ECSE-543
Numerical Methods in EE
Assignment #3

Andrei Purcarus, 260631911, *McGill University*

Code Listings and Unit Testing

The source code used for this assignment is listed in the appendices. In order to save space, we did not include the unit tests. For the full code, see the GitHub repository.

Section A defines the main function, which contains the core code that produces answers to the questions in this assignment. Sections B and C define a polynomial class. Sections D and E define functions that perform interpolation on data. Sections F and G define nonlinear equation solvers. Section H defines a matrix library. Section I defines functions that perform Cholesky decomposition using banded and non-banded methods. Section J defines a generic solver for systems of equations that have a positive-definite coefficient matrix. Finally, Sections K and L define functions that perform numerical integration.

Question 1

(a)

We started by performing interpolation on the first 6 points in the B-H data for M19 steel using a Lagrange polynomial on the entire domain. To do this, we used the Lagrange interpolation function defined in Sections D and E. The result is shown in Figure 1. From this figure, we can see that the interpolated curve matches the data well over this range, since it is monotonically increasing and does not have any significant "wiggles".

(b)

Next, we performed the same Lagrange polynomial interpolation on the 6 points at $B = 0.0\,\mathrm{T}$, $B = 1.3\,\mathrm{T}$, $B = 1.4\,\mathrm{T}$, $B = 1.7\,\mathrm{T}$, $B = 1.8\,\mathrm{T}$, and $B = 1.9\,\mathrm{T}$. The results are shown in Figure 2. From this figure, we can see that the result matches the data well for $B \geq 1.3\,\mathrm{T}$, but has a large deviation between $B = 0.0\,\mathrm{T}$ and $B = 1.3\,\mathrm{T}$ that does not seem plausible for the given data. This is likely to have occurred due to the lack of data in the latter range, as well as the high order polynomial we used to perform the interpolation.

(c)

As an alternative to full-domain Lagrange interpolation, we also used cubic Hermite interpolation on each of the 5 sub-domains between the 6 points used in the previous section. To do this, we used the Hermite interpolation function shown in Sections D and E. However, to do this, we had to

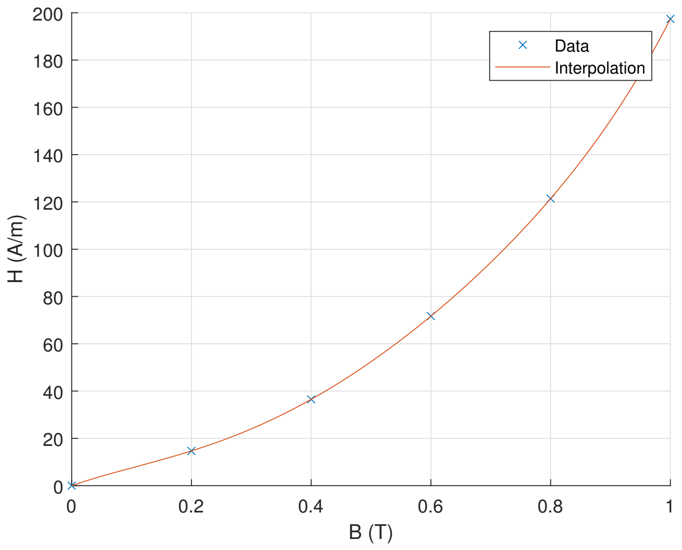


Fig. 1. A Lagrange interpolation of the first 6 points in the B-H data for M19 steel.

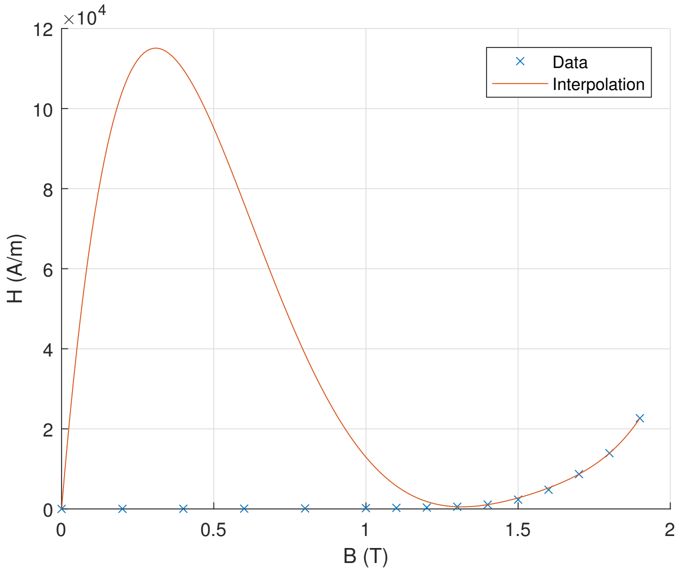


Fig. 2. A Lagrange interpolation of the 6 points at $B = 0.0\,\mathrm{T}$, $B = 1.3\,\mathrm{T}$, $B = 1.4\,\mathrm{T}$, $B = 1.7\,\mathrm{T}$, $B = 1.8\,\mathrm{T}$, and $B = 1.9\,\mathrm{T}$ in the B-H data for M19 steel.

provide values for the derivatives of the function at the given points. The simplest approach to this is to provide a numerical estimate of the derivative using the given data. Therefore, we used the slope between the points immediately before and after a given point to estimate the derivative at that point. For $B = 0.0\,\mathrm{T}$ and $B = 1.9\,\mathrm{T}$, we instead used a one-sided estimate of the derivative since points were not available on both sides. The results of doing this are shown in Figure 3. From this figure, we can see that the interpolation is better than the one using Lagrange polynomials. However, it still has some problems. Due to the large gap in data, the curve initially decreases below $0.0\,\mathrm{A/m}$, which is not a realistic scenario.

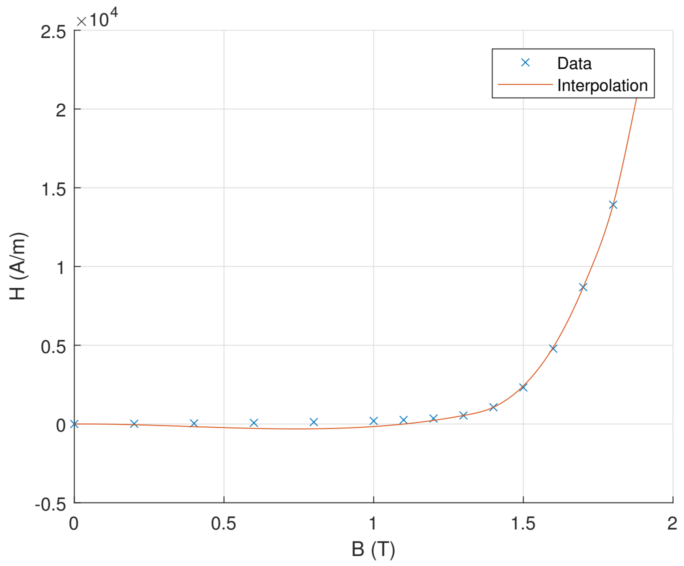


Fig. 3. A piecewise cubic Hermite interpolation of the 6 points at $B = 0.0\,\mathrm{T}$, $B = 1.3\,\mathrm{T}$, $B = 1.4\,\mathrm{T}$, $B = 1.7\,\mathrm{T}$, $B = 1.8\,\mathrm{T}$, and $B = 1.9\,\mathrm{T}$ in the B-H data for M19 steel.

Question 2

(a)

We started by deriving an equation for the flux Ψ in the magnetic circuit. From Ampere's law, we have

$$NI = H_c l_c + H_g l_g \tag{1}$$

Since the cross-sectional area S is uniform in the solenoid, so are the magnetic flux Ψ and the magnetic flux density B. Given that H_c depends on B according to the non-linear M19 steel data, and that H_g depends on B in a linear fashion, we can write

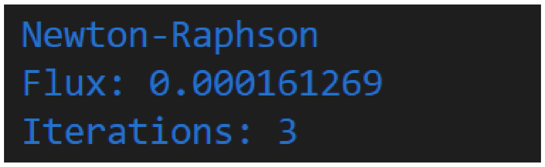
$$NI = H_c(B)l_c + Bl_g/\mu_0 \tag{2}$$
$$NI = H_c(\Psi/S)l_c + \Psi l_g/(\mu_0 S) \tag{3}$$

Thus, the equation to solve is $f(\Psi) = 0$, where

$$f(\Psi) = H_c(\Psi/S)l_c + \Psi l_g/(\mu_0 S) - NI \tag{4}$$
$$f'(\Psi) = H_c'(\Psi/S)l_c/S + l_g/(\mu_0 S) \tag{5}$$

(b)

With the equations derived in the previous section, we used the Newton-Raphson method to solve for the flux Ψ. The code that does this is shown in Sections F and G. We used a piecewise linear interpolation for $H_c(B)$ and $H_c'(B)$. We then started the solver with $\Psi = 0.0\,\mathrm{Tm}^2$ and stopped when $|f(\Psi)/f(0)| < 10^{-6}$. The resulting output of the program is shown in Figure 4, which shows that the final result was $\Psi = 0.000\,161\,269\,\mathrm{Tm}^2$, corresponding to a flux density of $B = 1.612\,69\,\mathrm{T}$. The figure also shows that convergence took only 3 iterations.



```
Newton-Raphson
Flux: 0.000161269
Iterations: 3
```

Fig. 4. The output of the program for the magnetic circuit solver using the Newton-Raphson method.

(c)

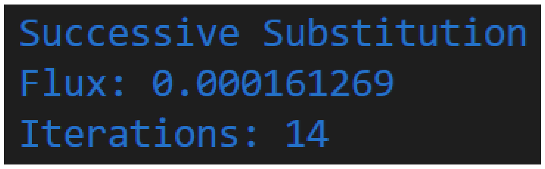
We next tried solving the same magnetic circuit using successive substitution. We first modified the equation to be in the form $\Psi = f(\Psi)$, as

$$\Psi l_g/(\mu_0 S) = NI - H_c(\Psi/S)l_c \tag{6}$$
$$\Psi = \mu_0 S/l_g(NI - H_c(\Psi/S)l_c) \tag{7}$$

We tried using the successive substitution solver shown in Sections F and G, but the problem failed to converge to a solution. In order to make the problem converge, we performed an inversion of the given equation, as

$$H_c(\Psi/S)l_c = NI - \Psi l_g/(\mu_0 S) \tag{8}$$
$$H_c(\Psi/S) = (NI - \Psi l_g/(\mu_0 S))/l_c \tag{9}$$
$$\Psi = SH_c^{-1}((NI - \Psi l_g/(\mu_0 S))/l_c) \tag{10}$$

This reformulation of the problem did converge using successive substitution, and the results are shown in Figure 5. This figure shows that the solver converged to the same solution of $\Psi = 0.000\,161\,269\,\mathrm{Tm}^2$, but said convergence was a lot slower, taking 14 iterations.



```
Successive Substitution
Flux: 0.000161269
Iterations: 14
```

Fig. 5. The output of the program for the magnetic circuit solver using the successive substitution method.

Question 3

(a)

We next attempted to solve the non-linear electric circuit shown in Figure 6 by using a node voltage method to derive a system of equations. We defined the node voltages v_1 and v_2 as shown in the figure, and derived the system $\boldsymbol{f}(\boldsymbol{v}) = \boldsymbol{0}$, where

$$\boldsymbol{f}(\boldsymbol{v}) = \begin{bmatrix} (v_1 - E)/R + I_{sA}(e^{(v_1-v_2)/v_t} - 1) \\ -I_{sA}(e^{(v_1-v_2)/v_t} - 1) + I_{sB}(e^{v_2/v_t} - 1) \end{bmatrix} \tag{11}$$

Note that we used $v_t = kT/q = 25\,\mathrm{mV}$.

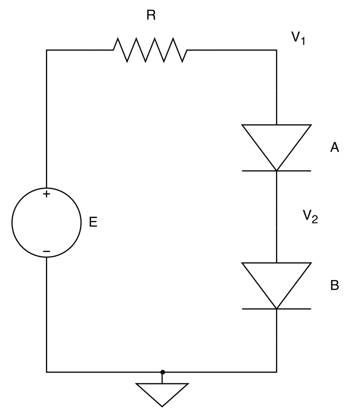


Fig. 6. The non-linear electric circuit used in Question 3.

(b)

In order to solve the system of equations using the Newton-Raphson method, we also derived a formula for the Jacobian of $\boldsymbol{f}$ as

$$\boldsymbol{J}(\boldsymbol{v}) = \begin{bmatrix} \partial f_1/\partial v_1 & \partial f_1/\partial v_2 \\ \partial f_2/\partial v_1 & \partial f_2/\partial v_2 \end{bmatrix} \tag{12}$$

where

$$\partial f_1/\partial v_1 = 1/R + I_{sA} e^{(v_1 - v_2)/v_t}/v_t \tag{13}$$
$$\partial f_1/\partial v_2 = -I_{sA} e^{(v_1 - v_2)/v_t}/v_t \tag{14}$$
$$\partial f_2/\partial v_1 = -I_{sA} e^{(v_1 - v_2)/v_t}/v_t \tag{15}$$
$$\partial f_2/\partial v_2 = I_{sA} e^{(v_1 - v_2)/v_t}/v_t + I_{sB} e^{v_2/v_t}/v_t \tag{16}$$

Since this Jacobian is positive definite, we used the Cholesky decomposition based solver we implemented in Assignment #1 to solve the update equation given by

$$\boldsymbol{J}(\boldsymbol{v}^k)\boldsymbol{v}^{k+1} = \boldsymbol{J}(\boldsymbol{v}^k)\boldsymbol{v}^k - \boldsymbol{f}(\boldsymbol{v}^k) \tag{17}$$

The code that does this is shown in Section J. In addition, the Newton-Raphson solver for matrix equations is shown in Sections F and G. We defined the error ϵ_k at each iteration as $\epsilon_k = \sum_i |f_i(\boldsymbol{v}^k)| / \sum_i |f_i(\boldsymbol{v}^0)|$, and stopped when $\epsilon_k < 10^{-6}$.

The results of the solver are shown in Figures 7 to 10. In Figure 7, we can see that the voltages across the diodes converged to values of $V_A = 0.107\,563\,3\,\mathrm{V}$ and $V_B = 0.090\,570\,7\,\mathrm{V}$ in only 6 iterations. Figure 8 shows the values of the function $\boldsymbol{f}$ across these 6 iterations. Finally, Figure 9 shows the error for each iteration.

In order to evaluate if the convergence was quadratic, we plotted $\epsilon_k/\epsilon_{k-1}^2$ in Figure 10. This figure shows that starting with iteration 2, this ratio tended asymptotically towards a constant value. This indicates that the convergence is indeed quadratic after a few iterations. Unfortunately, the convergence of the function was so quick that we could not observe the effect in more detail.

Question 4

(a)

Next, we performed numerical integration on the function $f(x) = cos(x)$ from $x = 0$ to $x = 1$ using one-point

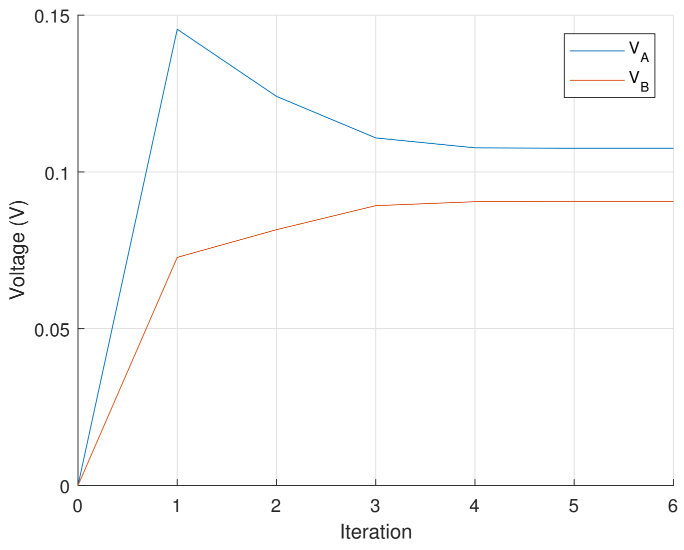


Fig. 7. Diode voltages vs. iteration for the Newton-Raphson solver used to solve the circuit in Figure 6.

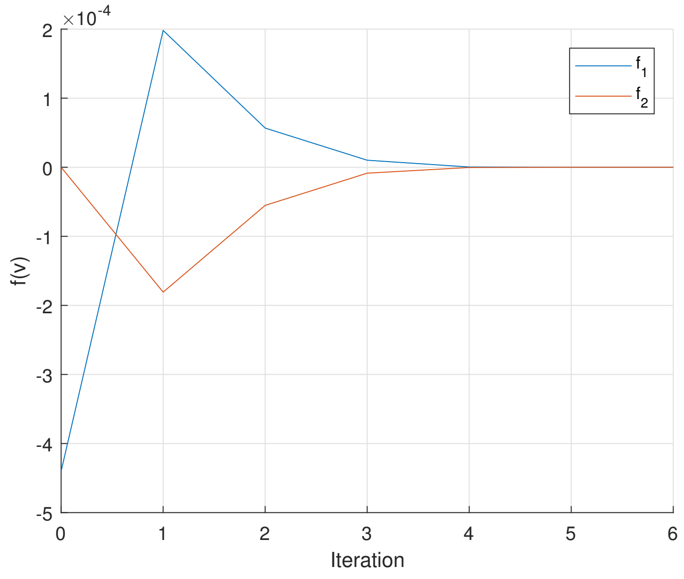


Fig. 8. Residual vs. iteration for the Newton-Raphson solver used to solve the circuit in Figure 6.

Gauss-Legendre integration over N equal segments. The code that performs this function is shown in Sections K and L. The results are shown in Figure 11, which shows the natural logarithm of the absolute error plotted against the natural logarithm of the number of segments. This figure shows a clear linear trend, with a mean slope of -2.0031. This indicates that the error is approximately proportional to $1/N^2$, as we would expect since the one-point method integrates linear functions exactly and thus the error should be quadratic.

(b)

Next, we applied the same method to the function $f(x) = ln(x)$ from $x = 0$ to $x = 1$. The results are shown in Figure 12, which shows the natural logarithm of the absolute error plotted against the natural logarithm of the number of segments.

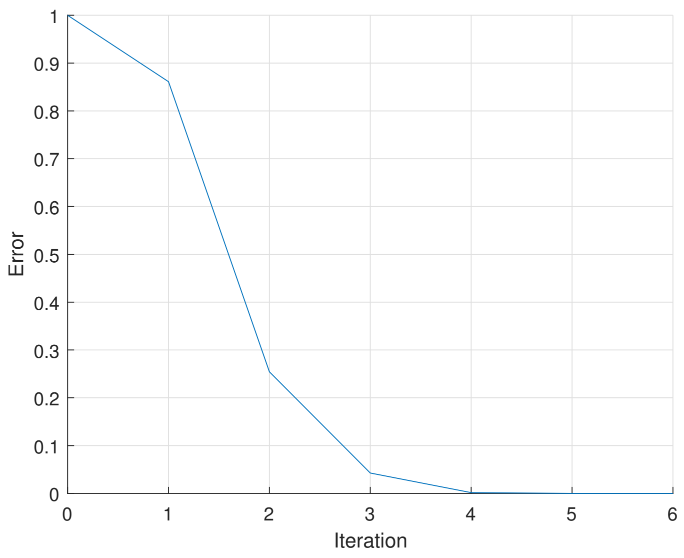


Fig. 9. Error vs. iteration for the Newton-Raphson solver used to solve the circuit in Figure 6.

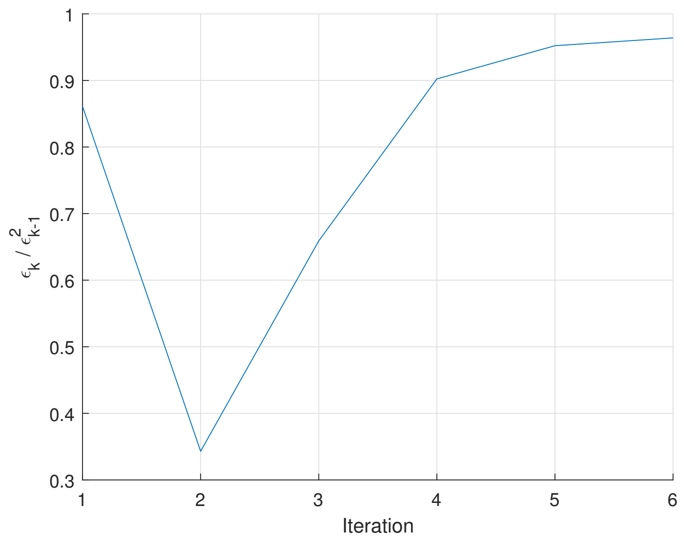


Fig. 10. Error ratio vs. iteration for the Newton-Raphson solver used to solve the circuit in Figure 6.

This figure shows a clear linear trend, with a mean slope of -0.9981. This indicates that the error is approximately proportional to $1/N$. Thus the error for this function has a lower order than for $f(x) = cos(x)$. This can be explained by the fact that $f(x) = ln(x)$ is unbounded on the integration interval. Indeed, since the second derivative tends to $-\infty$ as $x \to 0$, we cannot use a simple Taylor series to show that the error should be quadratic on this interval.

(c)

Finally, we attempted to integrate the function $f(x) = ln(x)$ as accurately as possible using a one-point Gauss-Legendre method with only 10 segments. We wrote a function that accepts a vector of segment lengths and performs this computation. The code for this is shown in Sections K and L.

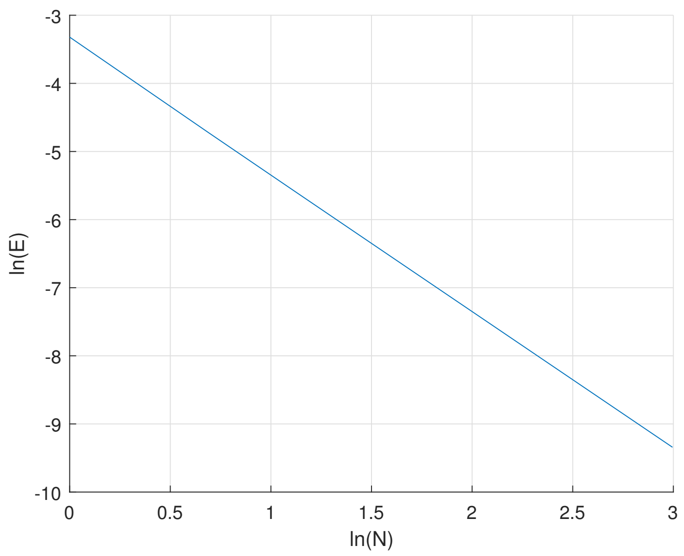


Fig. 11. $ln(E)$ vs. $ln(N)$ for the integral of $f(x) = cos(x)$ from $x = 0$ to $x = 1$ using one-point Gauss-Legendre integration over N equal segments.

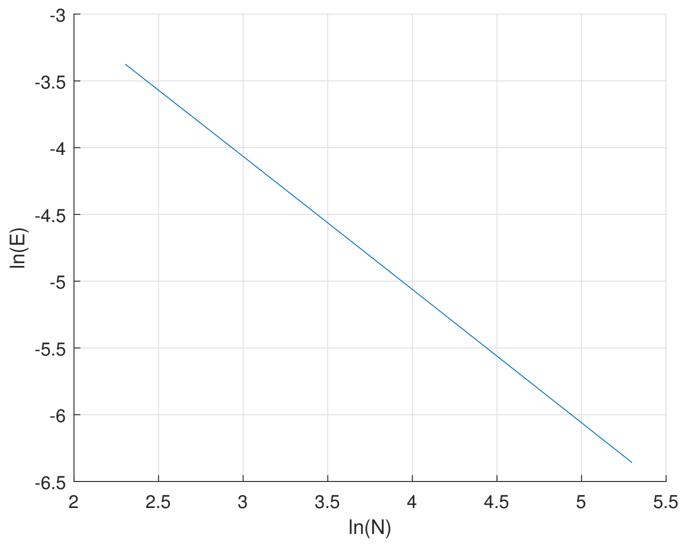


Fig. 12. $ln(E)$ vs. $ln(N)$ for the integral of $f(x) = ln(x)$ from $x = 0$ to $x = 1$ using one-point Gauss-Legendre integration over N equal segments.

In order to perform a methodical study, we decided to divide the lengths according to a geometric ratio, which would make the segments close to the singularity at $x = 0$ shorter compared to the segments close to $x = 1$. Thus, we used segments of the form α, αr, ..., αr^9. We varied r from 1 to 2 and computed α using

$$\alpha(1 + r + ... + r^9) = 1 \quad (18)$$

$$\alpha(r^{10} - 1)/(r - 1) = 1 \quad (19)$$

$$\alpha = (r - 1)/(r^{10} - 1) \quad (20)$$

The results of this approach are shown in Figure 13. This figure shows that the error reaches a minimum of 0.0106114 at $r = 1.45$, which corresponds to $\alpha = 0.0112262$.

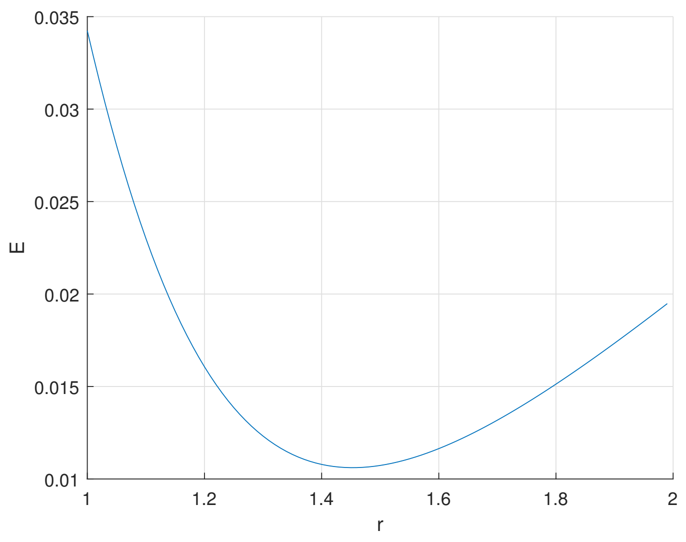


Fig. 13. E vs. r for the integral of $f(x) = ln(x)$ from $x = 0$ to $x = 1$ using one-point Gauss-Legendre integration over 10 unequal segments.

APPENDIX A
MAIN.CPP

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <utility>
#include <functional>
#include <cmath>

#include "interpolation.h"
#include "solver.h"
#include "matrix.h"
#include "integral.h"

using namespace Numeric;

void question1();
void question2();
void question3();
void question4();

void printToFile(
    const std::vector<std::pair<double, double>>& data,
    const std::string& filename,
    const std::string& header);

int main()
{
    question1();
    question2();
    question3();
    question4();
    return 0;
}

void question1()
{
    std::cout << "========== Question 1 ==========" << std::endl;
    std::cout << std::endl;

    static const std::vector<std::pair<double, double>> data{{0.0, 0.0},
                                                             {0.2, 14.7},
                                                             {0.4, 36.5},
                                                             {0.6, 71.7},
                                                             {0.8, 121.4},
                                                             {1.0, 197.4},
                                                             {1.1, 256.2},
                                                             {1.2, 348.7},
                                                             {1.3, 540.6},
                                                             {1.4, 1062.8},
                                                             {1.5, 2318.0},
                                                             {1.6, 4781.9},
                                                             {1.7, 8687.4},
                                                             {1.8, 13924.3},
                                                             {1.9, 22650.2}};
    printToFile(data, "q1data.csv", "B (T),H (A/m)");
    auto f = lagrange({data.begin(), data.begin() + 6});
    auto interpol = sample(f, 0.0, 1.01, 0.01);
    printToFile(interpol, "q1a.csv", "B (T),H (A/m)");
    std::cout << "Question 1 (a) data successfully printed to q1a.csv"
              << std::endl;
    std::cout << std::endl;

    f = lagrange({data[0], data[8], data[9], data[12], data[13], data[14]});
    interpol = sample(f, 0.0, 1.91, 0.01);
    printToFile(interpol, "q1b.csv", "B (T),H (A/m)");
    std::cout << "Question 1 (b) data successfully printed to q1b.csv"
              << std::endl;
    std::cout << std::endl;

    f = hermite(
        {data[0], data[8], data[9], data[12], data[13], data[14]},
        {slope(data[0], data[1]),
         slope(data[7], data[9]),
         slope(data[8], data[10]),
         slope(data[11], data[13]),
         slope(data[12], data[14]),
         slope(data[13], data[14])});
    interpol = sample(f, 0.0, 1.91, 0.01);
    printToFile(interpol, "q1c.csv", "B (T),H (A/m)");
    std::cout << "Question 1 (c) data successfully printed to q1c.csv"
              << std::endl;
    std::cout << std::endl;
}

void question2()
{
    std::cout << "========== Question 2 ==========" << std::endl;
    std::cout << std::endl;

    static const std::vector<std::pair<double, double>> data{{0.0, 0.0},
                                                             {0.2, 14.7},
                                                             {0.4, 36.5},
                                                             {0.6, 71.7},
                                                             {0.8, 121.4},
                                                             {1.0, 197.4},
                                                             {1.1, 256.2},
                                                             {1.2, 348.7},
                                                             {1.3, 540.6},
                                                             {1.4, 1062.8},
                                                             {1.5, 2318.0},
                                                             {1.6, 4781.9},
                                                             {1.7, 8687.4},
                                                             {1.8, 13924.3},
                                                             {1.9, 22650.2}};
    static const double N = 1000;
    static const double I = 8;
    static const double S = 1e-4;
    static const double Lc = 30e-2;
    static const double Lg = 0.5e-2;
    static const double mu0 = 4 * M_PI * 1e-7;

    auto H = pwl(data);
    auto Hprime = pwlprime(data);
    auto B = pwlinverse(data);

    std::function<double(double)> f = [&](double flux) {
        return flux * Lg / (mu0 * S) + H(flux / S) * Lc - N * I;
    };
    std::function<double(double)> fprime = [&](double flux) {
        return Lg / (mu0 * S) + Lc * Hprime(flux / S) / S;
    };

    double flux;
    int iterations;
    std::tie(flux, iterations) = newtonRaphson(f, fprime, 0.0);
    std::cout << "Newton-Raphson" << std::endl;
    std::cout << "Flux: " << flux << std::endl;
    std::cout << "Iterations: " << iterations << std::endl;
    std::cout << std::endl;

    f = [&](double flux) {
        return S * B((N * I - Lg * flux / (mu0 * S)) / Lc);
    };

    std::tie(flux, iterations) = successiveSubstitution(f, 0.0);
    std::cout << "Successive Substitution" << std::endl;
    std::cout << "Flux: " << flux << std::endl;
    std::cout << "Iterations: " << iterations << std::endl;
    std::cout << std::endl;
}

void question3()
{
    std::cout << "========== Question 3 ==========" << std::endl;
    std::cout << std::endl;

    static const double E = 0.22;
    static const double R = 500;
    static const double vt = 25e-3;
    static const double IsA = 0.6e-6;
    static const double IsB = 1.2e-6;

    auto f = [&](const Matrix<double>& v) {
        Matrix<double> result{2, 1};
        result(0) = (v(0) - E) / R + IsA * (std::exp((v(0) - v(1)) / vt) - 1);
        result(1) = -IsA * (std::exp((v(0) - v(1)) / vt) - 1) +
            IsB * (std::exp(v(1) / vt) - 1);
        return result;
    };
    auto jacobian = [&](const Matrix<double>& v) {
        Matrix<double> result{2, 2};
        result(0, 0) = 1 / R + IsA / vt * std::exp((v(0) - v(1)) / vt);
        result(0, 1) = -IsA / vt * std::exp((v(0) - v(1)) / vt);
        result(1, 0) = -IsA / vt * std::exp((v(0) - v(1)) / vt);
        result(1, 1) = IsA / vt * std::exp((v(0) - v(1)) / vt) +
            IsB / vt * std::exp(v(1) / vt);
        return result;
    };

    Matrix<double> v;
    int iterations;
    std::cout << "Newton-Raphson" << std::endl;
    std::cout << "iteration,v1,v2,f1,f2,error" << std::endl;
    std::tie(v, iterations) = newtonRaphson(
        f, jacobian, "[0.0; 0.0]", [&](auto v, auto error, auto iteration) {
            auto residual = f(v);
            std::cout << iteration << "," << v(0) << "," << v(1) << ","
                      << residual(0) << "," << residual(1) << "," << error
                      << std::endl;
        });
    std::cout << std::endl;
}

void question4()
{
    std::cout << "========== Question 4 ==========" << std::endl;
    std::cout << std::endl;

    std::cout << "Integrating cos(x)" << std::endl;
    std::cout << "segments,error" << std::endl;
    for (int n = 1; n <= 20; ++n)
    {
        double result =
            integral([](double x) { return std::cos(x); }, 0.0, 1.0, n);
        double error = std::abs(std::sin(1.0) - result);
        std::cout << n << "," << error << std::endl;
    }
    std::cout << std::endl;

    std::cout << "Integrating log(x)" << std::endl;
    std::cout << "segments,error" << std::endl;
    for (int n = 10; n <= 200; n += 10)
    {
        double result =
            integral([](double x) { return std::log(x); }, 0.0, 1.0, n);
        double error = std::abs(-1.0 - result);
        std::cout << n << "," << error << std::endl;
    }
    std::cout << std::endl;

    std::cout << "Integrating log(x) with uneven segments" << std::endl;
    std::cout << "r,alpha,error" << std::endl;
    for (double r = 1.0; r < 2.0; r += 0.01)
    {
        double alpha =
            r == 1.0 ? 1 / 10.0 : (r - 1.0) / (std::pow(r, 10) - 1.0);
        std::vector<double> segments;
        double totalLength = 0.0;
        for (int i = 0; i < 9; ++i)
        {
            double segment = alpha * std::pow(r, i);
            segments.push_back(segment);
            totalLength += segment;
        }
        double segment = 1.0 - totalLength;
        segments.push_back(segment);
        double result =
            integral([](double x) { return std::log(x); }, 0.0, segments);
        double error = std::abs(-1.0 - result);
        std::cout << r << "," << alpha << "," << error << std::endl;
    }
    std::cout << std::endl;
}

void printToFile(
    const std::vector<std::pair<double, double>>& data,
    const std::string& filename,
    const std::string& header)
{
    std::ofstream out{filename};
    out << header << std::endl;
    for (auto p : data)
        out << p.first << "," << p.second << std::endl;
}
```

APPENDIX B
POLYNOMIAL.H

```cpp
#pragma once

#include <vector>
#include <iosfwd>

namespace Numeric {

struct Polynomial
{
    std::vector<double> coefficients;

    Polynomial() = default;

    // Contructs a polynomial from the given coefficients in ascending order.
    Polynomial(std::vector<double> coefficients) : coefficients{coefficients}
    {
    }

    // Implicit conversion from double.
    Polynomial(double a0) : coefficients{{a0}}
    {
    }

    // Evaluates the polynomial at the given value of x.
    double operator()(double x) const;

    bool operator==(const Polynomial& other) const;
    bool operator!=(const Polynomial& other) const;
};

// Polynomial addition.
Polynomial operator+(Polynomial lhs, const Polynomial& rhs);
Polynomial& operator+=(Polynomial& lhs, const Polynomial& rhs);

// Polynomial subtraction.
Polynomial operator-(Polynomial lhs, const Polynomial& rhs);
Polynomial& operator-=(Polynomial& lhs, const Polynomial& rhs);

// Polynomial multiplication.
Polynomial operator*(const Polynomial& lhs, const Polynomial& rhs);
Polynomial& operator*=(Polynomial& lhs, const Polynomial& rhs);

// Scalar division.
Polynomial operator/(Polynomial lhs, double rhs);
Polynomial& operator/=(Polynomial& lhs, double rhs);

// Stream I/O.
std::ostream& operator<<(std::ostream& out, const Polynomial& polynomial);

} // namespace Numeric
```

APPENDIX C
POLYNOMIAL.CPP

```
#include "polynomial.h"

#include <iostream>

namespace Numeric {

double Polynomial::operator()(double x) const
{
    double result = 0.0;
    double power = 1.0;
    for (auto coefficient : coefficients)
    {
        result += coefficient * power;
        power *= x;
    }
    return result;
}

bool Polynomial::operator==(const Polynomial& other) const
{
    return coefficients == other.coefficients;
}

bool Polynomial::operator!=(const Polynomial& other) const
{
    return !(*this == other);
}

Polynomial operator+(Polynomial lhs, const Polynomial& rhs)
{
    lhs += rhs;
    return lhs;
}

Polynomial& operator+=(Polynomial& lhs, const Polynomial& rhs)
{
    if (rhs.coefficients.size() > lhs.coefficients.size())
        lhs.coefficients.resize(rhs.coefficients.size());
    for (size_t i = 0; i < rhs.coefficients.size(); ++i)
        lhs.coefficients[i] += rhs.coefficients[i];
    return lhs;
}

Polynomial operator-(Polynomial lhs, const Polynomial& rhs)
{
    lhs -= rhs;
    return lhs;
}

Polynomial& operator-=(Polynomial& lhs, const Polynomial& rhs)
{
    if (rhs.coefficients.size() > lhs.coefficients.size())
        lhs.coefficients.resize(rhs.coefficients.size());
    for (size_t i = 0; i < rhs.coefficients.size(); ++i)
        lhs.coefficients[i] -= rhs.coefficients[i];
    return lhs;
}

Polynomial operator*(const Polynomial& lhs, const Polynomial& rhs)
{
    std::vector<double> coefficients(
        lhs.coefficients.size() + rhs.coefficients.size() - 1);
    for (size_t i = 0; i < lhs.coefficients.size(); ++i)
        for (size_t j = 0; j < rhs.coefficients.size(); ++j)
            coefficients[i + j] += lhs.coefficients[i] * rhs.coefficients[j];
    return Polynomial{coefficients};
}

Polynomial& operator*=(Polynomial& lhs, const Polynomial& rhs)
{
    lhs = lhs * rhs;
    return lhs;
}

Polynomial operator/(Polynomial lhs, double rhs)
{
    lhs /= rhs;
    return lhs;
}

Polynomial& operator/=(Polynomial& lhs, double rhs)
{
    for (auto& coefficient : lhs.coefficients)
        coefficient /= rhs;
    return lhs;
}

std::ostream& operator<<(std::ostream& out, const Polynomial& polynomial)
{
    int n = static_cast<int>(polynomial.coefficients.size());
    for (int i = n - 1; i > 0; --i)
        out << polynomial.coefficients[i] << "x^" << i << " + ";
    out << polynomial.coefficients[0];
    return out;
}

} // namespace Numeric
```

Appendix D
Interpolation.h

```
#pragma once

#include <vector>
#include <utility>
#include <functional>

namespace Numeric {

// Interpolates the given data using a single Lagrange polynomial.
std::function<double(double)>
    lagrange(const std::vector<std::pair<double, double>>& data);

// Interpolates the given data using piecewise cubic Hermite polynomials.
std::function<double(double)> hermite(
    const std::vector<std::pair<double, double>>& data,
    const std::vector<double>& derivatives);

// Interpolates the given data using piecewise linear polynomials.
std::function<double(double)>
    pwl(const std::vector<std::pair<double, double>>& data);

// Computes the derivative of the piecewise linear interpolation of the data.
std::function<double(double)>
    pwlprime(const std::vector<std::pair<double, double>>& data);

// Interpolates the inverse of the given data using piecewise linear
// polynomials. This function assumes that the y coordinates are increasing.
std::function<double(double)>
    pwlinverse(const std::vector<std::pair<double, double>>& data);

// Computes the slope between the given points.
double slope(std::pair<double, double> p1, std::pair<double, double> p2);

// Samples the given function at uniformly spaced points in [x0, x1].
// This function samples with a difference dx between points.
std::vector<std::pair<double, double>>
    sample(std::function<double(double)> f, double x0, double x1, double dx);

} // namespace Numeric
```

APPENDIX E
INTERPOLATION.CPP

```
#include "interpolation.h"

#include "polynomial.h"

namespace Numeric {

std::function<double(double)>
    lagrange(const std::vector<std::pair<double, double>>& data)
{
    int n = static_cast<int>(data.size());
    Polynomial y{std::vector<double>(n)};

    for (int i = 0; i < n; ++i)
    {
        Polynomial lagrangePolynomial{{1.0}};
        for (int j = 0; j < n; ++j)
        {
            if (i == j)
                continue;
            lagrangePolynomial *= Polynomial{{-data[j].first, 1.0}};
            lagrangePolynomial /= data[i].first - data[j].first;
        }
        y += data[i].second * lagrangePolynomial;
    }

    return y;
}

std::function<double(double)> hermite(
    const std::vector<std::pair<double, double>>& data,
    const std::vector<double>& derivatives)
{
    int n = static_cast<int>(data.size());
    std::vector<std::pair<double, Polynomial>> y;

    for (int i = 0; i < n - 1; ++i)
    {
        Polynomial interval{std::vector<double>(2 * n - 1)};

        double diff = data[i + 1].first - data[i].first;
        Polynomial F1{{-data[i].first, 1.0}};
        Polynomial F2{{-data[i + 1].first, 1.0}};

        auto L1 = F2 / -diff;
        auto L2 = F1 / diff;
        auto L1prime = 1 / -diff;
        auto L2prime = 1 / diff;

        auto U1 = (1.0 - 2.0 * L1prime * F1) * L1 * L1;
        auto U2 = (1.0 - 2.0 * L2prime * F2) * L2 * L2;
        auto V1 = F1 * L1 * L1;
        auto V2 = F2 * L2 * L2;

        interval = data[i].second * U1 + data[i + 1].second * U2 +
            derivatives[i] * V1 + derivatives[i + 1] * V2;
        y.emplace_back(data[i + 1].first, interval);
    }

    return [=](double x) {
        for (const auto& interval : y)
            if (x < interval.first)
                return interval.second(x);
        return y.back().second(x);
    };
}

std::function<double(double)>
    pwl(const std::vector<std::pair<double, double>>& data)
{
    return [=](double x) {
        int index = static_cast<int>(data.size()) - 1;
        for (size_t i = 1; i < data.size(); ++i)
        {
            if (x < data[i].first)
            {
                index = i;
                break;
            }
        }
        return slope(data[index - 1], data[index]) * (x - data[index].first) +
            data[index].second;
    };
}

std::function<double(double)>
    pwlprime(const std::vector<std::pair<double, double>>& data)
{
    return [=](double x) {
        int index = static_cast<int>(data.size()) - 1;
        for (size_t i = 1; i < data.size(); ++i)
        {
            if (x < data[i].first)
            {
                index = i;
                break;
            }
        }
        return slope(data[index - 1], data[index]);
    };
}

std::function<double(double)>
    pwlinverse(const std::vector<std::pair<double, double>>& data)
{
    return [=](double y) {
        int index = static_cast<int>(data.size()) - 1;
        for (size_t i = 1; i < data.size(); ++i)
        {
            if (y < data[i].second)
            {
                index = i;
                break;
            }
        }
        return 1 / slope(data[index - 1], data[index]) *
            (y - data[index].second) +
            data[index].first;
    };
}

double slope(std::pair<double, double> p1, std::pair<double, double> p2)
{
    return (p2.second - p1.second) / (p2.first - p1.first);
}

std::vector<std::pair<double, double>>
    sample(std::function<double(double)> f, double x0, double x1, double dx)
{
    std::vector<std::pair<double, double>> data;
    for (double x = x0; x <= x1; x += dx)
        data.emplace_back(x, f(x));
    return data;
}

} // namespace Numeric
```

APPENDIX F
SOLVER.H

```
#pragma once

#include <utility>
#include <functional>

#include "matrix.h"

namespace Numeric {

// Solves the equation f(x) = 0 using the Newton-Raphson method.
std::pair<double, int> newtonRaphson(
    std::function<double(double)> f,
    std::function<double(double)> fprime,
    double x0);

// Solves the equation x = f(x) using successive substitution.
std::pair<double, int>
    successiveSubstitution(std::function<double(double)> f, double x0);

// Solves the matrix equation f(x) = 0 using the Newton-Raphson method.
std::pair<Matrix<double>, int> newtonRaphson(
    std::function<Matrix<double>(const Matrix<double>&)> f,
    std::function<Matrix<double>(const Matrix<double>&)> jacobian,
    const Matrix<double>& x0,
    std::function<void(const Matrix<double>&, double, int)> callback =
        [](auto x, auto error, auto iteration) {});

} // namespace Numeric
```

APPENDIX G
SOLVER.CPP

```
#include "solver.h"

#include <algorithm>
#include <cmath>

#include "matrix-solver.h"

namespace Numeric {

std::pair<double, int> newtonRaphson(
    std::function<double(double)> f,
    std::function<double(double)> fprime,
    double x0)
{
    auto error = [&](auto x) { return std::fabs(f(x) / f(x0)); };

    int iterations = 0;
    double x = x0;
    while (error(x) >= 1e-6)
    {
        ++iterations;
        x -= f(x) / fprime(x);
    }
    return {x, iterations};
}

std::pair<double, int>
    successiveSubstitution(std::function<double(double)> f, double x0)
{
    auto error = [&](auto x) { return std::fabs((f(x) - x) / (f(x0) - x0)); };

    int iterations = 0;
    double x = x0;
    while (error(x) >= 1e-6)
    {
        ++iterations;
        x = f(x);
    }
    return {x, iterations};
}

std::pair<Matrix<double>, int> newtonRaphson(
    std::function<Matrix<double>(const Matrix<double>&)> f,
    std::function<Matrix<double>(const Matrix<double>&)> jacobian,
    const Matrix<double>& x0,
    std::function<void(const Matrix<double>&, double, int)> callback)
{
    auto error = [&](const auto& x) {
        auto func = f(x);
        auto numerator = std::accumulate(
            func.begin(), func.end(), 0.0, [](auto lhs, auto rhs) {
                return lhs + std::abs(rhs);
            });
        func = f(x0);
        auto denominator = std::accumulate(
            func.begin(), func.end(), 0.0, [](auto lhs, auto rhs) {
                return lhs + std::abs(rhs);
            });
        return numerator / denominator;
    };

    int iterations = 0;
    Matrix<double> x = x0;
    while (error(x) >= 1e-6)
    {
        callback(x, error(x), iterations);
        ++iterations;
        x = solve(jacobian(x), jacobian(x) * x - f(x));
    }
    callback(x, error(x), iterations);
    return {x, iterations};
}

} // namespace Numeric
```

APPENDIX H
MATRIX.H

```
#pragma once

#include <vector>
#include <iostream>
#include <string>
#include <sstream>
#include <utility>

namespace Numeric {

template <typename T>
class Matrix;

// Matrix addition.
template <typename T>
Matrix<T> operator+(Matrix<T> lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator+=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Matrix subtraction.
template <typename T>
Matrix<T> operator-(Matrix<T> lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator-=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Matrix multiplication.
template <typename T>
Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Scalar multiplication.
template <typename T>
Matrix<T> operator*(Matrix<T> lhs, const T& rhs);
template <typename T>
Matrix<T> operator*(const T& lhs, Matrix<T> rhs);
template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const T& rhs);

// Scalar division.
template <typename T>
Matrix<T> operator/(Matrix<T> lhs, const T& rhs);
template <typename T>
Matrix<T>& operator/=(Matrix<T>& lhs, const T& rhs);

// Matrix transpose.
template <typename T>
Matrix<T> transpose(const Matrix<T>& m);

// Computes the lower and upper bandwidths of matrix m.
template <typename T>
std::pair<int, int> bandwidth(const Matrix<T>& m);

// Matrix I/O.
template <typename T>
```

```
std::istream& operator>>(std::istream& in, Matrix<T>& m);
template <typename T>
std::ostream& operator<<(std::ostream& out, const Matrix<T>& m);

template <typename T>
class Matrix
{
public:
    using Iterator = typename std::vector<T>::iterator;
    using ConstIterator = typename std::vector<T>::const_iterator;

    // Constructs a matrix of size 0.
    Matrix() = default;

    // Constructs a matrix of size [rows x cols] filled with zeros.
    Matrix(int rows, int cols)
        : m_rows{rows}, m_cols{cols}, m_elements(rows * cols)
    {
    }

    // Implicit conversion from a string.
    Matrix(const std::string& str)
    {
        std::stringstream ss{str};
        ss >> *this;
    }

    // Implicit conversion from a c string.
    Matrix(const char* str)
    {
        std::stringstream ss{str};
        ss >> *this;
    }

    // Implicit conversion to the element type for a single element matrix.
    operator T() const
    {
        if (size() != 1)
            throw std::runtime_error{"cannot convert matrix to single element"};
        return (*this)(0);
    }

    // Indexes the matrix in a row-major order.
    T& operator()(int index)
    {
        return m_elements.at(index);
    }

    // Indexes the matrix in a row-major order.
    const T& operator()(int index) const
    {
        return m_elements.at(index);
    }

    T& operator()(int row, int col)
    {
        return (*this)(row * m_cols + col);
    }
```

```
const T& operator()(int row, int col) const
{
    return (*this)(row * m_cols + col);
}

// Returns the total number of elements in the matrix.
int size() const
{
    return m_rows * m_cols;
}

int rows() const
{
    return m_rows;
}

int cols() const
{
    return m_cols;
}

Iterator begin()
{
    return m_elements.begin();
}

Iterator end()
{
    return m_elements.end();
}

ConstIterator begin() const
{
    return m_elements.begin();
}

ConstIterator end() const
{
    return m_elements.end();
}

bool operator==(const Matrix<T>& other) const
{
    if (size() == 0 && other.size() == 0)
        return true;
    return m_rows == other.m_rows && m_cols == other.m_cols &&
        m_elements == other.m_elements;
}

bool operator!=(const Matrix<T>& other) const
{
    return !(*this == other);
}

friend std::istream& operator>><>(std::istream& in, Matrix<T>& m);
friend std::ostream& operator<<<>(std::ostream& out, const Matrix<T>& m);
```

```
private:
    int m_rows{};
    int m_cols{};
    std::vector<T> m_elements;

    // Constructs a matrix by interpreting the elements
    // as a matrix of size [rows x cols].
    Matrix(int rows, int cols, std::vector<T> elements)
        : m_rows{rows}, m_cols{cols}, m_elements{elements}
    {
        if (rows * cols != elements.size())
            throw std::runtime_error{"inconsistent matrix dimensions"};
    }
};

template <typename T>
Matrix<T> operator+(Matrix<T> lhs, const Matrix<T>& rhs)
{
    return lhs += rhs;
}

template <typename T>
Matrix<T>& operator+=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols())
        throw std::runtime_error{"add: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};

    int rows = lhs.rows(), cols = lhs.cols();
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            lhs(i, j) += rhs(i, j);
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator-(Matrix<T> lhs, const Matrix<T>& rhs)
{
    return lhs -= rhs;
}

template <typename T>
Matrix<T>& operator-=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols())
        throw std::runtime_error{"subtract: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};
    int rows = lhs.rows(), cols = lhs.cols();
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            lhs(i, j) -= rhs(i, j);
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.cols() != rhs.rows())
        throw std::runtime_error{"multiply: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};

    int rows = lhs.rows(), cols = rhs.cols(), innerSize = lhs.cols();
    Matrix<T> result{rows, cols};
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            for (int k = 0; k < innerSize; ++k)
            {
                result(i, j) += lhs(i, k) * rhs(k, j);
            }
        }
    }
    return result;
}

template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    lhs = lhs * rhs;
    return lhs;
}

template <typename T>
Matrix<T> operator*(Matrix<T> lhs, const T& rhs)
{
    return lhs *= rhs;
}

template <typename T>
Matrix<T> operator*(const T& lhs, Matrix<T> rhs)
{
    return rhs *= lhs;
}

template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const T& rhs)
{
    for (int i = 0; i < lhs.rows(); ++i)
    {
        for (int j = 0; j < lhs.cols(); ++j)
        {
            lhs(i, j) *= rhs;
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator/(Matrix<T> lhs, const T& rhs)
{
    return lhs /= rhs;
}

template <typename T>
Matrix<T>& operator/=(Matrix<T>& lhs, const T& rhs)
{
    if (rhs == 0)
        throw std::runtime_error{"divide: division by zero"};

    for (int i = 0; i < lhs.rows(); ++i)
    {
        for (int j = 0; j < lhs.cols(); ++j)
        {
            lhs(i, j) /= rhs;
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> transpose(const Matrix<T>& m)
{
    Matrix<T> result{m.cols(), m.rows()};
    for (int i = 0; i < m.rows(); ++i)
    {
        for (int j = 0; j < m.cols(); ++j)
        {
            result(j, i) = m(i, j);
        }
    }
    return result;
}

template <typename T>
std::pair<int, int> bandwidth(const Matrix<T>& m)
{
    int lower = 0;
    int upper = 0;
    for (int i = 0; i < m.rows(); ++i)
    {
        for (int j = 0; j < m.cols(); ++j)
        {
            if (m(i, j) != 0)
            {
                auto diag = j - i;
                if (diag == 0)
                {
                    lower = std::max(lower, diag + 1);
                    upper = std::max(upper, diag + 1);
                }
                else if (diag > 0)
                {
                    upper = std::max(upper, diag + 1);
                }
                else
                {
                    lower = std::max(lower, -diag + 1);
                }
            }
        }
    }
    return {lower, upper};
}

template <typename T>
std::istream& operator>>(std::istream& in, Matrix<T>& m)
{
    int rows = 1;
    int size = 0;
    std::vector<T> elements;

    char ch;
    in >> ch;
    if (!in || ch != '[')
        throw std::runtime_error{"input: missing '['"};

    while (true)
    {
        T elem;
        in >> elem;
        if (!in)
            throw std::runtime_error{"input: invalid element"};
        elements.push_back(elem);
        ++size;

        in >> ch;
        if (!in)
            throw std::runtime_error{"input: missing ']'"};

        if (ch == ',')
        {
            // Go to next element.
        }
        else if (ch == ';')
        {
            ++rows;
        }
        else if (ch == ']')
        {
            break;
        }
        else
        {
            throw std::runtime_error{"input: invalid separator"};
        }
    }

    int cols = size / rows;
    if (rows * cols != size)
        throw std::runtime_error{"input: inconsistent matrix dimensions"};

    m = Matrix<T>{rows, cols, std::move(elements)};
    return in;
}

template <typename T>
std::ostream& operator<<(std::ostream& out, const Matrix<T>& m)
{
    out << "[" << std::endl;
    for (int i = 0; i < m.rows(); ++i)
    {
        out << "    ";
        for (int j = 0; j < m.cols(); ++j)
        {
            out << m(i, j);
            if (j != m.cols() - 1)
                out << ", ";
        }
        if (i != m.rows() - 1)
            out << ";";
        out << std::endl;
    }
    out << "]" << std::endl;
    return out;
}
}
```

APPENDIX I
CHOLESKY.H

```
#pragma once

#include <utility>
#include <cmath>

#include "matrix.h"

namespace Numeric {

// Returns a pair consisting of the lower triangular matrix that forms the
// cholesky decomposition of the matrix m, if it exists, and a boolean
// indicating if the function succeeded (the matrix m is positive-definite).
template <typename T>
std::pair<Matrix<T>, bool> cholesky(const Matrix<T>& m)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"cholesky: matrix must be square"};

    int n = m.rows();
    Matrix<T> lower{n, n};
    for (int j = 0; j < n; ++j)
    {
        auto result = m(j, j);
        for (int i = 0; i < j; ++i)
        {
            result -= lower(j, i) * lower(j, i);
        }
        // If the square of L(j,j) is negative, the square root would fail.
        // If the square of L(j,j) is zero, then the matrix is singular.
        if (result <= 0)
            return {lower, false};
        lower(j, j) = sqrt(result);

        for (int i = j + 1; i < n; ++i)
        {
            auto result = m(i, j);
            for (int k = 0; k < j; ++k)
            {
                result -= lower(i, k) * lower(j, k);
            }
            result /= lower(j, j);
            lower(i, j) = result;
        }
    }
    return {lower, true};
}

// Returns a pair consisting of the lower triangular matrix that forms the
// cholesky decomposition of the matrix m, if it exists, and a boolean
// indicating if the function succeeded (the matrix m is positive-definite).
// This function takes advantage of the sparsity of the matrix to use the
// half-bandwidth in the decomposition.
template <typename T>
std::pair<Matrix<T>, bool> bcholesky(const Matrix<T>& m)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"bcholesky: matrix must be square"};

    auto b = bandwidth(m);
    if (b.first != b.second)
        return {{}, false}; // Matrix is not symmetric.
    auto bw = b.first;

    int n = m.rows();
    Matrix<T> lower{n, n};
    for (int j = 0; j < n; ++j)
    {
        auto result = m(j, j);
        for (int i = std::max(j - bw + 1, 0); i < j; ++i)
        {
            result -= lower(j, i) * lower(j, i);
        }
        // If the square of L(j,j) is negative, the square root would fail.
        // If the square of L(j,j) is zero, then the matrix is singular.
        if (result <= 0)
            return {lower, false};
        lower(j, j) = sqrt(result);

        auto upperBound = std::min(j + bw, n);
        for (int i = j + 1; i < upperBound; ++i)
        {
            auto result = m(i, j);
            for (int k = std::max(i - bw + 1, 0); k < j; ++k)
            {
                result -= lower(i, k) * lower(j, k);
            }
            result /= lower(j, j);
            lower(i, j) = result;
        }
    }
    return {lower, true};
}
}
```

APPENDIX J
MATRIX-SOLVER.H

```
#pragma once

#include "matrix.h"
#include "cholesky.h"

namespace Numeric {

// Solves the system of equations L * x = b, where L is a non-singular,
// lower-triangular [n x n] matrix, and b is an [n x 1] vector. Note that this
// function assumes that L is lower-triangular and will only use that part of
// the matrix to solve without checking the validity of this assumption.
template <typename T>
Matrix<T> lsolve(const Matrix<T>& lower, const Matrix<T>& b);

// Solves the system of equations U * x = b, where U is a non-singular,
// upper-triangular [n x n] matrix, and b is an [n x 1] vector. Note that this
// function assumes that U is upper-triangular and will only use that part of
// the matrix to solve without checking the validity of this assumption.
template <typename T>
Matrix<T> usolve(const Matrix<T>& upper, const Matrix<T>& b);

// Solves the system of equations m * x = b, where m is a positive-definite
// [n x n] matrix, and b is an [n x 1] vector.
template <typename T>
Matrix<T> solve(const Matrix<T>& m, const Matrix<T>& b)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"solve: matrix must be square"};
    if (b.rows() != m.cols() || b.cols() != 1)
        throw std::runtime_error{"solve: b must be an [nx1] vector"};

    auto p = cholesky(m);
    if (!p.second)
        throw std::runtime_error{"solve: matrix must be positive-definite"};

    auto y = lsolve(p.first, b);
    auto x = usolve(transpose(p.first), y);
    return x;
}

// Solves the system of equations m * x = b, where m is a positive-definite
// [n x n] matrix, and b is an [n x 1] vector.
// This function takes advantage of the sparsity of the matrix to use the
// half-bandwidth in the cholesky decomposition.
template <typename T>
Matrix<T> bsolve(const Matrix<T>& m, const Matrix<T>& b)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"bsolve: matrix must be square"};
    if (b.rows() != m.cols() || b.cols() != 1)
        throw std::runtime_error{"bsolve: b must be an [nx1] vector"};

    auto p = bcholesky(m);
    if (!p.second)
        throw std::runtime_error{"bsolve: matrix must be positive-definite"};
    auto y = lsolve(p.first, b);
    auto x = usolve(transpose(p.first), y);
    return x;
}

template <typename T>
Matrix<T> lsolve(const Matrix<T>& lower, const Matrix<T>& b)
{
    if (lower.rows() != lower.cols())
        throw std::runtime_error{"lsolve: matrix must be square"};
    if (b.rows() != lower.cols() || b.cols() != 1)
        throw std::runtime_error{"lsolve: b must be an [nx1] vector"};

    int n = lower.rows();
    Matrix<T> x{n, 1};
    for (int i = 0; i < n; ++i)
    {
        auto result = b(i);
        for (int j = 0; j < i; ++j)
        {
            result -= lower(i, j) * x(j);
        }
        if (lower(i, i) == 0)
            throw std::runtime_error{"lsolve: matrix must be non-singular"};
        result /= lower(i, i);
        x(i) = result;
    }
    return x;
}

template <typename T>
Matrix<T> usolve(const Matrix<T>& upper, const Matrix<T>& b)
{
    if (upper.rows() != upper.cols())
        throw std::runtime_error{"usolve: matrix must be square"};
    if (b.rows() != upper.cols() || b.cols() != 1)
        throw std::runtime_error{"usolve: b must be an [nx1] vector"};

    int n = upper.rows();
    Matrix<T> x{n, 1};
    for (int i = n - 1; i >= 0; --i)
    {
        auto result = b(i);
        for (int j = n - 1; j >= i + 1; --j)
        {
            result -= upper(i, j) * x(j);
        }
        if (upper(i, i) == 0)
            throw std::runtime_error{"usolve: matrix must be non-singular"};
        result /= upper(i, i);
        x(i) = result;
    }
    return x;
}
}
```

APPENDIX K
INTEGRAL.H

```
#pragma once

#include <vector>
#include <functional>

namespace Numeric {

// Computes the definite integral of f(x) from x = a to x = b using
// Gauss-Legendre integration with n equal segments.
double integral(std::function<double(double)> f, double a, double b, int n);

// Computes the definite integral of f(x) starting from x = a using
// Gauss-Legendre integration with the given segment lengths.
double integral(
    std::function<double(double)> f,
    double a,
    const std::vector<double>& segments);

} // namespace Numeric
```

APPENDIX L
INTEGRAL.CPP

```
#include "integral.h"

namespace Numeric {

double integral(std::function<double(double)> f, double a, double b, int n)
{
    double result = 0.0;
    double dx = (b - a) / n;
    for (int i = 0; i < n; ++i)
    {
        double x0 = a + i * dx;
        double x1 = a + (i + 1) * dx;
        result += (x1 - x0) * f((x0 + x1) / 2);
    }
    return result;
}

double integral(
    std::function<double(double)> f,
    double a,
    const std::vector<double>& segments)
{
    double result = 0.0;
    double x0 = a;
    for (auto segment : segments)
    {
        double x1 = x0 + segment;
        result += (x1 - x0) * f((x0 + x1) / 2);
        x0 = x1;
    }
    return result;
}

} // namespace Numeric
```