Computional Homework7 Report

Student: 纪浩正, jihz2023@mail.sustech.edu.cn

1 Simpson Rule

Suppose the integral interval is $[a, b]$, which is divided into n sub-intervals where n is an even number. There are $n+1$ equally spaced nodes $x_0, x_1, \ldots, x_n$, with spacing $h = \frac{b-a}{n}$. For each pair of adjacent sub-intervals, Simpson's rule approximates the integral by fitting a quadratic polynomial over each group of three points.

1.1 Formula Derivation

For each sub-interval $[x_{2k}, x_{2k+2}]$, the local approximation is:

$$\int_{x_{2k}}^{x_{2k+2}} f(x)\,dx \approx \frac{h}{3}\left[f(x_{2k}) + 4f(x_{2k+1}) + f(x_{2k+2})\right]$$

Thus, the composite Simpson's rule over the interval $[a, b]$ is:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{\substack{i=1\\ i\text{ odd}}}^{n-1} f(x_i) + 2\sum_{\substack{i=2\\ i\text{ even}}}^{n-2} f(x_i) + f(x_n)\right]$$

```
def f(x):
    return np.exp(3*x)*np.cos(x)

class Quadrature:
    def __init__(self,f,x0,xn,tol=1e-5):
        self.f=f
        self.x0=x0
        self.xn=xn
        self.tol=1e-5
        self.points=[]

    def CompositeSimpsonRule(self,n):
        x=np.linspace(self.x0,self.xn,n+1)
        h=(self.xn-self.x0)/n
        fn=self.f(x)
        for ii in range(1,n):
            if ii%2==0:
                fn[ii]=2*fn[ii]
```

```
            else:
                fn[ii]=4*fn[ii]
        I=h/3*np.sum(fn)
        return I

x0=0
n=4
xn=np.pi

SimpsonRule4 = Interval.CompositeSimpsonRule(n)
```

2 Adaptive Simpson Quadrature

The adaptive Simpson algorithm automatically subdivides intervals where the approximation is not accurate enough, applying Simpson's rule recursively until the local error is below the specified tolerance.

2.1 Algorithm Description

For an interval $[a, b]$, define the Simpson approximation as:

$$S(a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Divide at the midpoint $m = (a+b)/2$, and compute:

$$S_L = S(a,m), \quad S_R = S(m,b)$$

Error estimation criterion: If

$$|S_L + S_R - S(a,b)| < \varepsilon$$

then accept the result with Richardson extrapolation:

$$I \approx S_L + S_R$$

Otherwise, recursively apply the method to both sub-intervals $[a, m]$ and $[m, b]$.

```
class Quadrature:
    def __init__(self,f,x0,xn,tol=1e-5):
        self.f=f
        self.x0=x0
```

```
        self.xn=xn
        self.tol=1e-5
        self.points=[]

    def _SimpsonRule(self,a,b):
        m = (a + b) / 2
        self.points.extend([a, m, b])
        return (b-a)/6*(self.f(a)+4*self.f((a+b)/2)+self.f(b))

    def _AdaptiveQuad(self,x0,xn):
        S=self._SimpsonRule(x0,xn)
        m=(x0+xn)/2
        L=self._SimpsonRule(x0,m)
        R=self._SimpsonRule(m,xn)
        self.points.append(m)
        if abs(L+R-S)<self.tol:
            return L+R
        else:
            L=self._AdaptiveQuad(x0,m)
            R=self._AdaptiveQuad(m,xn)
            return L+R

    def AdaptiveQuad(self,tol):
        self.tol=tol
        self.points.append(self.x0)
        self.points.append(self.xn)
        return self._AdaptiveQuad(self.x0,self.xn)

x0=0
n=4
xn=np.pi

Adaptive_value = Interval.AdaptiveQuad(1e-5)
```

3 Results

3.1 Test Function and True Value

The test function is:

$$f(x) = e^{3x}\cos(x)$$

The analytical antiderivative is:

$$F(x) = \frac{1}{10}(\sin x + 3\cos x)e^{3x}$$

Therefore, the true value of the integral is:

$$I_{\text{true}} = F(\pi) - F(0) = \frac{1}{10}(0 + 3(-1))e^{3\pi} - \frac{1}{10}(0 + 3)e^{0} = -\frac{3}{10}(e^{3\pi} + 1)$$

```
def I_f(x):
    return 1/10*(np.sin(x)+3*np.cos(x))*np.exp(3*x)

True_value = I_f(np.pi) - I_f(0)
```

3.2 Numerical Results

```
SimpsonRule4 = Interval.CompositeSimpsonRule(n)
Adaptive_value = Interval.AdaptiveQuad(1e-5)

True_value = I_f(np.pi) - I_f(0)
abs_err = [SimpsonRule4 - True_value, Adaptive_value - True_value]
rel_err = [(SimpsonRule4 - True_value) / True_value, (Adaptive_value - True_value) /
    True_value]
```

Output:

```
Simpson (4 intervals) integral value: -4105.733274669396
Adaptive Simpson integral value:      -3717.7943506329984
True value:                            -3717.794342375009
Absolute errors:
    Simpson:   -387.93893229438754
    Adaptive:  -8.257989520643605e-06
Relative errors:
    Simpson:   1.04e-01
    Adaptive:  2.22e-09
Number of adaptive sampling points: 257
```

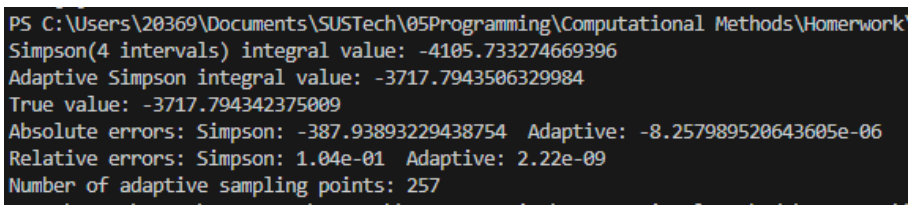

Figure 1 Console output showing numerical integration results and error analysis

3.3 Visualization

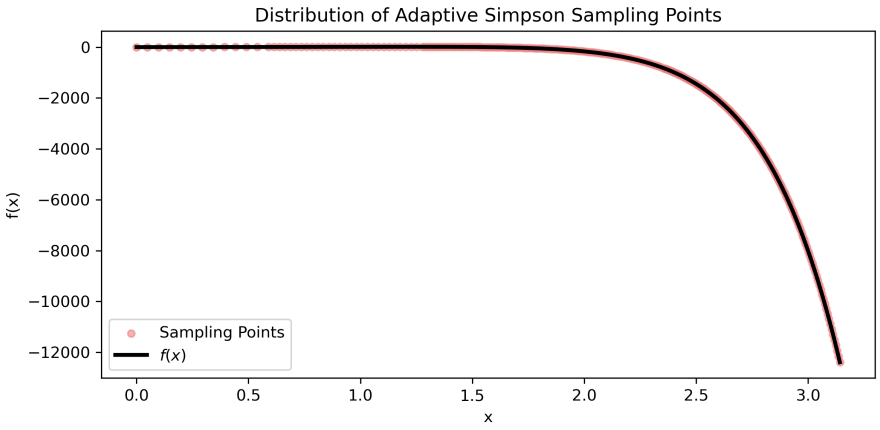

Figure 2 **Distribution of adaptive Simpson sampling points. Red dots indicate function evaluation locations. The density of points increases dramatically near $x = \pi$ where the function magnitude and curvature are largest.**

Figure 2 displays the distribution of all 257 adaptive sampling points overlaid on the function curve, consisting of 129 endpoints (interval boundaries) and 128 midpoints (used for Simpson's rule approximation). We observe that sampling points are sparse in the region $[0, 1]$ where the function is relatively smooth, but become increasingly dense as x approaches π, where both the magnitude $|f(x)|$ and the curvature increase rapidly.

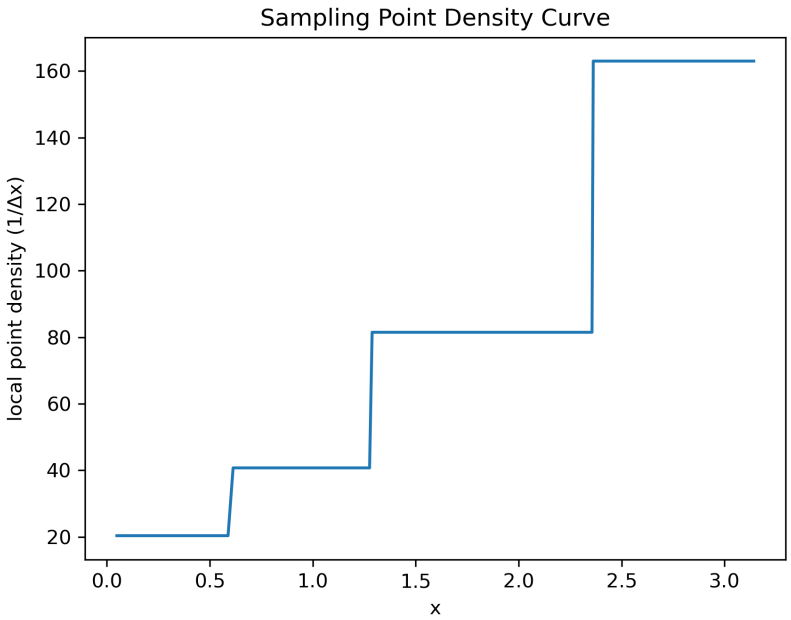

Figure 3 **Local sampling point density $\rho(x) = 1/\Delta x$ as a function of position. The sharp increase near $x = \pi$ demonstrates the adaptive refinement strategy.**

Figure 3 quantifies the local sampling density $\rho(x) = 1/\Delta x$, where Δx is the spacing between consecutive sampling points. The density remains relatively low for $x < 1.5$, then increases exponentially as $x \to \pi$.

This adaptive behavior is the key advantage over fixed-grid methods: computational effort is automatically allocated where it provides the greatest improvement in accuracy, resulting in optimal efficiency.

```
points = np.array(sorted(set(Interval.points)))
points_density=np.array([1/(points[ii+1]-points[ii]) for ii in range(points.shape[0]-1)])

# Plot sampling points
print(f"Number of adaptive sampling points: {points.shape[0]}")
plt.figure(figsize=(8,4))
plt.scatter(points, f(points), color='lightcoral', s=20, alpha=0.6, label='Sampling Points')
plt.plot(points, f(points), 'k',linewidth=2.5, label='$f(x)$')
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Distribution of Adaptive Simpson Sampling Points")
plt.legend()
plt.tight_layout()
plt.savefig('images/adaptive_simpson_sampling_points.png', dpi=300)
plt.show()

# Plot point density
plt.plot(points[1:],points_density)
plt.xlabel("x")
plt.ylabel("local point density (1/Δx)")
plt.title("Sampling Point Density Curve")
plt.savefig('images/Sampling_Point_Density_Curve.png',dpi=300)
plt.show()
```