Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы численного анализа

**ОТЧЁТ**

к лабораторной работе

на тему

Решение систем линейных алгебраических уравнений (СЛАУ)

методом Гаусса и с помощью его модификаций

Выполнил: студент группы 053506

Слуцкий Никита Сергеевич

Проверил: Анисимов Владимир Яковлевич

**Оглавление**

[**Цели выполнения задания** **Error! Bookmark not defined.**](#_Toc65424824)

[**Краткие теоритические сведения** 3](#_Toc65424825)

[**Задание** 6](#_Toc65424826)

[**Программная реализация** 6](#_Toc65424827)

[**Полученные результаты**](#_Toc65424828) 10

[**Оценка** 1](#_Toc65424829)0

**[Выводы](#_Toc65424830)** [11](#_Toc65424830)

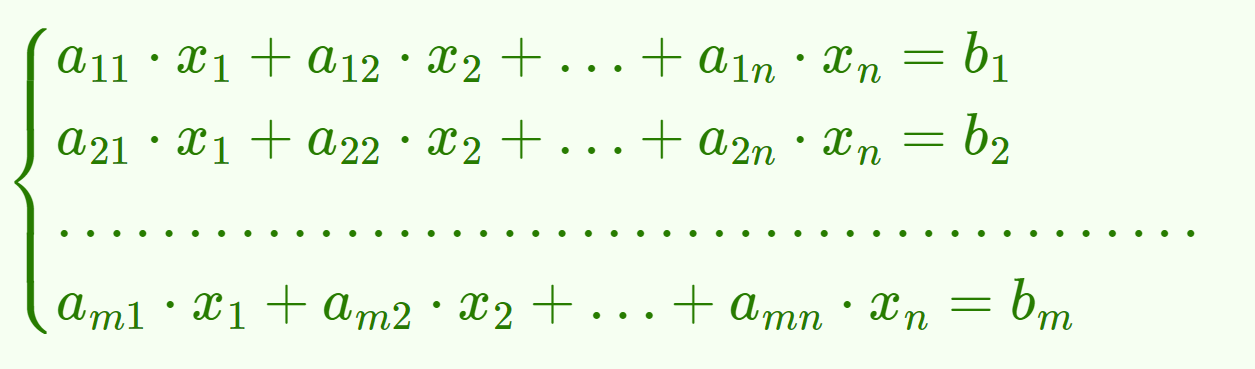
**Вариант 6 (Номер в журнале – 21)**

**Цели выполнения задания:**

* Изучить метод Гаусса для решения СЛАУ и его модификации
* Получить численное решение заданной СЛАУ
* Составить алгоритм решения СЛАУ указанными методами для организации вычислений на ЭВМ
* Составить программу решения по разработанному алгоритму
* Проверить правильность работы программы

**Краткие теоретические сведения:**

В общем виде систему линейных алгебраических уравнений можно записать в виде:



Имея отдельно матрицу коэффициентов A и матрицу свободных членов B это можно компактно записать в виде A \* x = B. В случае наличия решений система является совместной, иначе — несовместной.

Всегда можно пробовать систему методом Крамера и даже получить решение, если определитель матрицы не равен нулю. Но при больших размерностях вычисления определителей требуют большого количества вычислений (порядка n!). В этом случае удобнее пользоваться методом Гаусса.

Определю элементарные преобразования над системой (или полной матрицей системы).

* умножение всей строки (уравнения) на число, не равное нулю
* прибавление к одной строке (уравнению) другой строки (уравнения), умноженной на число
* обмен мест двух строк (уравнений) в системе

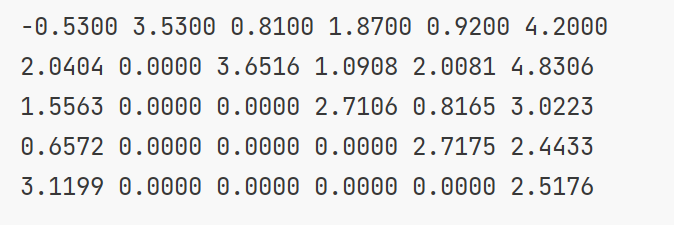
Метод Гаусса по-другому называется методом последовательного исключения. В общем случае он состоит из прямого и обратного ходов. Рассматриваемые здесь модификации метода привносят модификации в часть прямого хода.

Сначала рассмотрю классическую схему **единственного деления**. Пусть a11 ­не равняется нулю. Тогда можно исключить из всех последующих строк ([2..n]) переменную x1­. Просто вычту последовательно из 2-го, 3-го .. n-го уравнений системы 1-е, умноженное на a­i1 / a11. Получу обновлённую матрицу. Теперь коэффициенты в строках [2..n] обновлены, а при x1 в них теперь вообще стоят нули. Далее пусть a22 не равняется нулю. Аналогично исключу x2 из строк [3..n]. В итоге после n-1 таких шагов я должен получить треугольную матрицу с нулями под главной диагональю. Это был классический метод Гаусса.

Рассмотрю также схему частичного и полного выбора. В классическом методе при исключении переменных из уравнений уравнения сохранялись в исходном порядке.

**Метод частичного выбора**. Пусть сейчас k-я строка ( => k-й шаг и k-я переменная на очереди). Найду в k-ой колонке под рассматриваемой переменной (в строках i = [k..n]) максимальный элемент a­ik и поменяю местами k-ю строку и найденную i-ю. Зачем ? Во избежание сильного роста коэффициентов. Ведь при вычитании строк и исключении неизвестного мы умножаем строку на множитель aik / akk. А при малом << 0 akk множитель стремится к большим значениям. Поэтому нахожу максимальный элемент в колонке, меняю строки местами и сохраняю “адекватность” множителя.

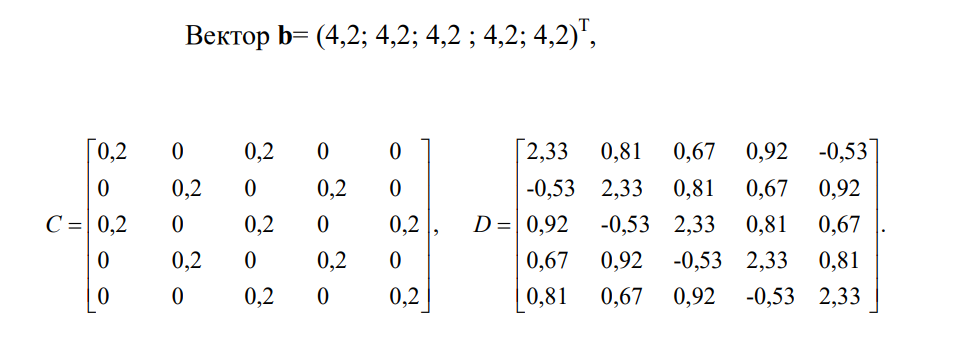
Метод полного выбора. Если в прошлой модификации я выбирал максимальный элемент в колонке, далее менял строки и делал всё как обычно, то здесь уже буду выбирать максимальный элемент во всей оставшейся (то есть ниже, чем текущая строка) матрице. Предположим, я сейчас на 1-м шаге. Начинаю искать максимальный элемент в матрице (именно матрице главных коэффициентов) в строках [1..n]. Положим, что я нашёл какое-то akl, k и l – номер строки и столбца. Первым делом я меняю 1-ю строку с k-ой, а далее исключаю из уравнения НЕ 1-ю переменную (как это делалось ранее), а l-ую. Таким образом после процесса под a1l будут нули. Перехожу на следующую строчку, начинаю искать максимальный элемент в части матрицы [2..n], найду его, получу номер строки и столбца, поменяю 2-ю и найденную строки местами и исключу переменную в найденном столбце в уравнениях [2+1 .. n]. Аналогично проделываю n-1 раз. На выходе получится не треугольная матрица (в своём явном виде), но получится матрица, ровно подходящая под дальнейшее вычисление переменных обратным ходом. Например, такая:



Обратный ход. После получения треугольной матрицы либо матрицы аналогичной по смыслу (именно для интересуемой нас цели – метода обратного хода) можно приступать к обратному ходу. Начинается он с последней строки матрицы. Там исключены все переменные, кроме какой-то одной. Таким образом, можно сразу её численно выразить. В предпоследней строке исключены все переменные, кроме той же, что была в последней, и ещё одной. Значение последней у меня уже есть. Таким образом вторая переменная также выражается без проблем. Обобщённо – в каждой новой строке (если идти снизу вверх) добавляется новая неизвестная, а остальные из этой строки были вычислены ещё на этапе прошлых строк. В этом и заключается суть метода обратного хода.

**Задание**

Методом Гаусса (3-мя описанными его разновидностями) найти с точностью 0.0001 численное решение системы A \* x = b, где A = k \* C + D. Матрицы C, D, b задаются ниже.



**Программная реализация:**

Ниже можно ознакомиться с программной реализацией главных функций по триангуляции, обратному ходу, а также вызов этого всего из главного файла. Вспомогательные функции и перегруженные операторы (реализации) опущены.

**std::vector<std::size\_t> Triangulate(std::vector<std::vector<double>> &full\_matrix,  
 const GaussSolvingType &solution\_type)  
{  
 const auto subtract\_rows{[&](const std::size\_t from, const std::size\_t which, const double ratio) -> void {  
 for (std::size\_t col = 0; col < full\_matrix.size() + 1; ++col)  
 full\_matrix[from][col] -= full\_matrix[which][col] \* ratio;  
 }};  
  
 const auto swap\_rows{[&](const std::size\_t first, const std::size\_t second) -> void {  
 for (std::size\_t col = 0; col < full\_matrix.size() + 1; ++col)  
 std::swap(full\_matrix[first][col], full\_matrix[second][col]);  
 }};  
  
 const auto find\_row\_with\_max\_main\_element{[&](const std::size\_t from) -> std::size\_t {  
 auto response{from};  
 for (std::size\_t row = from + 1; row < full\_matrix.size(); ++row)  
 if (std::abs(full\_matrix[row][from]) > std::abs(full\_matrix[response][from]))  
 response = row;  
 return response;  
 }};  
  
 auto find\_position\_with\_max\_matrix\_element{[&](const std::size\_t from) -> std::pair<std::size\_t, std::size\_t> {  
 auto maximum{full\_matrix[from][0]};  
 for (std::size\_t row = from + 1; row < full\_matrix.size(); ++row)  
 maximum = std::max(maximum, \*std::max\_element(  
 std::cbegin(full\_matrix[row]),  
 --std::cend(full\_matrix[row]),  
 [](const auto first, const auto second) -> bool { return std::abs(first) < std::abs(second); }  
 ));  
  
 for (std::size\_t row = from + 1; row < full\_matrix.size(); ++row)  
 for (std::size\_t col = 0; col < full\_matrix.size(); ++col)  
 if (full\_matrix.at(row).at(col) == maximum) return {row, col};  
  
 return {from, 0};  
 }};  
  
 std::vector<std::size\_t> variables\_excluding\_order{};  
  
 switch (solution\_type)  
 {  
 case GaussSolvingType::kSchemeOfTheOnlyDivision:  
 for (std::size\_t row = 0; row < full\_matrix.size(); ++row)  
 {  
 variables\_excluding\_order.push\_back(row);  
 if (IsNear(full\_matrix.at(row).at(row), 0.0))  
 throw std::runtime\_error("Error: Main diagonal element = 0");  
 for (std::size\_t lower\_row = row + 1; lower\_row < full\_matrix.size(); ++lower\_row)  
 {  
 const auto ratio{full\_matrix.at(lower\_row).at(row) / full\_matrix.at(row).at(row)};  
 subtract\_rows(lower\_row, row, ratio);  
 }  
 }  
 break;  
 case GaussSolvingType::kSchemeOfPartialSelection:  
 for (std::size\_t row = 0; row < full\_matrix.size(); ++row)  
 {  
 const auto row\_with\_max\_first\_item{find\_row\_with\_max\_main\_element(row)};  
 swap\_rows(row\_with\_max\_first\_item, row);  
 variables\_excluding\_order.push\_back(row);  
  
 if (IsNear(full\_matrix.at(row).at(row), 0.0))  
 throw std::runtime\_error("Error: Main diagonal element = 0");  
  
 for (std::size\_t lower\_row = row + 1; lower\_row < full\_matrix.size(); ++lower\_row)  
 {  
 const auto ratio{full\_matrix.at(lower\_row).at(row) / full\_matrix.at(row).at(row)};  
 subtract\_rows(lower\_row, row, ratio);  
 }  
 }  
 break;  
 case GaussSolvingType::kSchemeOfFullSelection:  
 for (std::size\_t row = 0; row < full\_matrix.size(); ++row)  
 {  
 const auto[row\_with\_max, col\_with\_max]{find\_position\_with\_max\_matrix\_element(row)};  
 variables\_excluding\_order.push\_back(col\_with\_max);  
  
 swap\_rows(row\_with\_max, row);  
  
 if (IsNear(full\_matrix.at(row).at(col\_with\_max), 0.0))  
 throw std::runtime\_error("Error: Main element = 0");  
  
 for (std::size\_t lower\_row = row + 1; lower\_row < full\_matrix.size(); ++lower\_row)  
 {  
 const auto ratio{full\_matrix[lower\_row][col\_with\_max] / full\_matrix[row][col\_with\_max]};  
 subtract\_rows(lower\_row, row, ratio);  
 }  
 }  
 break;  
 }  
  
 return variables\_excluding\_order;  
}**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**std::vector<double> GetSolutionByBackSubstitution(std::vector<std::vector<double>> &triangulated,  
 const std::vector<std::size\_t> &variables\_counting\_order)  
{  
 std::vector<double> response(triangulated.size());  
  
 std::size\_t var\_counter{0};  
  
 for (std::size\_t current\_row = triangulated.size() - 1;  
 current\_row >= 0; --current\_row) *// current row in system !! (from last)* {  
 if (var\_counter >= variables\_counting\_order.size()) break;  
 *// which var we can count on this row* const auto current\_variable\_number{variables\_counting\_order.at(var\_counter++)};  
 response.at(current\_variable\_number) = triangulated[current\_row].at(triangulated.size()) /  
 triangulated.at(current\_row).at(current\_variable\_number);  
  
 for (int rest\_row = 0; rest\_row < current\_row; ++rest\_row)  
 triangulated.at(rest\_row).at(triangulated.size()) -= triangulated.at(rest\_row).at(current\_variable\_number)  
 \* response.at(current\_variable\_number);  
 }  
  
 return response;  
}**

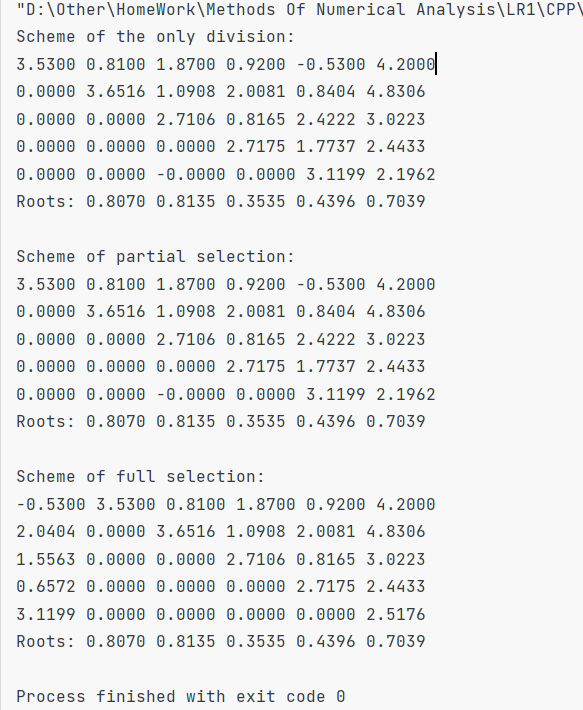
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**std::vector<double> SolveByGauss(const std::vector<std::vector<double>> &main\_coefficients,  
 const std::vector<double> &free\_coefficients,  
 const GaussSolvingType &solution\_type)  
{  
 auto to\_triangulate{GetFullSystemMatrix(main\_coefficients, free\_coefficients)};  
  
 auto excluding\_order{Triangulate(to\_triangulate, solution\_type)};  
 std::reverse(std::begin(excluding\_order), std::end(excluding\_order));  
  
 std::cout << to\_triangulate;  
  
 const auto last\_row\_variables\_count = std::accumulate(  
 std::cbegin(to\_triangulate.back()),  
 --std::cend(to\_triangulate.back()),  
 0,  
 [&](const auto response, const auto current) -> std::size\_t {  
 return IsNear(current, 0.0) ? response : response + 1;  
 });  
  
 if (last\_row\_variables\_count >= 2)  
 throw std::runtime\_error("System has infinite number of solutions");  
  
 auto response{GetSolutionByBackSubstitution(to\_triangulate, excluding\_order)};  
  
 return response;  
}**

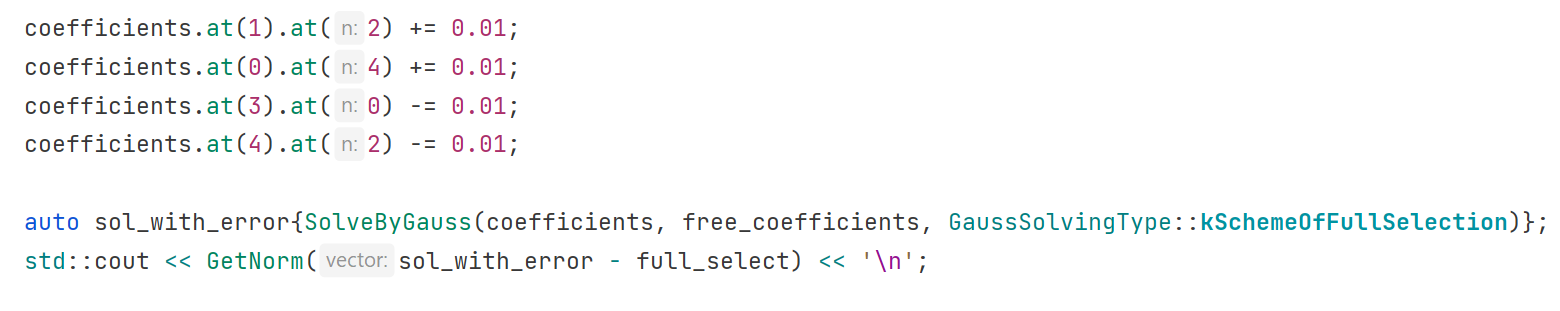
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**int main()  
{  
 const auto accuracy{GetNumberOfSignsAfterDot(kAccuracy)};  
 const auto coefficients{kMatrixC \* kOption + kMatrixD};  
 const auto &free\_coefficients{kVectorB};  
  
 try  
 {  
 std::cout << "Scheme of the only division: \n";  
 const auto only\_sol{SolveByGauss(coefficients, free\_coefficients, GaussSolvingType::kSchemeOfTheOnlyDivision)};  
 std::cout << std::setprecision(accuracy) << "Roots: " << only\_sol << '\n';  
  
 std::cout << "Scheme of partial selection: \n";  
 const auto part\_select{  
 SolveByGauss(coefficients, free\_coefficients, GaussSolvingType::kSchemeOfPartialSelection)};  
 std::cout << std::setprecision(accuracy) << "Roots: " << part\_select << '\n';  
  
 std::cout << "Scheme of full selection: \n";  
 const auto full\_select{SolveByGauss(coefficients, free\_coefficients, GaussSolvingType::kSchemeOfFullSelection)};  
 std::cout << std::setprecision(accuracy) << "Roots: " << full\_select;  
 }  
 catch (const std::exception &exception)  
 {  
 std::cout << exception.what();  
 }  
  
 return 0;  
}**

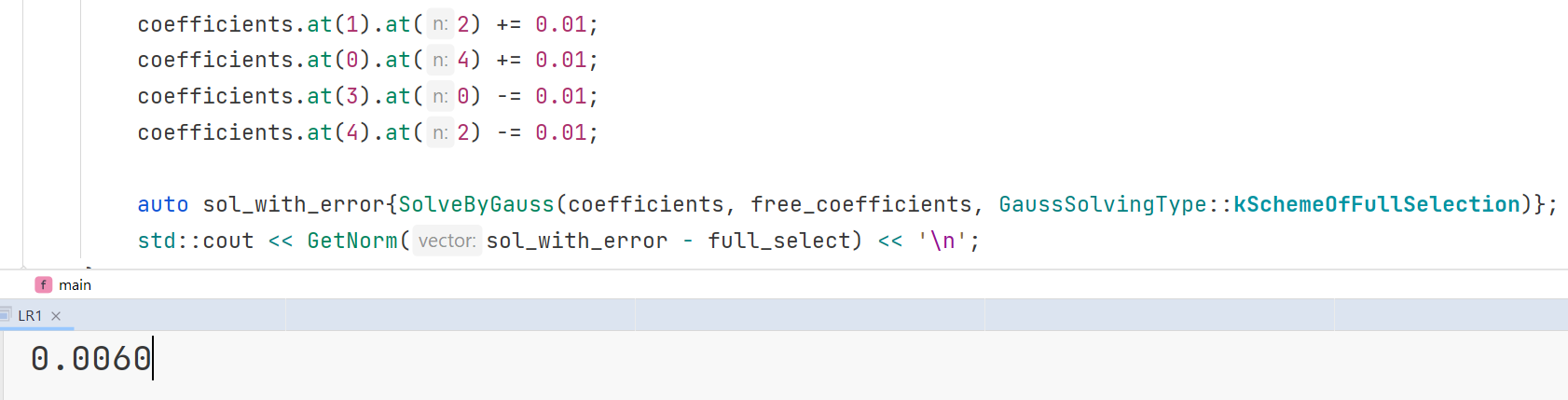
**Полученные результаты:**



**Оценка**



При изменении случайно выбранных коэффициентов матрицы на значение +- 0.01 и вычислении решений СЛАУ с новыми значениями получил новое решение. Норма разности векторов старого и нового решения отличается не более, чем на указанные 0.01.



# **Выводы**

Метод Гаусса – хороший прямой метод для решения СЛАУ. Тут не надо вычислять много определителей, находить обратную матрицу. А если учесть, что для нахождения определителя или обратной матрицы, собственно, я скорее всего и буду использовать прямой ход Гаусса (приведение к треугольной), то это тем более это хороший способ. Другие методы используют его как часть себя, а сам по себе для решения СЛАУ он самостоятелен и, следовательно, не является таким трудоёмким, как метод Крамера или метод Обратной матрицы (которые могут ссылаться к нему для своих подзадач).

В реализованной программе контролируется деление на ноль или на очень малый элемент — поэтому в случае чего будет сформировано исключение.