Lecture 7

Numerical Integration and Programming

Jung-Il Choi

School of Mathematics and Computing (Computational Science and Engineering)

YONSEI UNIVERSITY

Contents

Basic concept

- **Numerical integration**
  - Weight sum

$$I(f;a,b) = \int_a^b f(x)dx = \sum_{k=0}^{Q} \int_{x_k}^{x_{k+1}} f(x)dx \approx \sum_{k=0}^{Q} w_k f(x_k) = I_N(f),$$

$$\text{where } x_0 = a, x_N = b, \text{and } Q = N-1$$

$$||I(f) - I_N(f)||_\infty \to 0 \quad as \quad N \to \infty$$

➔ Numerical integration points $\{x_k\}_{k=1}^{Q}$

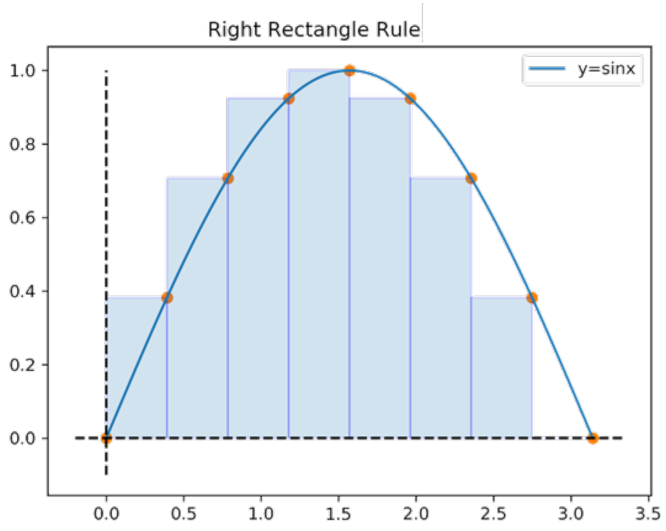
➔ Numerical integration weights $\{w_k\}_{k=1}^{Q}$

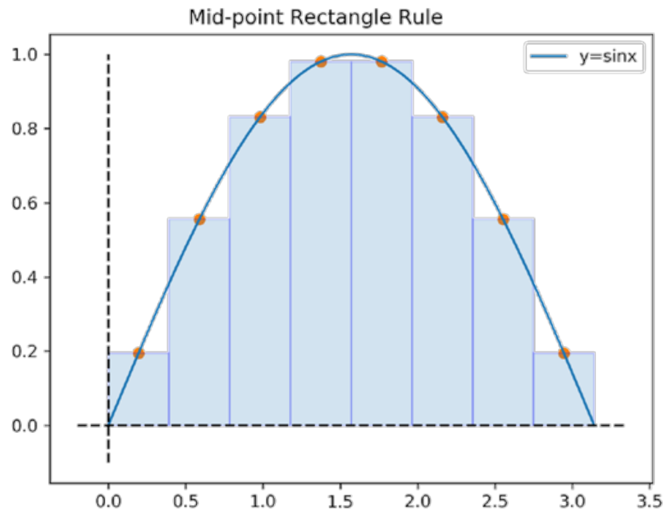
➔ **How do we define the weights for each data point?**

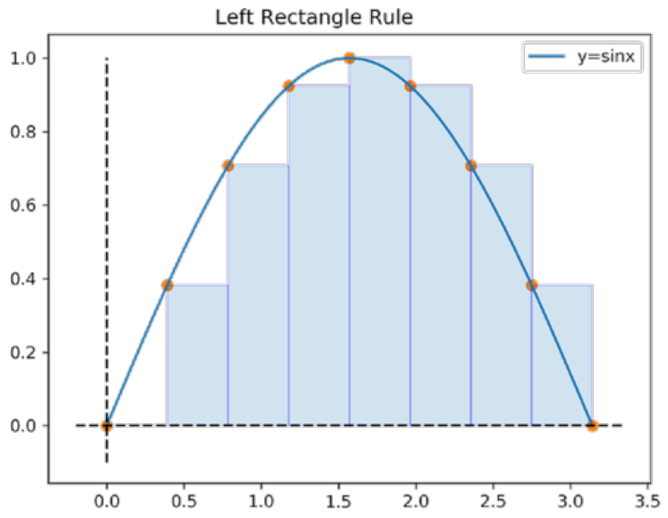
✓ **Geometry, Interpolation, Statistics**

Various numerical integrations and error analysis

- **Geometrical approach**
  - Rectangular rule (Riemann sum)







$$\int_{x_k}^{x_{k+1}} f(x)dx \approx h_k f(y_k), \qquad \text{where } h_k = x_{k+1} - x_k$$

$$I = \int_a^b f(x)dx \approx \sum_{k=0}^{N-1} h_k f(x_{k+1}) \text{ or } \sum_{k=0}^{N-1} h_k f\left(\frac{x_k + x_{k+1}}{2}\right) \text{ or } \sum_{k=0}^{N-1} h_k f(x_k)$$

For uniform spacing, h,

$$h = \frac{b-a}{N-1}, \qquad \text{for all } k$$

Various numerical integrations and error analysis

- **Geometrical approach**
  - Error analysis for the rectangular rule (Riemann sum) error using Taylor's expansion

➔ Consider the mid-point rule

$$\int_{x_k}^{x_{k+1}} f(x)dx \approx h_k f(y_k),$$

$$\text{where } y_k = (x_k + x_{k+1})/2, \qquad \text{and } h_k = x_{k+1} - x_k, \qquad \text{on } [x_k, x_{k+1}]$$

➔ Then,

$$f(x) = f(y_k) + (x - y_k)f'(y_k) + \frac{(x-y_k)^2}{2!}f''(y_k) + \frac{(x-y_k)^2}{3!}f'''(y_k) + \cdots$$

$$\int_{x_k}^{x_{k+1}} f(x)dx = h_k f(y_k) + \frac{1}{3}\frac{(x-y_k)^3}{2!}\Big|_{x_k}^{x_{k+1}} f''(y_k) + \frac{1}{4}\frac{(x-y_k)^3}{3!}\Big|_{x_k}^{x_{k+1}} f'''(y_k) + \cdots$$

$$= h_k f(y_k) + \frac{h_k^3}{24}f''(y_k) + \cdots = h_k f(y_k) + O\left(h_k^3\right)$$

➔ For one interval, the rectangular rule is third-order accurate

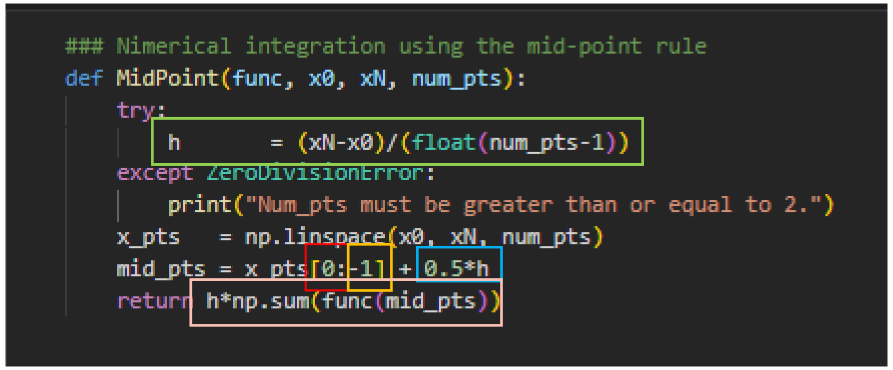
➔ For entire domain → Confirm the trapezoidal rule! (second-order accuracy)

Various numerical integrations and error analysis

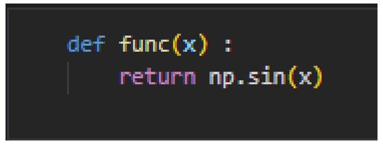
- **Geometrical approach**
  - Code for the rectangular mid-point rule (Uniform spacing)
    - ➔ Verifying the answer using, $\int_0^{\pi} \sin(x)\,dx = 2$

$$I_N(f) = \boxed{\sum_{k=\boxed{0}}^{\boxed{N-1}} \boxed{h_k} f\left(\boxed{\frac{x_k + x_{k+1}}{2}}\right)}$$

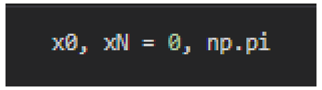
```
import numpy as np
✓ 0.0s
```



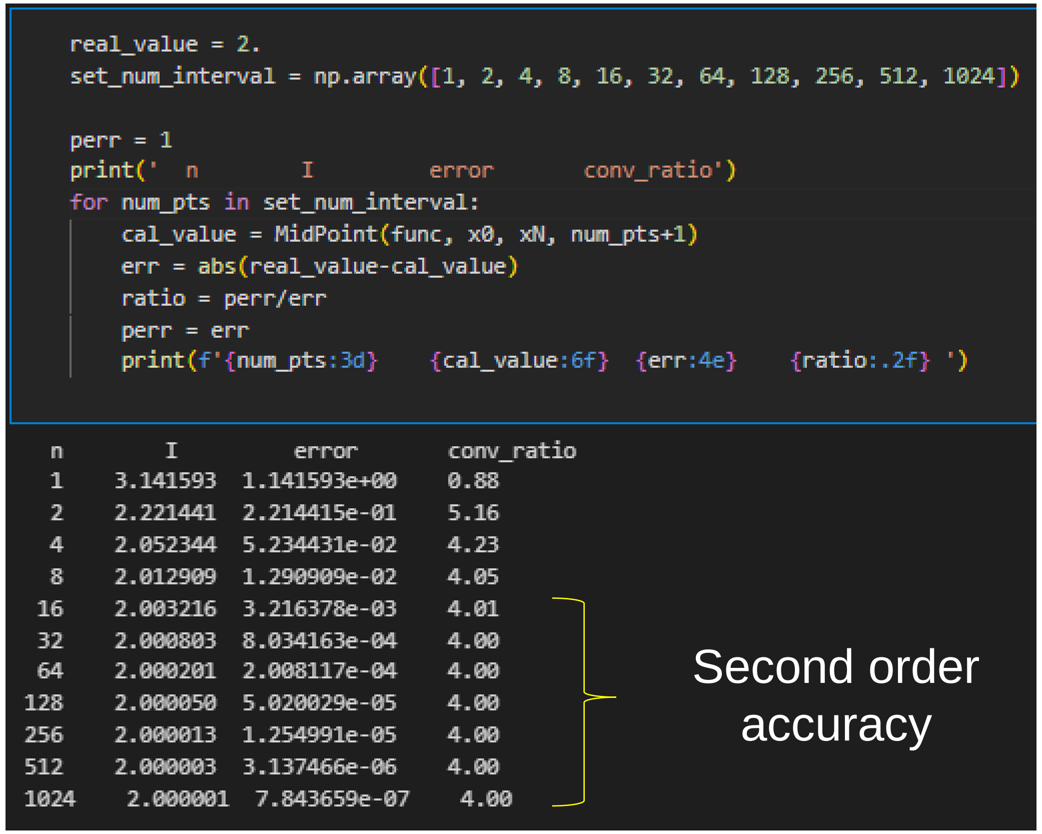
```
### Nimerical integration using the mid-point rule
def MidPoint(func, x0, xN, num_pts):
    try:
        h       = (xN-x0)/(float(num_pts-1))
    except ZeroDivisionError:
        print("Num_pts must be greater than or equal to 2.")
    x_pts   = np.linspace(x0, xN, num_pts)
    mid_pts = x_pts[0:-1] + 0.5*h
    return h*np.sum(func(mid_pts))
```



```
def func(x) :
    return np.sin(x)
```



```
x0, xN = 0, np.pi
```



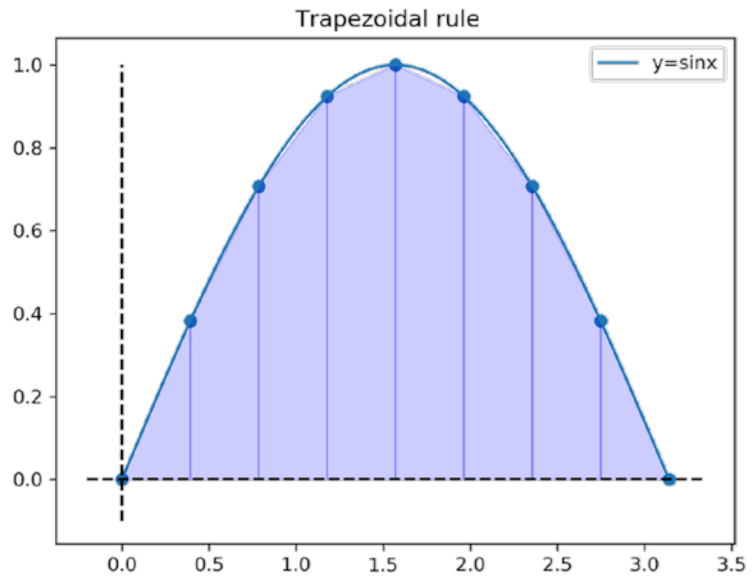
```
real_value = 2.
set_num_interval = np.array([1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])

perr = 1
print('  n        I         error       conv_ratio')
for num_pts in set_num_interval:
    cal_value = MidPoint(func, x0, xN, num_pts+1)
    err = abs(real_value-cal_value)
    ratio = perr/err
    perr = err
    print(f'{num_pts:3d}    {cal_value:6f}  {err:4e}    {ratio:.2f} ')
```

```
  n        I         error       conv_ratio
  1    3.141593  1.141593e+00    0.88
  2    2.221441  2.214415e-01    5.16
  4    2.052344  5.234431e-02    4.23
  8    2.012909  1.290909e-02    4.05
 16    2.003216  3.216378e-03    4.01
 32    2.000803  8.034163e-04    4.00
 64    2.000201  2.008117e-04    4.00
128    2.000050  5.020029e-05    4.00
256    2.000013  1.254991e-05    4.00
512    2.000003  3.137466e-06    4.00
1024     2.000001  7.843659e-07     4.00
```

Second order accuracy

Various numerical integrations and error analysis

- **Geometrical approach**
  - Trapezoidal rule



$$\int_{x_k}^{x_{k+1}} f(x)dx \approx \frac{h_k}{2}(f_k + f_{k+1}), \qquad \text{where } h_k = x_{k+1} - x_k$$

For uniform spacing, $h_k \to h$ the formular is,

$$I = \int_a^b f(x)dx \approx \sum_{k=0}^{N-1} \frac{b-a}{N-1}\left[\frac{f(x_{k+1}) + f(x_k)}{2}\right] = h\left[\frac{1}{2}f_0 + \frac{1}{2}f_N + \sum_{k=0}^{N-1} f_k\right]$$

Various numerical integrations and error analysis

- **Geometrical approach**
  - Error analysis for the trapezoidal rule using Taylor's expansion

$$f(x) = f(y_k) + (x - y_k)f'(y_k) + \frac{(x - y_k)^2}{2!}f''(y_k) + \frac{(x - y_k)^3}{3!}f'''(y_k) + \cdots$$

$$f(x_k) = f(y_k) - \frac{h_k}{2}f'(y_k) + \frac{1}{2!}\left(\frac{h_k}{2}\right)^2 f_k''(y_k) - \frac{1}{3!}\left(\frac{h_k}{2}\right)^3 f'''(y_k) + \cdots$$

$$f(x_{k+1}) = f(y_k) + \frac{h_k}{2}f'(y_k) + \frac{1}{2!}\left(\frac{h_k}{2}\right)^2 f_k''(y_k) + \frac{1}{3!}\left(\frac{h_k}{2}\right)^3 f'''(y_k) + \cdots$$

$$\frac{f(x_k) + f(x_{k+1})}{2} = f(y_k) + \frac{1}{2!}\left(\frac{h_k}{2}\right)^2 f_k''(y_k) + \cdots$$

$$\int_{x_k}^{x_{k+1}} f(x)dx = h_k\left[\frac{f(x_k) + f(x_{k+1})}{2}\right] - \frac{h_k^3}{12}[f''(y_k)] + \cdots = h_k\left[\frac{f(x_k) + f(x_{k+1})}{2}\right] + O\left(h_k^3\right)$$

➔ For one interval, the trapezoidal rule is also third-order accurate

Various numerical integrations and error analysis

- **Geometrical approach**
  - For entire domain,
    - ➔ let consider the uniform spacing, then,

$$I = \int_a^b f(x)dx = \sum_{k=0}^{Q} \int_{x_k}^{x_{k+1}} f(x)dx = \sum_{k=0}^{Q} \left\{ h_k \left[\frac{f(x_k) + f(x_{k+1})}{2} \right] + O\left(h_k^3\right) \right\}$$

$$= h_k \left[\frac{1}{2} f_0 + \frac{1}{2} f_N + \sum_{k=0}^{N-1} f_k \right] - \sum_{k=0}^{Q} \frac{h_k^3}{12} [f''(y_k)] \ + \cdots$$

$$= h_k \left[\frac{1}{2} f_0 + \frac{1}{2} f_N + \sum_{k=0}^{N-1} f_k \right] - \frac{h_k^2}{12} (b-a)[f''(\xi)] + \cdots = h_k \left[\frac{1}{2} f_0 + \frac{1}{2} f_N + \sum_{k=0}^{N-1} f_k \right] + O(h^2)$$

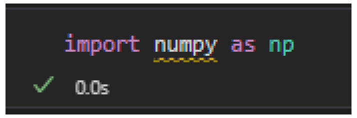
$$\text{for some } \xi \in [a, b], \text{and the uniform spacing, } h$$

- ➔ The trapezoidal rule for the entire interval is second-order accurate
- ➔ It is also same as the rectangular mid-point rule

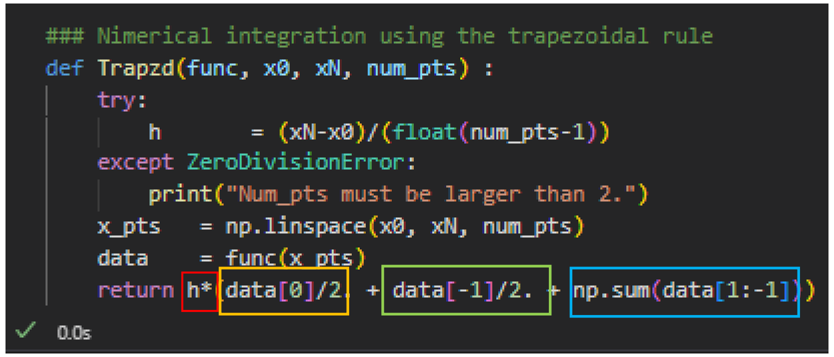
Various numerical integrations and error analysis

- **Geometrical approach**
  - Code for the trapezoidal rule,
    - ➔ Verifying the answer using, $\int_0^{\pi} \sin(x)\,dx = 2$

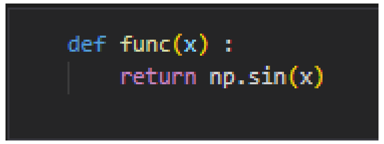
$$I_N(f) = \boxed{h}\left[\boxed{\frac{1}{2} f_0} + \boxed{\frac{1}{2} f_N} + \boxed{\sum_{k=0}^{N-1} f_k}\right]$$



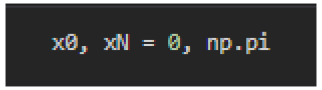
```
import numpy as np
✓ 0.0s
```



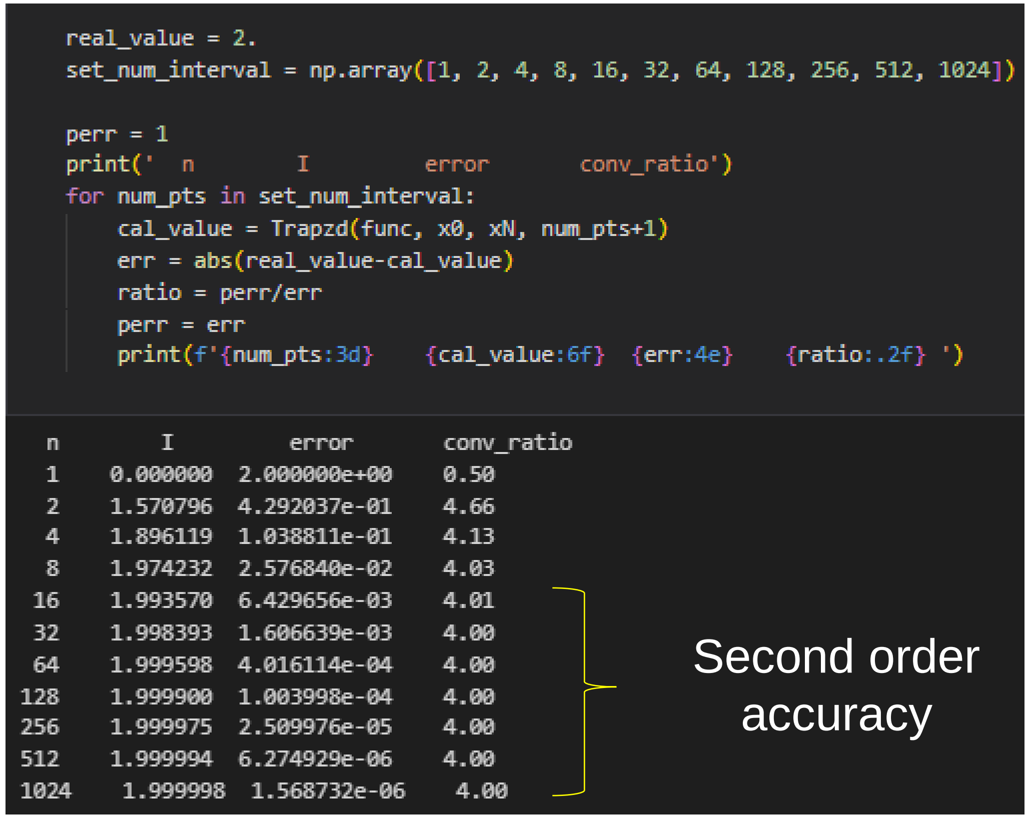
```
### Nimerical integration using the trapezoidal rule
def Trapzd(func, x0, xN, num_pts) :
    try:
        h       = (xN-x0)/(float(num_pts-1))
    except ZeroDivisionError:
        print("Num_pts must be larger than 2.")
    x_pts   = np.linspace(x0, xN, num_pts)
    data    = func(x_pts)
    return h*data[0]/2. + data[-1]/2. + np.sum(data[1:-1]))
✓ 0.0s
```



```
def func(x) :
    return np.sin(x)
```



```
x0, xN = 0, np.pi
```



```
real_value = 2.
set_num_interval = np.array([1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])

perr = 1
print('  n         I          error        conv_ratio')
for num_pts in set_num_interval:
    cal_value = Trapzd(func, x0, xN, num_pts+1)
    err = abs(real_value-cal_value)
    ratio = perr/err
    perr = err
    print(f'{num_pts:3d}    {cal_value:6f}  {err:4e}    {ratio:.2f} ')
```

```
  n       I          error        conv_ratio
  1    0.000000  2.000000e+00    0.50
  2    1.570796  4.292037e-01    4.66
  4    1.896119  1.038811e-01    4.13
  8    1.974232  2.576840e-02    4.03
 16    1.993570  6.429656e-03    4.01
 32    1.998393  1.606639e-03    4.00
 64    1.999598  4.016114e-04    4.00
128    1.999900  1.003998e-04    4.00
256    1.999975  2.509976e-05    4.00
512    1.999994  6.274929e-06    4.00
1024    1.999998  1.568732e-06    4.00
```

Second order accuracy

Various numerical integrations and error analysis

- **Geometrical approach**
  - Trapezoidal rule with end-correction

$$I_k = \int_{x_k}^{x_{k+1}} f(x)dx = h_k\left[\frac{f(x_k)+f(x_{k+1})}{2}\right] - \frac{h_k^3}{12}[f''(y_k)] + \cdots$$

$$f''(y_k) = \frac{f'(x_{k+1})-f'(x_k)}{h_k} + O(h_k^2)$$

$$I_k = h_k\left[\frac{f(x_k)+f(x_{k+1})}{2}\right] - \frac{h_k^3}{12}\left[\frac{f'(x_{k+1})-f'(x_k)}{h_k}\right] + O(h_k^5)$$

➔ For uniform spacing, h,

$$I_N = \sum_{k=0}^{Q} I_k = \frac{h}{2}\sum_{k=0}^{Q}(f(x_k)+f(x_{k+1})) - \frac{h^2}{12}\sum_{k=0}^{Q}(f'(x_{k+1})-f'(x_k)) + O(h^4)$$

➔ Cancellations in the second summation on the right-hand-side lead to

$$I = \frac{h}{2}\sum_{k=0}^{Q}\left(f(x_k)+f(x_{k+1})\right) - \frac{h^2}{12}\left(f'(b)-f'(a)\right) + O(h^4)$$

Various numerical integrations and error analysis

- **Interpolation approach**

$$I(f;a,b)=\int_a^b f(x)dx=\sum_{k=0}^{Q}\int_{x_k}^{x_{k+1}} f(x)dx \approx \sum_{k=0}^{Q} w_k f(x_k)=I_N(f)$$

- For whole interested domain with the number of degree, d

$$f(x)\approx p(x)=\sum_{j=0}^{d} f(x_j)l_j(x)$$

- Or, piece-wisely,

$$f(x)\approx g_i(x)=\sum_{j=0}^{d} f(x_j)\alpha_j(x)\ , \qquad \text{for some intervals } x_j\ \in\ [x_i,x_{i+1}]$$

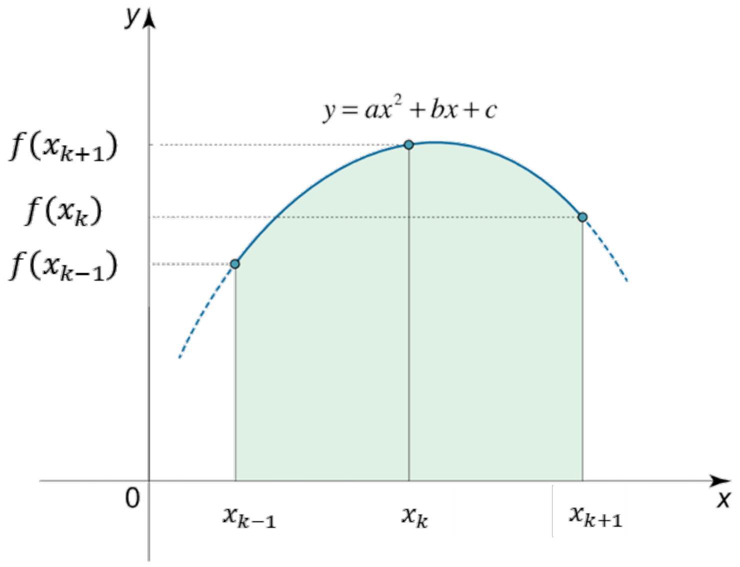
- For implementation, we matches the interval to the integration points, $i,\ j\ \rightarrow k$, then

$$I_N(f)=\sum_{k=0}^{Q} f(x_k)w_k(x)=\sum_{k=0}^{Q} f(x_k)\int_a^b l_k(x)dx$$

Various numerical integrations and error analysis

- **Interpolation approach**
  - Simpson's rule
    - ➔ Using piece-wise interpolation, each interpolation interval is $[x_{k-1}, x_{k+1}]$ with three data points.

$$a = x_0 \le x_2 \le x_4 \le \cdots \le x_{N-2} \le x_N = b$$



$$\int_{x_{k-1}}^{x_{k+1}} f(x)dx \approx \int_{x_{k-1}}^{x_{k+1}} a_k x^2 + b_k x + c_k dx = \left[\frac{a_k}{3}x^3 + \frac{b_k}{2}x^2 + c_k x\right]_{x_{k-1}}^{x_{k+1}}$$

Various numerical integrations and error analysis

- **Interpolation approach**
  - Simpson's rule
    - ➔ Where $f(x_k) = f_k$, (or using Lagrange interpolation → next page)

$$\begin{bmatrix} 1 & 1 & 1 \\ x_{k-1} & x_k & x_{k+1} \\ x_{k-1}^2 & x_k^2 & x_{k+1}^2 \end{bmatrix}^T \begin{bmatrix} c_k \\ b_k \\ a_k \end{bmatrix} = \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \end{bmatrix}$$

    - ➔ For uniform spacing, $h_k = x_{k+1} - x_k$ for all k,

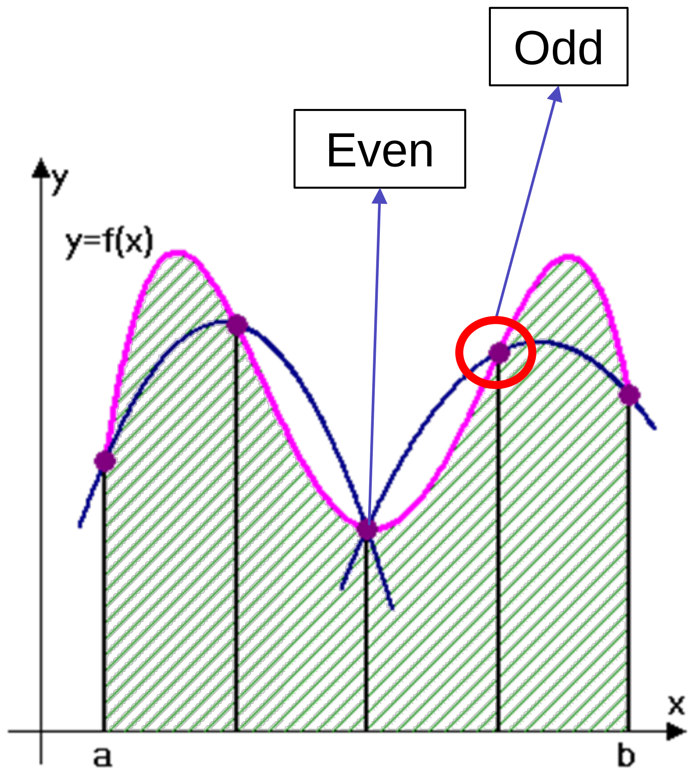
$$\left[\frac{a_k}{3}x^3 + \frac{b_k}{2}x^2 + c_k\right]_{x_{k-1}}^{x_{k+1}} = \frac{h}{3}[f_{k-1} + 4f_k + f_{k+1}]$$

    - ➔ For the entire domain,

$$I \approx \frac{h_k}{3}\left(f_0 + f_N + 4\sum_{\substack{k=1 \\ k=odd}}^{N-1} f_k + 2\sum_{\substack{k=2 \\ k=even}}^{N-2} f_k\right)$$

    - ➔ Total number of points $n + 1$ must be odd

Highly weighted at center



Various numerical integrations and error analysis

- **Interpolation approach**
  - Error analysis for Simpson's rule(DIY)
    - ➔ (Hint) Using the Lagrange interpolation,

$$f(x) = f(y_k) + (x-y_k)f'(y_k) + \frac{(x-y_k)^2}{2!}f''(y_k) + \frac{(x-y_k)^3}{3!}f'''(y_k) + \cdots$$

$$p_2(x) = \frac{(x-x_k)(x-x_{k+1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})}f(x_{k-1}) + \frac{(x-x_{k-1})(x-x_{k+1})}{(x_k-x_{k-1})(x_k-x_{k+1})}f(x_k) + \frac{(x-x_{k-1})(x-x_k)}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)}f(x_{k+1})$$

$$E_{Simp}(x) = \left|\left|\int_a^b f(x)\,dx - \int_a^b p_2(x)\,dx\right|\right|$$

    - ➔ Simpson's rule has the 5th order accuracy for each interval, and 4th order accuracy for whole domain.
    - ➔ (Another) Just observing the form of the Simpson's rule,

$$S(f) = \frac{2}{3}R(f) + \frac{1}{3}T(f),$$

$$\text{where S: Simpson, R: Rectangle, and T: Trapezoidal}$$

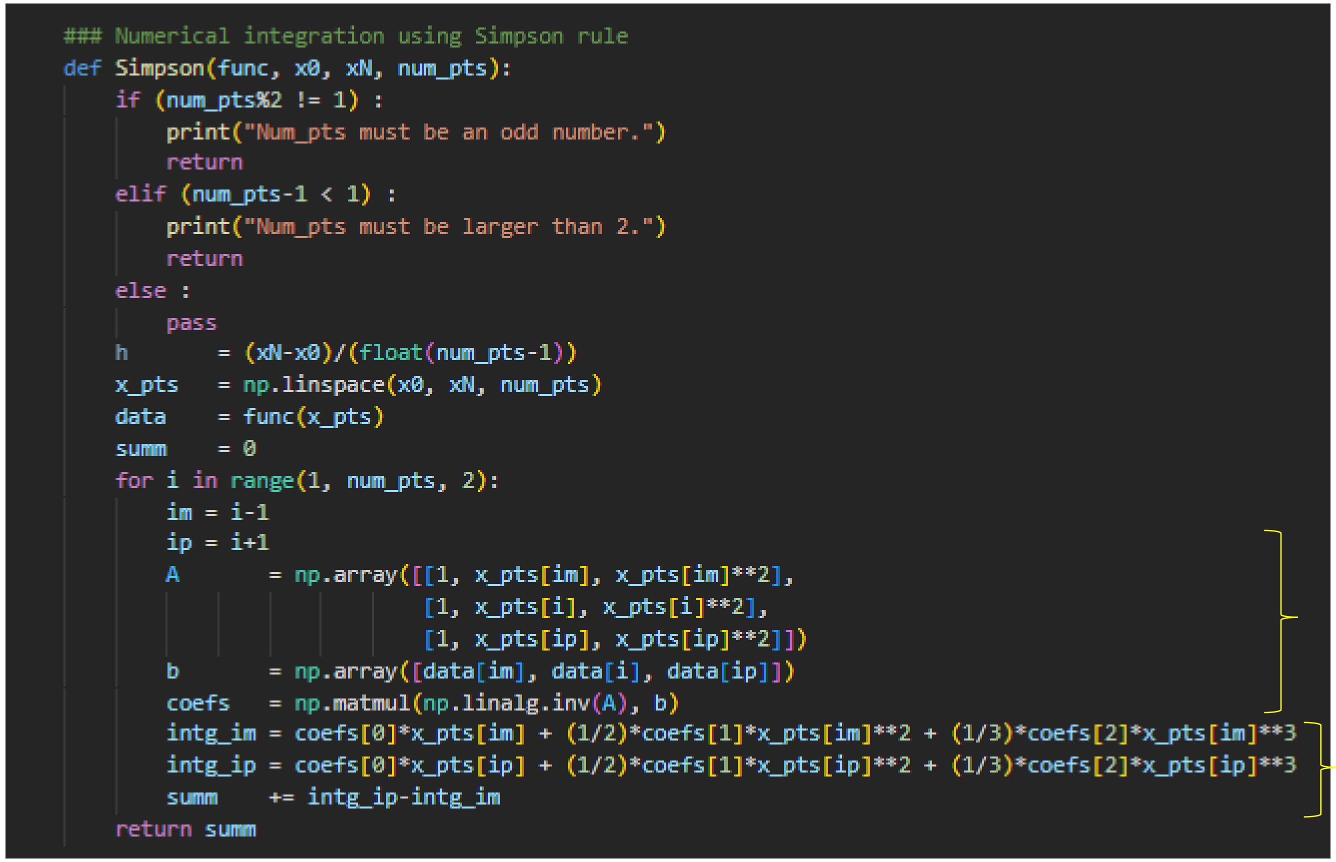
$$R: \int_{x_{k-1}}^{x_{k+1}} f(x)dx \approx 2h_k f(y_k), \qquad T: \int_{x_k}^{x_{k+1}} f(x)dx \approx \frac{h_k}{2}(f_k + f_{k+1})$$

Various numerical integrations and error analysis

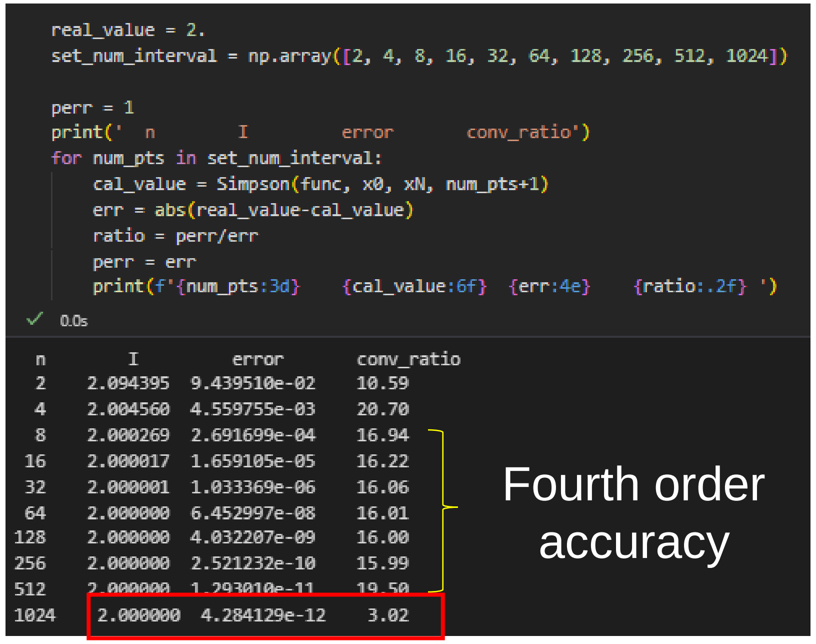
- **Interpolation approach**
  - Code for Simpson's rule (Type1)

```python
import numpy as np
```

✓ 0.0s



```python
### Numerical integration using Simpson rule
def Simpson(func, x0, xN, num_pts):
    if (num_pts%2 != 1) :
        print("Num_pts must be an odd number.")
        return
    elif (num_pts-1 < 1) :
        print("Num_pts must be larger than 2.")
        return
    else :
        pass
    h       = (xN-x0)/(float(num_pts-1))
    x_pts   = np.linspace(x0, xN, num_pts)
    data    = func(x_pts)
    summ    = 0
    for i in range(1, num_pts, 2):
        im = i-1
        ip = i+1
        A       = np.array([[1, x_pts[im], x_pts[im]**2],
                            [1, x_pts[i], x_pts[i]**2],
                            [1, x_pts[ip], x_pts[ip]**2]])
        b       = np.array([data[im], data[i], data[ip]])
        coefs   = np.matmul(np.linalg.inv(A), b)
        intg_im = coefs[0]*x_pts[im] + (1/2)*coefs[1]*x_pts[im]**2 + (1/3)*coefs[2]*x_pts[im]**3
        intg_ip = coefs[0]*x_pts[ip] + (1/2)*coefs[1]*x_pts[ip]**2 + (1/3)*coefs[2]*x_pts[ip]**3
        summ    += intg_ip-intg_im
    return summ
```



```python
real_value = 2.
set_num_interval = np.array([2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])

perr = 1
print('  n         I           error       conv_ratio')
for num_pts in set_num_interval:
    cal_value = Simpson(func, x0, xN, num_pts+1)
    err = abs(real_value-cal_value)
    ratio = perr/err
    perr = err
    print(f'{num_pts:3d}    {cal_value:6f}  {err:4e}    {ratio:.2f} ')
```

✓ 0.0s

```
  n        I          error       conv_ratio
  2    2.094395  9.439510e-02    10.59
  4    2.004560  4.559755e-03    20.70
  8    2.000269  2.691699e-04    16.94
 16    2.000017  1.659105e-05    16.22
 32    2.000001  1.033369e-06    16.06
 64    2.000000  6.452997e-08    16.01
128    2.000000  4.032207e-09    16.00
256    2.000000  2.521232e-10    15.99
512    2.000000  1.293010e-11    19.50
1024    2.000000  4.284129e-12    3.02
```

Fourth order accuracy

$$\begin{bmatrix} 1 & 1 & 1 \\ x_{k-1} & x_k & x_{k+1} \\ x_{k-1}^2 & x_k^2 & x_{k+1}^2 \end{bmatrix}^T \begin{bmatrix} c_k \\ b_k \\ a_k \end{bmatrix} = \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \end{bmatrix}$$

$$\left[\frac{a_k}{3}x^3 + \frac{b_k}{2}x^2 + c_k x\right]_{x_{k-1}}^{x_{k+1}}$$

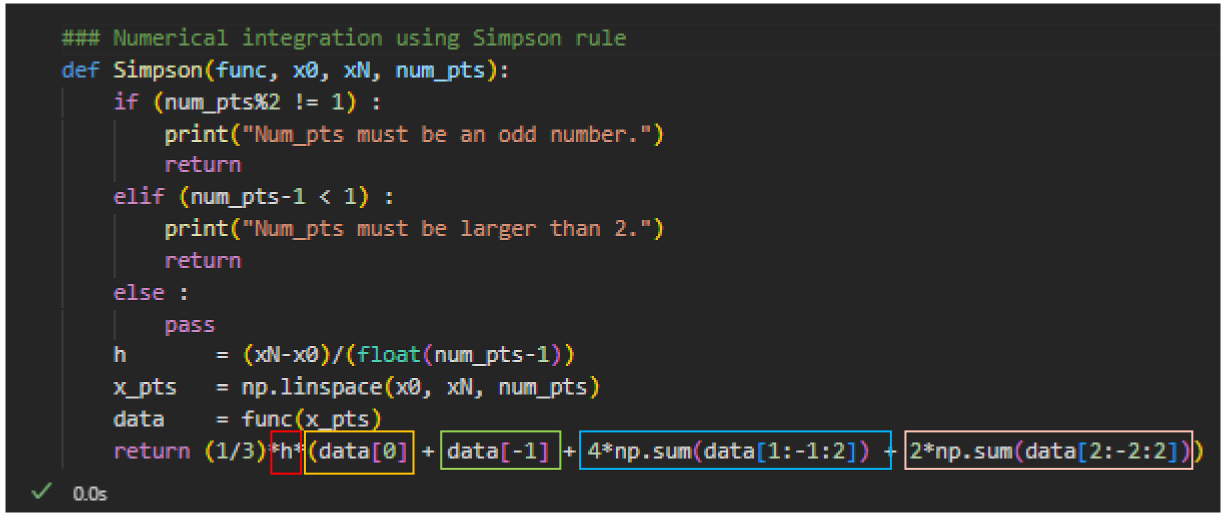
Various numerical integrations and error analysis

- **Interpolation approach**
  - Code for Simpson's rule(Type2)

$$I_N(f) = \frac{\boxed{h}}{3}\left(\boxed{f_0} + \boxed{f_N} + \boxed{4\sum_{\substack{k=1\\k=odd}}^{N-1} f_k} + \boxed{2\sum_{\substack{k=2\\k=even}}^{N-2} f_k}\right)$$

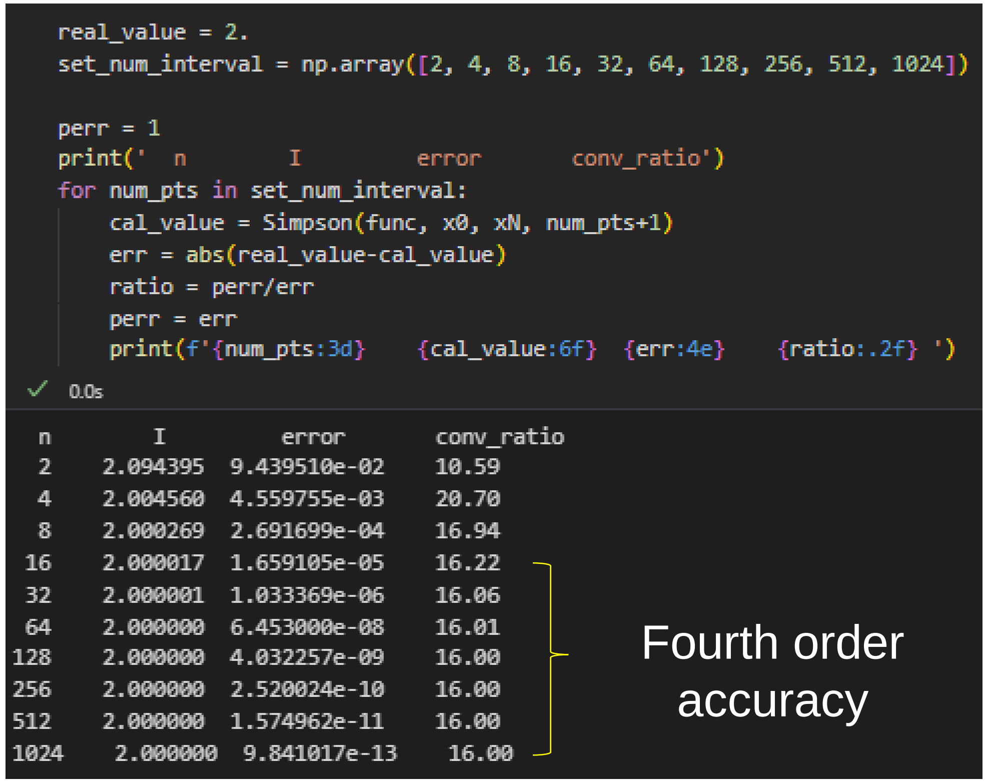
```python
import numpy as np
```

✓ 0.0s



```python
### Numerical integration using Simpson rule
def Simpson(func, x0, xN, num_pts):
    if (num_pts%2 != 1) :
        print("Num_pts must be an odd number.")
        return
    elif (num_pts-1 < 1) :
        print("Num_pts must be larger than 2.")
        return
    else :
        pass
    h      = (xN-x0)/(float(num_pts-1))
    x_pts  = np.linspace(x0, xN, num_pts)
    data   = func(x_pts)
    return (1/3)*h*(data[0] + data[-1] + 4*np.sum(data[1:-1:2]) + 2*np.sum(data[2:-2:2]))
```

✓ 0.0s



```python
real_value = 2.
set_num_interval = np.array([2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])

perr = 1
print('  n        I          error        conv_ratio')
for num_pts in set_num_interval:
    cal_value = Simpson(func, x0, xN, num_pts+1)
    err = abs(real_value-cal_value)
    ratio = perr/err
    perr = err
    print(f'{num_pts:3d}    {cal_value:6f}  {err:4e}    {ratio:.2f} ')
```

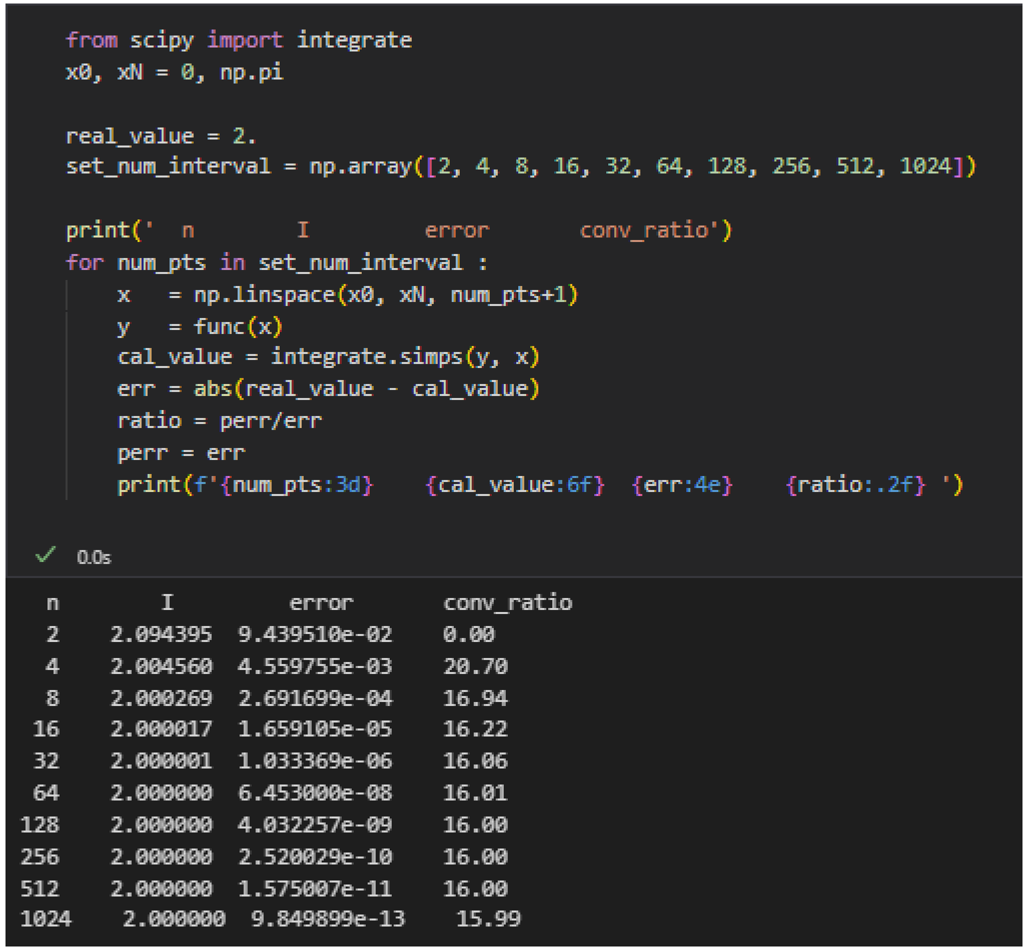
✓ 0.0s

```
  n        I          error        conv_ratio
  2    2.094395  9.439510e-02    10.59
  4    2.004560  4.559755e-03    20.70
  8    2.000269  2.691699e-04    16.94
 16    2.000017  1.659105e-05    16.22
 32    2.000001  1.033369e-06    16.06
 64    2.000000  6.453000e-08    16.01
128    2.000000  4.032257e-09    16.00
256    2.000000  2.520024e-10    16.00
512    2.000000  1.574962e-11    16.00
1024    2.000000  9.841017e-13    16.00
```

Fourth order accuracy

Various numerical integrations and error analysis

- **Interpolation approach**
  - Code for Simpson's rule,
    - ➔ Using Scipy library,
      - ✓ $\mathrm{scipy.integrate.simps}(\mathrm{y}, \mathrm{x})$
    - ➔ y is the data at the data point x



```
from scipy import integrate
x0, xN = 0, np.pi

real_value = 2.
set_num_interval = np.array([2, 4, 8, 16, 32, 64, 128, 256, 512, 1024])

print('  n        I         error       conv_ratio')
for num_pts in set_num_interval :
    x    = np.linspace(x0, xN, num_pts+1)
    y    = func(x)
    cal_value = integrate.simps(y, x)
    err = abs(real_value - cal_value)
    ratio = perr/err
    perr = err
    print(f'{num_pts:3d}    {cal_value:6f}  {err:4e}    {ratio:.2f} ')
```

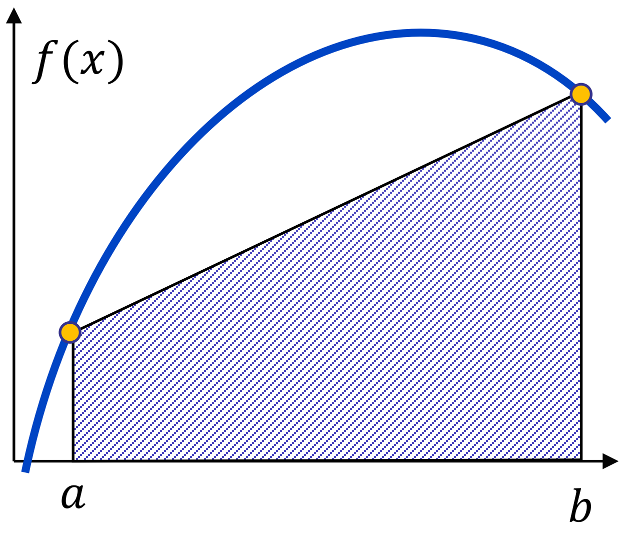
✓ 0.0s

```
  n        I         error       conv_ratio
  2     2.094395  9.439510e-02    0.00
  4     2.004560  4.559755e-03    20.70
  8     2.000269  2.691699e-04    16.94
 16     2.000017  1.659105e-05    16.22
 32     2.000001  1.033369e-06    16.06
 64     2.000000  6.453000e-08    16.01
128     2.000000  4.032257e-09    16.00
256     2.000000  2.520029e-10    16.00
512     2.000000  1.575007e-11    16.00
1024     2.000000  9.849899e-13    15.99
```

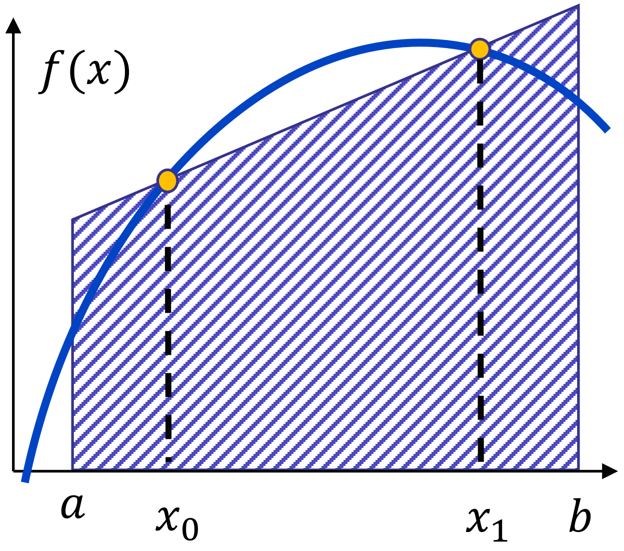
Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ➔ Good data points make good numerical integration!

Using end points



Using proper points in the interval



Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ➔ (Condition 1) Let $q(x)$ be a nontrivial polynomial of degree $n+1$ on $[a,b]$ such that

$$\int_a^b x^i q(x)\,dx = 0, \qquad \text{where } 0 \le i \le n$$

and for $x_i \in [a,b],\ i = 0,1,2,\cdots,n\ ,\ q(x_i) = 0$ → Roots of $q(x)$, later called 'the gaussian nodes'

- ➔ (Conditions 2) Let $f(x)$ be any polynomial of degree $\le 2n+1$ and divide $f(x)$ by $q(x)$ then, we could obtain a quotient $p(x)$ and a remainder $r(x)$ which have degree at most n. So,

$$f(x) = p(x)q(x) + r(x)$$

- ➔ Then,

$$\int_a^b p(x)q(x)dx = \int_a^b (\gamma_0 + \gamma_1 x + \cdots + \gamma_n x^n)q(x)dx = 0$$

$$f(x_i) = p(x_i)q(x_i) + r(x_i) = r(x_i)$$

Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature

➔ So,

$$\int_a^b f(x)dx = \int_a^b p(x)q(x)dx + \int_a^b r(x)dx = \int_a^b r(x)dx$$
$$= \sum_{i=0}^{n} \lambda_i r(x_i) = \sum_{i=0}^{n} \lambda_i f(x_i) = \sum_{i=0}^{n} f(x_i) \int_a^b l_i(x)dx \rightarrow (12page)$$

➔ In other words,

✓ If we could find such $q(x)$ with proper data points, x_i(gaussian nodes), and proper weights, λ_i,

✓ So that, we could calculate 'exact' value for the integral of polynomial, $\int_a^b f(x)dx$ with $\sum_{i=0}^{n} \lambda_i f(x_i)$ at most the degree of $2n+1$.

➔ But,

✓ Only for polynomials? Is it useful? → Interpolation!

Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature

➔ (Optional) By the Stone-Weierstrass theorem,

Any continuous real(or complex) function, $f(x)$, on $[a, b]$, there exist a sequence of polynomials $\phi_m(x)$ such that,

$$\lim_{m\to\infty} \phi_m(x) = f(x)$$

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m$$

✓ Basic philosophy for the polynomial interpolation

➔ (Remaining question) then how to determine such $q(x)$ and gaussian nodes, x_i?

✓ Start from the domain $[-1, 1]$ and extends to $[a, b]$.

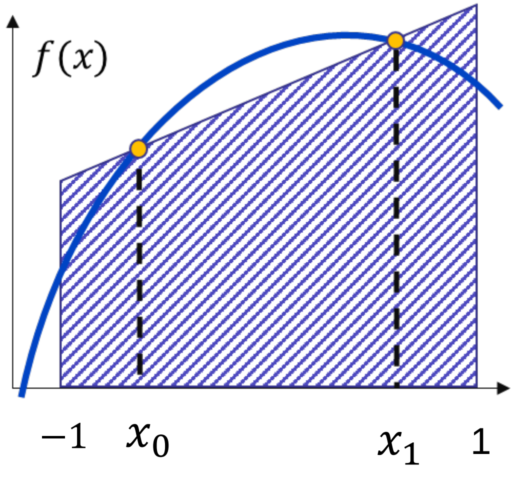
- Ex1) Two-point Gaussian quadrature,

$$q(x) = c_0 + c_1 x + c_2 x^2$$

$$\int_{-1}^{1} q(x)dx = \int_{-1}^{1} xq(x)dx = 0 \rightarrow \begin{cases} 2c_0 + \frac{2}{3}c_2 = 0 \\ c_1 = 0 \end{cases}$$

Choose, $c_0 = 1, c_2 = -3,$ then

$$q(x) = -3x^2 + 1, \quad \text{and its roots are,} \quad x_0 = -\sqrt{\frac{1}{3}}, \quad x_1 = \sqrt{\frac{1}{3}}$$



Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - Ex2) Three-points Gaussian quadrature

$$q(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

$$\int_{-1}^{1} q(x)dx = \int_{-1}^{1} xq(x)dx = \int_{-1}^{1} x^2 q(x)dx = 0$$

$$\begin{cases} 2c_0 + \frac{2}{3}c_2 = 0 \\ \frac{2}{3}c_1 + \frac{2}{5}c_3 = 0 \\ \frac{2}{3}c_0 + \frac{2}{5}c_2 = 0 \end{cases} \rightarrow c_0 = c_2 = 0, \qquad \text{Choose } a\ convenient\ solution, \qquad c_1 = -3, \qquad c_3 = 5, \qquad \text{then}$$

$$q(x) = 5x^3 - 3x, \qquad \text{and its roots are,} \qquad x_0 = -\sqrt{\frac{3}{5}}, \qquad x_1 = 0, \qquad x_2 = \sqrt{\frac{3}{5}}$$

Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ➔ (Remaining question) then how to determine weights, λ_i?
      - ✓ Since the integration is a linear process,

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=0}^{n} \lambda_i f(x_i) \rightarrow \int_{-1}^{1} x^i dx = \sum_{j=0}^{n} \lambda_j x^i$$

      - ✓ Ex1) Two-point Gaussian quadrature

$$\int_{-1}^{1} f(x)\,dx \approx \lambda_0 f(x_0) + \lambda_1 f(x_1) = \lambda_0 f\left(-\sqrt{\frac{1}{3}}\right) + \lambda_1 f\left(\sqrt{\frac{1}{3}}\right)$$

$$\lambda_0 + \lambda_1 = \int_{-1}^{1} 1\,dx = 2$$

$$-\sqrt{\frac{1}{3}}\lambda_0 + \sqrt{\frac{1}{3}}\lambda_1 = \int_{-1}^{1} x\,dx = 0$$

$$\lambda_0 = 1, \qquad \lambda_1 = 1$$

Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ✓ Ex2) Three-point Gaussian quadrature

$$\int_{-1}^{1} f(x)\,dx \approx \lambda_0 f(x_0) + \lambda_1 f(x_1) + \lambda_2 f(x_2) = \lambda_0 f\left(-\sqrt{\frac{3}{5}}\right) + \lambda_1 f(0) + \lambda_2 f\left(\sqrt{\frac{3}{5}}\right)$$

$$\begin{cases} \lambda_0 + \lambda_1 + \lambda_2 = \int_{-1}^{1} 1\,dx = 2 \\ -\sqrt{\frac{3}{5}}\lambda_0 + \sqrt{\frac{3}{5}}\lambda_2 = \int_{-1}^{1} x\,dx = 0 \\ \frac{3}{5}\lambda_0 + \frac{3}{5}\lambda_2 = \int_{-1}^{1} x^2\,dx = \frac{2}{3} \end{cases} \rightarrow \begin{cases} \lambda_0 + \lambda_1 + \lambda_2 = 2 \\ \lambda_0 - \lambda_2 = 0 \\ \lambda_0 + \lambda_2 = \frac{10}{9} \end{cases} \rightarrow \lambda_0 = -\frac{5}{9}, \quad \lambda_1 = \frac{8}{9}, \quad \lambda_2 = \frac{5}{9}$$

→ If f is of the from $ax^2 + bx + c$, then, the Gaussian quadrature has the exact solution for $\int_{-1}^{1} f(x)\,dx$.

Various numerical integrations and error analysis

- **Interpolation approach**
  - Table of Gauss-Legendre quadrature nodes and weights

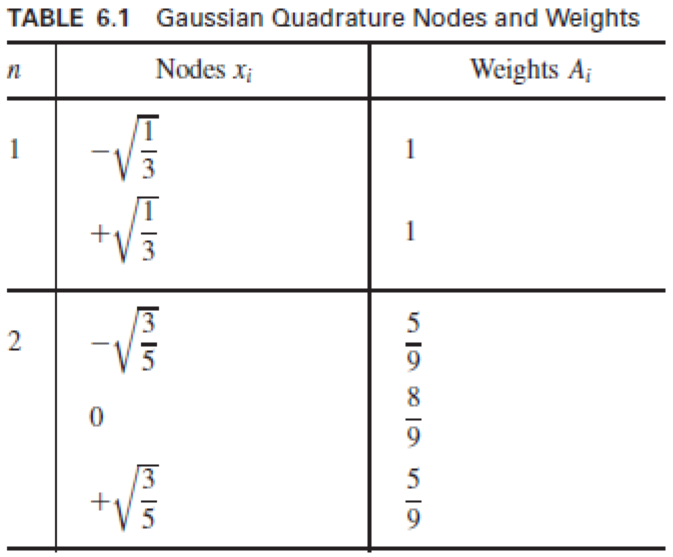
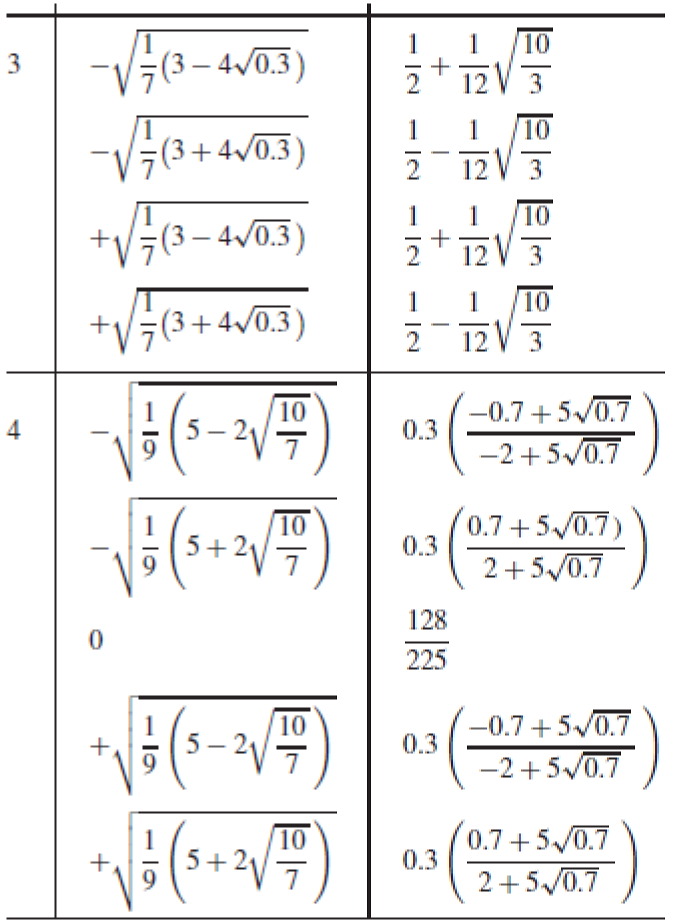


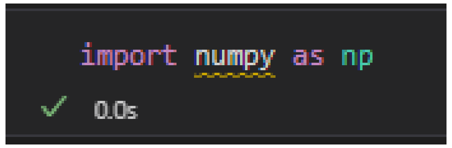
TABLE 6.1 Gaussian Quadrature Nodes and Weights



n	Nodes x_i	Weights A_i
1	$-\sqrt{\frac{1}{3}}$	1
	$+\sqrt{\frac{1}{3}}$	1
2	$-\sqrt{\frac{3}{5}}$	$\frac{5}{9}$
	0	$\frac{8}{9}$
	$+\sqrt{\frac{3}{5}}$	$\frac{5}{9}$
3	$-\sqrt{\frac{1}{7}(3-4\sqrt{0.3})}$	$\frac{1}{2}+\frac{1}{12}\sqrt{\frac{10}{3}}$
	$-\sqrt{\frac{1}{7}(3+4\sqrt{0.3})}$	$\frac{1}{2}-\frac{1}{12}\sqrt{\frac{10}{3}}$
	$+\sqrt{\frac{1}{7}(3-4\sqrt{0.3})}$	$\frac{1}{2}+\frac{1}{12}\sqrt{\frac{10}{3}}$
	$+\sqrt{\frac{1}{7}(3+4\sqrt{0.3})}$	$\frac{1}{2}-\frac{1}{12}\sqrt{\frac{10}{3}}$
4	$-\sqrt{\frac{1}{9}\left(5-2\sqrt{\frac{10}{7}}\right)}$	$0.3\left(\frac{-0.7+5\sqrt{0.7}}{-2+5\sqrt{0.7}}\right)$
	$-\sqrt{\frac{1}{9}\left(5+2\sqrt{\frac{10}{7}}\right)}$	$0.3\left(\frac{0.7+5\sqrt{0.7})}{2+5\sqrt{0.7}}\right)$
	0	$\frac{128}{225}$
	$+\sqrt{\frac{1}{9}\left(5-2\sqrt{\frac{10}{7}}\right)}$	$0.3\left(\frac{-0.7+5\sqrt{0.7}}{-2+5\sqrt{0.7}}\right)$
	$+\sqrt{\frac{1}{9}\left(5+2\sqrt{\frac{10}{7}}\right)}$	$0.3\left(\frac{0.7+5\sqrt{0.7}}{2+5\sqrt{0.7}}\right)$

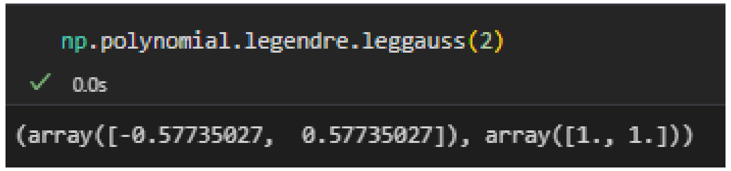
Various numerical integrations and error analysis

- **Interpolation approach**
  - Codes for Gauss-Legendre quadrature
    - ➔ Using the function "numpy.polynomial.legendre.leggauss()", we could obtain the gaussian nodes and their weights easily.



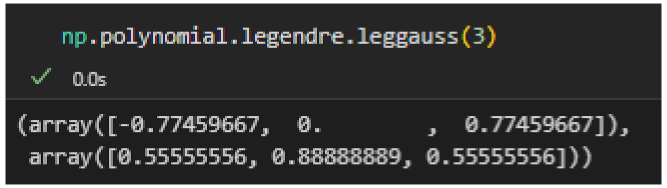
```
import numpy as np
✓ 0.0s
```

✓ Ex1) Nodes and weights for two-point Gauss-Legendre quadrature



```
np.polynomial.legendre.leggauss(2)
✓ 0.0s
(array([-0.57735027,  0.57735027]), array([1., 1.]))
```

✓ Ex2) Nodes and weights for three-point Gauss-Legendre quadrature



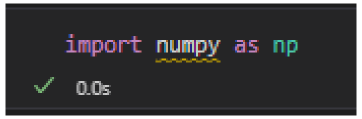
```
np.polynomial.legendre.leggauss(3)
✓ 0.0s
(array([-0.77459667,  0.        ,  0.77459667]),
 array([0.55555556, 0.88888889, 0.55555556]))
```

Various numerical integrations and error analysis

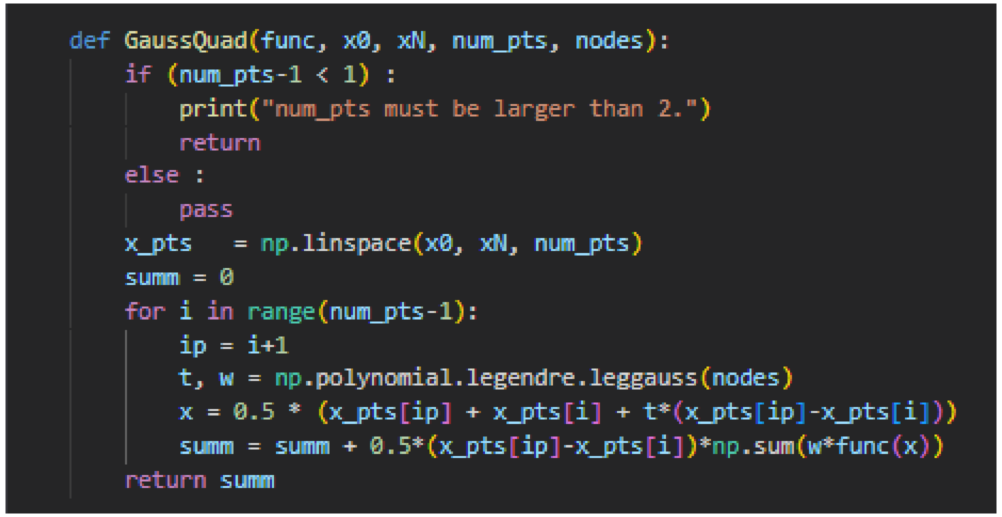
- **Interpolation approach**
  - Codes for Gauss-Legendre quadrature
    - ➔ Since the Gaussian nodes and weights come from the integration on $[-1, 1]$, the change of variables is required.

$$x = \frac{b + a + t(b-a)}{2}, \qquad t \in [-1, 1]$$

$$\int_a^b f(x)\,dx \quad \Rightarrow \quad \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b+a+t(b-a)}{2}\right) dt$$



```
import numpy as np
✓ 0.0s
```



```
def GaussQuad(func, x0, xN, num_pts, nodes):
    if (num_pts-1 < 1) :
        print("num_pts must be larger than 2.")
        return
    else :
        pass
    x_pts   = np.linspace(x0, xN, num_pts)
    summ = 0
    for i in range(num_pts-1):
        ip = i+1
        t, w = np.polynomial.legendre.leggauss(nodes)
        x = 0.5 * (x_pts[ip] + x_pts[i] + t*(x_pts[ip]-x_pts[i]))
        summ = summ + 0.5*(x_pts[ip]-x_pts[i])*np.sum(w*func(x))
    return summ
```

Exercise!

Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ➔ (Optional) Generally, the x_i are the roots of Legendre polynomial,
      - ✓ (Gram-Schmidt orthogonalization process)

$$< P_m, P_n > = \int_{-1}^{1} P_m(x) P_n(x) dx = \frac{2}{2n+1} \delta_{nm}$$

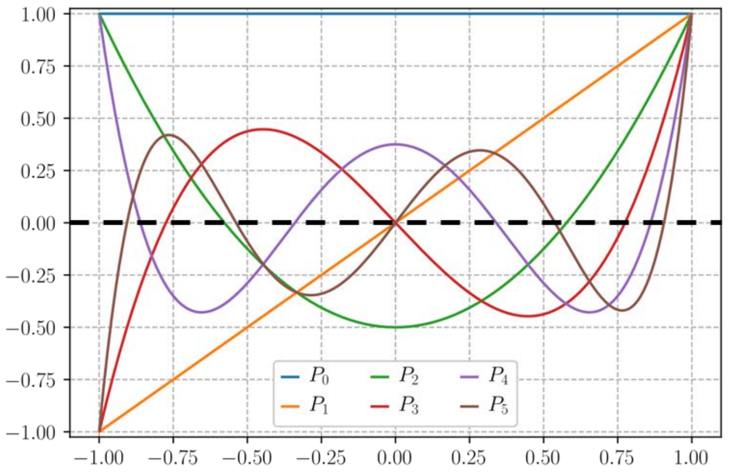
$$f(x) \approx f_n(x) = \sum_{l=0}^{n} \alpha_l P_l(x), \qquad \alpha_l = \frac{< f_n, P_l >}{< P_l, P_l >}$$

      - ✓ (Bonnet's recursion formula)

$$P_0 = 1,\ P_1 = x,\ \rightarrow (n+1)P_{n+1}(x) = (2n+1)P_n(x) - nP_{n-1}(x)$$

      - ✓ (Rodrigues' formula)

$$P_n = \frac{1}{2^n n!} \frac{d^n}{dx^n} \left(x^2 - 1\right)^n$$



Various numerical integrations and error analysis

- **Interpolation approach**
  - Basic concept of Gauss-Legendre quadrature
    - ✓ (Optional) Generally,
      - we could calculate weights for $n+1$ points Gaussian quadrature using Vandermonde matrix

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \vdots & x_2^n \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}, \lambda = \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix}, \quad \text{and } F = \begin{pmatrix} \int_{-1}^{1} x^0 dx \\ \int_{-1}^{1} x^1 dx \\ \int_{-1}^{1} x^2 dx \\ \vdots \\ \int_{-1}^{1} x^n dx \end{pmatrix} \rightarrow V^T\lambda = F$$

      - Or, using the Legendre polynomial,

$$\lambda_i = -\frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$$

Various numerical integrations and error analysis

- **Statistical approach**
  - Monte Carlo integration
    - ➔ Basic concept of the Monte Carlo integration

$$I(f) = \int_\Omega f(\bar{\boldsymbol{x}})\, d\bar{\boldsymbol{x}} \approx \frac{V}{n}\sum_{i=1}^{n} f(\bar{\boldsymbol{x}}_i) = I_n(f)$$

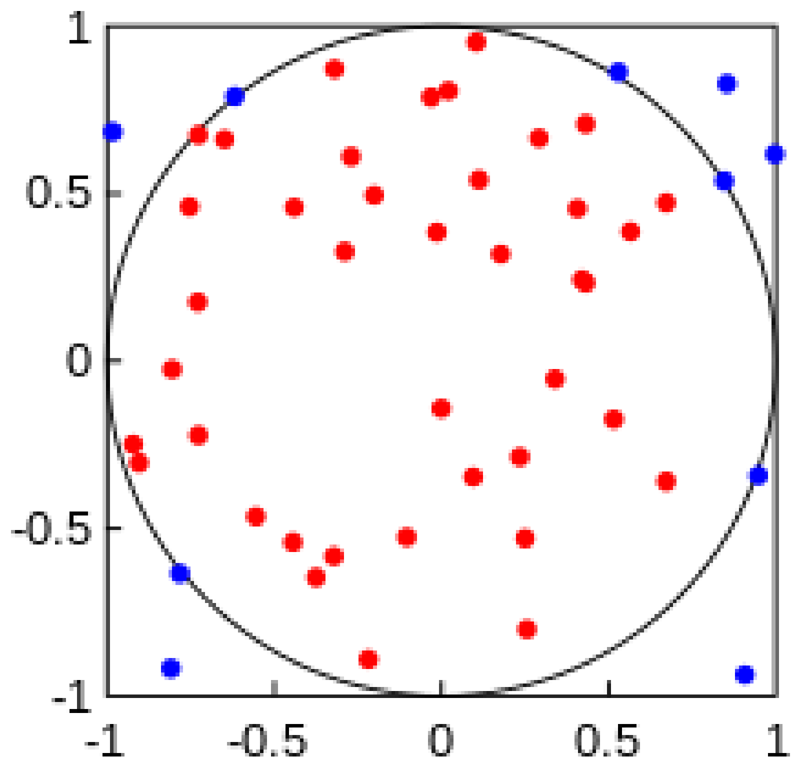
- ➔ $\bar{\boldsymbol{x}}_i \in S,\ i = 1, 2, \cdots, n$ are random variables in Ω, following the probability density function, $p(\bar{\boldsymbol{x}})$

$$V = \int_\Omega d\,\bar{\boldsymbol{x}}$$

- ➔ Expectation, and variance

$$E[f(\bar{\boldsymbol{x}}_i)] = \int_\Omega f(\bar{\boldsymbol{x}})p(\bar{\boldsymbol{x}})d\bar{\boldsymbol{x}} = \frac{I(f)}{V}$$

$$\sigma^2[f(\bar{\boldsymbol{x}}_i)] = \int_\Omega (f(\bar{\boldsymbol{x}}) - E[f(\bar{\boldsymbol{x}}_i)])^2\, p(\bar{\boldsymbol{x}})d\bar{\boldsymbol{x}} = E[f^2(\bar{\boldsymbol{x}})] - E[f(\bar{\boldsymbol{x}}_i)]^2$$



Various numerical integrations and error analysis

- **Statistical approach**
  - Monte Carlo integration

$$E(I_n) = \frac{V}{n} E\left[\sum_{i=1}^{n} f(\bar{\boldsymbol{x}}_i)\right] = \frac{V}{\mathrm{n}} \sum_{i=1}^{n} E[f(\bar{\boldsymbol{x}}_i)] = I$$

$$\sigma^2(I_n) = E[(I_n - E[I_n])^2] = \left(\frac{V}{n}\right)^2 E\left[\left(\sum_{i=1}^{n} f(\bar{\boldsymbol{x}}_i) - \sum_{i=1}^{n} E[f(\bar{\boldsymbol{x}}_i)]\right)^2\right] = \left(\frac{V}{n}\right)^2 \sum_{i=1}^{n}\left(E[f^2(\bar{\boldsymbol{x}}_i)] - E[f(\bar{\boldsymbol{x}}_i)]^2\right) = \frac{V^2}{n}\sigma^2[f(\bar{\boldsymbol{x}}_{\boldsymbol{i}})]$$

$$I = I_n \pm \frac{V}{\sqrt{n}} \sigma[f(\bar{\boldsymbol{x}}_{\boldsymbol{i}})]$$

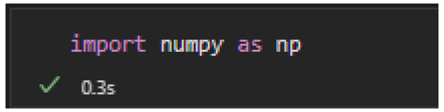
$$\rightarrow \lim_{n\to\infty} I_n = I$$

Various numerical integrations and error analysis

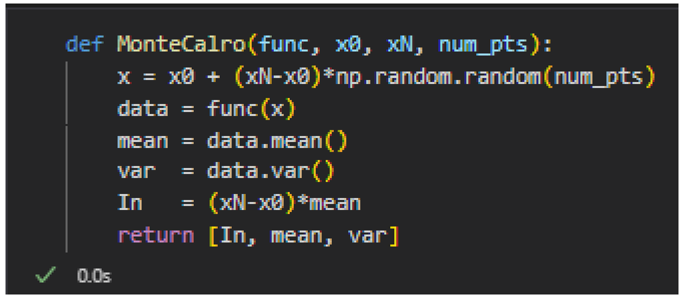
- **Statistical approach**
  - Monte Carlo integration for 1-dimensional with the uniform extraction
    - ➔ Uniform extraction → the probability density function is uniform on $[0, 1)$.

$$\bar{x}_i \in [a, b), \qquad p(\bar{x}) = \frac{1}{b-a}$$

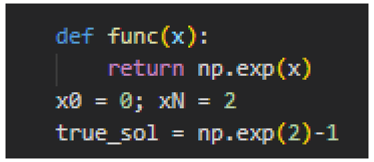
$$\int_0^2 e^x \, dx = ?$$



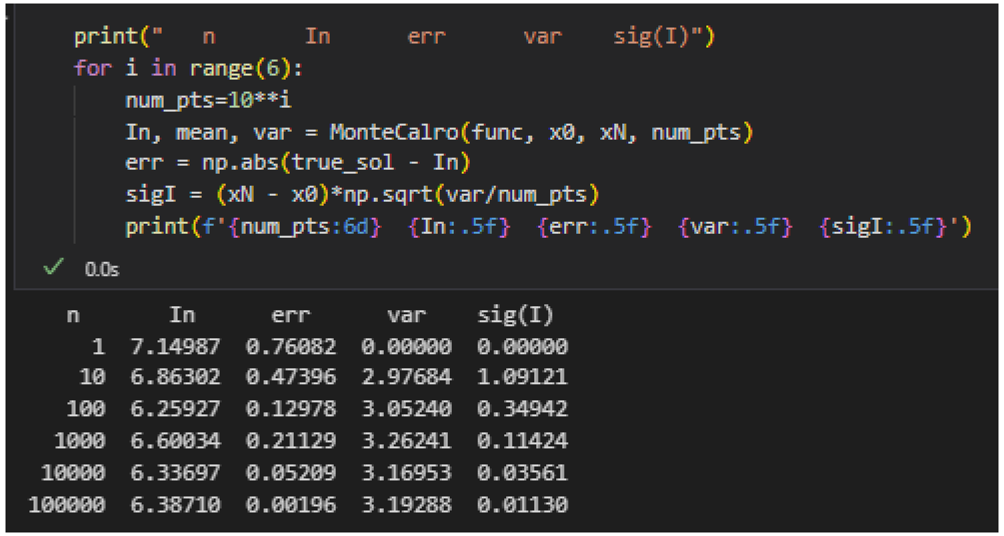
```
import numpy as np
```
✓ 0.3s



```
def MonteCalro(func, x0, xN, num_pts):
    x = x0 + (xN-x0)*np.random.random(num_pts)
    data = func(x)
    mean = data.mean()
    var  = data.var()
    In   = (xN-x0)*mean
    return [In, mean, var]
```
✓ 0.0s



```
def func(x):
    return np.exp(x)
x0 = 0; xN = 2
true_sol = np.exp(2)-1
```



```
print("    n        In       err       var     sig(I)")
for i in range(6):
    num_pts=10**i
    In, mean, var = MonteCalro(func, x0, xN, num_pts)
    err = np.abs(true_sol - In)
    sigI = (xN - x0)*np.sqrt(var/num_pts)
    print(f'{num_pts:6d}  {In:.5f}  {err:.5f}  {var:.5f}  {sigI:.5f}')
```
✓ 0.0s

```
     n       In       err       var     sig(I)
     1  7.14987  0.76082  0.00000  0.00000
    10  6.86302  0.47396  2.97684  1.09121
   100  6.25927  0.12978  3.05240  0.34942
  1000  6.60034  0.21129  3.26241  0.11424
 10000  6.33697  0.05209  3.16953  0.03561
100000  6.38710  0.00196  3.19288  0.01130
```

Q&A ***Thanks for listening***