Quadrature
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/quadrature/quadrature.pdf

Land surveyors use quadrature to estimate acreage.

Quadrature

- *Quadrature is the evaluation or estimation of integrals;*
- *The* `sympy` *package does symbolic integration for "reasonable" problems.*
- *Some easy quadrature methods include Riemann sums, trapezoid rule, Simpson's rule.*
- *Gauss quadrature can give high accuracy for smooth integrands;*
- `scipy` *includes many quadrature functions, such as* `quad()`*;*
- *the Monte Carlo method uses random sampling;*
- *2D regions such as the rectangle, triangle, and polygons can be handled as well.*

1 Estimating area and integrals

We understand an integral $I(f) = \int_z^b f(x)\,dx$ as the limit of finite sums of the form $I(x) \approx \sum_{i=0}^{i<n} f(x_i)h_i$, where the n points x_i are distributed in the interval $[a, b]$, broken into subintervals containing x_i, and having width h_i. The limit process increases n and drives the the maximum width h_i towards zero.

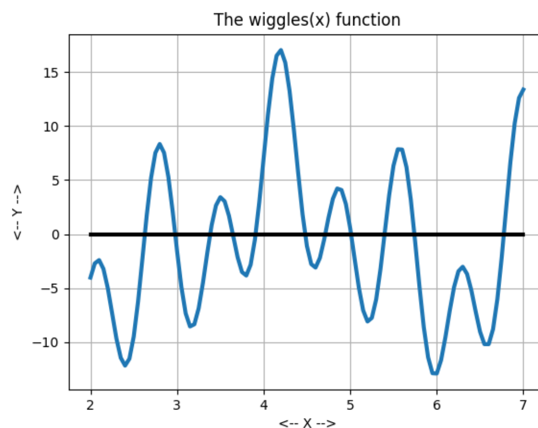
This fact suggests that almost any method that samples the region without big gaps will be able to approximate integrals, as long as we are willing to increase n sufficiently. So while almost any method will get there eventually, we will prefer methods that approximate integrals efficiently.

Before focusing on numerical methods, however, we will start by looking at a symbolic approach that gives us exact answers - if our integrand has a standard form for which a formulaic result can be determined.

2 Test problems

We will compare several integration procedures against test functions whose integrals we know.

The `wiggles` function is oscillatory, involving several frequencies:



Plot of $y = wiggles(x)$ *over* $2 \leq x \leq 7$.

The function has the formula

$$y(x) = \cos(x) + 5cos(1.6x) - 2\cos(2x) + 5\cos(4.5x) + 7cos(9x)$$

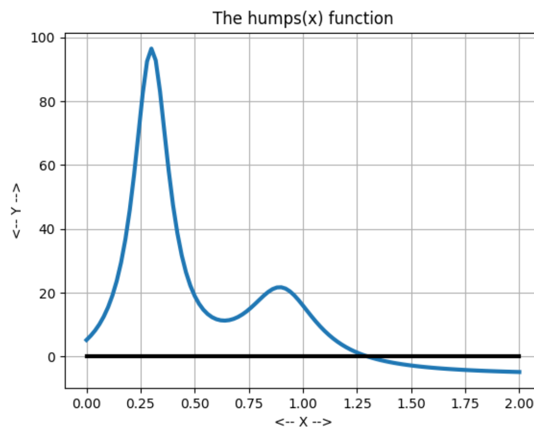
which we could write in Python as

```
def wiggles ( x ):
  import numpy as np
  value =        np.cos ( x )
        + 5.0 * np.cos ( 1.6 * x ) \
        - 2.0 * np.cos ( 2.0 * x ) \
        + 5.0 * np.cos ( 4.5 * x ) \
        + 7.0 * np.cos ( 9.0 * x )
  return value
```

The antiderivative $ya(x)$ is easy to compute and so we can determine the definite integral from 2 to 7:

$$\int_2^7 \text{wiggles}(x)\,dx = ya(7) - ya(2) = -4.5275696251606720278...$$

The `humps` function rises and falls very sharply over a short interval:



Plot of $y = humps(x)$ *over* $0 \leq x \leq 2$.

The function has the formula

$$y(x) = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6$$

which we would write in Python as

```
def humps ( x ):
  value = 1.0 / ( ( x - 0.3 )**2 + 0.01 ) \
        + 1.0 / ( ( x - 0.9 )**2 + 0.04 ) \
        - 6.0
  return value
```

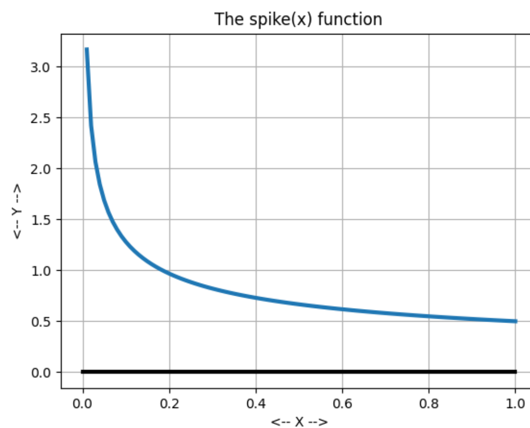
The antiderivative is

$$ya(x) = \frac{1}{0.1} \cdot \arctan(\frac{x-0.3}{0.1}) + \frac{1}{0.2} \cdot \arctan(\frac{x-0.9}{0.2}) - 6 \cdot x$$

In particular, then, we can determine the definite integral from 0 to 2:

$$\int_0^2 \text{humps}(x)\,dx = ya(2) - ya(0) = 29.326213804391148...$$

The `spike` function is singular at $x = 0$:



Plot of $y = spike(x)$ *over* $0 \leq x \leq 1$. $spike(0) = \infty$

The function has the formula

$$y(x) = \frac{1}{x^{\frac{1}{2}} + x^{\frac{1}{3}}}$$

The value of the definite integral from 0 to 1 is:

$$\int_0^1 \text{spike}(x)\,dx = 5 - 6\log(2) = 0.8411169166403281435 0...$$

3 Symbolic integration using sympy

The `sympy` package is a library of Python functions which can carry out certain symbolic calculations. A symbolic package will return a formula for the exact answer to a problem, as long as it knows a procedure for determining such an answer. A calculus textbook is full of problems for which such an answer can be

found. In practice, and in real life, most problems don't have solutions that can be expressed as a simple mathematical formula. Still, exact formulas are very useful when they can be found, and so we will begin by looking at how this package can be used.

After we import the `sympy` package as `sym`, we will use the function `sym.integrate()` to request our results. To do so, we have to express our integrand $f(x)$ using symbolic variables. Using the function `sym.symbols()`, it is enough to declare a few variables such as `x` and maybe `y`. Then we express the integrand as a formula involving the symbolic variable, and we get back an expression for an antiderivative of $f(x)$:

```
  x = sym.symbols ( 'x' )
  sym.integrate ( 3*x**2 + 6*x - 7 )

>> x**3 + 3*x**2 - 7*x
```

Now suppose that we actually wanted a definite integral? Then we add a second piece of information, containing the variable of integration, and the lower and upper bounds:

```
  sym.integrate ( 3*x**2 + 6*x - 7, ( x, 0, 1 ) )

>> -3
```

Of course, in this case, you can verify the result by evaluating the antiderivative at 0 and 1.

If our integrand has the form $g(x, y)$, then we can request the antiderivative or definite integral with respect to either x or y.

```
  sym.integrate ( x**2 * sym.cos ( y ), x )
  sym.integrate ( x**2 * sym.cos ( y ), ( x, 0, 1 ) )
  sym.integrate ( x**2 * sym.cos ( y ), y )
  sym.integrate ( x**2 * sym.cos ( y ), ( y, 0, sym.pi ) )
```

We can even do a double integral $\int_{\pi/2}^{\pi} \int_{0}^{x^2} \frac{1}{x} \cos(\frac{y}{x})\, dy\, dx$, by integrating y first and then x:

```
  i1 = sym.symbols ( 'i1' )
  i1 = sym.integrate ( (1/x) * sym.cos (y/x), ( y, 0, x**2 ) )
  i2 = sym.integrate ( i1, ( x, sym.pi/2, sym.pi ) )
>> 1
```

Now let's try our test function, for which we know there is an exact antiderivative. We issue the command:

```
  result = sym.integrate ( 1.0 / ( ( x - 0.3 )**2 + 0.01 ) \
                         + 1.0 / ( ( x - 0.9 )**2 + 0.04 ) - 6.0, ( x, 0, 2 ) )
```

Sadly, `sympy` fails to find an answer, and instead prints a page of warning messages. It's not a perfect package, and the integrand is tricky to deal with. But this shows that even when there is an answer that can be expressed as a formula, a symbolic integration package may not necessarily be able to find it!

4 Left, middle, right Riemann rules

A Riemann rule estimates the integral of $f(x)$ over the interval $[a, b]$ using a mesh of n subintervals, each of length h_i, each with a representative point x_i, and then writing

$$\int_a^b f(x)\, dx \approx \sum_{i=0}^{i<n} f(x_i) \cdot h_i$$

We often use equal subintervals, so that $h_i \frac{b-a}{n}$; the point x_i is often chosen as the left endpoint, the midpoint, or the right endpoint of the corresponding subinterval, resulting in the left, middle, or right Riemann rule.

Here is how we could compute the left Riemann rule for the `humps()` function:

```
  h = ( b - a ) / n
  x = np.linspace ( a, b, n + 1 )
  s = h * np.sum ( humps ( x[0:n+1] ) )
```

We could make a more useful tool by writing a function `riemann_left(f,a,b,n)` so that we could call

```
a = 0.0
b = 2.0
n = 10
qn = riemann_left ( humps, a, b, n )
```

5 Error and convergence

We expect an integration rule to converge if we use enough points n in a fine mesh. But sometimes we program a rule incorrectly, or call a quadrature rule in the wrong way, and we don't get good results. And if we have two rules to choose from, it is useful to be able to determine which one will compute an accurate estimate most efficiently.

To study this behavior in practice, we start with a function $f(x)$ whose integral over some interval $[a, b]$ we know exactly, as the value `q*`. We assume we have an automatic procedure for generating a mesh with characteristic mesh spacing h, using an order of n evaluation points, with which we estimate the integral value as q_n. Then we can define the quadrature error as $e_n = |q * -q_n|$.

While, for a given value of n, one rule might give a smaller error than another, what is more interesting is the behavior of the errors as we increase n. A straightforward investigation involves repeatedly increasing n (or, equivalently, reducing h) and recording the values of e_n. While the initial error sequence might be somewhat irregular, we will usually expect that eventually, doubling n will reduce the error by some factor of 2. If the error is simply halved, we say convergence is linear. If the error goes down by about a factor of 4, we are seeing quadratic convergence, and so on.

For a given quadrature rule, we might try to compute this convergence behavior as follows:

```
a = 0.0
b = 2.0
q = humps_antiderivative ( b ) - humps_antiderivative ( a )
for i in range ( 2, 11 ):
  n = 2**i
  qn = riemann_left ( humps, a, b, n )
  e = np.abs ( q - qn )
  if ( 2 == n ):
      print ( n, e )
  else:
    r = e_old / e
    print ( n, e, r )
  e_old = e
```

```
   n       e      e old / e
 ----   -------   ---------
    2    8.2987
    4   11.3409   0.7317
    8   11.2727   1.0060
   16    3.0865   3.6522
   32    1.4372   2.1475
   64    0.6986   2.0571
  128    0.3446   2.0270
```

```
 256     0.1711   2.0133
 512     0.0853   2.0066
1024     0.0425   2.0033
```

If you study these results, you can see that the error actually goes up once at the beginning, resulting in an error ratio that is less than 1. After a few more irregular steps, the error ratio begins to settle down, apparently heading towards a value of 2. In other words, for this problem and this method, doubling the number of intervals cuts the error in half.

If we expect this behavior to continue indefinitely, we could even estimate the number of intervals necessary to drive the error below any tolerance we care to name. In fact, this very regular error reduction can only be expected to last until the point where we've come close to getting all 16 digits of accuracy, or other roundoff effects occur. After that, the error ratio will generally break down. That's not a fault of the method, or of the problem, but instead of the finite precision arithmetic we are using.

We will see other methods that tend to have a faster rate of convergence; doubling n might cut the error by a factor of 4, or 8, or even more. On the other hand, we will also come across integration problems that "defeat" our methods, producing error ratios that are clearly much lower than we expect. In general, our error ratio analysis works very well for integrands that are well approximated by a Taylor series, but can fail when we have an integrand involving discontinuities, square roots or absolute values, or negative powers of x, and other complications.

6 The trapezoid rule

The trapezoid quadrature rule uses $n+1$ points to divide the interval $[a,b]$ into n subintervals. Assuming these subintervals are equal, we have a constant width $h = (b-a)/n$. Over the i-th subinterval with endpoints x_i and x_{i+1}, we approximate the area under the curve by the area of the trapezoid, $h * \frac{f(x_i)+f(x_{i+1})}{2}$. Summing these up, we approximate the integral $I(f)$ by the quadrature sum $Q(f)$:

$$I(f) \approx Q(f) = h \cdot (\frac{1}{2} f(x_0) + f(x_1) + ... + f(x_{n-1} + \frac{1}{2} f(x_n))$$

The exercises ask you to write a function `trapezoid(f,a,b,n)` which carries out this estimate.

7 `numpy.trapz()`

The `numpy` package includes a function `trapz()` which can carry out the trapezoid rule. It can be used in the following way:

```
value = trapz ( y, x, dx )
```

where

- `y` is an array of function values;
- `x` is an array of function evaluation points.
- `dx` is the spacing between the evaluation points.

Only one of the arguments `x` or `dx` is usually supplied:

- Calling with only the `y` argument, the function assumes the data is equally spaced with a spacing of 1.
- Calling with `y` and `x` the function can handle variably spaced data;
- Calling with `y` and `dx`, the function assumes all subintervals are `dx` wide.

8 Simpson's rule

Simpson's (first) quadrature rule uses $2n+1$ points to divide the interval $[a, b]$ into n subintervals. Assuming these subintervals are equal, we have a constant width $h = (b-a)/n$. Each subinterval involves two endpoints and a midpoint. In the i-th subinterval these points will have the indices $2*i, 2*i+1, 2*i+2$. The integral over the subinterval is approximated by $h * \frac{f(x_{2i})+4f(x_{2i+1})+f(x_{2i+2})}{6}$. These coefficients are determined by integrating the quadratic function that passes through these three data points.

If we now sum up the integral estimates over each subinterval, we approximate the integral $I(f)$ by the quadrature sum $Q(f)$:

$$I(f) \approx Q(f) = \frac{h}{6} \cdot (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + ... + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n}))$$

The function values of odd-indexed points have a multiplier of 4. The first and last function values have a multiplier of 1. The function values of the remaining even-indexed points have a multiplier of 2.

The exercises ask you to write a function `simpson(f,a,b,n)` which carries out this estimate.

9 `scipy.integrate.simpson()`

The `scipy.integrate` package includes a function `simpson()` which can carry out Simpson's rule. It can be used in the following way:

```
value = simpson ( y, x, dx )
```

where

- `y` is an array of function values;
- `x` is an array of function evaluation points.
- `dx` is the spacing between the evaluation points.

Only one of the arguments `x` or `dx` is usually supplied:

- Calling with only the `y` argument, the function assumes the data is equally spaced with a spacing of 1.
- Calling with `y` and `x` the function can handle variably spaced data;
- Calling with `y` and `dx`, the function assumes all subintervals are `dx` wide.

The exercises ask you to compare the results of the `numpy.trapz()` and `scipy.integrate.simpson()`.

10 Gauss rules

Gauss quadrature rules provide very high accuracy as long as the integrand is a smooth function. Unlike our previous examples, Gauss rules do not use equally spaced points. Instead, a Gauss rule that uses k points is defined by specifying the location of each point ξ in a reference interval $[-1, +1]$, along with a corresponding weight ω. In particular, the k-point Gauss rule estimate for the integral of a function $f(x)$ over this reference interval is defined by

$$\int_{-1}^{+1} f(x)\, dx \approx Q(f) = \sum_{i=0}^{i<k} \omega_i f(\xi_i)$$

If we wish to apply a Gauss rule to an interval $[a, b]$, then we must transform the points and weights as follows:

$$w_i = \omega_i \cdot \frac{b-a}{2}$$
$$x_i = \frac{(1+\xi_i)b + (1-\xi_i)a}{2}$$

and now we can compute

$$\int_a^b f(x)\,dx \approx Q(f) = \sum_{i=0}^{i<k} w_i f(x_i)$$

If we have made an integral estimate, and we want to improve its accuracy, we can either use a higher order Gauss rule (increasing k) or we use a composite rule, that is, we can divide the original interval $[a, b]$ into m subintervals, and apply our current Gauss rule to each, summing the result.

The exercises ask you to implement a Gauss rule for $k = 4$. For this exercise, you will need the following information:

i	ξ_i	ω_i
0	-0.861136311594052575223946488893	0.347854845137453857373063949222
1	-0.339981043584856264802665759103	0.652145154862546142626936050778
2	0.339981043584856264802665759103	0.652145154862546142626936050778
3	0.861136311594052575223946488893	0.347854845137453857373063949222

11 `scipy.integrate.fixed_quad()`

The `scipy.integrate` package provides a function `fixed_quad(f,a,b,k)` which applies a Gauss quadrature rule of order k to estimate the integral of $f(x)$ over $[a, b]$. To seek a higher accuracy estimate, it is possible to increase k. However, this will only be useful as long as the integrand function is sufficiently smooth. The `fixed_quad()` function is not capable, by itself, of improving the integral estimate by subdividing the original interval.

The exercises ask you to try this method on a smooth integrand, for which increasing k should improve the estimate, and on an integrand that is not smooth, for which accuracy will increase much more slowly than we might expect.

12 The `scipy.integrate.quadrature()`

The best way to work with Gauss quadrature is to start with a rule of moderate order k, and then break down the original interval into subintervals, over which a more accurate estimate can be made. If even more accuracy is needed, then it is often better to refine the interval further, rather than raising the degree k of the quadrature rule. This approach can also include an attempt to estimate the error that is being made in the estimate. The `scipy.integrate` function `quadrature()` uses this approach to adaptively estimate an integral. It can be used by a call like

```
q, e = quadrature ( f, a, b, tol = ? )
```

where

- input `tol` is an absolute error tolerance, whose default value is 1.49e-08;
- output `q` is the integral estimate;
- output `e` is an estimate of the error in `q`;

The exercises ask you to use this function to estimate integrals to a given precision.

13 The Monte Carlo method

Surprisingly, there are many integration problems for which a random number approach can be useful. This usually occurs for problems involving a multidimensional integration region, or where the integrand includes some very nonlinear density function. However, the idea can be applied even to our simple 1D examples.
This approach is known as the Monte Carlo method, named after a famous gambling resort in Europe. It gets its inspiration from the fact that the integral of a function over an interval $[a, b]$ is equal to $(b-a)$ times the "average" value of the function. One way to get the average of the function is simply to choose a lot of x values, let's say n of them, at random in the interval, and average the function values at those points. In other words:

```
def monte_carlo ( f, a, b, n )
  from numpy.random import default_rng
  import numpy as np
  rng = default_rng ( )
  r = rng.random ( size = n )
  x = ( 1.0 - r ) * a + r * b
  fx = f ( x )
  q = ( b - a ) * np.sum ( fx ) / n
  return q
```

In practice, the Monte Carlo method is a very simple way to get a crude estimate of the result. However, roughly speaking, to expect one more digit of accuracy in the result, you have to increase your sample number n by a factor of 100. On the other hand, this requirement is true no matter what dimension you are integrating over, so for moderate dimensional integration regions, sometimes the Monte Carlo method is a very strong contender.

14 Exercises

1. Use the symbolic package to determine $\int_0^1 \int_x^{x+1} x + y \, dy \, dx$.
2. Implement the midpoint Riemann rule and use it to estimate the integral of the `humps()` function over $0 \leq x \leq 2$ using $n = 10$ intervals. Print the error.
3. Use your midpoint Riemann rule again, with $n = 10, 20, 40, 80, 160$ intervals. How does the error decrease as n is increased?
4. Implement the right Riemann rule and use it to estimate the integral of the `humps()` function over $0 \leq x \leq 2$ using $n = 20$ intervals. Print the error.
5. Use your right Riemann sum again, with $n = 10, 20, 40, 80, 160$ intervals. How does the error decrease as n is increased?
6. Write a Python function `trapezoid(f,a,b,n)` which uses the trapezoid rule to estimate the integral of $f(x)$ over the interval $[a, b]$ using n intervals. Use your function to estimate the integral of the `humps()` function over $0 \leq x \leq 2$ using $n = 20$ intervals. Print the error.
7. Use the `numpy` function `trapz()` to estimate the integral of the `humps()` function over $0 \leq x \leq 2$ using $n = 20$ intervals. Print the error.
8. Write a Python function `simpson(f,a,b,n)` which uses Simpson's rule to estimate the integral of $f(x)$ over the interval $[a, b]$ using n intervals. Use your function to estimate the integral of the `humps()` function over $0 \leq x \leq 2$ using $n = 20$ intervals. Print the error.
9. Write a Python function `gauss4(f,a,b)` which uses a 4 point Gauss rule to estimate the integral of $f(x)$ over the interval $[a, b]$. A table of points and weights is listed above. Use your function to estimate the integral of the `humps()` function over $0 \leq x \leq 2$. Print the error.
10. Use the `numpy` function `trapz()` and the `scipy.integrate` function `simpson()` to estimate the integral of the `wiggles()` function over $2 \leq x \leq 7$. Use equally spaced `x` arrays of length n = 2, 4, 8, 16, 32. For each n, print the errors made by the two methods.

11. Use the `scipy.integrate` function `fixed_quad()` to estimate the integral of the `humps()` function over $0 \leq x \leq 2$. Compute estimates for a Gauss rule of order $k = 1$ through 10. Print k, the error, and the convergence rate.
12. Use the `scipy.integrate` function `fixed_quad()` to estimate the integral of the `spike()` function over $0 \leq x \leq 1$. Compute estimates for a Gauss rule of order $k = 1$ through 10. Print k, the errors, and the convergence rate. You may expect convergence to be slow for this example.
13. Use the `scipy.integrate` function `quadrature()` to estimate the integral of `humps()` over $0 \leq x \leq 2$ to a tolerance of 1.0e-08. Is your quadrature estimate as accurate as you requested?
14. Use the `scipy.integrate` function `quadrature()` to estimate the integral of the `spike()` function over $0 \leq x \leq 1$ to an absolute error tolerance of 1.0e-08. Is your quadrature estimate as accurate as you requested?
15. Implement the Monte Carlo method, and use it to estimate the integral of the `humps()` function over $0 \leq x \leq 2$. Start with $n = 10$ and print the integral estimate and error. Then repeat the process by multiplying n by 10, until you have reached $n = 1,000,000$. How does your accuracy compare with other methods? WHat happens if you repeat a calculation (which will use a different set of random numbers)?