Metody Numeryczne [MNUM]
Projekt 3.40

Tomasz Pawlak, Informatyka, 304104
Prowadzący: dr hab. inż. Andrzej Karbowski

Zad 1.

Ruch punktu na płaszczyźnie (x_1, x_2) jest opisany równaniami:

$$\frac{dx_1}{dt} = x_2 + x_1(0.3 - x_1^2 - x_2^2)$$

$$\frac{dx_2}{dt} = -x_1 + x_2(0.3 - x_1^2 - x_2^2)$$

Należy obliczyć przebieg trajektorii ruchu tego punktu w przedziale $t \in [0, 20]$ dla warunków początkowych:

$$x_1(0) = 0.001$$
$$x_2(0) = -0.02$$

Rozwiązanie znaleźć korzystając z zaimplementowanego solwera **metody Rungego-Kutty czwartego rzędu (RK4) przy zmiennym kroku z szacowaniem błędu metodą zdwajania kroku.**

Opisy algorytmów

Metoda Rungego-Kutty RK4 ze zmiennym krokiem omija problem wykonywania małych kroków w miejscach, gdzie nie jest to konieczne, przez co zmniejsza błąd spowodowany dokładnością reprezentacji liczb.

Samą metodę możemy formułować dla dowolnej rzędu, wybieramy w zadaniu czwarty, gdyż jest to kompromis pomiędzy najwyższą dokładnością – wyrażaną jako rząd metody *p* oraz nakład obliczeń na jedną iterację i powiązany z tym błąd zaokrągleń. Główny wzór definicji metody prezentuje się następująco:

$y_{n+1} = y_n + h\sum_{i=1}^{m} w_i k_i$, gdzie

$$k_1 = f(x_n, y_n)$$

$$k_i = f\left(x_n + c_i h, y_n + h\sum_{j=1}^{i-1} a_{ij} k_j\right), \quad i = 2,3,\dots,m$$

$$\sum_{j=1}^{i-1} a_{ij} = c_i, \quad i = 2,3,\dots,n$$

Przekładając wzory z rozdziału 7.1.1 podręcznika na nazwy parametrów użytych w zadaniu otrzymujemy wzór iteracyjnego postępowania znajdywania kolejnych iteracji dla metody 4-rzędu oraz układu dwóch funkcji równań różniczkowych z zadania:

$$(x_1)_{n+1} = (x_1)_n + \frac{1}{6}h_n(k_{1,1} + 2k_{2,1} + 2k_{3,1} + k_{4,1})$$
$$(x_2)_{n+1} = (x_2)_n + \frac{1}{6}h_n(k_{1,2} + 2k_{2,2} + 2k_{3,2} + k_{4,2})$$

gdzie

$$k_{1,1} = dx_1((x_1)_n, (x_2)_n)$$
$$k_{1,2} = dx_2((x_1)_n, (x_2)_n)$$
$$k_{2,1} = dx_1((x_1)_n + \frac{1}{2}h_n k_{1,1}, (x_2)_n + \frac{1}{2}h_n k_{1,2})$$
$$k_{2,2} = dx_2((x_1)_n + \frac{1}{2}h_n k_{1,1}, (x_2)_n + \frac{1}{2}h_n k_{1,2})$$
$$k_{3,1} = dx_1((x_1)_n + \frac{1}{2}h_n k_{2,1}, (x_2)_n + \frac{1}{2}h_n k_{2,2})$$
$$k_{3,2} = dx_2((x_1)_n + \frac{1}{2}h_n k_{2,1}, (x_2)_n + \frac{1}{2}h_n k_{2,2})$$
$$k_{4,1} = dx_1((x_1)_n + h_n k_{3,1}, (x_2)_n + h_n k_{3,2})$$
$$k_{4,2} = dx_2((x_1)_n + h_n k_{3,1}, (x_2)_n + h_n k_{3,2})$$

Wzory te mają zastosowanie nie tylko do obliczenia kroku całkowitego, ale również są wykorzystane w metodzie zdwajania półkroków.

Metoda zdwajania kroków wymaga również policzenia błędu pomiędzy przejściem między półkrokowym algorytmem oraz pełnym równaniem kroku. Aby tego dokonać należy oszacować błąd dwóch półkroków o długości $\frac{h}{2}$. Ogólny wzór na część główną błędu metody rzędu 4 dla kroku h jest postaci:

$$\delta_n(h) = \frac{r_n^5(0)}{5!}$$

Założenie dokładności obliczeń zadane przez użytkownika zakłada że dla kroku *h* błąd jest w proporcji:

$$\alpha^5|\delta_n(h)| = \varepsilon$$

Stąd możemy policzyć współczynnik α jako:

$$\alpha = (\frac{\varepsilon}{|\delta_n(h)|})^{1/5}$$

Parametry dokładności dane przez użytkownika określamy natomiast jako estymatę błędu:

$$\varepsilon = |y_n|\varepsilon_n + \varepsilon_b$$

A w algorytmie wprowadzone zostały stałe określające dane dokładności jako:

ε_n - dokładność względna
ε_b - dokładność bezwzględna

Prezentowany w poniższych punktach wykresy estymaty zakładają wybór mniejszego modułu błędu, który analogicznie został wypracowany dla parametru alpha.

Dla układu 2 równań przyjmuje się współczynnik wyliczony dla najbardziej krytycznego równania, zatem wybieramy nowy współczynnik α po każdej iteracji jako:

$$\alpha = \min_{1 \le i \le 2} \left(\frac{\left|(y_i)_n^{(2)}\right| \varepsilon_n + \varepsilon_b}{\left| \frac{(y_i)_n^{(2)} - (y_i)_n^{(1)}}{15} \right|} \right)^{1/5}$$

Realizacja algorytmu zakłada również wykorzystanie algorytmu z Rysunku 7.6/174 podręcznika do przedmiotu przebudowanego, lecz spełniającego warunki schematu blokowego.

Implementacja

W pliku ode.m zawarta została implementacja algorytmu ode45 celem rozwiązania równania. Metoda, aby obsłużyć układ równań różnicowych wymaga przekazania zewnętrznego uchwytu do metody, zatem jako parametry wejścia traktujemy tylko x_1, x_2 oraz długości obserwowanego przedziału *b*.

Reszta znaczących parametrów została wprowadzona jako stałe lokalne metody (Tolerancja Relatywna `RelTol` i Absolutna `AbsTol` są znane jest z poprzednich wzorów jako kolejno tolerancja względna i bezwzględna i domyślnie ustawiona na `1e-4` oraz `1e-8`. Wpływ tolerancji powoduje poprawianie wyników i wpływa niekorzystnie na czas operacji algorytmu. Samo wywołanie funkcji jest intuicyjne i kod nie wymaga dalszych objaśnień:

```
% Przedział czasu
tspan=[0 b];
% Punkty startowe
x0=[xs1;xs2];
% Zadana tolerancja
opts = odeset('RelTol',1e-4,'AbsTol',1e-8);
% Generacja celu zadania
[t, x]=ode45(@zad1_func, tspan, x0, opts);
```

Metoda zad1_func zawiera generację naszego układu równań.

Plik `rk4z.m` zawiera implementacje algorytmu RK4 „Z” zmiennym krokiem z szacowaniem błędu metodą zdwajania kroku. Parametrami wejścia do algorytmu są dwa równania różniczkowe dane jako funkcje, współrzędne startowe układu x_1 oraz x_2, a także parametr końca przedziału obserwacji i pierwotny krok h_0 . W solwerze zakładam, że solwer zawsze startuje z chwilą t=0.

Lokalne parametry solwera wylistowane są poniżej:

```
h=h0;
hh=h;
s = 0.3;
beta_ = 5;
E_n = 1e-4;
E_b = 1e-8;
hmin = 1e-12;
```

h – krok algorytmu zmieniany zgodnie ze schematem
hh – wektor kroków algorytmu
s – współczynnik bezpieczeństwa używany przy obliczaniu *h_new*, czyli proponowanych wartości kroku na kolejnych iteracjach zgodnie ze wzorem:

$$h_{n+1} = s\alpha h_n$$

*beta_* - heurystyczne ograniczenie maksymalnego wzrostu długości kroku w jednej iteracji
h_{min} – minimalny krok algorytmu, którego wpływ zakłada poprawianie wyników operacji kalkulacji następnych punktów.

Równanie kroku całkowitego zakłada operacje podane w algorytmie z 1p. niniejszego sprawozdania:

```
% Równania kroku całkowitego
k_1_1=dx1(x1,x2);
k_1_2=dx2(x1,x2);
k_2_1=dx1((x1+0.5*h*k_1_1),(x2+0.5*h*k_1_2));
k_2_2=dx2((x1+0.5*h*k_1_1),(x2+0.5*h*k_1_2));
k_3_1=dx1((x1+0.5*h*k_2_1),(x2+0.5*h*k_2_2));
k_3_2=dx2((x1+0.5*h*k_2_1),(x2+0.5*h*k_2_2));
k_4_1=dx1((x1+h*k_3_1),(x2+h*k_3_2));
k_4_2=dx2((x1+h*k_3_1),(x2+h*k_3_2));

x1=x1+(1/6)*h*(k_1_1+2*k_2_1+2*k_3_1+k_4_1);
x2=x2+(1/6)*h*(k_1_2+2*k_2_2+2*k_3_2+k_4_2);
X1=[X1; x1];
X2=[X2; x2];
```

Nowe rezultaty zostają również dopisane do wektorów X1 oraz X2, celem prezentacji danych.

Równania półkroku zakładają wykonanie dwóch półkroków o długości $\frac{h}{2}$ po sobie, zgodnie z algorytmem:

```
if(mod(n,2) == 1)
    % Równania półkroku 1
    k_1_1d=dx1(x1,x2);
    k_1_2d=dx2(x1,x2);
    k_2_1d=dx1((x1+0.25*h*k_1_1d),(x2+0.25*h*k_1_2d));
    k_2_2d=dx2((x1+0.25*h*k_1_1d),(x2+0.25*h*k_1_2d));
    k_3_1d=dx1((x1+0.25*h*k_2_1d),(x2+0.25*h*k_2_2d));
    k_3_2d=dx2((x1+0.25*h*k_2_1d),(x2+0.25*h*k_2_2d));
    k_4_1d=dx1((x1+0.5*h*k_3_1d), (x2+0.5*h*k_3_2d));
    k_4_2d=dx2((x1+0.5*h*k_3_1d), (x2+0.5*h*k_3_2d));

    x1d=x1+(1/12)*h*(k_1_1d+2*k_2_1d+2*k_3_1d+k_4_1d);
    x2d=x2+(1/12)*h*(k_1_2d+2*k_2_2d+2*k_3_2d+k_4_2d);

    % Równania półkroku 2
    x1h = x1 + 0.5*h;
    x2h = x2 + 0.5*h;
    k_1_1h=dx1(x1h,x2h);
    k_1_2h=dx2(x1h,x2h);
    k_2_1h=dx1((x1h+0.25*h*k_1_1h),(x2h+0.25*h*k_1_2h));
    k_2_2h=dx2((x1h+0.25*h*k_1_1h),(x2h+0.25*h*k_1_2h));
    k_3_1h=dx1((x1h+0.25*h*k_2_1h),(x2h+0.25*h*k_2_2h));
    k_3_2h=dx2((x1h+0.25*h*k_2_1h),(x2h+0.25*h*k_2_2h));
    k_4_1h=dx1((x1h+0.5*h*k_3_1h), (x2h+0.5*h*k_3_2h));
    k_4_2h=dx2((x1h+0.5*h*k_3_1h), (x2h+0.5*h*k_3_2h));

    x1h=x1d+(1/12)*h*(k_1_1h+2*k_2_1h+2*k_3_1h+k_4_1h);
    x2h=x2d+(1/12)*h*(k_1_2h+2*k_2_2h+2*k_3_2h+k_4_2h);
end
```

Na podstawie porównania obu wyznaczonych punktów zostają wyznaczane błędy oraz estymowany zostaje następny punkt jako h_new, w przypadku, gdy `sa>=1` możliwe jest że punkt nie jest najmniejszy z zakresu h_new, `beta_*h, b-t`. Ograniczenia wynikają z tego, aby różnice nie narastały gwałtownie lecz stopniowo w przypadku prób poprawiania wyniku:

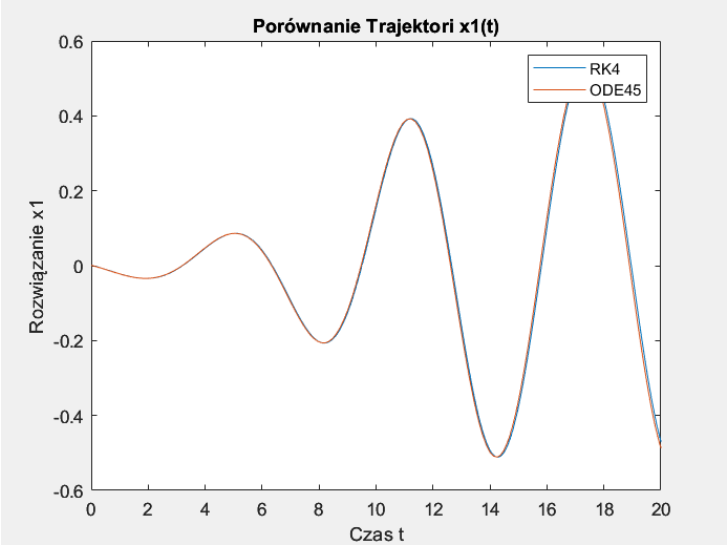
```
if(mod(n,2) == 0)
    T=[T; t];

    delta1=(1/15)*abs(x1h-x1);
    delta2=(1/15)*abs(x2h-x2);

    epse1=(abs(x1)*E_n + E_b);
    epse2=(abs(x2)*E_n + E_b);

    err=[err; min(epse1, epse2)];


    alfa=min(epse1/delta1, epse2/delta2);
    alfa=alfa^(1/5);


    % Ustalenie nowej wartości kroku dla kolejnej pętli
    h_new = s*alfa*h;
```

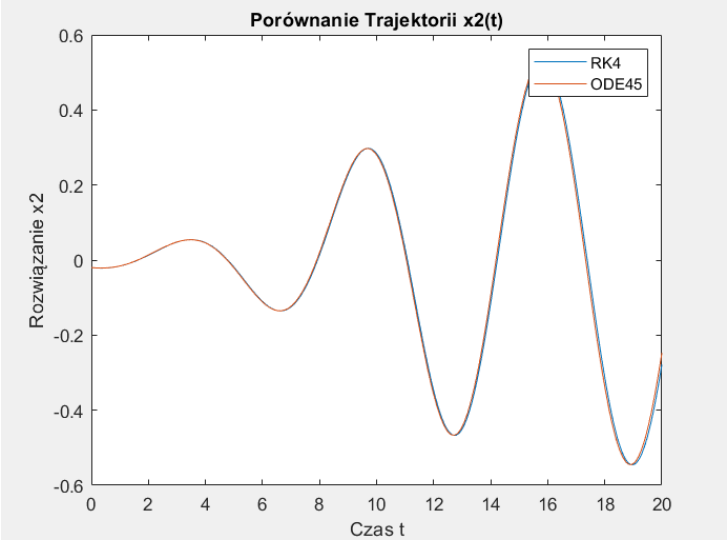
Poniżej znajduje się rozwiązanie schematu z Rys. 7.6 podręcznika.

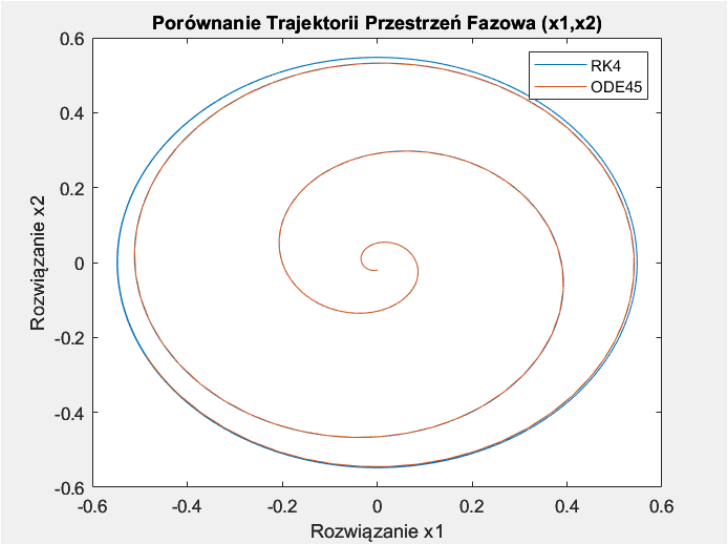
```
if s*alfa >= 1
    % Następny Punkt
    h=min([h_new, beta_*h, b-t]);
    t=t+h;

elseif h_new >= hmin
    % Nowa wartość kroku
    h=h_new;
    n=n-2;
else
    % Przypadek niemożliwy do rozwiązania - za duże E
    error('Niemożliwe rozwiązanie z zadaną dokładnością'
end
hh=[hh; h];
```

Wykresy porównawcze

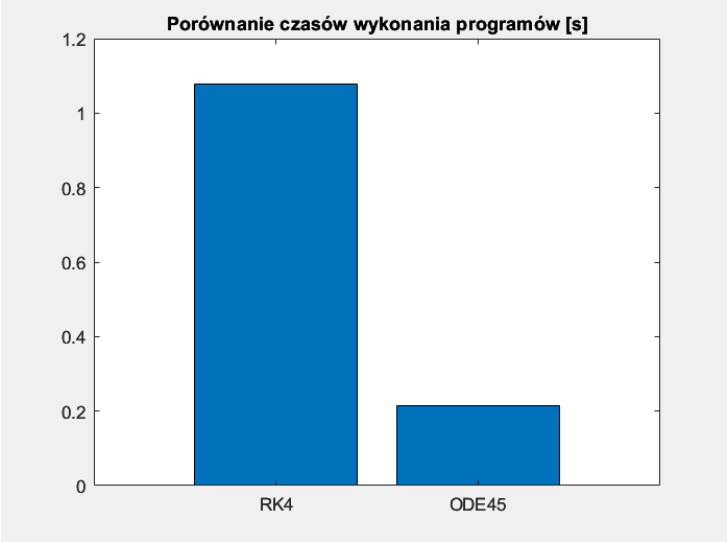
W pliku `comparison.m` zawarte są wywołania obu algorytmów, porównanie czasów wywołania algorytmów oraz wykresy porównawcze trajektorii punktów:





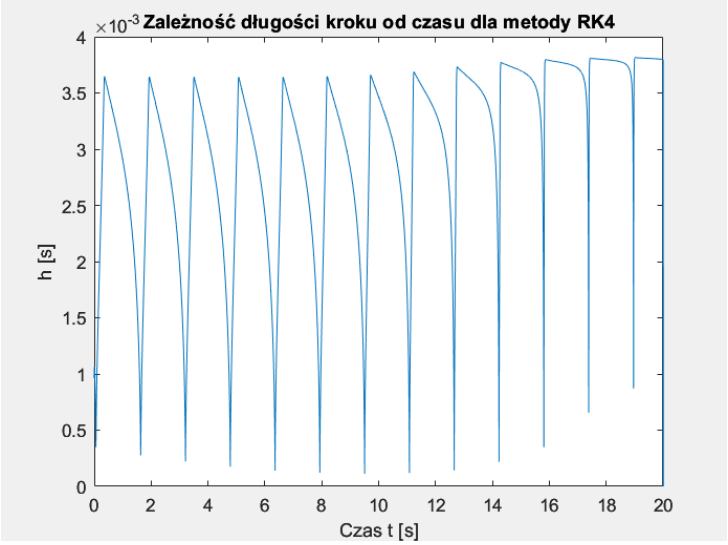


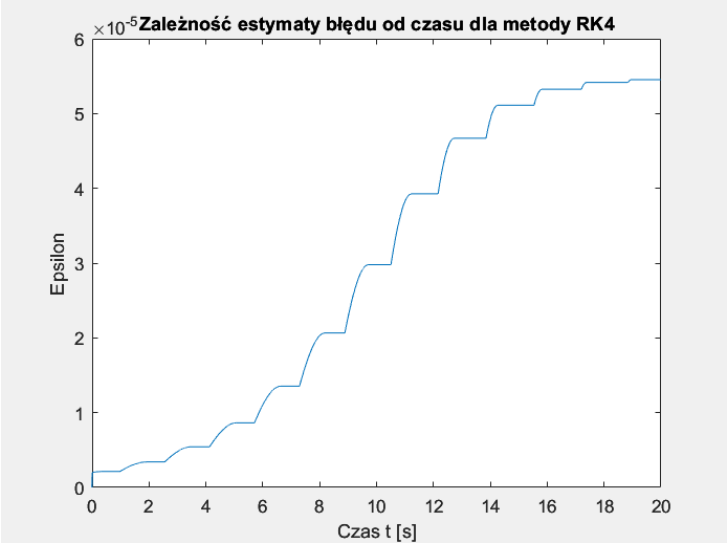
Wykresy są bardzo zbliżone do siebie, a ich trajektorie względnie się pokrywają.



Algorytm RK4 jest wolniejszy około 5-krotnie od implementacji ODE45. Jednym z przyczyn tego zjawiska jest na pewno addytywne rozszerzanie listy w moim algorytmie zamiast stałej alokacji pamięci. Aby to przyspieszyć należałoby opracować algorytm oszacowania ilości wyników bazujący na pochodnej oraz przewidywaniu długości maksymalnej kroku oraz alokujący obszary pamięci na potrzeby zapisywania wyników, przy czym w momencie przekroczenia zasobów alokowalibyśmy pamięć od nowa, tak aby nie wystąpiło *buffer overflow.*

Wykresy zależności dla metody RK4





Jak widać metoda zakłada schodkowe zwiększanie długości kroków, aż do momentu przekroczenia błędu. Wynika to z wyborów następnego kroku regularnie zmienianego w algorytmie.

Zależność estymaty błędu zgadza się z moimi założeniami – błąd narasta ze wzrostem iteracji.

Krok początkowy h_0, minimalny h_{min}, Wartość dokładności względnej i bezwzględnej.

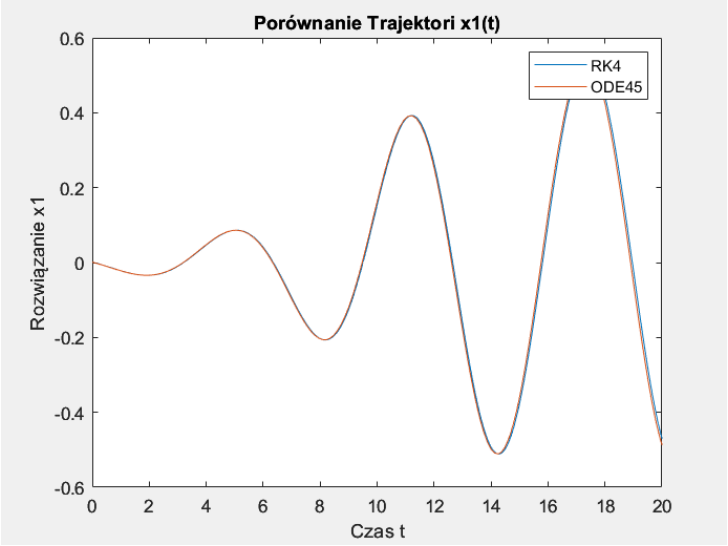
Krok początkowy definiuje stopień przeskoku między punktami, lecz nie musi być nawet użyty jeśli w drodze postępowania algorytm uzna krok za zbyt niedokładny – chcąc uzyskać płynniejszy wykres w przestrzeni fazowej musimy zastosować mniejszy krok zwiększając przy tym znacznie nakład mocy obliczeniowej procesora na generacje licznych iteracji. Krok początkowy w wyniku nie spełnienia zakładanej dokładności zostanie powtórzony, dlatego jego wybór nie wpływa skrajnie na dokładność algorytmu w przypadku metody

zmiennokrokowej. W przypadku implementacji metody ze stałym krokiem, wybór kroku początkowego ze stosunkowo dużą wartością powodowałby wrażenie nieciągłości trajektorii ruchu punktu oraz co za tym idzie – znaczącą niedokładność.

Krok minimalny definiuje natomiast dokładność wyników – program może zawiesić się – czas wzrasta z chwilowego do kilkunastosekundowego, gdy przy próbie zwiększania dokładności zmniejszamy nazbyt krok. Krok h_{min} posiada korelacje ze współczynnikami dokładności względnej i bezwzględnej.

Zależność stałej bezpieczeństwa s na oscylacje wyniku

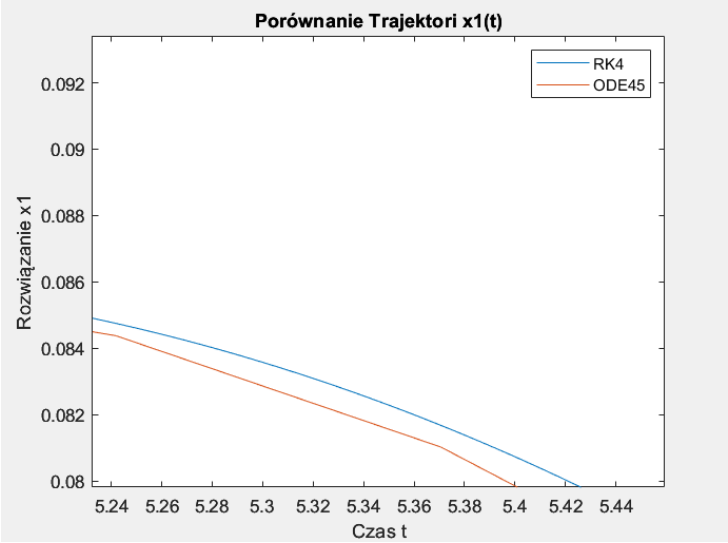
Poważnym błędem, nieakceptowalnym we wstępnym rozwiązaniu było występowanie zwiększania odległości pomiędzy punktami na wykresach porównawczych $x_i(t)$ dla solwerów ode45 i rk4z. Wynik, który zawarłem powyżej zdaje się na pierwszy rzut oka nie odstawać znacznie od modelowego rozwiązania. W rozwiązaniu stała *s* została ustalona arbitralnie przeze mnie na 0.3, co jest wyznaczonym kompromisem pomiędzy dokładnością rysunku trajektorii oraz czasem wykonania operacji – jeśli s ustawimy na wartość np. 0.1 liczba iteracji rośnie znacznie, przez co czas operacyjny wydłuża się wykładniczo.

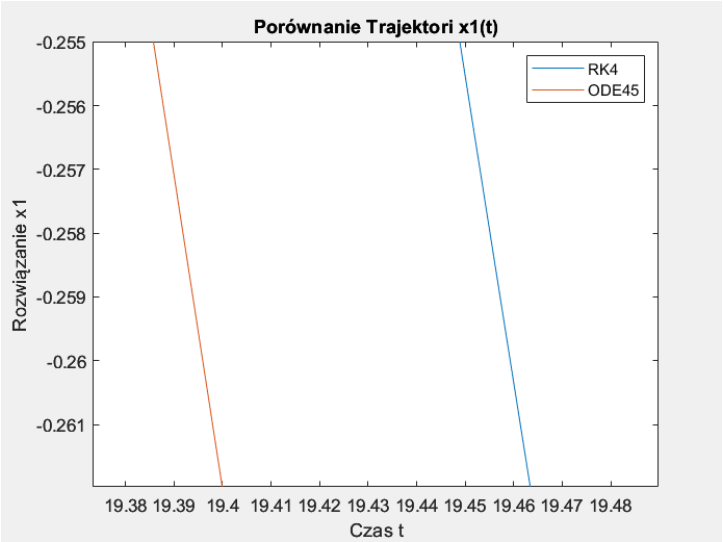


Ten wykres możemy dowolnie przeskalowywać i zawarłem go w pliku comp_x1_t_s_0_3.fig.

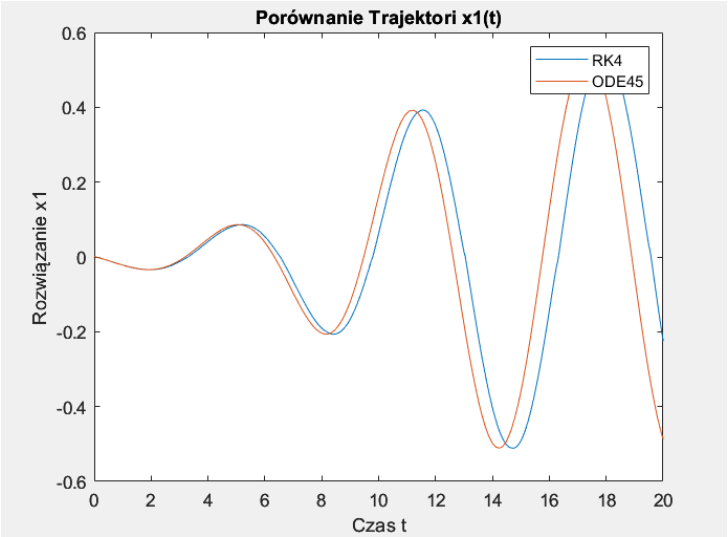
Analizując go również możemy dojść do wniosku, że aberracje dalej narastają.

Dla wartości czasu w okolicy *t=5.4s* aberracja wynosi $\Delta t = 0.025s$. Jednak ta wartość dalej rośnie po większej liczbie iteracji!!! Na przykład dla czasu w okolicy *t=19.4s* rozstęp rośnie do wartości $\Delta t = 0.063s$. Jest to niezauważalne przez użytkownika uruchamiającego plik `comparison.m` Poniżej dwa wykresy dla wspomnianych wartości w przybliżeniu:





Dla porównania zastosowanie wartości s domyślnej, jak w podręczniku *s=0.9* powoduje odchylenie pokazane poniżej:



Uważam, że geneza tego błędu może być powiązana z wzorem obliczającym współczynnik *alfa*. Moim zdaniem *współczynnik bezpieczeństwa* pełni funkcje szacowania rozmiaru błędu i na tej podstawie minimalizuje następny krok, aby nie dostąpiło do swego rodzaju „reakcji łańcuchowej” w generacji błędów. Istotne jest tu również podkreślenie, że *s* jest reakcją na błąd obliczany w funkcjach:

```
% Obliczanie błędów na podstawie kroku
delta1=(1/15)*abs(x1h-x1);
delta2=(1/15)*abs(x2h-x2);
epse1=abs(x1h)*E_n + E_b;
epse2=abs(x2h)*E_n + E_b;


alfa=min(epse1/delta1, epse2/delta2);
alfa=alfa^(1/5);
```