Computational Physics (PHYS6350)

Lecture 6: Numerical Integration: Part 1

- Basic methods for numerical integration (rectangle, trapezoid, Simpson)
- Adaptive quadrature
- Improper integrals

February 2, 2023

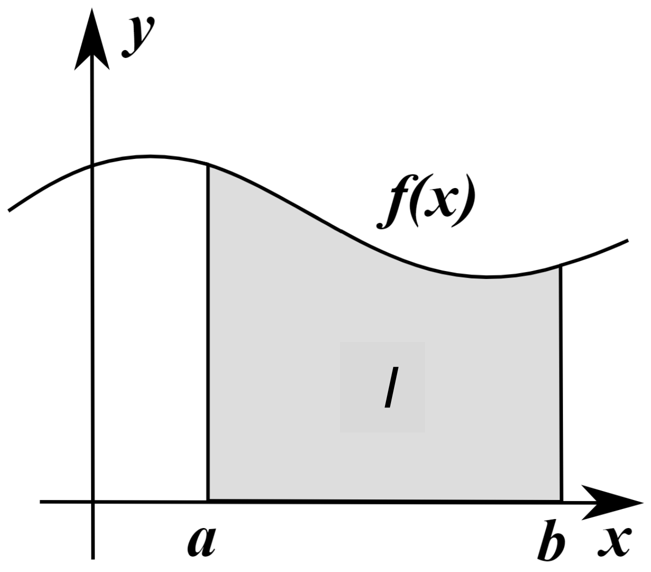
Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Numerical integration

Generic problem: evaluate

$$I = \int_a^b f(x)dx$$



We need numerical integration when

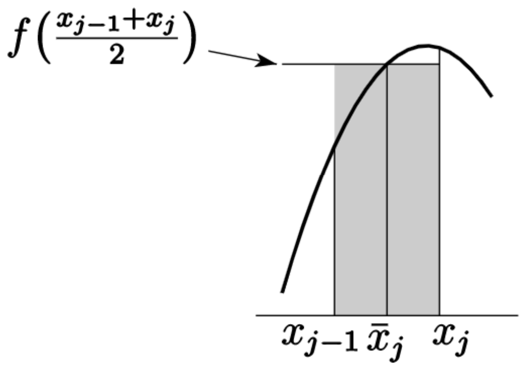
- Cannot/difficult integrate analytically
- Only know the integrand *f(x)* at certain points

References: Chapter 5 of *Computational Physics* by Mark Newman

Chapter 4 of *Numerical Recipes Third Edition* by W.H. Press et al.

Numerical integration: rectangular (midpoint) rule

Interpret the integral as the area under the curve and approximate by a rectangle evaluated at midpoint



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

Error (from Euler-McLaurin formula):

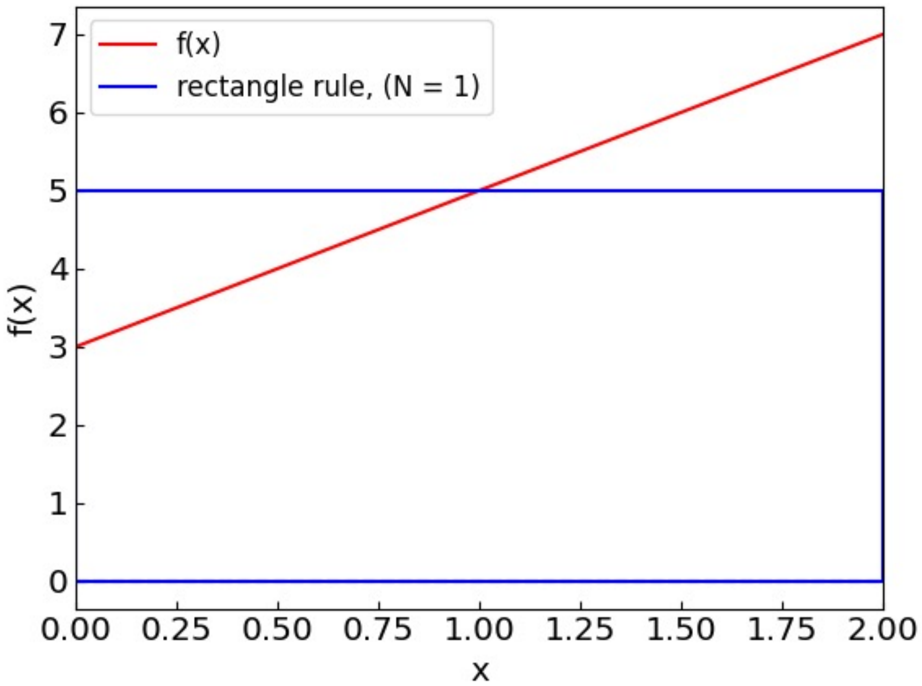
$$\int_a^b f(x)dx - (b-a)\, f\left(\frac{a+b}{2}\right) \approx \frac{(b-a)^3}{24} f''(a)$$

The rule is exact for the integration of linear functions

Numerical integration: rectangular (midpoint) rule

Example:

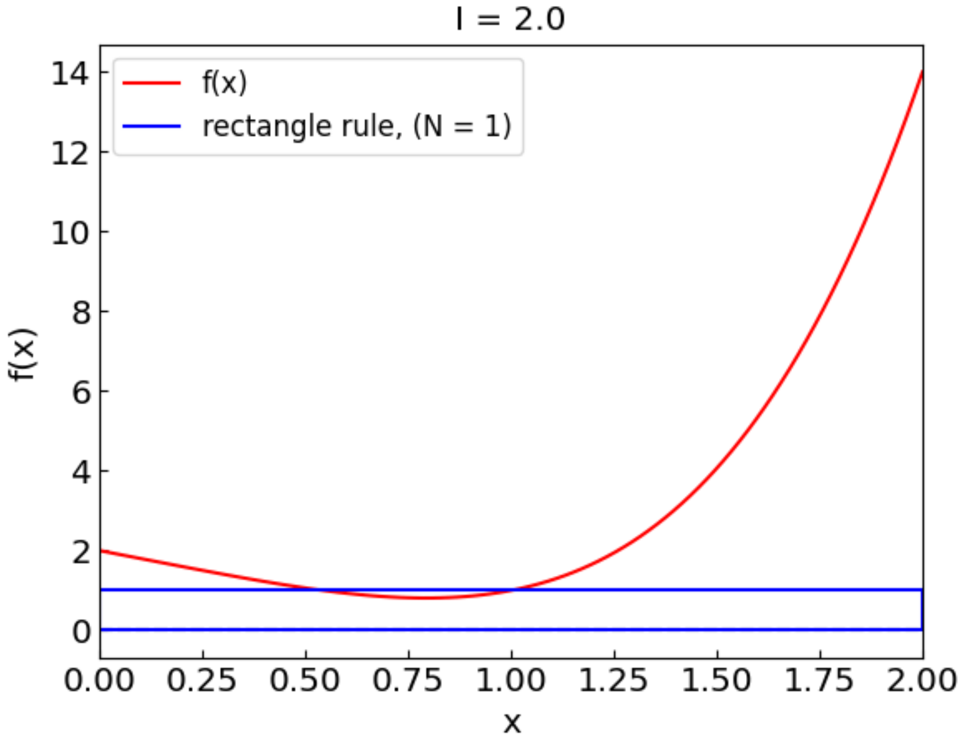
$$I = \int_0^2 2x + 3dx = 10$$



Although the rectangle is a poor approximate of the line (which is a trapezoid here), the errors cancel out

Numerical integration: rectangular (midpoint) rule

Another example: $$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Rectangle rule gives $I_{rect} = 2$ which is way off

Extended (composite) rectangular rule

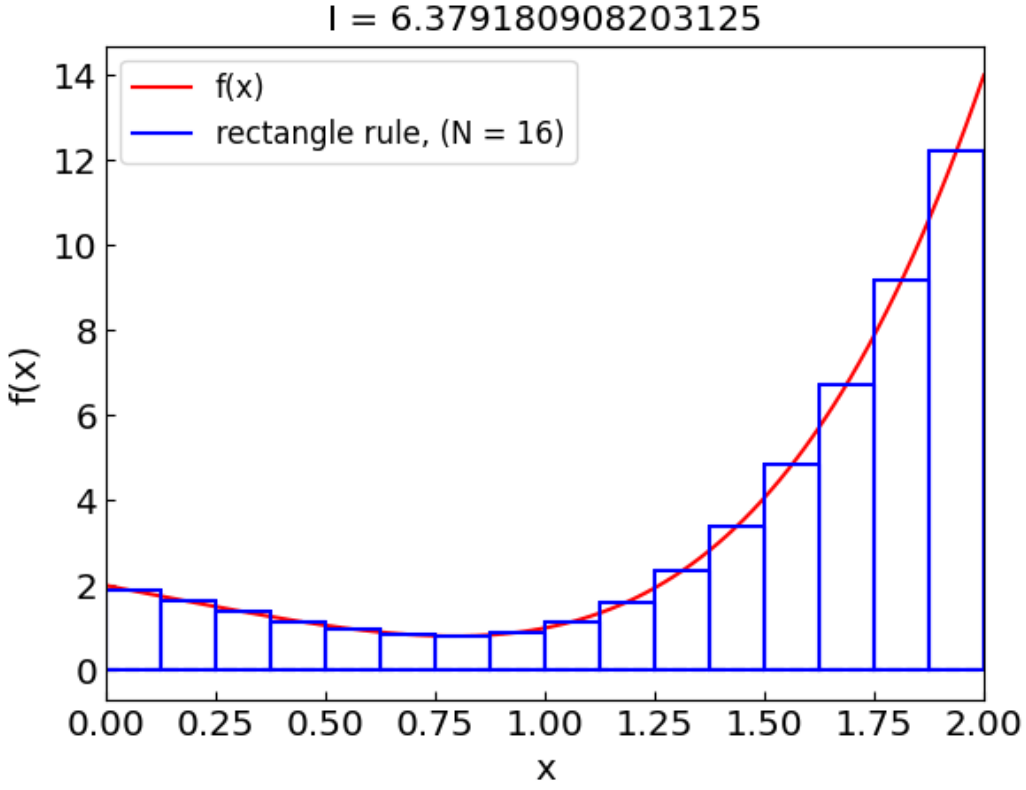
Split the integration interval into N sub-intervals
and apply the rectangle rule separately to each one

$$\int_0^2 x^4 - 2x + 2$$

$$\int_a^b f(x) \approx h \sum_{k=1}^{N} f(x_k), \qquad k = 1, \ldots, N$$

$$x_k = a + \frac{2k-1}{2} h \, .$$

$$h = (b-a)/N$$

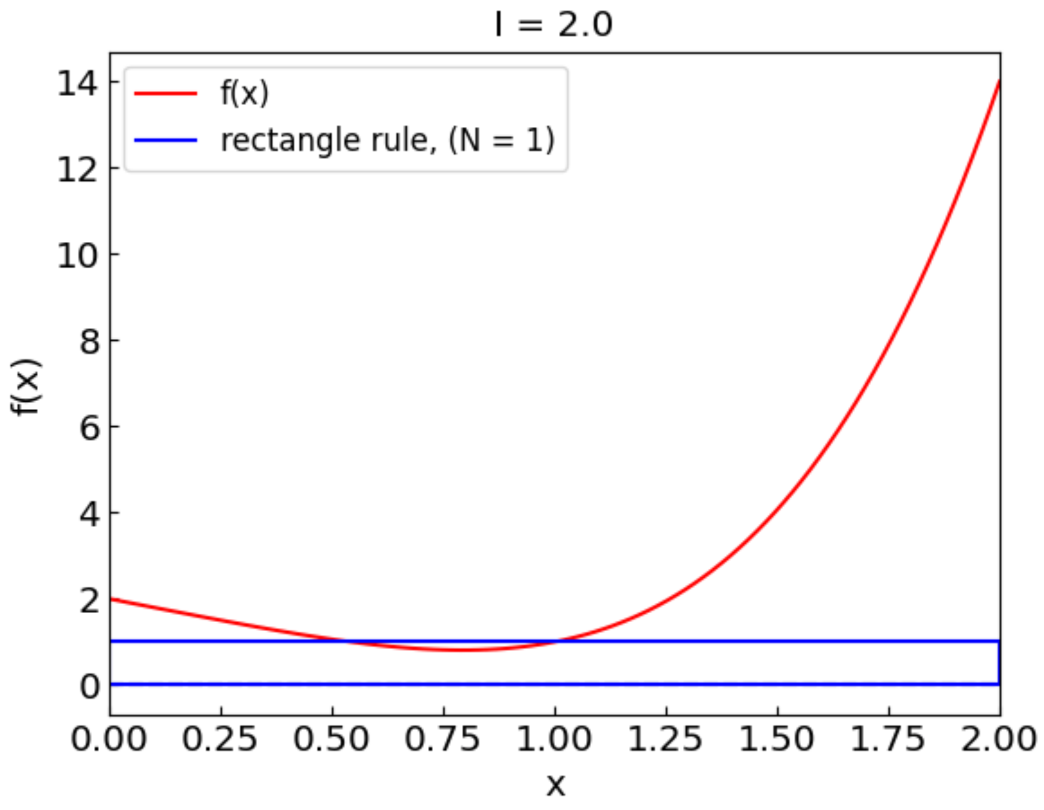


Error estimate:

$$I - I_{\text{rect}} = (b-a)\frac{h^2}{24} f''(a) + \mathcal{O}(h^4)$$

Extended (composite) rectangular rule

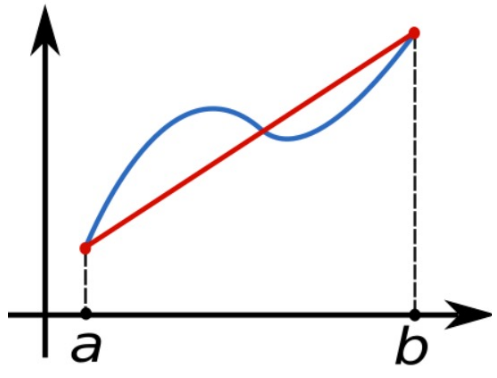
$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Numerical integration: trapezoidal rule

Approximate the integral by a trapezoid

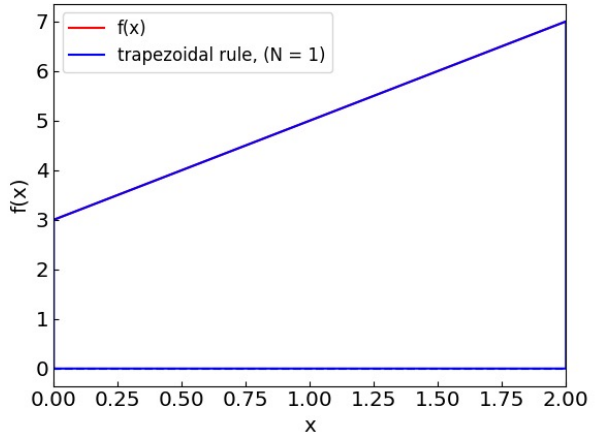
$$\int_a^b f(x)\,dx \approx (b-a)\,\frac{f(a)+f(b)}{2}$$



Error:

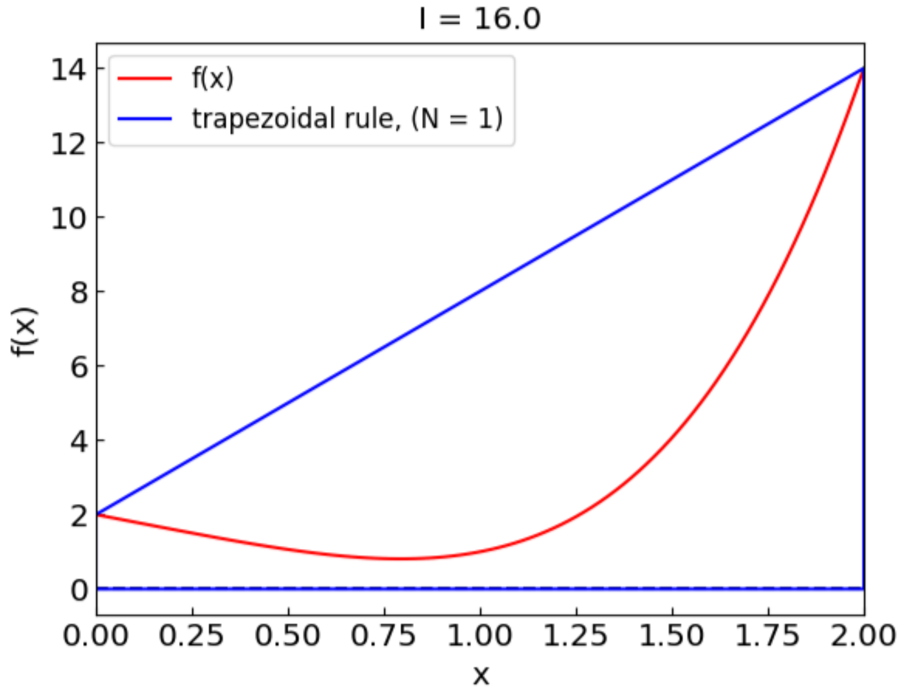
$$\int_a^b f(x)dx \; - \; (b-a)\,\frac{f(a)+f(b)}{2} \approx -\frac{(b-a)^3}{12} f''(a)$$

The rule is exact for the integration of linear functions



Numerical integration: trapezoidal rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Trapezoidal rule gives $\mathrm{I}_{\mathrm{trap}} = 16$, way off and in the opposite direction relative to rectangle rule

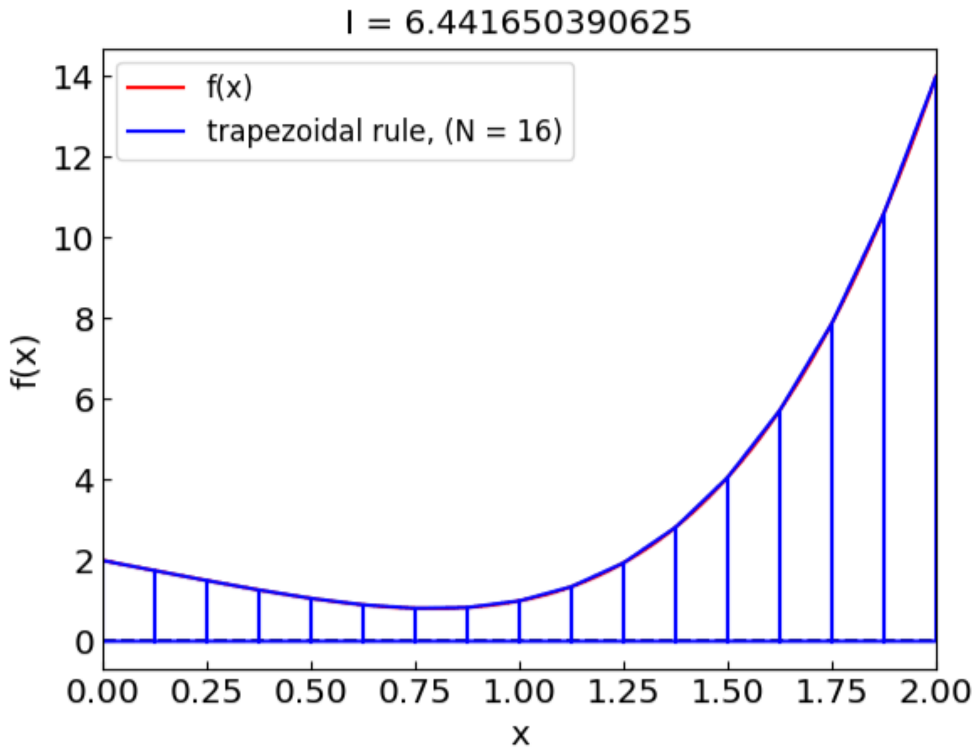
Extended trapezoidal rule

$$\int_0^2 x^4 - 2x + 2$$

$$\int_a^b f(x) \approx h \sum_{k=0}^{N} \frac{f(x_k) + f(x_{k+1})}{2}, \qquad i = 0, \ldots, N$$

$$x_k = a + kh \,.$$

$$h = (b - a)/N$$

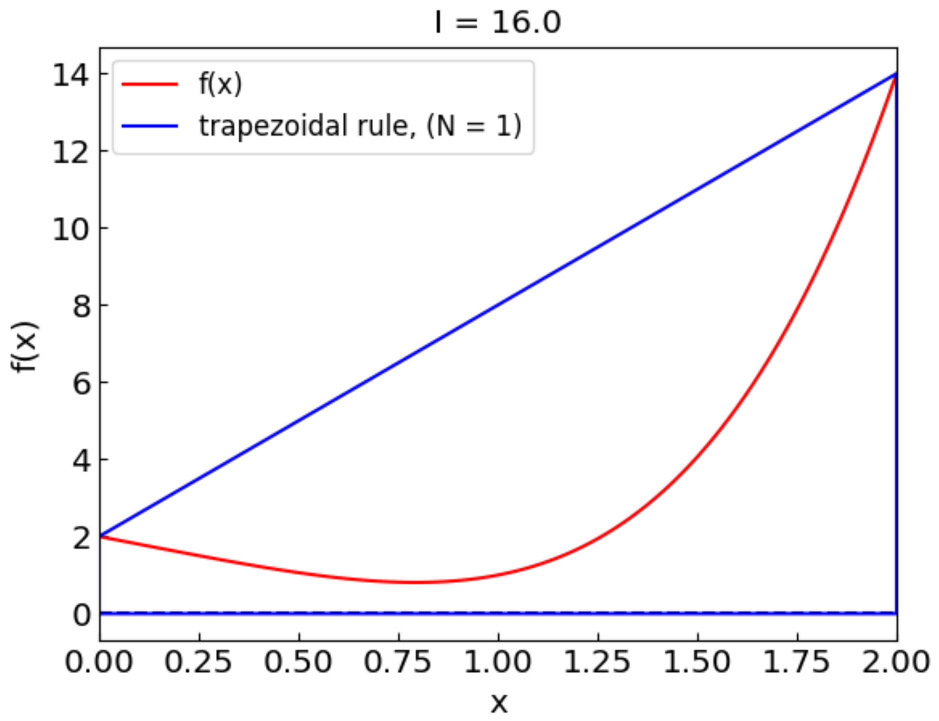


Error estimate:

$$I - I_{\text{trap}} = -(b - a)\frac{h^2}{12} f''(a) + \mathcal{O}(h^4)$$

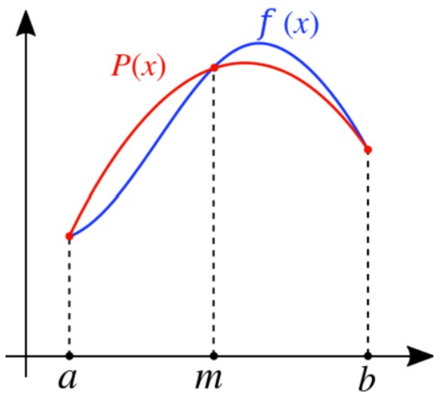
Extended (composite) trapezoidal rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



Numerical integration: Simpson's rule

Recall the error estimates for rectangular and trapezoidal rules



$$I - I_{\text{rect}} = (b-a)\frac{h^2}{24} f''(a) + \mathcal{O}(h^4) \qquad I - I_{\text{trap}} = -(b-a)\frac{h^2}{12} f''(a) + \mathcal{O}(h^4)$$

Combine them to eliminate the O(h^2) error term:

$$I_S = \frac{2I_{\text{rect}} + I_{\text{trap}}}{3}$$

i.e.

$$\int_a^b f(x)\,dx \approx \frac{(b-a)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

An equivalent way to obtain the rule: replace the integrand by the parabolic interpolation

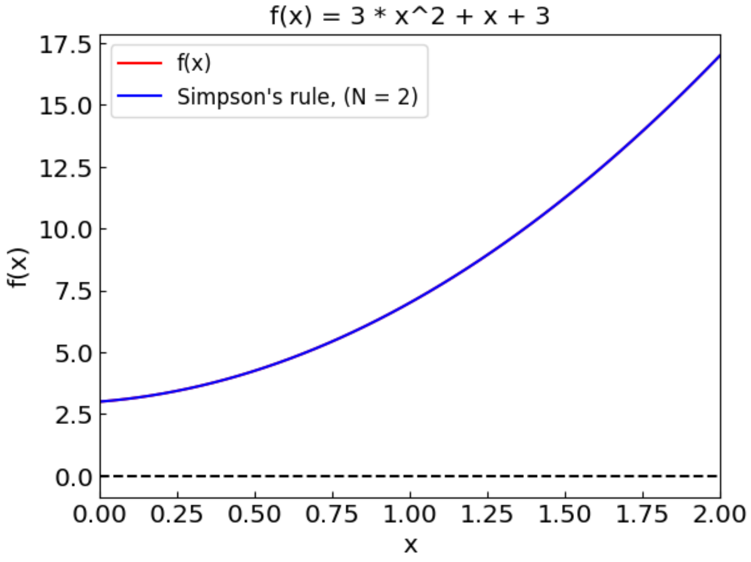
Numerical integration: Simpson's rule

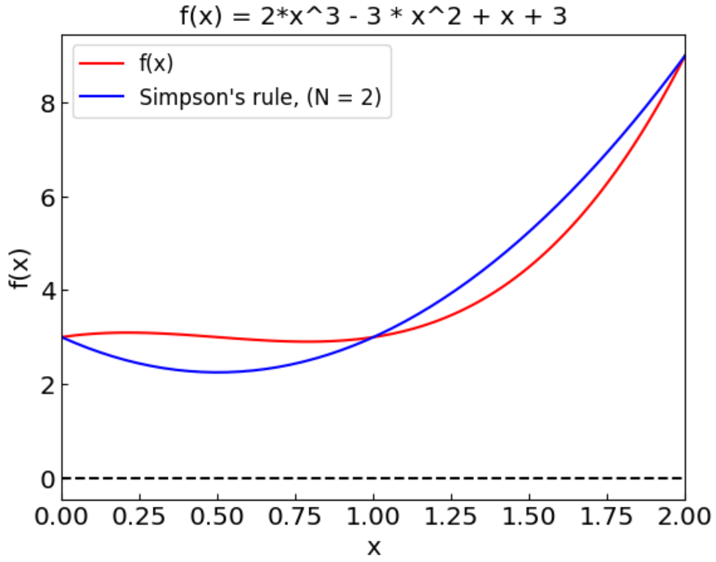
$$\int_a^b f(x)\,dx \approx \frac{(b-a)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

The error for the Simpson's rule is

$$I - I_S = C\,h^4 + \mathcal{O}(h^6)$$

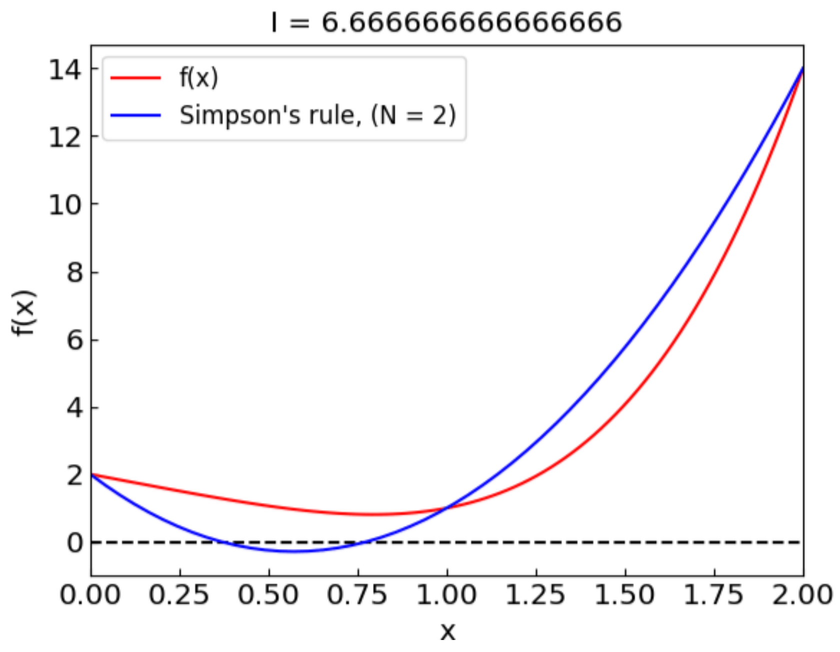
The method is exact for polynomials up to third order





Numerical integration: Simpson's rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$



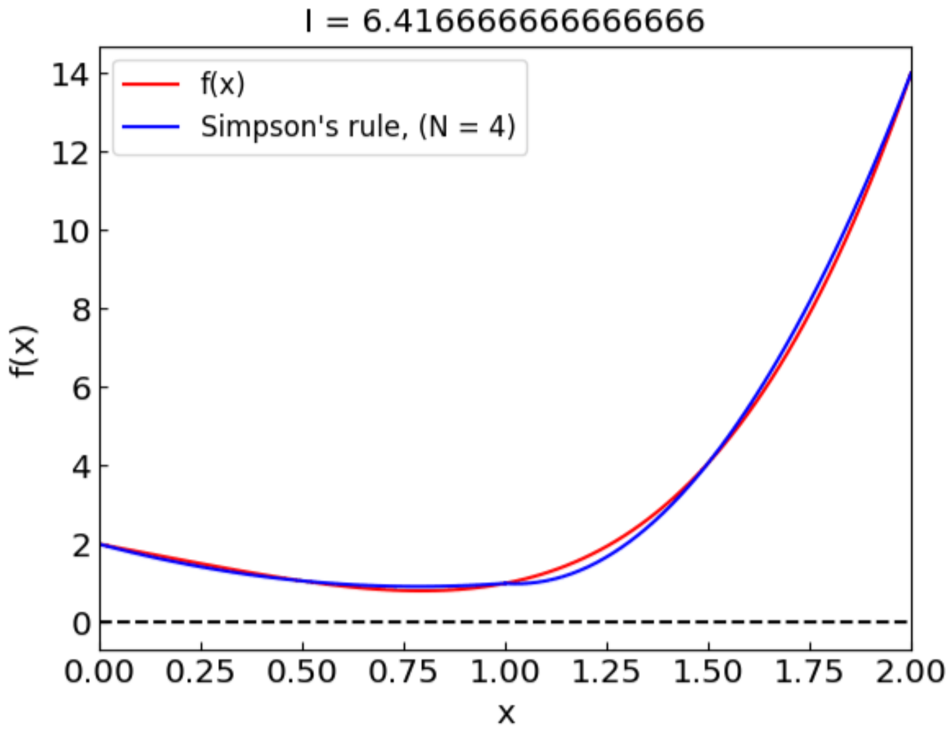
Simpson's rule gives $\mathrm{I}_{trap} = 6.66$ using three points, which is already not too bad!

Extended Simpson's rule

$$\int_a^b f(x) \approx \frac{h}{3}\left[f(x_0) + 4\sum_{k=1}^{N/2} f(x_{2k-1}) + 2\sum_{k=1}^{N/2-1} f(x_{2k}) + f(x_N)\right], \qquad i = 0, \ldots, N \qquad \int_0^2 x^4 - 2x + 2$$

$$h = (b-a)/N$$

N must be even!

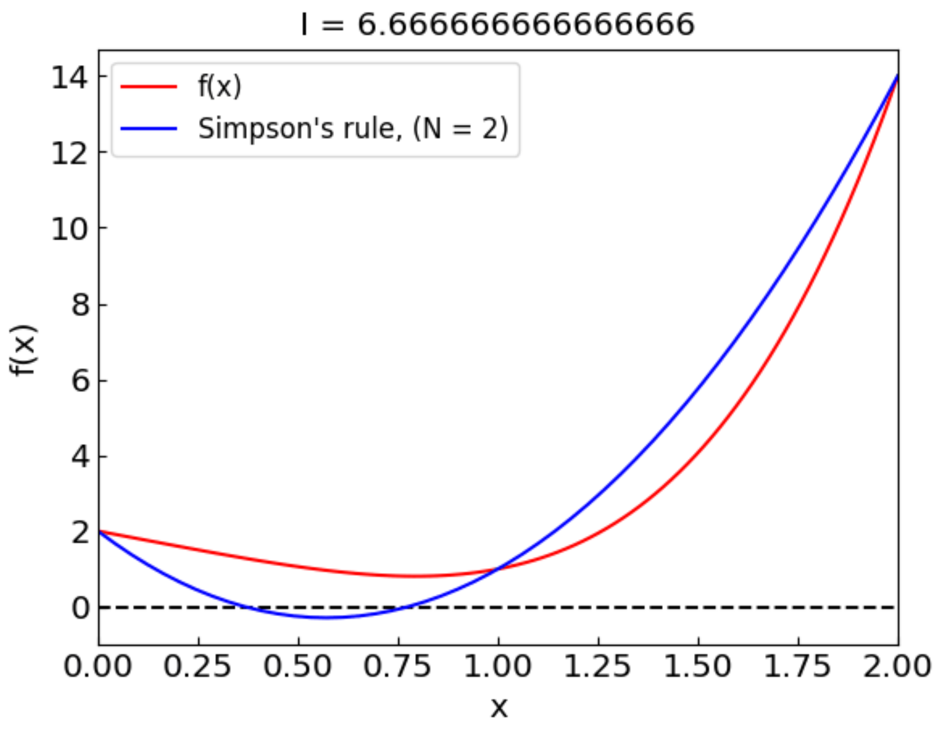


Error estimate:

$$I - I_S = C\,h^4 + \mathcal{O}(h^6)$$

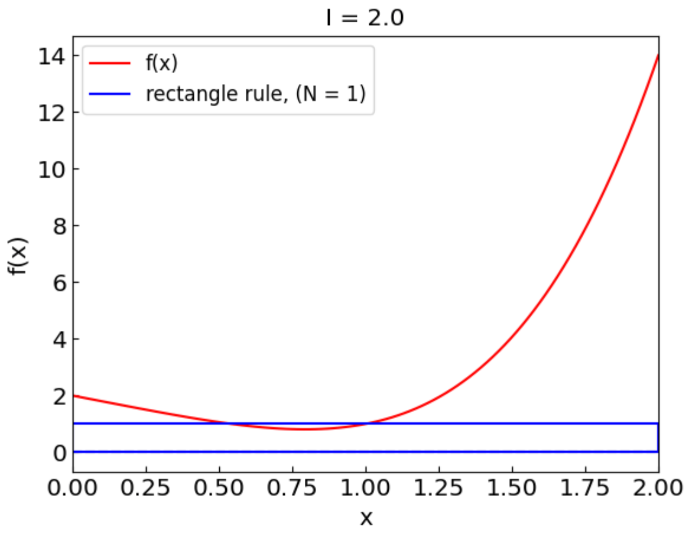
Extended Simpson's rule

$$I = \int_0^2 x^4 - 2x + 2 = 6.4$$

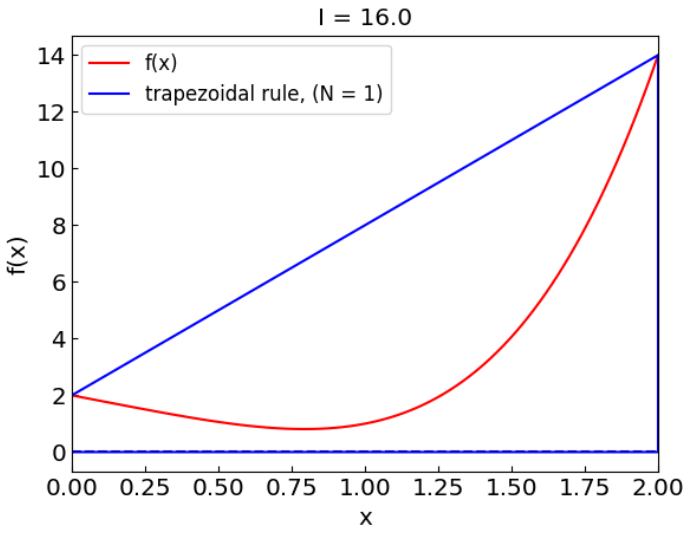


Comparing the methods

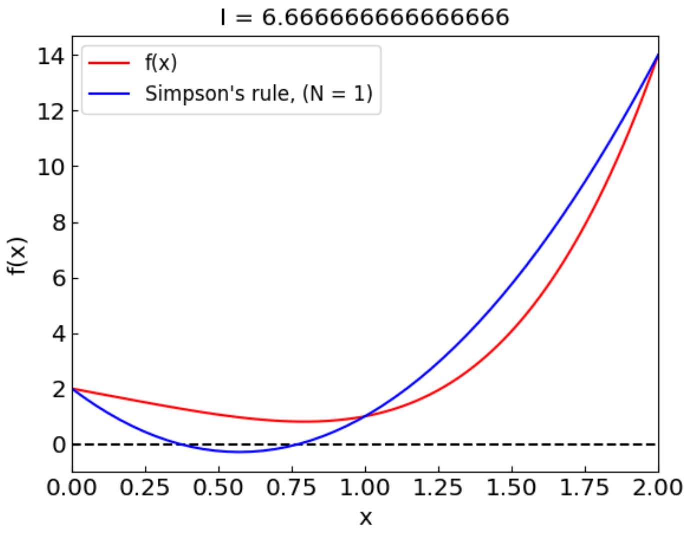
Rectangle



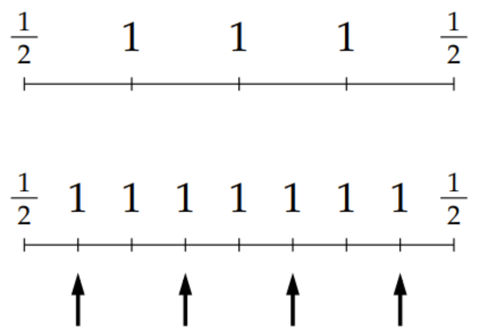
Trapezoid



Simpson



Adaptive quadrature



We would like to control the error in our calculation

This can be achieved by doubling the number of subintervals and keeping track of the error estimate

Recall that in the rectangle/trapezoidal rule the error is proportional to h^2

$$I - I_{\text{trap}} \approx ch^2$$

At step k we have $h_k = h_{k-1}/2$ therefore and the error at step k is estimated as

$$I - I_{\text{trap}}^{k} \approx ch_k^2, \qquad I - I_{\text{trap}}^{k-1} \approx 4ch_k^2,$$

$$\varepsilon_k \simeq (I_{\text{trap}}^{k} - I_{\text{trap}}^{k-1})/3$$

Adaptive trapezoidal rule

```
Trapezoidal rule for numerical integration with adaptive step
:f trapezoidal_rule_adaptive(f, a, b, nst = 1, tol = 1.e-8, max_iterations = 16):
  Iprev = 0.
  n = nst
  Iprev = trapezoidal_rule(f, a, b, n)
  print("Iteration: {0:5}, I = {1:20.15f}".format(1, Iprev))
  for k in range(1, max_iterations):
      n *= 2
      Inew = trapezoidal_rule(f, a, b, n)
      ek = (Inew - Iprev) / 3.
      print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(k+1, Inew, ek))
      if (abs(ek) < tol):
          return Inew
      Iprev = Inew

  print("Failed to achi
  return Inew
```

```
Computing the integral of x^4 - 2x + 2 over the interval ( 0.0 , 2.0 ) using adaptive trapezoidal rule
Iteration:     1, I =   16.000000000000000
Iteration:     2, I =    9.000000000000000, error estimate = -2.333333333333333
Iteration:     3, I =    7.062500000000000, error estimate = -0.645833333333333
Iteration:     4, I =    6.566406250000000, error estimate = -0.165364583333333
Iteration:     5, I =    6.441650390625000, error estimate = -0.041585286458333
Iteration:     6, I =    6.410415649414062, error estimate = -0.010411580403646
Iteration:     7, I =    6.402604103088379, error estimate = -0.002603848775228
Iteration:     8, I =    6.400651037693024, error estimate = -0.000651021798452
Iteration:     9, I =    6.400162760168314, error estimate = -0.000162759174903
Iteration:    10, I =    6.400040690088645, error estimate = -0.000040690026556
Iteration:    11, I =    6.400010172525072, error estimate = -0.000010172521191
Iteration:    12, I =    6.400002543131352, error estimate = -0.000002543131240
Iteration:    13, I =    6.400000635782950, error estimate = -0.000000635782801
Iteration:    14, I =    6.400000158945742, error estimate = -0.000000158945736
Iteration:    15, I =    6.400000039736406, error estimate = -0.000000039736446
Iteration:    16, I =    6.400000009934106, error estimate = -0.000000009934100
```

Adaptive Simpson rule

For Simpson's rule $\quad \varepsilon_k \simeq (I_S^k - I_S^{k-1})/15 \qquad$ (understand why 15 and not 3?)

```
# Simpson's rule for numerical integration with adaptive step
def simpson_rule_adaptive(f, a, b, nst = 2, tol = 1.e-8, max_iterations = 16):
    Iprev = 0.
    n = nst
    Iprev = simpson_rule(f, a, b, n)
    print("Iteration: {0:5}, I = {1:20.15f}".format(1, Iprev))
    for k in range(1, max_iterations):
        n *= 2
        Inew = simpson_rule(f, a, b, n)
        ek = (Inew - Iprev) / 15.

        print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(k+1, Inew, ek))
        if (abs(ek) < tol):
            return Inew
        Iprev = Inew

    print("Failed to ac
    return Inew
```

```
Computing the integral of x^4 - 2x + 2 over the interval ( 0.0 , 2.0 ) using adaptive Simpson's rule
Iteration:     1, I =    6.666666666666666
Iteration:     2, I =    6.416666666666666, error estimate = -0.016666666666667
Iteration:     3, I =    6.401041666666666, error estimate = -0.001041666666667
Iteration:     4, I =    6.400065104166666, error estimate = -0.000065104166667
Iteration:     5, I =    6.400004069010416, error estimate = -0.000004069010417
Iteration:     6, I =    6.400000254313150, error estimate = -0.000000254313151
Iteration:     7, I =    6.400000015894571, error estimate = -0.000000015894572
Iteration:     8, I =    6.400000000993410, error estimate = -0.000000000993411
```

Adaptive quadratures: Romberg method

Recall that we obtained error estimate for trapezoidal method at step k

$$\varepsilon_k \simeq (I^k_{\text{trap}} - I^{k-1}_{\text{trap}})/3$$

On the other hand, by definition, $\varepsilon_k = I - I^k_{\text{trap}}$

Therefore, we can improve our estimate of the integral as

$$I = R_{k,1} = I^k_{\text{trap}} + \frac{I^k_{\text{trap}} - I^{k-1}_{\text{trap}}}{3} + \mathcal{O}(h^4)$$

Romberg method: continue this procedure iteratively

$$R_{k,m+1} = R_{k,m} + \frac{R_{k,m} - R_{k-1,m}}{4^m - 1} .$$

until the desired accuracy is reached

Romberg method

```python
def romberg(
    f,
    a,
    b,
    accuracy=1e-8,
    max_order=10
):
    R = np.zeros((max_order, max_order))
    h = (b - a) / 2.
    R[0, 0] = h * (f(a) + f(b)) # The initial trapezoidal rule
    for n in range(1, max_order):
        trapezoid = 0.0
        for j in range(2**(n-1)):
            trapezoid += f(a + (2*j+1)*h)
        R[n, 0] = 0.5 * R[n-1, 0] + h * trapezoid # The trapezoidal rule
        l = 1
        # The Romberg iterations
        for m in range(1, n+1):
            l *= 4
            R[n, m] = (l * R[n, m-1] - R[n-1, m-1]) / (l-1)
        print("Iteration: {0:5}, I = {1:20.15f}, error estimate = {2:10.15f}".format(n, R[n, m], abs(R[n, m] - R[n-1, m-1])))
        if abs(R[n, m] - R[n-1, m-1]) < accuracy:
            return R[n, m]
        h /= 2.
    print("Romberg method did not converge to required accuracy")
    return R[-1, -1]
```

```
Computing the integral of x^4 - 2x + 2 over the interval ( 0.0 , 2.0 ) using Romberg method
Iteration:     1, I =    6.666666666666667, error estimate = 9.333333333333332
Iteration:     2, I =    6.400000000000000, error estimate = 0.266666666666667
Iteration:     3, I =    6.400000000000000, error estimate = 0.000000000000000
```

Improper integrals

- Contain integrable singularities (typically at the endpoints)

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_0^1 = 2$$

- (Semi-)infinite integration range

$$\int_0^\infty e^{-x} dx = 1 \qquad\qquad \int_{-\infty}^\infty e^{-x^2} dx = \sqrt{\pi}$$

Improper integrals: Singularities at endpoints

- Even though if the singularities at integration endpoints are integrable, the trapezoidal, Simpson, etc. methods will fail because they evaluate the integrand at the endpoints

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_0^1 = 2$$

```
def fsing1(x):
    return 1./np.sqrt(x)

trapezoidal_rule(fsing1,0.,1.,10)
```

```
/tmp/ipykernel_31240/847063500.py:2: RuntimeWarning: divide by zero encountered in double_scalars
  return 1./np.sqrt(x)
```

- **Solution:** use method that does use the endpoints (e.g. **rectangle rule**)

Improper integrals: Singularities at endpoints

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_0^1 = 2$$

```python
def fsing1(x):
    return 1./np.sqrt(x)
```

```python
print('Using rectangle rule to evaluate \int_0^1 1/\sqrt{x} dx')
nst = 1
rectangle_rule_adaptive(fsing1,0.,1.,1,1.e-3,20)
```

```
Using rectangle rule to evaluate \int_0^1 1/\sqrt{x} dx
Iteration:     1, I =     1.414213562373095
Iteration:     2, I =     1.577350269189626, error estimate = 0.054378902272177
Iteration:     3, I =     1.698844079579673, error estimate = 0.040497936796682
Iteration:     4, I =     1.786461001734842, error estimate = 0.029205640718390
Iteration:     5, I =     1.848856684639738, error estimate = 0.020798560968299
Iteration:     6, I =     1.893088359706383, error estimate = 0.014743891688882
Iteration:     7, I =     1.924392755699513, error estimate = 0.010434798664376
Iteration:     8, I =     1.946535279970520, error estimate = 0.007380841423669
Iteration:     9, I =     1.962194152677056, error estimate = 0.005219624235512
Iteration:    10, I =     1.973267083679453, error estimate = 0.003690977000799
Iteration:    11, I =     1.981096937261288, error estimate = 0.002609951193945
Iteration:    12, I =     1.986633507070365, error estimate = 0.001845523269692
Iteration:    13, I =     1.990548459938304, error estimate = 0.001304984289313
Iteration:    14, I =     1.993316751362098, error estimate = 0.000922763807931
```

Improper integrals: (Semi-)infinite intervals

Solution: map to a finite interval [e.g. (0,1)] by a change of variables

- Semi-infinite:

$$\int_a^\infty f(x)dx \Rightarrow x = a + \frac{t}{1-t} \Rightarrow \int_a^\infty f(x)dx = \int_0^1 f\left(a + \frac{t}{1-t}\right)\frac{dt}{1-t^2} = \int_0^1 g(t)dt$$

- Infinite:

$$\int_{-\infty}^\infty f(x)dx \Rightarrow x = \frac{t}{1-t^2} \Rightarrow \int_{-\infty}^\infty f(x)dx = \int_{-1}^1 f\left(\frac{t}{1-t^2}\right)\frac{1+t^2}{(1-t^2)^2}dt = \int_{-1}^1 g(t)dt$$

Then apply a standard method (e.g. rectangle rule to avoid endpoint singularities) to g(t)

NB: Other options for the change of variable are possible

Improper integrals: Semi-infinite intervals

$$\int_0^\infty e^{-x} dx = 1$$

```python
def fexp(x):
    return np.exp(-x)

def g(t, f, a = 0.):
    return f(a + t / (1. - t)) / (1. - t)**2
```

```python
a = 0.
def frect(x):
    return g(x, fexp, a)

print('Using change of variable and the rectangle rule to evaluate \int_0^\infty \exp(-x) dx')
rectangle_rule_adaptive(frect,0.,1.,1,1.e-6,20)
```

```
Using change of variable and the rectangle rule to evaluate \int_0^\infty \exp(-x) dx
Iteration:     1, I =    1.471517764685769
Iteration:     2, I =    1.035213267452946, error estimate = -0.145434832410941
Iteration:     3, I =    0.984670579385046, error estimate = -0.016847562689300
Iteration:     4, I =    1.001784913275257, error estimate = 0.005704777963404
Iteration:     5, I =    1.000155714391028, error estimate = -0.000543066294743
Iteration:     6, I =    1.000040642390661, error estimate = -0.000038357333456
Iteration:     7, I =    1.000010172618432, error estimate = -0.000010156590743
Iteration:     8, I =    1.000002543136036, error estimate = -0.000002543160799
Iteration:     9, I =    1.000000635783161, error estimate = -0.000000635784292
```

Improper integrals: Infinite intervals

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi} = 1.772454\ldots$$

```
def fexp2(x):
    return np.exp(-x**2)

def g2(t, f):
    return f(t / (1. - t**2)) * (1.+t**2) / (1. - t**2)**2
```

```
def frect2(x):
    return g2(x, fexp2)

print('Using change of variable and the rectangle rule to evaluate \int_{-\infty}^\infty \exp(-x^2) dx')
rectangle_rule_adaptive(frect2,-1.,1.,1,1.e-6,20)

print('Expected value: \sqrt{\pi} =', np.sqrt(np.pi))
```

```
Using change of variable and the rectangle rule to evaluate \int_{-\infty}^\infty \exp(-x^2) dx
Iteration:     1, I =    2.000000000000000
Iteration:     2, I =    2.849690615244243, error estimate = 0.283230205081414
Iteration:     3, I =    1.557994553948652, error estimate = -0.430565353765197
Iteration:     4, I =    1.808005109208286, error estimate = 0.083336851753211
Iteration:     5, I =    1.770118560572371, error estimate = -0.012628849545305
Iteration:     6, I =    1.772492101507391, error estimate = 0.000791180311673
Iteration:     7, I =    1.772453880915058, error estimate = -0.000012740197444
Iteration:     8, I =    1.772453850905505, error estimate = -0.000000010003185
Expected value: \sqrt{\pi} = 1.7724538509055159
```