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

Численное решение систем линейных алгебраических уравнений

методом простых итераций и методом Зейделя

Выполнил: студент группы 053506

Слуцкий Никита Сергеевич

Проверил: Анисимов Владимир Яковлевич

Минск 2022

**Оглавление**

[**Цели выполнения задания** 3](#_Toc65424824)

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

[**Задание** 4](#_Toc65424826)

[**Программная реализация** 5](#_Toc65424827)

[**Полученные результаты**](#_Toc65424828) 7

[**Выводы** 7](#_Toc65424830)

**Вариант 6 (Номер в журнале – 21)**

**Цели выполнения задания:**

* Изучить итерационные методы решения СЛАУ (метод простых итераций и метод Зейделя)
* Составить алгоритм решения указанными методами, применимый для организации вычислений на ЭВМ
* Составить программу решения СЛАУ по разработанному алгоритму
* Численно решить тестовые примеры и проверить правильность работы программы. Сравнить трудоёмкость решения методом простых итераций и методом Зейделя.

**Краткие теоретические сведения:**

Метод итерации – это численный и приближённый метод решения СЛАУ. Суть заключается в составлении рекуррентного соотношения с заданием начального вектора значений и нахождении по приближённому текущему значению величины следующего приближения, которое является более точным.

Метод позволяет получить значения корней системы с заданной точностью в виде предела последовательности некоторых векторов.

Пусть имеется система A\*x = b. Чтобы применить итерационный метод, необходимо сначала привести систему к эквивалентному виду: x = B\*x + d. Это значит, что из i-ой строки нужно выразить в строке i-ю переменную. Получили описанную матрицу. Затем выберу начальное приближение x0, например, x­0 = (0.0, 0.0, 0.0, … , 0.0). На каждой итерации имею текущее приближение и следующее приближение.

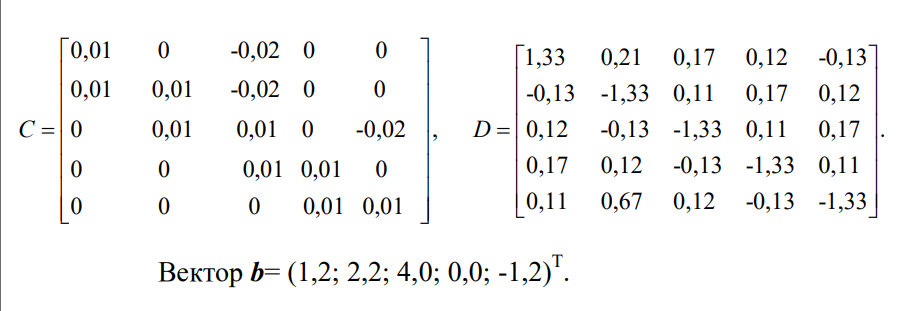
Что дальше ? На начальном этапе текущее приближение равно выбранному x0, а следующее – не определено (ну или также равно x0). Чтобы получить следующее приближение из имеющегося текущего, необходимо значения переменных из текущего приближения подставить в выраженное ранее B\*x + d. Таким образом получу новый вектор x1 с новым приближением. Далее процесс повторяется, то есть на место текущего уходит x1 с прошлой итерации и считается новое следующее приближение. Рекуррентно можно записать так: xi + 1 = B \* xi + d (где i – номер итерации, а не степень !). При этом при i = 0 (x0) x0 равняется выбранному начальному приближению (я выбрал (0, 0, 0, …, 0)).

Важно заметить, что для убеждения в сходимости этого итерационного процесса необходимо проверить достаточное условие сходимости. Оно заключается в том, что (в изначальной матрице, в матрице главных коэффициентов) в i-ой строке модуль коэффициента при i-й переменной был больше суммы модулей коэффициентов при остальных переменных в этой строке. И так для всех строк системы !

Теперь про метод Зейделя. Это “оптимизационная” разновидность метода простых итераций. На каждом этапе при вычислении x1[i] я буду подставлять значения не из предыдущего x0, а из самого же себя. То есть при использовании переменных с номерами 0..i-1 я уже буду обращаться к уже посчитанным НОВЫМ следующим значениям. Таким образом уменьшается количество итераций. Так в тестовом примере в данной лабораторной работе количество итераций снизилось с 9 до 6.

**Задание**

Методом простых итераций и Зейделя найти с точностью 0.0001 численное решение системы A\*x = b, где A = k \* C + D, исходные данные заданы ниже:



**Программная реализация:**

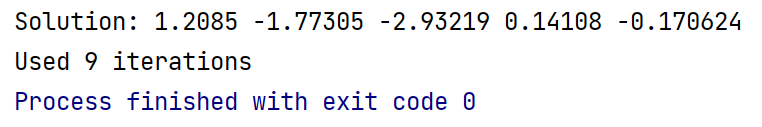
Ниже можно ознакомиться с программной реализацией главных функций по триангуляции, обратному ходу, а также вызов этого всего из главного файла. Вспомогательные функции и перегруженные операторы (реализации) опущены.

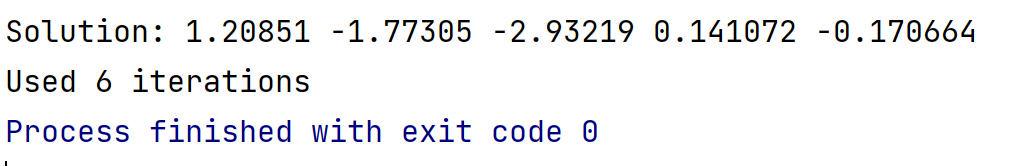
**bool CheckConvergence(const std::vector<std::vector<double>> &main\_coefficients)  
{  
 for (std::size\_t row = 0; row < main\_coefficients.size(); ++row)  
 {  
 const auto main\_absolute{std::abs(main\_coefficients.at(row).at(row))};  
 std::size\_t counter{0};  
 const auto rest\_sum\_absolute{std::accumulate(  
 std::begin(main\_coefficients.at(row)),  
 std::end(main\_coefficients.at(row)),  
 0.00,  
 [&](const double response, const double current) -> double {  
 return counter++ != row ? response + std::abs(current) : response;  
 })};  
  
 if (main\_absolute < rest\_sum\_absolute)  
 return false;  
 }  
  
 return true;  
}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
std::vector<std::vector<double>> ExpressMainVariables(const std::vector<std::vector<double>> &main\_coefficients,  
 const std::vector<double> &free\_coefficients)  
{  
 std::vector<std::vector<double>> response{};  
 response.resize(main\_coefficients.size());  
  
 for (std::size\_t row = 0; row < main\_coefficients.size(); ++row)  
 {  
 const auto full\_width{main\_coefficients.at(row).size() + 1};  
 const auto current\_variable\_ratio{main\_coefficients.at(row).at(row)};  
 response.at(row).resize(full\_width);  
  
 response.at(row).at(0) = free\_coefficients.at(row) / current\_variable\_ratio;  
  
 for (std::size\_t col = 1; col < full\_width; ++col)  
 response.at(row).at(col) =  
 col - 1 != row  
 ? -1 \* main\_coefficients.at(row).at(col - 1) / current\_variable\_ratio  
 : 0;  
 }  
  
 return response;  
}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
double GetError(const std::vector<double> &current\_variables\_set,  
 const std::vector<double> &previous\_variables\_set)  
{  
 double error{0.00};  
 for (std::size\_t counter = 0; counter < current\_variables\_set.size(); ++counter)  
 error = std::max(error, std::abs(current\_variables\_set.at(counter) - previous\_variables\_set.at(counter)));  
 return error;  
}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
std::pair<std::vector<double>, std::size\_t>  
SolveBySimpleIterations(const std::vector<std::vector<double>> &main\_coefficients,  
 const std::vector<double> &free\_coefficients,  
 const double epsilon,  
 const std::vector<double> &initial\_values,  
 const SolvingType &type)  
{  
 if (!CheckConvergence(main\_coefficients))  
 throw std::invalid\_argument("System of linear algebraic equations does not converge");  
  
 const auto check\_error([](const std::vector<double> &current\_variables\_set,  
 const std::vector<double> &previous\_variables\_set,  
 const double epsilon) -> bool {  
 return GetError(current\_variables\_set, previous\_variables\_set) <= std::abs(epsilon);  
 });  
  
 const auto copy\_vectors([](const std::vector<double> &from, std::vector<double> &to) -> void {  
 for (std::size\_t counter = 0; counter < from.size(); ++counter)  
 to.at(counter) = from.at(counter);  
 });  
  
 const auto count\_variable\_value([](const std::vector<double> &variable\_expression,  
 const std::vector<double> &variables\_set) -> double {  
 double response{variable\_expression.at(0)};  
 for (std::size\_t counter = 1; counter < variable\_expression.size(); ++counter)  
 response += variable\_expression.at(counter) \* variables\_set.at(counter - 1);  
 return response;  
 });  
  
 auto previous\_iteration{initial\_values}; *// предыдущая i-я итерация* auto current\_iteration{initial\_values}; *// текущая i+1-я итерация* const auto variables\_expressions{  
 ExpressMainVariables(main\_coefficients, free\_coefficients)}; *// выраженные переменные с главной диагонали* std::size\_t iterations\_count{0};  
 do  
 {  
 ++iterations\_count;  
 copy\_vectors(current\_iteration, previous\_iteration);  
 for (std::size\_t counter = 0; const auto &variable\_expression : variables\_expressions)  
 {  
 current\_iteration.at(counter) = count\_variable\_value(  
 variable\_expression,  
 type == SolvingType::*kSimpleIterations* ? previous\_iteration : current\_iteration  
 );  
 ++counter;  
 }  
 } while (!check\_error(current\_iteration, previous\_iteration, epsilon));  
  
 return {current\_iteration, iterations\_count};  
}**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**int main()  
{  
 const auto main\_coefficients{kMatrixC \* kOption + kMatrixD};  
 const auto &free\_coefficients{kVectorB};  
 const auto &initial\_values{kInitialValues};  
 const auto &epsilon{kAccuracy};  
  
 const auto[solution, number\_of\_iterations]{SolveBySimpleIterations(  
 main\_coefficients,  
 free\_coefficients,  
 epsilon,  
 initial\_values,  
 SolvingType::*kSeidel* )};  
  
 std::cout << "Solution: " << std::setprecision(6) << solution;  
 std::cout << "Used " << number\_of\_iterations << " iterations";  
  
 return 0;  
}**

**Полученные результаты:**





# **Выводы**

Метод простой итерации или метод Зейделя применяется для нахождения корней системы линейных уравнений с заданной точностью. В физике может быть полезно и, главное, быстро.