JSS Mahavidyapeetha
Sri Jayachamarajendra College of Engineering (SJCE),
Mysore – 570 006
An Autonomous Institute Affiliated to
Visvesvaraya Technological University, Belgaum

"Determine whether the system of linear equations are singular or not"

Report submitted in partial fulfillment of
curriculum prescribed for the Linear Algebra
for the award of the degree of
BACHELOR OF ENGINEERING IN
COMPUTER SCIENCE AND ENGINEERING
by

Nithin Prabhu G - 4JC15CS068
Mrudula N M - 4JC15CS137

Submitted to,
Dr. Roopamala T D
Associate Professor,
Department of Computer Science & Engineering,
SJCE, Mysore

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Contents

Chapter 1

INTRODUCTION

1.1 Matrix

A matrix (plural: matrices) is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. The individual items in an $m \times n$ matrix A, often denoted by $a_{i,j}$, where max i = m and max j = n, are called its elements or entries.

1.1.1 Linear equations

Matrices can be used to compactly write and work with multiple linear equations, that is, systems of linear equations. For example, if A is an m-by-n matrix, x designates a column vector (that is, $n \times 1 - matrix$) of n variables x_1, x_2, ..., x_n, and b is an $m \times 1 - column$ vector, then the matrix equation

$Ax = b$
is equivalent to the system of linear equations

$A_{1,1}x_1 + A_{1,2}x_2 + ... + A_{1,n}x_n = b_1$
$A_{2,1}x_1 + A_{2,2}x_2 + ... + A_{2,n}x_n = b_2$
.
.
.
$A_{m,1}x_1 + A_{m,2}x_2 + ... + A_{m,n}x_n = b_m$

Using matrices, this can be solved more compactly than would be possible by writing out all the equations separately. If n = m and the equations are independent, this can be done by writing
$x = A^{-1}b$
where A^{-1} is the inverse matrix of A. If A has no inverse, solutions if any can be found using its generalized inverse.

1.1.2 Invertible Matrices

In linear algebra, an n-by-n square matrix A is called invertible (also nonsingular or nondegenerate) if there exists an n-by-n square matrix B such that

$AB = BA = I_n$

where I_n denotes the n-by-n identity matrix and the multiplication used is ordinary matrix multiplication. If this is the case, then the matrix B is uniquely determined by A and is called the inverse of A, denoted by $A^{-1}1$.

A square matrix that is not invertible is called singular or degenerate. A square matrix is singular if and only if its determinant is 0. Singular matrices are rare in the sense that a square matrix randomly selected from a continuous uniform distribution on its entries will almost never be singular.

1.2 Determinant

The determinant $det(A)$ or $|A|$ of a square matrix A is a number encoding certain properties of the matrix. A matrix is invertible if and only if its determinant is nonzero. Its absolute value equals the area (in R_2) or volume (in R_3) of the image of the unit square (or cube), while its sign corresponds to the orientation of the corresponding linear map: the determinant is positive if and only if the orientation is preserved.

The determinant of 2-by-2 matrices is given by

$$det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - cb$$

The determinant of 3-by-3 matrices involves 6 terms (rule of Sarrus). The more lengthy Leibniz formula generalises these two formulae to all dimensions.

Adding a multiple of any row to another row, or a multiple of any column to another column, does not change the determinant. Interchanging two rows or two columns affects the determinant by multiplying it by -1. Using these operations, any matrix can be transformed to a lower (or upper) triangular matrix, and for such matrices the determinant equals the product of the entries on the main diagonal; this provides a method to calculate the determinant of any matrix. Finally, the Laplace expansion expresses the determinant in terms of minors, that is, determinants of smaller matrices. This expansion can be used for a recursive definition of determinants (taking as starting case the determinant of a 1-by-1 matrix, which is its unique entry, or even the determinant of a 0-by-0 matrix, which is 1), that can be seen to be equivalent to the Leibniz formula. Determinants can be used to solve linear systems using Cramer's rule, where the division of the determinants of two related square matrices equates to the value of each of the system's variables.

1.3 LU decomposition

In numerical analysis and linear algebra, **LU decomposition** (where 'LU' stands for 'lower upper', and also called **LU factorization**) factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. The LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using the LU decomposition, and

it is also a key step when inverting a matrix, or computing the determinant of a matrix. The LU decomposition was introduced by mathematician Tadeusz Banachiewicz in 1938.

1.3.1 Definition

Let A be a square matrix. An **LU factorization** refers to the factorization of A, with proper row and/or column orderings or permutations, into two factors, a **unit** lower triangular matrix L and an upper triangular matrix U,

$A = LU$,

In the lower triangular matrix all elements above the diagonal are zero, in the upper triangular matrix, all the elements below the diagonal are zero. For example, for a 3-by-3 matrix A, its LU decomposition looks like this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Chapter 2

PROBLEM STATEMENT

Design a program to check whether the system of linear equations are singular or non singular.

Chapter 3

ALGORITHM

Given an 3×3 matrix, $A =$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Step 1 : Start.

Step 2 : Check if A is a square matrix.
If yes, goto step 3.
Else,gote step 10.

Step 3 : Consider $A = LU$.
Where,

$$\text{(Lower Triangular Matrix) L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$$

$$\text{(Upper Triangular Matrix) U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Step 4 : Matrix A is equal to product of matrices L and U. Finding the product of matrices L and U.

$$i.e., \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{13} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix}$$

Step 5 : Equating substituting product of LU in equation $A = LU$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{13} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix}$$

Step 5a : $a_{11} = u_{11}$

$a_{12} = u_{12}$

$a_{13} = u_{13}$

Step 5b : $l_{21} = \frac{a_{21}}{a_{11}}$

$l_{21} = \frac{a_{31}}{a_{11}}$

Step 5c : $u_{22} = a_{22} - \frac{a_{21}a_{12}}{a_{11}}$

$u_{22} = a_{22} - \frac{a_{21}a_{12}}{a_{11}}$

Step 5d : $l_{23} = \frac{a_{32}a_{11} - a_{31}a_{12}}{a_{22}a_{11} - a_{21}a_{12}}$

Step 5e : $u_{33} = a_{33} - \frac{a_{31}a_{13}}{a_{a}11} - (\frac{a_{32}a_{11} - a_{31}a_{12}}{a_{22}a_{11} - a_{21}a_{12}})(a_{23} - \frac{a_{21}a_{31}}{a_{11}})$

Step 6 : Consturucting matrices L and U using the values from step 5.

Step 7 : Finding determinant of L and U. The determinant of a triangual matrix is equal to product of principal diagonal elements.
det(L) = 1 (i.e., principal dialgonal elements are equal to 1)
det(U) = $u_{11}.u_{22}.u_{33}$

Step 8 : Determinant of A is given by,
det(A) = det(LU) = det(L).det(U)
Since det(L) = 1, implies det(A) = det(U)

Step 9 : If det(A) = 0, print matrix A is singular.
Else, print matrix A is non-singular.

Step 10 : Stop.

Chapter 4

PYTHON CODE

my_algorithms.py

```
import sys

    Comment : Ensure Python 3 compatibility.
if sys.version_info[0] == 3:
Comment : unicode is not defined in Python 3. However, unicode is
needed when
Comment : dealing with non-ascii characters in Python 2.
unicode = str

    def __is_number(n):
"""Find out if a given argument is a number or not."""
return isinstance(n, float) or isinstance(n, int)

    def __is_string(s):
"""Find out if a given argument is a string or not."""
return isinstance(s, str) or isinstance(s, unicode)

    def my_abs(n):
"""Calculate the absolute value of a given number."""
if n < 0:
return -n
return n

    def my_gcd(n, m):
"""Calculate the gcd of n and m using Stein's algorithm.

    For a description of Stein's algorithm, see for example
https://en.wikipedia.org/wiki/Binary_GCD_algorithm"""

    Comment : Unit is needed to mimic the default Python implementation of gcd. Python
Comment : thinks that if m < 0, then the gcd shall be negative.
unit = 1
if m < 0:
```

```
unit = -1
n = my_abs(n)
m = my_abs(m)

    Comment : Base cases.
if n == m == 0:
return 0
if n == 0:
return unit * m
if m == 0:
return n
if n == m:
return unit * n

    Comment : For convenience, make sure n is the bigger one of the two.
if n < m:
n, m = m, n

    Comment : Exp is the largest power of two that divides both n and m.
exp = 0
Comment : While n and m are both even, divide both by 2, and increment exp.
while (n | m) & 1 == 0:
n >>= 1
m >>= 1
exp += 1

    Comment : Remove possible leftover factors of 2 from n. Hence n will be odd.
while n & 1 == 0:
n >>= 1

    while n != 0:
Comment : Here we ensure that n is always the bigger of the two. Thus n will be
Comment : positive until n == m.
if n < m:
n, m = m, n

    n -= m
Comment : N is even if m was odd, so remove any leftover factors of 2 from n.
while n != 0 and n & 1 == 0: n >>= 1

    Comment : At this point, n == 0 and hence gcd(n, m) == m. Multiply the resulting m
Comment : by 2êxp, which is also a common factor of the original n and m.
return unit * (m << exp)

    def my_max(x, *args):
"""Return the maximum of the given numbers or a given list."""

    Comment : If x is a non-empty list, find the maximum value in it.
```

```
if isinstance(x, list) and len(x) > 0:
greatest = x[0]
i = 0
while i < len(x):
Comment : The list x must contain only numbers (floats and/or integeres.)
if not (__is_number(x[i])):
raise TypeError('Expected an integer or a float.')
if x[i] > x[0]:
greatest = x[i]
i += 1
return greatest

    Comment : If x is a number, find the maximum of the given n-arguments.
if __is_number(x):
greatest = x
i = 0
Comment : Iterate through the rest of the arguments given by the user.
while i < len(args):
Comment : All arguments must be numbers.
if not __is_number(args[i]):
raise TypeError('Expected an integer or a float.')
if args[i] > x:
greatest = args[i]
i += 1
return greatest

    raise TypeError( 'Expected a list of ints/floats, or a sequence of ints/floats')

    def my_range(x, *args):
"""Return a list of integers, mimicing the default my_range() call."""
Comment : By default, the lower bound is zero.
lower = 0
Comment : By default, the upper bound is x.
upper = x

    Comment : User gave two arguments i.e. user gave also a lower bound. Then x is the
lower-bound and the second argument (args[0]) is the upper bound.
if len(args) == 1:
lower = x
upper = args[0]

    Comment : Here we generate the my_range.
ret = [ ]
i = lower
while i < upper:
ret.append(i)
i += 1
```

```
    return ret

    def my_reversed(x):
"""Reverse a list."""

    Comment : Copy the elements of x to ret, starting from the end of x.
ret = [ ]
i = len(x) - 1
while i >= 0:
ret.append(x[i])
i -= 1
return ret

    def my_split(s, char):
"""Split string into a list by treating char as a separator."""
if not __is_string(s):
raise TypeError('Expected a string.')

    result = [ ]
current_string = ""
for i in my_range(len(s)):
Comment : i:s character of s
c = s[i]
Comment : If c is the separator character char, append
current_string to the
Comment : result list.
if c == char:
result.append(current_string)
current_string = ""
continue
Comment : c is not the separator char, so we append c to
current_string
current_string += c
Comment : Done iterating through s. Add the last string to result.
result.append(current_string)
return result

    def my_strip(s):
"""Remove spaces from the beginning and ending of a string."""
if not __is_string(s):
raise TypeError('Expected a string.')

    Comment : Find the first non-space character and let i be its index.
i = 0
while i < len(s) and s[i] == ' ':
i += 1

    Comment : Find the last non-space character and let j be its index.
```

```
j = len(s) - 1
while j >= 0 and s[j] == ' ':
j -= 1

    Comment : The result will be the substring s[i] + ... + s[j].
result = ""
for char in my_range(i, j+1):
result += s[char]

    return result

    def my_lower(s):
"""Convert uppercase letters of the Finnish alphabet into lowercase."""
if not __is_string(s):
raise TypeError('Expected a string.')

    uppers = u"ABCDEFGHIJKLMNOPQRSTUVWXYZÅÄÖ"
lowers = u"abcdefghijklmnopqrstuvwxyzåäö"

    result = ""
Comment : Iterate through the characters of s.
for i in my_range(len(s)):
char = s[i]
Comment : Compare char to uppercase letters.
for j in my_range(len(uppers)):
Comment : Char is an uppercase letter, convert it to a lowercase one.
if char == uppers[j]:
char = lowers[j]
result += char

    return result
```

Matrix.py

```
from .my_algorithms import my_range

    class Matrix:
"""Matrix object"""

    def __init__(self, rows, n, m):
"""Construct a matrix.

    Keyword arguments:
rows – the list of rows in the matrix
n – the number of rows in the matrix
m – the number of column in the matrix
"""
self.rowAmount = n
```

```
self.colAmount = m
self.rowArray = rows
self.colArray = [[ ] for i in my_range(n)]
self.scalar = 1

    def __str__(self):
"""Print the matrix in a readable format"""
output = ""

    for row in self.rowArray:
output += "["

    for i in my_range(len(row)):
cell = row[i]
elem = self.scalar * 1.0 * cell[0] / cell[1]

    if elem elem = int(elem)

    if i == len(row) - 1:
output += str(elem)
else:
output += str(elem) + " "

    output += "] "
return output[:-1]

    def multiplyScalar(self, n):
"""Multiply the current scalar value.

    self.scalar handles the scalar multiplication of the matrix. Each time
a value(s) is returned via a getter, the returned values have to be
multiplied by the scalar.
"""
self.scalar *= n

    def getRowAmount(self):
"""Return the amount of rows.

    This is needed mainly for testing purposes."""
return self.rowAmount

    def getRowArray(self):
"""Return the row array.

    This is needed mainly for testing purposes."""
if self.scalar == 1:
return self.rowArray
returnArray = [[self.scalar * elem for elem in row]
```

```
for row in self.rowArray]
return returnArray

    def getColAmount(self):
"""Return the amount of columns.

    This is needed mainly for testing purposes."""
return self.colAmount

    def getColArray(self):
"""Return the current column array.

    This is needed mainly for testing purposes"""
return self.colArray

    def getScalar(self):
"""Return the current scalar value.

    This is needed mainly for testing purposes"""
return self.scalar

    def getCell(self, row, col):
"""Return the content of the requested cell"""
return (self.scalar * self.rowArray[row][col][0],
self.rowArray[row][col][1])

    def getRow(self, row):
"""Return the requested row."""
if self.scalar == 1:
return self.rowArray[row]

    returnRow = [(self.scalar * elem[0], elem[1])
for elem in self.rowArray[row]]
return returnRow

    def genColArray(self, col):
"""Generate only the requested column and store it in self.colArray.

    Generating only the needed column at a time keeps the worst case
scenario at a reasonable O(m). As the same column vector is requested
again, then getCol becomes O(1).
"""
for i in my_range(self.rowAmount):
self.colArray[col].append(self.rowArray[i][col])
return self.colArray[col]

    def getCol(self, col):
"""Return the requested column of the matrix.
```

```
    If this is the first time requesting a column, the column must be first
generated. Otherwise, we may just look the column up from self.colArray.
"""
if len(self.colArray[col]) == 0:
self.genColArray(col)

    if self.scalar == 1:
return self.colArray[col]

    returnCol = [(self.scalar * elem[0], elem[1])
for elem in self.colArray[col]]
return returnCol
```

calculator.py

```
from .Matrix import Matrix
from .my_algorithms import my_gcd, my_abs, my_range, my_reversed

    def frac_add(frac_a, frac_b):
"""Return the sum of fractions frac_a and frac_b."""
if frac_a[1] == frac_b[1]:
return (frac_a[0] + frac_b[0], frac_a[1])

    numerator = frac_a[0] * frac_b[1] + frac_b[0] * frac_a[1]
denominator = frac_a[1] * frac_b[1]
return (numerator, denominator)

    def frac_sub(frac_a, frac_b):
"""Return the difference of fractions frac_a and frac_b."""
if frac_a[1] == frac_b[1]:
return (frac_a[0] - frac_b[0], frac_a[1])

    numerator = frac_a[0] * frac_b[1] - frac_b[0] * frac_a[1]
denominator = frac_a[1] * frac_b[1]
return (numerator, denominator)

    def frac_mult(frac_a, frac_b): """Return the product of fractions frac_a and frac_b."""
numerator = frac_a[0] * frac_b[0]
denominator = frac_a[1] * frac_b[1]
return (numerator, denominator)

    def frac_div(frac_a, frac_b):
"""Return frac_a divided by frac_b."""
numerator = frac_a[0] * frac_b[1]
denominator = frac_a[1] * frac_b[0]
return (numerator, denominator)

    def frac_abs(frac):
```

```
"""Return the absolute value of fraction frac."""
return (my_abs(frac[0]), my_abs(frac[1]))

    def frac_ge(frac_a, frac_b):
"""Return true, if frac_a is greater than frac_b"""
return frac_a[0] * frac_b[1] > frac_b[0] * frac_a[1]

    def frac_reduc(frac):
"""Reduce the given fraction."""
syt = my_gcd(frac[0], frac[1])

    Comment : This condition here ensures we don't end up dividing by 0. Namely, in
Comment : Python gcd(0, 0) == 0.
if syt == 0:
syt = 1

    return (int(frac[0] / syt), int(frac[1] // syt))

    def matrixAddition(A, B):
"""Add matrix B to matrix A if the sum matrix is defined."""

    if not A or not B:
return None
Comment : Addition undefined.
if A.getColAmount() != B.getColAmount():
return None
Comment : Addition undefined.
if A.getRowAmount() != B.getRowAmount():
return None

    Comment : Resulting matrix.
C = [ ]
for rowIndex in my_range(A.getRowAmount()):
resultRow = [ ]
for colIndex in my_range(A.getColAmount()):
cellOfA = A.getCell(rowIndex, colIndex)
cellOfB = B.getCell(rowIndex, colIndex)
result = frac_add(cellOfA, cellOfB)
resultRow.append(frac_reduc(result))
C.append(resultRow)

    return Matrix(C, A.getRowAmount(), A.getColAmount())

    def matrixSubstraction(A, B):
"""Substract matrix B from A if the difference is defined."""

    if not A or not B:
return None
```

```
    Comment : A very special case that should never happen (except in my tests).
if A == B:
Comment : Return a zero matrix of the same size as A (and B).
return Matrix([[(0, 1) for i in my_range(A.getColAmount())][for j in my_range(A.getRowAmount())],
A.getRowAmount(),A.getColAmount())

    Comment : Multiply B by a scalar of -1. This is because A-B == A+(-1*B).
B.multiplyScalar(-1)
resultMatrix = matrixAddition(A, B)

    Comment : Set B's scalar back to original.
B.multiplyScalar(-1)

    return resultMatrix

    def matrixScalarMultiplication(A, scalar):
"""Multiply a matrix by a scalar."""
A.multiplyScalar(scalar)
return A

    def matrixMultiplication(A, B):
"""Multiply two matrices if the product is defined."""

    if not A or not B:
return None
Comment : Multiplication is undefined if this condition holds.
if A.getColAmount() != B.getRowAmount():
return None

    n = A.getRowAmount()
m = A.getColAmount()
p = B.getColAmount()

    Comment : This will be the result matrix.
C = [[(0, 1) for i in my_range(p)] for j in my_range(n)]

    for i in my_range(n):
for j in my_range(p):
cellValue = (0, 1)

    for k in my_range(m):
toAdd = frac_mult(A.getCell(i, k), B.getCell(k, j))
cellValue = frac_add(cellValue, toAdd)

    cellValue = frac_reduc(cellValue)
C[i][j] = cellValue
```

```
    return Matrix(C, n, p)

    def matrixTranspose(A):
"""Calculate the transpose of matrix A."""
n = A.getRowAmount()
m = A.getColAmount()
result = [[A.getCell(j, i) for j in my_range(n)] for i in my_range(m)]
return Matrix(result, m, n)

    def __pivot(A):
"""Pivot A so that largest element of each column is on the diagonal.

    More specifically, A is modified so that every diagonal element has a value
that is has at least as large absolute value as every cell below it."""
n = A.getRowAmount()

    Comment : At first, P is the identity matrix.
P = [[(int(i == j), 1) for i in my_range(n)] for j in my_range(n)]

    Comment : Needed for calculating the determinant of P.
totalPivots = 0

    Comment : Iterate through columns.
for j in my_range(n):
greatest = (0, 1)
swapWith = j

    Comment : We don't have to care about column values above the diagonal.
for row in my_range(j+1, n):
if frac_ge(frac_abs(A.getCell(row, j)), greatest):
greatest = frac_abs(A.getCell(row, j))
swapWith = row

    Comment : True if we need to swap rows.
if swapWith != j:
Comment : Swap rows so that the greatest element is now on the diagonal.
P[j], P[swapWith] = P[swapWith], P[j]
totalPivots += 1

    return (Matrix(P, n, n), totalPivots)

    def __LUP_decomposition(A):
"""Calculate the LUP decomposition of A."""
n = A.getRowAmount()

    Comment : Format L and U as the zero matrix. Note that we are dealing with fractions
Comment : here for 100L = [[(0, 1) for i in my_range(n)] for j in my_range(n)]
U = [[(0, 1) for i in my_range(n)] for j in my_range(n)]
```

```
    pivot_info = __pivot(A)
P = pivot_info[0]

    Comment : Set mult equal to the determinant of P.
mult = (-1) ** pivot_info[1]

    Prod = matrixMultiplication(P, A)

    Comment : We use the general algorithm described here:
Comment : https://rosettacode.org/wiki/LU_decomposition to calculate cell values for
Comment : U and L. There is a summation formula given for U_ij and L_ij, which is
Comment : implemented here.
for j in my_range(n):
Comment : Calculate U[i][j].
for i in my_range(j+1):
the_sum = (0, 1)

    for k in my_range(i):
toAdd = frac_mult(U[k][j], L[i][k])
the_sum = frac_add(the_sum, toAdd)

    the_sum = frac_reduc(the_sum)
Comment : U_ij == Prod_ij - the_sum
U[i][j] = frac_sub(Prod.getCell(i, j), the_sum)

    Comment : Calculate L[i][j].
for i in my_range(j, n):
the_sum = (0, 1)

    for k in my_range(j):
toAdd = frac_mult(L[i][k], U[k][j])
the_sum = frac_add(the_sum, toAdd)

    the_sum = frac_reduc(the_sum)
L[i][j] = frac_sub(Prod.getCell(i, j), the_sum)
L[i][j] = frac_div(L[i][j], U[j][j])

    L = Matrix(L, n, n)
U = Matrix(U, n, n)

    return (L, U, P, mult)

    def matrixDeterminant(A):
"""Calculate the determinant of matrix A."""

    Comment : Determinant is undefined for non-square matrices.
if A.getRowAmount() != A.getColAmount():
```

```
return None

    Comment : Decompose the matrix. Return value is (L, U, P, mult).
decomposition = __LUP_decomposition(A)

    U = decomposition[1]

    Comment : The determinant is the product of U's diagonal values.
ans = (1, 1)
for i in my_range(U.getRowAmount()):
ans = frac_mult(ans, U.getCell(i, i))
ans = frac_reduc(ans)

    Comment : If the determinant is zero, ans[1] might also be zero so we treat this
Comment : case separately.
if ans[1] == 0:
return 0

    det_of_P = decomposition[3]
det = ans[0] * det_of_P * 1.0 / ans[1]
if det // 1 == det:
return int(det)
return det

    def __forward_substitution(L):
"""Invert L using forward substitution.

    For more information, see for example
http://en.wikipedia.org/wiki/Triangular_matrixComment : Forward_and_back_substitution
"""
m = L.getRowAmount()
inverse = [[(0, 1) for i in my_range(m)] for j in my_range(m)]

    Comment : bVector is the a:th column of the identity matrix. Solve the a:th column
Comment : of the inverse of L.
for a in my_range(m):
bVector = [(0, 1) for i in my_range(m)]
bVector[a] = (1, 1)

    Comment : This will be the a:th column of the inverse matrix of L.
xVector = [ ]
for x in my_range(m):
Comment : Calculate the summation of x_index using the formula given for x_m
Comment : in the article above.
the_sum = (0, 1)

    for i in my_range(x):
toAdd = frac_mult(L.getCell(x, i), xVector[i])
```

```
the_sum = frac_add(the_sum, toAdd)

    the_sum = frac_reduc(the_sum)
value = frac_sub(bVector[x], the_sum)
value = frac_mult(value, L.getCell(x, x))
value = frac_reduc(value)
xVector.append(value)

    Comment : xVector is now the a:th column of L's inverse.
for i in my_range(m):
inverse[i][a] = xVector[i]

    return Matrix(inverse, m, m)

    def __backward_substitution(U):
"""Invert U using backward substitution.

    This is basically the same as forward substitution, but done working
backwards."""
m = U.getRowAmount()
inverse = [[(0, 1) for i in my_range(m)]
for j in my_range(m)]

    for a in my_range(m):
bVector = [(0, 1) for i in my_range(m)]
bVector[a] = (1, 1)

    xVector = [ ]
for x in my_reversed(my_range(m)):
the_sum = (0, 1)

    for i in my_reversed(my_range(x+1, m)):
toAdd = frac_mult(U.getCell(x, i), xVector[m-1 - i])
the_sum = frac_add(the_sum, toAdd)

    the_sum = frac_reduc(the_sum)
value = frac_sub(bVector[x], the_sum)
value = frac_div(value, U.getCell(x, x))
value = frac_reduc(value)
xVector.append(value)

    for i in my_range(m):
inverse[m-1-i][a] = xVector[i]

    return Matrix(inverse, m, m)

    def matrixInverse(A):
"""Invert matrix A."""
```

```
Comment : Inverse not defined iff the determinant is zero.
if matrixDeterminant(A) == 0:
return None

    Comment : Calculate the LUP decomposition of A. PA = LU
decomposition = __LUP_decomposition(A)

    Comment : Invert L using forward substitution.
L_inv = __forward_substitution(decomposition[0])
Comment : Invert U using forward substitution.
U_inv = __backward_substitution(decomposition[1])

    P = decomposition[2]

    Comment : PA = LU
Comment : -> (PA)^-1 = (LU)^-1
Comment : -> A^-1 * P^-1 = U^-1 * L^-1
Comment : -> A^-1 = U^-1 * L^-1 * P
C = matrixMultiplication(U_inv, L_inv)
return matrixMultiplication(C, P)
```

parser.py

```
from .my_algorithms import my_split, my_strip, my_lower
from .calculator import frac_reduc
from .Matrix import Matrix
import sys

    Comment : Make everything work with python2.
if sys.version_info[0] == 2:
input = raw_input

    def __ask_input(string):
"""Ask input from user with 'string'."""
try:
input_data = input(string)
except (KeyboardInterrupt, EOFError):
print("Bye!")
sys.exit(0)
return input_data

    def __is_number(n):
"""Find out if a given argument is a number or not."""
try:
Comment : Strings for example, can be converted to floats if the string is of the correct form.
float(n)
return True
except:
return False
```

```
    def __parse_float(string): """Try parsing a float from string."""
elem = my_split(string, ".")

    Comment : String is not of the form of 4389.109903.
if len(elem) == 1:
elem = my_split(string, ",")

    Comment : String is of the form 1234.5678 or 1234,5678.
if len(elem) == 2:
Comment : The whole part or the decimal part is not a number.
if not (__is_number(elem[0])) or not (__is_number(elem[1])):
return None

    numerator = int(elem[0]) * 10 ** len(elem[1])
if numerator >= 0:
numerator += int(elem[1])
else:
numerator -= int(elem[1])

    denominator = 10 ** len(elem[1])
frac = (numerator, denominator)

    return frac_reduc(frac)

    return None

    def __parse_fraction(string):
"""Parse a fraction from a string. Return none if input's not a fraction."""
elem = my_split(string, "/")

    if len(elem) == 2:
Comment : Do things this weirdly just to handle the case where user gives a value in the form
1.3/3,7.
frac_1 = __parse_float(elem[0])
frac_2 = __parse_float(elem[1])

    if not frac_1:
if not __is_number(elem[0]):
return None
frac_1 = (int(elem[0]), 1)

    if not frac_2:
if not __is_number(elem[1]):
return None
frac_2 = (int(elem[1]), 1)

    numerator = frac_1[0] * frac_2[1]
```

```
denominator = frac_1[1] * frac_2[0]
frac = (numerator, denominator)
return frac_reduc(frac)

    return None

    def __parse_values(row):
"""Parse numbers from a string into a list."""
values = [ ]
args = my_split(row, " ")

    for i in range(len(args)):
elem = __parse_fraction(args[i])

    if not elem: Comment : elem is not a fraction
elem = __parse_float(args[i])

    if not elem: Comment : elem is not a float nor a fraction
if not __is_number(args[i]): Comment : elem is not even a whole number
return None
elem = (int(args[i]), 1)

    values.append(elem)

    return values

    def parseMatrix():
"""Ask user to input a matrix and return a new Matrix object."""

    rows = [ ]

    print("")
print("Input a matrix row by row. Plain enter stops.")

    row = my_strip(__ask_input("row: "))

    Comment : Loop until an empty row is encountered.
while row:
newRow = __parse_values(row)

    Comment : User inputs an invalid argument.
if not newRow:
print("")
print("You inputted an invalid argument. Don't do that.")
print("Please, input the row correctly.")
print("Only integers, fractions and floats (decimals) are accepted. No parenthesis please.")

    Comment : User gives more values than there is in the first row.
```

```
elif len(rows) > 0 and len(newRow) != len(rows[0]):
print("")
print("You inputted an invalid amount of numbers.")
print("Please, input the row correctly.")

    else:
rows.append(newRow)

    row = my_strip(__ask_input("row: "))

    Comment : No rows given.
if len(rows) == 0:
return None

    ret = Matrix(rows, len(rows), len(rows[0]))
return ret

    def parseOperator():
"""Ask user, which operation they would like to perform next."""

    print("")

    print("Which operation you'd like to perform next?")

    print("")

    print("det: Calculate the determinant of the current matrix.")
Comment : print("inverse: Invert the given matrix if possible.")

    print("print: Print the current matrix.")

    print("")

    operator = my_lower(my_strip(__ask_input("Operator: ")))
while operator not in ["det", "inverse", "invert", Comment : Alternative inputs for inversion.
"-1", Comment : "print", ]:
print("Please choose a valid operator.")
operator = my_lower(my_strip(__ask_input("Operator: ")))

    return operator

    def askToContinue():
"""Ask the user to continue performing operations on the current matrix."""

    print("")
print("Do you want to apply more options to the current matrix?")

    a = my_lower(__ask_input("Y/N: "))
```

```
print("")

    while(a != "y" and a != "n"):
a = my_lower(__ask_input("Y/N: "))

    if a == "y":
return True
return False
```

Chapter 5

APPLICATIONS

Some of applications of matrices in computer science :

- In computer based applications, matrices play a vital role in the projection of three dimensional image into a two dimensional screen, creating the realistic seeming motions.
- Stochastic matrices and Eigen vector solvers are used in the page rank algorithms which are used in the ranking of web pages in Google search.
- The matrix calculus is used in the generalization of analytical notions like exponentials and derivatives to their higher dimensions.
- One of the most important usages of matrices in computer side applications are encryption of message codes.
- Matrices and their inverse matrices are used for a programmer for coding or encrypting a message.
- A message is made as a sequence of numbers in a binary format for communication and it follows code theory for solving. Hence with the help of matrices, those equations are solved.
- With these encryptions only, internet functions are working and even banks could work with transmission of sensitive and private data's.

Chapter 6

REFERENCES

1. *https://en.wikipedia.org/wiki/Matrix_(mathematics)*
2. *https://en.wikipedia.org/wiki/LU_decomposition*
3. **Linear Algebra and Its Applications, Fourth Edition (2007)**
   by *Gilbert Strang*