Санкт-Петербургский Государственный Политехнический Университет

Институт прикладной математики и механики

Кафедра «Прикладная математика»

**Курсовой проект по дисциплине**

**«Численные методы»**

Студент: Варисов Арсен Магомедгаджиевич

Группа: 23631/2

Преподаватель: Павлова Людмила Владимировна

Оглавление

[Часть 1. Решение алгебраических и трансцендентных уравнений: метод секущих и метод дихотомии 3](#_Toc533637061)

[Задача 4](#_Toc533637062)

[Описание методов и их применимость 4](#_Toc533637063)

[Предварительный анализ задачи 5](#_Toc533637064)

[Проверка условий применимости 5](#_Toc533637065)

[Тестовый пример с детальными расчётами (для метода секущих) 5](#_Toc533637066)

[Перечень контрольных тестов 6](#_Toc533637067)

[Модульная структура программы 6](#_Toc533637068)

[Численный анализ решения и результаты 7](#_Toc533637069)

[Выводы 8](#_Toc533637070)

[Приложение 9](#_Toc533637071)

[Часть 2. Решение СЛАУ прямыми методами: метод квадратного корня 10](#_Toc533637072)

[Задача 11](#_Toc533637073)

[Алгоритм и условия применимости метода 11](#_Toc533637074)

[Предварительный анализ задачи 11](#_Toc533637075)

[Проверка условий применимости 11](#_Toc533637076)

[Тестовый пример с детальными расчётами 11](#_Toc533637077)

[Перечень контрольных тестов 12](#_Toc533637078)

[Модульная структура программы 12](#_Toc533637079)

[Численный анализ решения и результаты 12](#_Toc533637080)

[Выводы 13](#_Toc533637081)

[Приложение 14](#_Toc533637082)

[Часть 3. Решение СЛАУ итерационными методами: метод Зейделя 16](#_Toc533637083)

[Задача 17](#_Toc533637084)

[Алгоритм и условия применимости метода 17](#_Toc533637085)

[Предварительный анализ задачи 17](#_Toc533637086)

[Тестовый пример с детальными расчётами 17](#_Toc533637087)

[Перечень контрольных тестов 18](#_Toc533637088)

[Модульная структура программы 18](#_Toc533637089)

[Численный анализ решения и результаты 19](#_Toc533637090)

[Выводы 19](#_Toc533637091)

[Приложение 20](#_Toc533637092)

[Часть 4. Решение алгебраической проблемы собственных значений: Метод Якоби 23](#_Toc533637093)

[Задача 24](#_Toc533637094)

[Алгоритм и условия применимости метода 24](#_Toc533637095)

[Предварительный анализ задачи 24](#_Toc533637096)

[Тестовый пример с детальными расчётами 25](#_Toc533637097)

[Перечень контрольных тестов 25](#_Toc533637098)

[Модульная структура программы 25](#_Toc533637099)

[Численный анализ решения и результаты 25](#_Toc533637100)

[Выводы 27](#_Toc533637101)

[Приложение 27](#_Toc533637102)

[Выводы 30](#_Toc533637103)

# Часть 1. Решение алгебраических и трансцендентных уравнений: метод секущих и метод дихотомии

### Задача

Даны два уравнения: (алгебраическое) и  (трансцендентное). Необходимо найти корни уравнений с заданной допустимой ошибкой .

Для решения необходимо использовать два разных итерационных метода: метод дихотомии и метод секущих.

### Описание методов и их применимость

#### 1. Метод дихотомии

Пусть дано уравнение  и Тогда, и можно производить многократное разбиение отрезка на две половины с последующим выбором той, в которой содержится корень, до тех пор, пока разность между концами отрезка не станет приемлемо малой.

Алгоритм:

1. 
2. Если то 
3. Если  то иначе 
4. Если то  найден, иначе вернуться к шагу 1.

#### 2. Метод секущих

Пусть  , или сохраняет знак на . Тогда с удовлетворяющей условию  стороны отрезка можно взять две точки , лежащие на графике функции, и провести через них секущую, продолжив её до пересечения с осью абсцисс и, проведя от неё перпендикуляр к графику функции, получить новую точку . Процесс повторяется уже с парами точек ; и т.д. до тех пор, пока точка не будет достаточно близка корню.

Алгоритм:

Пусть



1. Если , то , иначе .
2. .
3. 
4. Если , то ответ найден, иначе увеличить  на 1 и перейти к п. 3.

### Предварительный анализ задачи

По теореме о верхней границе подберем промежуток поиска корней алгебраического уравнения.

Для положительных корней:

, , . Следовательно:



Для отрицательных:

, , . Следовательно:



, ,

следовательно, существует как минимум один корень для каждого уравнения соответственно.

Для трансцендентной функции промежутки корней найдём графически.

При последующем анализе было выявлено, что имеет 2 корня,  - 4 корня.

Исходя из графиков функций найдём промежутки для всех корней:

: :

|  |  |
| --- | --- |
| a | b |
| -8 | -6 |
| 0 | -2.5 |

|  |  |
| --- | --- |
| a | b |
| -1.1 | -0.9 |
| -0.8 | -0.7 |
| -0.001 | 0.1 |
| 0.5 | 0.6 |

### Проверка условий применимости

Т.к. метод дихотомии не имеет других требований кроме наличия корня, то он применим к каждому промежутку.

Проверим условия применимости метода секущих для каждого промежутка:

1. 

2. a) e) 



b)  f) 

c) 

d) 

3.  знакопостоянна на каждом промежутке.

Следовательно, метод секущих применим для поиска всех корней обоих уравнений.

### Тестовый пример с детальными расчётами (для метода секущих)

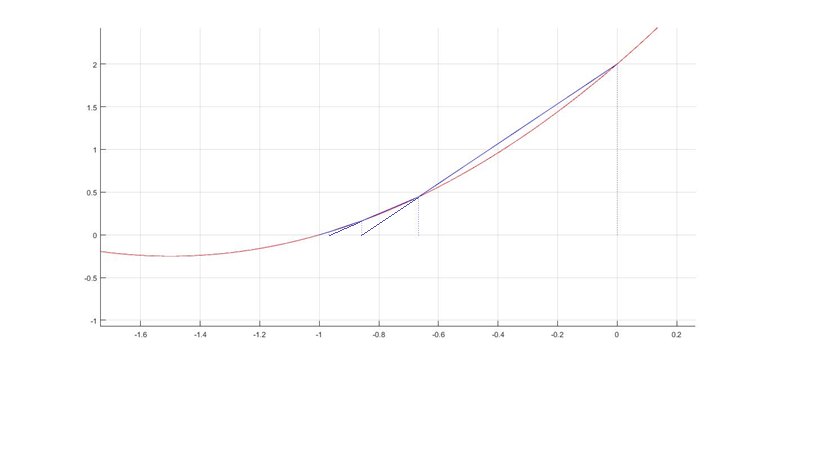
Проведем поиск корня уравнения на промежутке . знакопостоянна на промежутке, следовательно, есть единственный корень. , а значит, метод секущих применим. Возьмём допустимую ошибку .













Графическая интерпретация метода

### Перечень контрольных тестов

Для демонстрации возможностей методов были найдены все корни каждого уравнения: 2 для алгебраического, 4 – для трансцендентного.

Поиск каждого корня проводился неоднократно методами секущих и дихотомии с несколькими заданными значениями допустимой ошибки:

, , .

### Модульная структура программы

FunAlg (x) – алгебраическая функция, принимает некое число x и возвращает значение функции в данной точке.

FunAlgDer1 (x) – первая производная алгебраической функции, принимает некое число x и возвращает значение первой производной в данной точке.

FunAlgDer2 (x) – вторая производная алгебраической функции, принимает некое число x и возвращает значение второй производной в данной точке.

FunTran (x) – трансцендентная функция, принимает некое число x и возвращает значение функции в данной точке.

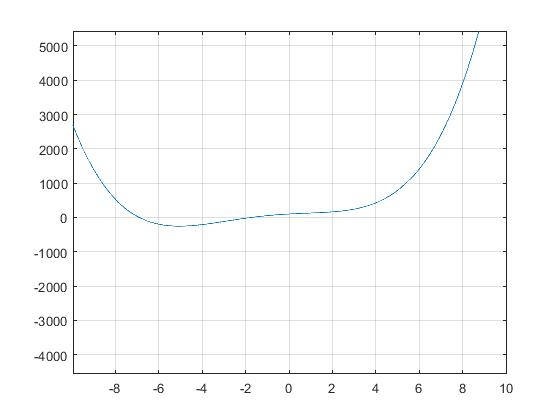
FunTranDer1 (x) – первая производная трансцендентной функции, принимает некое число x и возвращает значение первой производной в данной точке.

FunTranDer2 (x) – вторая производная трансцендентной функции, принимает некое число x и возвращает значение второй производной в данной точке.

Secant (Fun, FunDer1, FunDer2, Xa, Xb, Err) – модуль, осуществляющий поиск корня методом секущих на заданном промежутке с заданной точностью. Принимает функцию, её первую и вторые производные, концы промежутка и допустимую ошибку. Возвращает значение корня с заданной точностью, значение функции в найденной точке и число итераций.

Dichotomy (Fun, Xa, Xb, Err) – модуль, осуществляющий поиск корня методом дихотомии на заданном промежутке с заданной точностью. Принимает функцию, концы промежутка и допустимую ошибку. Возвращает значение корня с заданной точностью, значение функции в найденной точке и число итераций.

### Численный анализ решения и результаты

Графически определим количество корней и выберем подходящие промежутки.

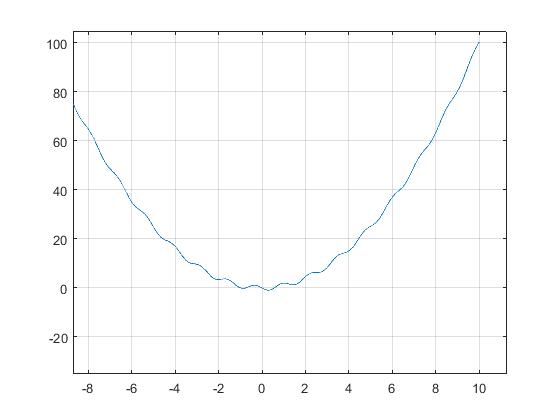
График алгебраической функции

График трансцендентной функции

Далее для каждого метода проведем многократные поиски корней с несколькими значениями допустимой погрешности. Запишем соответствующие значения корней, значения функций в найденных точках и количество прошедших в каждом случае итераций.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | -1.7347732042 | 0.0012 | 5 | -1.7402343750 | -0.4463 | 9 |
| 0.001 | -1.7347732042 | 0.0012 | 5 | -1.7351074218 | -0.0262 | 12 |
| 10^-8 | -1.7347875329 | -8.6686e-13 | 7 | -1.7347875293 | 2.9677e-07 | 29 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | -0.9880275933 | 0.0020 | 3 | -0.9812500000 | -0.0184 | 5 |
| 0.001 | -0.9873938459 | 5.8937e-05 | 4 | -0.9867187500 | -0.0020 | 8 |
| 10^-8 | -0.9873747603 | 1.3349e-11 | 6 | -0.9873747646 | 1.3414e-08 | 25 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | -0.7461061621052 | 0.0012 | 2 | -0.743750000000 | 0.0075 | 4 |
| 0.001 | -0.7461061621052 | 0.0012 | 4 | -0.746093750000 | 0.0012 | 7 |
| 10^-8 | -0.7465554721437 | 1.9394e-08 | 5 | -0.746555477380 | 1.3414e-08 | 24 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | 0.56367532270324 | 1.1216e-04 | 2 | 0.568750000000000 | 0.0300 | 4 |
| 0.001 | 0.56367532270324 | 1.1216e-04 | 2 | 0.563281250000000 | -0.0022 | 7 |
| 10^-8 | 0.56365620971683 | 1.1486e-12 | 4 | 0.563656204938889 | -2.8037e-08 | 24 |

Ввиду симметричности промежутка в данном случае метод дихотомии нашёл корень сразу:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | 7.18547780530e-04 | -0.0036 | 2 | 0 | 0 | 1 |
| 0.001 | 1.32427665848e-06 | -6.6214e-06 | 3 | 0 | 0 | 1 |
| 10^-8 | -0.7465554721437 | 1.9394e-08 | 4 | 0 | 0 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ε | Метод секущих | | | Метод дихотомии | | |
|  | x | F(x) | k | x | F(x) | k |
| 0.01 | -6.8973963753 | 7.0227e-08 | 6 | -6.8984375000 | 0.3343 | 8 |
| 0.001 | -6.8973963753 | 7.0227e-08 | 6 | -6.8974609375 | 0.0207 | 11 |
| 10^-8 | -6.8973963750 | 4.2633e-14 | 7 | -6.8973963782 | 1.0056e-06 | 28 |

### Выводы

Из полученных результатов можно заключить, что метод секущих в среднем имеет значительно меньшее число итераций, в целом превосходя метод дихотомии по всем параметрам. Однако метод дихотомии выделяется простотой исполнения и рекомендуется к использованию в случае необходимости быстрого решения несложной задачи.

### Приложение

#### Программный код

function Xs = Secant(Fun, FunDer1, FunDer2, Xa, Xb, Err)

i=0;

dif=inf;

x=Xa-1:0.01:Xb+1;

y=Fun(x);

hold on

grid on

plot (x, y, 'r')

XDer1=Xa:0.01:Xb;

YDer1=FunDer1(XDer1);

XDer2=Xa:0.01:Xb;

YDer2=FunDer2(XDer2);

m1=min(abs(YDer1));

M2=max(abs(YDer2));

if Fun(Xa)\*FunDer2(Xa)>0

X0=Xa;

X1=Xa+abs(Xb-Xa)\*0.000000000001;

else

X0=Xb;

X1=Xb-abs(Xb-Xa)\*0.000000000001;

end

plot ([X0, X0], [0,feval(Fun,X0)], 'b:')

plot ([X1, X1], [0,feval(Fun,X1)], 'b:')

while dif > Err

FunX1 = feval(Fun, X1);

Xi = X1 - FunX1\*(X0-X1)/(feval(Fun,X0)-FunX1);

plot ([Xi, Xi], [0,feval(Fun,Xi)], 'b:')

Y0=feval(Fun,X0);

Y1=feval(Fun,X1);

plot ([X0, X1], [Y0, Y1], 'b')

X0=X1;

dif=abs((Xi-X1)^2\*(M2)/2\*m1);

X1=Xi;

i=i+1;

end

hold off

Xs=X1;

i = i

Yk = feval(Fun,Xs)

function Xr = Dichotomy(Fun, Xa, Xb, Err)

i=0;

dif=abs(Xb-Xa);

x=Xa-0.5:0.01:Xb+0.5;

Xc=0;

y=Fun(x);

hold on

grid on

plot (x,y, 'r');

while (dif>Err)

Xc=(Xa+Xb)/2;

if Fun(Xc)==0

Xr=Xc;

Yk=feval(Fun,Xa);

return

end

FunXc=feval(Fun, Xc);

plot ([Xc,Xc],[0,feval(Fun,Xc)], 'b:')

if(FunXc\*feval(Fun,Xa))<0

Xb=Xc;

else

Xa=Xc;

end

i=i+1;

dif=dif/2;

end

hold off

Xr=Xc;

i=i

Yk=Fun(Xr)

# Часть 2. Решение СЛАУ прямыми методами: метод квадратного корня

### Задача

Дана система линейных алгебраических уравнений, представленная симметричной положительно определенной матрицей системы и столбцом свободных членов . Используя метод разложения Холецкого (метод квадратного корня), найти решение системы, то есть столбец , такой, что: .

Исследовать решения для матриц систем с разными числами обусловленности.

### Алгоритм и условия применимости метода

Рассмотрим матрицу . Пусть . Тогда матрицу можно представить в виде , где – нижнетреугольная матрица со строго положительными коэффициентами. Элементы вычисляются по формулам:

,

Воспользовавшись разложением, перепишем систему:

*.* Пусть , тогда сперва решим простую систему по формуле

а затем систему по формуле

### Предварительный анализ задачи

Так как – симметричная и положительно определенная матрица, то она является невырожденной, а значит, система имеет единственное решение.

Оценка погрешности будет производиться по формуле , где – столбец корней, полученных методом квадратного корня.

### Проверка условий применимости

Для исследования метода использовались случайно сгенерированные симметричные положительно определенные матрицы, следовательно, метод применим в каждом из рассмотренных случаев.

### Тестовый пример с детальными расчётами

Пусть , . Найдём матрицу :

, , отсюда

### Перечень контрольных тестов

Для исследования работы метода было решено 35 СЛАУ размерностью : 7 наборов по пять матриц с числами обусловленности от 1 до 50, от 50 до , от до , от до , от до , от до и от до соответственно. Для каждой системы были произведены возмущения матриц и столбцов , вычислены коэффициенты кторов невязки (погрешности).

### Модульная структура программы

double\* **Cholesky** (double arr[], double b[], int n) – функция разложения системы по методу квадратного корня и вычисления решения СЛАУ. Принимает массив arr[n\*n] (матрица системы , массив b[n] (столбец свободных членов и размерность системы n. Возвращает столбец решений x.

double **ErrorEval** (int n, double \*A, double \*x, double \*b) – функция нахождения нормы вектора невязки (погрешности). Принимает размерность системы, матрицу А, столбец решений x и столбец свободных членов b. Возвращает значение нормы вектора невязки.

double\* **MatrixMultiply** (double\* A, double\* B, int n) – функция вычисления произведения двух квадратных матриц. Принимает матрицу A, матрицу B и размерность n, возвращает произведение матриц – квадратную матрицу С = AB.

### Численный анализ решения и результаты

В результате тестирования были получены следующие данные:

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 5 | 0 | 1.174494 | 1.731329 |
| 10 | 0 | 0.642079 | 3.074055 |
| 20 | 0 | 1.549722 | 10.249204 |
| 30 | 0 | 3.240801 | 5.295227 |
| 40 | 0 | 1.951597 | 10.399062 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 50 | 0 | 1.259466 | 36.990184 |
| 60 | 0 | 1.289450 | 38.347038 |
| 70 | 0 | 1.541486 | 50.839042 |
| 80 | 0 | 0.578454 | 30.821030 |
| 90 | 0 | 2.222635 | 53.544731 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 100 | 0 | 0.966594 | 48.952640 |
| 250 | 0 | 0.139375 | 55.447697 |
| 400 | 10^-7 | 0.653421 | 79.847620 |
| 780 | 10^-7 | 0.342376 | 30.821030 |
| 990 | 2\*10^-7 | 0.115163 | 201.231607 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 2000 | 0 | 0.066197 | 95.739842 |
| 4000 | 2\*10^-7 | 1.904994 | 124.737049 |
| 6000 | 10^-6 | 0.167026 | 105.473021 |
| 8000 | 1.5\*10^-6 | 1.793372 | 103.160991 |
| 10000 | 5\*10^-7 | 5.041853 | 101.346612 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 12000 | 1.3\*10^-6 | 1.755981 | 171.573562 |
| 20000 | 8\*10^-7 | 1.417224 | 102.479278 |
| 40000 | 3.7\*10^-6 | 1.287564 | 101.265553 |
| 60000 | 1.08\*10^-5 | 2.003891 | 100.456560 |
| 80000 | 1.43\*10^-5 | 0.292699 | 99.416988 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 100000 | 5.8\*10^-5 | 3.615279 | 102.936298 |
| 200000 | 5.24\*10^-5 | 0.895358 | 100.119715 |
| 400000 | 9.07\*10^-5 | 0.735894 | 101.901324 |
| 600000 | 3.9\*10^-5 | 0.855377 | 101.788287 |
| 800000 | 4.02\*10^-5 | 0.729486 | 102.417076 |

|  |  |  |  |
| --- | --- | --- | --- |
| Cond(A) | ε | k1 | k2 |
| 1000000 | 9.04\*10^-5 | 2.136280 | 103.414101 |
| 2000000 | 5.57\*10^-5 | 1.477084 | 101.572444 |
| 4000000 | 5.99\*10^-5 | 0.532050 | 101.790871 |
| 6000000 | 133\*10^-5 | 10.821453 | 203.077989 |
| 8000000 | N/A | N/A | N/A |

При возмущенная матрица теряет положительную определенность, ввиду чего дальнейшие данные получены не были.

### Выводы

Из полученных данных можно заключить о корреляции между числом обусловленности матрицы и различием в найденных решениях при незначительных изменениях входных данных: в целом заметна тенденция роста значения коэффициента при повышении числа обусловленности. Коэффициент в каждом случае оказался значительно меньше и проявлял менее устойчивый рост при увеличении числа обусловленности.

Исследование метода Холецкого показало его высокую точность и скорость работы, однако условия сходимости метода значительно ограничивают область его применения.

Тем не менее, его использование выгодно в случаях работы с симметричными положительно определенными матрицами (которые часто встречаются в некоторых разделах физики и механики).

### Приложение

#### Программный код

double\* **VectorDiff** (double \*A, double \*B, double \*Dif, int n)

{

int i;

for (i = 0; i<n; i++)

Dif[i] = A[i]-B[i];

return Dif;

}

double \***MatrixDiff** (double \*A, double \*B, double \*Dif, int n)

{

int i;

for (i = 0; i<n\*n; i++)

Dif[i] = A[i]-B[i];

return Dif;

}

double **Choleskysum** (double arr[], int n, int i, int j)

{

int k;

double sum = 0;

for (k = 0; k<j; k++)

sum+=arr[i\*n+k]\*arr[j\*n+k];

return sum;

}

double\* **Cholesky** (double arr[], double b[], int n)

{

int i, j;

double sum, temp;

double\* y = NULL;

double\* x = NULL;

double\* l = NULL;

double \*lt = NULL;

double\* d = NULL;

d = calloc ((unsigned)(n\*n), sizeof(double));

l = calloc ((unsigned)(n\*n), sizeof(double));

lt = calloc ((unsigned)(n\*n), sizeof(double));

y = malloc ((unsigned)n\*sizeof(double));

x = malloc ((unsigned)n\*sizeof (double));

if (arr[0]>=0)

{

l[0] = sqrt (arr[0]);

d[0] = 1;

}

else

{

l[0] = sqrt (-arr[0]);

d[0] = -1;

}

lt[0] = l[0];

for (j = 1; j<n; j++)

{

l[j\*n] = arr[j\*n]/l[0];

lt[j] = l[j\*n];

}

for (i = 1; i<n; i++)

{

temp = arr[i\*n+i]-Choleskysum(l, n, i, i);

if (temp>=0)

{

l[i\*n+i] = sqrt(temp);

d[i\*n+i] = 1;

}

else

{

l[i\*n+i] = sqrt(-temp);

d[i\*n+i] = -1;

}

lt[i\*n+i] = l[i\*n+i];

for (j = i+1; j<n; j++)

{

l[j\*n+i] = (arr[j\*n+i]-Choleskysum(l, n, j, i))/l[i\*n+i];

lt[i\*n+j] = l[j\*n+i];

}

}

l = MatrixMultiply(l, d, n);

y[0] = b[0]/l[0];

for (i = 1; i<n; i++)

{

sum = 0;

for (j = 0; j<i; j++)

sum+=l[i\*n+j]\*y[j];

y[i] = (b[i]-sum)/l[i\*n+i];

}

x[n-1] = y[n-1]/lt[n\*n-1];

for (j = n-2; j>=0; j--)

{

sum = 0;

for (i = n-1; i>j; i--)

sum+=x[i]\*lt[j\*n+i];

x[j] = (y[j]-sum)/lt[j\*n+j];

}

free (y);

free (l);

free(lt);

free(d);

return x;

}

double **ErrorEval** (int n, double \*A, double \*x, double \*b)

{

int i, j;

double sum = 0;

double \*help = NULL;

help = calloc (n, sizeof(double));

for (i = 0; i<n; i++)

{

for (j = 0; j<n; j++)

help[i]+=A[i\*n+j]\*x[j];

help[i]-=b[i];

}

for (i = 0; i<n; i++)

sum+=help[i]\*help[i];

free (help);

return sqrt(sum);

}

# Часть 3. Решение СЛАУ итерационными методами: метод Зейделя

### Задача

Дана система линейных алгебраических уравнений, представленная матрицей системы  и столбцом свободных членов . Используя метод Зейделя, найти решение системы с допустимой ошибкой ., то есть столбец , такой, что: .

Исследовать точность решения, когда определитель матрицы близок к 0.

### Алгоритм и условия применимости метода

Рассмотрим невырожденную матрицу системы и столбец свободных членов . Приведем систему к виду: . Возьмём полученную формулу за основу итерационного процесса: (за при этом принимается некий столбец произвольных значений). Усовершенствуем процесс, подставляя вместо значения уже вычисленных компонент . Такой метод решения называется методом решения Зейделя и выражается формулой:

*.* При этом если , то итерационный процесс сходится.

Закономерно возникает вопрос о поиске гарантирующего сходимость преобразования системы. Для его нахождения воспользуемся двумя теоремами:

1. Пусть – симметричная положительно определенная матрица системы. Возьмём для метода Зейделя коэффициенты вида: . Тогда итерационный процесс гарантированно сходится.

2. Пусть – невырожденная матрица. Тогда произведение представляет собой симметричную положительно определенную матрицу.

Из изложенного выше легко заключить, что достаточно умножить систему слева на матрицу и выразить соответствующие коэффициенты:

*.*

### Предварительный анализ задачи

При использовании метода будут рассматриваться невырожденные матрицы, следовательно, система имеет единственное решение.

Выход из итерационного процесса будет производиться по формуле: .

### Тестовый пример с детальными расчётами

Для демонстрации метода решим систему:

, ,

Тогда имеем:

Возьмём .

После 100 итераций получим ответ:

### Перечень контрольных тестов

Для исследования работы метода было решено:

- 2 СЛАУ с симметричными положительно определенными матрицами системы, обладающими числами обусловленности 100 0 1000 соответственно.

- 4 СЛАУ с определителями, близкими к нулю.

- 2 СЛАУ с невырожденными матрицами системы произвольного вида.

Для каждой системы найдено несколько решений при значениях допустимой ошибки  При решении произвольных систем были дополнительно вычислены при . В случае с симметричными положительно определенными матрицами метод применялся дважды: с предварительным преобразованием системы и без, со сравнением полученных результатов.

Для СЛАУ с определителями, близкими к нулю, вычисление проводилось единственный раз с точностью .

Все системы имеют размерность .

### Модульная структура программы

double\* **MatrixMultiply** (double\* A, double\* B, int n) – функция вычисления произведения двух квадратных матриц. Принимает матрицу A, матрицу B и размерность n, возвращает произведение матриц – квадратную матрицу С = AB.

double\* **MBVMultiply** (double\* A, double \*B, int n) – функция вычисления произведения квадратной матрицы A и вектора-столбца B. Принимает матрицу A, вектор-столбец В и размерность матрицы и столбца n. Возвращает результат произведения – вектор-столбец D = AB.

double\* **MatrixTransform** (double \*A, double \*b, int n) – функция, трансформирующая исходную систему в систему вида , имеющую те же корни и гарантирующую сходимость метода. Принимает матрицу А, столбец свободных членов b и размерность системы n. Возвращает матрицу, содержащую коэффициенты, использующиеся при итерации.

double **ErrorEval** (int n, double \*A, double \*x, double \*b) – функция нахождения нормы вектора невязки (погрешности). Принимает размерность системы, матрицу А, столбец решений x и столбец свободных членов b. Возвращает значение нормы вектора невязки.

double\* **Zeidel** (double \*A, double \*b, int n, double err) – функция решения системы методом итерации Зейделя. Принимает матрицу A, столбец свободных членов b, размерность системы n и значение допустимой ошибки ε. Возвращает столбец решений x.

### Численный анализ решения и результаты

В ходе исследования метода были получены следующие результаты:

ε' обозначает норму вектора невязки, ε – допустимую ошибку, i – число итераций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | С преобразованием | | Без преобразования | |
| ε | ε' | i | ε' | i |
| 0,001 | 1.553662 | 8119 | 0.036136 | 225 |
| 0,0001 | 0.155480 | 9989 | 0.003604 | 262 |
| 0,000001 | 0.001555 | 13730 | 0.000036 | 336 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | С преобразованием | | Без преобразования | |
| ε | ε' | i | ε' | i |
| 0,001 | 283.439639 | 470072 | 0.487554 | 2051 |
| 0,0001 | 28.343787 | 833353 | 0.048649 | 2544 |
| 0,000001 | 0.283438 | 1559913 | 0.000487 | 3529 |

Произвольные СЛАУ:

|  |  |  |
| --- | --- | --- |
| ε | ε' | i |
| 0,001 | 2890.659767 | 271 |
| 0,0001 | 285.647089 | 759 |
| 0,000001 | 2.859432 | 1729 |
| 0,00000001 | 0.028624 | 2699 |

|  |  |  |
| --- | --- | --- |
| ε | ε' | i |
| 0,001 | 8054.150280 | 415 |
| 0,0001 | 805.786045 | 1768 |
| 0,000001 | 8.051575 | 4475 |
| 0,00000001 | 0.080590 | 7181 |

Для получения матриц с определителем, близким к нулю, при случайной генерации собственные числа (порядком от 10 до 10000 включительно) делились на заданную = величину k. Полученные данные:

|  |  |  |
| --- | --- | --- |
| k | ε' | i |
| 10 | 0.347182 | 1866 |
| 100 | 0.027244 | 1795982875 |
| 1000 | 0.035243 | 1008528080 |
| 10000 | N/A | N/A |

### Выводы

Результаты исследования метода выявили значительное ухудшение в скорости сходимости метода при применении изложенного выше преобразования в случае, когда матрица системы уже соответствует условиям сходимости. Данная особенность объясняется тем фактом, что собственные числа матрицы являются собственными числами , возведенными квадрат, а исходя из этого:

То есть в результате число обусловленности также возводится в квадрат, что может привести к проблемам со скоростью вычисления корней. Из этого вытекает необходимость предварительного изучения системы для получения приемлемых результатов.

Тестирование метода на матрицах с определителем, близким к нулю, показало крайне плохую сходимость и метода. Так, при делении собственных чисел на k=1000 процесс сошёлся за около 2000000000 (!) итераций, а при k=10000 процесс начал расходиться.

### Приложение

#### Программный код

double\* **MatrixMultiply** (double\* A, double\* B, int n)

{

int i, j, k;

double \*result = NULL;

long double sum = 0;

result = calloc (n\*n, sizeof(double));

for (i = 0; i<n; i++)

for (j = 0; j<n; j++)

{ sum = 0;

for (k = 0; k<n; k++)

sum+=A[i\*n+k]\*B[k\*n+j];

result[i\*n+j] = sum;

}

return result;

}

double\* **MBVMultiply** (double\* A, double \*B, int n)

{

int i, j;

double \*result = NULL;

result = calloc (n, sizeof(double));

for (i = 0; i<n; i++)

for (j = 0; j<n; j++)

result[i]+=A[i\*n+j]\*B[j];

return result;

}

double\* **TRSPN** (double \*matrix, double \*trmatrix, int n)

{

int i, j;

for (i = 0; i<n; i++)

for (j = 0; j<n; j++)

trmatrix[i\*n+j] = matrix[j\*n+i];

return trmatrix;

}

double **MatrixInfiniteNorm** (double \*matrix, int n)

{

int i, j;

double norm = 0;

for (i = 0; i<n; i++)

for (j = 0; j<=i; j++)

if (norm<fabs(matrix[i\*n+j]))

norm = fabs(matrix[i\*n+j]);

return norm;

}

double **VectorInfiniteNorm** (double \*vector, int n)

{

int i;

double norm = 0;

for (i = 0; i<n; i++)

if (norm<fabs(vector[i]))

norm = fabs(vector[i]);

return norm;

}

double\* **VectorDiff** (double \*A, double \*B, double \*Dif, int n)

{

int i;

for (i = 0; i<n; i++)

Dif[i] = A[i]-B[i];

return Dif;

}

double \***MatrixDiff** (double \*A, double \*B, double \*Dif, int n)

{

int i;

for (i = 0; i<n\*n; i++)

Dif[i] = A[i]-B[i];

return Dif;

}

double **ErrorEval** (int n, double \*A, double \*x, double \*b)

{

int i, j;

double sum = 0;

double \*help = NULL;

help = calloc (n, sizeof(double));

for (i = 0; i<n; i++)

{

for (j = 0; j<n; j++)

help[i]+=A[i\*n+j]\*x[j];

help[i]-=b[i];

}

for (i = 0; i<n; i++)

sum+=help[i]\*help[i];

free (help);

return sqrt(sum);

}

double\* **MatrixTransform**(double \*A, double \*b, int n)

{

enum {

sympos,

standard

} convmode;

int i, j, k;

double \*At = NULL, \*C = NULL, \*D = NULL;

convmode = TRANSFORMMODE;

At = malloc (n\*n\*sizeof(double));

if (convmode == standard)

{

At = malloc (n\*n\*sizeof(double));

At = TRSPN (A, At, n);

C = MatrixMultiply (At, A, n);

D = MBVMultiply (At, b, n);

for (i = 0; i<n; i++)

{

for (j = 0; j<n; j++)

if (j==i)

continue;

else

{

At[i\*n+j] = -C[i\*n+j]/C[i\*n+i];

}

At[i\*n+i] = D[i]/C[i\*n+i];

}

free (C);

free (D);

}

else

{

for (i = 0; i<n; i++)

{

for (j = 0; j<n; j++)

if (j==i)

continue;

else

{

At[i\*n+j] = -A[i\*n+j]/A[i\*n+i];

}

At[i\*n+i] = b[i]/A[i\*n+i];

}

}

return At;

}

double\* **Zeidel** (double \*A, double \*b, int n, double err)

{

int i, j;

double \*x = NULL, \*y = NULL, \*dif = NULL, \*C = NULL, errstimate = DBL\_MAX, errstimate2 = DBL\_MAX;

C = MatrixTransform (A, b, n);

x = malloc (n\*sizeof(double));

y = calloc (n, sizeof(double));

dif = malloc (n\*sizeof(double));

for (i = 0; i<n; i++)

x[i] = 1;

while (errstimate>=err)

{

for (i = 0; i<n; i++)

{

for (j = 0; j<i; j++)

y[i]+=C[i\*n+j]\*y[j];

for (j = i+1; j<n; j++)

y[i]+=C[i\*n+j]\*x[j];

y[i]+=C[i\*n+i];

}

dif = VectorDiff (y, x, dif, n);

errstimate = VectorInfiniteNorm(dif, n);

for (i = 0; i<n; i++)

{

x[i] = y[i];

y[i] = 0;

}

k++;

if (k>999999&&k%100000==0)

printf ("Current error: %lf\n", errstimate);

}

errstimate2 = ErrorEval (n, A, x, b);

printf ("Error: %lf\n", errstimate2);

free (A);

free (y);

free (C);

free (b);

free (dif);

return x;

}

# Часть 4. Решение алгебраической проблемы собственных значений: Метод Якоби

### Задача

Дана симметричная матрица , то есть такая , что . Используя итерационный метод Якоби, найти собственные числа и собственные векторы с заданным значением допустимой ошибки .

### Алгоритм и условия применимости метода

Рассмотрим симметричную матрицу и некую матрицу плоских вращений

, , где – некоторый угол.

Тогда преобразованная матрица также является симметричной и обладает теми же собственными числами, что и матрица . Подберём такое значение угла , чтобы при преобразовании матрицы элемент (и, ввиду симметричности, ) обращался в 0. При совершении двух матричных произведений имеем:

, откуда из тригонометрических формул вытекает: .

Вместо достаточно трудоёмкого вычисления угла найдём непосредственно значения и , воспользовавшись формулой тангенса двойного угла и решив квадратное уравнение. Пусть , тогда

Применяя изложенные выше сведения, можно построить следующий итерационный процесс, приводящий матрицу к матрице, близкой к диагональной:

1. Найдём элемент

2. Вычислим приведённым ранее методом необходимые для обнуления элемента значения , построим соответствующую матрицу

3. Совершим преобразование .

4. Повторим все шаги для новой матрицы (получив тем самым уже матрицу

Процесс протекает до тех пор, пока , где – заданное значение допустимой ошибки. В результате получим матрицу на диагонали которой располагаются элементы со значениями, близкими к собственным значениям исходной матрицы . При этом собственными векторами приближенно являются столбцы матрицы

*.*

Ввиду того, что в результате преобразования изменяются только элементы, лежащие на строке или столбце , достаточно совершать лишь следующие вычисления:

### Предварительный анализ задачи

Так как – симметричная матрица, то она всегда имеет полный набор различных собственных векторов.

### Тестовый пример с детальными расчётами

Возьмём матрицу . Для такой матрицы в качестве устраняемого элемента берётся Тогда:

.

Для задачи данной размерности алгоритм останавливается после первого шага. В итоге имеем:

### Перечень контрольных тестов

Для исследования метода были рассмотрены четыре разных матрицы: две матрицы с различными отделимостями собственных чисел (хорошо и плохо отделимые) и две матрицы, различающиеся по значению обусловленности (хорошо и плохо обусловленные). Для каждой матрицы были вычислены собственные значения с тремя допустимыми погрешностями: и

### Модульная структура программы

double\* **JacRotate** (double \*arr, int n, int j, int k, double sin, double cos) – функция, совершающая преобразование матрицы. Принимает матрицу arr, размерность n, строку j и столбец k, в которых совершается преобразование, а также значения sin и cos матрицы поворота. Возвращает преобразованную матрицу arr.

double\* **EigenUpdate** (double \*arr, int n, int j, int k, double sin, double cos) – функция обновления приближенной матрицы собственных векторов. Принимает матрицу собственных векторов из предыдущей итерации arr, размерность n, строку j и столбец k, в которых совершается преобразование, а также значения sin и cos матрицы поворота. Возвращает обновленную матрицу arr.

double\* **JacEigenFind** (double \*arr, int n, double err) – функция, вычисляющая приближенные значения собственных чисел и векторов симметричной матрицы. Принимает матрицу arr, размерность n и значение допустимой ошибки err. Возвращает столбец, содержащий приближенные собственные значения матрицы и приближенную матрицу собственных векторов (как глобальную переменную).

double **ErrorEval** (double \*arr, double \*eigenv, double \*eigenval, int n) – функция вычисления действительной ошибки. Принимает начальную матрицу arr, приближенную матрицу собственных векторов eigenv, столбец приближенных собственных значений и размерность n. Возвращает максимальное из значений .

### Численный анализ решения и результаты

В результате исследования метода были получены следующие данные:

ε' обозначает действительную ошибку, ε – допустимую ошибку, i – число итераций

Для матрицы с плохо отделимыми собственными числами:

|  |  |  |
| --- | --- | --- |
| ε | ε' | I |
| 0.001 | 0.000833 | 62 |
| 0.0001 | 0.000077 | 70 |
| 0.000001 | 0.000001 | 114 |

Рассмотрим более подробно действительные и вычисленные собственные значения при каждой допустимой ошибке:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0.001 | 0.0001 | 0.000001 |
|  |  |  |  |
| 6.000052 | 6.000070 | 6.000070 | 6.000052 |
| 6.000024 | 6.000024 | 6.000024 | 6.000024 |
| 6.000075 | 6.000075 | 6.000075 | 6.000075 |
| 6.000094 | 6.000075 | 6.000075 | 6.000094 |
| 142 | 141.9999 | 141.9999 | 141.9999 |
| 633 | 633 | 633 | 633 |
| 2636 | 2636 | 2636 | 2636 |
| 4 | 4 | 4 | 4 |
| 367 | 367 | 367 | 367 |
| 345 | 345 | 345 | 345 |

Для матрицы с хорошо отделимыми собственными числами:

|  |  |  |
| --- | --- | --- |
| ε | ε' | i |
| 0.001 | 0.000885 | 102 |
| 0.0001 | 0.000028 | 109 |
| 0.000001 | 0.000001 | 128 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0.001 | 0.0001 | 0.000001 |
|  |  |  |  |
| 142 | 142 | 142 | 142 |
| 645 | 645 | 645 | 645 |
| 877 | 877 | 877 | 877 |
| 1024 | 1023.9999 | 1023.9999 | 1023.9999 |
| 533 | 533 | 533 | 533 |
| 678 | 678 | 678 | 678 |
| 455 | 455 | 455 | 455 |
| 5 | 5 | 4 | 4 |
| 64 | 64 | 64 | 64 |
| 77 | 77 | 77 | 77 |

Для хорошо обусловленной матрицы (:

|  |  |  |
| --- | --- | --- |
| ε | ε' | i |
| 0.001 | 0.000481 | 91 |
| 0.0001 | 0.000087 | 99 |
| 0.000001 | 0.000001 | 116 |

Для плохо обусловленной матрицы (:

|  |  |  |
| --- | --- | --- |
| ε | ε' | i |
| 0.001 | 0.000872 | 106 |
| 0.0001 | 0.000101 | 131 |
| 0.000001 | 0.000001 | 562 |

### Выводы

Как видно из изложенных выше данных, при поиске плохо отделимых собственных чисел с недостаточно малой допустимой ошибкой метод Якоби работает с большой относительной погрешностью, что может приводить к получению ложной информации о кратности собственных чисел, а также вызвать ряд проблем для вычислений, где необходима высокая точность найденных собственных значений. Однако при допустимой ошибке того же порядка, что и разность между собственными числами, метод работает корректно.

При исследовании поведения метода при разных значениях обусловленности было выявлено небольшое различие в скорости сходимости при малых значениях допустимой ошибки и высокое различие в случае, когда .

В целом можно заключить о высокой удобности применения и точности метода Якоби, а также его преимущества в виде параллельного вычисления всех собственных векторов и значений. Последнее, однако, обуславливает невысокую скорость сходимости по сравнению с другими методами вычисления собственных значений.

### Приложение

#### Программный код

double **VectorInfiniteNorm** (double \*vector, int n)

{

int i;

double norm = 0;

for (i = 0; i<n; i++)

if (norm<fabs(vector[i]))

norm = fabs(vector[i]);

return norm;

}

double\* **MBVMultiply** (double\* A, double \*B, double \*result, int n)

{

int i, j;

for (i = 0; i<n; i++)

{

result[i] = 0;

for (j = 0; j<n; j++)

result[i]+=A[i\*n+j]\*B[j];

}

return result;

}

double\* **IGenerate** (int n)

{

int i;

double \*I = calloc (n\*n, sizeof(double));

for (i = 0; i<n; i++)

I[i\*n+i] = 1;

return I;

}

double **ErrorEval** (double \*arr, double \*eigenv, double \*eigenval, int n)

{

int i, j, k;

double err, maxerr = 0, \*h1 = NULL, \*errorvect = NULL, \*oneeig = NULL;

oneeig = malloc(n\*sizeof(double));

h1 = malloc (n\*sizeof(double));

errorvect = malloc(n\*n\*sizeof(double));

for (k = 0; k<n; k++)

{

for (i = 0; i<n; i++)

for (j = 0; j<n; j++)

{

errorvect[i\*n+j] = arr[i\*n+j];

if (i==j)

errorvect[i\*n+j]-=eigenval[k];

}

for (j = 0; j<n; j++)

oneeig[j] = eigenv[k\*n+j];

h1 = MBVMultiply (errorvect, oneeig, h1, n);

err = VectorInfiniteNorm (h1, n);

if (err>maxerr)

maxerr = err;

}

free (oneeig);

free (errorvect);

return maxerr;

}

double\* **JacRotate** (double \*arr, int n, int j, int k, double sin, double cos)

{

int m;

double arrjj, arrjm, arrkm, arrkk, arrjk;

arrjj = arr[j\*n+j];

arrkk = arr[k\*n+k];

arrjk = arr[j\*n+k];

for (m = 0; m<n; m++)

{

if (m==j||m==k)

continue;

arrjm = arr[j\*n+m];

arrkm = arr[k\*n+m];

arr[j\*n+m] = cos\*arrjm - sin\*arrkm;

arr[m\*n+j] = arr[j\*n+m];

arr[k\*n+m] = sin\*arrjm + cos\*arrkm;

arr[m\*n+k] = arr[k\*n+m];

}

arr[j\*n+j] = cos\*cos\*arrjj - 2\*sin\*cos\*arr[j\*n+k] + sin\*sin\*arrkk;

arr[k\*n+k] = sin\*sin\*arrjj + 2\*sin\*cos\*arr[j\*n+k] + cos\*cos\*arrkk;

arr[j\*n+k] = (cos\*cos-sin\*sin)\*arrjk+sin\*cos\*(arrjj-arrkk);

arr[k\*n+j] = arr[j\*n+k];

return arr;

}

double\* **EigenUpdate** (double \*arr, int n, int j, int k, double sin, double cos)

{

int m;

double arrjm, arrkm;

for (m = 0; m<n; m++)

{

arrjm = arr[j\*n+m];

arrkm = arr[k\*n+m];

arr[j\*n+m] = cos\*arrjm - sin\*arrkm;

arr[k\*n+m] = sin\*arrjm + cos\*arrkm;

}

return arr;

}

double\* **JacEigenFind** (double \*arr, int n, double err)

{

double sin, cos, t, tau, errstimate, \*eigenval = NULL, \*arrcpy = NULL;

int iter = 0, i, l, j, k;

eigenval = malloc (n\*sizeof(double));

eigenvect = IGenerate (n);

while (1)

{

j = 1;

k = 0;

for (l = 0; l<n-1; l++)

for (i = l+1; i<n; i++)

if (fabs(arr[i\*n+l]) > fabs(arr[j\*n+k]))

{

j = i;

k = l;

}

for (i = 0; i<n; i++)

eigenval[i] = arr[i\*n+i];

errstimate = fabs(arr[j\*n+k]);

if (errstimate<=err)

{

printf ("Number of iterations: %d\n", iter);

return eigenval;

}

if (arr[j\*n+j]==arr[k\*n+k])

{

sin = sqrt(2)/2;

cos = sin;

}

else

{

tau = (arr[k\*n+k] - arr[j\*n+j])/(2\*arr[j\*n+k]);

if (tau<0)

t = -sqrt(1+tau\*tau) - tau;

else

t = sqrt(1+tau\*tau) - tau;

cos = 1/sqrt(1+t\*t);

sin = t\*cos;

}

arr = JacRotate (arr, n, j, k, sin, cos);

eigenvect = EigenUpdate (eigenvect, n, j, k, sin, cos);

iter++;

}

}

# Выводы

После проделанной работы кратко повторим выводы о каждом методе:

1. Метод дихотомии показал себя как неприхотливый и простой способ решения алгебраических и трансцендентных уравнений. Рекомендуется к использованию в случаях с небольшими требованиями к точности и необходимостью быстрой реализации алгоритма.

2. Метод секущих, хоть и обладает чуть меньшей чем у метода Ньютона скоростью сходимости, не требует многократного вычисления значения производной, что может ускорить процесс поиска корня, особенно в случаях, когда вычисление производной является ресурсоёмким процессом.

3. Метод квадратного корня является эффективным и быстрым прямым методом решения СЛАУ в пределах своих условий применимости. Однако в случае для СЛАУ, представленных матрицами с большим числом обусловленности, даже небольшие возмущения могут вызвать потерю положительной определённости и, как следствие, работоспособности метода. Существует возможность расширить применимость, путём умножения системы слева на , однако такой способ может повлечь за собой ещё большее расхождение с действительным ответом при малых погрешностях ввода начальных данных.

4. Метод Зейделя – несложный в реализации метод решения СЛАУ, обладающий при этом достаточно быстрой сходимостью. Однако вопрос его сходимости может вызывать затруднения, а потому для эффективной работы метода рекомендуется предварительное изучение системы.

5. Метод Якоби для нахождения собственных значений привлекателен относительной простотой стоящей за ним идеи, точностью и возможностью одновременного вычисления всех собственных значений матрицы. В некоторых случаях этим преимущества могут перекрываться ограниченной областью применения и не самой высокой скоростью сходимости.