// Autor: Daniel Szarek
//====================

Zawartość folderu z zadaniem:
main.cpp – Kod z rozwiązaniem zadania napisanym w języku C++
funkcje.h – Biblioteka z własnymi funkcjami wykorzystywanymi do rozwiązania zadania N6
opracowanie.pdf – Ten dokument z opracowaniem zadania N6
funkcje.cpp – Plik zawierający implementację funkcji podanych w własnej bibliotece funkcje.h
Makefile – Do uruchomienia programu main.cpp (Makefile oferuje możliwość uruchomienia programu za pomocą komendy 'make run', usunięcia plików po uruchomieniu programu komendą 'make clean' oraz możliwość utworzenia paczki .tar.gz z zawartością foldera z zadaniem za pomocą komendy 'make tar'
wynikiGS.txt – Plik tekstowy z wynikami metody Gausa-Seidla dla naszego problemu w poleceniu
wynikiJ.txt – Plik tekstowy z wynikami metody Jacobiego dla naszego problemu w poleceniu
wynikiR.txt – Plik tekstowy z wynikami metody Relaksacyjnej Richardsona dla naszego problemu w poleceniu
wynikiS.txt – Plik tekstowy z wynikami Successive OverRelaxation dla naszego problemu w poleceniu
folder: wykresy zawiera wykresy do graficznej wizualizacji wyników poszczególnych metod, wykres ogólny z wszystkimi czterema metodami naraz oraz wykresy porównujący wszystkie cztery wyniki metod na jednym wykresie
folder: skrypty gnuplot zawierający skrypty do programu gnuplot do utworzenia wyżej wspomnianych wykresów

Do zrealizowania zadania wykorzystałem GSL - GNU Scientific Library, w Makefile zawarłem komendy do uruchomienia zadania na swojej maszynie w systemie Ubuntu, w celu uruchomienia zadania na swoim sprzęcie należy w prawidłowy sposób zainstalować bibliotekę GSL oraz zmienić scieżkę INCLUDEPATH1 dla swojej maszyny.

W swoich plikach źródłowych zamieszczam liczne komentarze, które na bieżąco tłumaczą działanie oraz funkcjonalności mojego kodu. W tym dokumencie przedstawię omówienie metod wykorzystanych do rozwiązania zadania.

Treść Zadania:

N6 *Zadanie numeryczne*
Zaimplementować metodę:

- relaksacyjną (Richardsona)

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

- Jacobiego:

$$x^{(n+1)} = D^{-1}\left(b - Rx^{(n)}\right),$$

- Gausa-Seidla

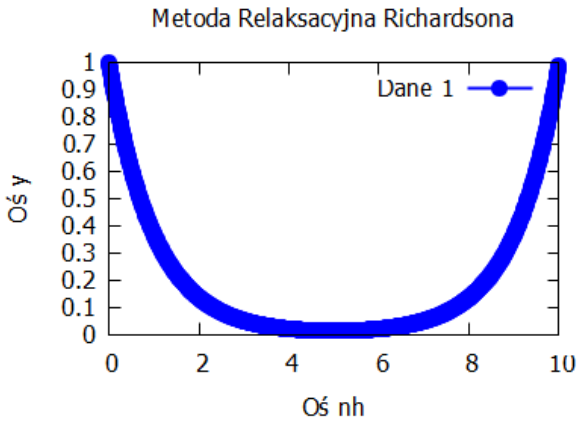
$$x^{(n+1)} = L^{-1}\left(b - Ux^{(n)}\right),$$

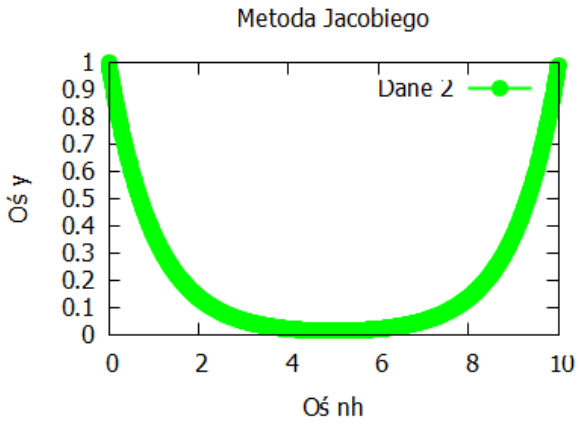
- Successive OverRelaxation

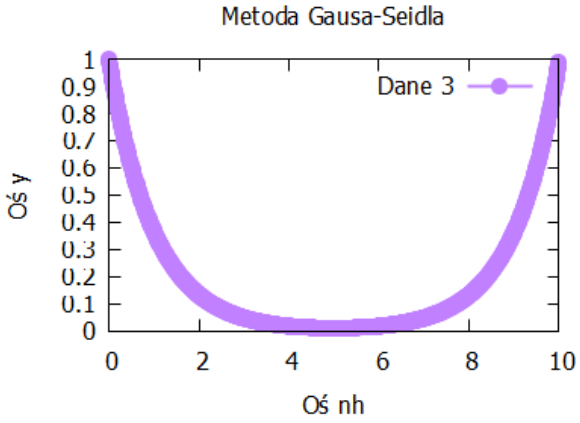
$$x_i^{(n+1)} = (1-\omega)x_i^{(n)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(n+1)} - \sum_{j>i} a_{ij}x_j^{(n)}\right), \quad i = 1, 2, \ldots, N.$$

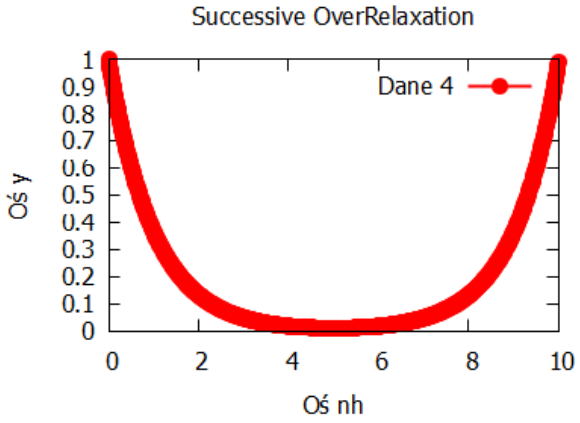
Znaleźć rozwiązania układu z zadania N3 z dokładnością 10^{-10}. Która metoda jest najszybsza? Proszę uwzględnić strukturę układu równań.

Rozwiązanie zadania:









Omówienie:

Zadanie zrealizowałem dla macierzy:

$$\begin{cases} y_0 & = 1 \\ -(D_2 y)_n + y_n & = 0, \quad n = 1 \ldots (N-1) \\ -\frac{y_{N-1} - 2y_N + y_0}{h^2} & = 0, \end{cases}$$

Zdecydowałem się na takie podejście ponieważ pierwotny układ w poleceniu zadania nie spełniał wymogów metod iteracyjnych. W celu otrzymania obecnej macierzy należało zmienić znak przy wyrażeniu D2yn oraz zmienić ostatni wiersz układu równań. Jeżeli nie zmienilibyśmy znaku przy D2yn to macierz nie byłaby dodatnio określona, wówczas metody iteracyjne byłyby rozbieżne. Ponad to modyfikacja ostatniego wiersza układu równań pozwoliła uzyskać zbieżność w metodzie relaksacyjnej Richardsona. Dla swojej implementacji przemnożyłem równania dla n od 1 do N przez h^2, owy zabieg miał na celu uniknięcie niepożądanego numerycznie dzielenia przez małą liczbę. Finalnie otrzymuje poniższy układ dla którego realizuje implementację tego zadania:

$$\begin{cases} y_0 = 1 \\ -(D_2 y)_n + y_n = 0 \quad n = 1...(N-1) \\ -y_{N-1} + 2y_N - y_0 = 0 \end{cases}$$

Poniżej znajduję się zrzut ekranu terminala na mojej maszynie wirtualnej, który wskazuje rozwiązanie zadania moją implementacją. Najszybsza okazała się Successive OverRelaxation, następnie Metoda Gausa-Seidla, Jacobiego, a na końcu metoda Relaksacyjna Richardsona.

```
vboxuser@Ubuntu:~/Desktop/MetodyNumeryczne/Zadanie6$ make run
g++ -Wall -std=c++11 -I/home/vboxuser/Desktop/gsl/include -c main.cpp -o main.o
ar rsv libMojeFunkcje.a funkcje.o
ar: creating libMojeFunkcje.a
a - funkcje.o
mkdir -p ./lib
mv libMojeFunkcje.a ./lib
g++ -o main.x main.o -Wall -std=c++11 -L./lib -lMojeFunkcje -lgsl -lgslcblas
./main.x
Metoda Relaksacyjna Richardsona:
Ilosc iteracji: 216167
Czas wykonywania metody: 6.632 sekund.

Metoda Jacobiego:
Ilosc iteracji: 215843
Czas wykonywania metody: 5.057 sekund.

Metoda Gausa-Seidla:
Ilosc iteracji: 114205
Czas wykonywania metody: 2.685 sekund.

Successive OverRelaxation:
Ilosc iteracji: 3142
Czas wykonywania metody: 0.109 sekund.
```

Rezultat dominacji metody Successive OverRelaxation nad resztą sposób wynika z tego, że w zadaniu dobrałem odpowiedni parametr relaksacji dla naszego układu, który pozwala kontrolować tempo zbieżności. Następnie metoda SOR bierzę pod uwagę informację z obecnej jak i poprzedniej iteracji. Ponad to jest wydajna dla macierzy zdominowanych na diagonali, co pokrywa się z właściwościami naszego układu. Metoda relaksacyjna Richardsona również do działania wykorzystuje parametr relaksacji, niemniej jednak w odróżnieniu do poprzedniej metody nie uwzględnia informacji z obecnej iteracji oraz jest mniej elastyczna niż poprzedniczka, natomiast w efektywności działania blisko jej do metody Jacobiego. Warto pokreślić, że jeżeli parametr relaksacji zostanie dobrany w sposób nieprawidłowy, sprawi to, że metody będą wolniej uzyskiwały zbieżność, a nawet mogą doprowadzić do rozbieżności metody i uzyskania złego rozwiązania.

Poniżej przedstawiam wzory, dla których wykonałem implementację metod w swoim programie:

Metoda Richardsona z parametrem $\tau \in \mathbb{R}$ jest określona wzorem

$$x_{k+1} = x_k + \tau(b - Ax_k).$$

5.3.1. Metoda Jacobiego

Biorąc w (5.12) $M = D$, gdzie D jest macierzą diagonalną składającą się z wyrazów stojących na głównej przekątnej macierzy A (zob. (5.14)), otrzymujemy (o ile na przekątnej macierzy A nie mamy zera) metodę iteracyjną

$$x_{k+1} = D^{-1}(b - (L+U)x_k),$$

zwaną *metodą Jacobiego*.

Rozpisując ją po współrzędnych, dostajemy układ rozszczepionych równań (numer iteracji wyjątkowo zaznaczamy w postaci górnego indeksu):

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right),$$

co znaczy dokładnie tyle, że w i-tym równaniu wyjściowego układu przyjmujemy za współrzędne x wartości z poprzedniej iteracji i na tej podstawie wyznaczamy wartość x_i.

Widzimy więc, że metoda rzeczywiście jest banalna w implementacji, a dodatkowo jest w pełni równoległa: każdą współrzędną nowego przybliżenia możemy wyznaczyć niezależnie od pozostałych.

5.3.2. Metoda Gaussa–Seidela

Heurystyka tej metody opiera się na zmodyfikowaniu metody Jacobiego tak, by w każdym momencie iteracji korzystać z najbardziej „aktualnych" współrzędnych przybliżenia rozwiązania x. Rzeczywiście, przecież wykonując jeden krok metody Jacobiego, czyli rozwiązując kolejno równania skalarne względem $x_i^{(k+1)}$ dla $i = 1, \ldots, N$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right),$$

nietrudno zauważyć, że w części sumy, dla $j < i$, moglibyśmy odwoływać się — zamiast do „starych" $x_j^{(k)}$ — do „dokładniejszych", świeżo wyznaczonych, wartości $x_j^{(k+1)}$, tzn. ostatecznie wyznaczać

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right).$$

W języku rozkładu macierzy $A = M - Z$ i iteracji $x_{k+1} = M^{-1}(Zx_k + b)$ mielibyśmy więc $M = L + D$ (dolny trójkąt macierzy A z diagonalą) oraz $Z = -U$ (ściśle górny trójkąt A) i konsekwentnie zapis macierzowy iteracji

$$x_{k+1} = (L+D)^{-1}(b - Ux_k).$$

W metodzie SOR skorzystałem z dodatkowego przekształcenia zwiększającego wydajność:

Note

$(1-\omega)\phi_i + \frac{\omega}{a_{ii}}(b_i - \sigma)$ can also be written $\phi_i + \omega\left(\frac{b_i - \sigma}{a_{ii}} - \phi_i\right)$, thus saving one multiplication in each iteration of the outer *for*-loop.

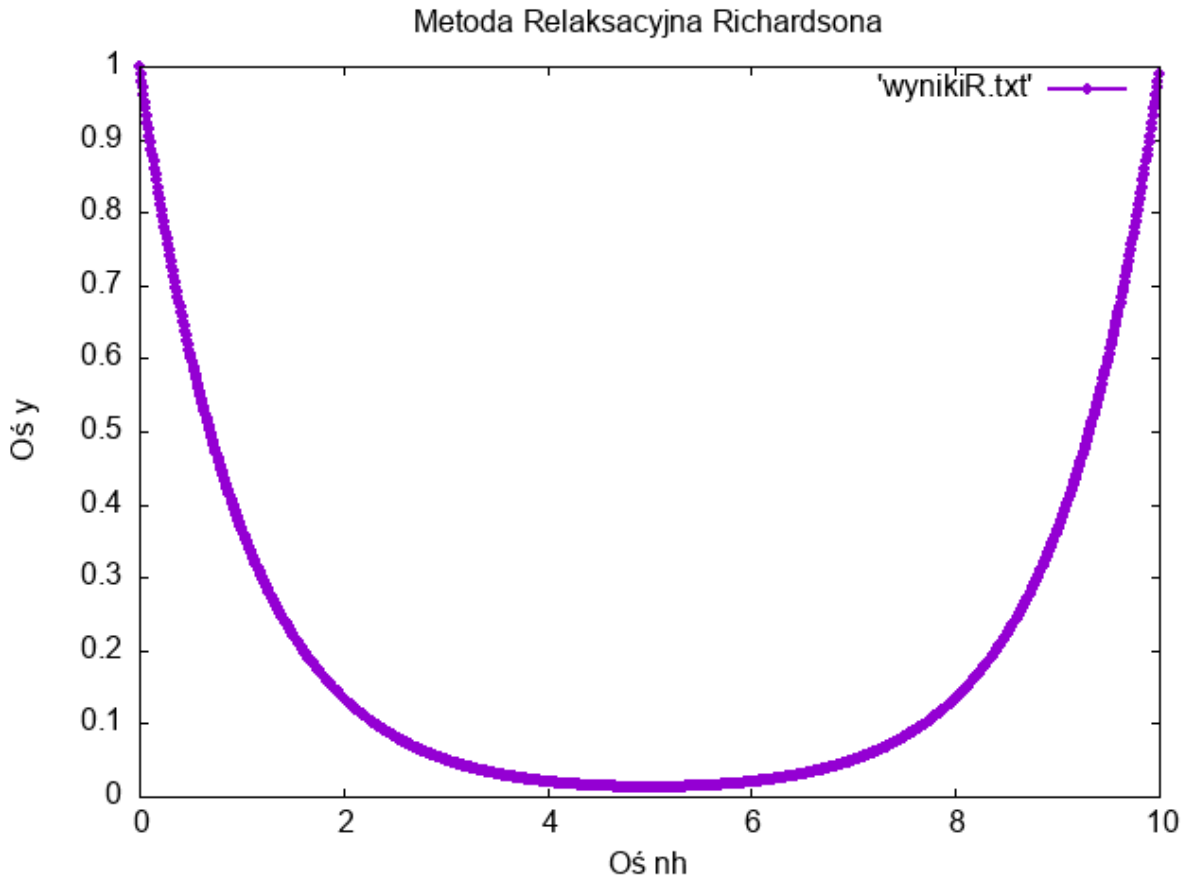
5.3.3. Metoda SOR

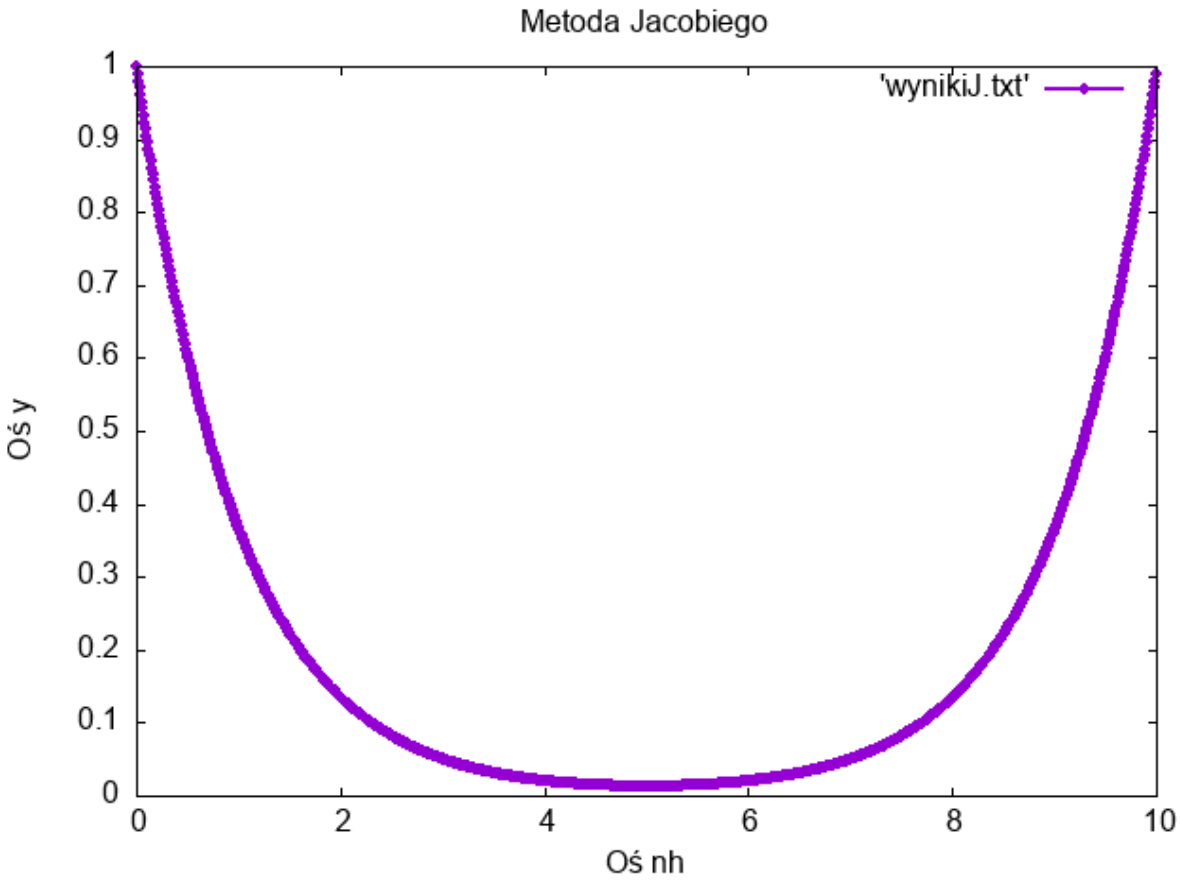
Zbieżność metody Gaussa–Seidela można przyspieszyć, wprowadzając parametr relaksacji ω i kolejne współrzędne nowego przybliżenia x_{k+1} wyznaczać, kombinując ze sobą poprzednie przybliżenie $x_i^{(k)}$ oraz współrzędną nowego przybliżenia $\tilde{x}_i^{k+1}$, uzyskanego metodą Gaussa–Seidela:

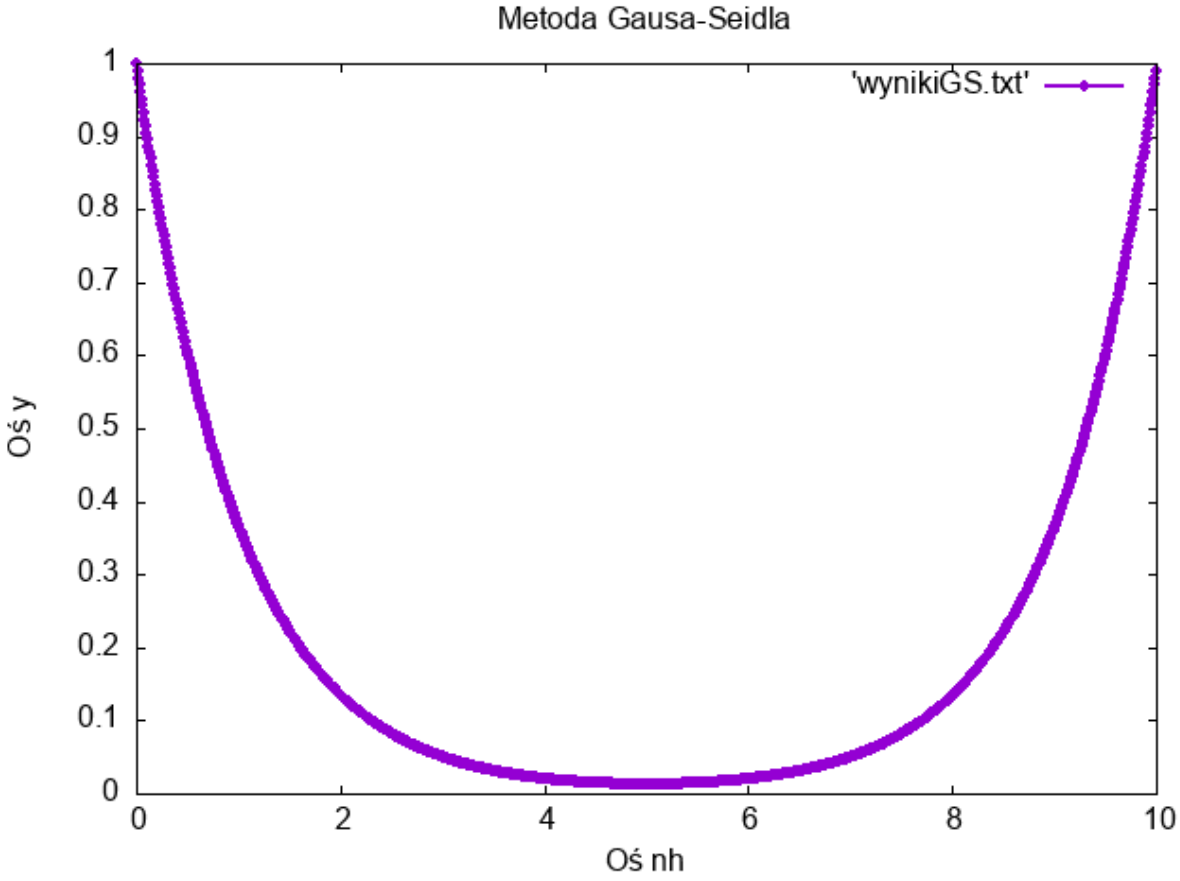
$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \omega\tilde{x}_i^{k+1}.$$

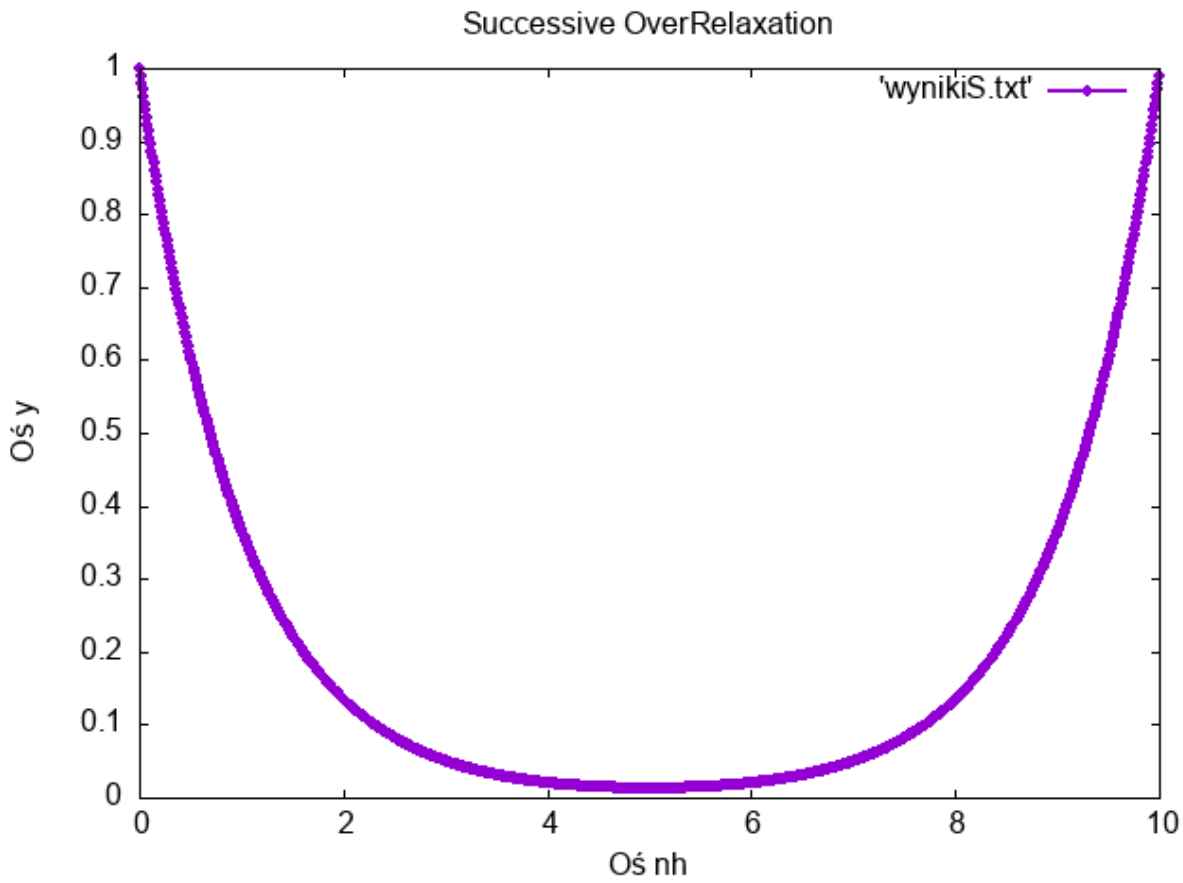
Jako kryterium zakończenia pętli while(true) użyłem warunku, który sprawdza, czy różnica między wynikami kolejnych iteracji, jest mniejsza niż pewna wartość epsilon podana w poleceniu N3. Innymi słowy, po każdej iteracji obliczam różnicę między wynikiem z poprzedniej iteracji a aktualną iteracją, a następnie obliczam normę tej różnicy. Jeśli ta norma przekracza ustaloną wartość 10e-10, to kontynuuję kolejne iteracje. W przeciwnym razie uznaję, że osiągnięto wystarczającą dokładność i przerywam proces iteracyjny.

Poniżej wklejam wyniki, które otrzymałem dzięki implementacji metoda na podstawie powyższych metod w pliku funkcje.cpp



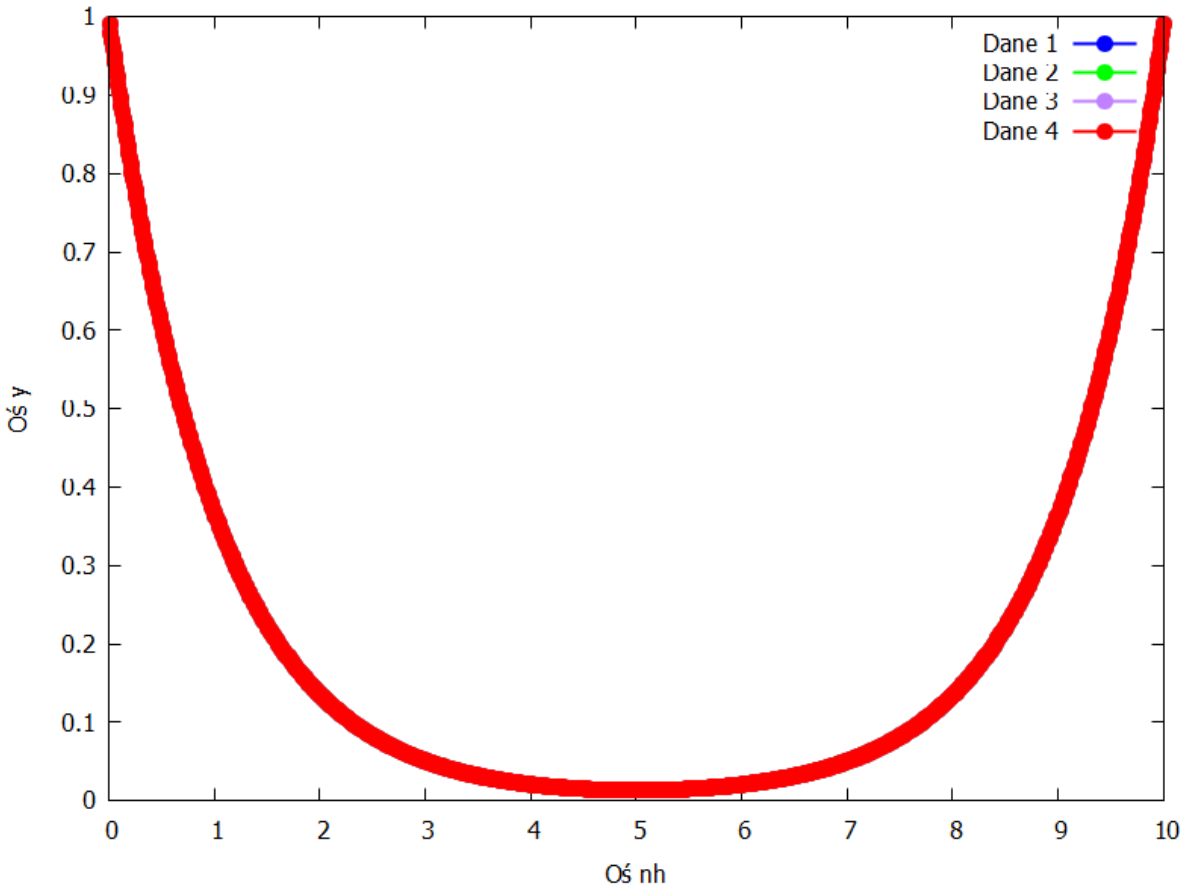

Metoda Jacobiego
'wynikiJ.txt'
Oś y
Oś nh
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10


Metoda Gausa-Seidla
'wynikiGS.txt'
Oś y
Oś nh
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10



Jak widać na pierwszy rzut oka wykresy wyglądają identycznie, niemniej jednak różnią się one zawartością. Różnice pomiędzy poszczególnymi metodami łatwiej zaobserwować bezpośrednio w plikach z rozwiązaniami są one marginalne i występują dopiero od piątego miejsca po przecinku.

Poniżej dla formalności wklejam wykres z wszystkimi wynikami dla czterech metod na jednym wykresie.


Dane 1
Dane 2
Dane 3
Dane 4
Oś y
Oś nh
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
1
2
3
4
5
6
7
8
9
10