PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science
Dalhousie University

tom.duck@dal.ca

February 2, 2010

CHAPTER

TWO

INTERPOLATORY DIFFERENTIATION

In this chapter, we seek to determine the first and second derivatives of a curve represented by discrete points that are spaced evenly along the x-axis. The coordinates for the points may be represented by arrays of values on the x and y axes, as discussed before.

Let's begin with some review. Consider two points at coordinates (x_0, y_0) and (x_1, y_1) that lie on a smooth curve $y = f(x)$ (see Figure 2.1). The slope of the straight line between the two points is just the rise over run; i.e.,

$$\text{slope} = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$$

The line is said to *interpolate* the two points because it goes between them. Notice that the slope of the interpolatory line is given by the ratio of the *finite differences* $\Delta y = y_1 - y_0$ and $\Delta x = x_1 - x_0$.

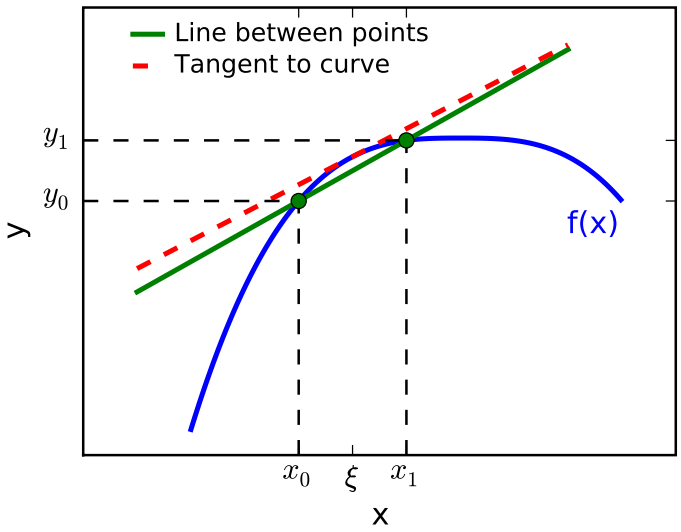

Figure 2.1: Two discrete points on a continuous curve. The slope of the straight line that interpolates the two points is very nearly the same as the slope of the tangent line taken at $x = \xi$ between them.

Now, the first derivative at any point along the curve is given by the slope of the tangent line at that point. In contrast with the interpolatory line, the slope of the tangent line is given by a ratio of *infinitesimal differences*; i.e.,

$$f' \equiv \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} \quad .$$

Inspection of Figure 2.1 reveals that the slopes of the tangent and interpolatory lines are nearly the same. The figure suggests that we can estimate derivatives along a continuous curve from the coordinates of discrete points along the curve, an approach called *interpolatory differentiation*.

It is necessary to put our idea on a firm theoretical footing. We need to derive formulas for derivate estimates and establish some understanding of the uncertainties. To accomplish both we will use *Taylor polynomials*. The overall approach is called the *Euler method*.

2.1 Taylor's Theorem and the Euler Method

Consider an arbitrary continuous function $y = f(x)$. If $f(x)$ is well-behaved, then it can be expressed as a polynomial series, i.e.,

$$f(x) = c_0 + c_1(x-a) + c_2(x-a)^2 + c_3(x-a)^3 + \ldots$$

for some constant a with a correct choice of coefficients $(c_0, c_1, c_2, \ldots)$. Taylor's theorem states that correctly-chosen coefficients are given by the formula

$$f(x) = f(a) + f'(a)\frac{(x-a)^1}{1!} + f''(a)\frac{(x-a)^2}{2!} + \ldots + f^{(n)}(a)\frac{(x-a)^n}{n!} + R_n$$

where the remainder is

$$R_n = f^{(n+1)}(\xi)\frac{(x-a)^{(n+1)}}{(n+1)!}$$

for some value ξ between x and a. Taylor's theorem says that the value of a function at some location x can be determined if the value of the function and its derivatives are known at some other location a along the x-axis (see Figure 2.2). If the value x is "around" (i.e., suitably close to) a, then $x - a$ will be small, as will the remainder. Thus, $f(x)$ can be approximated by a truncated Taylor polynomial in the neighbourhood of a.

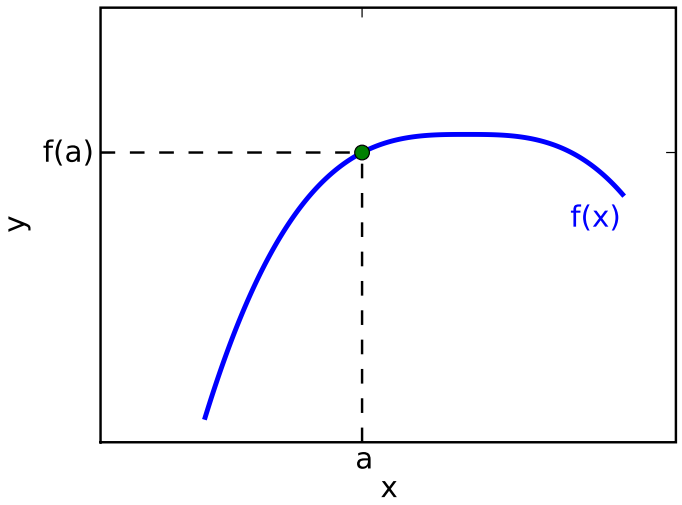

Figure 2.2: A curve $y = f(x)$ with the point $x = a$ and $y = f(a)$ marked on it.

We can use Taylor's theorem to construct a formula for estimating derivatives. Consider a series of x-values

$$\ldots, x_0, x_1, x_2, x_3, \ldots, x_i, x_{i+1}, \ldots$$

equispaced such that $x_{i+1} - x_i = h$ (see Figure 2.3), with corresponding function values

$$\ldots, f_0, f_1, f_2, f_3, \ldots, f_i, f_{i+1}, \ldots$$

where $f_i \equiv f(x_i)$ for given index i.

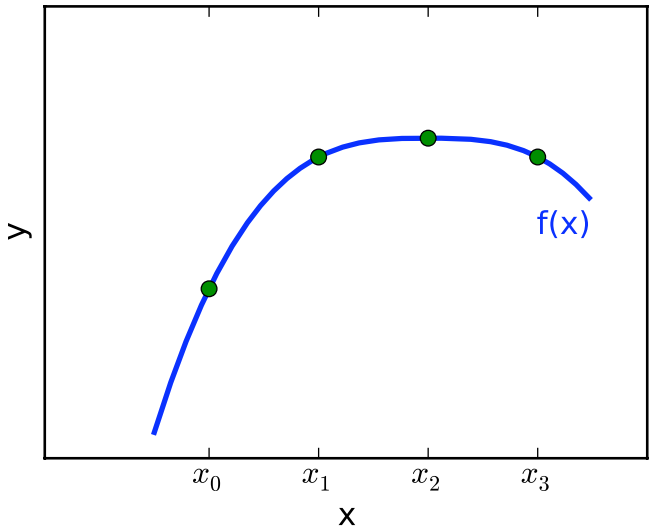

Figure 2.3: Points along the continuous curve $f(x)$ that are equispaced on the x-axis.

Evaluating the Taylor polynomial of $f(x)$ at $x = x_{i+1}$ which is around $a = x_i$ gives

$$f(x_{i+1}) = f(x_i) + f'(x_i)\frac{(x_{i+1} - x_i)}{1!} + f''(x_i)\frac{(x_{i+1} - x_i)^2}{2!} + \ldots + f^{(n)}(x_i)\frac{(x_{i+1} - x_i)^n}{n!} + O(h^{n+1})$$

which simplifies to

$$f_{i+1} = f_i + f_i' h + f_i''\frac{h^2}{2} + \ldots + f_i^{(n)}\frac{h^n}{n!} + O(h^{n+1})$$

Here, $O(h^{n+1})$ indicates that the *truncation error* determined using the remainder formula is of order $n + 1$ in the x-spacing h.

If we truncate the series after first order in h so that

$$f_{i+1} = f_i + f_i' h + O(h^2)$$

and rearrange for f_i', then we have the *forward difference formula* for the first derivative:

$$f_i' = \frac{f_{i+1} - f_i}{h} + O(h)$$

The derivative estimate at $x = x_i$ is the slope for a straight line interpolating the points at $x = x_i$ and the adjacent point forward at $x = x_{i+1}$ (see Figure 2.4).

The *backward difference formula* can be determined by evaluating f at $x = x_{i-1}$ for a series expansion around $a = x_i$:

$$f_i' = \frac{f_i - f_{i-1}}{h} + O(h)$$

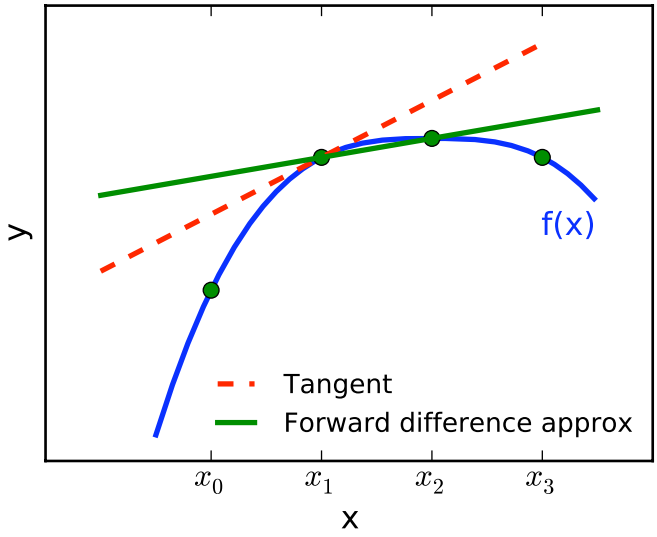

Figure 2.4: Forward difference estimate of the derivative at $x = x_i$. The true derivative is given by the tangent line.

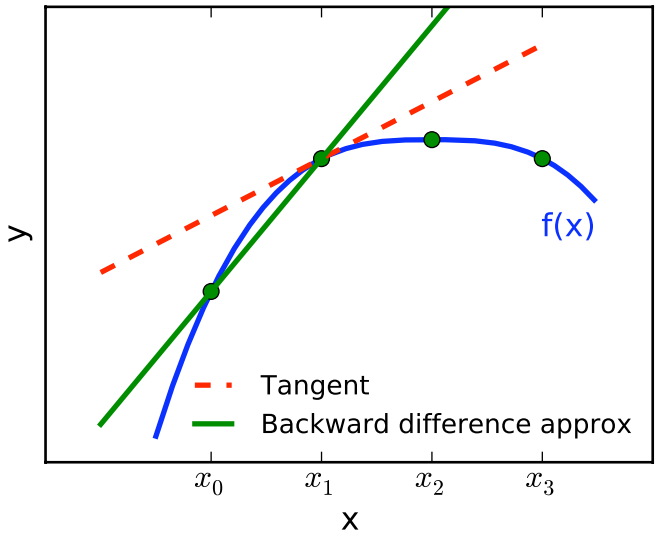

Figure 2.5: Backward difference estimate of the derivative at $x = x_i$. The true derivative is given by the tangent line.

The derivative estimate at $x = x_i$ is the slope for a straight interpolating the point at $x = x_i$ and the adjacent point backward at $x = x_{i-1}$ (see Figure 2.5).

The *central difference formula* can be obtained using the forward expansion truncated after second order

$$f_{i+1} = f_i + f_i' h + f_i'' \frac{h^2}{2} + O(h^3)$$

and the backward expansion truncated after second order

$$f_{i-1} = f_i - f_i' h + f_i'' \frac{h^2}{2} + O(h^3)$$

Subtracting the second equation from the first and rearranging gives

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2)$$

Notice that the second derivatives subtracted out.

The derivative estimate at $x = x_i$ is the slope for a straight line interpolating the adjacent points forward at $x = x_{i+1}$ and backward at $x = x_{i-1}$ (see Figure 2.6).

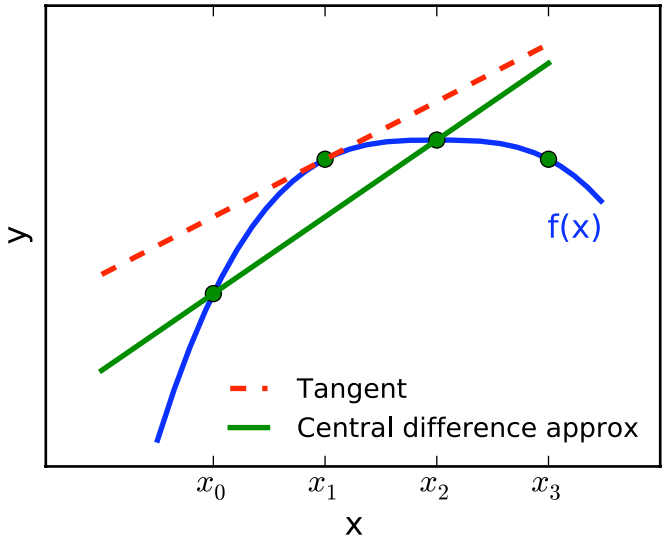

Figure 2.6: Central difference estimate of the derivative at $x = x_i$. The true derivative is given by the tangent line.

Inspection of the plots suggests that the central difference formula (error $O(h^2)$) is more accurate than either the forward or backward difference formulas (error $O(h)$). In general, higher order difference formulas have lower error, and others can be derived by retaining more terms in the Taylor expansion. Second derivative formulas can also be determined. The lowest-order formula for second derivatives is

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2)$$

2.2 Array Indexing, Looping, and Slicing

For demonstration purposes, consider the function $f(x) = x^2$, for which the first derivative is $f'(x) = 2x$. We can express the function in a series of discrete points as follows:

```
>>> import numpy
>>> x = numpy.arange(6)   #i.e., [0,1,2,3,4,5]
>>> f = x**2
```

I have deliberately considered only a few points along the curve for demonstration purposes. The x-spacing is $h = 1.0$.

An array consists of a series of *elements* in memory, and they are indexed sequentially from the left using positive values starting with 0, or from the right using negative values starting at -1 (see Figure 2.7).

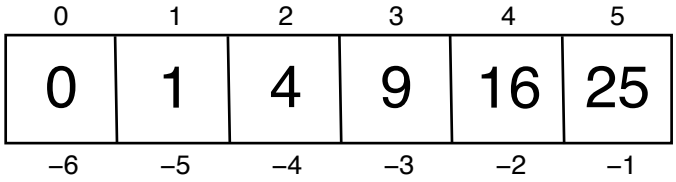

Figure 2.7: An array of numerical values. The elements of the array are large and in bold. Indices above increment from the left side of the array, and those below decrement from the right side of the array.

To implement the forward difference formula

$$f'_i = \frac{f_{i+1} - f_i}{h} + O(h)$$

we need to be able to retrieve the array values at selected indices. This can be done using positive indices as follows:

```
>>> f[0]   # Evaluates first element
0
>>> f[1]   # Second element
1
>>> f[2]   # Third element
4
```

Similarly, the elements can be retrieved using negative indices as follows:

```
>>> f[-1]  # Last element (there is no such thing as "-0")
25
>>> f[-2]  # Second-last element
16
```

Assignment to an array works as you might expect:

```
>>> print f
[ 0  1  4  9 16 25]
>>> f[0] = 100
>>> print f
[100   1   4   9  16  25]
```

Let's fix the array and continue on:

```
>>> f[0] = 0
```

Using the index notation we can calculate the first derivative $f_i^{(1)}$ at each index `i`:

```
>>> h = 1.0  # Your h value should always be a float
>>> f1_0 = (f[1]-f[0])/h
>>> f1_1 = (f[2]-f[1])/h
>>> f1_2 = (f[3]-f[2])/h
>>> f1_3 = (f[4]-f[3])/h
>>> f1_4 = (f[5]-f[4])/h
>>> f1_5 = (f[6]-f[5])/h
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of bounds
```

Everything went fine until we reached index $i = 5$. The problem here is that `f[6]` does not exist. So, we cannot calculate a derivative corresponding to the last x value.

It would be more useful to have the derivative values in an array. This can be done manually as follows:

```
>>> f1 = numpy.array([f1_0,f1_1,f1_2,f1_3,f1_4])
```

How good is our derivative estimate? Let's see:

```
>>> print f1  # Estimate
[ 1.  3.  5.  7.  9.]
>>> print 2*x  # Truth
[ 0  2  4  6  8 10]
```

Not too bad. There is obviously some error, but we expected that.

You may have noticed that the processes is getting very repetitive. Programming languages are great at performing repetitive tasks, and so you should be thinking at this point that there must be a better way. There is, so let's see how.

First, define an array to hold the derivative values:

```
>>> f1 = numpy.zeros(5)
>>> print f1
[ 0.  0.  0.  0.  0.]
```

We can fill the array with derivative values using a *while loop*:

```
>>> i = 0
>>> while i<5:
...     f1[i] = (f[i+1]-f[i])/h
...     i = i + 1
...
>>> print f1
[ 1.  3.  5.  7.  9.]
```

The while loop is a new programming construct for us. So long as the condition $i < 5$ in the while clause is `True`, the block of code that follows it executes repeatedly. The variable `i` is 0 the first time the block of code executes. At the end of the block `i` is *incremented*, and so the second time the block executes it will be equal to 1. The third time through it will be 2, and so on, until `i` is 5 and the loop terminates. Notice that the forward difference formula is implemented in code almost exactly as it appears in the mathematics.

The same process can be implemented more compactly using a for loop:

```
>>> for i in [0,1,2,3,4]:
...     f1[i] = (f[i+1]-f[i])/h
...
>>> print f1
[ 1.  3.  5.  7.  9.]
```

The for loop *iterates* the variable `i` over a sequence of values, in this case a list of numbers between 0 and 4. In the first iteration, the block of code in the for loop has `i` as 0. In the second iteration `i` is 1, and so on until there are no more values left in the list.

The above code can be written even more compactly as

```
>>> for i in range(5):
...     f1[i] = (f[i+1]-f[i])/h
...
>>> print f1
[ 1.  3.  5.  7.  9.]
```

The `range()` function is another built-in function. It behaves in much the same way as `numpy.arange()` except that a list of values is returned instead of an array.

The forward difference formula can be implemented without any looping at all. Looping is generally an option of last resort in python (and any interpreted language), because it is computationally *expensive* (looping, however, can be quite efficient in compiled programming languages like C). An alternative approach can be devised by noting that the `f[i+1]` values in the loops above have indices ranging between 1 and 5, whereas the `f[i]` values have indices ranging between 0 and 4 (see Figure 2.8). If we can select subsets of the array `f` then we can calculate the derivative in one fell swoop (no iterating) using standard array mathematics.

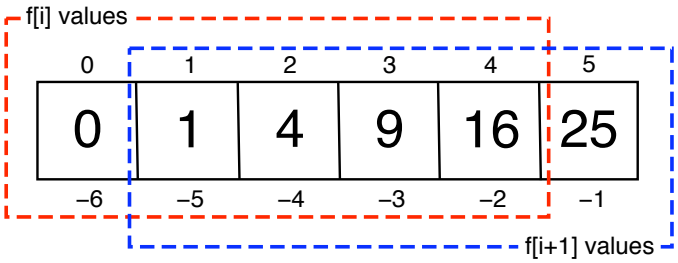

Figure 2.8: In the derivative calculation, the f[i+1] values form a certain subset, or “slice”, of the array values, and the f[i] values form another.

A subset of elements may be retrieved by using *slice notation* available to sequence types. For example:

```
>>> print f[0:5]
[3 2 6 8 2]
>>> print f[1:6]
[2 6 8 2 7]
```

In the first case we retrieved the values in `f` for indices 0 through 4, and in the second case for indices 1 through 5. The derivative in this case is therefore:

```
>>> f1 = (f[1:6]-f[0:5])/h
>>> print f1
[-1  4  2 -6  5]
```

The advantage of computational efficiency is gained at the expense of an implementation that is harder to identify as the forward difference formula.

If either index in the slice is omitted, the left or right edge is assumed as appropriate. We can also use negative indices as described earlier. These features allows us to write the code more compactly, without any need to know the lenth of the array:

```
>>> f1 = (f[1:]-f[:-1])/h
>>> print f1
[-1  4  2 -6  5]
```

The difference formulas are implemented as functions as follows:

```
def der_forward(f,x,h):
    """Forward difference estimate of the first derivative."""
    return (f[1:]-f[:-1])/h, x[:-1]

def der_backward(f,x,h):
    """Backward difference estimate of the first derivative."""
    return (f[1:]-f[:-1])/h, x[1:]

def der_central(f,x,h):
    """Central difference estimate of the first derivative."""
    return (f[2:]-f[:-2])/(2.*h), x[1:-1]
```

Save these functions to the module tools.py for later use. Each function returns the derivative and corresponding x-values. Notice that for the backward difference formula there is no derivative value for the first x-value, and for the central difference formula there is no derivative value at either end. Can you explain why?

2.3 Example

Let's calculate the relative errors for the forward and backward difference formulas for the exponential function $y = exp(x)$. Note that the exponential function is its own derivative.

```
>>> import numpy
>>> import tools
>>> h = 0.1
>>> x = numpy.arange(-1,1,h)
>>> y = numpy.exp(x)
>>> yder1,x1 = tools.der_forward(y,x,h)
>>> yder2,x2 = tools.der_central(y,x,h)
>>> def rel_err(y1,y2):
...     return numpy.fabs(y1-y2)/y2 * 100.
...
>>> print rel_err(yder1,y[:-1])
[ 5.17091808  5.17091808  5.17091808  5.17091808  5.17091808  5.17091808
  5.17091808  5.17091808  5.17091808  5.17091808  5.17091808  5.17091808
  5.17091808  5.17091808  5.17091808  5.17091808  5.17091808  5.17091808
  5.17091808]
>>> print numpy.max(rel_err(yder1,y[:-1]))
5.17091807565
>>> print numpy.max(rel_err(yder2,y[1:-1]))
0.166750019844
```

Notice that the errors for the central difference formula are smaller than for the forward difference formula.

Calculating the derivatives for a wide range of h values results in the following plot (from deriv_errors.py):

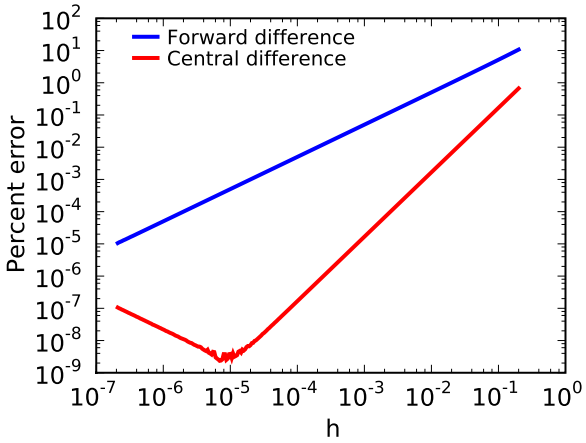

Figure 2.9: Relative error as a function of the x-spacing h.

As we expected, truncation errors generally decrease as h gets smaller. At very small h, however, the errors in the central-difference approximation are elevated. This problem is due to something called *roundoff error*, and to understand it requires an appreciation for how numbers are represented on a computer.