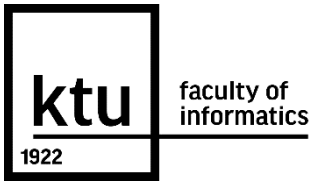


**Faculty of Informatics**
**Department of Information Systems**

# P170B115 Numerical methods and algorithm
# Laboratory work 2

**Lecturer: Assoc. prof. Andrius Kriščiūnas,**
**Dalia ČALNERYTĖ**
**Students: Arsenii Ziubin**

**KAUNAS, 2025**

# Part 1

**Task**

Implement Simple Iteration, QR and Gaussian methods to solve follows systems of linear equations:

$$\begin{cases} 4x_1 + 3x_2 - x_3 + x_4 = 12 \\ 3x_1 + 9x_2 - 2x_3 - 2x_4 = 10 \\ -x_1 - 2x_2 + 11x_3 - x_4 = -28 \\ x_1 - 2x_2 - x_3 + 5x_4 = 16 \end{cases}$$

**Figure 1 First system of linear equations**

$$\begin{cases} x_1 - 2x_2 + 3x_3 + 4x_4 = 11 \\ x_1 - x_3 + x_4 = -4 \\ 2x_1 - 2x_2 + 2x_3 + 5x_4 = 7 \\ -7x_2 + 3x_3 + x_4 = 2 \end{cases}$$

**Figure 2 Second system of linear equations**

$$\begin{cases} 2x_1 + 4x_2 + 6x_3 - 2x_4 = 2 \\ x_1 + 3x_2 + x_3 - 3x_4 = 1 \\ x_1 + x_2 + 5x_3 + x_4 = 7 \\ 2x_1 + 3x_2 - 3x_3 - 2x_4 = 2 \end{cases}$$

**Figure 3 Third system of linear equations**

## Simple Iteration method

### Code of the method

```
D_inv_matrix = np.diag(1.0 / diag_A)

Atld = D_inv_matrix.dot(A) - np.diag(alpha)
btld = D_inv_matrix.dot(b)

x = np.zeros(shape=(n, 1))
x1 = np.zeros(shape=(n, 1))

for it in range(0, nitmax):
    x1 = ((btld-Atld.dot(x)).transpose()/alpha).transpose()
    denominator = np.linalg.norm(x) + np.linalg.norm(x1)

    if denominator < eps:
        prec = 0.0
    else:
        prec = np.linalg.norm(x1 - x) / denominator
    if prec < eps:
        print(f"\nConverged in {it + 1} iterations.")
        x[:] = x1[:]
        break

    x[:] = x1[:]
```

**Figure 4 Simple Iteration method code**

This code implements the Simple Iteration method, an iterative algorithm for solving Ax = b.First, the setup stage (D_inv_matrix, Atld, btld) rearranges the system into an iterative form, which allows calculating a new guess for x from an old one.Then, the iteration loop starts with an initial guess (x). In each step, it calculates a new, refined guess (x1) and measures the relative difference (prec) between the new and old guesses. If this difference is smaller than a tiny tolerance (eps), the solution has converged, and the loop stops. If not, the new guess becomes the current guess, and the process repeats until convergence or the maximum iteration limit (nitmax) is reached.

## First system of linear equations

```
A8 = np.matrix([[4.0, 3.0, -1.0, 1.0],
                [3.0, 9.0, -2.0, -2.0],
                [-1.0, -2.0, 11.0, -1.0],
                [1.0, -2.0, -1.0, 5.0]])
b8 = (np.matrix([12.0, 10.0, -28.0, 16.0])).transpose()
```

**Figure 8**

Results

```
olving System 8 using Simple Iteration

Converged in 74 iterations.

Solution x:
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]

Verification (A*x):
[[ 12.]
 [ 10.]
 [-28.]
 [ 16.]]
Matches original b: True
```

## Second system of linear equations

```
A13 = np.matrix([[1.0, -2.0, 3.0, 4.0],
                 [1.0, 0.0, -1.0, 1.0],
                 [2.0, -2.0, 2.0, 5.0],
                 [0.0, -7.0, 3.0, 1.0]])
b13 = (np.matrix([11.0, -4.0, 7.0, 2.0])).transpose()
```

**Figure 13**

Results

```
olving System 13 using Simple Iteration

ERROR: Zero element found on the diagonal.
The method cannot proceed due to division by zero.
RESULT: METHOD FAILED
```

## Third system of linear equations

```
A20 = np.matrix([[2.0, 4.0, 6.0, -2.0],
                 [1.0, 3.0, 1.0, -3.0],
                 [1.0, 1.0, 5.0, 1.0],
                 [2.0, 3.0, -3.0, -2.0]])
b20 = (np.matrix([2.0, 1.0, 7.0, 2.0])).transpose()
```

**Figure 20**

Results

```
Last computed solution x:
[[nan]
 [nan]
 [nan]
 [nan]]
RESULT: FAILED TO CONVERGE (Diverged)
```

# Gaussian method

## Code of the method

```
try:
    for i in range (0,n-1):

        if np.abs(A1[i, i]) < 1e-12:
            if np.abs(A1[i, n]) > 1e-12:
                print(f"\nERROR")
                print(f"Row {i} shows a contradiction (0 = {A1[i,n]}).")
                print("RESULT: NO SOLUTION")
            else:
                print(f"\nERROR")
                print(f"Row {i} is all zeros (0 = 0).")
                print("RESULT: INFINITELY MANY SOLUTIONS")
            return

        for j in range (i+1,n):
            A1[j,i:n+nb]=A1[j,i:n+nb]-A1[i,i:n+nb]*A1[j,i]/A1[i,i];
            A1[j,i]=0;
```

```
x=np.zeros(shape=(n,nb))
for i in range (n-1,-1,-1):
    sum_ax = np.dot(A1[i, i+1:n], x[i+1:n, :])
    x[i,:] = (A1[i, n:n+nb] - sum_ax) / A1[i,i]
```

**Figure 9 Gaussian method**

This function solves the system Ax = b using Gaussian Elimination. It first performs forward elimination by creating an augmented matrix A1 (combining A and b) and applying row operations to transform it into an upper triangular form. During this process, it strictly checks pivot elements. If a zero pivot is found, it determines if there is NO SOLUTION (a contradiction like 0x = 5) or INFINITELY MANY SOLUTIONS (a redundancy like 0x = 0).If a unique solution exists, the code performs Backward Substitution, iterating from the last row upwards. It solves for each xi using the already-computed values from the rows below. Finally, it prints the solution x and verifies it.

Results

```
Solving System 8
Initial Matrix
[[  4.   3.  -1.   1.  12.]
 [  3.   9.  -2.  -2.  10.]
 [ -1.  -2.  11.  -1. -28.]
 [  1.  -2.  -1.   5.  16.]]

Solution x
[[ 1.]
 [ 1.]
 [-2.]
 [ 3.]]

Verification (A*x)
[[ 12.]
 [ 10.]
 [-28.]
 [ 16.]]
Matches original b: True

Solving System 13
Initial Matrix
[[ 1. -2.  3.  4. 11.]
 [ 1.  0. -1.  1. -4.]
 [ 2. -2.  2.  5.  7.]
 [ 0. -7.  3.  1.  2.]]

ERROR
Row 2 is all zeros (0 = 0).
RESULT: INFINITELY MANY SOLUTIONS

Solving System 20
Initial Matrix
[[ 2.  4.  6. -2.  2.]
 [ 1.  3.  1. -3.  1.]
 [ 1.  1.  5.  1.  7.]
 [ 2.  3. -3. -2.  2.]]

ERROR
Row 2 shows a contradiction (0 = 6.0).
RESULT: NO SOLUTION
```

**Figure 11 Results for first, second and third matrices**

We can see from the result that the 1 system was solved correctly, second has infinite solutions and third has no solutions because of the division by 0.

# QR method

## Code of the method

```
    Q = np.identity(n)

    for i in range (0,n-1):
        z = A[i:n, i:i+1]

        zp = np.zeros_like(z)
        zp[0, 0] = np.linalg.norm(z)

        omega = z - zp

        norm_omega = np.linalg.norm(omega)
        if norm_omega < 1e-12:
            continue

        omega = omega / norm_omega
        Qi = np.identity(n-i) - 2 * np.dot(omega, omega.T)
        A[i:n, :] = Qi.dot(A[i:n, :])
        Q[:, i:n] = Q[:, i:n].dot(Qi)
    R = A
b1 = Q.transpose().dot(b)
x = np.zeros(shape=(n,nb))

for i in range (n-1,-1,-1):
    sum_ax = np.dot(R[i, i+1:n], x[i+1:n, :])
    x[i,:] = (b1[i,:] - sum_ax) / R[i,i]
```

**Figure 12 QR**

This function solves Ax = b using QR Decomposition. It iteratively decomposes the matrix A into an orthogonal matrix Q and an upper-triangular matrix R. This transforms the original

system into the equivalent, easily solvable system Rx = Q^Tb. The code then performs Backward Substitution on this new system to find the solution vector x and includes a singularity check to handle non-unique solutions.

Results:

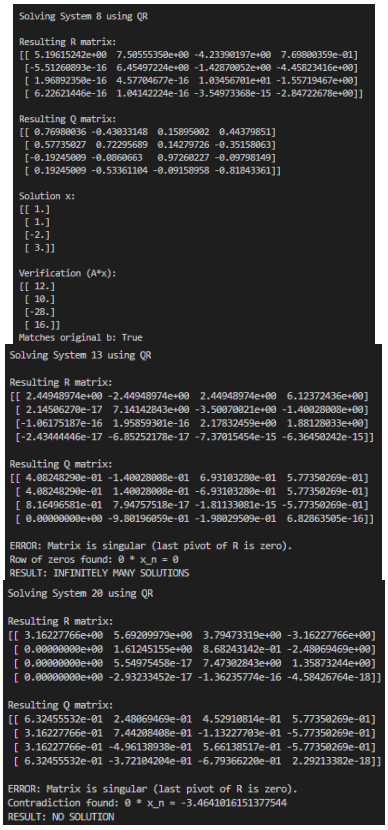


**Figure 14 Results for first, second and third soles**

From the result we see that first matrix was correctly calculated, second has infinite number of solutions and third has no solution because includes zeros.

# Part 2

**Task**
Implement the Newton method to solve the system of non-linear equations.

| | |
|---|---|
| 10 | $\begin{cases} x_1^2 + 2(x_2 - \cos(x_1))^2 - 20 = 0 \\ x_1^2 x_2 - 2 = 0 \end{cases}$ |

**Figure 15 System of non-linear equations**

**Graphical solution**

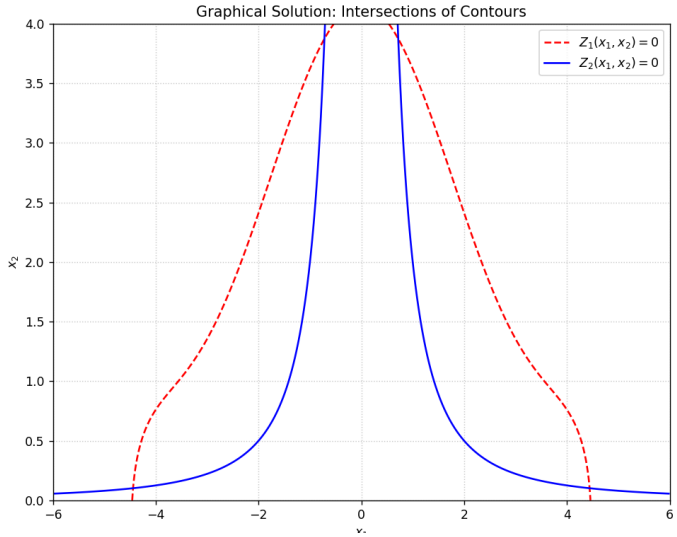


**Figure 16 Graphical solution for system of non-linear equations**

Red line: first equation, Blue line: second equation.
the results are
[-4.44160772  0.10137937]
[-0.71851577  3.8739801 ]
[4.44160772 0.10137937]
[0.71851577 3.87398007]

**Mesh of initial guesses**

To draw the mesh of initial guesses I have used Newton Method.

```
def funct(x1, x2):
    Z1 = x1**2 + 2 * (x2 - np.cos(x1))**2 - 20
    Z2 = x1**2 * x2 - 2
    return np.array([[Z1], [Z2]])
```

**Figure 15 Code of system of non-linear equations**

```
def dfunct(x1, x2):
    dZ1_dx1 = 2*x1 + 4*(x2 - np.cos(x1)) * np.sin(x1)
    dZ1_dx2 = 4*(x2 - np.cos(x1))
    dZ2_dx1 = 2*x1*x2
    dZ2_dx2 = x1**2
    return np.array([
        [dZ1_dx1, dZ1_dx2],
        [dZ2_dx1, dZ2_dx2]
    ])
```

**Figure 16 Code of Jacobian matrix**

```
def Z1(x1, x2):
    return x1**2 + 2 * (x2 - np.cos(x1))**2 - 20

def Z2(x1, x2):
    return x1**2 * x2 - 2
```

**Figure 19**

I divided the system into two separate functions, Z1 and Z2, for plotting the 2D contour graphs.

The Newton-Raphson method is an approach to find the solutions (or "roots") of an equation. It starts with an initial guess. It then looks at the function's slope at that guess to find a new, better guess that should be closer to the actual solution. It repeats this process, getting progressively closer with each step, until the guess is so good that the function's value is almost zero.

```
def newton_method(x1_0, x2_0, max_iter=100, tolerance=1e-6):
    x = np.array([[x1_0], [x2_0]], dtype=float)

    for _ in range(max_iter):
        f_x = funct(x[0,0], x[1,0])
        norm_f = np.linalg.norm(f_x)

        if norm_f < tolerance:
            return x.flatten() # Success

        J_x = dfunct(x[0,0], x[1,0])

        if abs(np.linalg.det(J_x)) < 1e-9:
            return None # Singular matrix

        x = x - np.linalg.inv(J_x) @ f_x

    return None # Failed to converge
```

**Figure 20 Code of the Newton method**

Inside the loop, it first calls funct to check the current error—that is, how far the guess is from being a solution. If this error is smaller than the tolerance, the guess is considered a success and is returned.

If the error is still too large, the function calls dfunct to get the Jacobian matrix, which is the system's equivalent of a slope. It checks if this Jacobian is valid (not singular). If it isn't, the method can't continue and stops.
If the Jacobian is valid, the code uses it along with the current error to calculate the next, improved guess. This process repeats until a solution is found or the loop hits its maximum iterations, in which case it reports failure.

```
for i, sol in enumerate(solutions):
    color = colors[i % len(colors)]
    basin_x = []
    basin_y = []

    for x1 in x1_vals:
        for x2 in x2_vals:
            sol_i = newton_method(x1, x2)
            if sol_i is not None and np.linalg.norm(sol_i - sol) < convergence_tolerance:
                basin_x.append(x1)
                basin_y.append(x2)

    plt.plot(basin_x, basin_y, color + 'o', markersize=8)
    label_text = f"Solution {i+1}: [{sol[0]:.4f} {sol[1]:.4f}]"
    plt.plot(sol[0], sol[1], color + 's', markersize=15, markeredgecolor='black')
    legend_handles.append(plt.Line2D([0], [0], marker='o', color=color, label=label_text, markersize=8, linestyle=''))

non_conv_x = []
non_conv_y = []
for x1 in x1_vals:
    for x2 in x2_vals:
        if newton_method(x1, x2) is None:
            non_conv_x.append(x1)
            non_conv_y.append(x2)
```

**Figure 21 Mesh of initial guesses code**

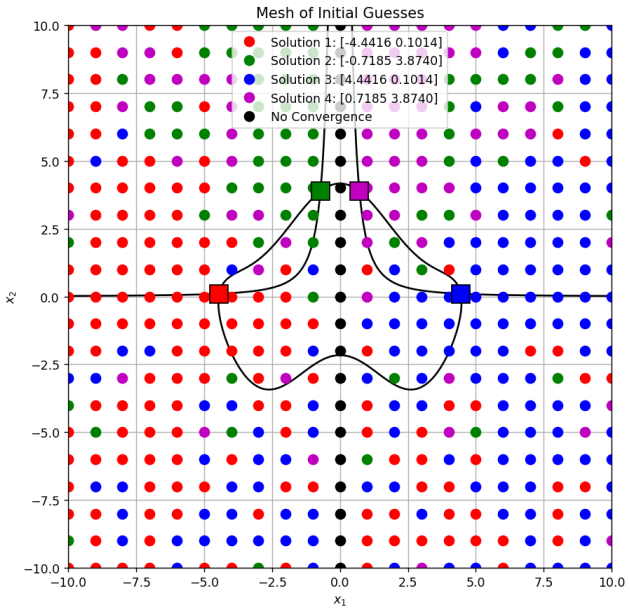


**Figure 22 Mesh of initial guesses**

From the picture 22 we can see that Newton method correctly found 4 different solutions to the system. The black points show where Newton's method could not find a solution, meaning the method did not converge for those values.

### Calculated solutions of the system in a table

```
Calculated Solutions (Newton's Method)
================================================
Solution      | Calculated Solution (x1, x2)
1             | (-4.441608, 0.101379)
2             | (-0.718516, 3.873980)
3             | (4.441608, 0.101379)
4             | (0.718516, 3.873980)
```

**Figure 23 Table with calculated solutions**

### Verifying the obtained results using python fsolve()

Obtained results were verified with python function fsolve() from scipy.optimize library.

```python
def fsolve_func(x):
    x1 = x[0]
    x2 = x[1]
    Z1 = x1**2 + 2 * (x2 - np.cos(x1))**2 - 20
    Z2 = x1**2 * x2 - 2
    return [Z1, Z2]

print("\nVerifying Solutions with scipy.optimize.fsolve")
for idx, sol in enumerate(solutions):
    fsolve_sol = fsolve(fsolve_func, sol)
    print(f"Solution {idx+1}:")
    print(f"  Our Newton Result:  ({sol[0]:.7f}, {sol[1]:.7f})")
    print(f"  scipy.fsolve Result: ({fsolve_sol[0]:.7f}, {fsolve_sol[1]:.7f})")
    print(f"  Match: {np.allclose(sol, fsolve_sol)}")
    print("-" * 30)
```

**Figure 24 Python code for fsolve() verification**

```
Verifying Solutions with scipy.optimize.fsolve
Solution 1:
  Our Newton Result:  (-4.4416077, 0.1013794)
  scipy.fsolve Result: (-4.4416077, 0.1013794)
  Match: True
------------------------------
Solution 2:
  Our Newton Result:  (-0.7185158, 3.8739801)
  scipy.fsolve Result: (-0.7185158, 3.8739801)
  Match: True
------------------------------
Solution 3:
  Our Newton Result:  (4.4416077, 0.1013794)
  scipy.fsolve Result: (4.4416077, 0.1013794)
  Match: True
------------------------------
Solution 4:
  Our Newton Result:  (0.7185158, 3.8739801)
  scipy.fsolve Result: (0.7185158, 3.8739801)
  Match: True
------------------------------
```

**Figure 25 fsolve() results and comparison with With my results**

# Part 3

### Task description

According to the scheme provided in Table 8 (see Table 7 for individual schema number), implement the architecture of an artificial neural network for predicting real estate (apartment) prices based on historical data (provided in TSV format in the file “data_for_optimization_task2.tsv”). Using the optimization method indicated in Table 7, find such values of the weight vector W that the mean square error (MSE) of the price predictions on the dataset provided in “data_for_optimization_task2.tsv” is as small as possible.

```
while True:
    W_new = W - temp_step * gradient
    mse_new = funct_mse(W_new, X_norm, y_norm)

    if mse_new < mse_old:
        W = W_new
        step = temp_step * 1.2
        break

    temp_step = temp_step / 2.0

    if temp_step < 1e-12:
        W = W_new
        break

if temp_step < 1e-12:
     print(f"\nStep size too small. Stopping at iteration {i}.")
     break
```

**Figure 26 fastest gradient descend method**

Fastest Gradient Descent is an optimization algorithm used to find the minimum value of a function, in my case, the Mean Square Error (MSE).
It works by iteratively updating a set of parameters (the weights W). In each iteration, it first calculates the gradient, which is a vector that points in the direction of the "steepest uphill climb" for the error. To minimize the error, the algorithm moves in the exact opposite direction.
What makes it the "Fastest" method is how it chooses its step size. Instead of using a small, fixed step (like standard Gradient Descent), it performs a search in each iteration to find a step size that causes the largest possible decrease (or at least a guaranteed decrease) in the error.

```
for i in range(max_iter):
    # Calculate current errors
    mse_old = funct_mse(W, X_norm, y_norm)
    mae_old = funct_mae(W, X_norm, y_norm)
    mse_history.append(mse_old)
    mae_history.append(mae_old)

    if (i % 50 == 0) or (i == max_iter - 1):
        print(f"Iteration {i:4d}: MSE = {mse_old:.6f}, MAE = {mae_old:.6f}")

    gradient = grad(W, X_norm, y_norm)

    if np.linalg.norm(gradient) < tol:
        print(f"\nConvergence reached at iteration {i}.")
        break

    # Find the "fastest" step (Backtracking Line Search)
    temp_step = step
    while True:
        W_new = W - temp_step * gradient
        mse_new = funct_mse(W_new, X_norm, y_norm)

        if mse_new < mse_old:
            W = W_new
            step = temp_step * 1.2
            break

        temp_step = temp_step / 2.0

        if temp_step < 1e-12:
```

**Figure 27 main logic**

The program's core logic is a large optimization loop that runs for a set number of iterations. The goal of this loop is to progressively improve the network's weights W to minimize the Mean Square Error.
In each iteration, this process begins by calling the grad function to calculate the gradient. The gradient is a vector that points in the direction where the error increases the most. To reduce the error, the algorithm must move in the exact opposite direction.
This is where the "Fastest" Gradient Descent method is implemented. Instead of moving by a fixed amount, the code enters a secondary while loop to find the best possible step size. It "tries" a step in the downhill direction to get a new set of weights. It then checks if the error for these new weights is actually lower than the old error.
If the step was successful and the error decreased, the program accepts these new, better weights and continues to the next main iteration. If the step was bad and the error did not decrease, it means the step was too large. The program rejects the step, cuts the step size in half, and tries again. This search continues until it finds a step size that guarantees the error goes down, ensuring the most efficient path to the best solution.

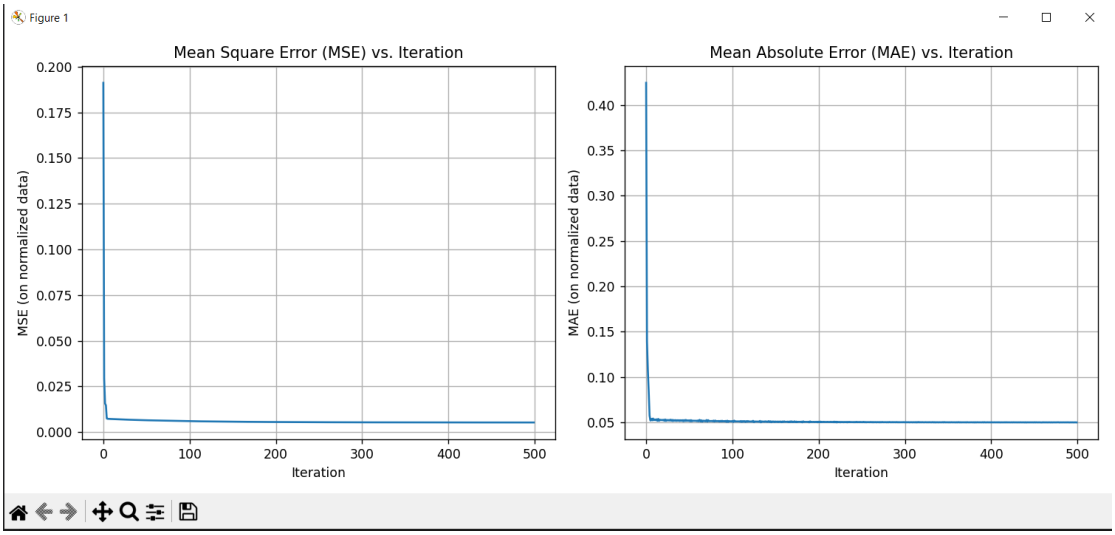


**Figure 28**

These graphs show how the network's prediction error improved during the training process. The horizontal axis represents each step, or "Iteration," of the optimization, starting from the initial guess at Iteration 0. The vertical axis shows the average error (MSE and MAE) across all the data.

```
Starting optimization
Iteration    0: MSE = 0.191196, MAE = 0.424340
Iteration   50: MSE = 0.006510, MAE = 0.052377
Iteration  100: MSE = 0.006011, MAE = 0.051611
Iteration  150: MSE = 0.005711, MAE = 0.051099
Iteration  200: MSE = 0.005537, MAE = 0.050854
Iteration  250: MSE = 0.005434, MAE = 0.050547
Iteration  300: MSE = 0.005372, MAE = 0.050425
Iteration  350: MSE = 0.005335, MAE = 0.050291
Iteration  400: MSE = 0.005312, MAE = 0.050267
Iteration  450: MSE = 0.005299, MAE = 0.050291
Iteration  499: MSE = 0.005291, MAE = 0.050081
Optimization finished.
```

**Figure 29**

This is a summary log of program's learning process from start to finish.
"Iteration 0" shows the error at the very beginning, using the initial untrained weights. This is why the MSE (0.191) and MAE (0.424) are at their highest values.
By "Iteration 50", there is a huge, dramatic drop in both errors. This shows the optimization method worked very effectively, finding a much better set of weights almost immediately.

```
Report 2: Error Values (Before vs. After)
Metric  Before Optimization  After Optimization
   MSE             0.191196            0.005291
   MAE             0.424340            0.050081
```

**Figure 30**

It shows the error values at the very beginning (Before Optimization) compared to the final error values at the very end (After Optimization).

```
Report 4: Price Predictions (First 10 Objects)
   LotArea  OverallQual  YearBuilt  SalePrice  Predicted (Initial W)  Predicted (Optimized W)
0     8450            7       2003     208500             579,647.89                208,433.47
1     9600            6       1976     181500             488,318.58                182,958.92
2    11250            7       2001     223500             587,724.98                212,338.55
3     9550            7       1915     140000             400,783.33                214,506.52
4    14260            8       2000     250000             638,813.73                244,917.50
5    14115            5       1993     143000             503,551.88                159,762.45
6    10084            8       2004     307000             628,890.36                239,040.41
7    10382            7       1973     200000             525,484.09                212,617.45
8     6120            7       1931     129900             419,171.09                209,017.61
9     7420            5       1939     118000             361,546.18                153,484.34
PS D:\NUMERICAL METHODS AND ALGORITHMS\NumericalMethods_Project2>
```

**Figure 31**

This report table shows model's practical performance by comparing its predictions against the actual prices for the first 10 houses in dataset.

The SalePrice column is the true, correct price for each house.

The Predicted (Initial W) column shows the model's predictions before any optimization. You can see these guesses are very bad; for example, it predicted $579,647 for a house that actually cost $208,500.

The Predicted (Optimized W) column shows the model's predictions after the optimization was finished. These numbers are much closer to the actual SalePrice. For that same first house, the final prediction of $208,433 is extremely accurate.