PHY407 – University of Toronto
Lecture 3: Gaussian Quadrature, Numerical Derivatives

Nicolas Grisouard, nicolas.grisouard@utoronto.ca

28 September 2020
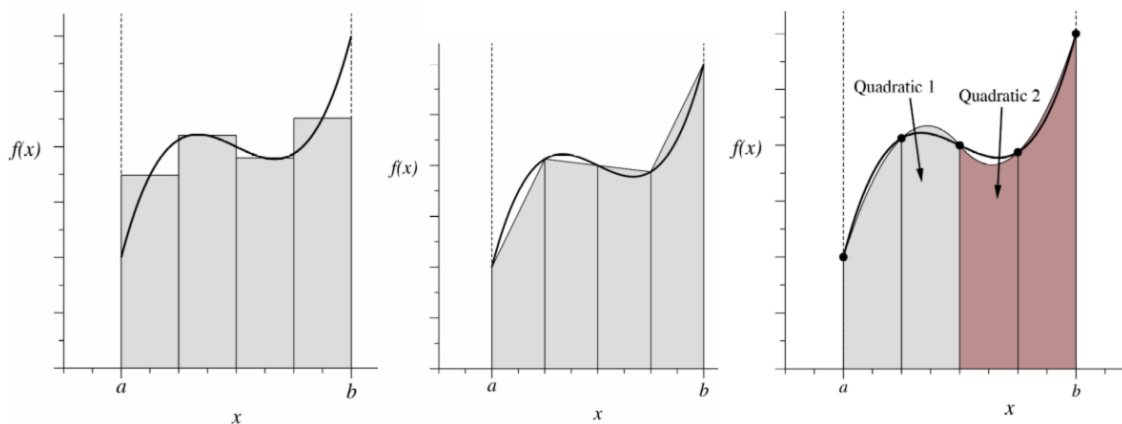
Supporting textbook chapters for week 3: 5.5, 5.6, 5.10

[ ]:

Lecture 3, topics: * Integration by Gaussian quadrature * Numerical differentiation

1   Numerical integration

- Think of integrals as areas under curves.
- Approximate these areas in terms of simple shapes (rectangles, trapezoids, rectangles with parabolic tops)

1.1   Trapezoidal rule

1.2   Simpson's rule

1.3   Newton-Cotes formulas

1.3.1   General idea

Trapezoid and Simpson's Rules are part of a more general set of integration rules: * Break your interval into small **equal** sub-intervals, * approximate your function by a polynomial of some degree, e.g. * 0 for mid-point rule (that's just summing all elements and multiplying by h) * 1 for Trapz, * 2 for Simpson on that sub-interval. * this class of methods leads to Newton-Cotes (N-C) formulas.

- All Newton-Cotes formulas can be written in the form:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N+1} w_k f(x_k).$$

- w_k: "weights".
- x_k: "sample points". Notice above we are using $N+1$ points (N slices) to sample.
- N-C formulas of degree N: exact for polynomials of degree N (which require $N+1$ points to determine)
- For N-C formulas, the sample points are **evenly spaced**.

1.3.2 Examples

- All Newton-Cotes formulas can be written in the form:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N+1} w_k f(x_k).$$

Example: trapezoidal rule

$$I(a,b) = h \left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{n=1}^{N-1} f(a+nh) \right].$$

- weights:
  - $w_k = h/2$ for $k = 1$ or $N+1$,
  - $w_k = h$ otherwise.

- All Newton-Cotes formulas can be written in the form:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N+1} w_k f(x_k).$$

Example: Simpson's rule

$$I(a,b) = \frac{h}{3} \left[f(a) + f(b) + 4 \sum_{\substack{n \text{ odd} \\ 1...N}} f(a+nh) + 2 \sum_{\substack{n \text{ even} \\ 2...N-1}} f(a+nh) \right].$$

- weights:
  - $w_k = h/3$ for $k = 1$ or $N+1$,
  - $w_k = 4h/3$ for $k = 2, 4, \ldots, N$ (recall: N even),
  - $w_k = h$ for $k = 3, 5, \ldots, N-1$,

1.3.3 Generalization

| Degree	Polynomial	Coefficients
1 (trapezoidal)	Straight line	$\frac{1}{2}, 1, 1, \ldots, 1, \frac{1}{2}$
2 (Simpson)	Parabola	$\frac{1}{3}, \frac{4}{3}, \frac{2}{3}, \frac{4}{3}, \ldots, \frac{2}{3}, \frac{4}{3}, \frac{1}{3}$
3	Cubic	$\frac{3}{8}, \frac{9}{8}, \frac{9}{8}, \frac{3}{4}, \frac{9}{8}, \frac{9}{8}, \frac{3}{4}, \ldots, \frac{9}{8}, \frac{3}{8}$
4	Quartic	$\frac{14}{45}, \frac{64}{45}, \frac{8}{15}, \frac{64}{45}, \frac{28}{45}, \frac{64}{45}, \ldots, \frac{64}{45}, \frac{14}{45}$

1.4 Gaussian quadrature

1.4.1 Presentation

Newton-Cotes: * had to use N equally-spaced sampled points. * N^{th}-order N-C exact for polynomial of degree N. * A Nth-order polynomial approximates a well-behaved function better than a $(N-1)^{\text{th}}$-order polynomial, because of the added degree of freedom.

Gaussian quadrature: * N unequally-spaced points $\Rightarrow N$ more degrees of freedom, * exact for $(2N-1)^{th}$-order polynomial. * other way to look at it: it will give the same level of accuracy as an approximation by a $(2N-1)^{th}$-order polynomial.

Remarkably, there is a universal rule to choose the w_k and x_k: * $x_k = $ roots of nth Legendre polynomial $P_N(x)$, * $w_k = \left[\dfrac{2}{1-x^2} \left(\dfrac{dP_N}{dx} \right)^{-2} \right]_{x=x_k}$.
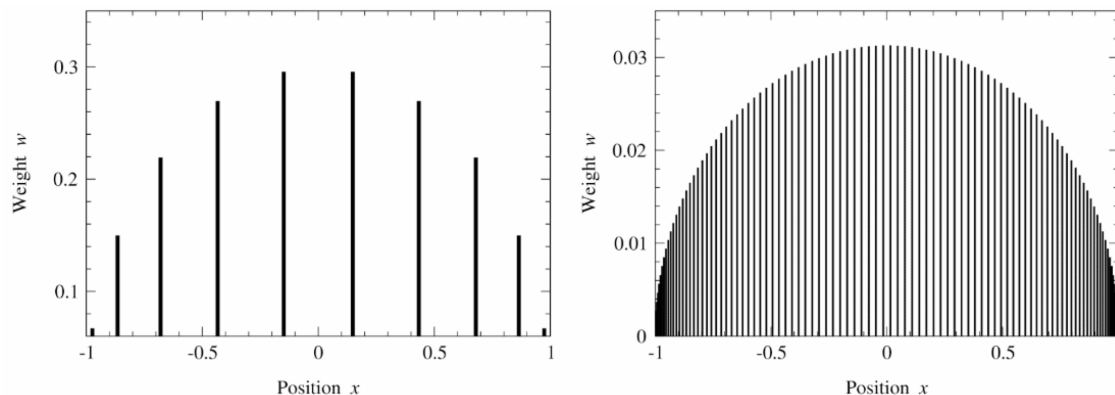
- That there is such a universal rule is beautiful (see Appendix C of textbook), but in the context of this course, we'll just accept that it works.
- Don't even write your own program to find sample points and weights. Usually use given subroutines. We will have some practice next week on how to find them.
- E.g. you have
    - gaussxw.py for integration limits from -1 to $+1$,
    - gaussxwab.py for integration limits from a to b.
- The calculation of weights and points is expensive. Use gaussxw.py if you are going to change the limits repeatedly (and see end of §5.6.1, pp. 167-168, for how to do).

```python
# %load gaussxw
from pylab import *
def gaussxw(N):

    # Initial approximation to roots of the Legendre polynomial
    a = linspace(3,4*N-1,N)/(4*N+2)
    x = cos(pi*a+1/(8*N*N*tan(a)))

    # Find roots using Newton's method
    epsilon = 1e-15
    delta = 1.0
```

```
    while delta>epsilon:
        p0 = ones(N,float)
        p1 = copy(x)
        for k in range(1,N):
            p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx = p1/dp
        x -= dx
        delta = max(abs(dx))

    # Calculate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

    return x, w

def gaussxwab(N,a,b):
    x, w = gaussxw(N)
    return 0.5*(b-a)*x + 0.5*(b+a), 0.5*(b-a)*w
```

Pros * complicated error formula, but in general: approximation error improves by a factor c/N^2 when you increase # of sample points by 1! * e.g., going form $N = 10$ to $N = 11$ sample points improves your estimate by a factor of $\sim 100 \Rightarrow$ converge very quickly to true value of the integral.

Cons * only works well if function is reasonably smooth (since sample points are farther apart), * really hard to get an accurate estimate of the error, if needed.
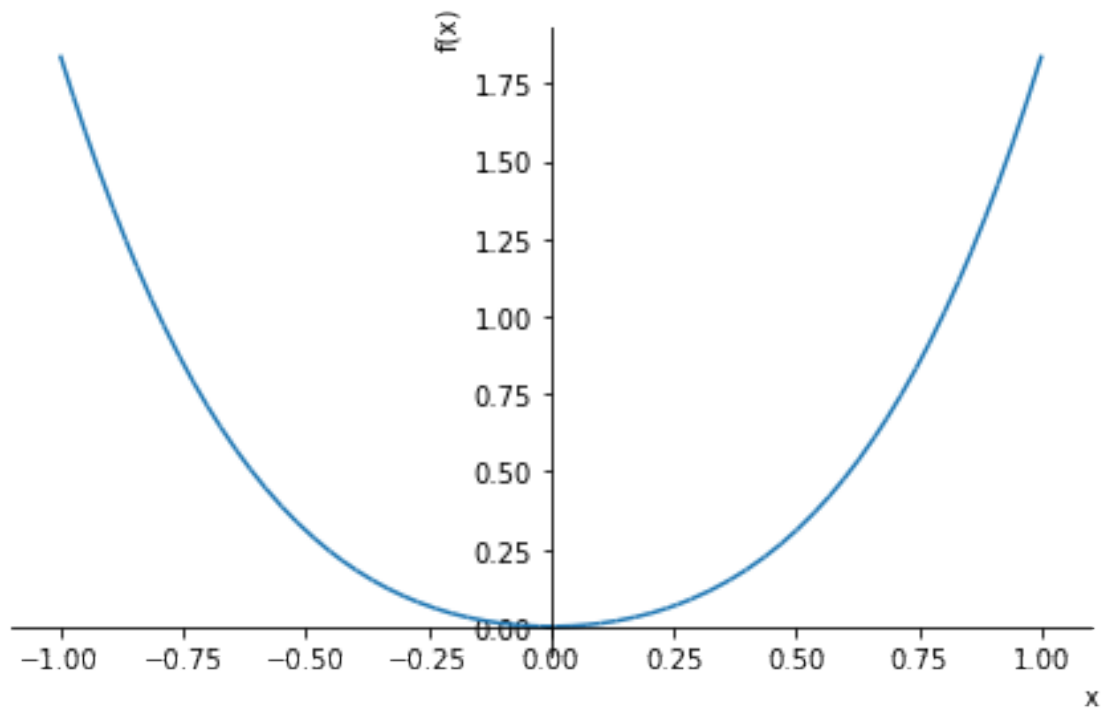
1.4.2   Example:

Compute $\int_{-1}^{1} \left[x^4 + \sin(x^2) \right] dx$ using Gaussian quadrature.

Let's see what is going on with sympy, Python's symbolic math package:

```
[2]: from sympy import *
     init_printing()
     x = symbols('x', real=True)
     f = x**4 + sin(x**2)
     plotting.plot(f, (x, -1, 1))
```

[2]: `<sympy.plotting.plot.Plot at 0x7f8761e2dc40>`

[3]:
```
# Integrate f
integrate(f, x)
```

[3]:
$$\frac{x^5}{5} + \frac{3\sqrt{2}\sqrt{\pi}S\left(\frac{\sqrt{2}x}{\sqrt{\pi}}\right)\Gamma\left(\frac{3}{4}\right)}{8\Gamma\left(\frac{7}{4}\right)}$$

[ ]:
```
integrate(f, x)
```

[4]:
```
# from -1 to +1
```

[5]:
```
res=integrate(f, (x, -1, +1))
```

[6]:
```
# Numerical evaluation
N(res)
```

[6]:
1.02053660344676

Now, we could do this as long as the function has an analytical expression that sympy (or Maple, Mathematica, Wolfram Alpha...) knows how to solve.

Gaussian quadrature is more reliable for arbitrary functions, let's try.

```
[ ]:  # This cell for pseudo-code
      # NOTE HOW I KEEP IT AS A SEPARATE BLOCK OF CODE
      # Load Newman's functions for gaussxw
      # Load numpy
      # define function
      # define N
      # call gausswx for xi, wi
      # initialize integral to 0.
      # loop over sample points to compute integral
      # print
```

```
[7]:  # This cell for code
      # Load Newman's functions for gaussxw
      # Load numpy
      import gaussxw as gsx
      import numpy as np
      # define function
      def f(x):
          """ lil description """
          return x**4 + np.sin(x**2)

      # define N
      N = 10
      # call gausswx for xi, wi
      x, w = gsx.gaussxw(N)

      # initialize integral to 0.
      I = 0.
      # loop over sample points to compute integral
      for k in range(N):
          I += w[k]*f(x[k])
      # print
      print(I)
```

1.0205366034471288

2 Numerical derivatives

- Simpler than numerical integration, in a way.
- Computing errors is usually a doozey though.
- Based on Taylor series approximations.
- Use Taylor series approximations to estimate errors.

1. Forward difference approximation: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$,

2. Backward difference approximation: $f'(x) \approx \dfrac{f(x) - f(x-h)}{h}$,

2.1 Basic philosophy for error estimation

Use Taylor series to find error in these approximations:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + h.o.t.$$

Isolate for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + h.o.t.$$

$\Rightarrow$ error is 1st-order in h (same is true for backward difference method).

2.2 Central differences

- Using Taylor series to find sneaky improvements to finite difference (FD) schemes.
- Example: central FD method:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

- Notice it still only involves subtracting 2 points, it's just that the location of the 2 points is different.

- Error:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + h.o.t.$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + h.o.t.$$

- Substract:

$$f(x+h) - f(x+h) = 2hf'(x) + \frac{h^3}{3}f'''(x) + h.o.t.$$

- Isolate for $f'(x)$ and add:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \boxed{\frac{h^2}{3}f'''(x)} + h.o.t.$$

- So we see that this formula is accurate to 2nd order in h.
- Can get higher order methods by including more points (see table 5.1 on page 196).
- Might have to do different things near the boundaries.
- Partial derivatives: similar techniques
- Higher order derivatives (e.g. f'') similar techniques.

2.3 Roundoff error, a comeback

- Let's take another look at this formula:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + h.o.t.$$

- What happens when we consider roundoff error? Recall that substracting numbers that are close is dangerous!

- Each of the terms $f(x + h)$ and $f(x)$ have error $\sim \quad C|f(x)|$. Their difference will have approximate error $2Cf(x)$ ("worst case" error).
- So in fact there are two sources of error and this leads to (eqn. (5.91) in book):

$$\epsilon = \underbrace{\frac{2C|f(x)|}{h}}_{\text{round-off error}} + \underbrace{\frac{1}{2}h|f''(x)| + h.o.t.}_{\text{approximation error}}$$

- Factor $1/h \Rightarrow$ increasing N could increase the error.
- The error has a minimum when h satisfies

$$h = \sqrt{4C \left| \frac{f(x)}{f''(x)} \right|} \qquad (5.93).$$

   (*Note: x is not a variable in this discussion. $f(x)$ should be interpreted as the typical magnitude of f.*)
- Example: $f(x), f''(x) = O(1) \Rightarrow h \sim O(\sqrt{C}) = O(10^{-8})$. In this case, the error is $\epsilon = O(10^{-8})$, from

$$\epsilon = h|f''(x)| = \sqrt{4C\,|f(x)f''(x)|} \qquad (5.94).$$

- There are two points: there's a limit to the improvement you can obtain by going to finer resolution, and the precision expected on differentiation is orders of magnitude less than that of other operations we have discussed.