Министерство образования Республики Беларусь  
Учреждения образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе 2

студент гр. 853504  
Лукашёнок В.Д.

Минск 2021

**Лабораторная работа 2. Вариант 19.**

Конспектировал на лабораторной, совершаю действие в действие, как было продемонстрировано на лр. Всё ещё написано на golang. Если в одной папке в файлике 1.go вначале указан package с одним названием, то из файла 2.go можно вызывать функции 1.go без прямого импорта этого файла. Как и просили, нахождение обратной базисной матрицы на первой итерации происходит через встроенную функцию gonum, все остальные разы нахожу оптимизированным алгоритмом из первой лабораторной (функция invOptimized). Чтобы код был более читаем и не пришлось писать много повторяющегося кода, замкнул всё в рекурсии, в которую последним параметром передаю номер итерации (из-за этого каждую итерацию дальше приходится делать лишнюю проверку, чтобы обратная матрица считалась не встроенной в gonum функцией). Код и первые две итерации вставил и в документ и в архив, который отправил.

**Код программы**

**2.go**

package main

import (

"fmt"

"io/ioutil"

"math"

"strconv"

"strings"

"gonum.org/v1/gonum/mat"

)

func readVector(lines []string, number int) ([]string, \*mat.VecDense) {

vector := make([]float64, number)

vectorValues := strings.Split(lines[0], " ")

for i := 0; i < number; i++ {

number, \_ := strconv.ParseFloat(vectorValues[i], 64)

vector[i] = number

}

return lines[1:], mat.NewVecDense(number, vector)

}

// Must be in canonical form

func readOptimizationProblem(input string, varNumber, conditionsNumber int) (\*mat.VecDense, \*mat.Dense, \*mat.VecDense, \*mat.VecDense, \*mat.VecDense) {

var (

scalesVector \*mat.VecDense

conditionsMatrix []float64

freeMembersVector \*mat.VecDense

baselineVector \*mat.VecDense

baselineIndexes \*mat.VecDense

)

//opening file

str, err := ioutil.ReadFile(input)

if err != nil {

panic(err)

}

lines := strings.Split(string(str), "\n")

//scalesVector

lines, scalesVector = readVector(lines, varNumber)

//conditionsMatrix

for i := 0; i < conditionsNumber; i++ {

conditionsMatrixNumbers := strings.Split(lines[i], " ")

for j := 0; j < varNumber; j++ {

number, \_ := strconv.ParseFloat(conditionsMatrixNumbers[j], 64)

conditionsMatrix = append(conditionsMatrix, number)

}

}

lines = lines[conditionsNumber:]

//freeMembersVector

lines, freeMembersVector = readVector(lines, conditionsNumber)

//baselineVector

lines, baselineVector = readVector(lines, varNumber)

//determining baseline indexes

baselineIndexes = mat.NewVecDense(conditionsNumber, nil)

j := 0

for i := 0; i < varNumber; i++ {

if baselineVector.AtVec(i) != 0 {

baselineIndexes.SetVec(j, float64(i))

j++

}

}

return scalesVector, mat.NewDense(conditionsNumber, varNumber, conditionsMatrix), freeMembersVector, baselineVector, baselineIndexes

}

// SimplexMainPhase - solves optimization problem in canonical form

func SimplexMainPhase(scalesVector \*mat.VecDense, conditionsMatrix, inversedBaselineMatrix \*mat.Dense, baselineVector, baselineIndexes \*mat.VecDense, lowestIndex int, iteration int) \*mat.VecDense {

conditionsNumber, varNumber := conditionsMatrix.Dims()

// Building baselineMatrix from baselineIndexes of conditionsMatrix

baselineMatrix := mat.NewDense(conditionsNumber, conditionsNumber, nil)

for i := 0; i < conditionsNumber; i++ {

baselineMatrix.SetCol(i, RawVector(conditionsMatrix.ColView(int(baselineIndexes.AtVec(i)))))

}

inversedBaselineMatrixTmp := mat.NewDense(conditionsNumber, conditionsNumber, nil)

if iteration == 0 {

// First iteration. Inversing via gonums mat.Inverse()

inversedBaselineMatrix.Inverse(baselineMatrix)

inversedBaselineMatrixTmp.Copy(inversedBaselineMatrix)

} else {

// Other operations. Inversing via invOptimized() from 1.go file from 1 lab

inversedBaselineMatrixTmp.Copy(invOptimized(baselineMatrix, inversedBaselineMatrix, mat.VecDenseCopyOf(conditionsMatrix.ColView(int(baselineIndexes.AtVec(lowestIndex)))), lowestIndex))

}

// finding components of scalesVector

components := mat.NewVecDense(conditionsNumber, nil)

for i := 0; i < conditionsNumber; i++ {

components.SetVec(i, scalesVector.AtVec(int(baselineIndexes.AtVec(i))))

}

// potentials vector = components \* inversed baselineMatrix

potentials := vecMulMat(components, inversedBaselineMatrixTmp)

scoreVector := vecMulMat(potentials, conditionsMatrix)

scoreVector.AddScaledVec(scoreVector, -1, scalesVector)

// find nonBaseline indexes

nonBaseLineIndexes, j := mat.NewVecDense(varNumber-conditionsNumber, nil), 0

for i := 0; i < conditionsNumber; i++ {

if !Find(RawVector(baselineIndexes), float64(i)) {

nonBaseLineIndexes.SetVec(j, float64(i))

j++

}

}

// First exit condition, if current case is optimal

isOptimalCase := true

lowestIndex = 0

for i := 0; i < scoreVector.Len(); i++ {

if scoreVector.AtVec(i) < 0 {

isOptimalCase = false

lowestIndex = i

break

}

}

if isOptimalCase {

// THIS IS OPTIMAL CASE

fmt.Printf("every deltas element of \n")

matPrint(scoreVector)

fmt.Printf("> 0, baseline vector is optimal case \n")

return baselineVector

}

fmt.Printf("scoreVector[%v] of delta\n", lowestIndex+1)

matPrint(scoreVector)

fmt.Printf("%v < 0\n", scoreVector.AtVec(lowestIndex))

// The lowest nonBaseline index (Blends rule) for vector z

var conditionsColumn = conditionsMatrix.ColView(lowestIndex)

var zVector = mat.VecDenseCopyOf(conditionsColumn)

zVector.MulVec(inversedBaselineMatrixTmp, conditionsColumn)

// Theta

minTheta, minThetaIndex, thetaValue := math.Inf(+1), 0, 0.0

for j := 0; j < conditionsNumber; j++ {

z := zVector.AtVec(j)

if z > 0 {

thetaValue = baselineVector.AtVec(int(baselineIndexes.AtVec(j))) / z

} else {

thetaValue = math.Inf(+1)

}

if thetaValue < minTheta {

minTheta = thetaValue

minThetaIndex = j

}

}

if math.IsInf(minTheta, 1) {

panic("Problem is unsolvable")

}

// changing baseline indexes

baselineIndexes.SetVec(minThetaIndex, float64(lowestIndex))

newBaselineIndexes := mat.VecDenseCopyOf(baselineIndexes)

// creating new baselineVector (new baseline case)

newBaselineVector := mat.NewVecDense(varNumber, nil)

// minThetaIndex value equals to theta, others following the formula

newBaselineVector.SetVec(int(baselineIndexes.AtVec(minThetaIndex)), minTheta)

for i := 0; i < conditionsNumber; i++ {

newValue := 0.0

if i != minThetaIndex {

newValue = baselineVector.AtVec(int(baselineIndexes.AtVec(i))) - minTheta\*zVector.AtVec(i)

} else {

newValue = minTheta

}

newBaselineVector.SetVec(int(baselineIndexes.AtVec(i)), newValue)

}

fmt.Printf("New baseline vector on %v iteration is\n", iteration+1)

matPrint(newBaselineVector)

// Run again with new baseline vector and new baseline indexes ()

return SimplexMainPhase(scalesVector, conditionsMatrix, inversedBaselineMatrixTmp, newBaselineVector, newBaselineIndexes, minThetaIndex, iteration+1)

}

func main() {

c, r := 6, 4

scalesVector, conditionsMatrix, \_, baselineVector, baselineIndexes := readOptimizationProblem("input.txt", c, r)

fmt.Printf("scales vector is\n")

matPrint(scalesVector)

fmt.Printf("matrix of conditions is\n")

matPrint(conditionsMatrix)

fmt.Printf("first baseline vector is\n")

matPrint(baselineVector)

fmt.Printf("and it's baseline indexes\n")

matPrint(baselineIndexes)

result := SimplexMainPhase(scalesVector, conditionsMatrix, mat.NewDense(r, r, nil), baselineVector, baselineIndexes, 0, 0)

fmt.Printf("result is\n")

matPrint(result)

}

**1.go**

package main

import (

"fmt"

"io/ioutil"

"strconv"

"strings"

"gonum.org/v1/gonum/mat"

)

// Find - find element in array

func Find(array []float64, value float64) bool {

for \_, curr := range array {

if curr == value {

return true

}

}

return false

}

// Min - find min element in array

func Min(array []float64) float64 {

min := array[0]

for i := 0; i < len(array); i++ {

if min > array[i] {

min = array[i]

}

}

return min

}

func matMulVec(m \*mat.Dense, v \*mat.VecDense) \*mat.VecDense {

r, c := m.Dims()

sum := 0.0

vector := mat.NewVecDense(c, nil)

for i := 0; i < c; i++ {

sum = 0

for j := 0; j < r; j++ {

sum += v.AtVec(i) \* m.At(i, j)

}

vector.SetVec(i, sum)

}

return vector

}

func vecMulMat(v \*mat.VecDense, m \*mat.Dense) \*mat.VecDense {

r, c := m.Dims()

sum := 0.0

vector := mat.NewVecDense(c, nil)

for i := 0; i < c; i++ {

sum = 0

for j := 0; j < r; j++ {

sum += v.AtVec(j) \* m.At(j, i)

}

vector.SetVec(i, sum)

}

return vector

}

// RawVector - return mat.Vector as []float64

func RawVector(v mat.Vector) []float64 {

n := v.Len()

r := make([]float64, n)

for i := 0; i < n; i++ {

r[i] = v.AtVec(i)

}

return r

}

func eye(n int) \*mat.Dense {

d := make([]float64, n\*n)

for i := 0; i < n\*n; i += n + 1 {

d[i] = 1

}

return mat.NewDense(n, n, d)

}

func matPrint(X mat.Matrix) {

fa := mat.Formatted(X, mat.Prefix(""), mat.Squeeze())

fmt.Printf("%v\n", fa)

}

func readMatrixFromStringArray(n, shift int, lines []string) \*mat.Dense {

var matrix []float64

for i := shift \* n; i < (shift+1)\*n; i++ {

numbersRaw := strings.Split(lines[i], " ")

for j := 0; j < n; j++ {

number, \_ := strconv.ParseFloat(numbersRaw[j], 64)

matrix = append(matrix, number)

}

}

return mat.NewDense(n, n, matrix)

}

func readMatrixMatrixInvVectorIndex(input string) (\*mat.Dense, \*mat.Dense, \*mat.VecDense, int) {

str, err := ioutil.ReadFile(input)

if err != nil {

panic(err)

}

lines := strings.Split(string(str), "\n")

n := len(strings.Split(lines[0], " "))

// reading matrix

matrix := readMatrixFromStringArray(n, 0, lines)

// reading matrixInv

matrixInv := readMatrixFromStringArray(n, 1, lines)

// reading vector

vector := make([]float64, n)

for i := 2 \* n; i < 3\*n; i++ {

number, \_ := strconv.ParseFloat(lines[i], 64)

vector[i%n] = number

}

// reading vector

index, \_ := strconv.Atoi(lines[len(lines)-1])

return matrix, matrixInv, mat.NewVecDense(n, vector), index - 1

}

func mulOptimized(a, b \*mat.Dense, index int) \*mat.Dense {

n, \_ := a.Dims()

result, subSum := mat.NewDense(n, n, nil), float64(0)

for i := 0; i < n; i++ {

for j := 0; j < n; j++ {

if i != index {

subSum = a.At(i, i)\*b.At(i, j) + a.At(i, index)\*b.At(index, j)

} else {

subSum = a.At(i, index) \* b.At(index, j)

}

result.Set(i, j, subSum)

}

}

return result

}

func invOptimized(matrix, matrixInv \*mat.Dense, vector \*mat.VecDense, index int) \*mat.Dense {

n := len(vector.RawVector().Data)

// step 0

matrix.SetCol(index, vector.RawVector().Data)

// step 1

AInv := mat.NewDense(n, n, make([]float64, n\*n)) // clonning matrixInv

AInv.CloneFrom(matrixInv)

l := mat.VecDenseCopyOf(vector)

l.MulVec(AInv, vector)

if l.At(index, 0) == 0 {

panic("Matrix is uninversable!!!")

}

// step 2

storedNumber := l.At(index, 0)

l.SetVec(index, -1)

// step 3

l.ScaleVec((-1 / storedNumber), l)

// step 4

Q := eye(n)

Q.SetCol(index, l.RawVector().Data)

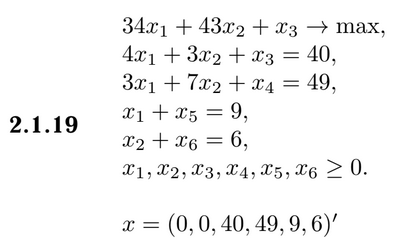
// step 5

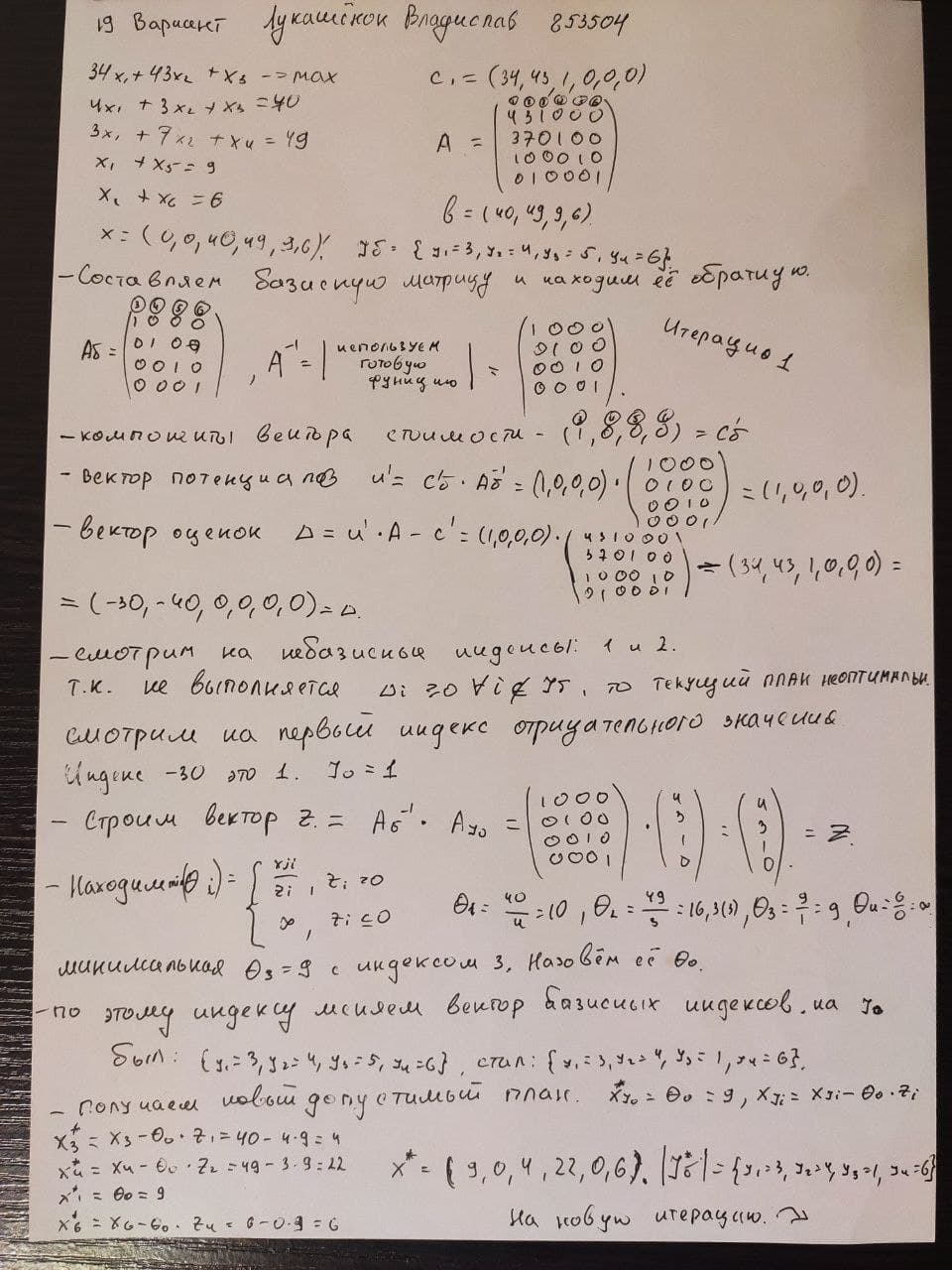
result := mulOptimized(Q, AInv, index)

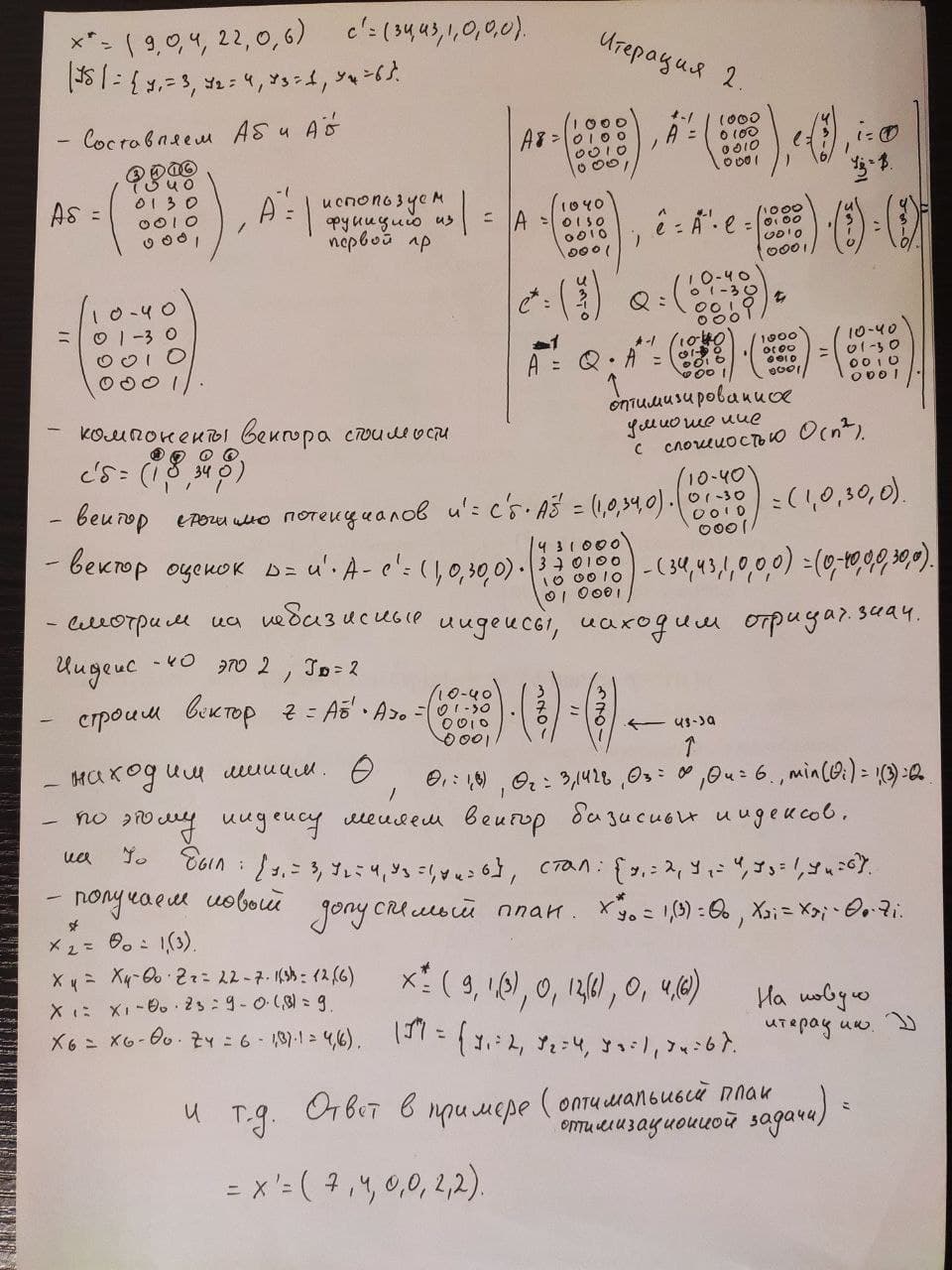
return result

}

**Первые две итерации**

****





**Демонстрация работы**

