(size >= N);
    wx_.resize(size);
  }
  size_t capacity() const { return wx_.capacity(); }
  void reserve(size_t size) { wx_.reserve(size); }

  // Computes dy/dx and writes to iterator dy.
  // set_x must be called first.
  template <class ForwardIt, class OutputIt>
  void dydx(ForwardIt y, OutputIt dy) const {
    typename std::vector<std::array<T, N>>::const_iterator a = std::cbegin(wx_);

    for (int i = 0; i < N / 2; ++i, ++a, ++dy) {
      *dy = std::inner_product(std::begin(*a), std::end(*a), y,
```

```
                                static_cast<T>(0));
    }
    int const interior = wx_.size() - N;
    for (int i = 0; i < interior; ++i, ++y, ++dy, ++a) {
      *dy = std::inner_product(std::begin(*a), std::end(*a), y,
                                static_cast<T>(0));
    }
    for (int i = N / 2; i < N; ++i, ++a, ++dy) {
      *dy = std::inner_product(std::begin(*a), std::end(*a), y,
                                static_cast<T>(0));
    }
  }

  // Returns dy/dx at the left boundary.
  // set_x must be called first.
  template <class ForwardIt>
  T dydx_L(ForwardIt y) const {
    return std::inner_product(std::begin(wx_.front()), std::end(wx_.front()), y,
                               static_cast<T>(0));
  }

  // Returns dy/dx at the right boundary.
  // set_x must be called first.
  template <class ForwardIt>
  T dydx_R(ForwardIt y) const {
    std::advance(y, wx_.size() - N);
    return std::inner_product(std::begin(wx_.back()), std::end(wx_.back()), y,
                               static_cast<T>(0));
  }

  // Compute coefficients for a set of x
  template <class ForwardIt>
  void set_x(ForwardIt x) {
    typename std::vector<std::array<T, N>>::iterator a = std::begin(wx_);

    weight_left(x, a);
    std::advance(a, N / 2);

    const int interior = wx_.size() - N;
    for (int i = 0; i < interior; ++i, ++x, ++a) {
      weight_in(x, a);
    }

    weight_right(x, a);
  }

```

```
private:
 // The ``Real Math'' follows:
 // The weights are w_ij = \frac{\partial}{\partial y_j}(y'_i)

 template <int K, class ForwardIt, class OutputIt,
           std::enable_if_t<!(K < N / 2), int> = 0>
 static inline void weight_left(ForwardIt const, OutputIt const) {}

 template <int K = 0, class ForwardIt, class OutputIt,
           std::enable_if_t<(K < N / 2), int> = 0>
 static inline void weight_left(ForwardIt const x, OutputIt const a) {
   denominators(x, std::begin(*a));
   numerators<K>(x, std::begin(*a));
   weight_left<K + 1>(x, std::next(a));
 }

 template <int K, class ForwardIt, class OutputIt,
           std::enable_if_t<!(K < N), int> = 0>
 static inline void weight_right(ForwardIt const, OutputIt const) {}

 template <int K = (N / 2), class ForwardIt, class OutputIt,
           std::enable_if_t<(K < N), int> = 0>
 static inline void weight_right(ForwardIt const x, OutputIt const a) {
   denominators(x, std::begin(*a));
   numerators<K>(x, std::begin(*a));
   weight_right<K + 1>(x, std::next(a));
 }

 template <class ForwardIt, class OutputIt>
 static inline void weight_in(ForwardIt const x, OutputIt const a) {
   denominators(x, std::begin(*a));
   numerators<N / 2>(x, std::begin(*a));
 }

 // The denominators for the weights.
 // d_i = den(w_ij)
 template <class ForwardIt, class OutputIt>
 static inline void denominators(ForwardIt const x, OutputIt a) {
   ForwardIt y = x;
   std::fill_n(a, N, 1.0);
   for (int i = 0; i < N; ++i, ++y, ++a) {
     ForwardIt z = x;
     for (int j = 0; j < N; ++j, ++z) {
       if (j == i) continue;
       *a *= *y - *z;
     }
```

```
        *a = 1.0 / *a;
      }
    }

    // Multiplies by the numerators for the weights.
    // n_kj = num(w_kj)
    template <int K, class ForwardIt, class OutputIt>
    static inline void numerators(ForwardIt const x, OutputIt a) {
      ForwardIt const y = std::next(x, K);
      OutputIt const c = std::next(a, K);

      OutputIt b = a;
      for (int i = 0; i < N; ++i, ++b) {
        if (i == K) continue;
        ForwardIt z = x;
        for (int j = 0; j < N; ++j, ++z) {
          if (j == K || j == i) continue;
          *b *= *y - *z;
        }
      }

      *c = 0.0;
      for (int i = 0; i < N; ++i, ++a) {
        if (i == K) continue;
        *c -= *a;
      }
    }

    // Coefficient storage
    std::vector<std::array<T, N>> wx_;
};

#endif  // DX_H_
```