L02 - Errors and integrals

September 17, 2018

Author: Nico Grisouard, nicolas.grisouard@physics.utoronto.ca
Supporting textbook chapters for week 2: 4.3, 5.1-5.3.
This week's topics: * Numerical errors, * How to compute integrals (basic methods).

1 Numerical errors

Aside from errors in programming or discretizing the physical model, there are 2 types of errors in computations: 1. Rounding errors: errors in how the computer stores or manipulates numbers. 2. Approximation errors (sometimes called truncation errors): errors in approximations to various functions or methods.

Reason: computers are machines, with inherent limitations.

Which begs the question: how does a computer represent a number?

1.1 Preliminary step: integers and floats

- Variables `a`, `b` and `c` below are all equal to 1000.
- Yet, `a` is treated very differently by da python than `b` and `c`:

```
In [1]: a = 1000
        b = 1000.
        c = 1e3
        print('The type of a is', type(a))
        print('The type of b is', type(b))
        print('The type of c is', type(c))
```

```
The type of a is <class 'int'>
The type of b is <class 'float'>
The type of c is <class 'float'>
```

1.2 Error #1: rounding errors:

1.2.1 General principle

- Integer numbers: python does not limit the number of digits stored. Can cause problems!

```
In [9]: print(2**100000)  # increase the exponent from 10 to 10,000,000
```

```
99900209301438450794403276433003359098042913905418169177152927386314583246425734832748733133244
```

- Non-integer numbers with more than 16 significant figures: **rounding** after 16 figures.
- Each mathematical operation (FLOPS, Floating-point Operations Per Second) introduces errors in the 16th digit. You can't assume `1.3+2.4=3.7`, it might be `3.699999999999999` even though the 2 numbers you are adding only have 2 significant figures.
- Know this, accept it, work with it. You are better at adding 1.3 and 2.4 than your computer, but you are much slower. This is why we use computers.

There are also limitations as to the largest number representable in python, the closest to zero, etc. If you want to know what they are on your machine, you can run the code below:

```
In [11]: from numpy import finfo, float64, float32
         # float64 contains double-precision floats, float32 is single-precision
         print("attributes you can access in finfo(float64) ", dir(finfo(float64)))
         print( "maximum numbers in 64 bit and 32 bit precision: ",
               finfo(float64).max, finfo(float32).max)
         print( "minimum numbers in 64 bit and 32 bit precision: ",
               finfo(float64).min, finfo(float32).min)
         print( "epsilon for 64 bit and 32 bit: ",
               finfo(float64).eps, finfo(float32).eps)
         print( "Should be epsilon for this machine if it's 64 bit",
               float64(1)+finfo(float64).eps-float64(1))
         print( "Should be zero",
               float64(1)+finfo(float64).eps/2.0-float64(1))
```

```
attributes you can access in finfo(float64)  ['__class__', '__delattr__', '__dict__', '__dir__
maximum numbers in 64 bit and 32 bit precision:  1.7976931348623157e+308 3.4028235e+38
minimum numbers in 64 bit and 32 bit precision:  -1.7976931348623157e+308 -3.4028235e+38
epsilon for 64 bit and 32 bit:  2.220446049250313e-16 1.1920929e-07
Should be epsilon for this machine if it's 64 bit 2.220446049250313e-16
Should be zero 0.0
```

```
In [16]: finfo(float64).eps
```

```
Out[16]: 2.220446049250313e-16
```

1.2.2 A dangerous example

Remember this one?

```
In [17]: x = 1e20
         y = -1e20
         z = 1.
         print("(x+y) + z = ", (x+y)+z)
         print("x + (y+z) = ", x+(y+z))
```

```
(x+y) + z =  1.0
x + (y+z) =  0.0
```

And how about this one?

```
In [18]: 7./3. - 4./3 - 1.
```

```
Out[18]: 2.220446049250313e-16
```

1.2.3 Error constant

- Newman: $\sigma = C|x|$.
  - $x =$ number you want to represent.
  - $\sigma =$ standard deviation of error
  - $C =$ fractional error for a single floating point number
- For 64 bit float: $C \sim 10^{-16} \sim$ machine precision ϵ_M.
- We cannot know a number better than this on the computer (otherwise it wouldn't be a limit on the precision).
- This fractional error is different on different computers but should not depend on x.

1.2.4 Propagation of errors

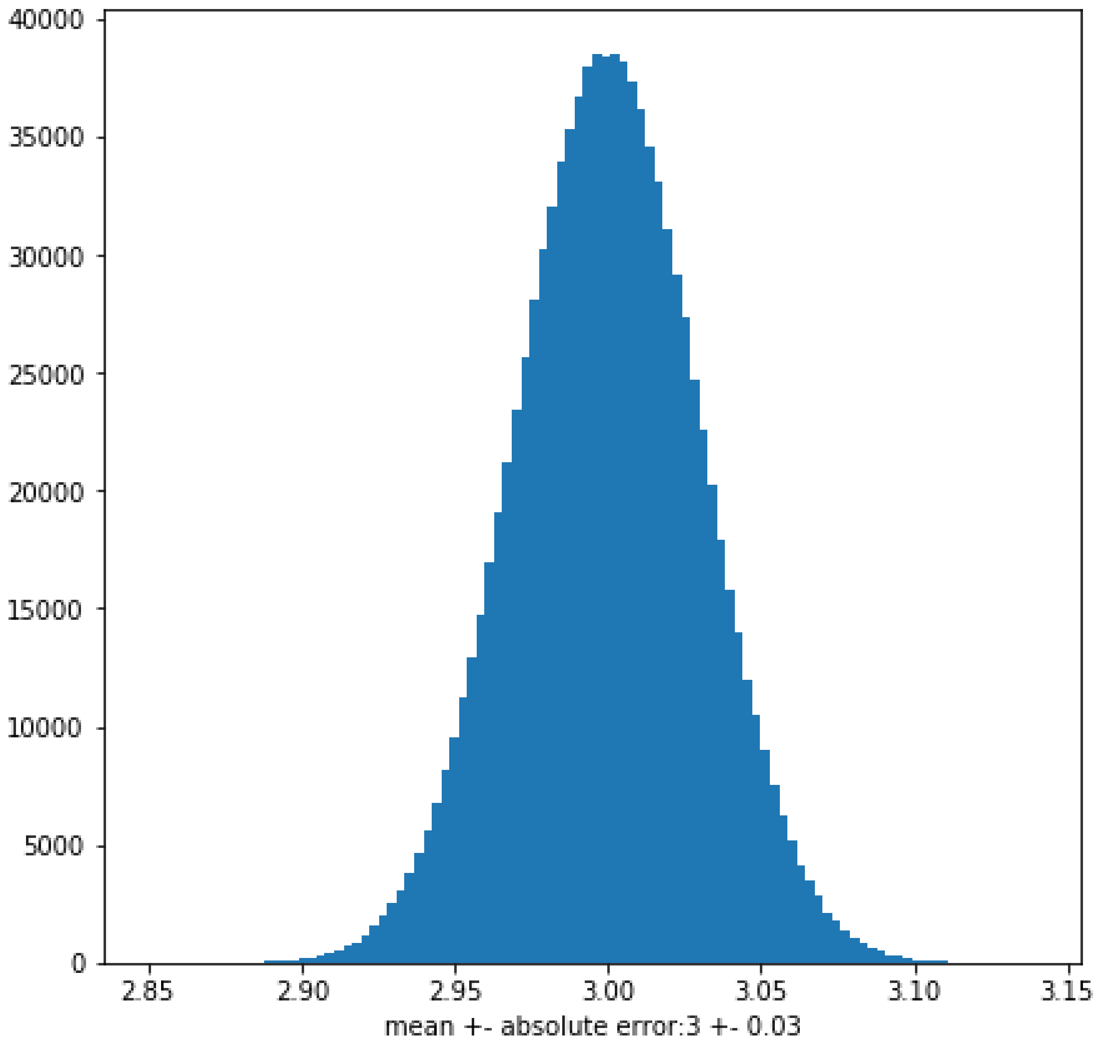
- Errors propagate statistically like they do in experimental physics.
- Example that follows: let $C = 1\%$ (to exagerate the effects) and add two numbers.

```
In [19]: # example to illustrate machine error using large error constant C
         # to make it easier to see graphically

         from numpy.random import normal  # import normal distribution
         from numpy import arange
         # import plotting functions below:
         from pylab import hist, show, subplot, figure, xlabel, xticks, legend, savefig
         from matplotlib import rcParams  # import rcparams to change font size
         rcParams.update({'font.size':14, 'legend.fontsize':10})

         # define array size
         N = 1000000
         # define number of bins for histogram
         N_Bins = 100
         # define C, which is our simulated error constant
         C = 1e-2
```

```
In [20]: # define numbers
         (x1, x2) = (3, -3.2)
         # define error standard deviations in terms of C
         sigma1 = C*abs(x1)
         sigma2 = C*abs(x2)
```

```
        # define distributions to those numbers satisfying sigma = Cx
        # This is how we simulate error.
        d1 = normal(loc=x1, scale=sigma1, size=N)
        d2 = normal(loc=x2, scale=sigma2, size=N)

        # then add up the distributions
        # then calculate the distribution of the sums.
        sumd = d1 + d2
```

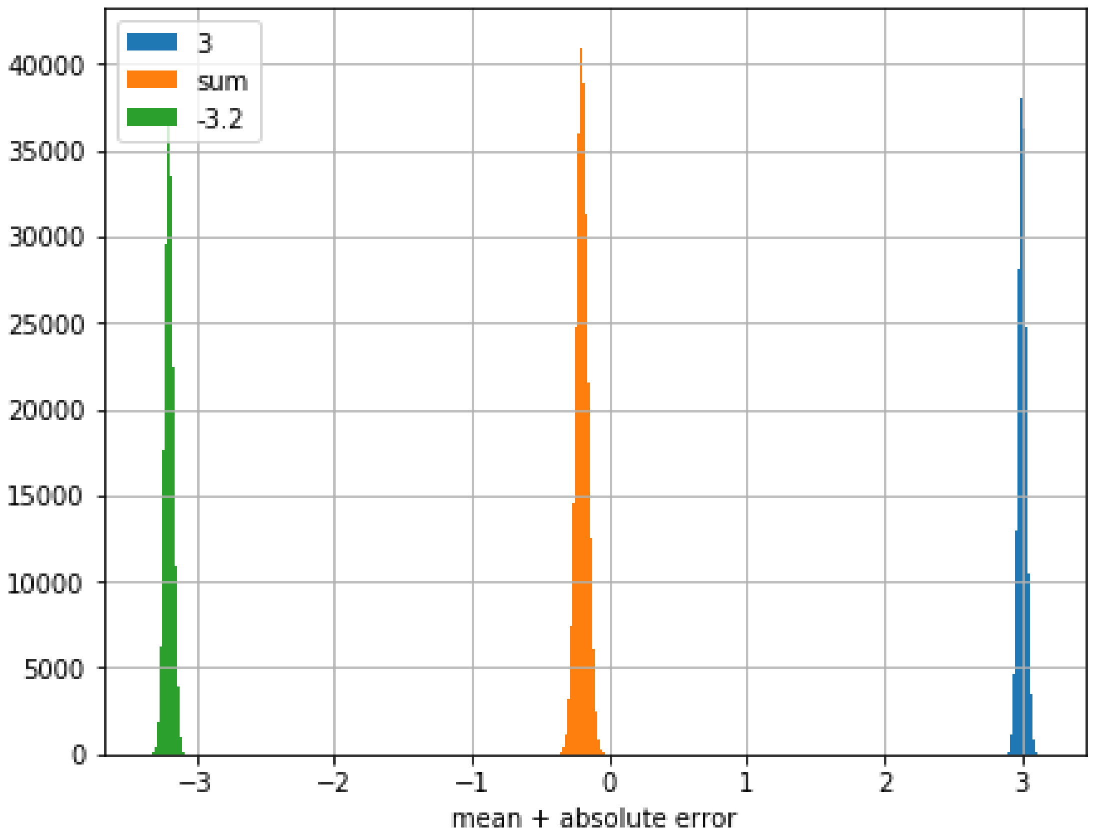
```
In [21]: # this plot just shows the a single number and its error
         figure(2, figsize=((7, 7)))
         hist(d1, N_Bins, histtype='stepfilled')
         xlabel('mean +- absolute error:' + str(x1) + ' +- ' + str(sigma1))
         savefig('MachineError_x1=' + str(x1) + '.png')
```



```
In [ ]: show()
```

```
In [22]: figure(1, figsize=((7, 12)))
         # plot histograms of the two numbers and their sum
         ax = subplot(2, 1, 1)
         hist(d1, N_Bins, histtype='stepfilled')
         hist(sumd, N_Bins,  histtype='stepfilled')
         hist(d2, N_Bins, histtype='stepfilled')
         leg=legend((str(x1), 'sum', str(x2)), loc='upper left')
         ax.grid(True)
         xlabel('mean + absolute error')
```

```
Out[22]: Text(0.5,0,'mean + absolute error')
```

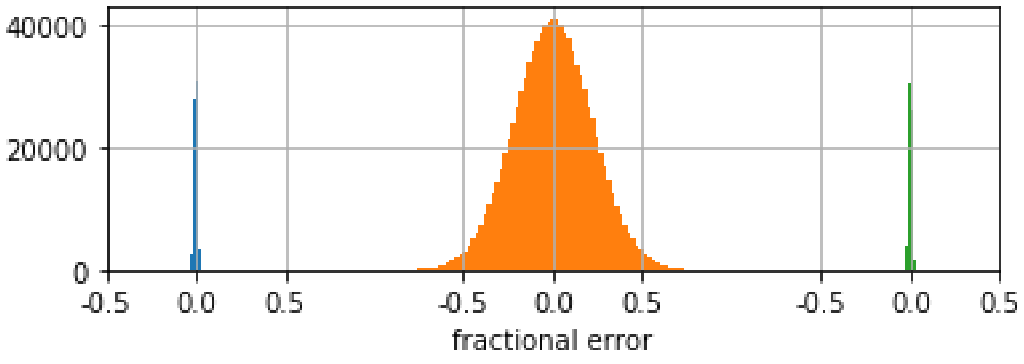


```
In [ ]: show()
```

```
In [23]: # then plot fractional error by diving by the mean values.
         # fractional errors will be large for opposite-signed large numbers
         # offset and spread have to be adjusted to make the plots clear.
         ax = subplot(2, 1, 2)
         offset = 2
```

```
spread = offset/4.0
hist(-offset+(d1-x1)/x1, N_Bins, histtype='stepfilled')

hist((sumd-(x1+x2))/(x1+x2), N_Bins, histtype='stepfilled')
hist(offset+(d2-x2)/x2, N_Bins, histtype='stepfilled')

xticks([-offset-spread, -offset, -offset+spread, -spread, 0,
        spread, offset-spread, offset, offset+spread],
       [str(-spread), str(0.0), str(spread), str(-spread),
        str(0.0), str(spread), str(-spread), str(0.0), str(spread)])

ax.grid(True)

xlabel('fractional error')
```

```
Out[23]: Text(0.5,0,'fractional error')
```



```
In [ ]: show()
```

These errors in differences become **very important** when taking numerical derivatives:

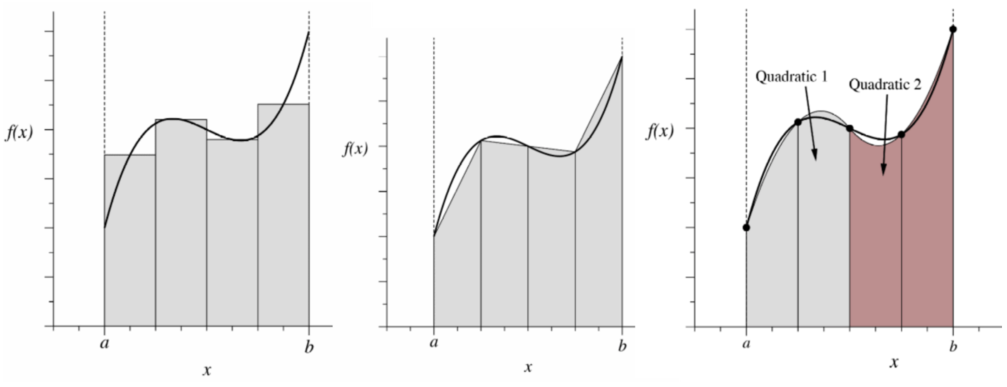
$$\frac{df}{dt} \approx \frac{f_{i+1} - f_i}{\Delta t} = \text{danger zone}$$

as we will see next week.

1.2.5 One important rule

Never, ever. Never ever ever. Do something like this:

```
In [24]: if 7./3. - 4./3. - 1. == 0:
             print('7/3 - 4/3 - 1 = 0')
         else:
             print('7/3 - 4/3 - 1 is not equal to 0')
```



From Newman, composite of figs. 5.1 and 5.2.

```
7/3 - 4/3 - 1 is not equal to 0
```

Instead:

```
In [26]: delta = 1e-15
         if abs(7./3. - 4./3. - 1.) < delta:
             print('7/3 - 4/3 - 1 = 0')
         else:
             print('7/3 - 4/3 - 1 is not equal to 0')
```

```
7/3 - 4/3 - 1 = 0
```

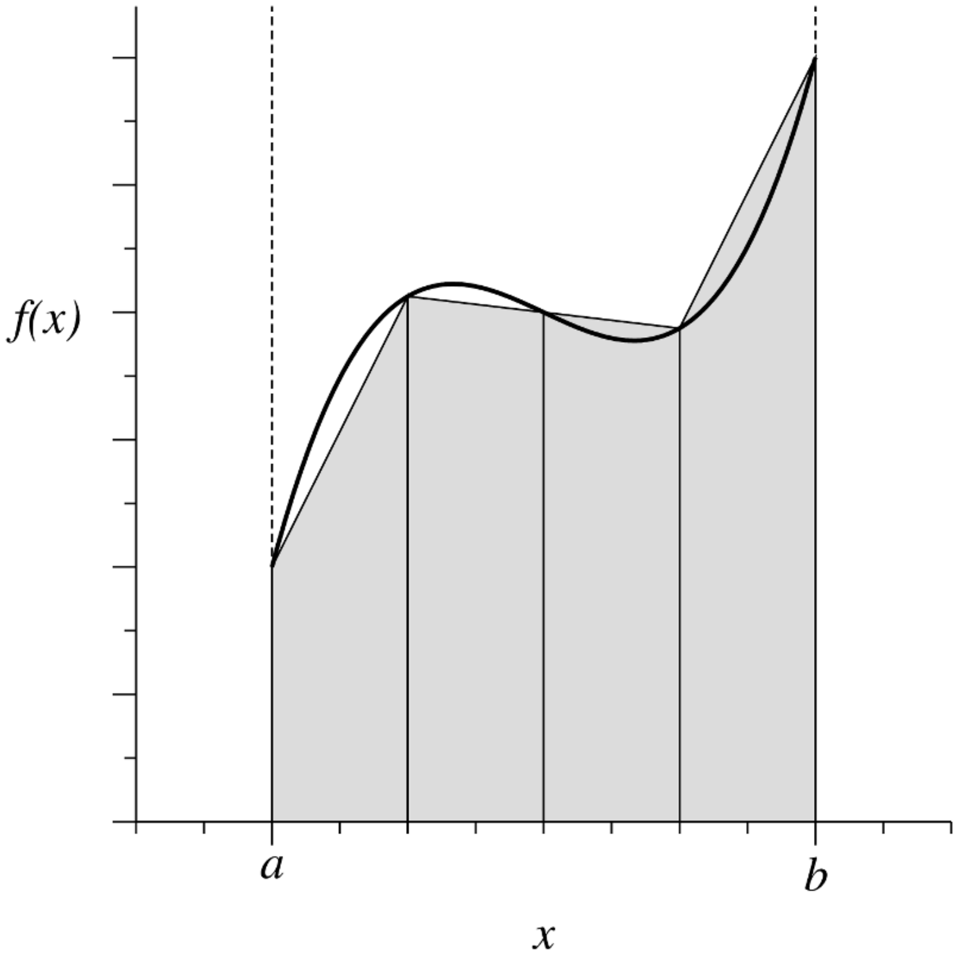
1.3 Error #2: Approximation errors

- errors introduced in functions due to approximations
- very important to consider for **integration** & **differentiation** algorithms
- we approximate these operations and there is an error in that approximation
- Most integration/differentiation algorithms are somehow based on Taylor series expansions, so you can usually figure out the approximation error by looking at the terms in the Taylor expansion that you ignore.

This should become clearer once we illustrate it with numerical integrations.

2 Numerical integration

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



From Newman: fig. 5.1b

2.1 Trapezoidal rule

- Break up internal into N slices,
- Approximate function as segments on each slice.

- N slices from a to b means that slice width:

$$h = (ba)/N$$

- area of 'k' slice's trapezoid: (Rectangle + Triangle)

$$A_k = f(x_k)h + \frac{h[f(x_k+h) - f(x_k)]}{2} = \frac{h[f(x_k) + f(x_k+h)]}{2}.$$

- Total area (our approximation for the integral) (and using $x_k = a + kh$):

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

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

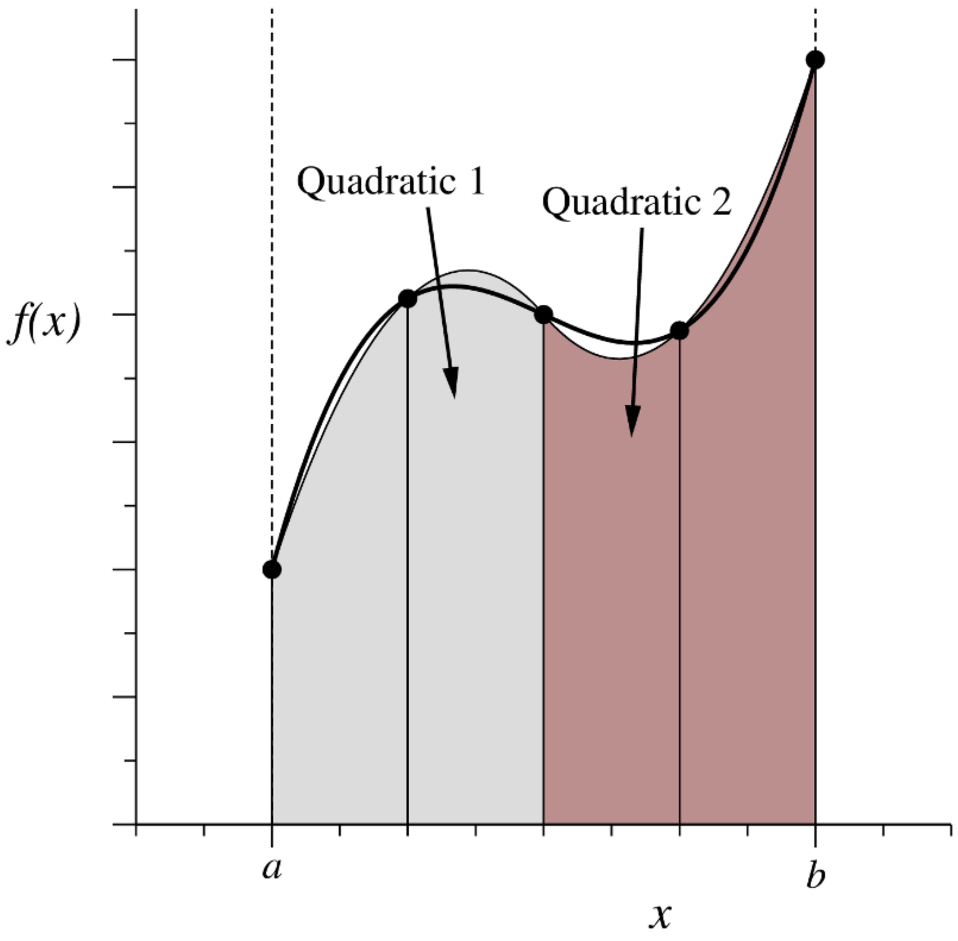
For future reference, let's do a symbolic integral of $f(x) = x^4 - 2x + 1$ on the interval $[0,2]$

```
In [33]: import sympy  # the symbolic math package
         sympy.init_printing()
         xs = sympy.Symbol('xs', real=True)  # the variable of integration
         sympy.integrate(xs**4 - 2*xs + 1, (xs,0,2.))

Out[33]:
```

$$4.4$$

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

```
In [43]: # Newman's example on pp. 142-143
         def f(x):
             return x**4 - 2*x + 1

         N = 100  # number of slices; try and increase it
         a = 0.0  # beginning of interval
         b = 2.0  # end of interval
         h = (b-a)/N  # width of slice

         s = 0.5*f(a) + 0.5*f(b)  # the end bits
         for k in range(1,N):  # adding the interior bits
             s += f(a+k*h)  # and

         print(h*s)

4.401066656
```



From Newman: fig. 5.2

2.2 Simpson's rule

- Break up internal into N slices,
- approximate function as a **quadratic** for every 2 slices
- need 2 slices because you need 3 points to define a quadratic
- more slices $\Rightarrow$ better approximation to function
- Number of slices need to be even! If uneven, either discard one, or use trapezoidal rule on one slice.

- Area of each 2-slice quadratic (see text for formula):

$$A_k = \frac{h}{3}\left[f(a+(2k-2)h) + 4f(a+(2k-1)h) + f(a+2kh)\right].$$

- Adding up the slices:

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

- In python, you can easily sum over even and odd values:

`for k in range(1, N, 2)` for the odd terms, and
`for k in range(2, N, 2)` for the even terms.

2.3 Error estimation

- If you tried the trapezoidal integration routine, you noticed that the error (difference between true value of the integral and computed value) goes down as N increases.
- How fast?

2.3.1 Euler-MacLaurin formulas

Example, for the trapezoidal rule:

$$I(a,b) = \int_a^b f(x)dx = \underbrace{h\left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a+kh)\right]}_{\text{the method}} + \underbrace{\epsilon}_{\text{the error}}$$

- using Taylor expansions, can find order of error for different Newton-Cotes (see text; trapeze and Simpson are examples) formulas.
- These error formulas are called "Euler-MacLaurin" formulas.

- Trapezoidal rule is a "1st-order" integration rule, i.e. accurate up to and including terms proportional to h. Leading order approximation error is of order h^2:

$$\boxed{\epsilon = \frac{h^2[f'(a) - f'(b)]}{12} + h.o.t.}$$

(see text for derivation)

- Simpson's rule is a "3rd-order" integration rule, i.e., accurate up to and including terms proportional to h^3. Leading order approximation error is of order h^4 (even though we go from segments to quadratics!)

$$\boxed{\epsilon = \frac{h^4[f'''(a) - f'''(b)]}{180} + h.o.t.}$$

(see text for non-derivation, and typo)

2.3.2 A more practical estimate

- What if you don't know f', f''', etc.?
- if you know the order of the error, (e.g., $\epsilon \propto h^2$ for trapeze), there is a way.

1. choose N intervals, compute $I_N(a,b)$. You **know** that

$$I(a,b) = I_N(a,b) + Ch^2$$

(for trapz). But you don't know C.

2. Double N: compute $I_{2N}(a,b)$. You **know** that

$$\underbrace{I(a,b)}_{\textit{true}} = \underbrace{I_{2N}(a,b)}_{\textit{computed}} + \underbrace{C\left(\frac{h}{2}\right)^2}_{\textit{error}} = I_{2N}(a,b) + \frac{C}{4}h^2.$$

   C does not change: e.g., for trapz, $C = [f'(a) - f'(b)]/12$.
3. Both equations above are $= I(a,b)$, the "true" value. Substract and re-arrange:

$$\boxed{\underbrace{\frac{I_{2N}(a,b) - I_N(a,b)}{h^2}}_{\textit{known}} = \frac{3}{4}\underbrace{C}_{\textit{unknown}}}$$

- For Simpson's rule, we would find:

$$\frac{I_{2N}(a,b) - I_N(a,b)}{h^4} = \frac{16}{15}C.$$

- Works in principle for any Newton-Cotes formula.
- Basic idea:

  1. compute integral from N points,
  2. double N, compute again,
  3. compare.

- Note: the texbook uses a different convention, using $h/2$ as reference instead of my h, hence the different coefficients.