Matlab/Octave - ODE

Jednym z trudnych zadań, jakie może napotkać inżynier, jest rozwiązywanie równań różniczkowych zwyczajnych. Równania takie to powiązanie wielkości aktualnie występujących z różnymi przyrostami (wyrażonymi przez pochodne). Do ich rozwiązywania analityczne podejście jest nieskuteczne niemal w każdym praktycznym przypadku i dlatego stosuje się metody przybliżone. Z programem takim jak Matlab nasze zadanie sprowadza się tylko do podania równań i warunków rozwiązywania – resztę pracy wykona za nas komputer.

Prosty przypadek – oscylator harmoniczny

Klasycznym problemem, i to nie tylko mechaniki, są drgania harmoniczne tłumione pod wpływem zewnętrznej siły wymuszającej. Mamy ruch opisany równaniem:

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0^2 x = (\omega^2 - \omega_0^2)A\sin\omega t$$

Jak użyć programu Matlab dla zbadania co się dzieje, jeżeli $\zeta = 0.1$, $\omega_0 = 500$ Hz, $\omega = 2000$ Hz, $A = 5$ cm, początkowe położenie $x(t=0) = 15$ cm, a prędkość dx/dt początkowo wynosi zero?

Zaczynamy od rozbicia równania na równania stopnia pierwszego, bo Matlab będzie rozwiązywać układy równań pierwszego stopnia:

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -2\zeta\omega_0 v - \omega_0^2 x - (\omega^2 - \omega_0^2)A\sin\omega t \end{cases}$$

W pliku *oscylator.m* zapisujemy[1] funkcję *oscylator*, która musi mieć dwa parametry: pierwszy reprezentuje zmienną niezależną, drugi jest wektorem ze zmiennymi zależnymi (czyli w konkretnym przypadku są to położenie i prędkość).

```
function dqdt = oscylator(t,q)

  omega0 = 1000.0;
  omega  = 2000.0;
  zeta   =    0.1;
  A      =    0.05;

  dqdt = [ q(2)
          -2*zeta*omega0*q(2) - omega0^2*q(1)-(omega^2- omega0^2)*A*sin(omega*t) ];

end
```

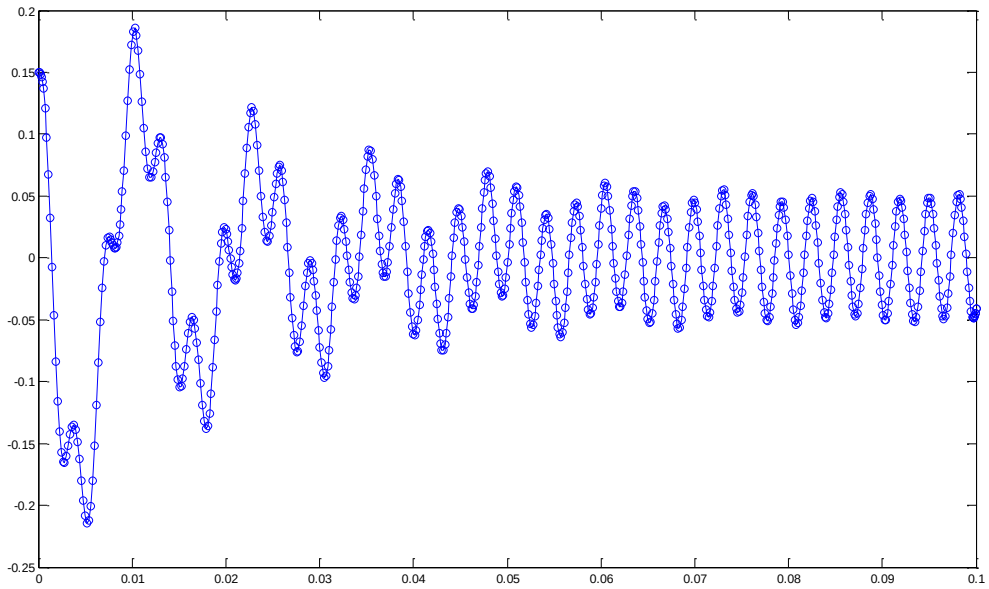
Aby wykonać obliczenia wydajemy Matlabowi polecenie:

```
ode45(@oscylator,[0 0.10],[0.15 0],odeset('OutputSel',1));
```

Funkcja *ode45* zostaje wywołana z czterema parametrami – uchwytem do funkcji *oscylator*, zakresem zmiennej niezależnej dla jakiego należy wykonać obliczenia, wektorem wartości początkowych i

[1] Nazwa funkcji *oscylator* musi być zgodna z nazwą pliku *oscylator.m*. Identyfikator *dqdt* oznaczający zwracany wektor, choć jest bezpośrednio po słowie kluczowym *function*, nie jest nazwą funkcji.

opcjami utworzonymi przez *odeset* (jako *'OutputSel'* wybieramy zmienną 1, czyli położenie). Matlab wykona obliczenia rysując wykres[2] obrazujący rozwiązanie.



Rysunek 1. Drgania tłumione zanikają zastępowane przez drgania wymuszone

Po co tyle funkcji *ode*?

Matlab oferuje łatwą zmianę algorytmu rozwiązywania równań – wystarczy zamiast np. *ode45* użyć funkcji *ode113* lub *ode23* – aktualny wykaz znajdziemy w dokumentacji on-line do programu Matlab, np. pisząc polecenie *doc ode45*. Po co jednak tyle funkcji *ode*[3] – czy nie wystarczyłaby jedna, *ode45*?

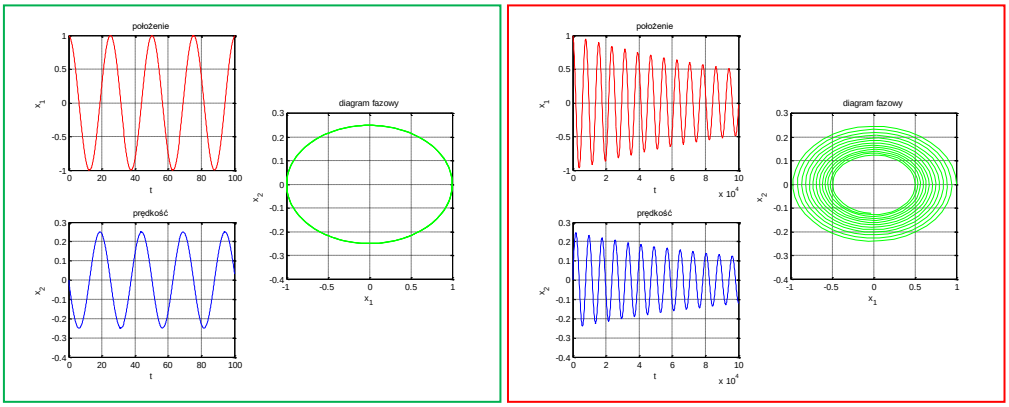
Na kolejnym rysunku poniżej widzimy zły efekt działania *ode45* dla bardzo długich czasów: fałszywy okres i zanik amplitudy. Fikcyjny okres powstał dzięki aliasingowi – jak to się stało wyjaśnia kolejny rysunek. Zanik amplitudy powstał wskutek kumulacji błędów numerycznych. Wybór innego schematu obliczeń (np. *ode23t*) pozwala na ocenę w jakim stopniu otrzymane wyniki są poprawne. Szczególnie uważać należy na takie równania, które określa się jako odporne[4] (*stiff equations*). Gdy próbujemy rozwiązać je numerycznie nieodpowiednimi dla nich metodami, takimi jak właśnie *ode45*, to automatyczny wybór kroku zawodzi. Wydaje się nawet, że w praktycznych problemach najczęściej

[2] Zależność $x(t)$. Jeżeli wybierzemy 2, to zostanie narysowany wykres $v(t)$. Jeżeli pominiemy czwarty parametr lub po prostu damy *optset()*, to wykreślone zostaną wszystkie zmienne na jednym wykresie.

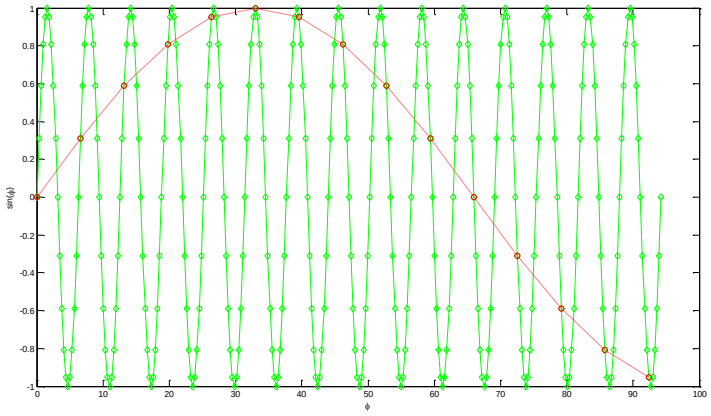
[3] Funkcje *ode23* (Shampine-Bogacki) i *ode45* (Dormand-Prince) implementują schematy Runge-Kutty rzędu odpowiednio 3 i 4 oraz 4 i 5. Każda z nich w trakcie działania używa pary metod po to, aby móc ocenić dokładność obliczeń i odpowiednio dobrać krok. Funkcja *ode113* używa wielokrokowej, z predykatorem i korektorem, automatycznym dopasowaniem rzędu i kroku, metody Adamsa-Bashfortha-Moultona (i jest przeniesiona z fortranowskich DE/STEP/INTERP Shampine'a i Gordona). Funkcja *ode15s* implementuje techniki „różniczkowania do tyłu" NDF (Klopfenstein-Shampine) i BDF wybierając wzory rzędu od 1 do 5. Funkcja *ode23s* wywodzi się z metody Rosenbrocka, ale ma pewne udoskonalenia (Shampine-Reichelty). Funkcja *ode23t* to niejawna metoda trapezów, funkcja *ode23tb* to *ode23t* z różniczkowaniem BDF. Jak widać metody te są dość zaawansowane (i można ewentualnie wywoływać je z różnymi ustawieniami, np. dostarczając jakobian). Ponadto Octave ma wbudowaną funkcję *lsode* (Livermore Solver of ODE), która wymaga nieco innej kolejności parametrów niż funkcje znane z Matlaba.

[4] Rozpowszechniło się, niezbyt poprawne, tłumaczenie *stiff equations* na *równania sztywne*.

będziemy mieli do czynienia z równaniami odpornymi, a przezorne traktowanie wszystkich równań jako sztywnych nie jest szkodliwe.



Rysunek 2. Rozwiązania tego samego równania (oscylator harmoniczny bez tłumienia i bez wymuszenia, $\zeta = 0$, $A = 0$), ta sama procedura, tj. *ode45*, ale różny zakresu czasu symulacji. Po lewej czas w symulacji kończy się na 100 sekundach, po prawej po 100 tysiącach sekund (czyli 1000 razy więcej niż po lewej).



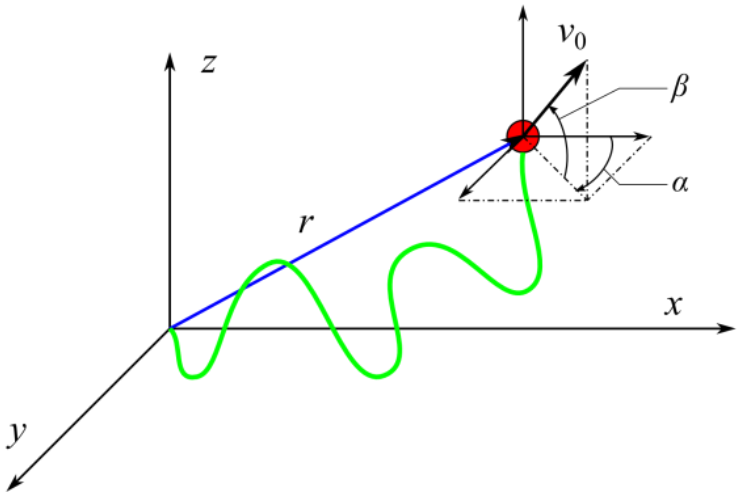
Rysunek 3. Zjawisko aliasingu – przeskakiwane są całe okresy, powstaje krzywa (czerwona) o fałszywie wydłużonym okresie.

Trudniejsze zadanie - wahadło na gumce

Ruch oscylatora harmonicznego jest opisany w tysiącach książek i istnieją jego rozwiązania analityczne. Spróbujmy teraz zastosować Matlab do problemu[5] lepiej wykazującego skuteczność obliczeń numerycznych.

Mamy ciężarek o masie 100 gramów przywiązany do jednometrowej nici gumowej, tak że wisząc na niej rozciągnął ją o 10 centymetrów. Masę gumki pomijamy, opór powietrza pomijamy, siłę Coriolisa pomijamy, przyspieszenie normalne ziemskie wynosi $g = 9.81\,\mathrm{m/s^2}$. Gumka jest przywiązana drugim końcem do początku układu współrzędnych, tj. w tym miejscu współrzędne x, y, z wynoszą zero. Chcemy zobaczyć, jak będzie poruszać się ciężarek na gumce.

[5] W podręcznikach zwykle nie ma innych sił sprężystych niż liniowo i symetrycznie zależne od odkształcenia. Nie jest to realistyczne, ale znakomicie ułatwia wyprowadzanie wzorów matematycznych. Fizyka aż do XX wieku starannie omijała problemy nieliniowe, zwłaszcza prowadzące do chaosu dynamicznego.



Rysunek 4. Układ osi współrzędnych dla zadania „gumka i ciężarek”.

Najtrudniejsze jest uwzględnienie, że czasami gumka jest napięta – a czasami luźna – i to wyklucza[6] rozwiązanie analityczne, czyli zapisane matematycznym wzorem. Dodatkowo warto zauważyć, że „wahadło na gumce” może wychylać się o duże kąty, tj. takie dla których nie można uznać iż $\varphi \approx \sin\varphi$. I oczywiście – mamy trzy stopnie swobody (trzy współrzędne). Wyprowadzenia są w kolejnym akapicie, ale można je pominąć – najważniejsze są równania otrzymane na końcu – właśnie to te równania będziemy rozwiązywali numerycznie.

Początkową długość gumki oznaczamy l_0. Jeżeli ciężarek jest w punkcie o współrzędnych (x, y, z), to jest w odległości $r = \sqrt{x^2 + y^2 + z^2}$ od początku układu współrzędnych. Jeżeli $r < l_0$, to gumka jest nienaciągnięta i na ciężarek działa tylko jego ciężar $P = mg$. Jeżeli $r \geq l_0$, to na ciężarek działa także naciąg gumki, równy co do wartości $F = k(r - l_0)$. Stałą k wyliczamy ze wzoru $mg = k(l - l_0)$, w którym $l = l_0 + \Delta l = 1.10\,\mathrm{cm}$ jest długością gumki po rozciągnięciu przez nieruchomo wiszący ciężarek. Problem jest trójwymiarowy. Składowe ciężaru $\vec{P}$ są oczywiste: $P_x = P_y = 0$, $P_z = -P = -mg$. Składowe siły $\vec{F}$ są też nietrudne do określenia: $F_x = -\frac{x}{r}F$, $F_y = -\frac{y}{r}F$, $F_z = -\frac{z}{r}F$, gdzie oczywiście F może być (dla nienapiętej gumki) równe zero. Stąd (i z drugiej zasady dynamiki Newtona) równania ruchu: $dv_x/dt = F_x/m$, $dv_y/dt = F_y/m$, $dv_z/dt = (F_z + P_z)/m$, gdzie $\vec{v}$ jest prędkością ciężarka. Warunki początkowe opiszmy jako bycie w miejscu o współrzędnych (x_0, y_0, z_0), gdzie $y_0 = 0$, i poruszanie się z prędkością $\vec{v}_0$ taką, że jej wektor ma kąt elewacji β, a jego rzut na płaszczyznę xy jest odchylony o kąt α od kierunku osi x (patrz rysunek). Pionowa składowa prędkości początkowej jest równa $v_z(t=0) = v_0 \sin\beta$, natomiast poziome składowe to $v_x(t=0) = v_0 \cos\alpha \cos\beta$, $v_y(t=0) = v_0 \sin\alpha \cos\beta$. Jak widać układ współrzędnych wybraliśmy tak, aby oś z była skierowana pionowo w górę, a oś x przechodziła przez punkt w którym jest ciężarek w czasie $t = 0$. (Nie zmniejsza to ogólności rozwiązań.) Jeżeli ktoś w tym miejscu trochę się pogubił – to nie powinien się martwić – do tej pory omawialiśmy szczegóły wymagające koncentracji, ale dość nudne i drugoplanowe.

[6] Jeżeli jakiś geniusz dałby sobie radę z tym problemem… to wystarczy nieco zmodyfikować np. użyty model sprężystości… albo uwzględnić masę gumki… albo dodać opór powietrza… Każda taka modyfikacja jest dość łatwa w modelu numerycznym, lecz dla „ścisłego” rozwiązania wprost zabójcza.

Mamy więc układ sześciu równań:

$$\begin{cases} \frac{dx}{dt} = v_x \\ \frac{dy}{dt} = v_y \\ \frac{dz}{dt} = v_z \\ \frac{dv_x}{dt} = -\frac{x}{r}\frac{k}{m}\Delta l \\ \frac{dv_y}{dt} = -\frac{y}{r}\frac{k}{m}\Delta l \\ \frac{dv_z}{dt} = -\frac{z}{r}\frac{k}{m}\Delta l - g \end{cases}$$

gdzie Δl jest równe $\left(\sqrt{x^2+y^2+z^2} - l_0\right)$ jeżeli to wyrażenie jest dodatnie, a zero w pozostałych przypadkach.

Chcąc numerycznie rozwiązać ten problem tworzymy w katalogu roboczym[7] plik o nazwie *gumka.m*. W pliku tym są, jak widać, dwie funkcje – jedna o nazwie *gumka* (czyli zgodnej z nazwą pliku i ta będzie uruchamiała obliczenia) – druga o nazwie *fcn* (czyli subfunkcja dostępna tylko z poziomu funkcji *gumka*). Nasza funkcja *gumka* będzie wywoływała biblioteczną funkcję *ode45*, a ta w czasie rozwiązywania problemu, aby obliczyć prawe strony układu równań, będzie używać funkcji *fcn*. Warto zauważyć, w jaki sposób przekazywane są dodatkowe parametry do funkcji *fcn* – dopisujemy je jako piąty i kolejne argumenty do wywołania *ode45*, a w funkcji *fcn* pojawiają się jako trzeci itd. Alternatywną możliwością jest używanie zmiennych globalnych, albo definiowanie ich jako lokalnych w samej funkcji *fcn*. Każde z tych rozwiązań ma wady i zalety.[8]

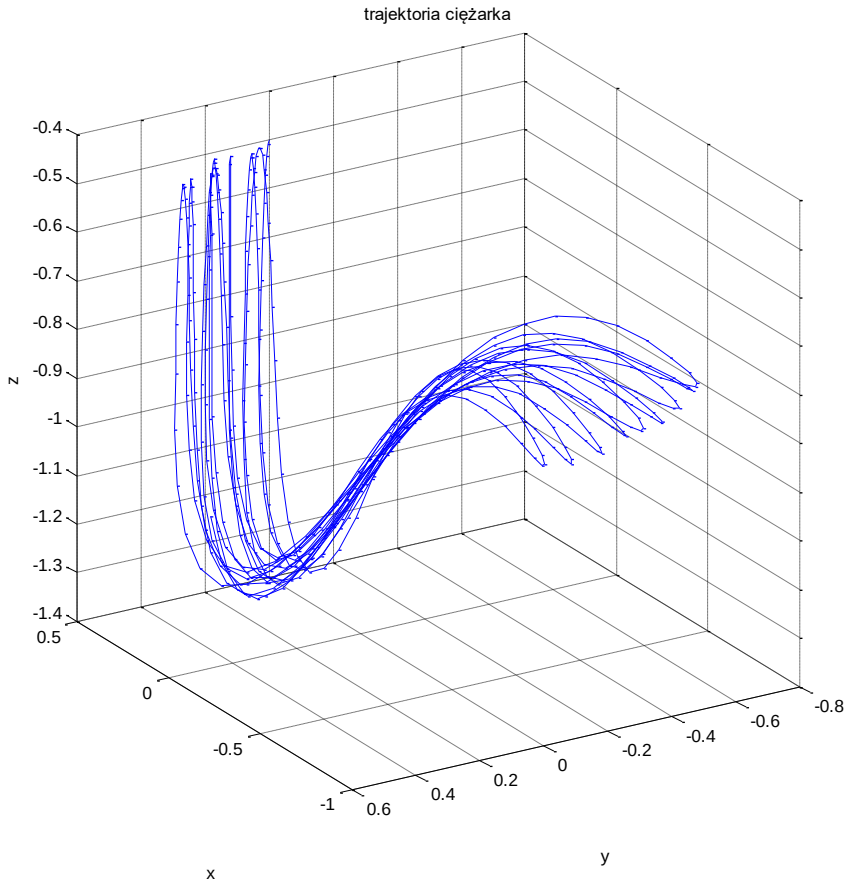
```
function gumka
%
% GUMKA - rozwiązywanie równań ODE (2012, Sławomir Marczyński)

  % Parametry i dane wejściowe

  m  =   0.1  ; % masa ciężarka w kilogramach
  l0 =   1.0  ; % długość nitki gumowej (nierozciągniętej), metry
  l  =   1.1  ; % długość nitki gumowej (po zawieszeniu ciężarka), metry
  g  =   9.81 ; % przyspieszenie ziemskie normalne, metry na sekundę^2

  k  =   m * g / (l - l0);


  % Warunki początkowe

  x0    =   0.5; % w metrach
```

[7] W Microsoft Windows Vista domyślnym katalogiem roboczym dla programu Matlab jest „Moje dokumenty/Matlab" zwykle na dysku systemowym C:, jednak w czasie zajęć w pracowni komputerowej ZUT należy używać dysku D:, bo dysk C: może być niedostępny dla niektórych operacji.

[8] Przekazywanie za każdym razem parametrów może nieco spowolnić obliczenia, globalne zmienne mogą więc dawać większą prędkość działania. Zmienne globalne… są globalne, to znaczy są globalne nie tylko w obrębie danego pliku, ale są wspólne dla wszystkich funkcji jakie są w Matlabie użyte. I to może prowadzić do trudnych do wykrycia błędów, konfliktów nazw. Natomiast zdefiniowanie zmiennych wyłącznie lokalnie nie pozwala na sensowne modyfikowanie ich wartości z zewnątrz.

```
  y0    =   0.0; % w metrach
  z0    =  -0.5; % w metrach

  v0    =   0.1; % prędkość początkowa, w metrach na sekundę
  alfa  =  90.0; % kąt w stopniach
  beta  =   0.0; % kąt w stopniach

  v0x   = v0 * cosd(beta) * cosd(alfa);
  v0y   = v0 * cosd(beta) * sind(alfa);
  v0z   = v0 * sind(beta);

  init  = [ x0, y0, z0, v0x, v0y, v0z];


  % Parametry sterujące symulacją

  n       = 500; % w ilu punktach ma być podane rozwiązanie
  t_start =   0; % czas startu symulacji, sekundy
  t_end   =  20; % czas stopu  symulacji, sekundy

  t   = linspace(t_start, t_end, n); % generowanie wektora czasu
  opt = odeset('InitialStep',0.0001/v0,'MaxStep',0.0001/v0);


  % W tym miejscu wywołujemy ode45 aby rozwiązać równanie

  [czas, wyniki] = ode45(@fcn, t, init, opt, m, l0, k, g);

% Równanie jest już rozwiązane numerycznie,
% pozostaje pokazać wyniki

  plot3(wyniki(:,1), wyniki(:,2), wyniki(:,3));
  axis square;
  grid on;
  xlabel 'x';
  ylabel 'y';
  zlabel 'z';
  title  'trajektoria ciężarka'
end


function dqdt = fcn(t, q, m, l0, k, g)
%
% Funkcja fcn oblicza prawe strony układu równań różniczkowych:
%
%   dq(1)/dt = f1(t,q(1),q(2),...,q(n))
%   dq(2)/dt = f2(t,q(1),q(2),...,q(n))
%         ... = ...
%   dq(n)/dt = fn(t,q(1),q(2),...,q(n))

  dqdt = zeros(size(q));

  x  = q(1);
  y  = q(2);
  z  = q(3);
  vx = q(4);
  vy = q(5);
  vz = q(6);

  r = sqrt(x^2 + y^2 + z^2);
  delta = r - l0;

  dqdt(1) = vx;
  dqdt(2) = vy;
  dqdt(3) = vz;
```

```
  if delta > 0.
    dqdt(4) = - (x/r) * k * delta / m;
    dqdt(5) = - (y/r) * k * delta / m;
    dqdt(6) = - (z/r) * k * delta / m - g;
  else
    dqdt(4) = 0.;
    dqdt(5) = 0.;
    dqdt(6) =  - g;
  end

end
```



Rysunek 5.: Trajektoria obliczona programem *gumka*, widać złożenie drgań w pionie, wahań w poziomie i obrotu.

Podany wyżej skrypt w języku Matlab jest przykładem i dlatego jest napisany z myślą o czytelności, nawet kosztem prędkości działania i zwięzłości zapisu. Doświadczeni użytkownicy Matlaba zapisaliby być może funkcję *fcn* używając długości wektora obliczanej przez *norm*, warunek $r > l_0$ zapisaliby jako wartość logicznego wyrażenia (wynosi ono 1 dla prawdy, 0 dla nieprawdy), być może nie użyliby dodatkowych zmiennych, a parametry m, l_0, k i g ustalali globalnie… i otrzymali np. coś takiego:

```
function dqdt = fcn(t, q)

  global m l0 k g
  dqdt(1:3) =   q(4:6);
  dqdt(4:6) = - q(1:3) * k * (norm(q(1:3) - l0) / m * (norm(q(1:3)) > l0);
  dqdt(  6) =   dqdt(6) - g;

end
```

Taki styl nie jest wzorem do naśladowania – najważniejsza jest poprawność obliczeń, a trudno ją sprawdzić, jeżeli program został napisany nieczytelnie i z nadmiernym użyciem rozmaitych, błyskotliwych, sztuczek.

Automatyczne kończenie obliczeń – strzelanie z wiatrówki

W poprzednich dwóch obliczenia były kończone po założonym czasie (w symulacji), w tym przykładzie warunek stopu będzie inny.

Chcemy oszacować, jaki jest maksymalny realistyczny zasięg pocisku z wiatrówki strzelającej kulkami *Ballistic Ball* (BB). Dane są[9] kaliber $D = 4.5\,\mathrm{mm}$, prędkość wylotowa $v_0 = 120\,\mathrm{m/s}$, masa pocisku $m = 0.33$ grama. Przypominamy sobie wzór na zasięg rzutu ukośnego, wartość przyspieszenia ziemskiego i podstawiamy dane:

$$\text{zasięg} = \frac{v_0^2}{g} \approx \frac{120^2}{10}\left[\frac{\mathrm{m}^2}{\mathrm{s}^2}\frac{\mathrm{s}^2}{\mathrm{m}}\right] = 1440 \text{ metrów}$$

Wynik, jaki otrzymujemy w ten sposób, realistyczny nie jest. Dlaczego? Zignorowaliśmy opór powietrza, a dla ruchu pocisków ma on bardzo duże znaczenie.[10] Jak go uwzględnić? Można założyć, że opór powierza wynika z tarcia lepkiego gazu o kulisty pocisk (wzór Stokesa) – do obliczeń siły $F_{\text{Stokes}} = 3\pi\eta D v$ potrzebna byłaby lepkość powietrza (około $17 \cdot \mu$Pa·s, zależnie od temperatury, wilgotności itd.). Można też założyć, tak jak tu zrobimy, opór aerodynamiczny wynikający z konieczności rozpędzenia powierza przed pociskiem do prędkości pocisku (model Newtona), otrzymując wzór:

$$F = c_x \frac{\rho v^2}{2} S$$

gdzie $S = \frac{\pi}{4} D^2$ jest polem powierzchni przekroju poprzeczego pocisku, v jego prędkością w danej chwili, $\rho = 1.2\,\mathrm{kg/m^3}$ jest gęstością powietrza, c_x jest współczynnikiem zależnym od kształtu, który dla kuli wynosi 0.45. Oczywiście $v = \sqrt{v_x^2 + v_y^2}$. Mamy więc równania:

$$\begin{cases} \dfrac{dx}{dt} = v_x \\ \dfrac{dy}{dt} = v_y \\ \dfrac{dv_x}{dt} = -\dfrac{v_x}{v}\dfrac{F}{m} \\ \dfrac{dv_y}{dt} = -\dfrac{v_y}{v}\dfrac{F}{m} - g \end{cases}$$

Dokładamy do nich warunki początkowe:

$$\begin{cases} x(t=0) = 0 \\ y(t=0) = 1.5 \text{ metra} \\ v_x(t=0) = v_0 \cos\alpha \\ v_y(t=0) = v_0 \sin\alpha \end{cases}$$

[9] Jak łatwo obliczyć średnia gęstość materiału z którego wykonano pocisk wynosi $6\,m/(\pi D^3) = 7.2\ \mathrm{g/cm^3}$, kinetyczna energia początkowa $E_k = \frac{1}{2}mv^2 = 2.4$ dżula (czyli poniżej określonej w przepisach granicy 17 dżuli).

[10] Chyba, że strzelamy np. na Księżycu – wtedy zasięg pocisku wzrośnie jeszcze 6 razy, do ponad $8.5\,\mathrm{km}$.

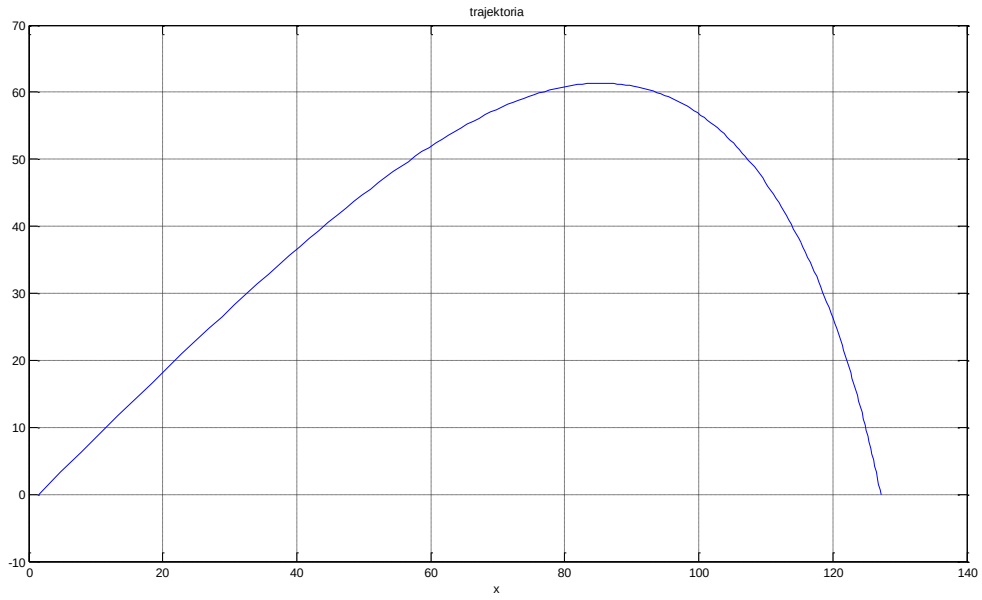
Kodujemy to w skrypcie Matlaba tworząc w pliku o nazwie *zasięg.m* funkcję *zasięg* oraz dwie subfunkcje *fcn* i *events*.[11] Funkcja *fcn* ma znaną już budowę – dostaje wartości zmiennej niezależnej i zmiennych zależnych jako dane – zwraca wartości pochodnych. Funkcja *events* jest funkcją wywoływaną z wnętrza *ode45* i kontrolującą „czy już" – zwróćmy uwagę, iż musi ona mieć dokładnie takie same parametry jak funkcja *fcn*. Czyli jeżeli stosujemy dodatkowe parametry (powyżej czwartego) w wywołaniu funkcji *ode*, to te parametry muszą być jako trzeci i kolejne w funkcjach *fcn* i *events*.[12] Uruchamiając ten skrypt widzimy, że tor lotu pocisku nie jest parabolą, a zasięg jest znacznie mniejszy niż obliczony z pominięciem oporów ruchu.

```
function zasieg
%
% Zasięg strzału pociskiem BB z typowej wiatrówki (2012, Sławomir Marczyński)


  % Warunki początkowe

  x0    =    1.5;          % metry
  y0    =    0.0;          % metry
  v0    =  120.0;          % prędkość początkowa, w metrach na sekundę
  alfa  =   45.0;          % kąt w stopniach

  init  = [ x0, y0, v0 * cosd(alfa), v0 * sind(alfa)];


  % Parametry sterujące symulacją

  n       = 500; % w ilu punktach ma być podane rozwiązanie
  t_start =   0; % czas startu symulacji, sekundy
  t_end   =  20; % czas stopu  symulacji, sekundy

  t   = linspace(t_start, t_end, n); % generowanie wektora czasu
  opt = odeset('Events',@events);    % dodawana jest funkcja kontrolująca


  % W tym miejscu wywołujemy ode45 aby rozwiązać równania,
  % pierwszy, niepotrzebny, wektor zwracany przez ode45 odrzucamy tyldą

  [~, wyniki] = ode45(@fcn, t, init, opt, 0.33E-3, 4.5E-3, 1.2, 9.81);


  % Równanie jest już rozwiązane numerycznie, prezentujemy wyniki

  plot(wyniki(:,1),wyniki(:,2));
  grid on;
  xlabel 'x';
  ylabel 'y';
  title  'trajektoria'
end


function dqdt = fcn(t, q, m, D, ro, g)
```

[11] Ponieważ nazwy subfunkcji nie muszą (a nawet nie mogą) być takie jak nazwa pliku, to przy refaktoryzacji wystarczyłoby zmienić tylko nazwę głównej funkcji (jedna zmiana identyfikatora w jednym miejscu). Wszystkie wywołania subfunkcji pozostałyby bez zmian. Użycie subfunkcji jest wygodniejsze niż pisanie oddzielnych plików *fcn.m* i *events.m*.

[12] Matlab nie sprawdza poprawności listy parametrów zanim dana funkcja nie zostanie wywołana. Dlatego, przy braku kompilacji innej niż JIT, niewłaściwe (nawet jeżeli nieużywane) parametry funkcji *fcn* i *events* będą powodowały błędy run-time, a nie compile-time.

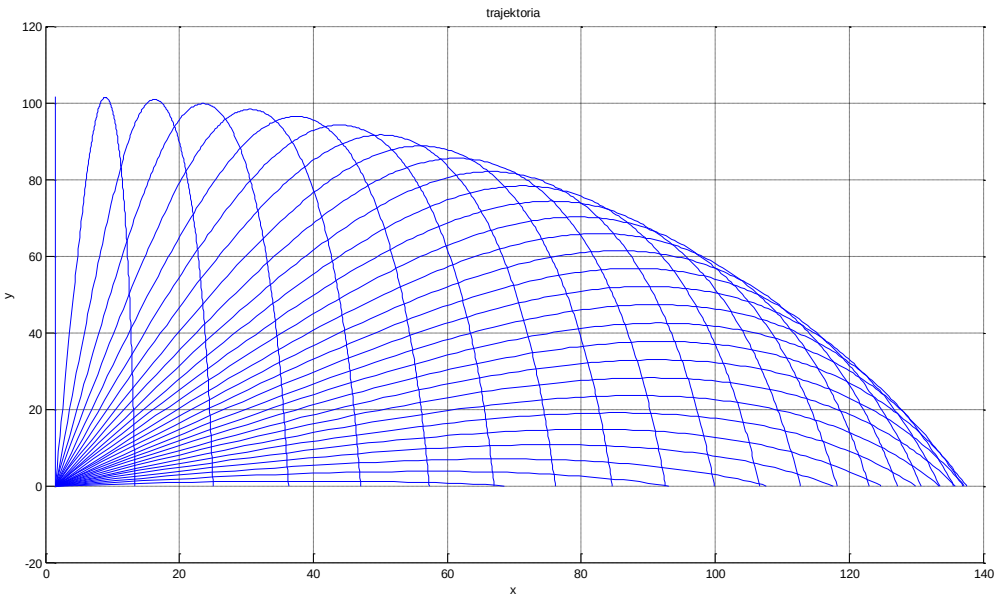
```
  cx = 0.45; % współczynnik oporu areodynamicznego

  dqdt = zeros(size(q));

  x  = q(1);
  y  = q(2);
  vx = q(3);
  vy = q(4);

  v  = sqrt(vx^2 + vy^2);
  F  = (cx/8) * (ro * D * v)^2 * pi;

  dqdt(1) = vx;
  dqdt(2) = vy;
  dqdt(3) = - (vx/v) * (F/m);
  dqdt(4) = - (vy/v) * (F/m) - g;

end


function [value,halt,direction] = events(t,q, m, D, ro, g)

% Obliczenia będą przerywane (halt = 1) gdy q(2) przejdzie przez zero (value = 0)
% od wartości dodatnich do ujemnych (direction = -1).

  value     = q(2);
  halt      =   1 ;
  direction =  -1 ;

end
```



Rysunek 6. Trajektoria pocisku BB wystrzelonego z wiatrówki.

Dla sprawdzenia, zmodyfikujemy plik *zasieg.m* zapisując go pod nazwą *zasiegi.m*: zamiast jednego kąta elewacji i jednej krzywej przebadamy cały zakres możliwych kątów elewacji, bo być może największy zasięg jest uzyskiwany dla kąta innego niż $45°$. Dodajemy funkcję *zasiegi*, zmieniamy bezparametrową funkcję *zasieg* na subfunkcję *zasieg1(alfa)*, usuwamy ustawienie wartości *alfa* w tej subfunkcji. Reszta pozostaje bez zmian, a pętla *for* i polecenie *hold on* wymuszą nałożenie kolejnych wykresów na siebie.

```
function zasiegi

  figure;
  hold on;
  for alfa = 3:3:90
    zasieg1(alfa)
  end
  hold off;

end

function zasieg1(alfa)
%
% Zasięgi strzałów pociskiem BB z typowej wiatrówki (2012, Sławomir Marczyński)


  % Warunki początkowe

  x0    =    1.5;          % metry
  y0    =    0.0;          % metry
  v0    =  120.0;          % prędkość początkowa, w metrach na sekundę
  % alfa                   % kąt w stopniach, jako parametr funkcji zasieg1
```



Rysunek 7. Trajektorie pocisków z wiatrówki przy różnych kątach elewacji, opór powietrza proporcjonalny do kwadratu prędkości. Jak widać tak obliczony zasięg strzału nie przekracza 140 metrów, czyli jest ponad dziesięciokrotnie mniejszy niż bez oporu powietrza. Naturalnie, wyniki te należałoby sprawdzić doświadczalnie. (Na laptopie z procesorem Intel® Core™ 2 Duo T7300@2.00GHz obliczenie i narysowanie wszystkich krzywych trwało około pół sekundy.)

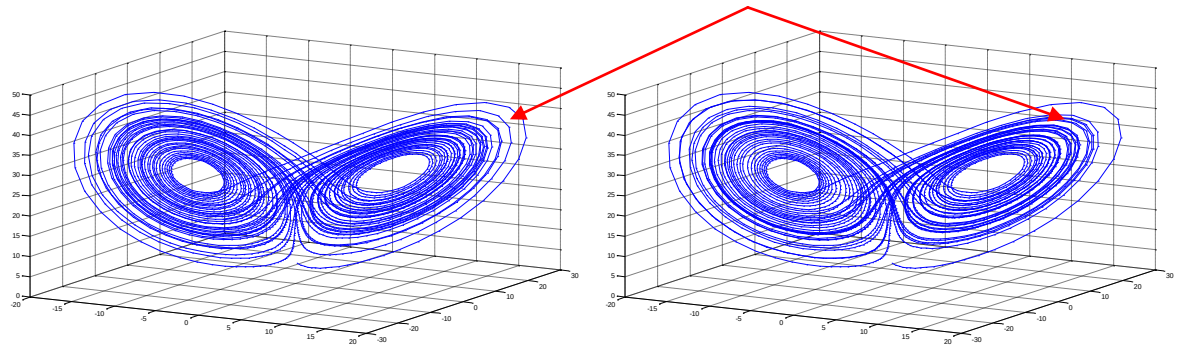
Bardzo krótko o chaosie dynamicznym

Istnieje możliwość, iż natrafimy na równania ODE, których rozwiązania będą po prostu chaotyczne: za każdym razem zupełnie inne, nawet jeżeli tylko nieznacznie (np. o jedną miliardową) zmieniliśmy dane. Na przykład gdy kąt początkowy zmieni się z $45°$ na $45.000000001°$.spodziewalibyśmy się, że rozwiązania równań są bardzo zbliżone do siebie. Jednak gdy równania są chaotyczne, to dowolnie mała zmiana warunków początkowych „na wejściu" prowadzi do wykładniczego wzrostu rozbieżności między rozwiązaniami „na wyjściu" (dokładniej mierzy to wykładnik Łapunowa) i nie istnieje

jakakolwiek metoda pozwalająca wiarygodnie obliczyć rozwiązania.[13]. Na przykład, taka jednomiliardowa zmiana kąta może spowodować inny wynik losowania stron boiska przez rzut monetą (orzeł-reszka). Dlatego trzeba być świadomym ograniczeń metod numerycznych, w tym funkcji *ode* z programu Matlab.

Typowym układem chaotycznych równań jest układ Lorenza:

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = -x(\rho - z) - y \\ \frac{dz}{dt} = xy - \beta z \end{cases}$$



Rysunek 8. Próby rozwiązywania układu równań Lorentza dla $\sigma = 10$, $\beta = 8/3$, $\rho = 28$, fragment dla $0 \le t \le 50$. Wykresy po lewej i prawej stronie zostały zrobione dla warunków początkowych (x_0, y_0, z_0) odpowiednio równych $(-1., 3, 4)$ oraz $(-1.000000001, 3, 4)$. Zmiana o jedną miliardową początkowego x_0 spowodowała widoczną zmianę rozwiązań.

Skrypty, pozwalające rozwiązywać ten układ równań, dla programów Matlab i Octave, są opisane i dostępne np. w Wikipedii[14]. Ale ponieważ układ jest chaotyczny, to wyniki ich działania nie są „prawdziwymi rozwiązaniami" – wystarczy bowiem dowolnie mała zmiana warunków początkowych (albo dokładności obliczeń) otrzymamy zupełnie inne wyniki.[15]

Podsumowując: środowiska obliczeniowe Matlab i Octave[16] są potężnymi narzędziami pozwalającymi na rozwiązywanie równań ODE w bardzo łatwy sposób. Nie jest do tego wymagany jakiś szczególny talent matematyczny, a jedynie umiejętność zapisania tych równań w postaci zrozumiałej dla komputera.

[13] Podobnie jak nie można napisać wszystkich cyfr liczby π.

[14] Patrz (23 maja 2012) np. http://en.wikipedia.org/wiki/Lorenz_system. Skrypt dla Octave używa *lsode*.

[15] W sensie wrażenia estetycznego będą one podobne.

[16] Przykłady powinny działać w Octave po zainstalowaniu dodatkowych pakietów z funkcjami *ode*.