Numerical methods
Computer project №1

Gleb Zhdanko

November 2024

General Information

The computations and visualizations were performed on a MacBook Pro (13-inch, 2020, Two Thunderbolt 3 Ports) equipped with a 1.4 GHz Quad-Core Intel Core i5 processor.

I used Python 3.12.4. The following Python libraries were utilized in the implementation:

- `numpy`: for numerical computations,
- `scipy`: for solving linear systems and other numerical methods,
- `sympy`: for symbolic manipulation and validation,
- `matplotlib`: for creating high-quality visualizations.

To enhance the clarity and aesthetic quality of the plots, the following font settings were applied:

```
font = {'weight': 'bold',
        'size': 15}
```

Exercise 1

Problem statement: Develop your own cubic spline interpolation code for an arbitrary distribution of grid points with parabolic and natural boundary conditions. Perform a study of the effect of the choice of interpolation points in the interval [0,1]. Use the following three grid distributions of $N + 1$ points each for cubic spline interpolation:

Equispaced: $x_k = \frac{k}{N}$

Chebyshev: $x_k = \frac{1}{2} - \frac{1}{2}\cos(\frac{\pi k}{N})$

arcsinepoints:$x_k = \frac{1}{2} + \frac{1}{\pi}\sin^{-1}(\frac{2k}{N} - 1)$,

where $k = 0, \ldots, N$ for all these distributions. For each of the following functions:

i. $\frac{1}{1+9x^2}$

ii. $|x - \frac{1}{2}$

iii. $\sqrt{1 - x^2}$

(a) Compute cubic spline on the 3 grid distributions for N = 10, 20, 40, and 80. For each function evaluate the cubic spline on the finer grid. Plot the actual function and its cubic spline for three different grid distributions in separate sub-windows.

(b) Estimate the maximum error the cubic spline makes on the entire interval [0, 1] by sampling the cubic spline at much finer set of grid points and plot these errors vs. N on semi-log scale in the forth sub-window.

(c) Plot these errors in log-log scale and estimate the order of convergence of the error as a function of N;

(d) Discuss the effectiveness of the choice for interpolation points.

For plotting and estimating error I used grid with 1000 points. Below are the plots with natural boundary conditions.

Solution Overview

Cubic Spline Implementation: To compute the cubic spline, I implemented a function that calculates the second derivatives (f'') for interpolation. For this, I constructed a tridiagonal matrix based on the system of equations:

$$\frac{h_i}{6}f''(x_{i-1}) + \frac{h_{i-1} + h_i}{3}f''(x_i) + \frac{h_i}{6}f''(x_{i+1}) = \frac{f(x_{i+1}) - f(x_i)}{h_i} - \frac{f(x_i) - f(x_{i-1})}{h_{i-1}},$$

where $h_i = x_{i+1} - x_i$. The second derivatives were calculated by solving this linear system.

Using these coefficients, the cubic spline for each interval $[x_i, x_{i+1}]$ was evaluated using:

$$f_i(x) = \frac{(x_{i+1}-x)^3}{6h_i} f''(x_i) + \frac{(x-x_i)^3}{6h_i} f''(x_{i+1}) + (x_{i+1}-x)\left(\frac{f(x_i)}{h_i} - \frac{h_i}{6} f''(x_i)\right) + (x-x_i)\left(\frac{f(x_{i+1})}{h_i} - \frac{h_i}{6} f''(x_{i+1})\right).$$

Error Calculation: To evaluate the error, I created a finer grid of 1000 points. The error was calculated as the maximum absolute difference between the true function and the interpolated values. The rate of convergence was obtained by plotting the errors in a log-log scale and using linear regression (`scipy.stats.linregress`) to determine the slope.

Results

Convergence Orders: The estimated convergence orders for both natural and parabolic boundary conditions are summarized below:

Natural Boundary Conditions:

- **For** $f_1(x) = \frac{1}{1+9x^2}$**:**
  - Equispaced: -2.0
  - Chebyshev: -4.1
  - Arcsine: -1.0
- **For** $f_2(x) = |x - \frac{1}{2}|$**:**
  - Equispaced: -1.0
  - Chebyshev: -0.99
  - Arcsine: -1.0
- **For** $f_3(x) = \sqrt{1-x^2}$**:**
  - Equispaced: -0.5
  - Chebyshev: -1.15
  - Arcsine: -0.26

Parabolic Boundary Conditions:

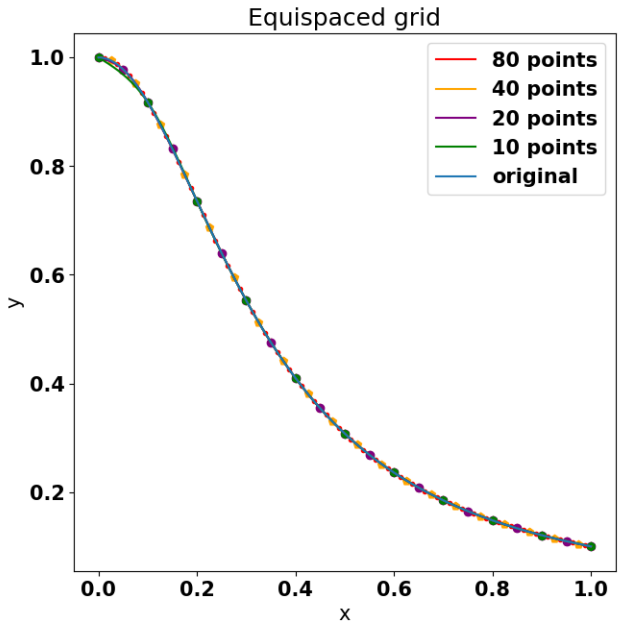
- **For** $f_1(x) = \frac{1}{1+9x^2}$**:**
  - Equispaced: -4.0
  - Chebyshev: -4.1
  - Arcsine: -1.7
- **For** $f_2(x) = |x - \frac{1}{2}|$**:**
  - Equispaced: -1.0
  - Chebyshev: -0.99
  - Arcsine: -1.0
- **For** $f_3(x) = \sqrt{1-x^2}$**:**
  - Equispaced: -0.5
  - Chebyshev: -1.2
  - Arcsine: -0.25

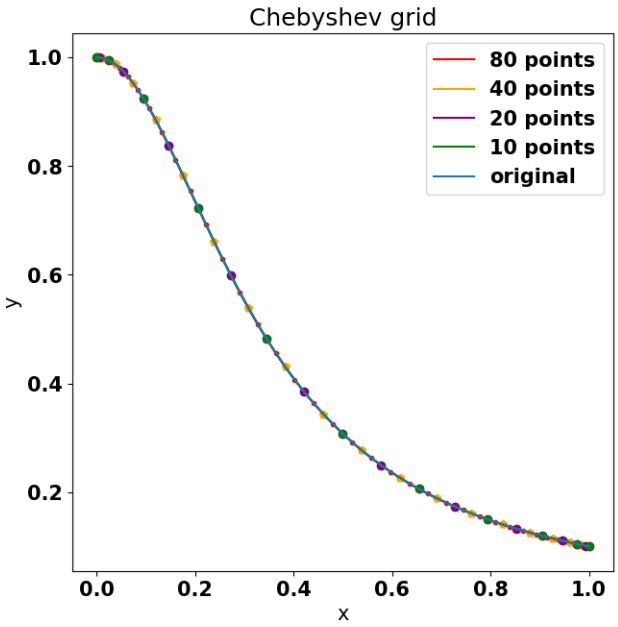
Discussion

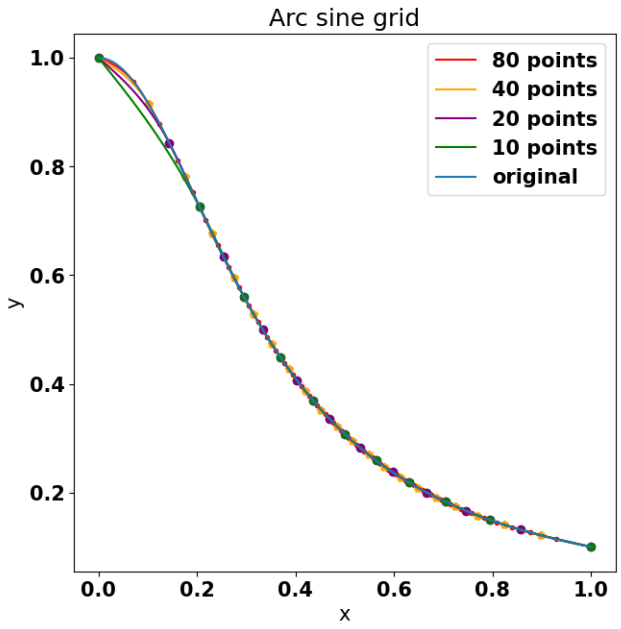
- **Choice of Grid:**
  - Chebyshev grids perform exceptionally well for functions with significant nonlinearity near the boundaries, such as $f_1(x)$.
  - Arcsine grids are more effective when the nonlinearity is concentrated in the center, such as $f_2(x)$.
  - Equispaced grids are the least effective, especially for highly nonlinear functions.
- **Boundary Conditions:** The choice of boundary conditions had minimal effect on the overall convergence for most cases, but parabolic conditions slightly improved performance for $f_1(x)$ on equispaced grids.
- **Anomalies:** In $f_3(x)$ with Chebyshev grids, the error curve exhibited some nonlinearity in log-log plots, which could be mitigated with a finer evaluation grid (e.g., 10,000 points).

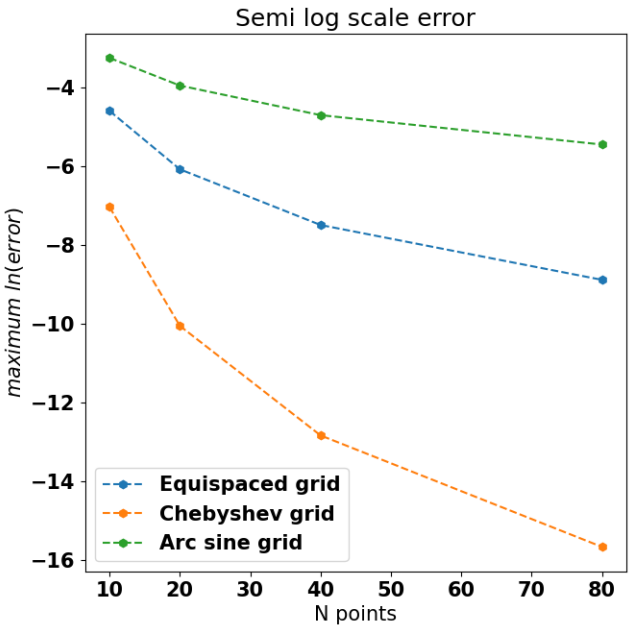
Zoomed-in plots highlight that Chebyshev grids outperform others near boundary nonlinearity, whereas arcsine grids excel at capturing central features.

Cubic spline for $\frac{1}{1+9x^2}$ with natural boundary condtions

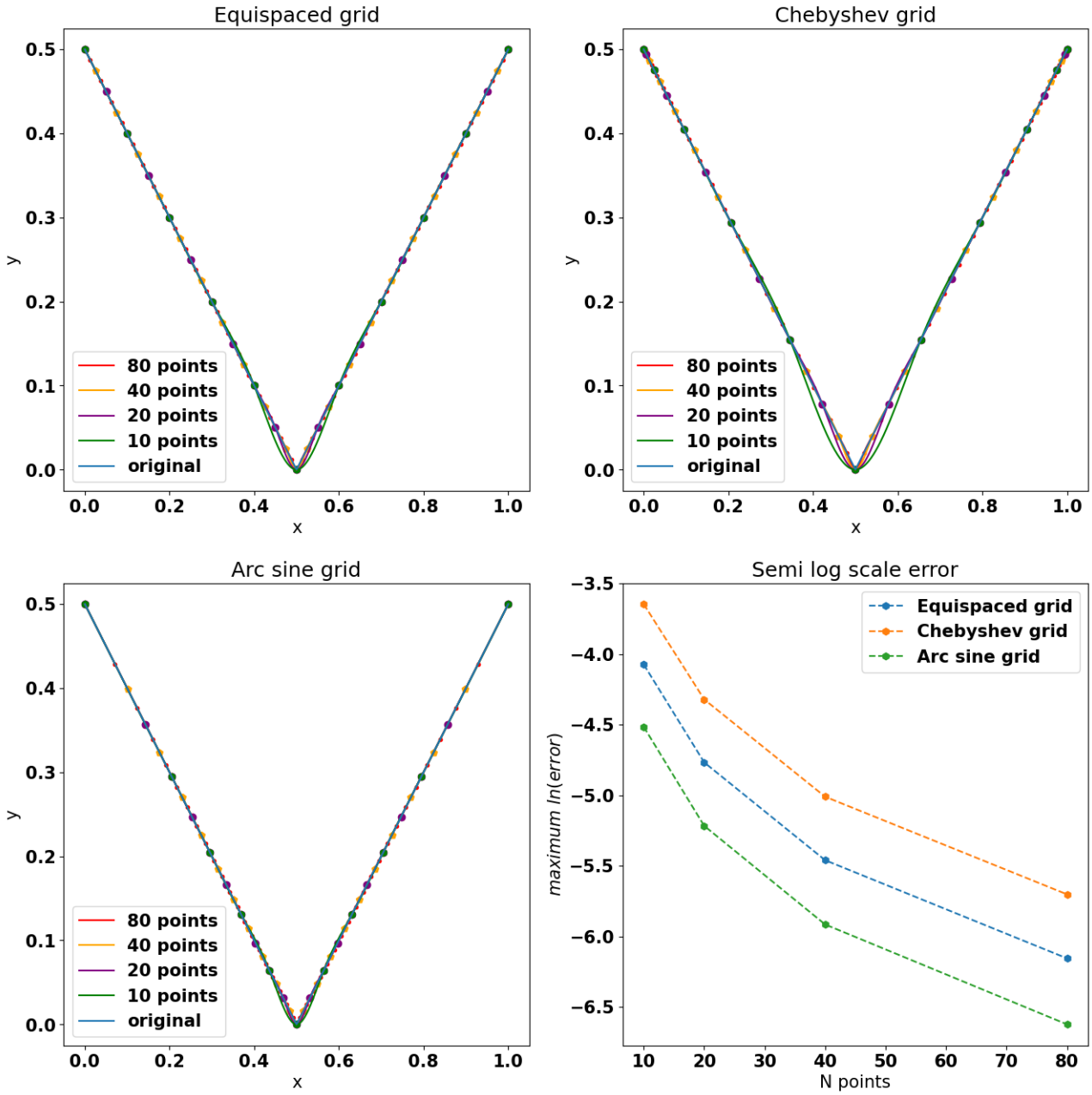




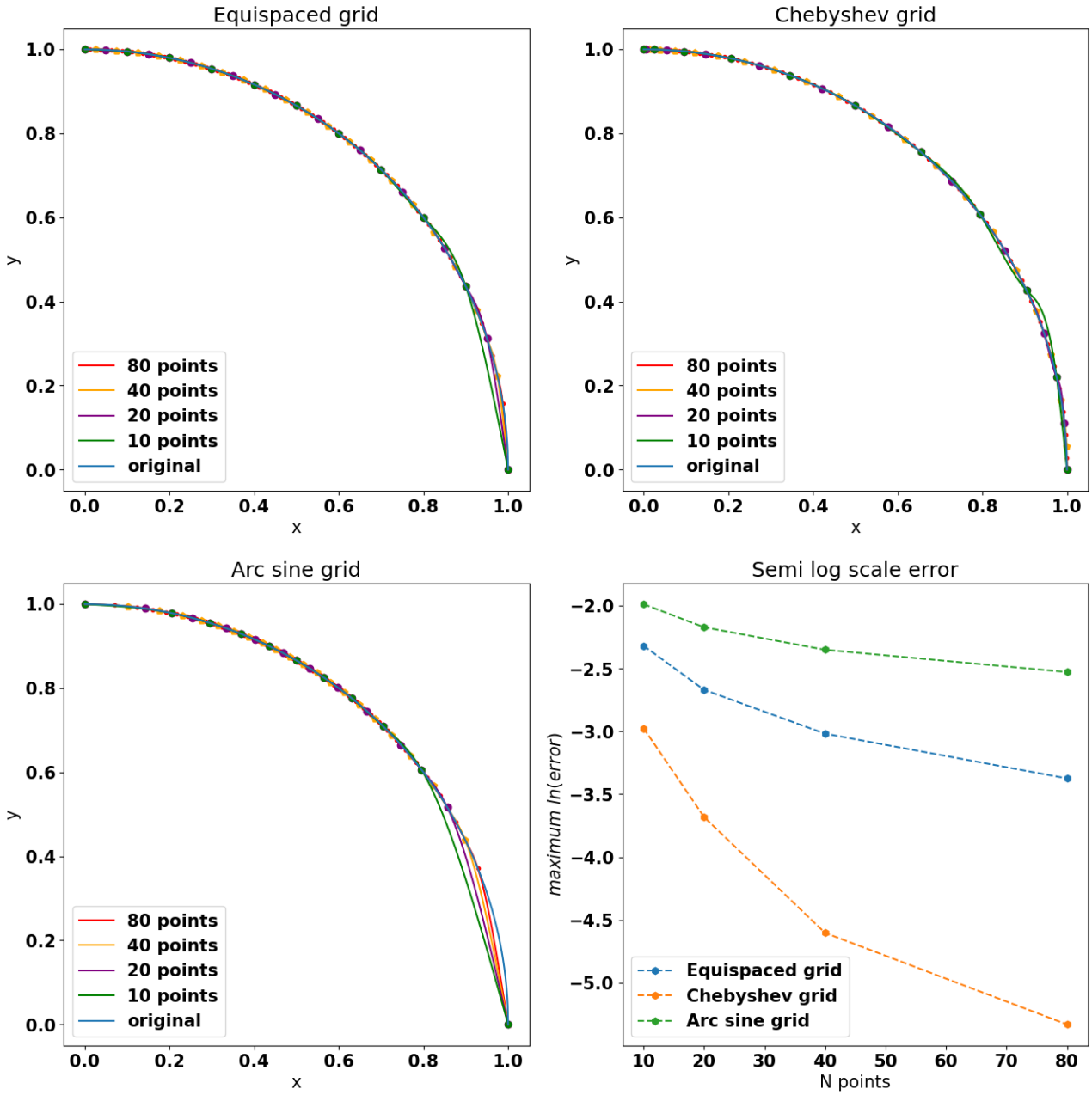




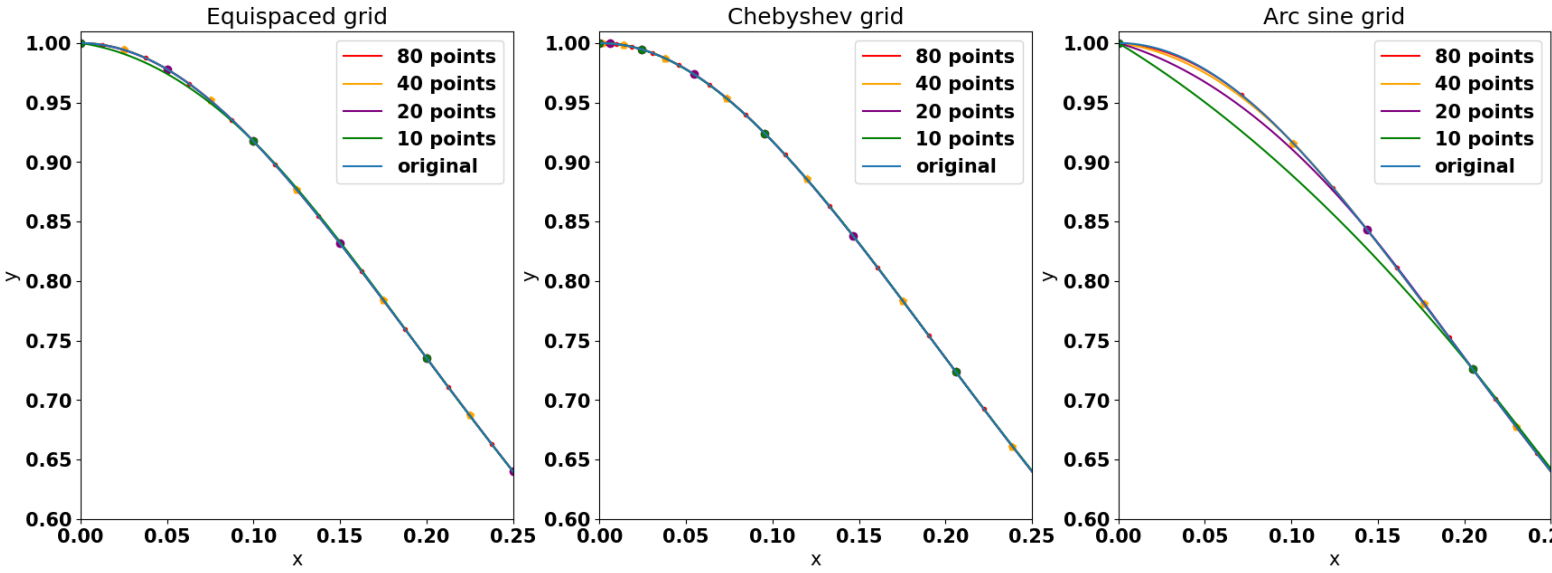
Cubic spline for $|x-\frac{1}{2}|$ with natural boundary condtions



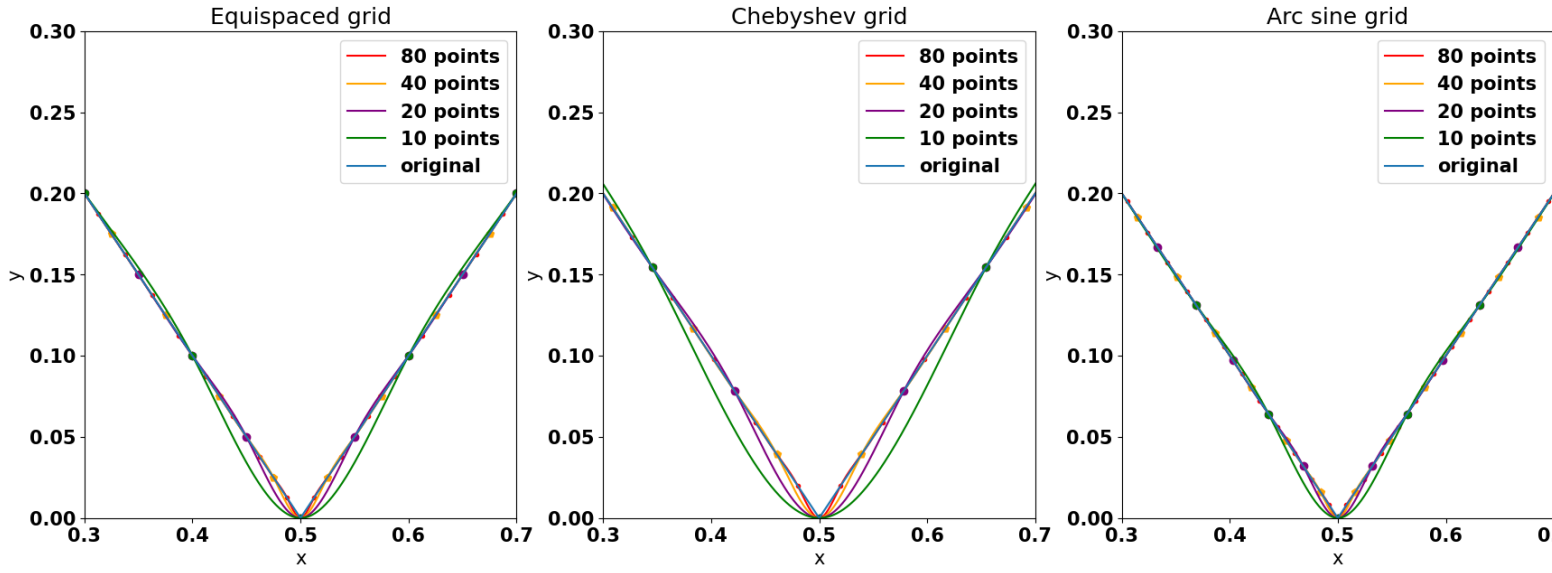
Cubic spline for $\sqrt{1-x^2}$ with natural boundary condtions



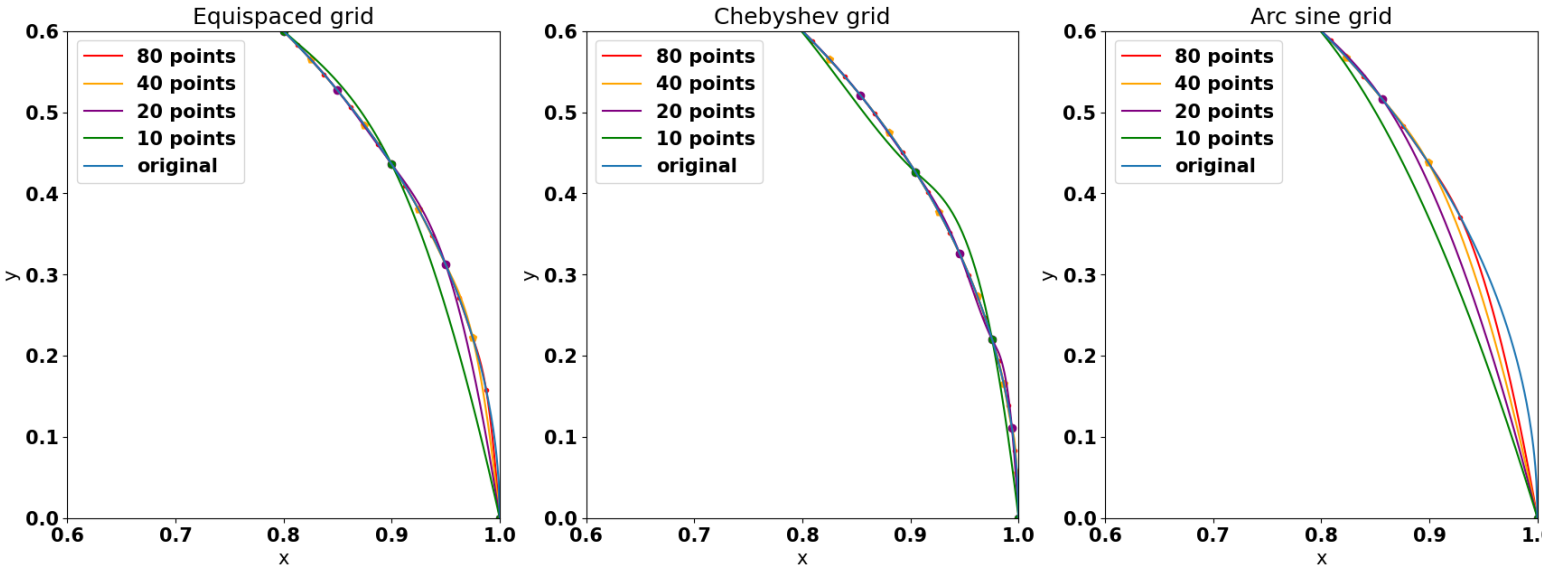
Zoomed Cubic spline for $\frac{1}{1+9x^2}$ with natural boundary condtions



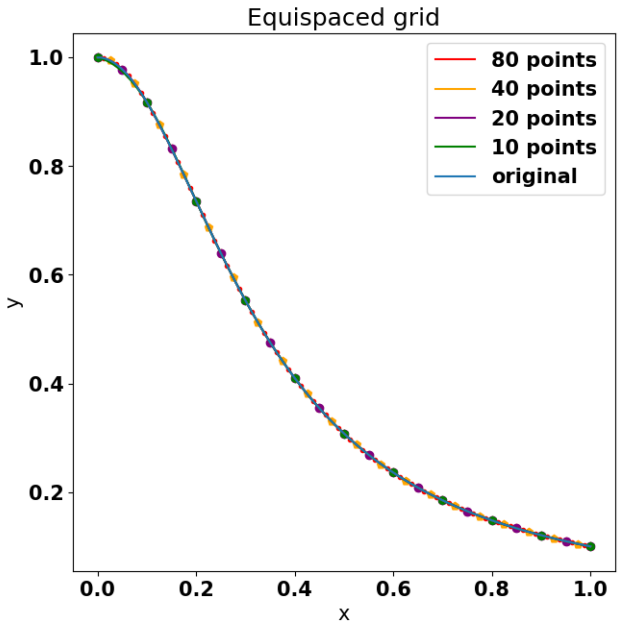
Zoomed Cubic spline for $|x-\frac{1}{2}|$ with natural boundary condtions

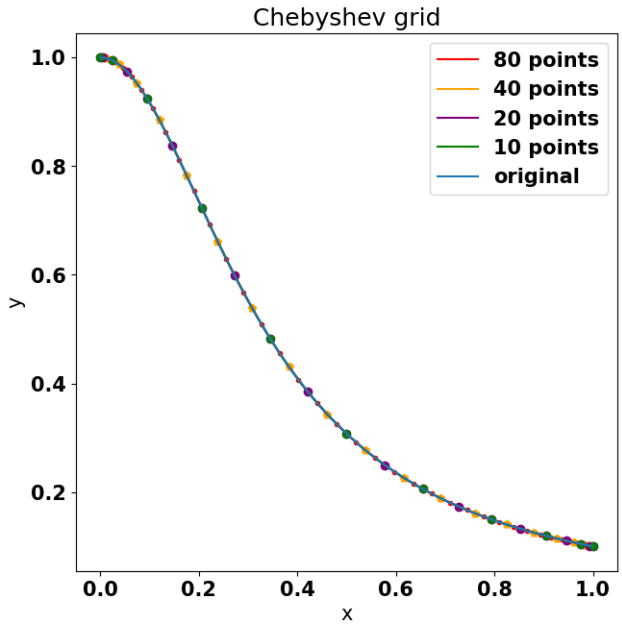


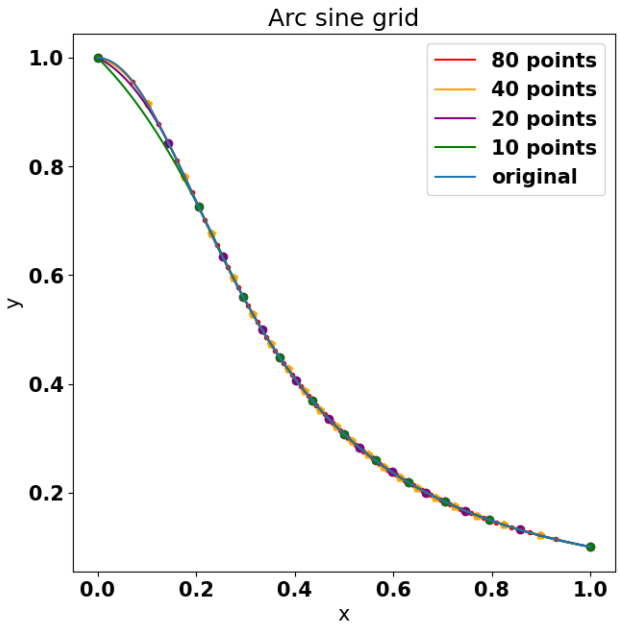
Zoomed Cubic spline for $\sqrt{1-x^2}$ with natural boundary condtions

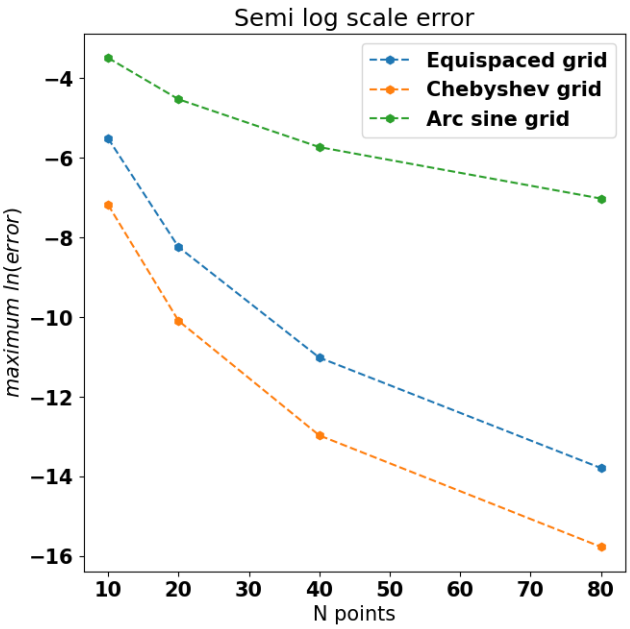


Cubic spline for $\frac{1}{1+9x^2}$ with parabolic boundary condtions

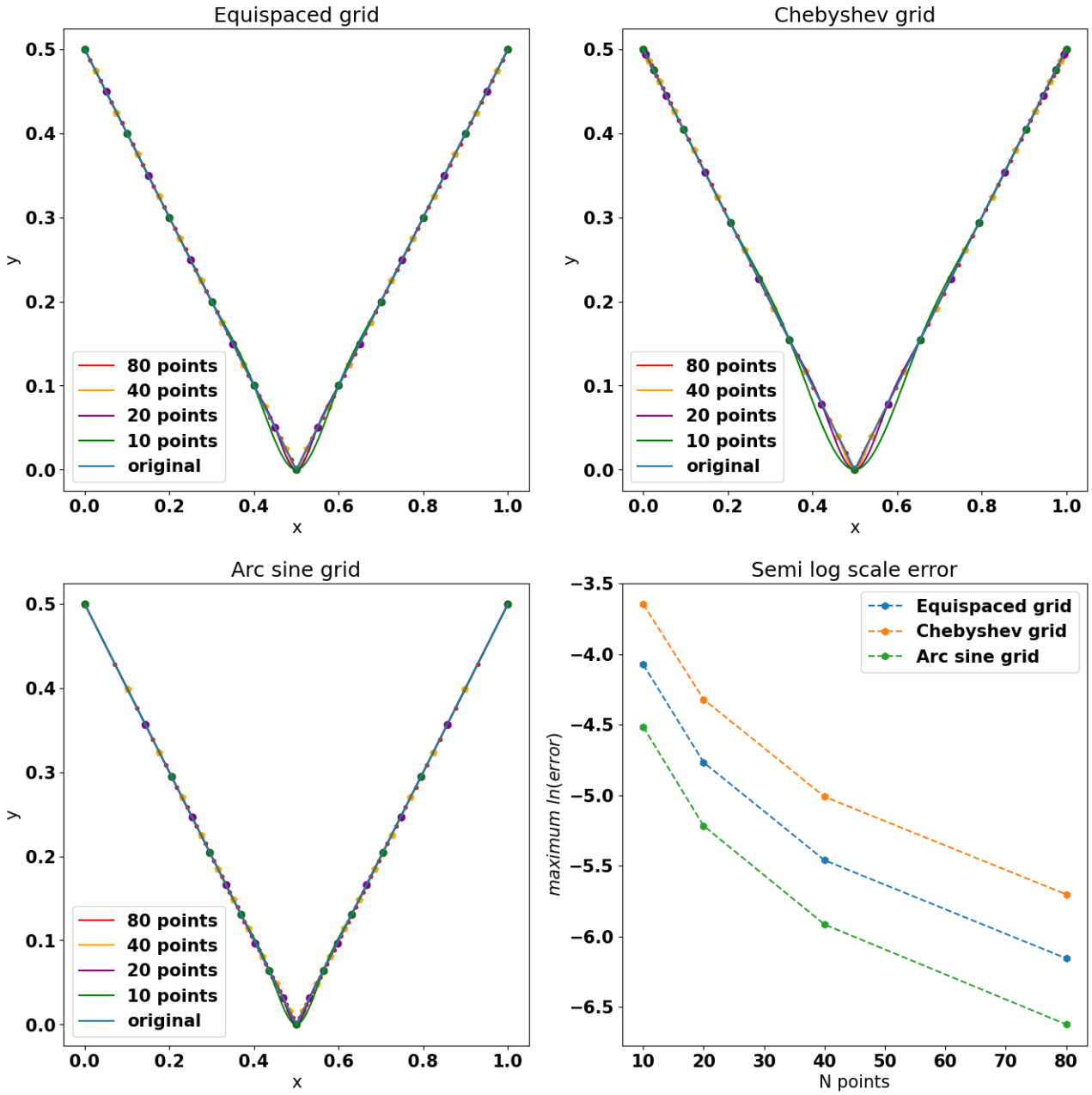




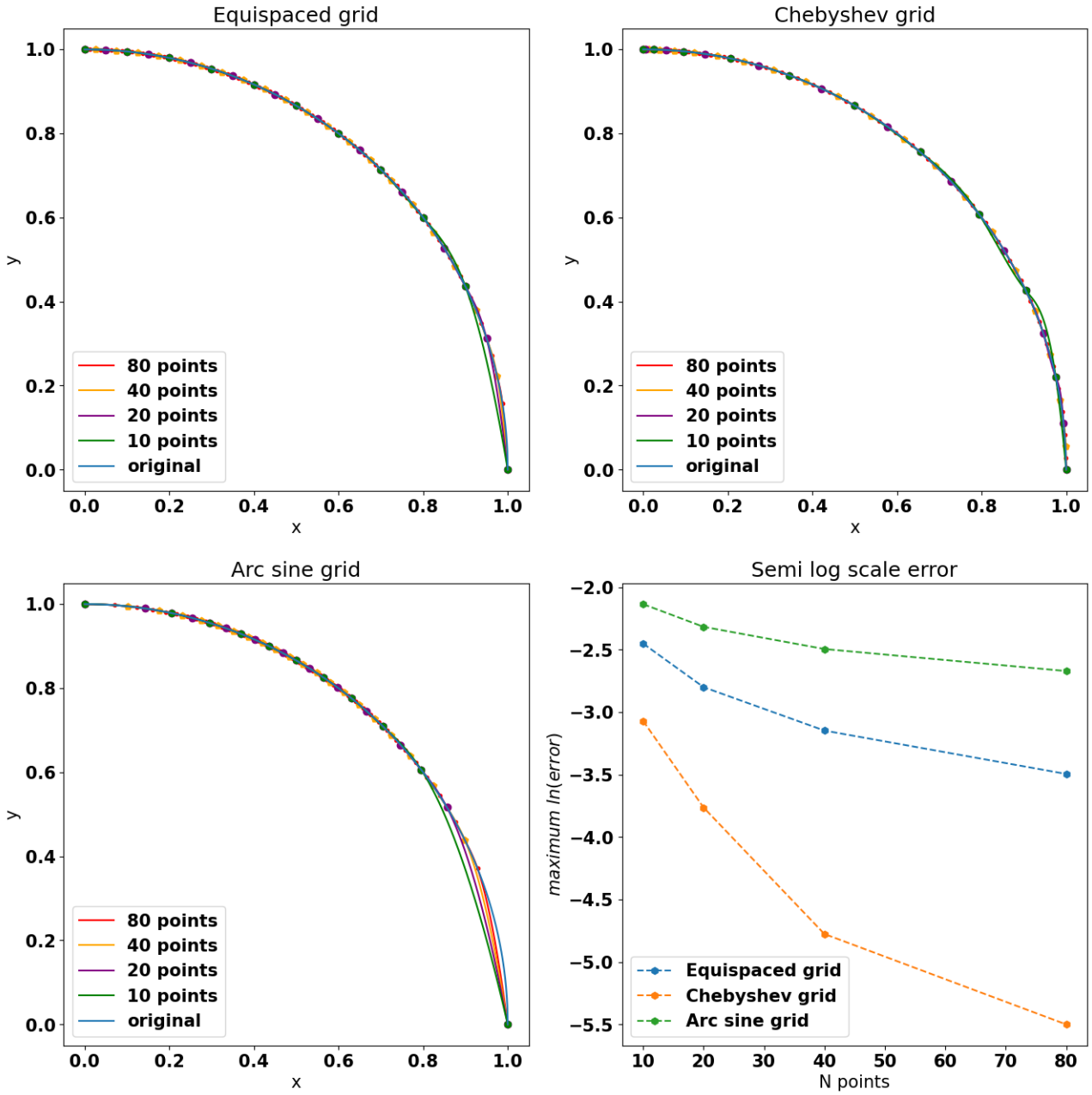




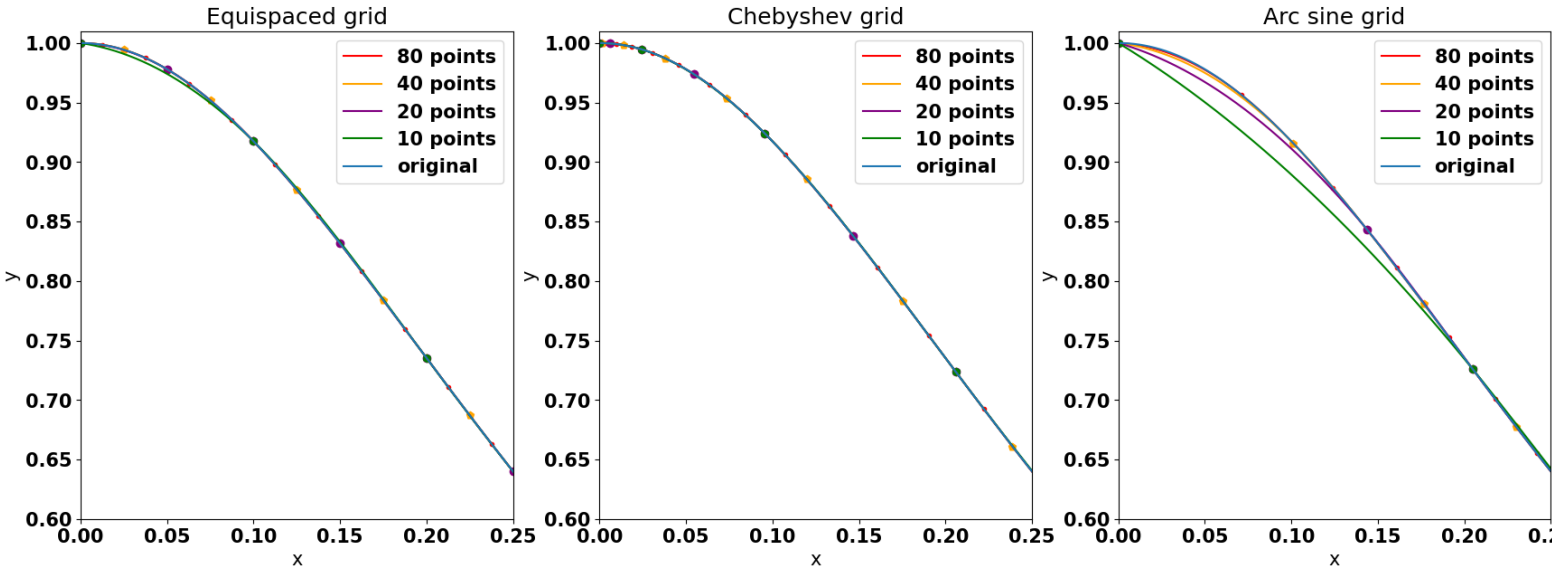
Cubic spline for $|x-\frac{1}{2}|$ with parabolic boundary condtions



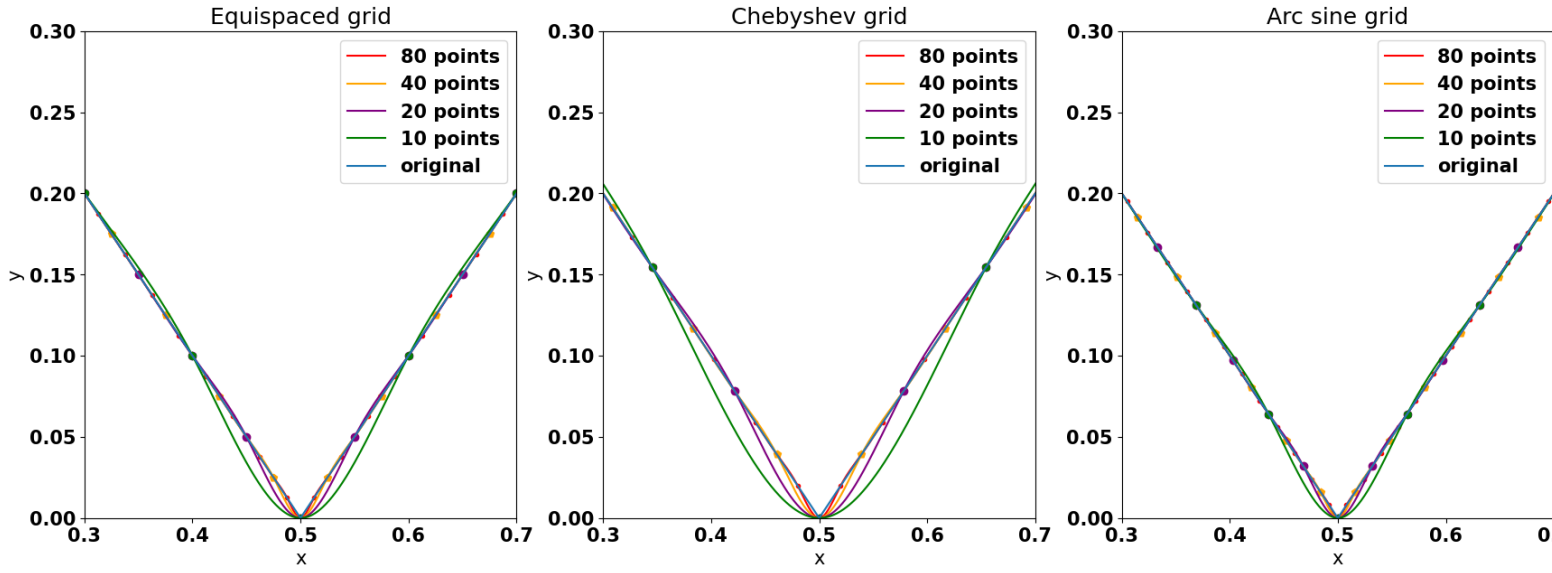
Cubic spline for $\sqrt{1-x^2}$ with parabolic boundary condtions



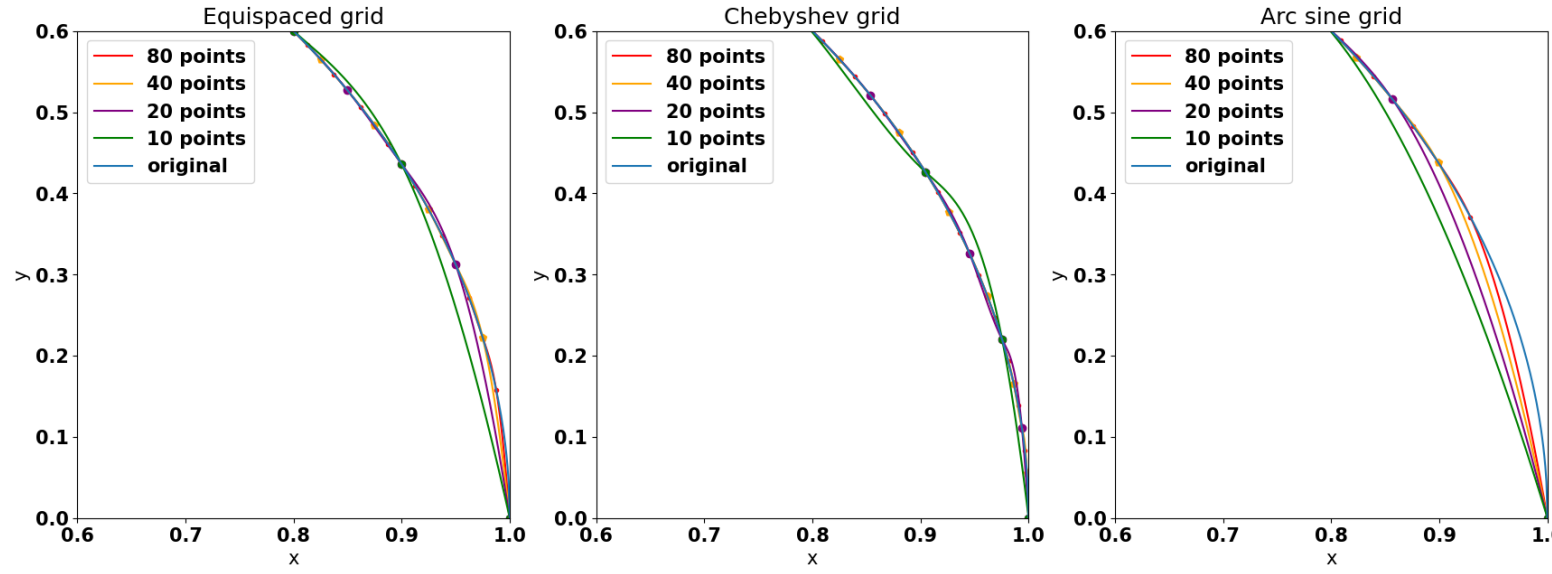
Zoomed Cubic spline for $\frac{1}{1+9x^2}$ with parabolic boundary condtions



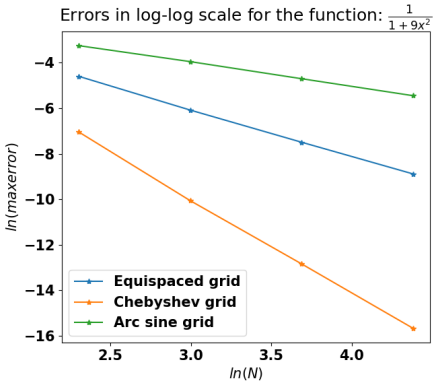
Zoomed Cubic spline for $|x-\frac{1}{2}|$ with parabolic boundary condtions

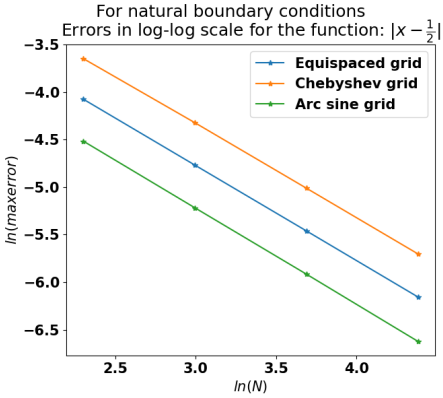


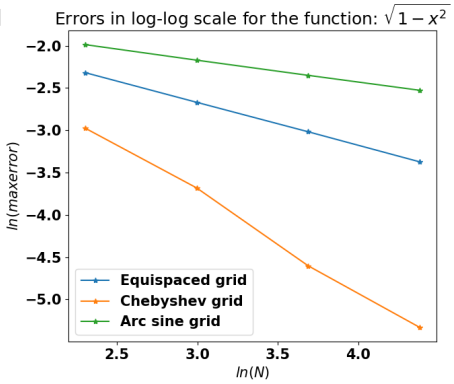
Zoomed Cubic spline for $\sqrt{1-x^2}$ with parabolic boundary condtions



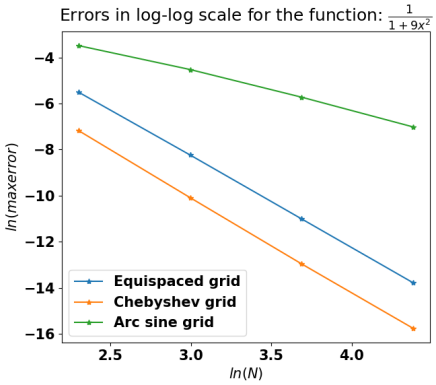
For natural boundary conditions

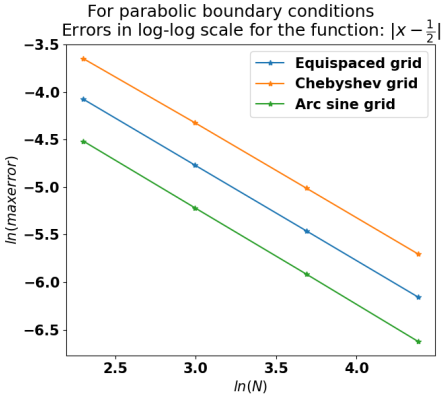


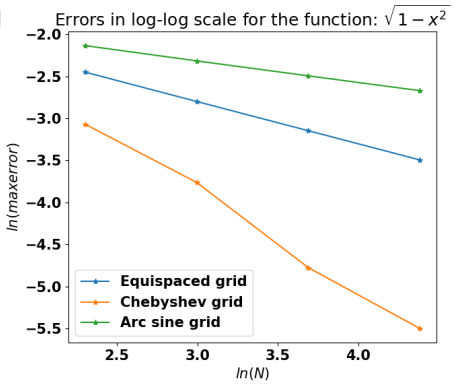




For parabolic boundary conditions







Exercise 2

Defining the difference operator δ^k by the relations:

$$\delta y_n = y_{n+1} - y_n, \delta^k y_n = \delta(\delta^{k-1} y_n)$$

a)Prove:

$$\delta(u_n v_n) = u_n \delta(v_n) + v_{n+1}\delta(u_n)$$

Proof:

$$\delta(u_n v_n) = u_{n+1}v_{n+1} - u_n v_n$$

$$u_n\delta(v_n) + v_{n+1}\delta(u_n) = u_n(v_{n+1} - v_n) + v_{n+1}(u_{n+1} - u_n) = u_{n+1}v_{n+1} - u_n v_n = \delta(u_n v_n)$$

b)Prove:

$$\sum_{n=0}^{N-1} u_n \delta v_n = u_N v_N - u_0 v_0 - \sum_{n=0}^{N-1} v_{n+1}\delta u_n$$

Proof:

$$u_n\delta(v_n) = \delta(u_n v_n) - v_{n+1}\delta(u_n)$$

$$\sum_{n=0}^{N-1} u_n \delta v_n = \sum_{n=0}^{N-1} \delta(u_n v_n) - v_{n+1}\delta(u_n) = \sum_{n=0}^{N-1}(u_{n+1}v_{n+1} - u_n v_n) - \sum_{n=0}^{N-1} v_{n+1}\delta(u_n) =$$

$$= u_N v_N - u_0 v_0 - \sum_{n=0}^{N-1} v_{n+1}\delta u_n$$

c) If $u_1 = u_N = v_1 = v_N$ then prove:

$$\sum_{n=1}^{N-1} u_n \delta^2 v_{n-1} = -\sum_{n=1}^{N-1} \delta u_n \delta v_n = \sum_{n=1}^{N-1} v_n \delta^2 u_{n-1}$$

Proof:

$$\sum_{n=1}^{N-1} u_n \delta^2 v_{n-1} = \sum_{n=1}^{N-1} u_n \delta(v_n - v_{n-1}) = \sum_{n=1}^{N-1} u_n \delta v_n - u_n \delta v_{n-1} =$$

$$[\text{using b case}] = u_N v_N - u_1 v_1 - \sum_{n=1}^{N-1} v_{n+1}\delta u_n - \sum_{n=1}^{N-1} u_n \delta v_{n-1} =$$

$$[\text{using b case}] = -\sum_{n=1}^{N-1} v_{n+1}\delta u_n + \sum_{n=1}^{N-1} \delta u_n v_n [=]$$

$$\text{Since } \delta u_n \delta v_n = (u_{n+1} - u_n)(v_{n+1} - v_n) = v_{n+1}\delta u_n - \delta u_n v_n \Rightarrow$$

$$[=] - \sum_{n=1}^{N-1} \delta u_n \delta v_n =$$

$$= [\text{due to symmetry}] = \sum_{n=1}^{N-1} v_n \delta^2 u_{n-1}$$

d) Let's $\delta u_n = u_{n+1} - u_{n-1}$. Then we have for case a:

$$\delta(u_n v_n) = u_{n+1}v_{n+1} - u_{n-1}v_{n-1} = u_{n-1}\delta(v_n) + v_{n+1}\delta(u_n)$$

Proof:

$$u_{n-1}\delta(v_n) + v_{n+1}\delta(u_n) = u_{n-1}(v_{n+1} - v_{n-1}) + v_{n+1}(u_{n+1} - u_{n-1}) = u_{n+1}v_{n+1} - u_{n-1}v_{n-1} = \delta(u_n v_n)$$

Case b will change in following way:

$$\sum_{n=1}^{N-1} u_{n-1}\delta v_n = u_N v_N + u_{N-1}v_{N-1} - u_0 v_0 - u_1 v_1 - \sum_{n=1}^{N-1} v_{n+1}\delta u_n$$

Proof:

$$u_{n-1}\delta(v_n) = \delta(u_n v_n) - v_{n+1}\delta(u_n)$$

$$\sum_{n=1}^{N-1} u_{n-1}\delta v_n = \sum_{n=1}^{N-1} \delta(u_n v_n) - v_{n+1}\delta(u_n) = \sum_{n=1}^{N-1} (u_{n+1}v_{n+1} - u_{n-1}v_{n-1}) - \sum_{n=1}^{N-1} v_{n+1}\delta(u_n) =$$

$$= u_N v_N + u_{N-1}v_{N-1} - u_0 v_0 - u_1 v_1 - \sum_{n=1}^{N-1} v_{n+1}\delta u_n$$

Discussion:

- $$\delta(u_n v_n) = u_n\delta(v_n) + v_{n+1}\delta(u_n)$$

  - resembles differentiation by parts

- $$\sum_{n=0}^{N-1} u_n\delta v_n = u_N v_N - u_0 v_0 - \sum_{n=0}^{N-1} v_{n+1}\delta u_n$$

  - resembles integration by parts

- $$\sum_{n=1}^{N-1} u_n\delta^2 v_{n-1} = -\sum_{n=1}^{N-1} \delta u_n \delta v_n = \sum_{n=1}^{N-1} v_n\delta^2 u_{n-1}$$

  I don't see any resemblance with differentiation or integration. Nonetheless, this formula showing symmetry.

- in case d) I tried to do the similar differentiation by parts and integration by parts, that's why i got such formulas. And they indeed seems as real differentiation and integration.

Exercise 3.

Problem statement:

Find the most accurate central difference finite difference formulas and the corresponding leading error terms for

(a) the first derivative, given the values of the function and its derivative, and

(b) the second derivative, given the values of the function and its second derivative

at $x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}$. Assume the points uniformly spaced.

Solution Overview

The solution was derived using Hermitian Methods and Padé Approximations. Instead of employing traditional Taylor series expansions, I used operator forms for greater flexibility and precision. Symbolic calculations were performed using the `sympy` library in Python.

Initially, I disregarded the D operator when calculating the coefficients because Python treated it as an independent parameter, which made elimination challenging. To simplify the process, I omitted D while determining coefficients but later reintroduced it to compute the leading error term and its order. This approach ensured that the symbolic calculations incorporated all necessary terms.

During the expansion of the exponential series on the right-hand side, I incremented the order by one to equate the powers of h with those on the left-hand side. The coefficients were obtained by forming polynomials as the difference between the left- and right-hand sides. By extracting the coefficients of powers of h and solving the resulting linear equations, I determined the required coefficients.

An adjustment was made by subtracting 1 from the coefficient of h (or h^2 for the second derivative) to ensure the derived formula had the form $f'(x_i) = O(h^8)$ (or analogous for the second derivative).

Finally, the leading error term was calculated from the derived formula.

The Python implementation is attached, and the results are summarized below.

a)

$$f'(x_i) = \left(-\frac{20}{27}f(x_{i+1}) - \frac{25}{216}f(x_{i+2}) + \frac{20}{27}f(x_{i-1}) + \frac{25}{216}f(x_{i-2})\right)/h^2 +$$
$$+\frac{4}{9}f'(x_{i+1}) + \frac{1}{36}f'(x_{i+2}) + \frac{4}{9}f'(x_{i-1}) + \frac{1}{36}f'(x_{i-2}) + \frac{h^8 f^{(9)}}{22680} + O(h^9)$$

b)

$$f''(x_i) = \left(\frac{265}{131}f(x_i) - \frac{320}{393}f(x_{i+1}) - \frac{155}{786}f(x_{i+2}) - \frac{320}{393}f(x_{i-1}) - \frac{155}{786}f(x_{i-2})\right)/h^2 +$$
$$+\frac{344}{1179}f''(x_{i+1}) + \frac{23}{2358}f''(x_{i+2}) + \frac{344}{1179}f''(x_{i-1}) + \frac{23}{2358}f''(x_{i-2}) + \frac{79h^8 f^{(10)}}{2971080} + O(h^9)$$

The derived formulas for the first and second derivatives achieve an error order of $O(h^8)$, making them suitable for applications requiring high precision.

Challenges and Limitations

A notable challenge was handling the symbolic elimination of the D operator in Python. The omission and subsequent reintroduction of D added complexity to the derivation process.

Significance of Results

The derived formulas demonstrate the potential of combining operator methods with symbolic computation for deriving highly accurate finite difference schemes. However, the approach relies on the symbolic capabilities of tools like Python's `sympy`.

Exercise 4.
Problem statement:
The function

$$f(x) = x^{\alpha}(1.2 - x)(1 - e^{\beta(x-1)})$$

is convex for all α, β, and x in $[0, 1]$. The parameters α and β allow one to make it very easy or very difficult to integrate. For

(a) $\alpha = 2,\ \beta = 0.2$,

(b) $\alpha = 0.1, \beta = 20$

integrate the equation using the trapezoidal rule, Simpson's rule, and an adaptive quadrature algorithm based on Trapezoidal and Simpson rules. Compare the number of function evaluations required to achieve various tolerances for each algorithm. Also compare the rates of convergence of the four integration methods.

Solution Overview

Numerical Integration Implementation: The trapezoidal and Simpson's integration rules were implemented for different number of points N. For the trapezoidal and Simpson's rule, the number of function evaluations is N

Choice of Points: The following values of N were used for the calculations:

$$N = [11, 21, 31, 41, 51, 61, 71, 101, 1001, 10001, 100001, 1000001].$$

Error Estimation: The true value of the integral was taken from Wolfram:

- For case (a), the exact value is 0.00954997.
- For case (b), the integral was approximated yielding 0.602298070979271.

The error was calculated as the absolute difference between the numerical result and the exact or approximate integral value.

Convergence Calculation: The error was plotted against the number of points in a log-log scale to determine the convergence rates. Linear regression using `scipy.stats.linregress` was applied to estimate the slope of the error curve, representing the convergence order. For consistency, I selected linear regions of the log-log plots for the regression.

Results

Case (a):

- For the trapezoidal rule, the convergence rate was -2.055.
- For Simpson's rule, the convergence rate was -4.468.

Interestingly, after $N = 10001$, the error plateaued. This is expected, as the integral in case (a) can be computed exactly.

Case (b):

- For the trapezoidal rule, the convergence rate was -1.109.
- For Simpson's rule, the convergence rate was -1.106.

Here, the error continued to decrease as N increased. However, convergence was slower compared to case (a), likely due to the function's increased complexity and the difficulty of approximating it on an equispaced grid.

Adaptive Quadrature: For the adaptive quadrature, recursive algorithms were implemented based on the trapezoidal and Simpson's rules. The following tolerances were used:

$$\text{Tolerances: } [10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}].$$

Initial number of points were chosen to be 10. The number of function evaluations was recorded using a counter function variable which is increased by one every time algorithm call it. After each integration I set this parameter back to 0. The error and number of evaluations were plotted in log-log scale, and the slopes of the plots were estimated to determine the convergence rates. Similarly to previous case, I have selected linear regions of the log-log plots for the regression.

Convergence Rates for Adaptive Quadrature:

- **Case (a):**
  - Trapezoidal quadrature: -1.928.
  - Simpson quadrature: -1.82.
- **Case (b):**
  - Trapezoidal quadrature: -1.99.
  - Simpson quadrature: -2.

Discussion

Comparison of Methods:

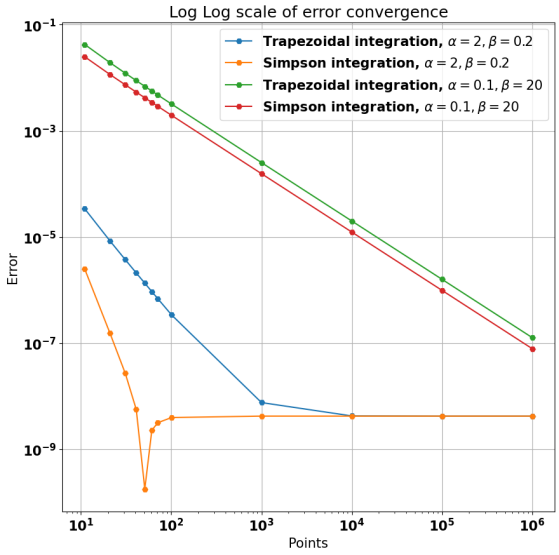
- In case (a), the adaptive quadrature methods were slightly less efficient than standard integration for this simple case, possibly due to additional overhead.
- In case (b), adaptive quadrature showed a significant advantage. Both quadrature methods achieved a convergence order close to -2, outperforming standard integration on an equispaced grid where convergence is around -1.1. Which means that quadrature methods outperform regular integration by approximately **an order of 10 in the number of steps required** for similar accuracy.
- Simpson's quadrature consistently requires nearly **half as many steps** as trapezoidal quadrature to achieve the same tolerance, demonstrating its superior efficiency.

Observations and Anomalies:

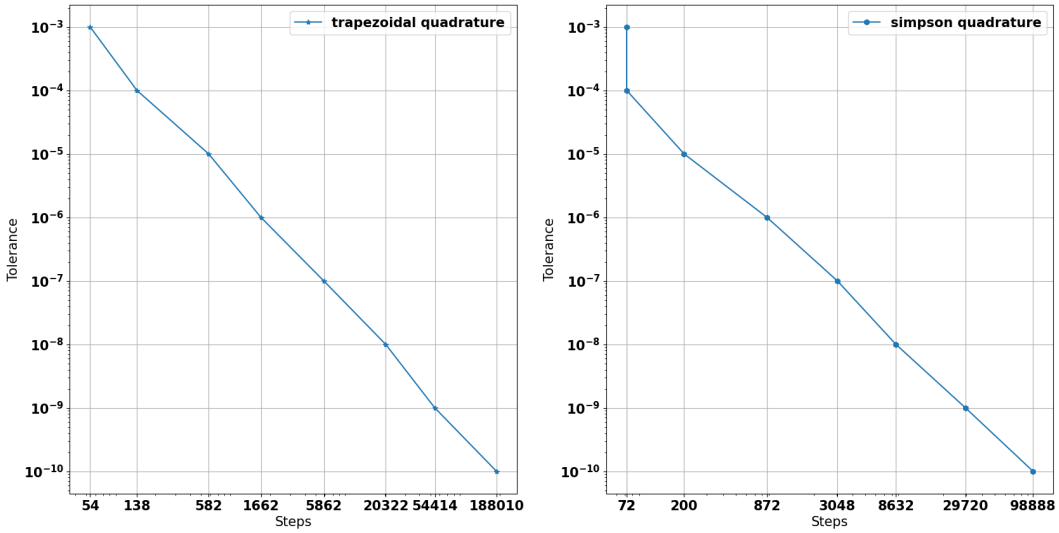
- In case (a) for standard integration the convergence rates for both methods matched theoretical expectations.
- For Simpson's rule in case (a), the convergence rate initially appeared to deviate from the theoretical value of -4. Upon refining the number of points (initially was taken $N = [11, 101, \ldots, 1000001]$), the issue was resolved. A dramatic jump in error was observed at $N = 51$, after which the error plateaued.
- In case (b), the slower convergence of standard integration methods is attributed to the function's complexity, which challenges equispaced grids.

Conclusion: The adaptive quadrature methods are more robust for integrating complex functions like in case (b), while standard methods work well for simpler cases like (a).

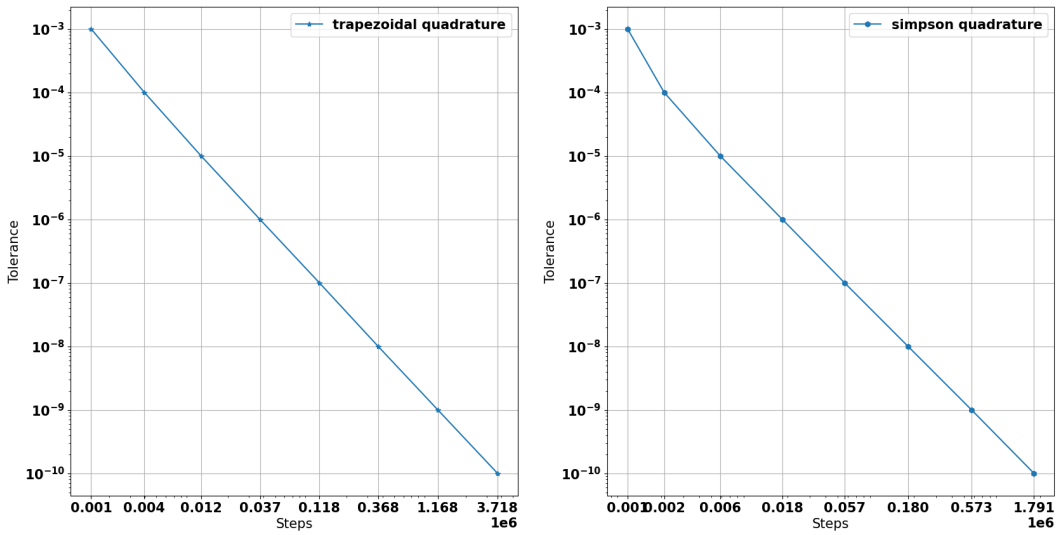
Below are the plots:



Convergence in log log scale for $\alpha = 2, \beta = 0.2$



Convergence in log log scale for $\alpha = 0.1, \beta = 20$



Exercise 5.

Problem statement: Given the forward-difference formula

$$f'(x_0) = \frac{1}{h}[f(x_0 + h) - f(x_0)] - \frac{h}{2}f''(x_0) - \frac{h^2}{6}f'''(x_0) + O(h^3)$$

use Richardson extrapolation to derive a difference operator for $f'(x_0)$, whose leading error term is $O(h^3)$.

Overview of solution

We aim to systematically cancel lower-order error terms $O(h)$ and $O(h^2)$ while constructing a new operator. The substitution: $\frac{f''(x_0)}{2} = c$, $\frac{f'''(x_0)}{6} = b$ simplify error term handling.

Solution:

Let's define operator D_h:

$$D_h f = \frac{f(x_0 + h) - f(x_0)}{h} = f'(x_0) + ch + bh^2 + O(h^3)$$

To reduce the error terms, we calculate D_{2h}, D_{3h}, D_{4h}.

$$D_{2h} f = \frac{f(x_0 + 2h) - f(x_0)}{2h} = f'(x_0) + 2ch + 4bh^2 + O(h^3)$$

$$\tilde{D}(f) = (2D_h - D_{2h})f = f'(x_0) - 2bh^2 + O(h^3)$$

$$D_{3h} f = \frac{f(x_0 + 3h) - f(x_0)}{3h} = f'(x_0) + 3ch + 9bh^2 + O(h^3)$$

$$D_{4h} f = \frac{f(x_0 + 4h) - f(x_0)}{4h} = f'(x_0) + 4ch + 16bh^2 + O(h^3)$$

$$\tilde{\tilde{D}} f = (4D_{3h} - 3D_{4h})f = f'(x_0) - 12bh^2 + O(h^3)$$

$$D_{new} f = \frac{6\tilde{D} - \tilde{\tilde{D}}}{5} = f'(x_0) + O(h^3)$$

$$D_{new} f = \frac{1}{5h}\left(12f(x_0 + h) - 3f(x_0 + 2h) - \frac{4}{3}f(x_0 + 3h) + \frac{3}{4}f(x_0 + 4h) - \frac{101}{12}f(x_0)\right)$$

Discussion:

- The grid points h, $2h$, $3h$, and $4h$ were initially selected for constructing the operators. However, for better performance and computational efficiency, it is more practical to use a finer grid consisting of h, $2h$, and $3h$, and then construct the second operator using D_{2h} and D_{3h}.
- **Significance:** The $O(h^3)$ accuracy achieved by the derived operator significantly reduces truncation error compared to traditional finite-difference formulas with $O(h)$ or $O(h^2)$ accuracy, making it highly suitable for precision-critical applications.
- **Challenges:** The derivation required careful manual verification of symbolic calculations to ensure the correctness of coefficients and cancellation of error terms.