Test NUM

Contents

1 Exact Analytical Value

The exact value of the integral is known and serves as a benchmark for evaluating the numerical results:

$$I_{\text{exact}} = \ln(2)$$

In R, this is computed using:

```r
I_analyticke <- log(2)
```

Listing 1: Exact value of the integral

2 Midpoint Rule Method

The midpoint rule approximates the integral of a function over an interval $[a, b]$ by dividing it into n equally sized subintervals of width $h = (b - a)/n$. For each subinterval, the function value at the midpoint is multiplied by the width:

$$I_N = \sum_{i=0}^{n-1} h \cdot f\left(a + ih + \frac{h}{2}\right)$$

This method is particularly effective for smooth, continuous functions and is simple to implement. In R:

```r
f <- function(x) {
  1 / x
}

midpoint_integral <- function(f, a, b, n) {
  h <- (b - a) / n
  I <- 0

  for (i in 0:(n - 1)) {
    xi <- a + i * h + h / 2
    Si <- h * f(xi)
    I <- I + Si
  }
  return(I)
}
```

Listing 2: Midpoint rule function in R

3 Error Computation

To evaluate how the midpoint approximation converges, we compute the error for increasing values of $n = 2^i$, with i ranging from 1 to 20. The error is computed as:

$$y(i) = I_N(i) - I_{\text{exact}}$$

This shows how the numerical result deviates from the true value.

```
i_val <- 1:20
y_val <- numeric(length(i_val))

for (j in seq_along(i_val)) {
  i <- i_val[j]
  n <- 2^i
  I_numericke <- midpoint_integral(f, 1, 2, n)
  y_val[j] <- I_numericke - I_analyticke
  if (j>1 && abs(y_val[j]) > abs(y_val[j-1])) {
    print(j)
  }
}
```

Listing 3: Compute midpoint error for i

4 Visualizing Error Behavior

The error is plotted as a function of i. The error should generally decrease with increasing n:

```
plot(i_val, y_val, type = "b", pch = 19, col = "blue",
     xlab = "i", ylab = expression(y(i) == I[N](i) - I[A
       ]))
abline(h = 0, col = "red", lty = 2)
```

Listing 4: Plot the numerical error

5 Least Squares Approximation of the Error Function

We want to approximate the numerical error $y(i)$ using a linear combination of basis functions:

$$y(i) \approx c_0 \cdot \varphi_0(i) + c_1 \cdot \varphi_1(i) + c_2 \cdot \varphi_2(i) + c_3 \cdot \varphi_3(i)$$

where:

- $\varphi_0(i) = 1$
- $\varphi_1(i) = \frac{1}{2^i}$ (exponential decay)
- $\varphi_2(i) = \frac{1}{i}$ (harmonic decay)
- $\varphi_3(i) = \frac{1}{i^2}$ (polynomial decay)

We construct a design matrix A where $A[i, j] = \varphi_j(i)$, and solve the normal equations:

$$(A^T A)\vec{c} = A^T \vec{y}$$

```r
least_squares_approx <- function(x, y, basis_functions) {
  if(length(x) != length(y)) stop("x and y must have the same length")

  m <- length(x)
  n <- length(basis_functions)

  A <- matrix(0, nrow = m, ncol = n)
  for (j in 1:n) {
    A[, j] <- basis_functions[[j]](x)
  }
  coeff <- solve(t(A) %*% A, t(A) %*% y)
  return(as.vector(coeff))
}
```

Listing 5: Least squares approximation function

6 Computing and Interpreting the Coefficients

The coefficients represent the best-fit parameters for the chosen basis functions to approximate the error behavior.

```
poly_basis <- list(
  function(x) 1,
  function(x) 1 / (2^x),
  function(x) 1 / x,
  function(x) 1 / (x^2)
)

coeff_poly <- least_squares_approx(i_val, y_val, poly_
   basis)
print(coeff_poly)
```

Listing 6: Define basis functions and compute coefficients

7 Evaluation and Plotting of the Approximation

The approximated function is evaluated at each point i:

```
aprox_y_val <- sapply(i_val, function(xi) {
  sum(mapply(function(c, phi) c * phi(xi), coeff_poly,
     poly_basis))
})

lines(i_val, aprox_y_val, col = "darkgreen", lwd = 2)
```

Listing 7: Evaluate and plot the approximation