W zadaniu poproszono mnie o zaimplementowanie czterech metod iteracyjnych oraz znalezienie nimi rozwiązania układu

$$\begin{cases} y_0 = 1 \\ -(D_2y)_n + y_n = 0 \quad i = 1...N-1\ , \\ -\frac{y_{N-1}-2y_N+y_0}{h^2} = 0 \end{cases} \qquad Gdzie (D_2y)_n = \frac{y_{n-1}-2y_n+y_{n+1}}{h^2}$$

i porównanie szybkości metod - która z nich jest najszybsza oraz dlaczego.

Układ zapisany w postaci macierzowej prezentuje się następująco (z dodanym pierwszym wierszem do ostatniego w ostatnim wierszu na potrzeby implementacji) :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1, & -2+h^2, & 1, & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1, & -2+h^2, & 1, & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1, & -2+h^2, & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -2+h^2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2+h^2 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Wszystkie te metody iteracyjne (przybliżone) rozwiązywania układu równań liniowych będą w pewien sposób do siebie podobny. Są one szczególnymi przypadkami metody kolejnych przybliżeń, która jest bardzo uniwersalna. W wielu przypadkach metody iteracyjne okazują się lepsze do zastosowania od metod dokładnych. Metody iteracyjne polegają na zbliżeniu się do wyniku właściwego w skończonej ilości kroków na tyle blisko, aby wynik był dla nas zadowalający. Można robić to w sposób przedstawiony poniżej, czyli z każdym przybliżeniem sprawdzać czy po podstawieniu pod wektor niewiadomych pomnożenie macierzy A przez ten wektor x da nam wektor rozwiązań (lub chociaż zbliży się do niego przynajmniej na podaną przez nas wartość). W tym zadaniu miałam uzyskać dokładność rzędu 10^{-10}i taka była też moja maksymalna różnica w wartości od rozwiązania czyli mój maksymalny błąd - ϵ.

Rozpocznę od metody Richardsona. Przedstawia się ona wzorem:

$$x^{(n+1)} = x^{(n)} + \gamma(b - Ax^{(n)})$$

Metoda ta bardzo przypomina metodę iteracji prostej, jednak czynnik $(b - Ax^{(n)})$ został dodatkowo pomnożony przez parametr γ. Wybór tego parametru jest kluczowy dla skuteczności metody, gdyż decyduje o jej zbieżności (czy jest zbieżna i jak szybko).

Parametr można wyznaczyć wyliczając wartości własne (istotna jest największa i najmniejsza) i $\gamma_{opt} = \frac{2}{\lambda_{min}+\lambda_{max}}$

Ja dobierałam parametr metodą prób i błędów.

$x^{(0)}$czyli pierwszy wektor $x^{(n)}$w metodzie iteracji prostej dobiera się najczęściej jako $\frac{b_i}{a_{ii}}$ czyli wektor rozwiązań podzielony przez współczynnik przy i-tej niewiadomej, jednak można wybrać w tym przypadku wektor zerowy jako pierwszy i nie wpłynie to na szybkość rozwiązania (w obu przypadkach wykonało mi się tyle samo obliczeń).

Ja w rozwiązaniu umieściłam $x^{(0)} = \frac{b_i}{a_{ii}}$

W przypadku tej specyficznej macierzy, bardzo łatwo jest zaimplementować tę metodę (jak wszystkie kolejne), gdyż w każdej linii Ax pojawia się związek trzech niewiadomych z i-tym rozwiązaniem a więc można łatwo zapisać to w pętli $A \cdot x_i = a_{j-1i}x_{i-1} + a_{ji}x_i + a_{j+1i}x_{i+1}$.

W programie stworzyłam 4 wektory początkowe- diagonalę, wartości pod diagonalą oraz nad nią, wektor rozwiązań. Dodatkowo stworzylam tez 2 nowe wektory, które mają za zadanie przechowywać wartości poprzedniego i nowego przybliżenia. W pętli while podstawiam do wzoru na kolejne przybliżenie wartości poprzedniego przybliżenia (w pierwszym przebiegu pętli wybrane przeze mnie $x^{(0)}$) a następnie obliczam wartość Ax i sprawdzam czy błąd jest mniejszy od 10^{-10}. Jeśli nie to ponawiam ten krok, aż do uzyskania zadowalającego wyniku.

W programie skorzystałam z funkcji obliczania wartości bezwzględnej dla liczb rzeczywistych (fabs) oraz z funkcji clock mierzącej czas procesora.

Ostatecznie dla powyższego układu równań i dla $\gamma = 0.49996$ otrzymuję wymaganą dokładność w 213736 (ponad 213 tyś) operacjach. Zajmuje to ok 2100 ms (ok. 2 sekundy). Dla $\gamma = 0.5$ ciąg nie jest już zbieżny, dla mniejszych wartości od wybranej przeze mnie zbiega wolniej.

Algorytm zaimplementowany przeze mnie jest złożoności czasowej O(N), co sprawdziłam zwiększając rozmiar mojej macierzy (czyli tak naprawde kilku wektorow) 10 razy, program zakończył obliczenia po ok 20 sekundach, czyli czas obliczeń również wydłużył sie 10-krotnie.

Poniżej kod programu implementującego metodę oraz wyniki i wykres znalezionego x:

Instrukcje uruchomienia programu:

1. Aby uruchomić poniższy program w języku c na linuxie należy przejść do katalogu w którym znajdują się pliki i wpisać w terminalu komendę:
   –> make all

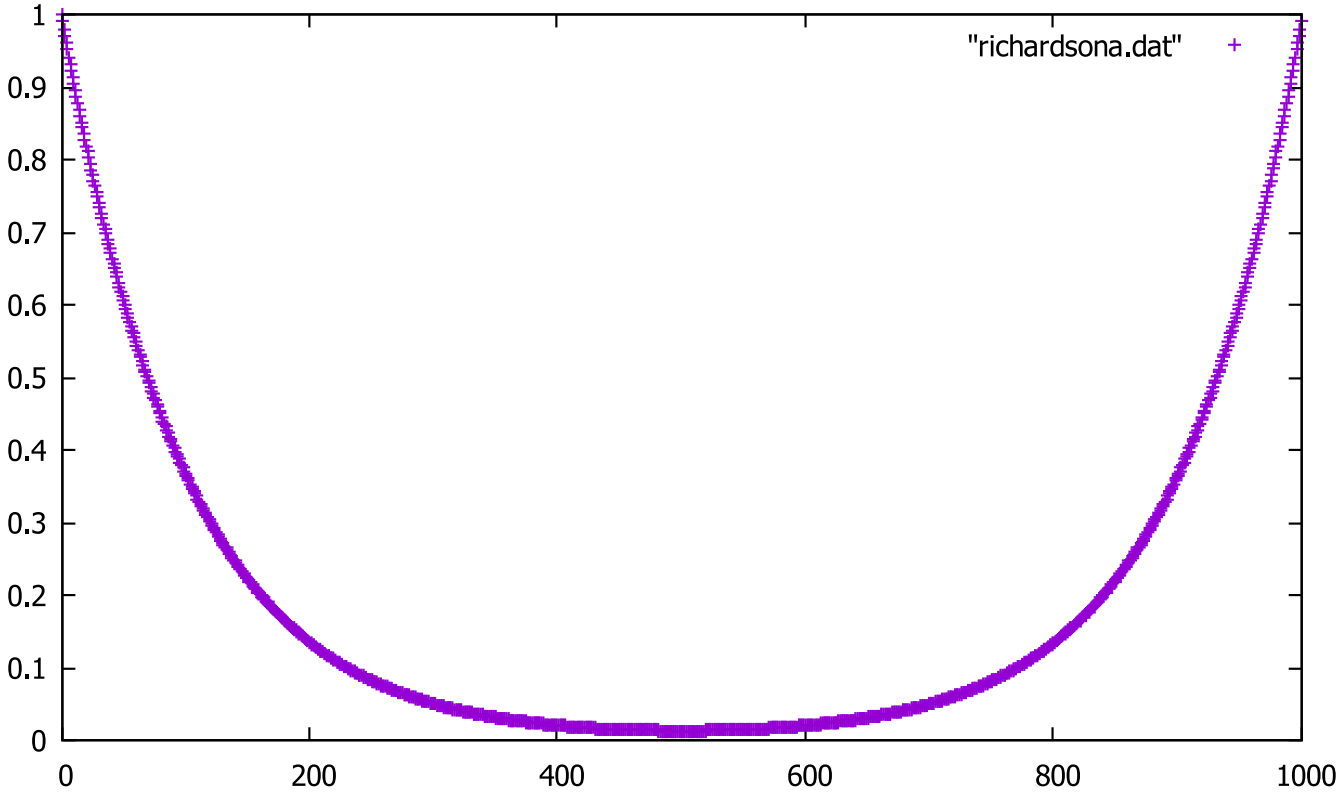
2. A następnie
   –> ./richardsona.x

3. Aby pozbyć się plików tymczasowych należy użyć komendy:
   –> make clean

   Spowoduje to utworzenie pliku tekstowego z wynikami “richardsona.dat”.

```
/////// metoda iteracji Richardsona

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main(){
FILE *plik;
if((plik=fopen("richardsona.dat","w"))==NULL) return -1;
clock_t start=clock();
int N = 1000;
double n=1000;
double h=0.01;
double hh=0.0001;
double down[N+1];
down[0]=0; down[N]=-1;
double diag[N+1];
diag[0]=1;diag[N]=2;
double up[N];
up[0]=0;
double b[N+1]; b[0]=1;
b[N]=1;
int i=0,j=0;
double e=0.0000000001;
double blad=100,blad2;
double gamma=0.49996;
double xNowe[N+1];
xNowe[0]=b[0]/diag[0];
xNowe[N]=b[N]/diag[N];
for(i=1;i<=N-1;++i){
down[i]=-1;
diag[i]=2+hh;
up[i]=-1;
b[i]=0;
xNowe[i]=b[i]/diag[i];
}



double xStare[N+1];
int counter =0;
while(blad>e){

xNowe[0]=xStare[0]+gamma*(b[0]-diag[0]*xStare[0]);
for(i=1;i<N;++i){
xNowe[i]=xStare[i]+gamma*(b[i]-(down[i]*xStare[i-1]+diag[i]*xStare[i]
+up[i]*xStare[i+1]));
}

xNowe[N]=xStare[N]+gamma*(b[N]-(down[N]*xStare[N-1]+diag[N]*xStare[N]));
```

```
blad=fabs(b[0]-(diag[0]*xNowe[0]));
xStare[0]=xNowe[0];
for(i=1;i<N;++i){
blad2=fabs(b[i]-(down[i]*xNowe[i-1]+diag[i]*xNowe[i]+up[i]*xNowe[i+1]));
if (blad2>blad) blad=blad2;
xStare[i]=xNowe[i];
}

blad2=fabs(b[N]-(down[N]*xNowe[N-1]+diag[N]*xNowe[N]));
if(blad2>blad) blad=blad2;
xStare[N]=xNowe[N];
counter++;
// printf("Blad to: %.14f\n",blad);
// printf("Counter wynosi: %d\n",counter);
}


printf("Last Counter wynosi: %d\n",counter);


clock_t stop=clock();

stop-=start;
stop/=CLOCKS_PER_SEC/1000;

printf("Czas trwania programu: %d ms\n",stop);

for(i=0;i<=N;i++){

 fprintf(plik,"%d %.14f\n",i,xNowe[i]);
}


fclose(plik);
return 0;
}
```


"richardsona.dat"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
200
400
600
800
1000

2. Metoda Jacobiego.

Metoda ta przedstawia się wzorem:

$$x^{(n+1)} = D^{-1}(b - Rx^{(n)})$$

Polega ona na zbudowaniu macierzy przekątniowej D o elementach z diagonali podanej w zadaniu macierzy.

Proces iteracyjny można zapisać jak we wzorze powyżej, ale można zapisać go też inaczej, w postaci:

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{n} a_{ij} x_j^{(n)} \right) \qquad \text{przy założeniu że i jest różne od j}$$

Aby metoda działała (była zbieżna) to $\|I - D^{-1}A\| < 1$ czyli dla wszystkich 'i' spełniona musi być nierówność $|a_{ii}| > \sum_{j=1}^{n} |a_{ij}|$

Można zatem powiedzieć, że jeśli macierz A jest dominująca przekątniowo, to dla dowolnego wektora początkowego metoda Jacobiego tworzy ciąg zbieżny do rozwiązania ukladu $Ax = b$.

Ja zamiast macierzy D utworzylam wektor zawierajacy elementy diagonalne (żeby nie inicjować niepotrzebnych elementów zerowych). Właściwie wykorzystałam zmienne oraz wektory zmiennych (tablice) z poprzedniego zadania.Obliczenie sumy również nie jest w tym przypadku trudne, gdyż mamy do czynienia z macierzą rzadką. Ponieważ jest to macierz pasmowa, to mogę zaimplementować prosty warunek iteracyjny dla niemal wszystkich wierszy (oprócz pierwszego i ostatniego), taki że:

$x^{(n+1)}[i] = \frac{b[i] - (a_{ij-1}[i] \cdot x^{(n)}[i-1] + a_{ij+1}[i] \cdot x^{(n)}[i+1])}{diag[i]}$

I powtarzać iteracje od 1 do N tyle razy, aż błąd będzie mniejszy niż błąd maksymalny. (w pętli while).

Tak jak w poprzednim przypadku zbadałam złożoność czasową i tutaj również rośnie ona liniowo, czyli O(N). Wynik poznajemy po ponad 226 tys iteracji i 2041 ms (ok. 2 sekund)

Co ciekawe, ilość iteracji wcale nie zwiększa się i również wynosi ok. 200 tys dla 10-krotnie wiekszej macierzy, lecz wynik poznajemy dopiero okolo po 20 sekundach.

Algorytm mógłby być nieco bardziej optymalny, gdybym nie tworzyla całych wektorów, a jedynie pojedyncze zmienne, gdyż macierz składa się z prawie wszystkich takich samych elementów. Jednak takie zastosowanie pozwala łatwo rozszerzać program na inne macierze.

Poniżej kod programu implementującego metodę oraz wyniki i wykres znalezionego x:

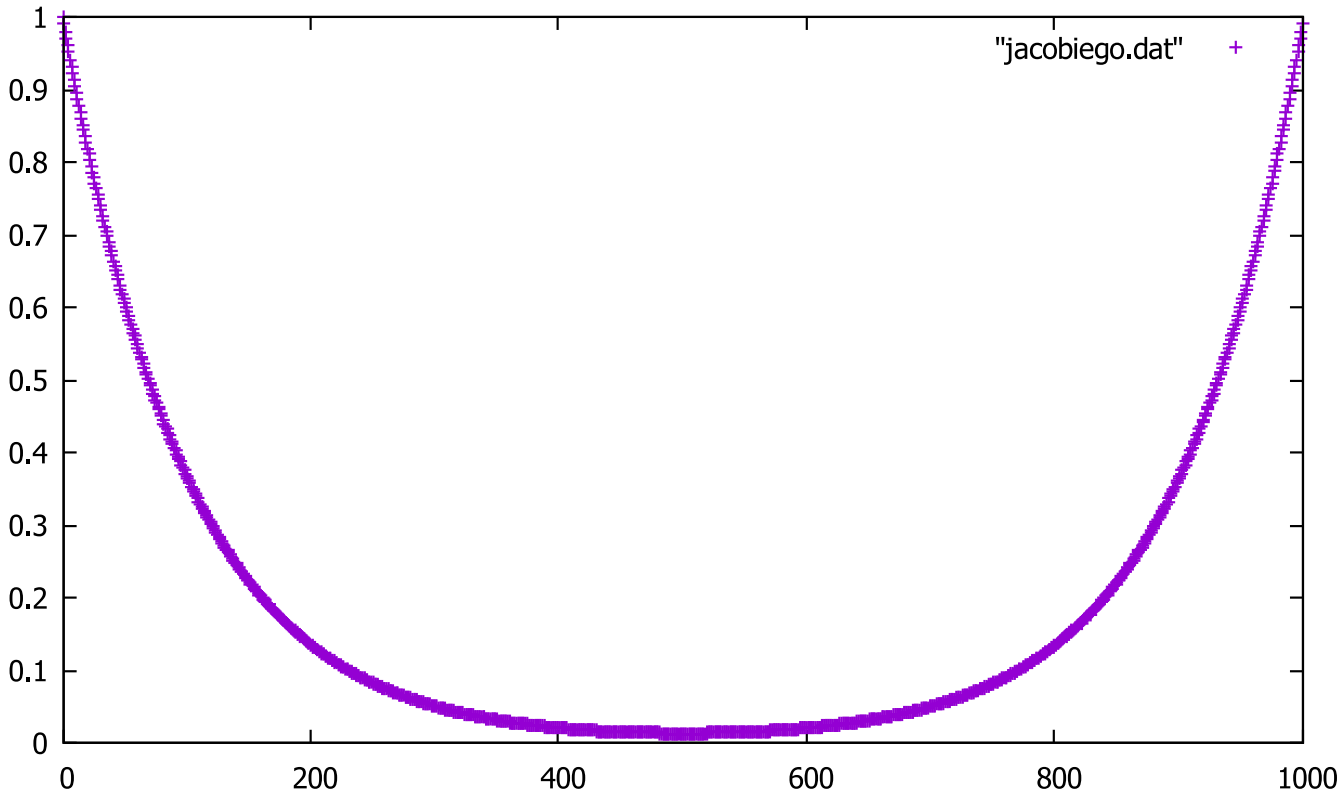
Instrukcje uruchomienia programu:

Aby uruchomić poniższy program w języku c na linuxie należy przejść do katalogu w którym znajdują się pliki i wpisać w terminalu komendę:

->make all

->./jacobiego.x

1. Spowoduje to utworzenie pliku tekstowego z wynikami “jacobiego.dat”.



```
/////// metoda iteracji Jacobiego

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main(){
FILE *plik;
if((plik=fopen("jacobiego.dat","w"))==NULL) return -1;
clock_t start=clock();
int N = 1000;
double n=1000;
double h=0.01;
double hh=0.0001;
double down[N+1];
down[0]=0; down[N]=-1;
double diag[N+1];
diag[0]=1;diag[N]=2;
double up[N];
up[0]=0;
double b[N+1]; b[0]=1;
b[N]=1;
int i=0,j=0;
double e=0.0000000001;
double blad=100,blad2;
for(i=1;i<=N-1;++i){
down[i]=-1;
diag[i]=2+hh;
up[i]=-1;
b[i]=0;
}


double xStare[N+1];
double xNowe[N+1];
int counter =0;
while(blad>e){


xNowe[0]=(b[0]-up[0]*xStare[1])/diag[0];
for(i=1;i<N;++i){

xNowe[i]=(b[i]-down[i]*xStare[i-1]-up[i]*xStare[i+1])/diag[i];

}
xNowe[N]=(b[N]-down[i]*xStare[N-1])/diag[N];


blad=fabs(b[0]-(diag[0]*xNowe[0]));
xStare[0]=xNowe[0];
```

```
for(i=1;i<N;++i){
blad2=fabs(b[i]-(down[i]*xNowe[i-1]+diag[i]*xNowe[i]+up[i]*xNowe[i+1]));
if (blad2>blad) blad=blad2;
xStare[i]=xNowe[i];
}

blad2=fabs(b[N]-(down[N]*xNowe[N-1]+diag[N]*xNowe[N]));
if(blad2>blad) blad=blad2;
xStare[N]=xNowe[N];
counter++;
//printf("Blad to: %.14f\n",blad);
//printf("Counter wynosi: %d\n",counter);
}


printf("Last Counter wynosi: %d\n",counter);


clock_t stop=clock();

stop-=start;
stop/=CLOCKS_PER_SEC/1000;

printf("Czas trwania programu: %d ms\n",stop);


for(i=0;i<=N;i++){
 fprintf(plik,"%d %.14f\n",i,xNowe[i]);
}


fclose(plik);
return 0;
}
```

3. Metoda Gaussa-Seidela
podana jest wzorem

$$x^{(n+1)} = L^{-1}(b - Ux^{(n)})$$

jednak w praktyce jest to metoda bardzo podobna do metody Jacobiego, z tą różnicą że wykorzystujemy obliczenia zarówno z poprzedniej iteracji jak i te obliczone już w danej iteracji. Dzięki temu czasami jesteśmy w stanie znacznie szybciej otrzymać rozwiązanie (niestety nie zawsze ta metoda jest szybsza od Jacobiego)

Różnicę łatwiej zauważyć korzystając ze wzoru na sume;
wzór równoważny to

$$GS: \quad x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(n+1)} - \sum_{j=1+1}^{n} a_{ij} x_j^{(n)} \right)$$

podczas gdy metoda Jacobiego to byłoby

$$J: \quad x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(n)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(n)} \right)$$

- czyli w metodzie Seidela $x_i^{(n+1)}$ które zostało obliczone wykorzystuje już do dalszych obliczeń, a nie jak w metodzie Jacobiego, uaktualniam $x^{(n+1)}$ dopiero na koniec iteracji po "i".

Tak jak w poprzednich programach utworzyłam wektory które zawierają wszystkie diagonale, wektor rozwiązań oraz 2 wektory przybliżenia niewiadomych.

Ponieważ jest to macierz pasmowa, to mogę zaimplementować prosty warunek iteracyjny, podobny do poprzedniego, dla niemal wszystkich wierszy (oprócz pierwszego i ostatniego), taki że:
$x^{(n+1)}[i] = \frac{b[i] - (a_{ij-1}[i] \cdot x^{(n+1)}[i-1] + a_{ij+1}[i] \cdot x^{(n)}[i+1])}{diag[i]}$

I dopóki warunek że błąd$<\epsilon$ niej jest spełniony to iteracja jest powtarzana.

Złożoność czasowa rośnie liniowo. Rozwiązanie otrzymałam po ponad 106 tys iteracjach w 1681 ms.

Dla macierzy o wymiarze N=10000 rozwiązanie otrzymuje po podobnej liczbie iteracji po

ok 16 sekundach.

Dla tej macierzy jest to szybsza metoda od Jacobiego i Richardsona.

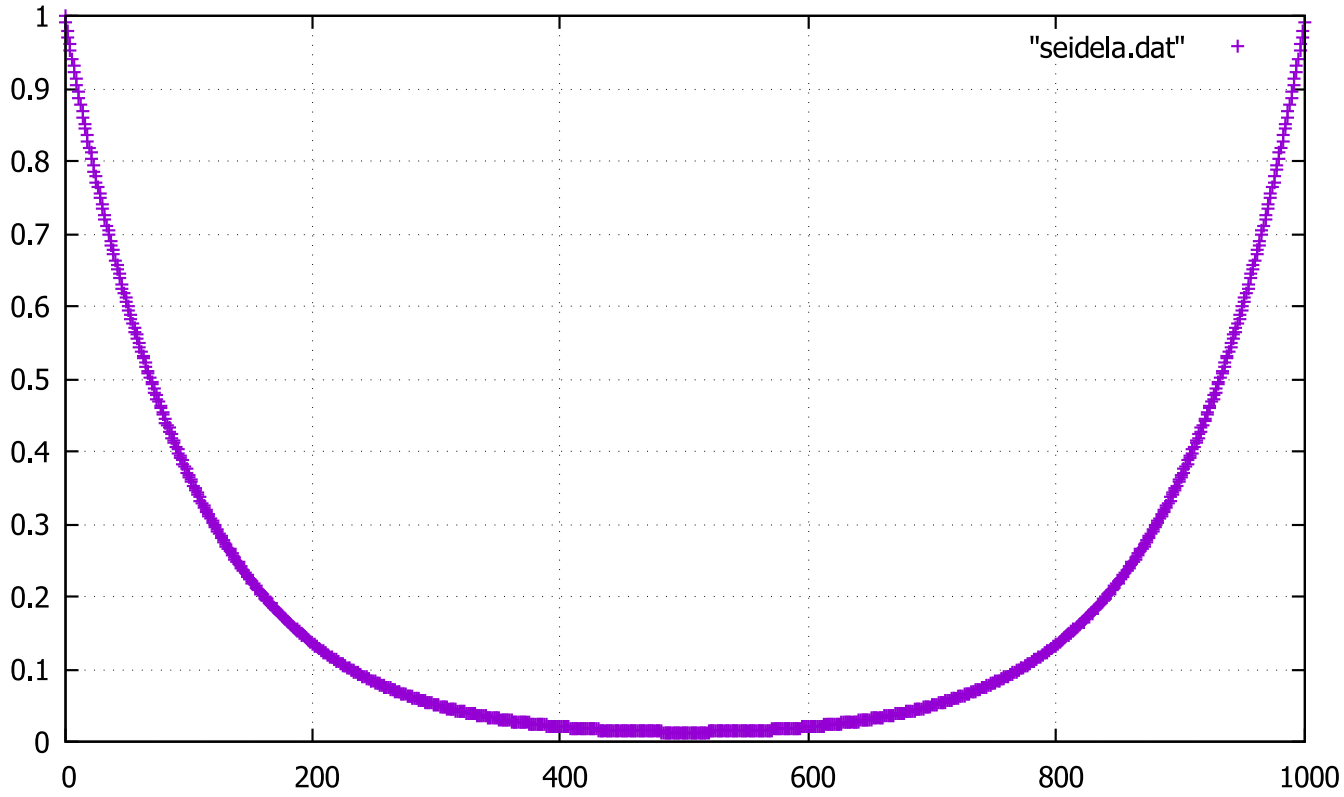
Poniżej kod programu implementującego metodę oraz wyniki i wykres znalezionego x:

Aby uruchomić poniższy program w języku c na linuxie należy przejść do katalogu w którym znajdują się pliki i wpisać w terminalu komendę:

->make all

->./gaussa-seidela.x

Spowoduje to utworzenie pliku tekstowego z wynikami “seidela.dat”.



```
/////// metoda iteracji Seidela

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main(){
FILE *plik;
if((plik=fopen("seidela.dat","w"))==NULL) return -1;
clock_t start=clock();
int N = 1000;
double n=1000;
double h=0.01;
double hh=0.0001;
double down[N+1];
down[0]=0; down[N]=-1;
double diag[N+1];
diag[0]=1;diag[N]=2;
double up[N];
up[0]=0;
double b[N+1]; b[0]=1;
b[N]=1;
int i=0,j=0;
double e=0.0000000001;
double blad=100,blad2;
for(i=1;i<=N-1;++i){
down[i]=-1;
diag[i]=2+hh;
up[i]=-1;
b[i]=0;
}


double xStare[N+1];
double xNowe[N+1];
int counter =0;
while(blad>e){


xNowe[0]=(b[0]-up[0]*xStare[1])/diag[0];
for(i=1;i<N;++i){

xNowe[i]=(b[i]-down[i]*xNowe[i-1]-up[i]*xStare[i+1])/diag[i];

}
xNowe[N]=(b[N]-down[i]*xNowe[N-1])/diag[N];


blad=fabs(b[0]-(diag[0]*xNowe[0]));
xStare[0]=xNowe[0];
```

```
for(i=1;i<N;++i){
blad2=fabs(b[i]-(down[i]*xNowe[i-1]+diag[i]*xNowe[i]+up[i]*xNowe[i+1]));
if (blad2>blad) blad=blad2;
xStare[i]=xNowe[i];
}

blad2=fabs(b[N]-(down[N]*xNowe[N-1]+diag[N]*xNowe[N]));
if(blad2>blad) blad=blad2;
xStare[N]=xNowe[N];
counter++;
//printf("Blad to: %.14f\n",blad);
//printf("Counter wynosi: %d\n",counter);
}


printf("Last Counter wynosi: %d\n",counter);


clock_t stop=clock();

stop-=start;
stop/=CLOCKS_PER_SEC/1000;

printf("Czas trwania programu: %d ms\n",stop);


for(i=0;i<=N;i++){
 fprintf(plik,"%d %.14f\n",i,xNowe[i]);
}


fclose(plik);
return 0;
}
```

4. Ostatnią metodą iteracyjną będzie metoda nadrelaksacji (successive over-relaxation method (SOR)).

Przedstawia ją wzór

$$x_i^{(n+1)} = (1-\omega)x_i^{(n)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(n+1)} - \sum_{j>i} a_{ij}x_j^{(n)}\right), \quad i = 1, 2, ..., N.$$

Podobnie jak metodę iteracji prostej można było przyspieszyć parametrem (metoda Richardsona), tak również metodę Gaussa Seidela da się ulepszyć. SOR jest uznawany za jedną z najlepszych prostych metod iteracyjnych.

Nazwa metody (nadrelaksacja) wzięła się stąd, że najczęściej parametr $\omega > 1$.

Jednak według lematu Kahana o dopuszczalnych wartościach parametru : Jeśli A ma niezerową diagonalę, a metoda SOR jest zbieżna, to musi być $0 < \omega < 2$.

Ja wybrałam metodą prób i błędów $\omega = 1.979$.

W tej metodzie wystarczyło zmodyfikować poprzedni program dodając człony w których występuje ω.

Złożoność czasowa również jest tu O(N), ale tym razem program wykonuje obliczenia znacznie szybciej w ok 1 tys iteracji (i zajmuje to ok 20 ms).

Jest to metoda która działa dla podanego układu równań zdecydowanie najszybciej.

Działanie programu można byłoby dodatkowo zoptymalizować nie tworząc wektorów a jedynie korzystać z pojedynczych wartości, gdyż na diagonalach występują takie same wartości. Ja jednak utworzyłam wektory, gdyż program jest bardziej uniwersalny, a i tak nie tworzę całej macierzy lecz wektory, więc złożoność nadal jest liniowa.

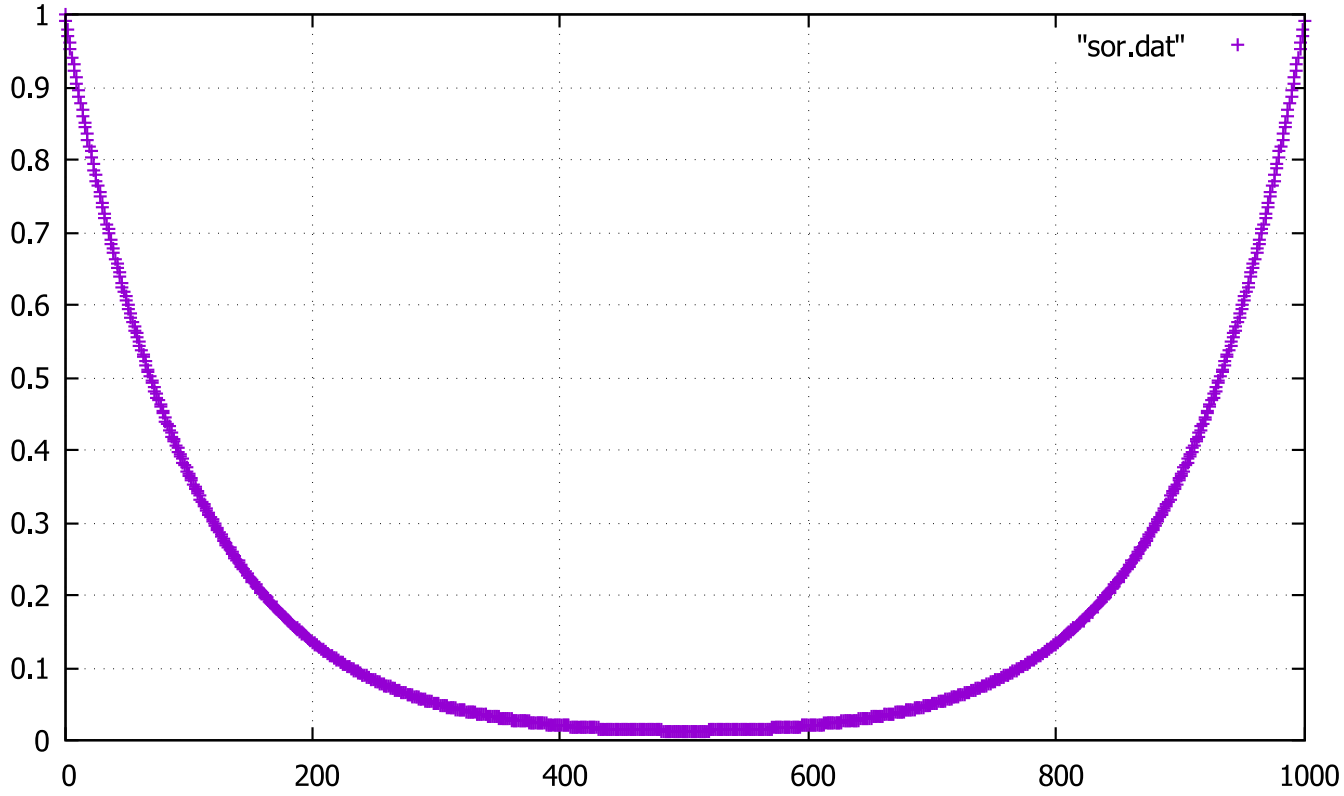
Poniżej kod programu implementującego metodę oraz wyniki i wykres znalezionego x:

Aby uruchomić poniższy program w języku c na linuxie należy przejść do katalogu w którym znajdują się pliki i wpisać w terminalu komendę:

->make all

->./sor.x

Spowoduje to utworzenie pliku tekstowego z wynikami “sor.dat”.


"sor.dat"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
200
400
600
800
1000

```
/////// metoda iteracji SOR

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main(){
FILE *plik;
if((plik=fopen("sor.dat","w"))==NULL) return -1;
clock_t start=clock();
int N = 1000;
double n=1000;
double h=0.01;
double hh=0.0001;
double down[N+1];
down[0]=0; down[N]=-1;
double diag[N+1];
diag[0]=1;diag[N]=2;
double up[N];
up[0]=0;
double b[N+1]; b[0]=1;
b[N]=1;
int i=0,j=0;
double e=0.0000000001;
double blad=100,blad2;
double omega=1.979;
for(i=1;i<=N-1;++i){
down[i]=-1;
diag[i]=2+hh;
up[i]=-1;
b[i]=0;
}


double xStare[N+1];
double xNowe[N+1];
int counter =0;
while(blad>e){


xNowe[0]=(1-omega)*xStare[0]+(omega*(b[0]-up[0]*xStare[1])/diag[0]);
xStare[0]=xNowe[0];
for(i=1;i<N;++i){

xNowe[i]=(1-omega)*xStare[i]+(omega*(b[i]-down[i]*xStare[i-1]-
up[i]*xStare[i+1])/diag[i]);
xStare[i]=xNowe[i];
}
xNowe[N]=(1-omega)*xStare[N] +(omega*(b[N]-down[i]*xStare[N-1])/diag[N]);
xStare[N]=xNowe[N];
```

```
blad=fabs(b[0]-(diag[0]*xNowe[0]));
for(i=1;i<N;++i){
blad2=fabs(b[i]-(down[i]*xNowe[i-1]+diag[i]*xNowe[i]+up[i]*xNowe[i+1]));
if (blad2>blad) blad=blad2;
}

blad2=fabs(b[N]-(down[N]*xNowe[N-1]+diag[N]*xNowe[N]));
if(blad2>blad) blad=blad2;
counter++;
// printf("Blad to: %.14f\n",blad);
//printf("Counter wynosi: %d\n",counter);
}


printf("Last Counter wynosi: %d\n",counter);


clock_t stop=clock();

stop-=start;
stop/=CLOCKS_PER_SEC/1000;



printf("Czas trwania programu: %d ms\n",stop);

for(i=0;i<=N;i++){

 fprintf(plik,"%d %.14f\n",i,xNowe[i]);
}
fclose(plik);
return 0;

}
```