Национальный исследовательский университет
"Московский Энергетический Институт"

Курсовая работа по дисциплине "Численные методы"

Рациональные числа

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»

Научный руководитель:

Выполнили:

Москва
2023 г.

Содержание

Введение

Задачи численного решения тех или иных математических задач возникают во многих инженерных, технических задачах. В качестве примера можно привести такие популярные библиотеки для машинного обучения как *Tensorflow*, *Pytorch*, *CatBoost*. В них особенно важную часть занимает численное решение СЛАУ. Поэтому очень важно иметь специализированные математические пакеты, способные решать данного вида задачи. Наша курсовая работа посвящена разработке библиотеки на языке *C++*, содержащую в себе реализацию класса рациональных чисел и методы численного решения различных математических задач

Актуальность заявленной темы заключается в том, что каждый день многих специалисты, такие как инженеры, ученые, программисты и другие сталкиваются в своей работе с различными математическими задачи. И поэтому наличия специализированного математического пакета, который позволяет работать с рациональными числами и решать с помощью них различные задачи могло бы во многом упростить работу таким людям.

Основная цель работы: разработать математический пакет, содержащий в себе реализацию класса рациональных чисел и решение различных математических задач с помощью написанного класса.

Задачи:

- Реализовать класс рациональных чисел
- С помощью написанного класса реализовать возведение числа в рациональную степень
- Реализовать решения СЛАУ $Ax = b$ с помощью LU - разложения, используя написанный класс
- Реализовать функции в виде кубического сплайна для синуса и косинуса, вычисляющие значения рационального числа с точностью 10^{-12}

Глава 1. Реализация класса рациональных чисел

1.1 Длинная арифметика

Для реализация класса рациональных чисел необходимо реализовать класс для длинной арифметики целых чисел. В данной секции пойдет речь о нашей реализации данного класса. Реализация во многом будет основана на примере из книг [**?**], [**?**]

1.1.1 Хранение чисел

Для хранения числа мы используем массив, в котором число храниться в перевернутом в виде

Пример: Число 1234 будет храниться в массиве $[4, 3, 2, 1]$

О преимуществах такого подхода будет написано ниже. Также для хранения знака числа мы завели булевую переменную *is_signed*. Она истина, если число отрицательное и ложна в противном случае

Псевдокод нашего класса

```
class BigInt:
    num: array // Array contains num
    is_signed: bool // Bool variable contains sign num
```

1.1.2 Арифметические операции с числами

Сложение чисел

Сложение чисел осуществляется по школьным правилам, складываем каждый разряд числа и если возникает перенос, то запоминаем его и переносим в следующие разряд. Вот тут и проявляется преимущество подхода хранения числа в перевернутом в виде. Так как если перенос возникнет в самом конце, то будет достаточно добавить в массив новый элемент в конец. Данная операция как известно имеет алгоритмическую сложность *O(1)*. И если хранить число, в том виде, в котором оно записано, то надо будет добавлять элемент в начало массива и данная операция уже как известно занимает *O(n)*, где *n* - длина массива.

Псевдокод данного алгоритма

```
function sum_nums(lhs: BigInt, rhs: BigInt):
    carry = 0
    res = []
    for i = 0 ... min(lhs.num.size(), rhs.num.size()):
        s = lhs.num[i] + rhs.num[i] + carry
        res.append(s % 10)
        carry = s / 10
    for i = min(lhs.num.size(), rhs.num.size())
        ... max(lhs.num.size(), rhs.num.size()):
        if lhs.num.size() > rhs.num.size():
            s = lhs.num[i] + carry
            res.append(s % 10)
            carry = s / 10
        else:
            s = rhs.num[i] + carry
            res.append(s % 10)
            carry = s / 10
    return res
```

Алгоритмическая сложность данного алгоритма очевидно, что составляет $O(n)$

Вычитание чисел

Вычитание числа мы также будет производить по школьным правилам. Вычитаем поразрядно и если возникает отрицательное значение в каком то отряде, то очевидно что надо это будет учесть в следующем разряде и вычесть из него единицу.

Псевдокод данного алгоритма

```
function diff_nums(lhs: BigInt, rhs: BigInt):
    carry = 0
    res = []
    for i = 0 ... min(lhs.num.size(), rhs.num.size()):
        s = lhs.num[i] − rhs.num[i] − carry
```

```
            if s < 0:
                s += 10
                carry = -1
            else:
                carry = 0
            res.append(s % 10)

        for i = min(lhs.num.size(), rhs.num.size())
            ... max(lhs.num.size(), rhs.num.size()):
            if lhs.num.size() > rhs.num.size():
                s = lhs.num[i] - carry
                if s < 0:
                    s += 10
                    carry = -1
                else:
                    carry = 0
                res.append(s % 10)
            else:
                s = rhs.num[i] - carry
                if s < 0:
                    s += 10
                    carry = -1
                else:
                    carry = 0
                res.append(s % 10)

        return res
```

Очевидно, что алгоритмическая сложность данного алгоритма составляет *O(n)*

Умножение чисел

Распишем два числа lhs, rhs в виде суммы их разрядов

$$lhs = lhs[0] \cdot 10^0 + lhs[1] \cdot 10^1 + \cdots + lhs[n] \cdot 10^n$$

$$rhs = rhs[0] \cdot 10^0 + rhs[1] \cdot 10^1 + \cdots + rhs[m] \cdot 10^m$$

Теперь распишем результат их умножения

$$\begin{aligned} lhs \cdot rhs = \big(lhs[0] \cdot 10^0 + lhs[1] \cdot 10^1 + \cdots + lhs[n] \cdot 10^n\big) \cdot \\ \cdot \big(rhs[0] \cdot 10^0 + rhs[1] \cdot 10^1 + \cdots + lhs[m] \cdot 10^m\big) = \\ = lhs[0] \cdot rhs[0] + lhs[0] \cdot rhs[1] \cdot 10^1 + \cdots + lhs[0] \cdot rhs[m] \cdot 10^m + \\ + \cdots + lhs[n] \cdot rhs[0] \cdot 10^n + lhs[n] \cdot rhs[1] \cdot 10^{n+1} + \cdots + lhs[n] \cdot rhs[m] \cdot 10^{n+m} \end{aligned}$$

Тогда получаем что результат умножения чисел на $i + j$ разряде будет равен $res[i + j] = lhs[i] \cdot rhs[j]$, где i изменяется от 0 до n, j от 0 до m

Напишем псевдокод данного алгоритма

```
function product_nums(lhs: BigInt, rhs: BigInt):
    res = [0] * (lhs.size() + rhs.size())
    carry = 0
    for i = 0 ... lhs.size():
        for j = 0 ... rhs.size():
            res[i + j] += carry + lhs[i] * rhs[j]
            carry = res[i + j] / 10
            res[i + j] %= 10
    return res
```

Очевидно, что сложность данного алгоритма $O(n^2)$. Хотелось бы получить улучшение данного алгоритма с точки зрения алгоритмической сложности. Одно из таких улучшений это **Алгоритм быстрого умножения Карацубы**

Пусть имеем два числа A, B одинаковой длины n и основанием в системе счисления t.

Распишем число A подробно

$$
\begin{aligned}
A = a_{n-1} \cdot t^{n-1} + a_{n-2} \cdot t^{n-2} + \cdots + a_1 \cdot t + a_0 = \\
= \left(a_{n-1} \cdot t^{n-1} + a_{n-2} \cdot t^{n-2} + \cdots + a_{n//2} \cdot t^{n//2}\right) + \\
+ \left(a_{n//2-1} \cdot t^{n//2-1} + a_{n//2-2} \cdot t^{n//2-2} + \ldots a_0\right) = \\
= t^{n//2}\left(a_{n-1} \cdot t^{n//2-1} + \cdots + a_{n//2}\right) + \\
+ \left(a_{n//2-1} \cdot t^{n//2-1} + \ldots a_0\right) = Pt^{n//2} + Q \\
P = a_{n-1} \cdot t^{n//2-1} + a_{n-2} \cdot t^{n//2-2} + \cdots + a_{n//2} \\
Q = a_{n//2-1} \cdot t^{n//2-1} + a_{n//2-2} \cdot t^{n//2-2} + \ldots a_0
\end{aligned}
$$

$n//2$ - целая часть от деления числа на число 2. Такие же преобразования можно провести и с числом B. Запишем его в таком же в виде как A. $B = Rt^{n//2} + S$

Теперь перемножим числа в таком виде

$$
\begin{aligned}
AB = \left(Pt^{n//2} + Q\right)\left(Rt^{n//2} + S\right) = \\
= PR + (PS + QR)\, t^{n//2} + QS
\end{aligned}
$$

Имеем на данном этапе, что мы разбили умножение двух чисел на три умножения и два сложения. Применим следующий прием. Введем следующие обозначения

$$
\begin{aligned}
X = (P + Q)(R + S) = PR + PS + QR + QS \\
Y = PR \\
Z = QS
\end{aligned}
$$

Тогда в наших обозначениях получим следующую формулу для умно-

жения чисел

$$AB = Yt^n + (X - Y - Z)t^{n//2} + Z$$

В итоге имеем 6 операций сложения(вычитания) и 3 операции умножения

Покажем на простом примере как работает данный алгоритма

Пример: Перемножим два числа $A = 1234$, $B = 5678$

$P = 12$

$Q = 34$

$R = 56$

$S = 78$

$X = (P + Q)(R + S) = (12 + 34)(56 + 78) = 46 \cdot 134 = 6164$

$Y = PR = 12 \cdot 56 = 672$

$Z = QS = 34 \cdot 78 = 2652$

$X - Y - Z = 6164 - 2652 - 672 = 2840$

$AB = 672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 7006652$

На примере был показан не полный алгоритм Карацубы, так как можно дальше продолжить рекурсию, для подсчетов 3-х произведений.

Напишем псевкодод данного алгоритма

```
function product_nums2(A: BigInt, B: BigInt):
    if A.num.size() == 1 or B.num.size() == 1:
        return product_nums1(A, B)

    n = max(A.num.size(), B.num.size())
    half = n // 2
    P = a[n // 2:]
    Q = a[:n // 2]
    R = b[n // 2:]
```

```
S = b[n // 2:]
X = product_nums2(sum_nums(P, Q), sum_nums(R, S))
Y = product_nums2(P, R)
Z = product_nums2(Q, S)

s1, s2, s3 = [0] * n * 2, [0] * n * 2, [0] * n * 2
for n + len(Y) - 1 ... n:
    s1[i] = Y[i - n - 1]

diff = diff_nums(X, diff_nums(Y, Z))

for n // 2 + len(diff) - 1 ... n // 2:
    s2[i] = diff[i - n // 2]

for i len(Z) - 1 ... 0:
    s3[i] = Z[i]

return sum_nums(s1, sum_nums(s2, s3))
```

Сложность данного алгоритма $O(n^{log_2 3})$

Деление чисел

Для деления числа, воспользуемся следующей идеей. Пусть мы делим число A на число B и получаем некоторое число C. Можно утверждать, что количество цифр в данном числе будет порядка $Count\_digits(A) - Count\_digits(B) + 1$. Где $Count\_digits(.)$ - количество цифр в данном числе

В числе $C = c_{n-1}c_{n-2}...c_0$ будем перебирать его разряды c_{n-1} до c_0 пока не получим уменьшения числа до нужного.

Псевдокод данного алгоритма

```
function divide_nums(a: BigInt, b: BigInt):
    res = [0] * (a.num.size() - b.num.size())
    for i = res.size() - 1 ... 0:
        while product_nums(b, res) < a:
            res.num[i] += 1
```

```
            res.num[i] -= 1
        return res
```

1.1.3 Операции сравнения

Операции сравнения похожи своей реализацией друг на друга. Поэтому приведем в данном примере реализацию только операции меньше. Сначала мы сравниваем два числа по длине, очевидно, что если левое число меньше по длине то результат операции истинен, иначе ложен Далее мы сравниваем числа поразрядно, если нашли несовпадение в соответствующих разрядах, то уже можно сразу делать вывод об результате операции. Псевдокод данного алгоритма

```
function less(a: BigInt, b: BigInt):
    if a.num.size < b.num.size:
        return true
    else if a.num.size > b.num.size:
        return false

    for i = 0 ... a.num.size():
        if a[i] > b[i]:
            return false
    return true
```

Очевидно, что сложность данной операции *O(n)*

1.1.4 Реализация поиск наименьшего общего кратного и наибольшего общего делителя

Для работы с рациональными числами необходимо уметь находить наименьшее общее кратное и наибольший общий делитель.

Для поиска наибольшего общего делителя был выбран алгоритм Евклида, известный всем еще со школы. Алгоритм основывается на том факте, что $GCD(A, B) = GCD(A - B, B)$ Где $GCD(A, B)$ - наибольший общий

делитель чисел A и B. Получается мы можем вычитать из большего числа меньшее, пока не получим результат, что они сравнялись

Псевдокод данного алгоритма

```
function GCM(A: BigInt, B: BigInt):
    while A != B:
        if A > B:
            A -= B
        else:
            B -= A
    return A
```

Для реализации наименьшего общего кратного воспользуемся следующей формулой $LCM(A,B) = \frac{AB}{GCD(A,B)}$

Псевдокод данного алгоритма

```
function LCM(A: BigInt, B: BigInt):
    return (A * B) / GCD(A, B)
```

1.2 Реализация рациональных чисел

Рациональное число мы храним в виде набора двух элементов класса *BigInt* Храним числитель(*numerator*) и знаменатель (*denominator*).

1.2.1 Хранение рационального числа

Псевдокод данного класс

```
class Rational:
    numerator: BigInt
    denominator: BigInt
```

1.2.2 Реализация арифметических операций

Нормализация числа

Для реализации арифметических операций, нам понадобятся вспомогательная операция нормализация числа, то есть приведение дроби к несо-

кращаемому виду. Для этого нам нужно найти наибольший обший делитель числителя и знаменателя и поделить на него числитель, знаменатель

```
function normalize(r: Rational):
    gcd = GCD(r.numerator, r.denominator)
    r.numerator /= gcd
    r.denominator /= gcd
```

Сложение и вычитание

Для того, чтобы сложить или вычесть два рациональных числа. Необходимо найти наименьшее общее кратное двух знаменателей. И это будет знаменатель полученного числа, а потом умножить числитель каждого числа на наименьшее общее кратное двух чисел поделить на знаменатель

Псевдокод функции сложения

```
function sum(r1: Rational, r2: Rational):
    lcm = LCM(r1.denominator, r2.denominator)
    function coef(r: rational):
        return lcm / r.denominator
    numerator = r1.numerator * coef(r1) + r2.numerator *
    return normalize(Rational(numerator, lcm))
```

Псевдокод функции вычитания

```
function diff(r1: Rational, r2: Rational):
    lcm = LCM(r1.denominator, r2.denominator)
    function coef(r: rational):
        return lcm / r.denominator
    numerator = r1.numerator * coef(r1) 0 r2.numerator *
    return normalize(Rational(numerator, lcm))
```

Умножение

Операцию умножения реализуется по школьному правилу, считает произведение числителя на числитель, знаменателя на знаменатель. Псевдокод данного алгоритма

```
function product(r1: Rational, r2: Rational):
    return normalize(Rational(r1.numerator * r2.numerator,
```

```
    r1.denominator * r2.denominator))
```

Деление

Деление, умножаем одну дробь на другую перевернутую.

Псевдокод данного алгоритма

```
function divide(r1: Rational, r2: Rational):
    return normalize(Rational(r1.numerator * r2.denominator
        r1.denominator * r2.numerator))
```

Глава 2. Реализация возведения в степень

По определению, чтобы возвести число a в рациональную степень $\frac{n}{m}$ необходимо $a^{\frac{n}{m}} = \sqrt[m]{a^n}$

Получается необходимо реализовать возведения числа в целую n. Это можно сделать с помощью **алгоритма быстрого возведения в степень** [**?**]

Число a^n можно представить следующим образом

$$a^n = \begin{cases} (a^2)^{\frac{n}{2}}, & \text{n - четное} \\ a \cdot (a^2)^{\frac{n-1}{2}}, & \text{n - нечетное} \end{cases}$$

Следовательно, можно с помощью указанного формулы сократить количество умножений. То есть получили алгоритм, работающий за *O($log(n)$)*

Напишем псевдокод данного алгоритма

```
function pow(basis: Rational, exponent: int)
    if exponent == 0:
        return Rational(1, 1)

    if exponent % 2 != 1:
        return pow(basis, exponent - 1) * basis

    buffer = pow(basis, exponent >> 1)
    return buffer * buffer
```

Теперь разберемся как найти корень из числа

Для этого воспользуемся методом Ньютона для решения нелинейного уравнения [1], он имеет следующую расчетную формулу $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

Где x_n, x_{n+1} - предыдущее и следующее приближения.

Данный метод имеет следующий критерий окончания $|x_{n+1} - x_n| < \varepsilon$ Где ε - точность, с которой необходимо найти корень

$x = \sqrt[n]{a}$.

Приведем уравнение к следующему виду

$x^n - a = 0$

Тогда мы можем записать следующую расчетную формулу для метода

Ньютона

$$f(x) = x^n - a$$
$$f'(x) = nx^{n-1}$$
$$x_{n+1} = x_n - \frac{x_n^n - a}{nx_n^{n-1}}$$

Теперь запишем псевдокод данного алгоритми

```
function root(basis: Rational, root: int)
    precision = Rational(1, 10**12)
    prev = basis / 2
    cur = prev -
        - (prev**root - basis) /
            / (root * prev**(root - 1))
    while abs(prev - cur) > precision:
        prev = cur
        cur =  prev - (prev**root - basis) /
          / (root * prev**(root - 1))
    return cur
```

Таким образом общая функция будет иметь вид

```
function pow(basis: Rational, exponent: Rational)
    return root(pow(basis, exponent.numerator),
        exponent.denominator)
```

Глава 3. Реализация синуса и косинуса с помощью сплайна

Глава 4. Реализация решения СЛАУ с помощью LU - разложения

Вспомним для начала алгоритм всем известного метода Гаусса [1]

Пусть дана СЛАУ $Ax = b$. Матрица A является квадратной порядка n и матрица системы является невырожденной, то есть $\det A \neq 0$. Тогда как известно из курса линейной алгебры система имеет единственное решение и его можно найти по следующему алгоритму.

Прямой ход метода Гаусса

Всего будет проделано $n - 1$ шагов

На k-ом шаге проделываем следующие операции

$$\mu_{i,k} = \frac{a_{i,k}}{a_{k,k}}$$
$$a_{i,j} = a_{i,j} - \mu_{i,k} a_{k,j}$$
$$b_i = b_i - \mu_{i,k} b_k$$
$$i = k, k+1, \ldots, n$$
$$j = 1, 2, \ldots, n$$

Прямой ход метода Гаусса можно представить следующим образом. [1] Введем матрицы M_1, M_2, ..., M_{n-1}, называемыми **элементарными матрицами**

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ -\mu_{2,1} & 1 & 0 & \ldots & 0 \\ -\mu_{3,1} & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ -\mu_{n-1,1} & 0 & 0 & \ldots & 1 \\ -\mu_{n,1} & 0 & 0 & \ldots & 1 \end{pmatrix}, \ldots, M_{n-1} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & -\mu_{n,n-1} & 1 \end{pmatrix}$$

Тогда прямой код метода Гаусса после 1-шага можно представить в виде $M_1 Ax = M_1 b$ проверяется непосредственной подстановкой

На втором шаге $M_2M_1Ax = M_2M_1b$

Всего таких умножений будет $n-1$ И система примет вид

$$M_{n-1}\dots M_2M_1Ax = M_{n-1}\dots M_2M_1b$$

Матрица системы является верхнетреугольной обозначим ее U

$$M_{n-1}\dots M_2M_1A = U$$

Тогда используя понятия обратной матрицы получим

$$A = M_1^{-1}\dots M_{n-2}^{-1}M_{n-1}^{-1}U$$

Если исследовать свойства элементарных матриц, то очевидно, что все они невырожденные. Поэтому к ним существуют обратные матрицы, которые имеют вид

$$M_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ \mu_{2,1} & 1 & 0 & \dots & 0 \\ \mu_{3,1} & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \mu_{n-1,1} & 0 & 0 & \dots & 1 \\ \mu_{n,1} & 0 & 0 & \dots & 1 \end{pmatrix}, \dots, M_{n-1}^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & \mu_{n,n-1} & 1 \end{pmatrix}$$

Проверить, что данные матрицы являются обратными можно также непосредственной подстановкой. Результирующая обратная матрица тогда будет иметь вид

$$M_1^{-1}\dots M_{n-1}^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ \mu_{2,1} & 1 & 0 & \dots & 0 \\ \mu_{3,1} & \mu_{3,2} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \mu_{n,1} & \mu_{n,2} & \mu_{n,3} & \dots & 1 \end{pmatrix}$$

Обозначим ее $L = M_1^{-1}\dots M_{n-1}^{-1}$. Получим тогда так называемое LU-

разложение матрицы $A = LU$

Приведем пример как оно работает

Пример: Дана матрица A. Найти ее LU- разложение

$$A = \begin{pmatrix} 9 & 5 & 3 \\ -81 & -50 & -25 \\ 45 & 55 & 7 \end{pmatrix}$$

Решение:

На первом шаге имеем ведущий элемент $a_{1,1} = 9$. И следующие коэффициенты $\mu_{2,1} = -9, \mu_{3,1} = 5$

$$A = \begin{pmatrix} 9 & 5 & 3 \\ -81 & -50 & -25 \\ 45 & 55 & 7 \end{pmatrix} \sim \begin{pmatrix} 9 & 5 & 3 \\ 0 & -5 & 2 \\ 0 & 30 & -8 \end{pmatrix}$$

На втором шаге имеем имеем ведущий элемент $a_{2,2} = -5$ и следующий коэффициент $\mu_{3,2} = -6$. Получим тогда следующую систему

$$A = \begin{pmatrix} 9 & 5 & 3 \\ 0 & -5 & 2 \\ 0 & 30 & -8 \end{pmatrix} \sim \begin{pmatrix} 9 & 5 & 3 \\ 0 & -5 & 2 \\ 0 & 0 & 4 \end{pmatrix}$$

Тогда получим следующим матрицы L, U

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -9 & 1 & 0 \\ 5 & -6 & 1 \end{pmatrix}, U = \begin{pmatrix} 9 & 5 & 3 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{pmatrix}$$

Теперь разберем как решать систему с помощью найденного LU-разложения

Имеем следующую СЛАУ $LUx = b$. Введем обозначение $y = Ux$. Тогда сначала мы можем решить СЛАУ $Ly = b$. А потом найдя вектор y смрдем решить СЛАУ $Ux = y$. Таким образом мы найдем вектор x.

Теперь напишем псевдокод LU-разложения

```
function LUdecomposition(A: array<array<Rational>>):
    n = A[0].size
    U = A
```

```
    L = [[0] * n] * n
    for k = 0 ... n - 2:
        for i = k + 1 ... n - 1:
            mu_i_k = U[i][k] / U[k][k]
            L[i][k] = mu_i_k
            U[i][k] -= mu_i_k * U[k][k]
            for j = k + 1 ... n - 1:
                U[i][j] -= mu_i_k * U[k][j]
    return L, U
```

Очевидно, что сложность LU-разложения $O(n^3)$

Теперь напишем функцию решающую систему с помощью найденного LU разложения

```
function solve_system_usingLUdecomposition(L,
            U: array<array<Rational>>,
            b: array<Rational>):
    n = L[0].size
    y = [0] * n
    y[0] = b[0]
    for i = 1 ... n - 1:
        y[i] = b[i]
        for j = 1 ... i - 1:
            y[i] -= L[i][j] * y[j]


    x = [0] * n
    x[n - 1] = y[n - 1] / U[n - 1][n - 1]
    for i = n - 2 ... 0:
        cur = y[i]
        for j = n - 1 ... i + 1:
            cur -= U[i][j] * x[j]
        x[i] = cur / U[i][i]
    return x
```

Очеивдно, что сложность решения системы с помощью LU-разложения

составляет $O(n^2)$. Основное преимущество LU-разложения состоить в том, что мы можем найти один раз данное разложение для матрицы A, а потом решать ее для различных вектором b

Заключение

В результате работы над данным математическим пакетом. Мы получили полноценно работающую библиотеку, которую можно использовать для решения различных математических задач. Были проведены юнит-тесты с использованием библиотеки *GoogleTest* [4], которые показали работоспособность нашей библиотеки. Также для того, чтобы убрать платформу-зависимость при сборке весь проект был упакован в *docker container* [5]. В результате для сборки проекта достаточно иметь на своем системе программу *docker*. В будущих планах упаковать нашу библиотеку в *deb-package* [6]. Для того чтобы иметь возможность установить ее с помощью стандартного пакетного менеджера *apt Ubuntu* и других дистрибутивов *Linux*. С исходным кодом нашей библиотеки можно ознакомиться по следующей ссылке `https://github.com/mpeicoursework/course_work.git`

Список литературы

[1] Амосов А. А., Дубинский Ю. А., Копченова Н. В. Вычислительные методы. — 2014.

[2] Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн - Алгоритмы. Построение и анализ - 2013.

[3] Donald E. Knuth - The Art of Computer Programming. Volume 1 - 3 - 1998.

[4] `http://google.github.io/googletest/`

[5] `https://www.docker.com/`

[6] `https://en.wikipedia.org/wiki/Deb_(file_format)`