Jacek Matulewski

Metody Runge-Kutty w praktyce

Czyli co każdy programista powinien wiedzieć o dokładnym całkowaniu równań różniczkowych zwyczajnych

W artykule pt. *Podstawy mechaniki klasycznej dla programistów gier, czyli rzecz o tym, jak całkować równanie ruchu* z numeru 7/2013 (14) *Programisty*, opisałem metody Eulera-Cromma i Verleta używane w grach do rozwiązywania równań ruchu fizyki klasycznej (równanie Newtona). Teraz chciałbym opisać metody, których należy użyć, gdy priorytetem jest dokładność wyników. Ich zastosowanie nie ogranicza się do gier.

Metody te są przeznaczone do symulacji ewolucji dowolnych układów opisywanych równaniami różniczkowymi zwyczajnymi. A takie równania znajdziemy nie tylko w fizyce i technice, ale również w biologii, geografii, ekonomii czy medycynie (epidemiologia). Nie mam wobec tego wątpliwości, że umiejętność poradzenia sobie z takimi równaniami może być bardzo przydatna programiście, szczególnie jeżeli w trakcie studiów nie zetknął się z metodami numerycznymi.

RÓWNANIA RÓŻNICZKOWE ZWYCZAJNE

Równania różniczkowe zwyczajne (ang. *ordinary differential equations*, ODE) *n*-tego stopnia mają ogólną postać:

$$y^{(n)}(x)=f\left(y,y^{(1)},\ldots,y^{(n-1)},x\right)$$

Nawias w górnym indeksie oznacza pochodną. Z lewej strony mamy pochodną *n*-tego rzędu, a z prawej dowolną funkcję, która wiąże zmienną *x* oraz pochodne funkcji *y* do rzędu *n*-1 włącznie. Interesuje nas znalezienie funkcji *y*(*x*). To bardzo ważna grupa równań, ponieważ jak wspomniałem wyżej bardzo wiele problemów fizyki, techniki, biologii czy ekonomii opisywanych jest tego typu równaniami.

Bez wątpienia najsłynniejszym i najważniejszym równaniem tego typu jest równanie ruchu punktu materialnego odkryte przez Newtona (kropki oznaczają tu pochodne po czasie):

$$\ddot{\vec{r}}(t)=\frac{\vec{F}(\vec{r},\dot{\vec{r}},t)}{m}.$$

To równanie na ewolucję wektora położenia w trójwymiarowej przestrzeni, ale bez problemu można je rozpisać na trzy równania dla poszczególnych składowych. Równanie ruchu jest równaniem drugiego stopnia. Korzystając z definicji prędkości jako pochodnej położenia po czasie, można je rozpisać na dwa równania pierwszego stopnia:

$$\dot{\vec{v}}(t)=\frac{\ddot{\vec{F}}(\vec{r},\dot{\vec{r}},t)}{m},$$

$$\dot{\vec{r}}(t)=\vec{v}(t).$$

Podobnie można zrobić z dowolnym równaniem różniczkowym zwyczajnym dowolnego stopnia *n*. Można je zamienić na *n* równań pierwszego stopnia. W powyższym przykładzie mamy tak naprawdę aż sześć równań (trzy dla składowych wektora położenia i trzy dla składowych wektora prędkości). Gdy układ składa się z większej liczby punktów materialnych, liczba równań wzrasta do równej iloczynowi stopnia równania, liczby wymiarów i liczby punktów. Naszym celem jest przygotowanie kodu C++ służącego do rozwiązywania dowolnie dużego zbioru równań różniczkowych zwyczajnych pierwszego rzędu. W ten sposób uzyskamy narzędzie, z pomocą którego będziemy mogli zmierzyć się z każdym problemem opisywanym tego typu równaniami. Użyjemy do tego metod z rodziny Runge-Kutty (RK). Testy przeprowadzimy na prostych układach fizycznych.

Zdaję sobie sprawę, że czytelnika mogą nie interesować szczegóły wyprowadzenia poszczególnych metod RK. Dla wyższych rzędów są to naprawdę skomplikowane rachunki. Bardziej interesujący jest zapewne gotowy do użycia kod. Dlatego przedstawię odpowiednie kody, robiąc tylko niezbędne uwagi potrzebne do ich prawidłowego użycia we własnych projektach.

Aby nasz kod był możliwie ogólny, załóżmy, że stan układu opisywany jest przez *N*-elementowy wektor, którego kolejne składowe mogą być na przykład składowymi wektorów położenia i prędkości. Równanie różniczkowe pierwszego rzędu dla każdej składowej tego wektora y_i będzie miało postać:

$$y_i^{(1)}(t)=f_i(y,t)=f_i(\{y_j\}_{i=0}^{N-1},t).$$

Argumentem funkcji *f* są wszystkie składowe wektora *y* oraz zmienna *t*. Ponieważ wszystkie testy, jakie przedstawię w artykule, będą oparte na przykładach ewolucji układów fizycznych zamiast zmiennej *x*, będę używał *t*, której standardowo używa się do oznaczenia czasu.

Kod przygotujemy w C++. Nie ma większego znaczenia, jakiego środowiska programistycznego i kompilatora użyjemy. Ja używam środowiska Microsoft Visual C++ 2015 (VC++) w wersji *preview* i do niego będę się odwoływał w opisie, ale równe dobrze mógłby to być kompilator g++ z dowolnym edytorem. W VC++ stworzyłem projekt o nazwie *OdeInt* typu *Empty Project*, aby uniknąć dodawanego do pozostałych projektów kodu, który utrudniałby przenaszalność projektu do innych kompilatorów. Do pseudokatalogu *Header Files* dodałem plik nagłówkowy *odeint.h*, a do *Source Files* plik *odeint.cpp*. To będą jedyne pliki projektu.

METODA EULERA (METODA RK PIERWSZEGO RZĘDU)

Zacznijmy od najprostszej metody RK, czyli metody Eulera. Jej implementacja jest łatwa i intuicyjna, ale z góry muszę uprzedzić, że niewiele będzie z niej pożytku. Nigdy nie należy jej używać, jeżeli zależy nam na dokładnych wy-

nikach. Funkcja rozwiązująca zbiór równań musi przyjmować cały wektor *y*, jego rozmiar, zmienną *t*, krok całkowania *h*, wektor, do którego zapiszemy wynik, oraz zbiór funkcji f_i, czyli w istocie opis problemu, który reprezentują równania. Kod solvera `odeint_Euler` widoczny na Listingu 1 umieśćmy w pliku nagłówkowym *odeint.h*. Funkcja `f`, drugi argument funkcji `odeint_Euler`, musi pozwalać na obliczenie każdej funkcji f_i, dlatego musi przyjmować indeks funkcji *i*, wektor *y* oraz bieżącą wartość zmiennej *t*.

Listing 1. Implementacja metody Eulera (plik odeint.h)

```
#pragma once

template<typename T>
T* odeint_Euler(int N, T(*f)(int, T*, T), T* y, T t, T h, T*
y_nast)
{
  for (int i = 0; i < N; ++i) y_nast[i] = y[i] + h* f(i, y, t);
  return y_nast;
}
```

Metodę Eulera można uzasadniać na wiele sposobów: jest prostym przekształceniem ilorazu różniczkowego, wynikiem całkowania po przedziale (*t*, *t+h*) funkcji *f* przybliżonej przez jej wartość w punkcie *t* czy rozwinięciem w szereg Taylora najniższego nietrywialnego rzędu. Tak czy inaczej jej wartość numeryczna jest bardzo marna i do poważnych symulacji się nie nadaje.

SPADEK SWOBODNY

Sprawdźmy to na przykładzie bardzo prostego układu, w którym punkt materialny w jednowymiarowej przestrzeni będzie poddawany działaniu stałej siły. Z taką sytuacją mamy do czynienia na przykład, gdy spuszczamy jakiś przedmiot – wówczas porusza się on pionowo (w jednym wymiarze) i działa na niego stałe przyspieszenie ziemskie. Oczywiście pomijamy opory powietrza i siłę ewentualnych podmuchów wiatru, jeżeli eksperyment wykonujemy na dworze. Równanie ruchu w takim przypadku wygląda następująco:

$$m\ddot{x}(t)=mg.$$

Zakładam przy tym, że oś *x* skierowana jest pionowo w dół, czyli wzdłuż wektora przyspieszenia ziemskiego. Możemy pominąć masy i rozpisać to równanie na dwa równania pierwszego rzędu:

$$\dot{x}(t)=v(t),$$

$$\dot{v}(t)=g.$$

Równania te możemy rozwiązać na kartce. Rozwiązując drugie, uzyskamy $v(t)=v_0+gt$. Wstawiając to rozwiązanie do pierwszego z równań, otrzymamy $x(t)=x_0+v_0t+gt^2/2$. To znane ze szkoły prędkość i położenie w ruchu jednostajnie przyspieszonym. Do obliczenia ich wartości niezbędne są dwie stałe całkowania – w całym artykule zakładamy, że są nimi położenie i prędkość początkowa w $t = 0$.

Dla tego układu wektor stanu *y* będzie się wobec tego składał tylko z dwóch składowych:

$$y=\{y_0, y_1\}=\{x, v\}.$$

Powyższe równania można zatem przepisać, korzystając z nowych oznaczeń jako:

$$\dot{y}_0=f_0(y,t)=v,$$

$$\dot{y}_1=f_1(y,t)=g.$$

Funkcja *f* dla spadku swobodnego (lub rzutu, o ile prędkość początkowa jest skierowana wzdłuż osi *x*) może być wobec tego zaimplementowana jak na Listingu 2. Wartość stałej przyspieszenia ziemskiego użyta w tej funkcji wyznacza jednostki, jakie używamy w programie. W przypadku z Listingu 2, a więc gdy stała g równa jest około 10, są to metry i sekundy, a zatem jednostki układu SI.

Listing 2. Spadek swobodny (plik odeint.h)

```
#pragma once

#include <stdexcept>
using namespace std;

template<typename T>
T rzut(int i, T* y, T t) //1D, N=2
{
  static const double g = 9.81; // => SI

  double wynik = 0;
  switch (i)
  {
  case 0:
    wynik = y[1];
    break;
  case 1:
    wynik = g;
    break;
  default:
    throw runtime_error("Zły numer równania");
  }
  return wynik;
}
```

Złóżmy dwa elementy w całość, aby móc przeprowadzić symulację. Listing 3 zawiera kod z pliku *odeint.cpp* wraz z funkcją `main`. W tej funkcji tworzymy dwa wektory *y* i *y_nast*, ustalamy wartości początkowe i uruchamiamy pętlę symulacji, w której obliczane są wartości *y* z kolejnych chwil czasu.

Listing 3. Kod pliku odeint.cpp

```
#include <iostream>
#include <fstream>
using namespace std;

#include "odeint.h"

int main()
{
  int N = 2;

  double* y = new double[N];
  double* y_nast = new double[N];

  //warunki początkowe
  for (int i = 0; i < N; ++i)
  {
    y[i] = 0;
    y_nast[i] = 0;
  }

  double tmax = 10;
  double h = 0.01;

  //plik
  ofstream plik_wy("wyniki.dat");
  plik_wy.precision(10);
  plik_wy.setf(ios::scientific);

  //ewolucja układu
  for (double t = 0; t < tmax; t += h)
  {
    odeint_Euler<double>(N, rzut<double>, y, t, h, y_nast);

    plik_wy << t;
    for (int i = 0; i < N; ++i) plik_wy << "\t" << y[i];
    plik_wy << "\t" << h;
    plik_wy << "\n";

    //zamiana tablic
    double* y_tmp = y;
    y = y_nast;
    y_nast = y_tmp;
  }

  plik_wy.close();

  cout << "\n\nOK.\n";

  delete[] y;
}
```

Teraz można skompilować i uruchomić program. W efekcie powinien powstać plik *wyniki.dat* (w katalogu *C:\Users\[login] \Documents\Visual Studio 2015\Projects\OdeInt\OdeInt*) zawierający w kolejnych kolumnach bieżący czas symulacji, położenie, przyspieszenie i krok czasowy. Zawartość tego pliku najlepiej obejrzeć za pomocą jakiegoś programu do rysowania wykresów. Ja używam darmowego gnuplota (http://www.gnuplot.info). Wykresy widoczne na Rysunku 1 pokazują, że uzyskane dzięki metodzie Eulera wyniki (czerwone linie) są zgodne z tym, co przewidują rozwiązania analityczne (zielona linia), a więc mamy liniową zmianę prędkości i kwadratową zmianę położenia. Nie dajmy się jednak zwieść tej zgodności i sprawdźmy zachowanie wielkości, która nie powinna się zmieniać w trakcie ewolucji – całkowitej energii mechanicznej tego układu. To dobry sposób weryfikowania poprawności obliczeń numerycznych. Na energię całkowitą składa się rosnąca energia kinetyczna spadającego punktu oraz jego energia potencjalna liniowo malejąca wraz z wysokością:

$$E_c = \frac{mv^2}{2} + mgx\,.$$

Jeżeli energię potencjalną wyskalujemy tak, że będzie miała wartość zero na wysokości, z której puszczamy ciało, to energia całkowita powinna zachowywać zerową wartość przez cały czas symulacji. Z Rysunku 2 widać jednak, że tak wcale nie jest. Możemy oczywiście zmniejszyć krok całkowania (zob. wykresy dla $h = 0.01$, $h = 0.001$, $h = 0.0001$ również widoczne są na Rysunku 2, prawym), ale to znacznie zwiększa koszt obliczeń, a przy tym tak naprawdę nie rozwiązuje problemu – błąd nadal rośnie liniowo, tyle że wolniej.

METODA PUNKTU ŚRODKOWEGO (METODA RK DRUGIEGO RZĘDU)

Znacznie lepszym rozwiązaniem jest zastosowanie bardziej dokładnej metody całkowania. Użyjmy metody RK drugiego rzędu (RK2), czyli tzw. metody punktu środkowego (Listing 4). Jej koszt jest mniej więcej dwukrotnie większy niż metody Eulera, ale precyzja wyników kompensuje to z nawiązką.

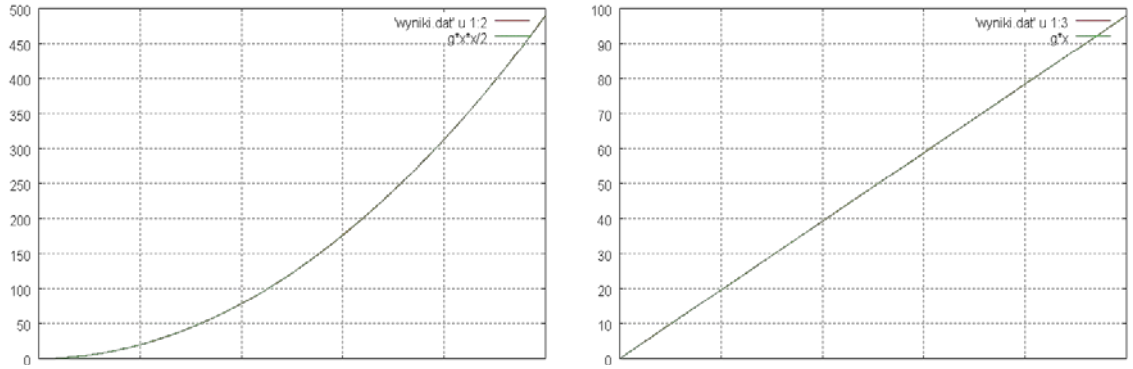
Listing 4. Implementacja metody punktu środkowego (plik odeint.h)

```
template<typename T>
T* odeint_MidPoint(int N, T(*f)(int, T*, T), T* y, T t, T h, T* y_nast)
{
  T* y_tmp = new T[N];
  for (int i = 0; i < N; ++i)
  {
    T k1 = h*f(i, y, t);
    y_tmp[i] = y[i] + 0.5*k1;
  }
  for (int i = 0; i < N; ++i)
  {
    T k2 = h * f(i, y_tmp, t + 0.5*h);
    y_nast[i] = y[i] + k2;
  }
  return y_nast;
}
```

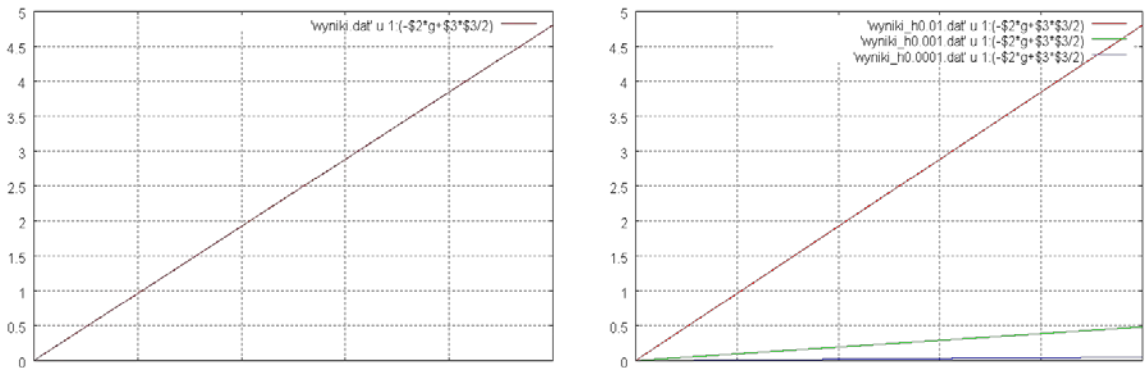
Ponieważ w funkcji `odeint_MidPoint` zachowaliśmy sygnaturę funkcji `odeint_Euler`, zmiana metody całkowania jest prosta. Wystarczy tylko w funkcji `main` zmienić nazwę wywoływanej funkcji:

```
//ewolucja układu
for (double t = 0; t < tmax; t += h)
{
  odeint_MidPoint<double>(N, rzut<double>, y, t, h, y_nast);
  ...
```

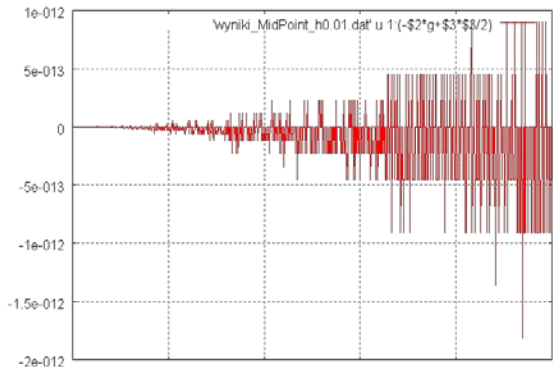
Wróćmy do pierwotnego kroku czasowego $h = 0.01$ i porównajmy działanie obu metod. Energię całkowitą policzoną z użyciem metody punktu środkowego pokazuje Rysunek 3. Jak widać utrzymuje się na poziomie zera numerycznego dla liczb rzeczywistych podwójnej precyzji (por. z Rysunkiem 2, lewym). Jej zmiana jest mniejsza od błędów zaokrągleń, widocznych na Rysunku 3 w postaci szumu. Takiej dokładności nie uzyskaliśmy metodą Eulera, nawet tysiąckrotnie zmniejszając długość kroku całkowania.



Rysunek 1. Położenie i prędkość w spadku swobodnym w funkcji czasu



Rysunek 2. Energia całkowita w funkcji czasu – jej zmiana jest dobrą miarą błędu całkowania



Rysunek 3. Energia całkowita w funkcji czasu policzona z użyciem metody punktu środkowego

OSCYLATOR HARMONICZNY

Zbadajmy inny, ale nadal jednowymiarowy układ fizyczny zbudowany także tylko z jednego punktu. Wektor stanu *y* pozostanie więc niezmieniony. Układem tym będzie oscylator harmoniczny. Można go sobie wyobrazić jako ciężarek zaczepiony na sprężynie. Położeniem *x* będzie wychylenie ciężarka z położenia równowagi. Wychylenie takie prowadzi do powstania siły, która w pewnym zakresie wychyleń zależy od tego wychylenia liniowo (harmonicznie):

$$F(x) = -kx.$$

Ściśnięcie lub rozciągnięcie sprężyny powoduje powstanie siły dążącej do przywrócenia położenia równowagi. Jednak ponieważ ciężarek nabiera prędkości, mija położenie równowagi i oscylator zaczyna oscylować. Jeżeli nie działają siły oporu, będzie oscylował bez końca ze stałą amplitudą. Do pliku nagłówkowego *odeint.h* dodajmy funkcję opisującą ten układ (Listing 5). Zmieniająca się siła powinna być większym wyzwaniem dla metody punktu środkowego. Funkcja napisana jest tak, że umożliwia włączenie sił oporu tłumiących oscylacje, ale na razie są one wyłączone (stała b równa jest zeru). Dla uproszczenia zakładamy, że masa ciężarka równa jest 1, a zamiast stałej *k* użyłem wyrażenia $k = \omega^2$, gdzie ω to częstość oscylacji. Wynika to wprawdzie z analitycznego rozwiązania równania ruchu oscylatora (położenie i prędkość opisane są funkcjami *sin* i *cos*), ale takie podstawienie nie zmienia ogólności funkcji `oscylator`. Zresztą przecież to nie ta funkcja, ani stojący za nią układ, jest bohaterem artykułu, a metoda numeryczna, dla której ma być wyzwaniem.

Listing 5. Oscylator harmoniczny

```
template<typename T>
T oscylator(int i, T* y, T t) //1D, N=2
{
  static const double w = 1;
  static const double b = 0;

  double wynik = 0;
  switch (i)
  {
  case 0:
    wynik = y[1];
    break;
  case 1:
    wynik = -w*w*y[0] - 2 * b*y[1];
    break;
  default:
    throw runtime_error("Zły numer równania");
  }
  return wynik;
}
```

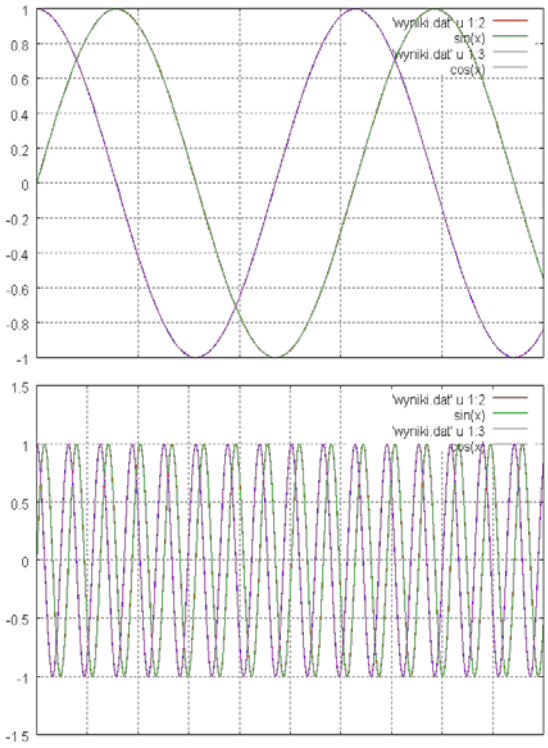
Aby zbadać ewolucję nowego układu, musimy jedynie zmienić funkcję podawaną jako drugi argument funkcji `odeint_MidPoint`:

```
odeint_MidPoint<double>(N, oscylator<double>, y, t, h, y_nast);
```

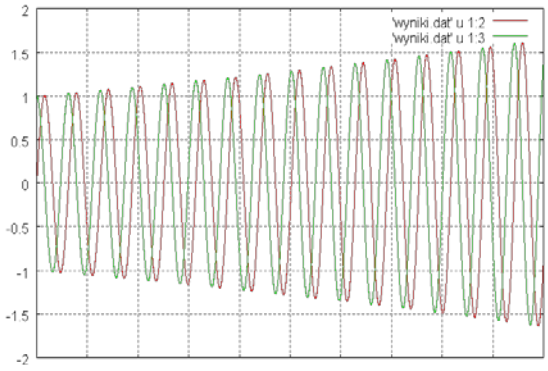
oraz warunki początkowe (prędkość równa 1):

```
y[1] = 1;
```

Przy wcześniejszych oscylator w nieskończoność pozostawałby w położeniu równowagi. Aby obserwować ruch oscylatora, należy albo naciągnąć sprężynę (y_0), albo nadać jej prędkość początkową (y_1). Ja wybrałem to drugie. Wyniki obliczeń (położenie i prędkość) uzyskanych metodą punktu środkowego dla $t_{max} = 10$ i $t_{max} = 100$ widoczne są na Rysunku 4. Dla porównania na Rysunku 5 widoczne są wyniki uzyskane metodą Eulera. Widać wyraźnie, że w tym drugim przypadku rośnie amplituda oscylacji, co nie powinno mieć miejsca.



Rysunek 4. Położenie i prędkość oscylatora uzyskana metodą punktu środkowego (nakrywające się linie dla rozwiązań analitycznych i numerycznych)



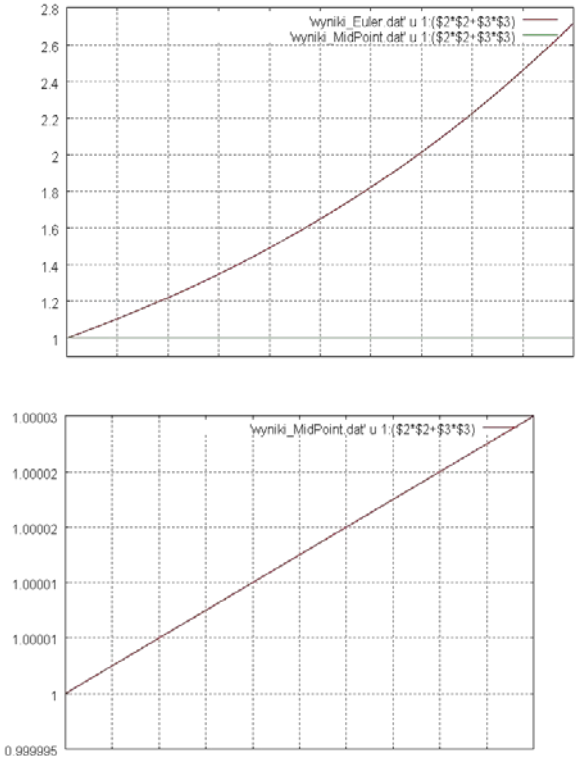
Rysunek 5. Jak na Rysunku 4, dolnym, ale dla metody Eulera

Nauczeni doświadczeniem uzyskanym przy symulowaniu spadku swobodnego, powinniśmy już wiedzieć, że jakość metody numerycznej najlepiej sprawdzić, sprawdzając wartości jakiejś wielkości, która powinna pozostawać

stała (tzw. całka ruchu). W przypadku oscylatora tą wielkością ponownie jest całkowita energia mechaniczna, tj.:

$$E_c = \frac{mv^2}{2} + \frac{kx^2}{2}$$

Jeżeli narysujemy tę wartość, uzyskamy wykresy widoczne na Rysunku 6. Potwierdzają one, że metoda punktu środkowego (RK2) znacznie przewyższa dokładnością metodę Eulera (RK1). Pokazuje jednak także, że przy dłuższych obliczeniach błąd metody punktu środkowego zacznie być istotny – rośnie bowiem liniowo z czasem. Również ta metoda nie jest wobec tego wystarczająca przy precyzyjnych obliczeniach.



Rysunek 6. U góry: porównanie energii całkowitej w funkcji czasu w obliczeniach metodą Eulera i punktu środkowego. Na dole: wykres tylko dla metody punktu środkowego

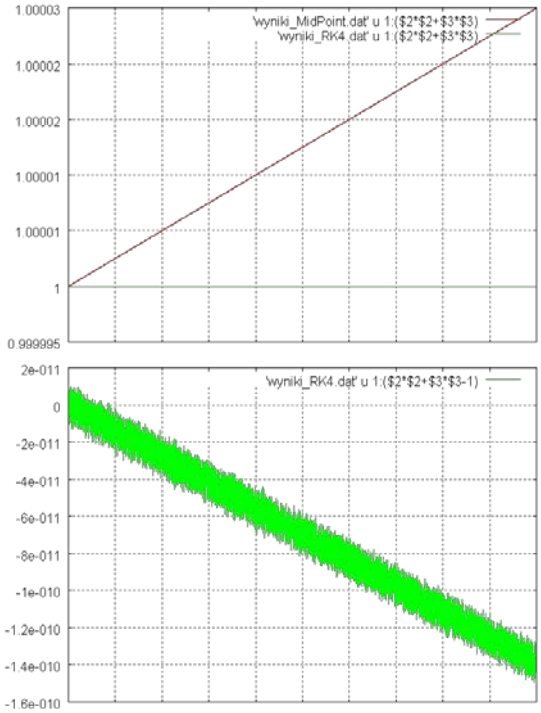
METODA RUNGE-KUTTY 4 (RK4)

Dotychczasowe testy sugerują, że aby uniknąć problemu stopniowo narastającego błędu obliczeń, lepiej zwiększyć rząd metody Runge-Kutty niż zmniejszać krok całkowania. Oczywiście, czas niezbędny do implementacji i testowania bardziej skomplikowanej metody może być znaczący. Może nawet przewyższać czas obliczeń, szczególnie jeżeli musimy je wykonać tylko raz dla jednego zestawu parametrów. Ale właśnie po to powstał ten artykuł, aby czas implementacji zminimalizować i zmniejszyć koszt użycia bardziej zaawansowanych metod. Do artykułu dołączony jest bowiem kod źródłowy (do pobrania z www.programistamag.pl) zawierający gotowe funkcje, z których każdy może korzystać we własnych projektach. W pliku nagłówkowym *odeint.h* znajduje się także funkcja `odeint_RK4` implementująca metodę Runge-Kutty czwartego rzędu (RK4, Listing 6). To ona jest chyba najczęściej stosowaną w symulacjach fizycznych wersją metody Runge-Kutty.

Listing 6. Implementacja metody Runge-Kutty 4 (plik odeint.h)

```
template<typename T>
T* odeint_RK4(int N, T(*f)(int, T*, T), T* y, T t, T h, T* y_nast)
{
  T* y_tmp = new T[N];
  T* k1 = new T[N];
  T* k2 = new T[N];
  T* k3 = new T[N];
  T* k4 = new T[N];
  for (int i = 0; i < N; ++i)
  {
    k1[i] = h*f(i, y, t);
    y_tmp[i] = y[i] + 0.5*k1[i];
  }
  for (int i = 0; i < N; ++i)
  {
    k2[i] = h*f(i, y_tmp, t + 0.5*h);
    y_nast[i] = y[i] + 0.5*k2[i];
  }
  for (int i = 0; i < N; ++i)
  {
    k3[i] = h*f(i, y_nast, t + 0.5*h);
    y_tmp[i] = y[i] + k3[i];
  }
  for (int i = 0; i < N; ++i)
  {
    k4[i] = h* f(i, y_tmp, t + h);
    y_nast[i] = y[i] + (k1[i] + 2.0*k2[i] + 2.0*k3[i] + k4[i]) / 6.0;
  }
  delete[] k1;
  delete[] k2;
  delete[] k3;
  delete[] k4;
  delete[] y_tmp;

  return y_nast;
}
```

Porównajmy wyniki, uzyskane dzięki nowej funkcji, z wynikami uzyskanymi metodą punktu środkowego. Na Rysunku 7 widoczna jest energia całkowita układu w przypadku całkowania obiema metodami. W tym drugim przypadku (zielona linia) zmiana jest o kilka rzędów mniejsza, niewiele większa od szumu wynikającego z błędów zaokrągleń. Zmniejszając krok, możemy ją jeszcze dodatkowo zmniejszyć.



Rysunek 7. Energia całkowita oscylatora harmonicznego w funkcji czasu. U góry porównanie energii uzyskanej za pomocą metody punktu środkowego i metody RK4, na dole energia dla metody RK4

OSCYLATORY SPRZĘŻONE

Zróbmy jeszcze jeden test. Zastąpmy pojedynczy oscylator przez grupę oscylatorów sprzężonych, czyli wiele ciężarków połączonych sprężynkami. Z wielu tego typu oscylatorów można zbudować na przykład linę do bungie.

W pliku nagłówkowym zdefiniujmy opisującą ten nowy układ funkcję `oscylatory_sprzezone` (Listing 7). Funkcja ta ma tę zaletę w porównaniu do poprzednich, że zadziała dla dowolnej parzystej liczby równań (wartość stałej N). Oddziaływania harmoniczne wiążą tylko bezpośrednio sąsiadujące pary punktów. W efekcie na każdy punkt, poza skrajnymi, działają dwie przeciwnie skierowane siły o wartościach proporcjonalnych do odchylenia odległości między tymi punktami od odległości równowagowej (zmienna `l` równa 1). Nadal wyłączone są wszelkie opory – eksperymenty z tłumieniem pozostawiam czytelnikom. Warto też zdefiniować funkcję opisującą dwuwymiarowy układ ciężarków połączonych sprężynami – siatkę, lub trójwymiarowy – sprężystą galaretkę.

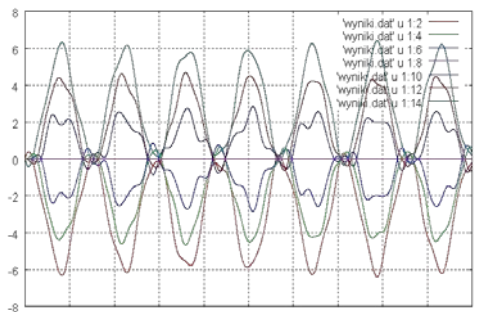
Listing 7. Oscylatory sprzężone

```
const int N = 14;

...

template<typename T>
T oscylatory_sprzezone(int i, T* y, T t) //1D, dla siedmiu
punktów N = 14
{
  static const double w = 1;
  static const double b = 0;
  static const double l = 1;

  double wynik = 0;
  if (i % 2 == 0)
  {
    //położenia
    wynik = y[i + 1];
  }
  else
  {
    //prędkości
    wynik = -2 * b*y[i];
    if (i > 1) wynik += -w*w*(y[i - 1] - y[i - 3] - l);
    if (i < N - 1) wynik += +w*w*(y[i + 1] - y[i - 1] - l);
  }

  return wynik;
}
```

Wynik obliczeń dla $N = 14$ (siedem oscylatorów), a konkretnie położenia kolejnych punktów, widoczne są na Rysunku 8. Jeżeli założymy symetryczne warunki początkowe, ustalając prędkości dwóch skrajnych punktów:

```
y[1] = 1;
y[N - 1] = -1;
```

to cała ewolucja układu powinna przebiegać w taki sposób, aby symetria układu względem środkowego punktu była zachowana. W efekcie czwarty punkt nie powinien w ogóle się poruszać, co widać na Rysunku 8 (purpurowa linia). Symetrię układu można też sprawdzić, sumując położenia odpowiednich dwóch punktów, np. pierwszego i ostatniego lub drugiego i szóstego. Sprawdziłem, że z dużą dokładnością taka suma równa jest zeru przez cały czas symulacji.



Rysunek 8. Położenia sprzężonych oscylatorów uzyskanych metodą RK4

ZAGADNIENIE DWÓCH CIAŁ (ZIEMIA-KSIĘŻYC)

W kolejnych testach użyjemy zagadnień wielu ciał, które mają zastosowanie w astronomii, w tzw. mechanice nieba badającej ruch ciał niebieskich, w szczególności planet. W pliku nagłówkowym (*odeint.h*) definiujemy kolejną funkcję o nazwie `grawitacja` (Listing 8) określającą nowy problem fizyczny – jest to zagadnienie dwóch ciał z oddziaływaniem grawitacyjnym:

$$\vec{F} = G\frac{Mm}{r^2}\frac{\vec{r}}{r},$$

w którym G to stała grawitacji, M i m – masy ciał, wektor $\vec{r}$ łączy te ciała, a r to odległość między nimi. Zgodnie z trzecią zasadą dynamiki Newtona na oba ciała działa taka sama siła, tyle że przeciwnie skierowana. Oddziaływanie grawitacyjne jest zawsze przyciągające (nie ma ujemnej masy). Podobnie, jak problemy rozważane w pierwszej części, również zagadnienie dwóch ciał można rozwiązać ściśle na kartce papieru. Wymaga to jednak sporego wysiłku. W przypadku trójwymiarowym, rozwiązaniem są tzw. krzywe stożkowe, czyli elipsa, parabola lub hiperbola. To są krzywe płaskie, co oznacza, że problem jest tak naprawdę dwuwymiarowy. I dlatego my, głównie ze względu na możliwość wizualizacji, ograniczymy się do dwóch wymiarów. Rozszerzenie obliczeń do trzech wymiarów jest jednak bardzo proste. Dwa ciała, dwa wymiary i równania drugiego stopnia – to oznacza, że tym razem będziemy mieli osiem równań różniczkowych pierwszego rzędu i wektor y będzie miał osiem elementów ($N = 8$). Zwróćmy uwagę na stałe widoczne na początku Listingu 8 – są to umowne jednostki odległości, masy i czasu wyrażone w jednostkach układu SI (tj. w metrach, sekundach, kilogramach itd.), odpowiadające skalom problemu. Jednostką odległości jest średnia odległość Ziemi od Księżyca, czyli 384403000 metrów (prawie czterysta tysięcy kilometrów), jednostką masy – masa Ziemi, czyli $5.9736 \cdot 10^{24}$ kilogramów, a jednostką czasu – dzień ziemski, czyli $24 \cdot 60 \cdot 60 = 86400$ sekund. Odpowiednie wartości znalazłem w Wikipedii. W obliczeniach, w których zależałoby nam na dokładności, te umowne jednostki powinny być równe jedności, aby wartości w wektorze stanu y były dalekie zarówno od minimalnej, jak i maksymalnej wartości liczby `double`. Należy wówczas pamiętać, aby w tych jednostkach wyrazić także wszystkie stałe fizyczne, np. stałą grawitacji G. Wyniki obliczeń zapisane byłyby oczywiście w tych jednostkach, ale z łatwością można je przeliczyć na jednostki układu SI. Jednak my na przekór tej dobrej praktyce użyjemy już w obliczeniach jednostek SI, pomimo że słabo przystają do zagadnień astronomicznych (zresztą tak jak i do atomowych).

Listing 8. Zagadnienie dwóch ciał. Ruch Ziemi i Księżyca w ich wzajemnym polu grawitacyjnym

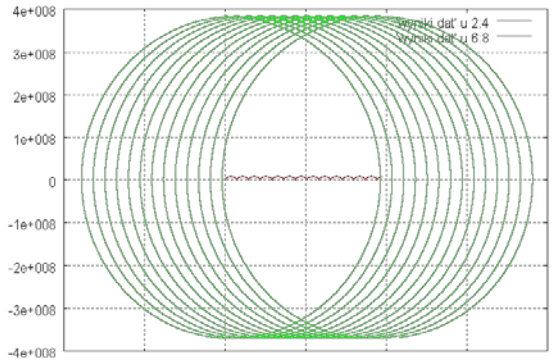
```
const int N = 8;

const double jednostkaOdleglosci = 384403000; //metrów
const double jednostkaMasy = 5.9736E24; //kilogramów
const double jednostkaCzasu = 24 * 60 * 60; //sekund

#define SQR(x) ((x)*(x))

double grawitacja2(int i, double* y, double t)
//zagadnienie 2 cial w 2D
{
  static const double G = 6.673848E-11; //[G] = m^3/kg/s^2
  static const double M = jednostkaMasy;
  static const double m = 0.0123*jednostkaMasy;

  double odleglosc_x = y[4] - y[0]; //(r12)_x
  double odleglosc_y = y[6] - y[2]; //(r12)_y
  double kwadrat_odleglosci = SQR(odleglosc_x) + SQR(odleglosc_y);
  //r12^2
  double odleglosc = sqrt(kwadrat_odleglosci); //r12
  double sila = G*M*m / kwadrat_odleglosci; //F

  double _x = odleglosc_x / odleglosc;
  double _y = odleglosc_y / odleglosc;

  switch (i)
  {
    //Ziemia
```

```
  case 0:
    return y[1];
  case 1:
    return sila*_x / M;
  case 2:
    return y[3];
  case 3:
    return sila*_y / M;

    //Ksiezyc
  case 4:
    return y[5];
  case 5:
    return -sila*_x / m;
  case 6:
    return y[7];
  case 7:
    return -sila*_y / m;
  default:
    throw runtime_error("Zły numer równania");
  }
}
```

Do numerycznego rozwiązania zagadnienia dwóch ciał możemy użyć metody RK4. Przeprowadźmy dla przykładu symulacje trwające jeden rok z krokiem równym jednej minucie. Bardzo ważne są dobrze dobrane warunki początkowe – tylko wówczas wyniki będą, przynajmniej mniej więcej, odpowiadały rzeczywistości, jaką znamy z własnego doświadczenia. Jak wspomniałem, ja wartości te wziąłem z Wikipedii. Początkowa prędkość Księżyca w ruchu wokół Ziemi równa jest 1022 m/s, czyli nieco ponad kilometr na sekundę. Natomiast początkowa odległość Ziemi i Księżyca równa jest ich średniej odległości. Te i pozostałe dane widoczne są na Listingu 9 zawierającym cały kod pliku *odeint.cpp*. Obliczenia będą trwały stosunkowo długo, ale wyniki będą sensowne (Rysunek 9). Wykonując kilka, siłą rzeczy dość długich eksperymentów, możemy znaleźć taką wartość kroku całkowania, który spełni nasze oczekiwania co do dokładności wyników, a jednocześnie nie będzie nadmiernie przedłużał obliczeń. Nasza sytuacja jest jednak dość wygodna – odległość między Ziemią i Księżycem jest w przybliżeniu stała, zatem dobrany na początku krok całkowania będzie także odpowiedni w trakcie całej symulacji. Jednak jeżeli wartości początkowe nie byłyby tak optymalnie dobrane i Księżyc poruszałby się po bardzo wydłużonej orbicie, w której duże przyspieszenia działałyby na niego tylko w krótkim przedziale czasu (takie orbity mają niektóre komety), to stały krok całkowania musiałby być dobrany dla tych krótkich, ale kluczowych okresów czasu, w których na ciała działają największe przyspieszenia. Jednocześnie w pozostałym czasie tak mały krok byłby niepotrzebny i tylko znacznie przedłużałby symulację.

Listing 9. Zawartość pliku odeint.cpp z metodą main

```
#include <iostream>
#include <fstream>
using namespace std;

#include "odeint.h"

int main()
{
  double* y = new double[N];
  double* y_nast = new double[N];

  //warunki początkowe
  for (int i = 0; i < N; ++i)
  {
    y[i] = 0;
    y_nast[i] = 0;
  }
  y[6] = -jednostkaOdleglosci;
  y[5] = 1.022E3;

  double tmax = 365 * jednostkaCzasu; //miesiac
  double h = 60; //minuta

  //plik
  ofstream plik_wy("wyniki.dat");
  plik_wy.precision(10);
  plik_wy.setf(ios::scientific);

  //ewolucja układu
  for (double t = 0; t < tmax; t += h)
  {
    odeint_RK4<double>(N, grawitacja2, y, t, h, y_nast);

    plik_wy << t;
    for (int i = 0; i < N; ++i) plik_wy << "\t" << y[i];
    plik_wy << "\t" << h;
    plik_wy << "\n";

    //zamiana tablic
    double* y_tmp = y;
    y = y_nast;
    y_nast = y_tmp;
  }

  plik_wy.close();

  cout << "\n\nOK.\n";

  delete[] y;
}
```



Rysunek 9. Tor ruchu Ziemi i Księżyca pod wpływem ich wzajemnego oddziaływania grawitacyjnego (na osi poziomej współrzędna x, na pionowej – y)

METODA RUNGE-KUTTY-FELDBERG RKF4(5)

Najlepszym rozwiązaniem tego problemu jest dopasowywanie długości kroku całkowania w trakcie symulacji. Jakiego jednak użyć kryterium? Można wziąć pod uwagę wartość sił działających w układzie, co przekłada się na odległość obu punktów. Niemiecki matematyk Erwin Fehlberg odkrył jednak w 1969 roku bardziej bezpośredni sposób. Jeżeli będziemy w każdym kroku, a przynajmniej raz na pewną liczbę kroków, liczyć ewolucję, stosując dwie metody Runge-Kutty o różnej dokładności, np. RK4 i RK5, to dzięki porównywaniu uzyskanych w ten sposób wyników będziemy mogli orzec, czy krok całkowania jest odpowiedni. Jeżeli różnica wyników z obu metod będzie mniejsza od zakładanej minimalnej wartości, krok można spokojnie wydłużyć. I odwrotnie, jeżeli różnica przekroczy zakładaną wartość maksymalną, krok należy zmniejszyć i powtórzyć obliczenia. W najprostszym podejściu krok może być zmniejszany o połowę lub zwiększany dwukrotnie. Wielką zasługą Fehlberga było znalezienie takich współczynników w metodach RK4 i RK5, których wartości były takie same. Dzięki temu, nie zwiększając kosztu obliczeń (jest on tylko nieco większy niż koszt RK5), możemy w każdym kroku weryfikować dokładność wyników i w elastyczny sposób dobierać krok całkowania. Ta metoda nazywa się metodą Runge-Kutty-Fehlberga 4(5) (RKF4(5)) i została zaimplementowana w funkcji `odeint_RK4F5` widocznej na Listingu 10.

Listing 10. Implementacja metody RKF4(5) w pliku odeint.h

```
const double blad_min = 1E-2;
const double blad_max = 1E1;

template<typename T>
T* odeint_RK4F5(int N, T(*f)(int, T*, T), T* y, T t, T& h, T* y_nast)
{
  static const T b31 = 3.0 / 32.0;
  static const T b32 = 9.0 / 32.0;
  static const T a3 = 3.0 / 8.0;

  static const T b41 = 1932.0 / 2197.0;
```

```
  static const T b42 = -7200.0 / 2197.0;
  static const T b43 = 7296.0 / 2197.0;
  static const T a4 = 12.0 / 13.0;

  static const T b51 = 439.0 / 216.0;
  static const T b53 = 3680.0 / 513.0;
  static const T b54 = -845.0 / 4104.0;

  static const T w41 = 25.0 / 216.0;
  static const T w43 = 1408.0 / 2565.0;
  static const T w44 = 2197.0 / 4104.0;

  static const T b61 = -8.0 / 27.0;
  static const T b63 = -3544.0 / 2565.0;
  static const T b64 = 1859.0 / 4104.0;
  static const T b65 = -11.0 / 40.0;

  static const T w51 = 16.0 / 135.0;
  static const T w53 = 6656.0 / 12825.0;
  static const T w54 = 28561.0 / 56430.0;
  static const T w55 = -9.0 / 50.0;
  static const T w56 = 2.0 / 55.0;

  T* y_tmp = new T[N];
  T* y4_nast = new T[N];
  T* k1 = new T[N];
  T* k2 = new T[N];
  T* k3 = new T[N];
  T* k4 = new T[N];
  T* k5 = new T[N];
  T* k6 = new T[N];

  T blad = 0;
  do
  {
    for (int i = 0; i < N; ++i)
    {
      k1[i] = h*f(i, y, t);
      y_tmp[i] = y[i] + 0.25*k1[i];
    }
    for (int i = 0; i < N; ++i)
    {
      k2[i] = h*f(i, y_tmp, t + 0.25*h);
      y_nast[i] = y[i] + b31*k1[i] + b32*k2[i];
    }
    for (int i = 0; i < N; ++i)
    {
      k3[i] = h*f(i, y_nast, t + a3*h);
      y_tmp[i] = y[i] + b41*k1[i] + b42*k2[i] + b43*k3[i];
    }
    for (int i = 0; i < N; ++i)
    {
      k4[i] = h*f(i, y_tmp, t + a4*h);
      y_nast[i] = y[i] + b51*k1[i] - 8.0*k2[i] + b53*k3[i] + b54*k4[i];
    }
    for (int i = 0; i < N; ++i)
    {
      k5[i] = h*f(i, y_nast, t + h);
      //RK4 na 5 wyrazach
      y4_nast[i] = y[i] + w41*k1[i] + w43*k3[i] + w44*k4[i] - 0.2*k5[i];
      y_tmp[i] = y[i] + b61 * k1[i] + 2 * k2[i] + b63*k3[i] +
      b64*k4[i] + b65*k5[i];
    }
    for (int i = 0; i < N; ++i)
    {
      k6[i] = h*f(i, y_tmp, t + 0.5*h);
      //RK5 na 6 wyrazach
      y_nast[i] = y[i] + w51*k1[i] + w53*k3[i] +
        w54*k4[i] + w55*k5[i] + w56*k6[i];
      blad += fabs(y_nast[i] - y4_nast[i]);
      blad /= N;
    }

    if (blad < blad_min) h *= 2;
    if (blad > blad_max) h /= 2;
  } while (blad<blad_min || blad>blad_max);

  delete[] y_tmp;
  delete[] y4_nast;
  delete[] k1;
  delete[] k2;
  delete[] k3;
  delete[] k4;
  delete[] k5;
  delete[] k6;

  return y_nast;
}
```

W Listingu 10 należy zwrócić uwagę na dwie rzeczy. Pierwszą jest zmiana sygnatury funkcji `odeint_RK4F5` w porównaniu do poprzednich, np. `odeint_RK4`. Krok (argument h), który może być wewnątrz tej funkcji zmieniany, przekazywany jest teraz przez referencję. Drugą rzeczą jest algorytm zmiany kroku użyty w tej funkcji. Zgodnie z wcześniejszym opisem zwiększamy go dwukrotnie, gdy różnica wyników RK4 i RK5 jest mniejsza od zakładanej, i zmniejszamy o połowę, gdy ten błąd zbytnio urośnie. To nie jest najlepsza metoda, bo oba progi należy dopasowywać do problemu i warunków początkowych, nie wspominając o tym, że błąd gromadzi zarówno różnice położeń, jak i prędkości, które mogą być różnych rzędów. Potraktujmy to jednak jako tymczasowe rozwiązanie. Obliczenia dla konkretnego kroku powtarzane są tak długo, jak długo różnica wyników dla RK4 i RK5 nie mieści się w wyznaczonym zakresie. Do tego służy dodana do funkcji `odeint_RK4F5` pętla `do..while`. Jeżeli ten zakres będzie źle dobrany, może zdarzyć się sytuacja, w której dwukrotne zwiększenie kroku prowadzi do błędu większego od maksymalnej wartości, a podział kroku – do zmniejszenia tej wartości poniżej minimalnej wartości. To oznacza zapętlenie symulacji w tym kroku.

Mimo zmiany sygnatury, sposób użycia nowego solvera nie zmienia się:

```
odeint_RK4F5<double>(N, grawitacja3, y, t, h, y_nast);
```

Należy jednak pamiętać, że nowa metoda może zmieniać krok całkowania. Ponieważ jest on zapisywany w ostatniej kolumnie pliku wynikowego, z łatwością można sprawdzić, że w pierwszym kroku został on zwiększony z 60s do 61440s i taki już pozostał do końca symulacji. To nic dziwnego, skoro odległość obu obiektów nie zmienia się zbytnio, zatem działają na nie przez cały czas mniej więcej stałe co do wartości siły. Wartość kroku zależy oczywiście od zakładanego poziomu zgodności rozwiązań uzyskanych za pomocą RK4 i RK5.

PRZEJŚCIE DO UKŁADU ŚRODKA MASY

Warunki początkowe użyte w symulacji, której wyniki widać na Rysunku 9, a mianowicie Ziemia stojąca w początku układu współrzędnych i Księżyc poruszający się w pewnej odległości od Ziemi, powodują, że cały układ powoli przesuwa się w kierunku dodatnich wartości współrzędnej *x*. Obliczenia tego typu zwykle prowadzi się w układzie środka masy, w którym takiego dryfu nie widać. Zmiana układu odniesienia nie zmusza do żadnych zmian w sposobie prowadzenia obliczeń lub funkcji opisujących rozwiązywane układy równań; to jedynie kwestia warunków początkowych. Przejścia do układu środka masy dokonamy w dwóch krokach: najpierw zmienimy prędkości początkowe, co wyeliminuje dryf, a następnie zmodyfikujemy początkowe położenia w taki sposób, aby oba ciała poruszały się wokół początku układu współrzędnych. Aby zmienić prędkości początkowe, wprowadźmy do funkcji `main` w pliku *odeint.cpp* zmiany widoczne na Listingu 11. Prędkości obu ciał zmieniamy o prędkość będącą sumą ważoną prędkości obu ciał, przy czym wagami tej sumy są masy obu ciał. Efekt tej modyfikacji widoczny jest na Rysunku 10. Dryf całego układu został wyeliminowany. W bardzo podobny sposób zmienione powinny zostać położenia początkowe obu ciał (Rysunek 11). Do kodu z Listingu 11 dodajemy zatem fragment widoczny na Listingu 12.

Listing 11. Zmiana prędkości początkowych związana z przejściem do układu środka masy

```
//warunki początkowe
for (int i = 0; i < N; ++i)
{
  y[i] = 0;
  y_nast[i] = 0;
}
y[6] = -jednostkaOdleglosci;
y[5] = 1.022E3;

static const double M = jednostkaMasy;
static const double m = 0.0123*jednostkaMasy;

//zmiana prędkości początkowych
double Vsm_x = (y[1] * M + y[5] * m) / (M + m);
double Vsm_y = (y[3] * M + y[7] * m) / (M + m);
y[1] -= Vsm_x;
y[5] -= Vsm_x;
y[3] -= Vsm_y;
y[7] -= Vsm_y;
```

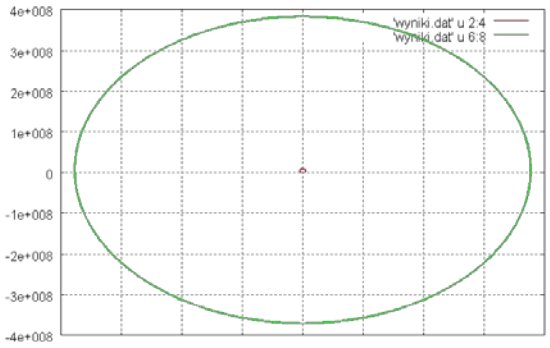
KEYLIGHT

KURSY MULTIMEDIALNE

Modelowanie gada z
BLENDER'em
[LOW POLY]

WWW.KEYLIGHT.COM.PL

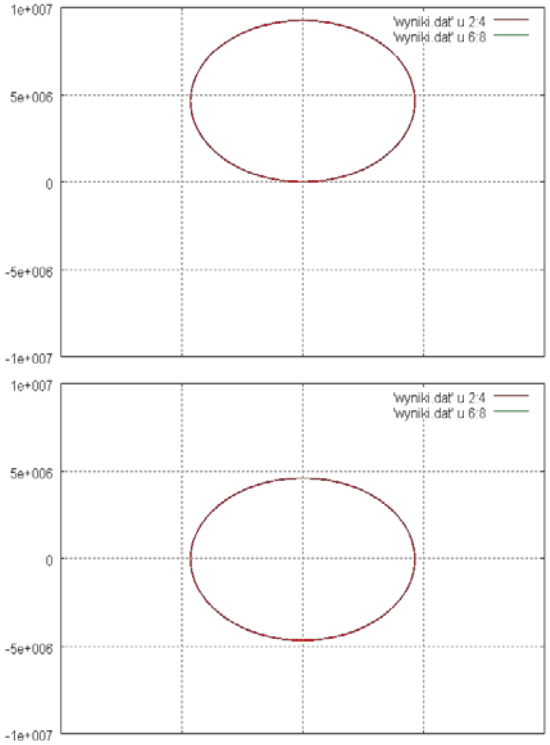
Modelowanie humanoida z
BLENDER'em
[LOW POLY]



Rysunek 10. Tor Ziemi i Księżyca po zmianie prędkości początkowych (por. z Rysunkiem 9)

Listing 12. Zmiana położeń początkowych

```
double Rsm_x = (y[0] * M + y[4] * m) / (M + m);
double Rsm_y = (y[2] * M + y[6] * m) / (M + m);
y[0] -= Rsm_x;
y[4] -= Rsm_x;
y[2] -= Rsm_y;
y[6] -= Rsm_y;
```



Rysunek 11. Tor Ziemi przed i po zmianie położeń początkowych

ZAGADNIENIE TRZECH CIAŁ (ZIEMIA-KSIĘŻYC-RAKIETA)

Na koniec przedstawię zagadnienie, które w pełni zaprezentuje zalety nowej metody: zagadnienie trzech ciał w dwóch wymiarach, czyli zagadnienie ruchu trzech ciał, gdy działają na nie tylko siły grawitacyjne pochodzące od dwóch pozostałych ciał. Funkcja opisująca ten problem widoczna jest na Listingu 13. W ogólności zagadnienia tego nie można rozwiązać na kartce. Można szukać przybliżeń tylko w pewnych szczególnych przypadkach, w których np. jedna z mas dominuje nad pozostałymi. W naszym przykładzie pierwsze i drugie ciało pozostanie Ziemią i Księżycem. Trzecie ciało będzie natomiast rakietą wystrzeloną z Ziemi. Zauważmy, że siły, jakie będą działać na rakietę, będą bardzo zmieniać się w miarę oddalania od Ziemi i później będą silnie zależeć od odległości od Ziemi i Księżyca. To powinno prowadzić do częstych zmian kroków całkowania. Bez możliwości adaptowania kroku do bieżącego stanu układu, obliczenia musiałyby być prowadzone asekuracyjnie dla bardzo małego kroku, co by je niemiłosiernie wydłużało.

Listing 13. Funkcja implementująca zagadnienie trzech ciał (odeint.h)

```
const int N = 12;

double grawitacja3(int i, double* y, double t) //zagadnienie 3
ciał w 2D
{
  static const double G = 6.6742867E-11;
  static const double mZ = jednostkaMasy;
  static const double mK = 0.0123*jednostkaMasy;
  static const double mR = 1E-18*jednostkaMasy;

  //Ziemia-Ksiezyc
  double odleglosc_x_ZK = y[4] - y[0]; //(r2-r1)x
  double odleglosc_y_ZK = y[6] - y[2]; //(r2-r1)y
  double kwadrat_odleglosci_ZK = SQR(odleglosc_x_ZK) +
  SQR(odleglosc_y_ZK); //r12^2
  double odleglosc_ZK = sqrt(kwadrat_odleglosci_ZK); //r12
  double sila_ZK = G*mZ*mK / kwadrat_odleglosci_ZK; //F
  double _x_ZK = odleglosc_x_ZK / odleglosc_ZK;
  double _y_ZK = odleglosc_y_ZK / odleglosc_ZK;

  //Ziemia-rakieta
  double odleglosc_x_ZR = y[8] - y[0]; //(r2-r1)x
  double odleglosc_y_ZR = y[10] - y[2]; //(r2-r1)y
  double kwadrat_odleglosci_ZR = SQR(odleglosc_x_ZR) +
  SQR(odleglosc_y_ZR); //r12^2
  double odleglosc_ZR = sqrt(kwadrat_odleglosci_ZR); //r12
  double sila_ZR = G*mZ*mR / kwadrat_odleglosci_ZR; //F
  double _x_ZR = odleglosc_x_ZR / odleglosc_ZR;
  double _y_ZR = odleglosc_y_ZR / odleglosc_ZR;

  //Ksiezyc-rakieta
  double odleglosc_x_KR = y[8] - y[4]; //(r2-r1)x
  double odleglosc_y_KR = y[10] - y[6]; //(r2-r1)y
  double kwadrat_odleglosci_KR = SQR(odleglosc_x_KR) +
  SQR(odleglosc_y_KR); //r12^2
  double odleglosc_KR = sqrt(kwadrat_odleglosci_KR); //r12
  double sila_KR = G*mK*mR / kwadrat_odleglosci_KR; //F
  double _x_KR = odleglosc_x_KR / odleglosc_KR;
  double _y_KR = odleglosc_y_KR / odleglosc_KR;

  double przyspieszenieCiaguRakiety = 0.1;

  const bool wplywRakiety = 0; //0 lub 1

  double wynik = 0;
  switch (i)
  {
    //Ziemia
  case 0:
    wynik = y[1];
    break;
  case 1:
    wynik = (sila_ZK*_x_ZK + (wplywRakiety ? 1 : 0)*sila_ZR*_x_ZR) / mZ;
    break;
  case 2:
    wynik = y[3];
    break;
  case 3:
    wynik = (sila_ZK*_y_ZK + (wplywRakiety ? 1 : 0)*sila_ZR*_y_ZR) / mZ;
    break;
    //Ksiezyc
  case 4:
    wynik = y[5];
    break;
  case 5:
    wynik = (-sila_ZK*_x_ZK + (wplywRakiety ? 1 : 0)*sila_KR*_x_KR) / mK;
    break;
  case 6:
    wynik = y[7];
    break;
  case 7:
    wynik = (-sila_ZK*_y_ZK + (wplywRakiety ? 1 : 0)*sila_KR*_y_KR) / mK;
    break;
    //rakieta
  case 8:
    wynik = y[9];
    break;
  case 9:
```

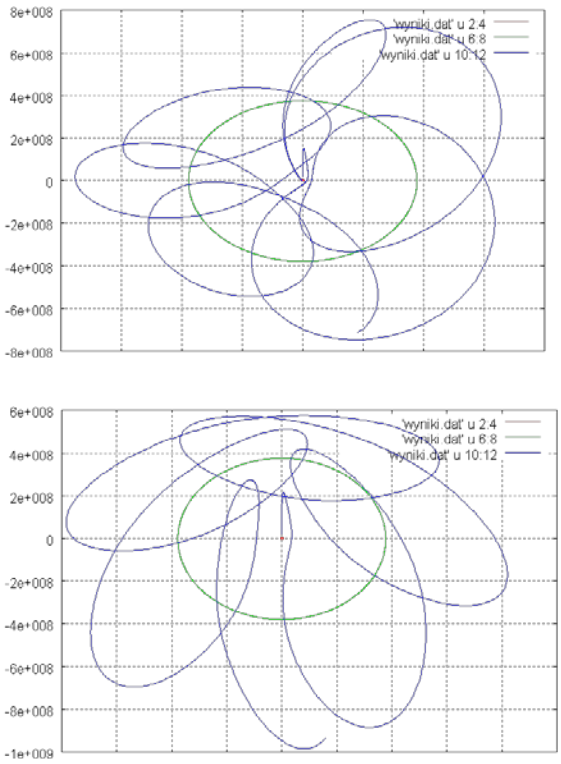
```
    wynik = (-sila_ZR*_x_ZR - sila_KR*_x_KR) / mR;
    wynik += (-_x_KR)*przyspieszenieCiaguRakiety;
    break;
  case 10:
    wynik = y[11];
    break;
  case 11:
    wynik = (-sila_ZR*_y_ZR - sila_KR*_y_KR) / mR;
    wynik += (-_y_KR)*przyspieszenieCiaguRakiety;
    break;
  default:
    throw runtime_error("Zły numer równania");
  }

  return wynik;
}
```

W funkcji `main` w pliku *odeint.cpp* musimy uzupełnić warunki początkowe (trzeba dodać położenie i prędkość rakiety), należy też zmienić czas i krok całkowania oraz oczywiście funkcję przekazywaną do funkcji całkującej. Zmienione linie widoczne są na Listingu 14. Jak widać z tego listingu pozostajemy w układzie środka masy Ziemi i Księżyca (masa rakiety jest w tym aspekcie pomijalna). Muszę też uczciwie zwrócić uwagę czytelnika na pewne poważne oszustwo. Prawdziwa rakieta spala paliwo, co powoduje, że zmniejsza swoją masę. Ponadto zazwyczaj rakiety są wieloczłonowe; poszczególne człony rakiety są odrzucane zanim wyleci ona poza atmosferę. Nasza rakieta ma natomiast stałą masę równą około $6 \cdot 10^6$ kg = 6 mln kg. Masa ta zgadza się co do rzędu wielkości z masami rzeczywistych rakiet. Dla przykładu masa rakiety Saturn V była równa 3 mln kg. Stały ciąg naszej rakiety powoduje, że ma ona stałe przyspieszenie równe 0.1 m/s działające w kierunku, w którym rakieta się aktualnie porusza. Prędkość początkowa nieco przewyższa pierwszą prędkość kosmiczną, czyli prędkość, jaką musiałby mieć pocisk wystrzelony z Ziemi (bez własnego ciągu), aby wejść na orbitę wokół Ziemi.

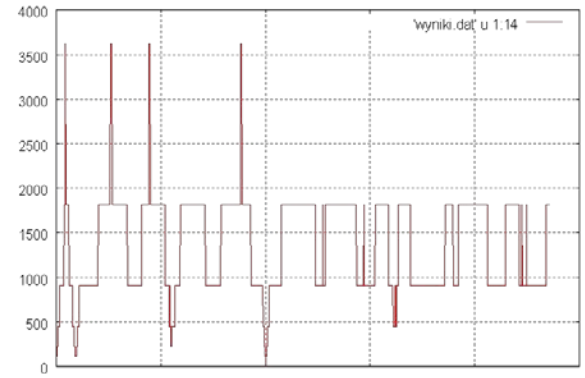
Listing 14. Zmiany w metodzie main

```
#include <iostream> //cout
#include <fstream> //pliki
using name space std;

#include "odeint.h"

int main()
{
  double* y = new double[N];
  double* y_nast = new double[N];

  //warunki początkowe
  for (int i = 0; i < N; ++i)
  {
    y[i] = 0;
    y_nast[i] = 0;
  }
  y[6] = -jednostkaOdleglosci;
  y[5] = 1.022E3;
  y[10] = 0.05*jednostkaOdleglosci;
  y[11] = 1.1*7.91E3; //110% pierwszej prędkości kosmicznej

  //przejście do układu środka masy
  ...

  double tmax = 27.3*jednostkaCzasu; //miesiac
  double h = 1E-8*jednostkaCzasu; //krok początkowy

  //plik
  ofstream plik_wy("wyniki.dat");
  plik_wy.precision(10);
  plik_wy.setf(ios::scientific);

  //ewolucja układu
  for (double t = 0; t < tmax; t += h)
  {
    odeint_RK4F5<double>(N, grawitacja3, y, t, h, y_nast);

    plik_wy << t;
    for (int i = 0; i < N; ++i) plik_wy << "\t" << y[i];
    plik_wy << "\t" << h;
    plik_wy << "\n";

    //zamiana tablic
    double* y_tmp = y;
    y = y_nast;
    y_nast = y_tmp;
  }

  plik_wy.close();

  cout << "\n\nOK.\n";

  delete[] y;
}
```

Wyniki są bardzo ciekawe (Rysunek 12) – rakieta lata po przestrzeni wokół Ziemi i Księżyca, jakby występowała w animowanym filmie dla dzieci, kręcąc pętle i fikołki. Co ciekawe wystarczy naprawdę niewielka zmiana warunków początkowych rakiety, aby jej tor wyglądał całkowicie inaczej. To sugeruje zachowanie chaotyczne. I rzeczywiście zagadnienie trzech ciał było badane w XIX wieku przez francuskiego matematyka i fizyka Henri Poincarégo (to jeden z tych naukowców w binoklach na sznurku przyczepionym do kieszonki, z wąsami i długą brodą) i dało początek teorii chaosu deterministycznego.



Rysunek 12. Tor ruchu rakiety dla dwóch różnych prędkości początkowych (110% i 120% pierwszej prędkości kosmicznej)



Rysunek 13. Zmiany kroku całkowania w funkcji czasu

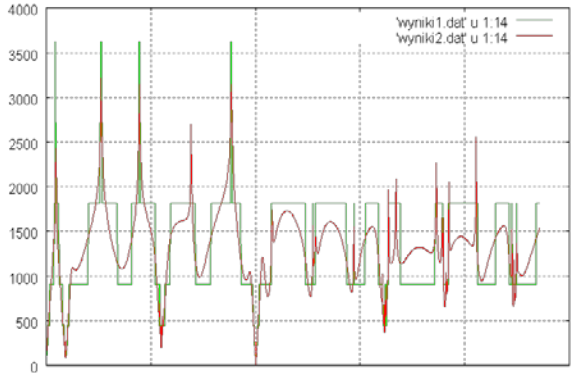
PŁYNNE DOBIERANIE KROKU CAŁKOWANIA

Zmieńmy prędkość początkową rakiety na 111% pierwszej prędkości kosmicznej (tylko 1% różnicy). Gdy uruchomimy symulację, ta zamiast szybko się skończyć, będzie trwała w nieskończoność, nie zapisując żadnych danych do pliku. Obliczenia niestety zapętlą się ze względu na opisany wyżej problem przy użytym sposobie zmiany kroku czasowego. Zastosujmy wobec tego sprytniejszą metodę pozwalającą na płynne dobieranie kroku, zmieniając krok według wzoru:

$$h_{nowy} = h_{poprzedni} \cdot 0.1 \cdot (1/\varepsilon)^{1/5}.$$

Symbol ε oznacza różnicę wyników uzyskanych za pomocą RK4 i RK5, czyli wartość zmiennej `blad` w metodzie `odeint_RK4F5`. Korzystając z tego podejścia, nie musimy ustalać zakresu błędu (stałe `blad_min` i `blad_max`), ale i tak warto tę wartość w jakiś sposób kontrolować. Ja zwykle pilnuję, czy jego wartość nie przekracza wartości maksymalnej. Aby zaimplementować płynne dobieranie kroku, należy zatem zmienić fragment funkcji `odeint_RK4F5` w pętli `do..while` oraz warunek jej przerwania (por. Rysunki 13 i 14):

```
T blad = 0;
do
{
  ...

  h *= 0.1*pow(1 / blad, 0.2);
} while (blad>blad_max);
```



Rysunek 14. Zmiana kroku nową metodą (czerwona linia) i starą metodą (zielona linia)

A zatem, jeżeli w kolejnym projekcie czytelnik napotka jakiekolwiek równanie lub zbiór równań różniczkowych zwyczajnych, należy rozłożyć je na zbiór takich równań pierwszego stopnia i zastosować jedną z metod przedstawionych w tym artykule. Jeżeli równania nie są zbyt „dzikie", możemy zastosować metodę Runge-Kutty 4, z odpowiednio dobranym krokiem całkowania. Natomiast jeżeli mamy przeczucie, że stały krok całkowania nie jest dobrym podejściem, należy zastosować metodę Runge-Kutty-Fehlberg 4(5), upewniając się jednak, że wyniki uzyskane tą metodą są rozsądne – metoda RKF4(5) może bowiem sprawiać przykre niespodzianki – warto być na nie gotowym i nie stosować jej bezkrytycznie.

Serdecznie dziękuję prof. Janowi Wasilewskiemu, który kilka lat temu pokazał mi metodę RKF4(5) i zagadnienie trzech ciał z rakietą jako świetny przykład wymagający jej użycia.

Jacek Matulewski

Fizyk (optyk kwantowy) na Wydziale Fizyki, Astronomii i Informatyki Stosowanej. Od 1998 r. zajmuje się także tworzeniem oprogramowania, w szczególności z użyciem platformy .NET i języka C#. Autor serii książek poświęconych programowaniu. Obecnie kieruje Pracownią Gier Terapeutycznych i Badania Procesów Poznawczych w Interdyscyplinarnym Centrum Nowoczesnych Technologii UMK w Toruniu.