Numerical analysis laboratory work №3

Ivan Zhytkevych FI-91

System

$$A = \begin{bmatrix} 2.12 & 0.42 & 1.34 & 0.88 \\ 0.42 & 3.95 & 1.87 & 0.43 \\ 1.34 & 1.87 & 2.98 & 0.46 \\ 0.88 & 0.43 & 0.46 & 4.44 \end{bmatrix}$$

$$b = \begin{bmatrix} 11.172 & 0.115 & 0.009 & 9.349 \end{bmatrix}$$

Program description

The program can calculate the matrix with diagonal dominance basing on the input matrix. The method that does this:

```
func (m Matrix) ToDiagonalDominance() *Matrix
```

The idea behind this method is to replace columns to create the matrix with the biggest elements on the diagonal.

Then iteratively find the best coefficient among other rows to minimize all elements of target row by absolute value except the element on diagonal. Because the biggest elements by absolute value are placed on diagonal we can surely minimize all other elements in that column.

At last we apply simple iteration method to solve the given system of linear equations.

Program output

```
$ go build && ./lab3
```

```
---------------------------------------------------------------
|                                                             |
|   2.120000   0.420000   1.340000   0.880000 |   11.172000   |
|   0.420000   3.950000   1.870000   0.430000 |    0.115000   |
|   1.340000   1.870000   2.980000   0.460000 |    0.009000   |
|   0.880000   0.430000   0.460000   4.440000 |    9.349000   |
|_____________________________________________________________|
Matrix is not with diagonal dominance
Row 0 closeness -0.520000
```

```
Coef after optimization: -0.10650634765625
Coef after optimization: -0.44952392578125
Coef after optimization: -0.19805908203125
Closeness values after merging i-th rows to 0 row: [0 0.09953186035156247 0.423390
5029296876 0.3613629150390627]
Closeness with maximum value is 2-th closeness: 0.423391
Coefficient: -0.449524
Linear rows sum: [1.5176379394531252 -0.4206097412109376 0.00041870117187503553 0.
673218994140625]
Row 1 closeness 1.230000
Row 2 closeness -0.690000
Coef after optimization: -0.88287353515625
Coef after optimization: -0.47332763671875
Coef after optimization: -0.103759765625
Closeness values after merging i-th rows to 2 row: [0.6038002700731155 0.696849975
5859372 0 -0.14249755859375002]
Closeness with maximum value is 1-th closeness: 0.696850
Coefficient: -0.473328
Linear rows sum: [1.1412023925781252 0.0003558349609376066 2.0948773193359376 0.25
64691162109375]
Row 3 closeness 2.670000
Average closeness: 1.255060
Current matrix:
-----------------------------------------------------------------------
|                                                                     |
|   1.51763794  -0.42060974   0.00041870   0.67321899 |   11.16795428  |
|   0.42000000   3.95000000   1.87000000   0.43000000 |    0.11500000  |
|   1.14120239   0.00035583   2.09487732   0.25646912 |   -0.04543268  |
|   0.88000000   0.43000000   0.46000000   4.44000000 |    9.34900000  |
|_____________________________________________________________________|
-----------------------------------------------------------------------
|                                                                     |
|   1.51763794  -0.42060974   0.00041870   0.67321899 |   11.16795428  |
|   0.42000000   3.95000000   1.87000000   0.43000000 |    0.11500000  |
|   1.14120239   0.00035583   2.09487732   0.25646912 |   -0.04543268  |
|   0.88000000   0.43000000   0.46000000   4.44000000 |    9.34900000  |
|_____________________________________________________________________|
Using Simple iteration...
Iteration 0 with solution:
-----------------
|               |
|   7.35877379  |
|   0.02911392  |
|  -0.02168751  |
|   2.10563063  |
|_______________|
```

```
And residual:
 ----------------
|                |
|  -1.40529585   |
|  -3.95555051   |
|  -8.93788984   |
|  -6.47826366   |
|________________|
Iteration 1 with solution:
 ----------------
|                |
|   6.43279808   |
|  -0.97229127   |
|  -4.28823322   |
|   0.64656224   |
|________________|
And residual:
 ----------------
|                |
|   0.56285818   |
|   8.99474968   |
|   1.43128800   |
|   3.20807388   |
|________________|
Iteration 2 with solution:
 ----------------
|                |
|   6.80367586   |
|   1.30486055   |
|  -3.60500085   |
|   1.36910140   |
|________________|
And residual:
 ----------------
|                |
|   0.47107908   |
|  -1.74410504   |
|  -0.60936588   |
|  -1.61983462   |
|________________|
Iteration 3 with solution:
 ----------------
|                |
|   7.11407867   |
|   0.86331497   |
|  -3.89588465   |
```

```
|   1.00427378  |
|______________|
And residual:
________________
|              |
|   0.06001230  |
|   0.57045940  |
|  -0.26050829  |
|   0.05051668  |
|______________|
Iteration 4 with solution:
________________
|              |
|   7.15362190  |
|   1.00773507  |
|  -4.02023957  |
|   1.01565141  |
|______________|
And residual:
________________
|              |
|   0.05313693  |
|   0.21104316  |
|  -0.04809623  |
|  -0.03969542  |
|______________|

...........................

Iteration 17 with solution:
________________
|              |
|   7.22006156  |
|   1.08330843  |
|  -4.07651471  |
|   0.99205454  |
|______________|
And residual:
________________
|              |
|   0.00000189  |
|   0.00000490  |
|  -0.00000284  |
|  -0.00000221  |
|______________|
Iteration 18 with solution:
```

```
----------------
|              |
|   7.22006281  |
|   1.08330967  |
|  -4.07651606  |
|   0.99205405  |
|______________|
And residual:
----------------
|              |
|   0.00000086  |
|   0.00000223  |
|  -0.00000129  |
|  -0.00000100  |
|______________|
Iteration 19 with solution:
----------------
|              |
|   7.22006337  |
|   1.08331023  |
|  -4.07651668  |
|   0.99205382  |
|______________|
And residual:
----------------
|              |
|   0.00000039  |
|   0.00000101  |
|  -0.00000059  |
|  -0.00000046  |
|______________|
Iteration 20 with solution:
----------------
|              |
|   7.22006363  |
|   1.08331049  |
|  -4.07651696  |
|   0.99205372  |
|______________|
And residual:
----------------
|              |
|   0.00000018  |
|   0.00000046  |
|  -0.00000027  |
|  -0.00000021  |
```

```
|_______________|
Solution:
-----------------
|               |
|   7.22006363  |
|   1.08331049  |
|  -4.07651696  |
|   0.99205372  |
|_______________|
```

The solution coincides with the one we got using `SolveSQRTMethod`

$$\begin{pmatrix} 7.22006363 \\ 1.08331049 \\ -4.07651696 \\ 0.99205372 \end{pmatrix}$$

Code

main.go

```go
package main

import "fmt"

func main() {
    M := NewMatrix([][]float64{
        {2.12, 0.42, 1.34, 0.88},
        {0.42, 3.95, 1.87, 0.43},
        {1.34, 1.87, 2.98, 0.46},
        {0.88, 0.43, 0.46, 4.44},
    })

    b := NewMatrix([][]float64{
        { 11.172 },
        {  0.115 },
        {  0.009 },
        {  9.349 },
    })

    M.AddExtention(b)

    fmt.Println(M)


    OutputPrecision = 8
```

```go
    var permutations *Matrix
    M, permutations = M.ToDiagonalDominance()
    fmt.Println(M)

    fmt.Println("Using Simple iteration...")

    solution := M.SolveSimpleIteration(0.000001)

    solution = solution.ApplyPermutations(permutations)

    fmt.Println("Solution:")
    fmt.Println(solution)
}
```

optimization.go

```go
package main

func divideByHalfOptimization(f func(float64) float64, a, b, eps float64) float64 {
    l := b - a
    x := (a + b) / 2
    var x1, x2, xValue, x1Value, x2Value float64
    for {
        x1 = a + l/4
        x2 = b - l/4
        xValue = f(x)
        x1Value = f(x1)
        x2Value = f(x2)

        if x1Value < xValue {
            b = x
            x = x1
        } else if x2Value < xValue {
            a = x
            x = x2
        } else {
            a = x1
            b = x2
        }

        l = b - a
        if l <= eps {
            break
        }
    }
    return x
```

```
}
```

matrix.go

```go
package main

import (
    "fmt"
    "math"
    "strings"
)

type Matrix struct {
    data [][]float64
    m uint // rows
    n uint // columns

    extention *Matrix
}

var OutputPrecision = 6

func MatrixInit(m, n uint) *Matrix {
    M := new(Matrix)
    M.m = m
    M.n = n
    M.data = make([][]float64, m)
    for mi := range M.data {
        M.data[mi] = make([]float64, n)
    }
    return M
}

func NewMatrix(m [][]float64) *Matrix {
    if len(m) == 0 {
        return MatrixInit(0, 0)
    }
    n := len(m[0])
    M := MatrixInit(uint(len(m)), uint(n))
    for i := range m {
        copy(M.data[i], m[i])
    }
    return M;
}

func (m *Matrix) AddExtention(ext *Matrix) *Matrix {
```

```
    m.extention = ext
    return m
}

func (m Matrix) AbsMax() (float64) {
    if len(m.data) == 0 { return 0 }
    if len(m.data[0]) == 0 { return 0 }
    max := math.Abs(m.data[0][0])
    for _, row := range m.data {
        for _, item := range row {
            if math.Abs(item) > max {
                max = math.Abs(item)
            }
        }
    }
    return max
}

func (m Matrix) String() (s string) {
    mainLength := len(strings.Split(fmt.Sprint(m.AbsMax()), ".")[0]) + OutputPrecision + 3
    mainLengthStr := fmt.Sprint(mainLength)
    outputPrecisionStr := fmt.Sprint(OutputPrecision)
    format := "% " + mainLengthStr + "." + outputPrecisionStr + "f"
    s += "\n"
    for i, row := range m.data {
        s += fmt.Sprintf("| ")
        for _, item := range row {
            s += fmt.Sprintf(format, item)
            s += " "
        }
        if m.extention != nil {
            mainLength := len(strings.Split(fmt.Sprint(m.extention.AbsMax()), ".")[0]) + Out
            mainLengthStr := fmt.Sprint(mainLength)
            outputPrecisionStr := fmt.Sprint(OutputPrecision)
            format := "% " + mainLengthStr + "." + outputPrecisionStr + "f"
            s += "| "
            for _, item := range m.extention.data[i] {
                s += fmt.Sprintf(format, item)
                s += " "
            }
        }
        s += fmt.Sprintf(" |\n")
    }
    headingLen := len(strings.Split(s, "\n")[1])
    s = strings.Repeat("_", headingLen) + "\n|" + strings.Repeat(" ", headingLen - 2) + "|"
    s += "|" + strings.Repeat("_", headingLen - 2) + "|"
```

```
    return s
}

func (m Matrix) ApplyPermutations(permutations *Matrix) *Matrix {
    result := NewMatrix(m.data)
    if m.extention != nil {
        result.AddExtention(m.extention)
    }
    for i, row := range permutations.data {
        for _, elem := range row {
            result.RearrangeRows(i, int(elem))
        }
    }
    return result
}

func (m Matrix) RowAbsMax(row int) (float64, int) {
    ind := 0
    max := math.Abs(m.data[row][ind])
    for i, item := range m.data[row] {
        if math.Abs(item) > max {
            max = math.Abs(item)
            ind = i
        }
    }
    return max, ind
}

func (m Matrix) RearrangeColumns(i, j int) {
    for _, row := range m.data {
        row[i], row[j] = row[j], row[i]
    }
}

func (m Matrix) RearrangeRows(i, j int) {
    m.data[i], m.data[j] = m.data[j], m.data[i]
}

func (m Matrix) ReorderOnlyToDiagonal() (*Matrix, *Matrix) {
    mt := NewMatrix(m.data)
    if m.extention != nil {
        mt.extention = NewMatrix(m.extention.data)
    }

    pertts := make([][]float64, mt.n)
    for i := range pertts {
```

```go
        pertts[i] = []float64{float64(i)}
    }
    permutations := NewMatrix(pertts)

    var rowMaxIdx int
    var elem float64
    var matrixMaxElem float64
    for i := range mt.data {
        elem, rowMaxIdx = mt.RowAbsMax(i)
        if rowMaxIdx <= i && elem > matrixMaxElem {
            matrixMaxElem = elem
        } else if rowMaxIdx <= i {
            continue
        }
        mt.RearrangeColumns(i, rowMaxIdx)
        permutations.RearrangeRows(i, rowMaxIdx)
    }
    return mt, permutations
}

func (m Matrix) IsWithDiagonalDominance() bool {
    absSumExcept := func (except int, row []float64) float64 {
        s := float64(0)
        for i, item := range row {
            if i == except { continue }
            s += math.Abs(item)
        }
        return s
    }
    for i, row := range m.data {
        if row[i] <= absSumExcept(i, row) {
            return false
        }
    }
    return true
}

func (m Matrix) computeLinearRowsSum(coef float64, targetRowIdx, sourceRowIdx int) []float64
    resultRow := make([]float64, m.n)
    copy(resultRow, m.data[targetRowIdx])
    for i, item := range resultRow {
        resultRow[i] = coef * m.data[sourceRowIdx][i] + item
    }
    return resultRow
}
```

```go
func (m *Matrix) LinearRowSum(coef float64, targetIdx, sourceIdx int) {
    m.data[targetIdx] = m.computeLinearRowsSum(coef, targetIdx, sourceIdx)
    if m.extention != nil {
        m.extention.data[targetIdx] = m.extention.computeLinearRowsSum(coef, targetIdx, sour
    }
}

func (m Matrix) ToDiagonalDominance() (*Matrix, *Matrix) {
    mt, permutations := m.ReorderOnlyToDiagonal()

    closeness := func (i int, row []float64) float64 {
        main := math.Abs(row[i])
        sum := float64(0)
        for j := range row {
            if j != i {
                sum += math.Abs(row[j])
            }
        }
        return main - sum
    }


    closenessAfterMerge := func (coef float64, targetRowIdx, sourceRowIdx int) float64 {
        resultRow := mt.computeLinearRowsSum(coef, targetRowIdx, sourceRowIdx)
        return closeness(targetRowIdx, resultRow)
    }


    mergeBestToRow := func (i int) bool {
        closenessValues := make([]float64, mt.n)
        var maxClosenessIdx int
        var coefficient float64

        var coef float64
        for j := 0; uint(j) < mt.m; j++ {
            if j == i { continue }

            // calculate coefficient (using optimization methods)
            coef = divideByHalfOptimization(func (x float64) float64 {
                return -closenessAfterMerge(x, i, j)
            }, -10, 10, 0.001)
            fmt.Println("Coef after optimization:", coef)

            closenessValues[j] = closenessAfterMerge(coef, i, j)
            if closenessValues[j] > closenessValues[maxClosenessIdx] {
                maxClosenessIdx = j
```

```
                coefficient = coef
            }
        }

        fmt.Printf("Closeness values after merging i-th rows to %d row: %v\n", i, closenessV
        fmt.Printf("Closeness with maximum value is %d-th closeness: %f\n", maxClosenessIdx,
        fmt.Printf("Coefficient: %f\n", coefficient)
        linearSum := mt.computeLinearRowsSum(coefficient, i, maxClosenessIdx)
        fmt.Printf("Linear rows sum: %v\n", linearSum)
        mt.LinearRowSum(coefficient, i, maxClosenessIdx)

        fmt.Print()

        return false
    }

    avgCloseness := func () float64 {
        c := float64(0)
        for i, row := range mt.data {
            c += closeness(i, row)
        }
        return c / float64(mt.n)
    }

    for !mt.IsWithDiagonalDominance() {
        fmt.Println("Matrix is not with diagonal dominance")
        for i, row := range mt.data {
            fmt.Printf("Row %d closeness %f\n", i, closeness(i, row))
            if (closeness(i, row) <= 0) {
                mergeBestToRow(i)
            }
        }
        fmt.Printf("Average closeness: %f\n", avgCloseness())
        fmt.Println("Current matrix:")
        fmt.Println(mt)
    }

    return mt, permutations
}

func (m Matrix) SolveSimpleIteration(eps float64) *Matrix {
    if !m.IsWithDiagonalDominance() {
        return nil
    }

    computeMatrixC := func () *Matrix {
```

```go
    c := MatrixInit(m.m, m.n)
    for i, row := range c.data {
        for j := range row {
            if i == j { continue }
            c.data[i][j] = - m.data[i][j] / m.data[i][i]
        }
    }
    return c
}

computeMatrixD := func () *Matrix {
    d := MatrixInit(m.m, 1)
    for i, row := range m.extention.data {
        for j := range row {
            d.data[i][j] = m.extention.data[i][j] / m.data[i][i]
        }
    }
    return d
}

computeQ := func (c *Matrix) float64 {
    var q float64
    var sum float64
    for _, row := range c.data {
        sum = 0
        for _, elem := range row {
            sum += math.Abs(elem)
        }
        if sum > q {
            q = sum
        }
    }
    return q
}


C := computeMatrixC()
D := computeMatrixD()
Q := computeQ(C)

convergenceCriteria := func (x *Matrix, xPrev *Matrix) bool {
    var maxDiff float64
    var diff float64
    for i, row := range x.data {
        for j := range row {
            diff = math.Abs(x.data[i][j] - xPrev.data[i][j])
```

```
                if diff > maxDiff {
                    maxDiff = diff
                }
            }
        }
        return (Q / (1 - Q)) * maxDiff < eps
    }

    computeNextX := func (x *Matrix) *Matrix {
        return C.Dot(x).Plus(D)
    }

    solution := MatrixInit(m.m, 1)
    var previousSolution *Matrix

    iteration := 0

    for {
        previousSolution = solution
        solution = computeNextX(solution)

        fmt.Printf("Iteration %d with solution:\n", iteration)
        fmt.Println(solution)

        fmt.Println("And residual:")
        fmt.Println(m.Residual(solution))

        if convergenceCriteria(solution, previousSolution) {
            break
        }

        iteration++
    }

    return solution
}

func (m Matrix) Residual(solution *Matrix) *Matrix {
    if m.extention == nil {
        return nil
    }
    return m.extention.Minus(m.Dot(solution))
}

// -----------------------------------------
//          PART OF 2nd LAB
```

```go
// ----------------------------------------

func (M Matrix) Factorization() *Matrix {
    if M.m != M.n { return nil }

    C := MatrixInit(M.m, M.n)

    // c_{ii} = sqrt{ a_{ii} - sum_{k=1}^{i-1} c_{ki}^2 }     i = \overline{2, n}
    calc_diag := func(i uint) float64 {
        var cSum float64 = 0
        for k := uint(0); k < i; k++ {
            cSum += math.Pow(C.data[k][i], 2)
        }
        return math.Sqrt(M.data[i][i] - cSum)
    }

    calc_other := func(i, j uint) float64 {
        var cSum float64
        for k := uint(0); k < i; k++ {
            cSum += C.data[k][i] * C.data[k][j]
        }
        return (M.data[i][j] - cSum) / C.data[i][i]
    }

    // c_{11} = \sqrt{a_{11}}
    C.data[0][0] = math.Sqrt(M.data[0][0])

    // c_{1j} = \frac {a_{1j}} {c_{11}}     j = \overline{2, n}
    for j := uint(1); j < M.n; j++ {
        if C.data[0][0] == 0 { return nil }
        C.data[0][j] = M.data[0][j] / C.data[0][0]
    }

    // fmt.Println("C after c_11 and c_1j inits:", C)

    for i := uint(1); i < C.m; i++ {
        for j := uint(0); j < C.n; j++ {
            fmt.Printf("C on %d,%d step", i, j)
            fmt.Println(C)
            if j < i {
                C.data[i][j] = 0
            } else if j == i {
                C.data[i][j] = calc_diag(i)
            } else if j > i {
                C.data[i][j] = calc_other(i, j)
            }
```

```
        }
    }
    return C
}

func (M *Matrix) SolveSQRTMethod(b *Matrix) *Matrix {
    if b.m != M.m { return nil }
    if b.n != 1   { return nil }
    if M.m != M.n { return nil }

    n := M.m

    C := M.Factorization()
    if C == nil { return nil }

    y := MatrixInit(n, 1)
    // y_1 = b_1 / c_11
    y.data[0][0] = b.data[0][0] / C.data[0][0]

    var cSum float64;
    for i := uint(1); i < n; i++ {
        cSum = 0
        for k := uint(0); k < i; k++ {
            cSum += C.data[k][i] * y.data[k][0]
        }
        y.data[i][0] = (b.data[i][0] - cSum) / C.data[i][i]
    }

    x := MatrixInit(n, 1)
    x.data[n-1][0] = y.data[n-1][0] / C.data[n-1][n-1]

    for i := uint(n-2); i < n-1; i-- {
        cSum = 0
        for k := uint(i+1); k < n; k++ {
            cSum += C.data[i][k] * x.data[k][0]
        }
        x.data[i][0] = (y.data[i][0] - cSum) / C.data[i][i]
    }

    return x
}

func (M Matrix) Dot(A *Matrix) *Matrix {
    if M.n != A.m { return nil }
    R := MatrixInit(M.m, A.n)
    for i := uint(0); i < M.m; i++ {
```

```go
        for j := uint(0); j < A.n; j++ {
            for k := uint(0); k < M.n; k++ {
                R.data[i][j] += M.data[i][k] * A.data[k][j]
            }
        }
    }
    return R
}

func (M Matrix) Plus(A *Matrix) *Matrix {
    if M.m != A.m { return nil }
    if M.n != A.n { return nil }

    R := MatrixInit(M.m, A.n)
    for i := uint(0); i < M.m; i++ {
        for j := uint(0); j < A.n; j++ {
            R.data[i][j] = M.data[i][j] + A.data[i][j]
        }
    }
    return R
}

func (M Matrix) Minus(A *Matrix) *Matrix {
    if M.m != A.m { return nil }
    if M.n != A.n { return nil }

    R := MatrixInit(M.m, A.n)
    for i := uint(0); i < M.m; i++ {
        for j := uint(0); j < A.n; j++ {
            R.data[i][j] = M.data[i][j] - A.data[i][j]
        }
    }
    return R
}
```