CHAPTER

15

Numerical Integration With Newton–Cotes Formulas

OUTLINE

Once the area is appointed,
They enclose the broad-squared plaza
With a mass of mighty tree-trunks
Hewn to stakes of firm endurance...

Alonso de Ercilla y Zúñiga,* The Araucaniad, *as translated by Charles Maxwell Lancaster and Paul Thomas Manchester

CHAPTER POINTS

- Newton–Cotes rules perform polynomial interpolation over the integrand and integrate the resulting interpolant.
- The midpoint, trapezoid, and Simpson's rules are based on constant, linear, and quadratic interpolation, respectively.
- The range of integration is typically broken up into several pieces and a rule is applied to each piece.
- We can combine Richardson extrapolation with Newton–Cotes rules to get highly accurate integral approximations.

In this and the next chapter we are going to discuss ways to compute the integral of a general function numerically. In particular we are interested in ways that we can approximate an

DOI: 10.1016/B978-0-12-812253-2.00017-0

integral by a sum with a finite number of terms:

$$\int_a^b f(x)\,dx \approx \sum_{\ell=1}^{L} w(x_\ell) f(x_\ell).$$

Such an approximate is called quadrature, but numerical integration is the more modern term. The term quadrature arose from a process in ancient Greek geometry of constructing a square (a quadrilateral) with the same area as a given shape.

Writing an integral as a finite sum is analogous to the definition of an integral as a Riemann sum, when the number of intervals goes to infinity. Therefore, just as in finite difference derivatives, we use finite mathematics to approximation the infinitesimals of calculus.

15.1 NEWTON–COTES FORMULAS

The Newton–Cotes formulas are ways to approximate an integral by fitting a polynomial through a given number of points and then doing the integral of that polynomial exactly. (Clearly, Newton is the larger numerical luminary in the name of this method. One might suspect that Cotes rode on Isaac Newton's coat tails here.) The polynomial can be integrated exactly because integration formulas for polynomials are straightforward. We will not delve into the general theory of Newton–Cotes formulas, rather we will give three important examples.

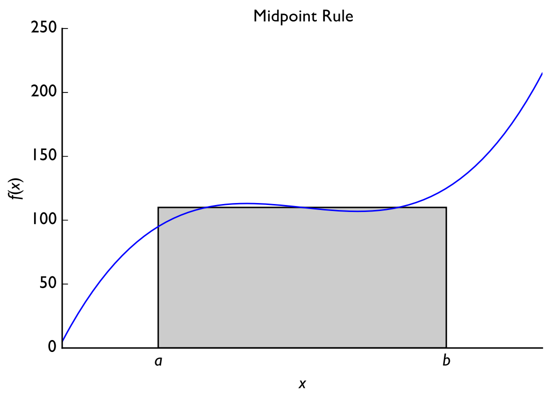
15.1.1 The Midpoint Rule

In the midpoint rule we approximate the integral by the value of the function in the middle of the range of integration times the length of the region. This simple formula is

$$I_{\text{midpoint}} = h\, f\left(\frac{a+b}{2}\right) \approx \int_a^b f(x)\,dx,$$

where $h = b - a$.

To demonstrate this rule we look at a simple function integrated over an interval with the midpoint rule:



From this demonstration, we see that the resulting approximation is not terrible, but there are clearly parts of the function where the rectangle does not match the function well. We can do better than a rectangle that approximates the function as flat. Namely we can approximate the integrand as linear; we do this next.

BOX 15.1 NUMERICAL PRINCIPLE

The midpoint rule approximates the integrand as a rectangle that touches the function at the midpoint of the interval of integration:

$$\int_a^b f(x)\,dx = hf(c),$$

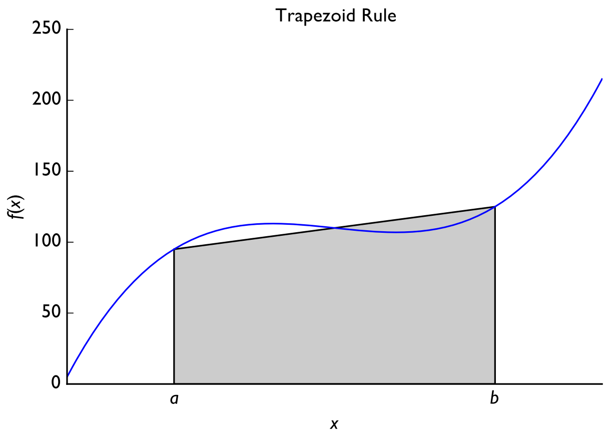
where $h = b - a$ and $c = (a+b)/2$.

15.1.2 The Trapezoid Rule

In this method we fit a line between a and b and then do the integration. The formula for this is

$$I_{\text{trap}} \equiv \frac{h}{2}(f(a) + f(b)) \approx \int_a^b f(x)\,dx,$$

where $h = b - a$. Here is a graphical example.



BOX 15.2 NUMERICAL PRINCIPLE

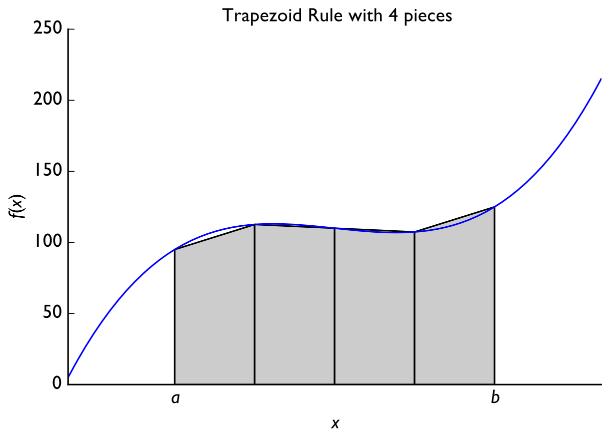
The trapezoid rule approximates the integrand as the area of a trapezoid with bases that touch the function at the endpoints of the interval of integration:

$$\int_a^b f(x)\,dx = \frac{h}{2}\left(f(a) + f(b)\right),$$

where $h = b - a$.

Additionally, in this demonstration we can see where the rule gets its name. The approximation to the integral is the area of a trapezoid. Indeed, the approximation formula is the same as the area of a trapezoid found in geometry textbooks. We can also see in the figure that the approximation is not exact because the trapezoid does not exactly follow the function, but if a and b are close enough together it should give a good approximation because any well-behaved function can be approximated linearly over a narrow enough domain.

That leads to a variation to the trapezoid rule (and any other rule for that matter). We can break up the domain $[a, b]$ into many smaller domains and integrate each of these. Here is an example where we break $[a, b]$ into 4 pieces:



BOX 15.3 NUMERICAL PRINCIPLE

Commonly, the interval of integration is broken up into many pieces and a quadrature rule is applied to each of the pieces. This allows the user to control the size of the interval over which the approximation is applied. This application of a quadrature rule over smaller ranges of the interval is called a composite quadrature rule.

As you can see the approximation is starting to look better. We can write a trapezoid rule function that will take in a function, a, b, and the number of pieces and perform this integration. Also, because the right side of each piece is the left side of the next piece, if we are clever we can only evaluate the function $N+1$ times where N is the number of pieces. The following function implements the trapezoid rule.

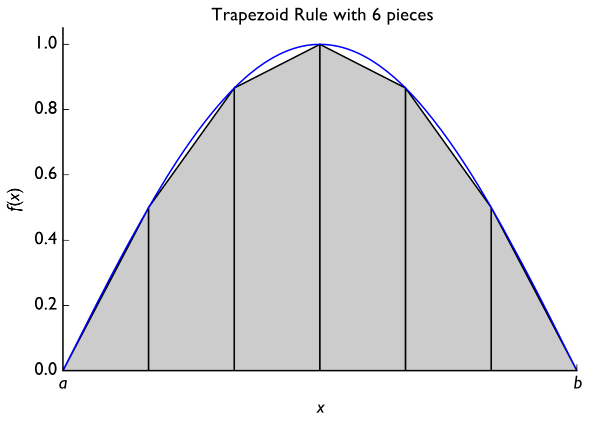
```
In [1]: def trapezoid(f, a, b, pieces):
            """Find the integral of the function f between a and b
            using pieces trapezoids
            Args:
                f: function to integrate
                a: lower bound of integral
                b: upper bound of integral
                pieces: number of pieces to chop [a,b] into

            Returns:
                estimate of integral
            """
            integral = 0
            h = b - a
            #initialize the left function evaluation
            fa = f(a)
            for i in range(pieces):
                #evaluate the function at the left end of the piece
                fb = f(a+(i+1)*h/pieces)
                integral += 0.5*h/pieces*(fa + fb)
                #now make the left function evaluation the right for the next step
                fa = fb
            return integral
```

We can test this method on a function that we know the integral of

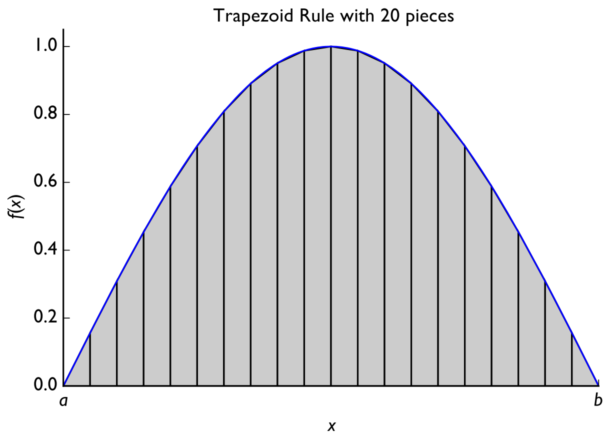
$$\int_0^{\pi} \sin x \, dx = 2.$$

In addition to the estimates, the approximations to the integral are plotted.

```
In [2]: integral_estimate = trapezoid(np.sin,0,np.pi,pieces=6,graph=True)
        print("Estimate is",integral_estimate,"Actual value is 2")

        integral_estimate = trapezoid(np.sin,0,np.pi,pieces=20,graph=True)
        print("Estimate is",integral_estimate,"Actual value is 2")
```

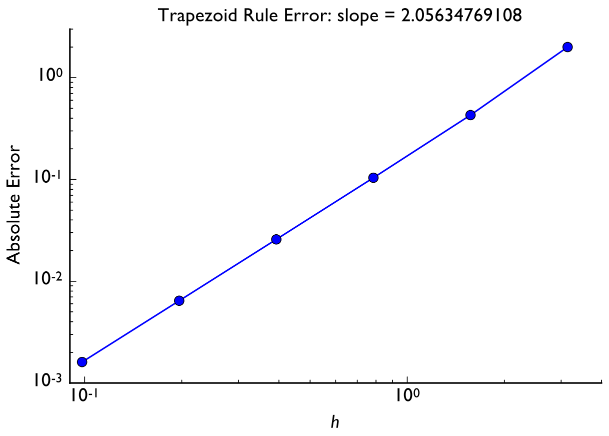


```
Estimate is 1.95409723331 Actual value is 2
```



```
Estimate is 1.99588597271 Actual value is 2
```

We can run this multiple times and see how the error changes. Similar to what we did for finite difference derivatives, we can plot the error versus number of pieces on a log-log scale. In this case, h is the width of each of the pieces: as the number of pieces grows, the value of h decreases.



The error in the trapezoid rule that we observe is second-order in h, because the slope of the error on the log-log scale is 2.

We can see that this is the expected error in the estimate by looking at the linear approximation to a function around $x = a$:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + O((x-a)^3).$$

We can approximate the derivative using a forward difference:

$$f'(a) \approx \frac{f(b) - f(a)}{h} + O(h),$$

where $h = b - a$. Now the integral of $f(x)$ from a to b becomes

$$\int\limits_a^b f(x)\,dx = hf(a) + \int_a^b (x-a)f'(a)\,dx + \int_a^b \frac{(x-a)^2}{2} f''(a)\,dx + O(h^4).$$

The integral

$$\int_a^b (x-a)f'(a) = \frac{(b-a)^2}{2} f'(a) = \frac{h^2}{2}\left(\frac{f(b)-f(a)}{h} + O(h)\right) = \frac{h}{2}\left(f(b) - f(a)\right) + O(h^3).$$

Additionally,

$$\int_a^b \frac{(x-a)^2}{2} f''(a)\,dx = -\frac{h^3}{6} f''(a) = O(h^3).$$

When we plug this into the original integral we get

$$\int\limits_a^b f(x)\,dx = \frac{h}{2}(f(a) + f(b)) + O(h^3).$$

This says that error in one piece of the trapezoid rule is third-order accurate, which means the error can be written as $Ch^3 + O(h^4)$. However, when we break the interval into N pieces, each of size $h = (b-a)/N$, the error terms add and each piece has its own constant so that

$$\sum_{i=1}^{N} C_i h^3 \leq N h^3 C_{\max} = (b-a) C_{\max} h^2,$$

where $C_{\max}$ is the maximum value of $|C_i|$. Therefore, the error in the sum of trapezoid rules decreases as h^2, which we observed above. This analysis can be extended to show that the error terms in the trapezoid rule only have even powers of h:

$$\text{Error} = C_2 h^2 + C_4 h^4 + \ldots$$

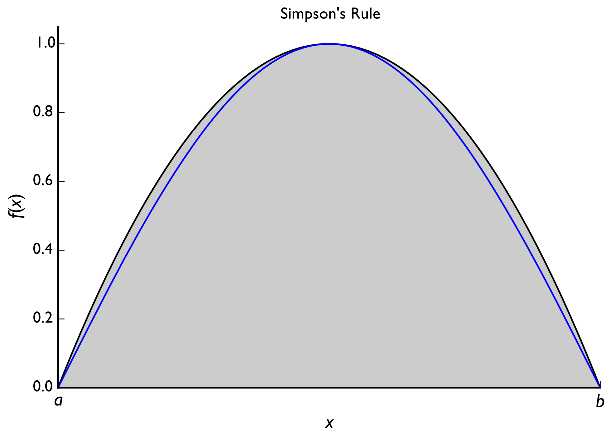
We will use this later when we combine the trapezoid rule with Richardson extrapolation.

15.2 SIMPSON'S RULE

Simpson's rule is like the trapezoid rule, except instead of fitting a line we fit a parabola between three points, a, b, and $(a+b)/2$. The formula for this is

$$I_{\text{Simpson}} \equiv \frac{h}{6}\left(f(a) + 4f\left(a + \frac{h}{2}\right) + f(b)\right) \approx \int_a^b f(x)\,dx,$$

where $h = b - a$. (This is sometimes called Simpsons 1/3 rule, because there is another Simpson rule that is based on quartic interpolation.) First, let's examine how this rule behaves on the integral of $\sin x$ with one piece:



It looks like the function can be well approximated by a parabola.

BOX 15.4 NUMERICAL PRINCIPLE

Simpson's rule approximates an integral by performing quadratic interpolation between the endpoints and midpoint of the interval of integration. The formula for this method is

$$\int_a^b f(x)\,dx = \frac{h}{6}\left(f(a) + 4f(c) + f(b)\right),$$

where $h = b - a$ and $c = (a+b)/2$.

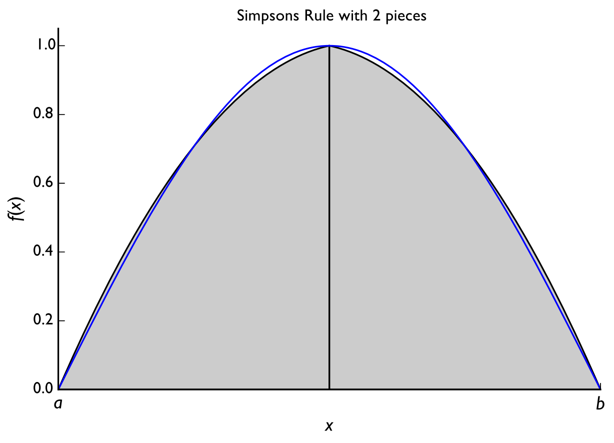
Here is a function to perform Simpson's rule just like we did for the trapezoid rule.

```
In [4]: def simpsons(f, a, b, pieces):
            """Find the integral of the function f between a and b
            using Simpson's rule
            Args:
                f: function to integrate
                a: lower bound of integral
```

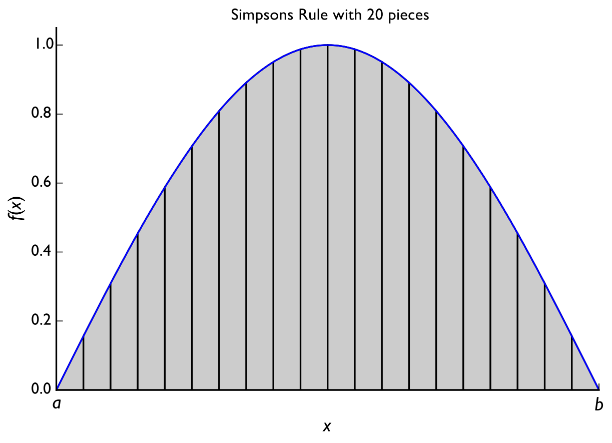
```
        b: upper bound of integral
        pieces: number of pieces to chop [a,b] into

    Returns:
        estimate of integral
    """
    integral = 0
    h = b - a
    one_sixth = 1.0/6.0
    #initialize the left function evaluation
    fa = f(a)
    for i in range(pieces):
        #evaluate the function at the left end of the piece
        fb = f(a+(i+1)*h/pieces)
        fmid = f(0.5*(a+(i+1)*h/pieces+ a+i*h/pieces))
        integral += one_sixth*h/pieces*(fa + 4*fmid + fb)
        #now make the left function evaluation the right for the next step
        fa = fb

    return integral
```

We then use this function to estimate the integral of the sine function using two and twenty pieces:

```
In [5]: integral_estimate = simpsons(np.sin,0,np.pi,pieces=2,graph=True)
        print("Estimate is",integral_estimate,"Actual value is 2")

        integral_estimate = simpsons(np.sin,0,np.pi,pieces=20,graph=True)
        print("Estimate is",integral_estimate,"Actual value is 2")
```

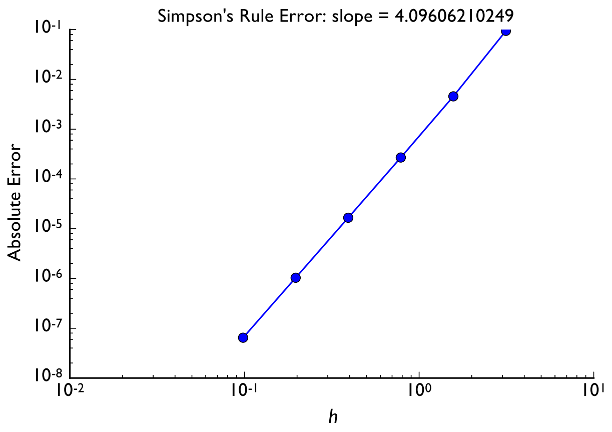


```
Estimate is 2.00455975498 Actual value is 2
```



```
Estimate is 2.00000042309 Actual value is 2
```

Just like the trapezoid rule, we can look at the error in Simpson's rule.

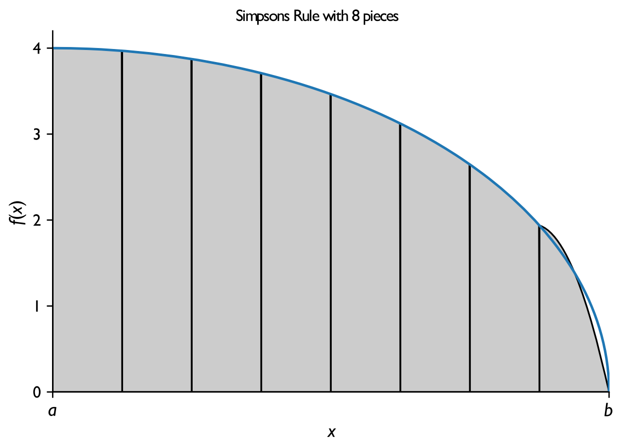


Simpson's rule is fourth-order in the piece size. This means that every time I double the number of pieces the error goes down by a factor of $2^4 = 16$.

Before moving on, we will use Simpson's rule to calculate π:

$$\int_0^1 4\sqrt{1-x^2}\,dx = \pi.$$

```
In [6]: integrand = lambda x: 4*np.sqrt(1-x**2)
        simpsons(integrand,0,1,pieces = 8,graph=True) #actual value is 3.14159
```



```
Out[6]: 3.1343976689845969
```

It looks like most of the error comes at $x = 1$. The reason for this is that function is changing rapidly near $x = 1$ because there is a singularity in the derivative:

$$\frac{d}{dx}4\sqrt{1-x^2} = -\frac{4x}{\sqrt{1-x^2}}.$$

Note that the denominator goes to 0 at $x = 1$. We will revisit this integral later.

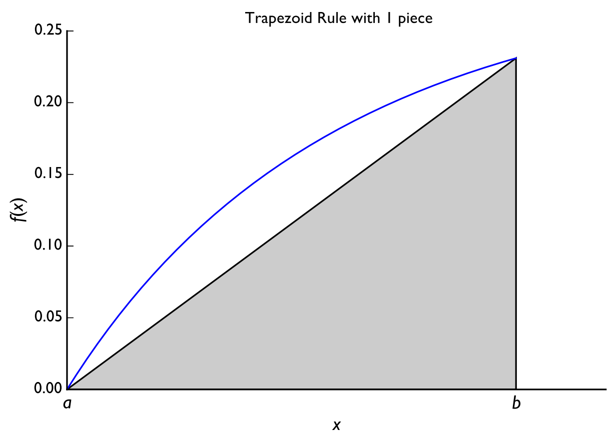
15.3 ROMBERG INTEGRATION

When we use trapezoid integration, we know that the error is second-order in the piece size. Using this information we can apply Richardson extrapolation. We can combine the approximation with one piece with that using two pieces to get a better approximation (one that is higher-order). Then, we can combine this approximation with the estimate using four pieces, to get an even better answer. To do this we need to use the fact that the trapezoid rule only has error terms that are even powers of h. To demonstrate this, we will compute the integral

$$\int_1^2 \frac{\ln x}{1+x}\,dx = 0.1472206769592413\ldots \tag{15.1}$$

The result from a one-piece trapezoidal integration is

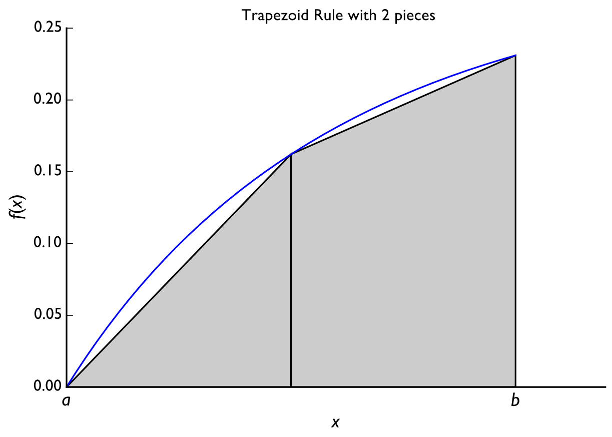
```
In [7]: integrand = lambda x: np.log(x)/(1.0+x)
        integral_estimate1 = trapezoid(integrand,1,2,pieces=1,graph=True)
        print("Estimate is",integral_estimate1,
              "Actual value is 0.1472206769592413, Error is",
              np.fabs(0.1472206769592413-integral_estimate1))
```



```
Estimate is 0.115524530093 Actual value is 0.1472206769592413,
Error is 0.0316961468659
```

Then we use two-pieces

```
In [8]: integral_estimate2 = trapezoid(integrand,1,2,pieces=2,graph=True)
        print("Estimate is",integral_estimate2,
              "Actual value is 0.1472206769592413, Error is",
              np.fabs(0.1472206769592413-integral_estimate2))
```



```
Estimate is 0.138855286668 Actual value is 0.1472206769592413,
Error is 0.00836539029095
```

We combine these estimates using Richardson Extrapolation. First, we need to define a new function for Richardson extrapolation. Our new implementation will use floating point numbers with higher precision than standard floating point numbers.

```
In [9]: import decimal
        #set precision to be 100 digits
```

```
decimal.getcontext().prec = 100
def RichExtrap(fh, fhn, n, k):
    """Compute the Richardson extrapolation based on
    two approximations of order k
    where the finite difference parameter h is used in fh and h/n in fhn.
    Inputs:
    fh:  Approximation using h
    fhn: Approximation using h/n
    n:   divisor of h
    k:   original order of approximation

    Returns:
    Richardson estimate of order k+1"""
    n = decimal.Decimal(n)
    k = decimal.Decimal(k)
    numerator = decimal.Decimal(n**k * decimal.Decimal(fhn)
                                - decimal.Decimal(fh))
    denominator = decimal.Decimal(n**k - decimal.Decimal(1.0))
    return float(numerator/denominator)
```

To make Richardson work well with high-order approximations we use arbitrary precision arithmetic using the `decimal` library.

BOX 15.5 PYTHON PRINCIPLE

The library `decimal` allows one to use higher precision floating point numbers than the standard floating point numbers in Python. It is necessary to set the desired precision with the command

```
decimal.getcontext().prec = Precision
```

where `Precision` is the integer number of digits of accuracy desired. Also, numbers that you want to be represented using this precision will need to be surrounded by the construct

```
decimal.Decimal(N)
```

where `N` is a number.

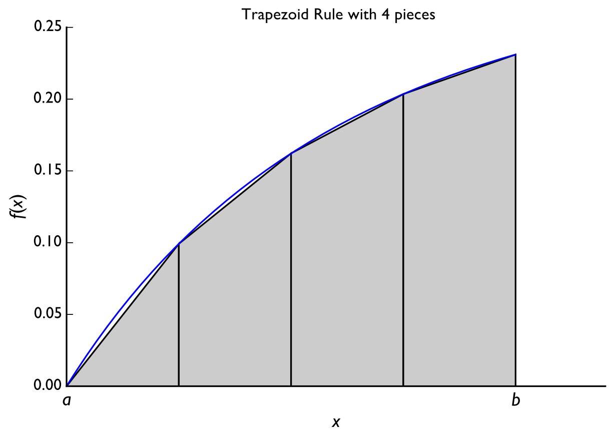
We will apply this function to the approximations with the trapezoid rule above.

```
In [10]: Richardson2 = RichExtrap(integral_estimate1,integral_estimate2,n=2,k=2)
         print("Estimate is",Richardson2,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-Richardson2))

Estimate is 0.1466322055266186 Actual value is 0.1472206769592413,
Error is 0.000588471432623
```

By applying Richardson extrapolation, we improved the estimate by an order of magnitude.

Now if we use 4 points, we get

```
In [15]: integral_estimate4 = trapezoid(integrand,1,2,pieces=4,graph=True)
         print("Estimate is",integral_estimate4,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-integral_estimate4))
```



```
Estimate is 0.145095533798 Actual value is 0.1472206769592413,
Error is 0.00212514316171
```

There are two Richardson extrapolations we can do at this point, one between the 4 and 2 piece estimates, and then one combines the two Richardson extrapolations:

```
In [11]: Richardson4 = RichExtrap(integral_estimate2,
                                          integral_estimate4, n=2,k=2)
         print("Estimate is",
         Richardson4,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-Richardson4))
```

```
Estimate is 0.14717561617394495 Actual value is 0.1472206769592413,
Error is 4.50607852963e-05
```

```
In [12]: Richardson42 = RichExtrap(Richardson2,Richardson4,
                                           n=2,k=4)
         #note this is fourth order
         print("Estimate is",Richardson42,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-Richardson42))
```

```
Estimate is 0.14721184355043337 Actual value is 0.1472206769592413,
Error is 8.83340880792e-06
```

Notice that the error from combining the two extrapolations is 3 orders of magnitude smaller than the error using 4 pieces. We could continue on by hand, but it is pretty easy to write a function for this. This procedure is called Romberg integration, and that is what we will name our function. The function will return a table of approximations where the first column is the original trapezoid rule approximations and the subsequent columns are a Richardson extrapolation of the column that came before.

```
In [13]: def Romberg(f, a, b, MaxLevels = 10, epsilon = 1.0e-6, PrintMatrix = False):
             """Compute the Romberg integral of f from a to b
```

```
    Inputs:
    f:  integrand function
    a: left edge of integral
    b: right edge of integral
    MaxLevels: Number of levels to take the integration to

    Returns:
    Romberg integral estimate"""

    estimate = np.zeros((MaxLevels,MaxLevels))

    estimate[0,0] = trapezoid(f,a,b,pieces=1)
    i = 1
    converged = 0
    while not(converged):
        estimate[i,0] = trapezoid(f,a,b,pieces=2**i)
        for extrap in range(i):
            estimate[i,1+extrap] = RichExtrap(estimate[i-1,extrap],
                                              estimate[i,extrap],
                                              2,2**(extrap+1))

        converged = np.fabs(estimate[i,i] - estimate[i-1,i-1]) < epsilon
        if (i == MaxLevels-1): converged = 1
        i += 1
    if (PrintMatrix):
        print(estimate[0:i,0:i])
    return estimate[i-1, i-1]
```

This function is defined to compute the integral estimate using a series of intervals and can print out the intermediate estimates and the extrapolated values. We will test this on the same integral as before.

```
In [14]: #this should give us what we got before
         integral_estimate = Romberg(integrand,1,2,MaxLevels=3, PrintMatrix=True)
         print("Estimate is",integral_estimate,
               "Actual value is 0.1472206769592413, Error is",
                     np.fabs(0.1472206769592413-integral_estimate))

[[ 0.11552453  0.          0.        ]
 [ 0.13885529  0.14663221  0.        ]
 [ 0.14509553  0.14717562  0.14721184]]
Estimate is 0.14721184355 Actual value is 0.1472206769592413,
Error is 8.83340880792e-06

In [15]: #Now let it converge, don't set Max Levels so low
         integral_estimate = Romberg(integrand,1,2,MaxLevels = 10, PrintMatrix=True,
                                     epsilon = 1.0e-10)
         print("Estimate is",integral_estimate,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-integral_estimate))

[[ 0.11552453  0.          0.          0.          0.         0.         0.]
 [ 0.13885529  0.14663221  0.          0.          0.         0.         0.]
 [ 0.14509553  0.14717562  0.14721184  0.          0.          0.          0.]
 [ 0.14668713  0.14721767  0.14722047  0.14722051  0.          0.          0.]
 [ 0.14708715  0.14722049  0.14722067  0.14722067  0.14722067 0.          0.]
 [ 0.14718729  0.14722066  0.14722068  0.14722068  0.14722068 0.14722068 0.]
 [ 0.14721233  0.14722068  0.14722068  0.14722068  0.14722068 0.14722068 0.14722068]]
Estimate is 0.147220676959 Actual value is 0.1472206769592413,
Error is 7.19035941898e-13
```

BOX 15.6 NUMERICAL PRINCIPLE

Romberg integration combines Richardson extrapolation with a known quadrature rule. It can produce integral estimates of high accuracy with a few function evaluations.

As one final example we will estimate π:

```
In [16]: integrand = lambda x: 4*np.sqrt(1-x**2)
         integral_estimate = Romberg(integrand,0,1,MaxLevels = 8,
                                     PrintMatrix=False, epsilon = 1.0e-10)
         print("Estimate is",integral_estimate,
               "Actual value is",np.pi,", Error is",
               np.fabs(np.pi-integral_estimate))

Estimate is 3.14131611425 Actual value is 3.141592653589793,
Error is 0.000276539339068
```

One thing to note is that our implementation of Romberg integration is not the most efficient. Technically, we are evaluating the function more times than we need to because when we call the trapezoid rule function with more pieces we are evaluating the function again at places we already did (for example, $f(a)$ and $f(b)$ are evaluated each time). However, making the most efficient algorithm would not make the most useful teaching example. For our purposes it suffices to know that this can be done in a smarter way if each function evaluation takes a long time.

There also is no reason we could not use the Romberg idea using Simpson's rule. Here is a function for that.

```
In [17]: def RombergSimps(f, a, b, MaxLevels = 10, epsilon = 1.0e-6,
                   PrintMatrix = False):
             """Compute the Romberg integral of f from a to b
             Inputs:
             f:  integrand function
             a: left edge of integral
             b: right edge of integral
             MaxLevels: Number of levels to take the integration to

             Returns:
             Romberg integral estimate"""

             estimate = np.zeros((MaxLevels,MaxLevels))
```

```
        estimate[0,0] = simpsons(f,a,b,pieces=1)
        i = 1
        converged = 0
        while not(converged):
            estimate[i,0] = simpsons(f,a,b,pieces=2**i)
            for extrap in range(i):
                estimate[i,1+extrap] = RichExtrap(estimate[i-1,extrap],
                                                  estimate[i,extrap],
                                                  n=2,k=2+2.0**(extrap+1))

            converged = np.fabs(estimate[i,i] - estimate[i-1,i-1]) < epsilon
            if (i == MaxLevels-1): converged = 1
            i += 1
        if (PrintMatrix):
            print(estimate[0:i,0:i])
        return estimate[i-1, i-1]
```

Using this function we can get a good estimate of the integral of the rational function from Eq. (15.1):

```
In [18]: #this should be better than what we got before,
         #Error was 8.83340880792e-06
         integrand = lambda x: np.log(x)/(1.0+x)
         integral_estimate = RombergSimps(integrand,1,2,MaxLevels=3,
                                          PrintMatrix=True)
         print("Estimate is",integral_estimate,
               "Actual value is 0.1472206769592413, Error is",
               np.fabs(0.1472206769592413-integral_estimate))
```

```
[[ 0.14663221  0.          0.        ]
 [ 0.14717562  0.14721184  0.        ]
 [ 0.14721767  0.14722047  0.14722061]]
Estimate is 0.147220608522 Actual value is 0.1472206769592413,
Error is 6.84372362669e-08
```

Applying this to estimate π using the default number of levels, we get

```
In [19]: integrand = lambda x: 4*np.sqrt(1-x**2)
         #trapezoid error was 0.00221405375506
         integral_estimate = RombergSimps(integrand,0,1,MaxLevels = 8,
                                          PrintMatrix=False, epsilon = 1.0e-10)
         print("Estimate is",integral_estimate,
               "Actual value is",np.pi,", Error is",
               np.fabs(np.pi-integral_estimate))
```

```
Estimate is 3.14149721605 Actual value is 3.141592653589793,
Error is 9.5437540061e-05
```

To get 10 digits of accuracy we need 20 levels or $2^{20} = 1\,048\,576$ intervals:

```
In [20]: integrand = lambda x: 4*np.sqrt(1-x**2)
         integral_estimate = RombergSimps(integrand,0,1,MaxLevels = 20,
                                          PrintMatrix=False, epsilon = 1.0e-14)
         print("Estimate is",integral_estimate,
               "Actual value is",np.pi,", Error is",
               np.fabs(np.pi-integral_estimate))
```

```
Estimate is 3.14159265323 Actual value is 3.141592653589793,
Error is 3.63962637806e-10
```

CODA

Here we have learned the basics of numerical integration using Newton–Cotes formulas. More importantly, we have shown how to combine these rules with Richardson extrapolation to get accurate estimates. In the next chapter we will discuss other types of quadrature rules and how to estimate multi-dimensional integrals.

FURTHER READING

The `decimal` package has a variety of further applications and can be a powerful tool. It is covered in detail in the official Python documentation at docs.python.org.

PROBLEMS

Short Exercises

Using the trapezoid rule and Simpson's rule estimate the following integrals with the following number of intervals: $2, 4, 8, 16, \ldots 512$. Compare your answers with Romberg integration where the maximum number of levels set to 9.

15.1. $\int_0^{\pi/2} e^{\sin x}\, dx \approx 3.104379017855555098181$.

15.2. $\int_0^{2.405} J_0(x) dx \approx 1.470300035485$, where $J_0(x)$ is a Bessel function of the first kind given by

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\,\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}.$$

Programming Projects

1. Inverse Fourier Transform

Consider the neutron diffusion equation in slab geometry an infinite, homogeneous medium given by

$$-D\frac{d^2}{dx^2}\phi(x) + \Sigma_\mathrm{a}\phi(x) = \delta(x),$$

where $\delta(x)$ is the Dirac delta function. This source is equivalent to a planar source inside the slab at $x = 0$. One way to solve this problem is to use a Fourier transform. The Fourier transform of a function can be defined by

$$\mathcal{F}\{f(x)\} = \hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dx\, f(x)(\cos kx - i \sin kx).$$

The Fourier transform of the diffusion equation above is

$$(Dk^2 + \Sigma_{\mathrm{a}})\hat{\phi}(k) = \frac{1}{\sqrt{2\pi}}.$$

We can solve this equation for $\hat{\phi}(k)$, and then apply the inverse Fourier transform:

$$\mathcal{F}^{-1}\{\hat{f}(k)\} = f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dk\, \hat{f}(k)(\cos kx + i \sin kx).$$

This leads to the solution being defined by

$$\phi(x) = \int_{-\infty}^{\infty} \frac{\cos kx\, dk}{2\pi(Dk^2 + \Sigma_{\mathrm{a}})} + i \int_{-\infty}^{\infty} \frac{\sin kx\, dk}{2\pi(Dk^2 + \Sigma_{\mathrm{a}})}.$$

The imaginary integral is zero because $\phi(x)$ is real. You can see that this is so because the integrand of the imaginary part is odd and the integral is symmetric about 0.

Your task is to compute the value of $\phi(x)$ at various points using $D = \Sigma_{\mathrm{a}} = 1$. Because you cannot integrate to infinity you will be computing integrals of the form

$$\int_{-L}^{L} f(x)\, dx,$$

for large values of L.

15.1. Compute value of $\phi(x)$ at 256 points in $x \in [-3, 3]$ using Simpson's and the trapezoidal rule with several different numbers of intervals (pieces) in the integration *and* using different endpoints in the integration, L. Plot these estimates of $\phi(x)$.

15.2. Plot the error between your estimate of $\phi(1)$ and the true solution of $\frac{1}{2}e^{-1}$. Make one graph each for trapezoid and Simpson's rule where the x-axis is h and the y-axis is the absolute error. On each plot show a curve for the error decay for $L = 10, 1000, 10^5, 10^8$.

15.3. Give your best estimate, using numerical integration, for the absorption rate density of neutrons, $\Sigma_{\mathrm{a}}\phi(x)$, at $x = 2$.