Numerical Differentiation

MAD4401 - Numerical Analysis

Devere Anthony Weaver

October 26, 2022

4.1 - Numerical Differentiation

This section covers different methods used for numerical differentiation of possibly complicated functions.

NOTE: The derivations of these formulae can be found in the typed up pdf notes or in Burden's Numerical Analysis text.

The *forward-difference formula* is given by,

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{h}{2}f''(\xi).$$

This formula's error is bounded by $\frac{M|h|}{2}$ where M is a bound on $|f''(x)|$ for $x_0 < x < x_0 + h$.

The *backward-difference formula* is given by,

$$f'(x_0) = \frac{f(x_0) - f(x_0-h)}{h} + \frac{h}{2}f''(\xi).$$

This formula's error is bounded by $\frac{M|h|}{2}$ where M is a bound on $|f''(x)|$ for $x_0 - h < x < x_0$.

The *centered-difference formula* is given by,

$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{h^2}{6}f'''(\xi).$$

Example 1 Use the *forward-difference formula* to approximate the derivative of $f(x) = \ln x$ at $x_0 = 1.8$ using (i) $h = 0.1$, (ii) $h = 0.05$, and (iii) $h = 0.01$ and determine the bounds for the approximation errors.

[1]:
```
import numpy as np
import math
```

[3]:
```
# implement forward-difference formula for this example
def fd(x,h):
    return ((math.log(x+h)-math.log(x))/h)
```

[4]:
```
# evaluation point
x = 1.8
```

[5]:
```
# h=0.1
fd(x,0.1)
```

[5]: 0.5406722127027574

[6]:
```
# h=0.05
fd(x,0.05)
```

[6]: 0.5479794837622887

[7]:
```
# h=0.01
fd(x,0.01)
```

[7]: 0.5540180375615322

To compute the error bound, we must use the second derivative of the given function,

$$f(x) = \ln x \Rightarrow f'(x) = \frac{-1}{x^2}.$$

Since we are using the *forward-difference formula*, we know that for the error term must be $x_0 < x < x_0 + h$. Here $x_0 = 1.8$ thus $x_0 + h = 1.9$.

So, to compute a bound for this approximation error,

$$\Rightarrow \frac{|hf''(\xi)|}{2} = \frac{|h|}{2\xi^2} < \frac{0.1}{2(1.8)^2} \approx 0.0154.$$

NOTE: Some steps are skipped to compute this error bound, fill in the details as need. Recall, we wanted to maximize our second derivative.

[8]:
```
# implement function to compute error bound
def fderr(x,h):
    return math.fabs(((h/2)*(-1/x**2)))
```

[11]:
```
# x=1.8, h=0.1
fderr(x,0.1)
```

[11]: 0.015432098765432098

[12]:
```
# x=1.8, h=0.05
fderr(x,0.05)
```

[12]: 0.007716049382716049

[13]:
```
# x=1.8, h=0.01
fderr(x,0.01)
```

[13]: 0.0015432098765432098

Since $f'(x) = \frac{1}{x}$, the exact value of $f'(1.8) \approx 0.555...$. This shows the error bounds are close to the true approximation error.

To compute the *true* approximation error,

$$|f'(x) - f'(x_0)|,$$

(i.e. the difference in the value of the function at x and the value of computed using the approximated derivative)

[14]:
```
# compute the true approximation error for x=1.8, h=0.1
math.fabs((1/x)-(fd(x,0.1)))
```

[14]: 0.014883342852798132

[15]:
```
# compute the true approximation error for x=1.8, h=0.05
math.fabs((1/x)-(fd(x,0.05)))
```

[15]: 0.007576071793266914

[16]:
```
# compute the true approximation error for x=1.8, h=0.01
math.fabs((1/x)-(fd(x,0.01)))
```

[16]: 0.0015375179940233519

Wow, they really are close!

We can use the methods learned from the Lagrange Interpolating Polynomial to help obtain general derivative approximation formulas.

Suppose we're give a set of $(n+1)$ distinct numbers $\{x_0, x_1, \ldots, x_n\}$ in some interval I. Also suppose our given function $f \in C^{n+1}(I)$. Recall from Theorem 3.3 (Lagrange Interpolating Polynomials) that we can approximate a function using,

$$f(x) = \sum_{k=0}^{n} f(x_k)L_k(x) + \frac{(x - x_0)\ldots(x - x_n)}{(n+1)!} f^{n+1}(\xi),$$

where $\xi \in I$.

We can then differentiate our Lagrange Interpolating Polynomial. In the derivative, we'll have issues estimating truncation error unless our x is one of the distinct numbers given in the set. When this happens, the terms involving the derivative of the error drop. We then have what is known as the *(n+1)-point formula* to approximate $f'(x_j)$ where $x_j \in \{x_0, \ldots, x_n\}$.

$$f'(x_j) = \sum_{k=0}^{n} f(x_k)L'_k(x_j) + \frac{f^{n+1}(\xi(x_j))}{(n+1)!} \prod_{k=0, k\neq j}^{n} (x_j - x_k).$$

Generally, more evaluation points produces greater accuracy; however, there may be trade-offs in the number of functional evaluations and growth of round-off error.

NOTE: The derivation of useful three-point formulas are contained in the text and/or in the typed up notes. Consult them as needed.

0.1 Three-Point Formulas

There are two three-point formulas that are of concern.

0.1.1 Three-Point Endpoint Formula

$$f'(x_0) = \frac{1}{2h}\left[-3f(x_0) + 4f(x_0+h) - f(x_0+2h)\right] + \frac{h^2}{3}f^{(3)}(\xi_0),$$

where ξ_0 lies between x_0 and $x_0 + 2h$.

0.1.2 Three-Point Midpoint Formula

$$f'(x_0) = \frac{1}{2h}\left[f(x_0+h) - f(x_0-h)\right] + \frac{h^2}{6}f^{(3)}(\xi_1),$$

where ξ_1 lies between $x_0 - h$ and $x_0 + h$.

A point to note is that the errors for these equations are both $O(h^2)$, but the midpoint error is approximately half of the endpoint formula.

The approximation for the endpoint formula is more useful near the ends of the interval since information about f outside of the interval may not be available.

0.2 Five-Point Formulas

These formulas use five evaluation points instead of the three above. They have a common error term that is $O(h^4)$.

0.2.1 Five-Point Midpoint Formula

$$f'(x_0) = \frac{1}{12h}\left[f(x_0-2h) - 8f(x_0-h) + 8f(x_0+h) - f(x_0+2h)\right] + \frac{h^4}{30}f^{(5)}(\xi),$$

where ξ lies between $x_0 - 2h$ and $x_0 + 2h$.

0.2.2 Five-Point Endpoint Formula

$$f'(x_0) = \frac{1}{12h}\left[-25f(x_0) + 48f(x_0+h) - 36f(x_0+2h) + 16f(x_0+3h) - 3f(x_0+4h)\right] + \frac{h^4}{5}f^{(5)}(\xi),$$

where ξ lies between x_0 and $x_0 + 4h$.

NOTE: Left-endpoint approximations are found using this formula when $h > 0$, else right-endpoint approximations.

Example 2 Values for $f(x) = xe^x$ are given in a table in the text.

Task: Use all the applicable three-point and five-point formulas to approximated $f'(2.0)$.

Let $x = 2.0$ and our evaluation points are $\{1.8, 1.9, 2.0, 2.1, 2.2\}$.

[36]:
```
# implement given function
def f(x):
    return x*math.pow(math.e,x)
```

[37]:
```
# vectorize our function
fv = np.vectorize(f)
```

[38]:
```
# set of evaluation points
x = np.array([1.8, 1.9, 2.0, 2.1, 2.2])
```

[39]:
```
# evaluate set of evaluation points with function
y = fv(x)
```

[40]:
```
y
```

[40]:
```
array([10.88936544, 12.70319944, 14.7781122 , 17.14895682, 19.8550297 ])
```

[51]:
```
# implement three-point endpoint formula
def threeEndPt(f,x0,h):
    return (1/(2*h))*(-3*f(x0)+4*f(x0+h)-f(x0+2*h))
```

[54]:
```
# three-point endpoint for x=2.0 and h=0.1
threeEndPt(f,2.0,0.1)
```

[54]:
```
22.03230486614645
```

[55]:
```
# three-point endpoint for x=2.0 and h=-0.1
threeEndPt(f,2.0,-0.1)
```

[55]:
```
22.05452134102383
```

[56]:
```
# implement three-point midpoint formula
def threeMidPt(f,x0,h):
    return (1/(2*h))*(f(x0+h)-f(x0-h))
```

[57]:
```
# three-point midpoint for x=2.0, h=0.1
threeMidPt(f,2.0,0.1)
```

[57]:
```
22.228786880307283
```

[58]:
```
# three-point midpoint for x=2.0, h=0.2
threeMidPt(f,2.0,0.2)
```

[58]:
```
22.414160657029424
```

[60]:
```
# implement five-point midpoint formula
def fiveMidPt(f,x0,h):
    return (1/(12*h))*(f(x0-2*h)-8*f(x0-h)+8*f(x0+h)-f(x0+2*h)))
```

[62]:
```
# five-point midpoint x=2.0, h=0.1
fiveMidPt(f,2.0,0.1)
```

[62]:
```
22.1669956213999
```

NOTE: Without any other information, we should assume the five-point midpoint result is the most accurate since it has more evaluation points than the three-point formulas.

However, if we are able to compute or know the exact value for the derivative, we can compare our approximations with the exact and see which one is most accurate.

[63]:
```
exact = (2+1)*math.pow(math.e,2)
```

[64]:
```
exact
```

[64]:
```
22.16716829679195
```

[69]:
```
# create vector of approximated values
vals = np.array([threeEndPt(f,2.0,0.1),threeEndPt(f,2.0,-0.1),threeMidPt(f,2.0,0.
↪1), threeMidPt(f,2.0,0.2), fiveMidPt(f,2.0,0.1)])
```

[73]:
```
# compute errors
def errors(e,x):
    return e-x
```

[74]:
```
errorsV = np.vectorize(errors)
```

[75]:
```
errorsV(exact,vals)
```

[75]:
```
array([ 1.34863431e-01,  1.12646956e-01, -6.16185835e-02, -2.46992360e-01,
        1.72675392e-04])
```

0.2.3 Second Derivative Midpoint Formula

Of course, we can use numerical methods that allow use to approximate higher order derivatives. The derivation of these formulae are algebraically tedious and can be found elsewhere as needed.

The summarized derivation in Burden's Numerical Analysis gives the following formula for approximating a second derivative,

$$f''(x_0) = \frac{1}{h^2}\left[f(x_0 - h) - 2f(x_0) + f(x_0 + h)\right] - \frac{h^2}{12}f^{(4)}(\xi)$$

where ξ lies between $x_0 - h$ and $x_0 + h$.

If $f^{(4)}$ is continuous on the interval, then it is also bounded and the approximation is $O(h^2)$.

Example 3 Given the same set of evaluation points as example 2, use them and the second derivative midpoint formula to approximate $f''(2.0)$.

[76]:
```
# implement second derivative midpoint formula
def secondMidPt(f,x0,h):
    return (1/h**2)*(f(x0-h)-2*f(x0)+f(x0+h))
```

[77]:
```
# second derivative midpoint for x=2.0, h=0.1
secondMidPt(f,2.0,0.1)
```

[77]:
```
29.59318610000778
```

[78]:
```
# second deriative midpoint for x=2.0, h=0.2
secondMidPt(f,2.0,0.2)
```

[78]:
```
29.704268474394354
```

[81]:
```
# exact value of second derivative at 2.0
exact = (2.0+2)*math.pow(math.e,2.0)
```

[83]:
```
exact
```

[83]:
```
29.556224395722598
```

[85]:
```
# create vector of approximated values
vals = np.array([secondMidPt(f,2.0,0.1),secondMidPt(f,2.0,0.2)])
```

[86]:
```
# compute errors
errorsV(exact, vals)
```

[86]:
```
array([-0.0369617 , -0.14804408])
```