МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

"Московский технологический университет"

МИРЭА

Институт кибернетики

(наименование института)

Кафедра высшей математики

(наименование кафедры)

КУРСОВОЙ ПРОЕКТ (РАБОТА)

по дисциплине

«Языки и методы программирования»

(наименование дисциплины)

Тема курсового проекта (работы) «Моделирование решения СЛАУ»

(наименование темы)

Студент группы КМБО-01-15 *Шарипов С.Н.*

(учебная группа) *(Фамилия И.О.)*

Руководитель курсового проекта (работы) *Ассистент кафедры Высшей математики,*
должность, звание, ученая степень *к.ф.-м.н. Петрусевич Д.А.*

(Фамилия И.О.)

Рецензент (*при наличии*)
должность, звание, ученая степень *(Фамилия И.О.)*

Работа представлена к защите «__»_______201___ г. *(подпись студента)*

«Допущен к защите» «__»_______201___ г. *(подпись руководителя)*

Москва 201_

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
"Московский технологический университет"

МИРЭА

Институт Кибернетики

Кафедра высшей математики

Утверждаю
Заведующий кафедрой______________*Ю.И.Худак*
«____» _________201_ г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Языки и методы программирования»

Студент Шарипов Салават Наилевич
Группа КМБО-01-15

1. **Тема: «Моделирование решения СЛАУ»**
2. **Исходные данные:**
   - Система линейных уравнений
3. **Перечень вопросов, подлежащих разработке, и обязательного графического материала:**
   - Класс Матрица, класс Дробь
   - Преобразование матрицы методом Гаусса-Жордана
   - Решение СЛАУ по заданной матрице коэффициентов и столбцу значений
4. **Срок представления к защите курсовой работы: до** « » __________ 201_ г.

Задание на курсовой проект выдал	«___»_____201_ г.	____________________	(____________________)
Задание на курсовой проект получил	«___»_____201_ г.	____________________	(____________________)

Введение

Решение линейных систем алгебраических уравнений одна из основных задач линейной алгебры, возникающая во многих отраслях: в экономике, физике, химии, математическом моделировании. Данная задача эквивалентна поиску прообраза вектора некоторого линейного преобразования. Соответсвенно решение, если оно есть, может быть вектором, в случае если преобразование инъективно или некоторым линейным многообразием, порожденным ядром линейного преобразования, в случае, когда ядро не тривиально.

Соответсвенно любой СЛАУ:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots a_{mn}x_n = b_m \end{cases}$$

Мы можем сопоставить матричное уравнение:

$$Ax = b$$

Где А - матрица составленная из коэффицентов СЛАУ, а b - вектор составленный из $\{b_i\}$.

Для нахождения x воспользуемся методом Гаусса-Жордана. Т.е. приведем расширенную матрицу $[A|b]$ элементарными преобразованиями над строками к каноническому ступенчатому виду $[A'|b']$, из которого далее легко извлечь ответ: (если система не вырождена)

$$x \in X = b' + Kern(A)$$

где $Kern(A)$ - ядро оператора, связанного с нашей матрицей. (Ядро легко находится из приведенной матрицы)

Для этого реализуем класс на C++ для матрицы над произвольным полем(с помощью шаблонов). И в качестве примера поля возьмем рациональные числа - реализуем класс для дробей(в отличии от float и double, позволяет избежать погрешностей).

В качестве полидрома для испытаний я выбрал Jupter Notebook, т.к. это удачная интерактивная среда, которая умеет отображать tex формулы, что намного интереснее, чем смотреть на матрицу из дробей в обычной консоли. Поэтому так-же далее будет написан класс-обертка на cython для создания Python модуля и дополнительные методы для генерации tex выражений.

Классы

Так-как достаточно часто придется пробегать по элементам матрицы, определим следующий макрос - синтаксический сахар для for:

```
#define range(i, begin, end) for(size_t i = (begin); i < (end); i++)
```

Matrix | Класс для матрицы

Рассмотрим определение класса **Matrix**:

```cpp
template <typename Field>
class Matrix {
    vector< vector<Field>* > _M;
    Matrix& self = *this;
    void clear();

public:
    Matrix(size_t rows, size_t cols);
    Matrix& operator = (const Matrix& A);
    Matrix& operator = (Matrix&& A);
    Matrix(const Matrix &A);
    Matrix(Matrix&& A);
    ~Matrix();

    // Индексация
    Field& operator() (size_t i, size_t j) const;

    // Элементарные преобразования
    void add(size_t to, size_t from, Field k);
    void swap(size_t first, size_t second);
    void mul(size_t row, Field k);
    void div(size_t row, Field k);

    // Метод Гаусса-Жордана
    pair< bool, vector<bool> > row_reduce(vector<Field>& b);

    // Работа с потоками
    friend istream& operator >> (istream& in, Matrix &M);
    friend ostream& operator << (ostream& out, const Matrix &M);

    // Операторы
    friend Matrix operator * (const Matrix& A, const Matrix& B);
    friend Matrix operator + (const Matrix& A, const Matrix& B);
    friend Matrix operator - (const Matrix& A);
    friend Matrix operator - (const Matrix& A, const Matrix& B);
    friend bool operator == (const Matrix& A, const Matrix& B);
    friend bool operator != (const Matrix& A, const Matrix& B);

    // Данный метод создан для
    void set(size_t i, size_t j, const Field& val);

    string tex_solution(vector<Field>& b);
    string str();

};
```

Решение использовать массив из указателей на вектор связаны с частой необходимостью переставлять строки местами. Гораздо проще переставить местами 2 указателя, чем перемещать целые массивы.

Поле **self** используется исключительно для удобства обращения к объекту:

```
(*this)(i, j); // не очень
self(i, j);    // уже лучше
```

И, так как освобождать память нужно будет не только в конструкторе, объявлен **clear()**:

```
    void clear() {
        for(auto& row: _M) delete row;
    }
```

Конструктор по размеру матрицы:

```
    Matrix(size_t rows, size_t cols) {
        _M.resize(rows);
        for(auto& row: _M) row = new vector<Field>(cols);
    }
```

Выделяем память под строки, устанавливая размеры массивов, исходя из параметров.

Далее копирующий оператор присваивания и конструктор:

```
    Matrix& operator = (const Matrix& A) {
        clear();
        _M.resize(A.rows());
        range(i, 0, A.rows()) _M[i] = new vector<Field>(*(A._M[i]));
        return self;
    }

    Matrix(const Matrix &A) {
        self = A;
    }
```

Они необходимы, т.к. мы работаем с динамической памятью, а значит нужно явно описать процесс копирования, т.к. копировать значение указателей это далеко не лучшая идея, а компилятор не знает, что мы там напридумывали. Теперь, имеем возможность передавать по значению и возвращать значения из функций.

Перемещающий опертор присваивания и конструктор:

```
Matrix& operator = (Matrix&& A) {
    clear();
    _M = A._M;
    A._M.resize(0);
    return self;
}

Matrix(Matrix&& A) {
    _M = A._M;
    A._M.resize(0);
}
```

Воспользуемся прелестями c++11 и определим поведение с rvalue. Зачем копировать объект, когда он вот-вот будет уничтожен? Можно просто забрать его ресурсы и сэкономить время на копировании и удалении.

Тяжело работать с матрицами, размеры которых не знаешь:

```
size_t rows() const {
    return _M.size();
}

size_t cols() const {
    if(rows() == 0) return 0;
    return _M[0]->size();
}
```

Полезно будет иметь двумерную индексацию:

```
Field& operator() (size_t i, size_t j) const {
    return (*_M[i])[j];
}
```

Я не буду приводить здесь все операторы, разберу только умножение матрицы, как самое нетривиальное. Пусть:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}, B = \begin{bmatrix} b_{11} & \dots & b_{1p} \\ \dots & \dots & \dots \\ b_{n1} & \dots & b_{np} \end{bmatrix}, C = A * B = \begin{bmatrix} c_{11} & \dots & c_{1p} \\ \dots & \dots & \dots \\ c_{m1} & \dots & c_{mp} \end{bmatrix}$$

Тогда элемент матрицы C находится как:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Естественно, нужно учесть согласование размеров:

```
    friend Matrix operator * (const Matrix& A, const Matrix& B) {
        if(A.cols() != B.rows()) {
            throw "Sizes don't match";
        }
        Matrix C(A.rows(), B.cols());
        range(i, 0, A.rows()) {
            range(j, 0, B.cols()) {
                range(k, 0, A.cols()) {
                    C(i, j) += A(i, k) * B(k, j);
                }
            }
        }
        return C;
    }
```

Элементарные преобразования со строками:

Самые необходимые методы, для приведения матрицы к каноническому ступенчатому виду.

Метод для сложения строк. К заданной строке **to** прибавляется строка **from** домноженная на **k**:

```
    void add(size_t to, size_t from, Field k) {
        range(j, 0, cols()) self(to, j) += k * self(from, j);
    }
```

Метод для перестановки строк. Меняем строки с номерами **first** и **second** местами:

```
    void swap(size_t first, size_t second) {
        std::swap(_M[first], _M[second]);
    }
```

Метод для умножения строки с номером **row** на скаляр **k**:

```
    void mul(size_t row, Field k) {
        range(j, 0, cols()) self(row,j) = self(row, j) * k;
    }
```

Метод для деление строки с номером **row** на скаляр **k**

```
    void div(size_t row, Field k) {
        range(j, 0, cols()) self(row, j) = self(row,j) / k;
    }
```

Приведение матрицы медотом Гаусса-Жордана:

Самый важный метод, ради чего все и задумывалось:

Изначально у нас есть расширенная матрица:

$$\left(\begin{array}{ccc|c} a_{11} & \dots & a_{1n} & b_1 \\ \dots & \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} & b_m \end{array}\right)$$

Пытаемся найти ненулевой элемент a_{i1}(далее буду называть опорным элементом и буду считать их кол-во счетчиком **p**) и если не находим, то переходим к следующему столбцу, если же нашли, то меняем первую строку и строку номером **i** местами, после чего делим первую строку на найденное значение и обнуляем оставшиеся значения в столбце, а также увеличиваем счетчик **p**.

> Важное замечание. Далее, я буду подразумевать под a_{ij} не изначальное значение, а именно значение в данной позиции в матрице на данном этапе. Т.е. каждое a_{ij} меняет свое значение от шагу к шагу.

Получаем следующую матрицу(в случае, если нашелся ненулевой элемент в первом столбце):

$$\left(\begin{array}{cccc|c} 1 & a_{12} & \dots & a_{1n} & b_1 \\ 0 & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & a_{m2} & \dots & a_{mn} & b_m \end{array}\right)$$

Или остается неизменной, если первый столбец был нулевым(при этом **p**=0):

$$\left(\begin{array}{cccc|c} 0 & a_{12} & \dots & a_{1n} & b_1 \\ 0 & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & a_{m2} & \dots & a_{mn} & b_m \end{array}\right)$$

Далее процесс повторяется для каждого **j** столбца, за тем исключением, что ненулевой элемент a_{ij} ищется такой, что $p \leq i$. Если же элемент в столбце не нашлеся, то запоминаем номер столбца и переходим к следющему. Запомнить номера подобных столбцов важно, т.к. подобные столбцы далее войдут в ядро оператора, свзяанного с нашей матрицей, т.е. будут решением однородной системы.

В итоге процесса получаем каноническую ступенчатую матрицу. Она может выглядить как-то так:

$$\left(\begin{array}{ccccc|c} 0 & 1 & 0 & -1 & 0 & b'_1 \\ 0 & 0 & 1 & 3 & 0 & b'_2 \\ 0 & 0 & 0 & 0 & 1 & b'_3 \\ 0 & 0 & 0 & 0 & 0 & b'_4 \end{array}\right)$$

Если приведенная матрица содержит нулевые строки и этим нулевым строкам соотвествует ненуловое значение в приведенном столбце b', то, очевидно, что система не совместна, т.к. не существует линейной комбинацией из нулей порождающих ненулевое значение.

В ином же случае решение есть и оно выглядит следующим образом:

$$x \in X = \{b' + n \,|\, n \in Kern(A)\}$$

В частности, в случае тривиального ядра, получаем одно решение.

Функция возвращает вердикт о том, есть решение или нет в виде bool и возвращает bool вектор, хранящий информацию о том, какие столбцы войдут а какие нет в ядро. Значения возвращаются в паре:

```
    pair< bool, vector<bool> > row_reduce(vector<Field>& b) {
        if(b.size() != rows()) {
            throw "length of b don't match with quantity of rows";
        }
        vector<bool> pivots(cols(), false);

        size_t pivot = 0;
        range(k, 0, cols()) {
            range(i, pivot, rows()) {
                if(self(i,k) != Field()) {
                    pivots[k] = true;

                    // swap
                    std::swap(b[pivot], b[i]);
                    swap(pivot, i);

                    // div
                    b[pivot] /= self(pivot, k);
                    div(pivot, self(pivot, k));

                    // обнуляем все элементы в столбце, кроме опорного
                    range(ii, 0, rows()) {
                        if(i != ii) {
                            b[ii] -= b[pivot]*self(ii, k);
                            add(ii, pivot, -self(ii,k));
                        }
                    }
                    pivot++;
                    break;
                }
            }
        }
        bool there_is_a_solution = true;
        range(i, pivot, rows()) {
            if(b[i] != Field()) {
                there_is_a_solution = false;
            }
        }

        return make_pair(there_is_a_solution, pivots);
    }
```

Извлечение ответа

Приведенная к каноническому ступенчатому виду матрица - это конечно здорово. Но хотелось бы видеть ответ как-то так:

$$X = \begin{bmatrix} \frac{-1}{3} \\ \frac{2}{3} \\ 0 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} C_1$$

Воспользуемся tex для генерации ответа:

```
    string tex_solution(vector<Field>& b) {
        Matrix A(self);

        stringstream out;

        bool is_there_a_solution;
        vector<bool> pivots;

        tie(is_there_a_solution, pivots) = A.row_reduce(b);

        if(!is_there_a_solution) {
            out << "There is no soulution\n";
            return out.str();
        }

        out << "X = \n";
        out << "\\begin{bmatrix}\n";

        range(j, 0, A.cols()) {
            out << "    " << (j < b.size() ? b[j] : Field())
            << ((j != A.cols()-1) ? "\\\\\n" : "\n");
        }

        out << "\\end{bmatrix}\n";

        size_t numeration = 0;
        range(k, 0, pivots.size()) {
            if(!pivots[k]) {
                out << " + \n" << "\\begin{bmatrix}\n";
                range(i, 0, A.cols()) {
                    out
                    << "    "
                    << ((i == k) ? -Field(1): ( i < A.rows() ? A(i, k) :
 Field()))
                    << ((i != A.cols()-1) ? "\\\\\n" : "\n");
                }
                numeration++;
                out << "\\end{bmatrix}C_{" << numeration << "}\n" ;
            }
        }
        return out.str();
    }
```

Далее в модуле для Python, именно результат этой функции будет рендериться как решение.

Frac | Класс для дроби

Реализация для дробей выглядит следующим образом: (Полная реализация в исходниках в Приложении)

```
class Frac {
private:
    int _m;
    unsigned int _n;

    void simplify(int& m, unsigned& n) {...}

public:
    Frac(int m, unsigned n): _m(m), _n(n) {...}

    Frac(int m) : _m(m), _n(1) {};
         Frac() : _m(0), _n(1) {};

         int m() const { return _m;}
    unsigned n() const { return _n;}

    Frac operator  =(const Frac& a) {...}
    Frac operator /=(const Frac& a) {...}
    Frac operator +=(const Frac& a) {...}
    Frac operator -=(const Frac& a) {...}
}
```

Имеем функцию, которая делит m, n на $\text{нод}(m, n)$. Т.к. имеет смысл хранить дробь в сокращенной форме.

```
    void simplify(int& m, unsigned& n) {
        unsigned gcd = GCD((unsigned) std::abs(m), n);
        m /= (int)gcd;
        n /= gcd;
    }
```

Ну и раз уж был упомянут НОД, то здесь классический алгоритм Евклида:

$$(u, v) \rightarrow (v, u \, mod \, v) = (u_1, v_1) \rightarrow (v_1, u_1 \, mod \, v_1) \rightarrow \ldots \rightarrow (u_n, 0)$$

Тогда, u_n и есть наш НОД:

```
unsigned GCD(unsigned u, unsigned v) {
    while ( v != 0) {
        unsigned r = u % v;
        u = v;
        v = r;
    }
    return u;
}
```

Обертка для матрицы на cython:

Определение:

Подключаем написанный на C++ класс:

```
cdef extern from "Frac.cpp":
    cdef cppclass Frac:
        Frac()
        Frac(int)
        Frac(int, unsigned)
        int m()
        unsigned n()


cdef extern from "vMatrix.cpp":
    cdef cppclass Matrix[T]:
        Matrix()
        Matrix(size_t, size_t)
        T& operator ()(size_t, size_t)
        size_t rows()
        size_t cols()
        string str()
        void set(size_t, size_t, T)
        string tex_solution(vector[T])
```

Обертка:

Далее класс-обертка, который содержит в себе экземпляр нашего класса Matrix и который умеет приводить питовские типы к типам C++ и наоборот

```
# distutils: language = c++
from sympy import Rational
from IPython.display import display, Math, Latex

cdef class PyMatrix:
    cdef Matrix[Frac] *c_matrix

    def __cinit__(self, L):
        self.c_matrix = new Matrix[Frac](len(L), len(L[0]))

        for i in range(self.c_matrix[0].rows()):
            for j in range(self.c_matrix[0].cols()):
                r = Rational(L[i][j])
                self.c_matrix[0].set(i, j, Frac(r.p, r.q))

    def solve(self, b):
        if(len(b) != self.c_matrix.rows()):
            raise ValueError("Sizes don't match")
        cdef vector[Frac] vec
        for element in b:
            element = Rational(element)
            vec.push_back(Frac(element.p, element.q))
        display(Math(self.c_matrix.tex_solution(vec).decode('UTF-8')))

    def __dealloc__(self):
        del self.c_matrix

    def __str__(self):
        return self.c_matrix.str().decode('UTF-8')

    def _repr_latex_(self):
        return str(self)
```

Тесты

Подготовим среду

In [7]:
```
from Matrix import PyMatrix
from sympy import Rational as r
```

In [8]:
```
def rationalize(M):
    M_copy = M.copy()
    for i, row in enumerate(M):
        for j, el in enumerate(row):
            M_copy[i][j] = sympy.Rational(el)
    return M_copy

def Matrix(M):
    return PyMatrix(rationalize(M))
```

Нет решений

In [9]:
```
M = Matrix([
    [1, 2, 3],
    [3, 2, 1/2],
    [0, 0, 0]])
```

In [10]:
```
M
```

Out[10]:

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & \frac{1}{2} \\ 0 & 0 & 0 \end{bmatrix}$$

In [11]:
```
M.solve([0, 0, 1])
```

There is no soulution

Нетривиальное ядро

In [12]:
```
M = Matrix([
    [1, 2, 3],
    [3, 2, 1/2]])
```

In [13]:
```
M
```

Out[13]:

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & \frac{1}{2} \end{bmatrix}$$

In [14]:
```
M.solve([1,3])
```

$$X = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{-5}{4} \\ \frac{17}{8} \\ -1 \end{bmatrix} C_1$$

Тривиальное ядро (Одно решение)

In [15]:
```
M = Matrix([[1, 2, 3],
            [1, 1, 3],
            [1/2, 1, 2]])
```

In [16]:
```
M
```

Out[16]:

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 3 \\ \frac{1}{2} & 1 & 2 \end{bmatrix}$$

In [17]:
```
M.solve([r(-1, 2), 3, -1])
```

$$X = \begin{bmatrix} 11 \\ \frac{-7}{2} \\ \frac{-3}{2} \end{bmatrix}$$

Заключение

Была разработана программа, для вычисления решения ситем линейных алгебраических уравнений. На данном примере мы убедились в гибкости языка C++. Система шаблонов позволила мне написать независимую от типа матрицу, т.е. программа пригодна для систем над произвольным полем. В то же время, в C++, мы можем оперировать с достаточно низкоуровневыми типами, тем самым обеспечивая высокую производительность.

Так-же из классов и функций написанных на C++ можно создать модуль для подключения к программам, написанным на более высокуровневых языках, которые не способны работать с архитектурой так-же, как C++, но зато способны предоставить пользователю простой интерфейс и высокий уровень абстракции и зачастую интерпретируемы. Объеденяя плюсы обоих инструментов, получаем неплохую производительность и интерактивность.

Библиография

1. D. Joyce "*Kernel, image, nullity, and rank*", Math 130 Linear Algebra 2015, Clarck University
2. Ильин В.А., Ким Г.А. "*Линейная алгебра аналитическая геометрия*" 2014, МГУ
3. B. Stroustrup "*The C++ Programming Language* (4th ed.)"

Приложение

vMatrix.hpp

```
#pragma once

#define range(i, begin, end) for(size_t i = begin; i < (end); i++)

//#include <fstream>
#include <vector>
#include <algorithm>
#include <sstream>
#include <string>

//#include "Frac.hpp"
using namespace std;

template <typename Field>
class Matrix {
    vector< vector<Field>* > _M;
    Matrix& self = *this;

    void clear() {
        for(auto& row: _M) delete row;
    }

public:

    size_t rows() const {
        return _M.size();
    }

    size_t cols() const {
        if(rows() == 0) return 0;
        return _M[0]->size();
    }


    Matrix(size_t rows, size_t cols) {
        _M.resize(rows);
        for(auto& row: _M) row = new vector<Field>(cols);
    }

    Matrix& operator = (const Matrix& A) {
        clear();
        _M.resize(A.rows());
        range(i, 0, A.rows()) _M[i] = new vector<Field>(*(A._M[i]));
```

```
        return self;
    }

    Matrix& operator = (Matrix&& A) {
        clear();
        _M = A._M;
        A._M.resize(0);
        return self;
    }

    Matrix(const Matrix &A) {
        self = A;
    }

    Matrix(Matrix&& A) {
        _M = A._M;
        A._M.resize(0);
    }

    ~Matrix() {
        clear();
    }



    Field& operator() (size_t i, size_t j) const {
        return (*_M[i])[j];
    }

    void add(size_t to, size_t from, Field k) {
        range(j, 0, cols()) self(to, j) += k * self(from, j);
    }

    void swap(size_t first, size_t second) {
        std::swap(_M[first], _M[second]);
    }

    void mul(size_t row, Field k) {
        range(j, 0, cols()) self(row,j) = self(row, j) * k;
    }

    void div(size_t row, Field k) {
        range(j, 0, cols()) self(row, j) = self(row,j) / k;
    }

    pair< bool, vector<bool> > row_reduce(vector<Field>& b) {
        if(b.size() != rows()) {
            throw "length of b don't match with quantity of rows";
        }
```

```
        vector<bool> pivots(cols(), false);

        size_t pivot = 0;
        range(k, 0, cols()) {
            range(i, pivot, rows()) {
                if(self(i,k) != Field()) {
                    pivots[k] = true;

                    // swap
                    std::swap(b[pivot], b[i]);
                    swap(pivot, i);

                    // div
                    b[pivot] /= self(pivot, k);
                    div(pivot, self(pivot, k));

                    // nullify all elements in the row except the pivot
                    range(ii, 0, rows()) {
                        if(i != ii) {
                            b[ii] -= b[pivot]*self(ii, k);
                            add(ii, pivot, -self(ii,k));
                        }
                    }
                    pivot++;
                    break;
                }
            }
        }
        bool there_is_a_solution = true;
        range(i, pivot, rows()) {
            if(b[i] != Field()) {
                there_is_a_solution = false;
            }
        }

        return make_pair(there_is_a_solution, pivots);
    }

    friend istream& operator >> (istream& in, Matrix &M) {
        range(i, 0, M.rows())
            range(j, 0, M.cols())
                in >> M(i,j);
    }


    friend ostream& operator << (ostream& out, const Matrix &M) {
        out << "\\begin{bmatrix}\n";
        range(i, 0, M.rows()) {
            out << "    ";
```

```
        range(j, 0, M.cols()) {
            out << M(i, j) << ((j < M.cols()-1) ? " & " : "");
        }
        out << (i < M.rows() ? "\\\\\n" : "\n");
    }
    out << "\\end{bmatrix}";
    return out;
}

friend Matrix operator * (const Matrix& A, const Matrix& B) {
    if(A.cols() != B.rows()) {
        throw "Sizes don't match";
    }
    Matrix C(A.rows(), B.cols());
    range(i, 0, A.rows()) {
        range(j, 0, B.cols()) {
            range(k, 0, A.cols()) {
                C(i, j) += A(i, k) * B(k, j);
            }
        }
    }
    return C;
}

friend Matrix operator + (const Matrix& A, const Matrix& B) {
    if( (A.cols() != B.cols()) && (A.rows() != B.rows()) ) {
        throw "Sizes don't match";
    }
    Matrix C(A.rows(), B.cols());
    range(i, 0, A.rows()) {
        range(j, 0, A.cols()) {
            C(i, j) = A(i, j) + B(i, j);
        }
    }
    return C;
}

friend Matrix operator - (const Matrix& A) {
    Matrix B(A);
    range(i, 0, B.rows()) {
        range(j, 0, B.cols()) {
            B(i, j) = -B(i, j);
        }
    }
    return B;
}

friend Matrix operator - (const Matrix& A, const Matrix& B) {
    // return A + (-B);
```

```
        if( (A.cols() != B.cols()) || (A.rows() != B.rows()) ) {
            throw "Sizes don't match";
        }
        Matrix C(A.rows(), B.cols());
        range(i, 0, A.rows()) {
            range(j, 0, A.cols()) {
                C(i, j) = A(i, j) - B(i, j);
            }
        }
        return C;

    }

    friend bool operator == (const Matrix& A, const Matrix& B) {
        if( (A.cols() != B.cols()) || (A.rows() != B.rows()) ) {
            throw "Sizes don't match";
        }
        range(i, 0, A.rows()) {
            range(j, 0, A.cols()) {
                if(A(i, j) != B(i, j)) return false;
            }
        }
        return true;
    }

    friend bool operator != (const Matrix& A, const Matrix& B) {
        return !(A == B);
    }

    void set(size_t i, size_t j, const Field& val) {
        self(i, j) = val;
    }


    string tex_solution(vector<Field>& b) {
        Matrix A(self);

        stringstream out;

        bool is_there_a_solution;
        vector<bool> pivots;

        tie(is_there_a_solution, pivots) = A.row_reduce(b);

        if(!is_there_a_solution) {
            out << "There\\ is\\ no\\ soulution\n";
            return out.str();
        }
```

```
        out << "X = \n";
        out << "\\begin{bmatrix}\n";

        range(j, 0, A.cols()) {
            out << "    " << (j < b.size() ? b[j] : Field())
            << ((j != A.cols()-1) ? "\\\\\n" : "\n");
        }

        out << "\\end{bmatrix}\n";

        size_t numeration = 0;
        range(k, 0, pivots.size()) {
            if(!pivots[k]) {
                out << " + \n" << "\\begin{bmatrix}\n";
                range(i, 0, A.cols()) {
                    out << "    " << ((i == k) ? -Field(1): ( i < A.rows()
? A(i, k) : Field()))
                    << ((i != A.cols()-1) ? "\\\\\n" : "\n");
                }
                numeration++;
                out << "\\end{bmatrix}C_{" << numeration << "}\n" ;
            }
        }
        return out.str();
    }

    string str()
    {
        stringstream s;
        s << self;
        return s.str();
    }

};
```

Frac.hpp

```
#pragma once

#include <cstdlib>
#include <iostream>

unsigned GCD(unsigned u, unsigned v);


class Frac;
bool operator==(const Frac& a, const Frac& b);
```

```
bool operator!=(const Frac& a, const Frac& b);
Frac operator+(const Frac& a, const Frac& b);
Frac operator-(const Frac&a);
Frac operator-(const Frac& a, const Frac& b);
Frac operator*(const Frac& a, const Frac& b);
Frac operator/(const Frac& a, const Frac& b);


using namespace std;
ostream& operator << (ostream& out, const Frac& a);
istream& operator >> (istream& in, Frac& a);

class Frac {
private:
    int _m;
    unsigned int _n;

    void simplify(int& m, unsigned& n) {
        unsigned gcd = GCD((unsigned) std::abs(m), n);
        m /= (int)gcd;
        n /= gcd;
    }

public:
    Frac(int m, unsigned n): _m(m), _n(n) {
        if(n == 0) {
            throw "Divide by zero";
        }
        simplify(_m, _n);
    }

    Frac(int m) : _m(m), _n(1) {};

    Frac() : _m(0), _n(1) {};

    int m() const { return _m;};
    unsigned n() const { return _n;};

    Frac operator=(const Frac& a) {
        _m = a.m();
        _n = a.n();
        return *this;
    }

    Frac operator/=(const Frac& a) {
        if(a.m() == 0) {
            throw "Divide by zero";
        }
        if(a.m() < 0) {
```

```
            _m = -_m;
        }
        _m *= a.n();
        _n *= std::abs(a.m());
        simplify(_m, _n);
        return *this;
    }


    Frac operator +=(const Frac& a) {
        _m *= a.n();
        _m += a.m() * _n;
        _n *= a.n();
        simplify(_m, _n);
        return *this;
    }

    Frac operator -=(const Frac& a) {
        *this += (-a);
        return *this;
    }
};
```

Frac.cpp

```
#include "Frac.hpp"

unsigned GCD(unsigned u, unsigned v) {
    while ( v != 0) {
        unsigned r = u % v;
        u = v;
        v = r;
    }
    return u;
}

bool operator==(const Frac& a, const Frac& b) {
    return a.m() == b.m() && a.n() == b.n();
}

bool operator!=(const Frac& a, const Frac& b) {
    return !(a == b);
}

Frac operator+(const Frac& a, const Frac& b) {
    return Frac(a.m()*b.n() + b.m()*a.n(), a.n()*b.n());
}
```

```cpp
Frac operator-(const Frac&a) {
    return Frac(-a.m(), a.n());
}

Frac operator-(const Frac& a, const Frac& b) {
    return a + (-b);
}

Frac operator*(const Frac& a, const Frac& b) {
    if(a.m() == 0 || b.m() == 0) return Frac(0);
    else return Frac(a.m() * b.m(), a.n() * b.n());
}

Frac operator/(const Frac& a, const Frac& b) {
    if(a.m() == 0 && b.m() != 0) return Frac(0);
    if(b.m() < 0) return Frac(-a.m() * b.n(), -a.n() * b.m());
    return Frac(a.m()*b.n(), a.n() * b.m());
}

ostream& operator << (ostream& out, const Frac& a) {
    if (a.n() == 1) return out << a.m();
    return out << "\\frac{" << a.m() << "}{" << a.n() << "}";
}

istream& operator >> (istream& in, Frac& a) {
    int m; unsigned n;
    in >> m >> n;
    a = Frac(m, n);
    return in;
}
```

Matrix.pxd

```
from libcpp.string cimport string

cdef extern from "<vector>" namespace "std":
    cdef cppclass vector[T]:
        vector()
        void push_back(T&)


cdef extern from "Frac.cpp":
    cdef cppclass Frac:
        Frac()
        Frac(int)
        Frac(int, unsigned)
        int m()
        unsigned n()


cdef extern from "vMatrix.cpp":
    cdef cppclass Matrix[T]:
        Matrix()
        Matrix(size_t, size_t)
        T& operator ()(size_t, size_t)
        size_t rows()
        size_t cols()
        string str()
        void set(size_t, size_t, T)
        string tex_solution(vector[T])
```

Matrix.pyx

```
# distutils: language = c++
from sympy import Rational
from IPython.display import display, Math, Latex

cdef class PyMatrix:
    cdef Matrix[Frac] *c_matrix

    def __cinit__(self, L):
        self.c_matrix = new Matrix[Frac](len(L), len(L[0]))

        for i in range(self.c_matrix[0].rows()):
            for j in range(self.c_matrix[0].cols()):
                r = Rational(L[i][j])
                self.c_matrix[0].set(i, j, Frac(r.p, r.q))

    def list(self):
        L = []
        for i in range(self.c_matrix[0].rows()):
            L.append([])
            for j in range(self.c_matrix[0].cols()):
                L[i].append(Rational(
                    self.c_matrix[0](i, j).m(),
                    self.c_matrix[0](i, j).n()
                    ))
        return L

    def solve(self, b):
        if(len(b) != self.c_matrix.rows()):
            raise ValueError("Sizes don't match")
        cdef vector[Frac] vec
        for element in b:
            element = Rational(element)
            vec.push_back(Frac(element.p, element.q))
        display(Math(self.c_matrix.tex_solution(vec).decode('UTF-8')))

    def __dealloc__(self):
        del self.c_matrix

    def __str__(self):
        return self.c_matrix.str().decode('UTF-8')

    def _repr_latex_(self):
        return str(self)

    # def __repr__(self):
        # return str(self.c_matrix.print())
```

setup.py

```python
from distutils.core import setup, Extension
from Cython.Build import cythonize


setup(
    ext_modules = cythonize(Extension(
        name = "Matrix",
        sources = ["Matrix.pyx", "vMatrix.cpp"],
        extra_compile_args=["-std=c++11"],
        extra_link_args=["-std=c++11"])))
```