```go
package main

import (
  "bufio"
  "fmt"
  "log"
  "os"
  "strconv"
  "strings"
)

var (
  MATRIX_FILE_PATH = "../matrix.txt"
)

func readMatrixPairToMemory(matrixFilePath string) (matrix1, matrix2
[][]float64) {
  //initialize matrices
  matrix1 = [][]float64{}
  matrix2 = [][]float64{}
  file, err := os.Open(matrixFilePath)
  if err != nil {
    log.Fatal(err)
  }
  defer file.Close()

  scanner := bufio.NewScanner(file)

  lineNum := 0
  matrixNum := 0
  for scanner.Scan() {
    //Grab first line
    lineText := scanner.Text()
    //Check for any error during line scan
    if err := scanner.Err(); err != nil {
      log.Fatal(err)
    }
    //If we are reading the first line, continue
    if lineText == "--A" {
      continue
    }
    //If we are reading matrix B line separator, update matrixNum and set lineNum to zero.
    if lineText == "--B" {
      lineNum = 0
      matrixNum = 1
      continue
    }
    //Split the line we read so we can work on each number.
    stringArray := strings.Split(lineText, ";")
    var numArray []float64
    for i := range stringArray {
      num, err := strconv.ParseFloat(stringArray[i], 64)
      if err != nil {
        panic(err.Error())
      }
      numArray = append(numArray, float64(num))
    }
    //fmt.Printf("Line %v numbers array: %v\n", lineNum, numArray)
    //Depending on the matrixNum, choose which matrix will receive the numbers.
    if matrixNum == 0 {
      matrix1 = append(matrix1, numArray)
      lineNum++
    } else {
      matrix2 = append(matrix2, numArray)
      lineNum++
    }

  }
  return matrix1, matrix2
}

func CheckIfMatricesCanMultiply(matrix1, matrix2 [][]float64) bool {
  //Numbers of columns of matrix1 == num of lines of matrix 2 ?
  return (len(matrix1[0]) == len(matrix2))
}

func InitializeMatrixWithZeros(numOfRows, numOfColumns int) [][]float64 {
  //fmt.Printf("Initializing %vX%v matrix\n", numOfRows, numOfColumns)
  //Innitialize an empty matrix
  var initializedMatrix [][]float64 = [][]float64{}

  //Append the right number of rows to the initializedMatrix
  for i := 0; i < numOfRows; i++ {
    //Create a row, with the right size filled with zeros.
    zeroFilledRow := []float64{}
    for j := 0; j < numOfColumns; j++ {
      zeroFilledRow = append(zeroFilledRow, float64(0))
    }
    initializedMatrix = append(initializedMatrix, zeroFilledRow)
  }
  //fmt.Printf("Initialized matrix: %v\n", initializedMatrix)
  return initializedMatrix
}

func MultiplyMatrices(matrix1, matrix2 [][]float64) (matrixResult [][]float64, canMultiply bool) {
  //fmt.Println(strings.Repeat("#", 15))
  //fmt.Println("Started matrix multiplication.")
  //start := time.Now()

  //Check if we can multiply the input matrices:
  canMultiply = CheckIfMatricesCanMultiply(matrix1, matrix2)
  if !canMultiply {
    fmt.Println("The given matrices cannot be multiplied.\nCheck if your input was correct on file matrix.txt")
    return [][]float64{}, false
  }

  //Define size of the resulting matrix:
```

```go
  resultMatrixNumOfColumns := len(matrix2[0])
  resultMatrixNumOfRows := len(matrix1)
  //initializeSaidMatrixWithZeros
  matrixResult = InitializeMatrixWithZeros(resultMatrixNumOfRows, resultMatrixNumOfColumns)

  for i := 0; i < len(matrixResult); i++ {
    for j := 0; j < len(matrixResult[0]); j++ {
      //fmt.Printf("Finding total for a%v%v\n", i, j)
      var total float64 = 0
      for k := 0; k < len(matrixResult); k++ {
        total = total + matrix1[i][k]*matrix2[k][j]
        //fmt.Printf("total is: %v\n", total)
      }
      matrixResult[i][j] = total
      //fmt.Printf("Matrix: %v\n", matrixResult)
    }
  }

  //fmt.Println("Finished multiplying matrices.")
  //fmt.Printf("Entry matrices %v X %v \n", matrix1, matrix2)
  //fmt.Printf("Resulting matrix: %v\n", matrixResult)
  //timeElapsed := time.Since(start)
  //fmt.Printf("This operation took %v.\n", timeElapsed)
  //fmt.Println(strings.Repeat("#", 15))
  return matrixResult, canMultiply
}

//Create the Mi matrix needed to zero out the element under the pivot specified. Pay attention that column 1 should be specified as 0 on this function.
func createPivotMatrixM(matrix1 [][]float64, pivotColumn int) (pMatrixM [][]float64) {
  //Get the size of the M matrix to produce.
  miSize := len(matrix1)
  //Initialize a zero matrix with this size.
  pMatrixM = InitializeMatrixWithZeros(miSize, miSize)

  for i := 0; i < miSize; i++ {
    for j := 0; j < miSize; j++ {
      //Fill in the matrix we just built, with 1s in the main Diagonal.
      if i == j {
        pMatrixM[i][j] = 1
      }
      //Fill the calculated numbers to zero out the values below the pivot specified.
      if i > j && j == pivotColumn {
        pMatrixM[i][j] = -1 * matrix1[i][j] / matrix1[pivotColumn][pivotColumn]
      }
    }
  }
  return pMatrixM
}

func generateLiMatrixFromUiMatrix(Ui [][]float64) (Li [][]float64) {
  matrixSize := len(Ui)
  Li = InitializeMatrixWithZeros(matrixSize, matrixSize)
  for i := 0; i < matrixSize; i++ {
    for j := 0; j < matrixSize; j++ {
      if i == j {
        Li[i][j] = 1
        continue
      }
      if Ui[i][j] == 0 {
        continue
      }
      Li[i][j] = -1 * Ui[i][j]
    }
  }
  return Li
}

func calculateDeterminantForUMatrix(matrix1 [][]float64) (det float64) {
  matrixSize := len(matrix1)
  det = matrix1[0][0]
  for i := 1; i < matrixSize; i++ {
    det = det * matrix1[i][i]
  }
  return det
}

func checkIfMatrixIsSquare(matrix1 [][]float64) (isSquare bool) {
  return len(matrix1) == len(matrix1[0])
}

func forwardSubstitution(matrixA, vectorB [][]float64) (res [][]float64) {
  res = append(res, []float64{vectorB[0][0] / matrixA[0][0]})
  for i := 1; i < len(matrixA); i++ {
    var sum float64
    for j := 0; j < i; j++ {
      //fmt.Printf("matrix%v%v:%v\tvecB%v%v:%v\n", i, j, matrixA[i][j], j, 0, vectorB[j][0])
      sum = sum + matrixA[i][j]*res[j][0]
    }
    //fmt.Printf("Sum is:%v\n", sum)
    //fmt.Printf("vectorB%v%v:%v\n", i, 0, vectorB[i][0])
    yi := (vectorB[i][0] - sum) / matrixA[i][i]
    res = append(res, []float64{yi})
  }
  return res
}

func backwardsSubstitution(matrixA, vectorB [][]float64) (res [][]float64) {
  vectorBNumOfRows := len(vectorB)
  res = InitializeMatrixWithZeros(vectorBNumOfRows, 1)
  res[vectorBNumOfRows-1][0] = vectorB[vectorBNumOfRows-1][0] / matrixA[vectorBNumOfRows-1][vectorBNumOfRows-1]
  for i := vectorBNumOfRows - 2; i >= 0; i-- {
    var sum float64
    for j := vectorBNumOfRows - 1; j > i; j-- {
      //fmt.Printf("matrix%v%v:%v\tvecB%v%v:%v\n", i, j, matrixA[i][j], j, 0, vectorB[j][0])
      sum = sum + matrixA[i][j]*res[j][0]
      //fmt.Printf("%v %v\t%v %v\n", matrixA[i][j], res[j][0], i, j)
    }
    yi := (vectorB[i][0] - sum) / matrixA[i][i]
    res[i][0] = yi
  }
  return res
}

func checaSeMatrizAEPositivaDefinida(c configuration) (ePositivaDefinida bool) {
  L, _ := LUViaCholeskyDecomposition(c)
  for i := 0; i < c.systemOrder; i++ {
    if L[i][i] == 0 {
      return false
    }
  }
  return true
}

func checaSeMatrizESimetrica(c configuration) (eSimetrica bool) {
  for i := 0; i < c.systemOrder; i++ {
    for j := 0; j < c.systemOrder; j++ {
      if c.matrixA[i][j] != c.matrixA[j][i] {
        return false
      }
    }
  }
  return true
}
```