dkq5juuex

April 18, 2023

1 Computer HW2

1.1 (a) Derive the discrete calculus integration filter for 4th order polynomial

[1]:
```
## Imports
import numpy as np

## Allocate Discrete Data
n = 15 # Simpsons rule requires odd n
x = np.linspace(-1, 1, num=n)
F = np.power(x,4)+3*np.power(x,2)
```

1.1.1 Use trapezoidal method:

$$I \approx \sum_{i=0}^{n-1} \frac{(x_{i+1} - x_i)}{2} \cdot (f(x_{i+1}) + f(x_i))$$

The filter is then (Δx being the constant linear spacing between function evaluation points and $*$ denoting the convolution operation):

$$\begin{bmatrix} \frac{\Delta x}{2} & \frac{\Delta x}{2} \end{bmatrix} * \begin{bmatrix} f(x_1) & f(x_2) & \dots & f(x_N) \end{bmatrix}$$

[2]:
```
def trapezoidal_filter(x_current: float, x_next:float) -> np.ndarray:
    """Returns a 1D trapezoidal integration filter"""
    assert( x_next > x_current )
    return np.array([0.5,0.5])*(x_next - x_current)

def trapezoidal_integration(x: np.ndarray, F: np.ndarray) -> float:
    """Integrates a function F over a given discrete domain x using trapezoidal␣
 ↪integration"""
    stride = 1
    integral_trapezoidal = 0

    ## Perform convolution:
    for ii in range(0, len(x)-1, stride):
        integral_trapezoidal += np.dot( trapezoidal_filter(x[ii],x[ii+1]), F[ii:
 ↪ii+1+stride] )
    return integral_trapezoidal
```

```
print("Integral computed using trapezoidal rule: ", trapezoidal_integration(x,␣
↪F))
```

```
Integral computed using trapezoidal rule:  2.433985839233653
```

1.1.2 Use Simpsons method:

$$I = \sum_{i=1,\ \text{stride}\ 2}^{n-1} \frac{(x_{i+1} - x_{i-1})}{6} \cdot (f(x_{i+1}) + 4 \cdot f(x_i) + f(x_{i-1}))$$

The filter is then (Δx being the constant linear spacing between function evaluation points and $*$ denoting the convolution operation with stride 2):

$$\begin{bmatrix} \frac{\Delta x}{3} & 4 \cdot \frac{\Delta x}{3} & \frac{\Delta x}{3} \end{bmatrix} * \begin{bmatrix} f(x_1) & f(x_2) & \dots & f(x_N) \end{bmatrix}$$

[3]:
```
def simpson_filter(x_previous: float, x_current: float, x_next:float) -> np.
↪ndarray:
    """Returns a 1D simpson integration filter"""
    assert( x_next > x_current > x_previous )
    return np.array([1/3, 4/3, 1/3])*(x_next - x_current)

def simpson_integration(x: np.ndarray, F: np.ndarray) -> float:
    """Integrates a function F over a given discrete domain x using Simpson␣
↪integration."""
    stride = 2
    integral_simpson = 0

    ## Perform convolution:
    for ii in range(1, len(x), stride):
        integral_simpson += np.dot( simpson_filter(x[ii-1],x[ii],x[ii+1]),␣
↪F[ii-1:ii+stride] )
    return integral_simpson

print("Integral computed using Simpsons rule: ", simpson_integration(x, F))
```

```
Integral computed using Simpsons rule:  2.4001110648340975
```

1.1.3 Use Gauss method:

As seen in the lecture, $n = 2$ point Gauss quadrature is accurate for polynomials of order $2n - 1 = 3$ or less. In order to integrate the given 4th-order polynomial correctly, we need $n = 3$ quadrature points and weights. The exact location can be computed using a system of equations, but are well-known for low n.

For $n = 3$, the quadrature points are (see Wikipedia):

$$(w_i, x_i) = \{(\frac{5}{9}, -\sqrt{\frac{3}{5}}), (\frac{8}{9}, 0), (\frac{5}{9}, +\sqrt{\frac{3}{5}})\}$$

and the filter is then (integral domain is $[-1, 1]$ so no change of variables necessary):

$$\begin{bmatrix} 0 & \frac{5}{9} & \frac{8}{9} & \frac{5}{9} & 0 \end{bmatrix} * \begin{bmatrix} f(-1) & f(-\sqrt{\frac{3}{5}}) & f(0) & +f(\sqrt{\frac{3}{5}}) & f(1) \end{bmatrix}$$

[4]:
```
def gauss_filter(x_0: float = -1, x_n: float = 1) -> np.ndarray:
    """Returns a 1D gauss integration filter"""
    return np.array([0, 5, 8, 5, 0]) / 9

def gauss_integration(x: np.ndarray, F: np.ndarray) -> float:
    """Integrates a function F over a given discrete domain x using Gauss␣
 ↪integration."""
    stride = 0
    integral_gauss = 0

    ## Perform convolution:
    for ii in range(1):
        integral_gauss += np.dot( gauss_filter(), F )
    return integral_gauss

x_gauss = np.array([-1, -np.sqrt(3/5), 0, np.sqrt(3/5), 1])
F_gauss = np.power(x_gauss,4)+3*np.power(x_gauss,2)

print("Integral computed using Gauss rule: ", gauss_integration(x_gauss,␣
 ↪F_gauss))
```

```
Integral computed using Gauss rule:  2.4000000000000004
```

1.2 (b) compute the integral using the derived discrete calculus integration filter and compare with the analytical result

We can compute the integral analytically:

$$\int_{-1}^{1} x^4 + 3x^2 dx = \left[\frac{1}{5}x^5 + \frac{3}{3}x^3\right]_{-1}^{1}$$
$$\left[\frac{1}{5}x^5 + \frac{3}{3}x^3\right]_{-1}^{1} = \left(\frac{1}{5}(1)^5 + \frac{3}{3}(1)^3\right) - \left(\frac{1}{5}(-1)^5 + \frac{3}{3}(-1)^3\right)$$
$$\left(\frac{1}{5} + \frac{3}{3}\right) - \left(-\frac{1}{5} - \frac{3}{3}\right) = \frac{12}{5} = 2.4$$

We can compare our results as follows from the previous computations and see that 3-point Gauss quadrature retrieves the exact integral as expected, while Simpson and trapezoidal have some error in them.

[5]:
```
print("Integral computed using Trapezoidal rule: ", trapezoidal_integration(x,
 ↪F))
print("Integral computed using Simpsons rule: ", simpson_integration(x, F))
print("Integral computed using Gauss rule: ", gauss_integration(x_gauss,
 ↪F_gauss))
```

```
Integral computed using Trapezoidal rule:  2.433985839233653
Integral computed using Simpsons rule:  2.4001110648340975
Integral computed using Gauss rule:  2.4000000000000004
```

Remember we used $n = 15$, so $\Delta x = 0.1\overline{3}$. Generally, the error in trapezoidal integration is third order in Δx and the error in Simpsons integration is fifth order in Δx.

$$\begin{aligned} I_{\text{analytical}} &= 2.4 \\ I_{\text{trapezoidal}} &= 2.433985 \\ I_{\text{simpson}} &= 2.400111 \\ I_{\text{gauss}} &= 2.400000 \end{aligned}$$

1.2.1 (c) Diverging integral shenanigans

We compute the analytical solution of the given integral:

$$\int_{-1}^{1} \frac{1}{x+1.1} dx = \int_{0.1}^{2.1} \frac{1}{u} du = \ln 2.1 - \ln 0.1 = 3.044522$$

We compute the integrals numerically in each subdomain numerically, then sum and compare:

[6]:
```
subdomains = np.array([-1, -0.6, -0.2, 0.2, 0.6, 1])

integral_trapezoid = 0
integral_simpson = 0
integral_gauss = 0

def f(x: np.ndarray) -> np.ndarray:
    return 1/(x+1.1)

for start, end in zip(subdomains[0:],subdomains[1:]):
    x_subdomain = np.linspace(start, end, num=n)
    integral_trapezoid += trapezoidal_integration(x_subdomain, f(x_subdomain))
    integral_simpson += simpson_integration(x_subdomain, f(x_subdomain))

    # Adjust integral bounds for Gauss quadrature (change of variables):
    # x = (start+end)/2 - (end-start)/2 * zeta
    # zeta in [-1, 1]
    integral_gauss += gauss_integration(x_gauss, f(x_gauss*(end-start)/2 +
 ↪(start+end)/2)) * 0.4/2
```

```python
print("Integral computed using Trapezoidal rule: ", integral_trapezoid)
print("Integral computed using Simpsons rule: ", integral_simpson)
print("Integral computed using Gauss rule: ", integral_gauss)
```

```
Integral computed using Trapezoidal rule:  3.0512561973305177
Integral computed using Simpsons rule:  3.044711667014311
Integral computed using Gauss rule:  3.037769562154289
```

$$I_{\text{analytical}} = 3.044522$$
$$I_{\text{trapezoidal}} = 3.051256$$
$$I_{\text{simpson}} = 3.044712$$
$$I_{\text{gauss}} = 3.037770$$

We see that the is some error to all methods. The reason is that our function is diverging and has a singularity in $x = -1.1$, so it is already greatly diverging around $x = -1$.

Surprisingly, we see that Simpsons rule does perform better than Gauss quadrature. The reason for this is that the implemented code further subdivides the subintervals into $n = 15$ subsubintervals, meaning that our Simpsons (and trapezoidal) scheme actually integrates over $5 \cdot 15 = 75$ subsubdomains!

Thus, Simpsons rule fits a quadratic curve into all 75 subsubdomains, which in this case is more accurate than our 3-point Gauss quadrature rule, which fits a fifth-order polynomial into only 5 subdomains.