MOwNiT2 Lab9

Zmiany w implementacjach z lab8 dla dowolnego układu równań (klasa implementująca interfejs IODE):

```
class IODE
  {
 public:
   virtual ~IODE() = default;

 public:
   virtual std::vector<double> getDerivatives(std::vector<double> values)
       = 0;

};

class LorenzSystem : public IODE
{
 public:
   LorenzSystem(double rho, double sigma, double beta) :
     rho(rho), sigma(sigma), beta(beta) {}

 private:
   std::vector<double> getDerivatives(std::vector<double> values){
     std::vector<double> result;
     result.push_back(sigma * values[1] - sigma * values[0]);
     result.push_back(-values[0] * values[2] + rho * values[0] - values[1]);
     result.push_back(values[0] * values[1] - beta * values[2]);
     return result;
   }

 private:
   double rho;
   double sigma;
   double beta;
};
```

Wszystkie metody zawarte w klasie ODESolver::Lab8:

```
namespace ODESolver
{

  class Lab8
  {
    public:
      Lab8(IODE& ode, double dt, int max_steps) :
        ode(ode), dt(dt), max_steps(max_steps)  {}

      // lab8 methods...

      // for backward Euler method
    private:
      bool isClose(std::vector<double> v1, std::vector<double> v2,
                   double err){

        for(int i=0; i<v1.size(); i++){
          if(abs(v1[i]-v2[i]) >= err) return false;
        }
        return true;
      }

    private:
      IODE& ode;
      double dt;
      int max_steps;
  };
}
```

Metoda Eulera:

```
void eulerMethod(std::vector<double> values){

  int step = 1;
  std::vector<double> result;
  std::ofstream outfile("explicit_euler.csv");

  while(step <= max_steps){

    result = ode.getDerivatives(values);

    for(int i=0; i<result.size(); i++){
      values[i] += result[i] * dt;
    }

    for(int i=0; i<values.size(); i++){
      outfile << values[i];
      if(i != values.size()-1) outfile << ",";
      else outfile << std::endl;
    }

    step++;
  }

}
```

Backward Euler:

```
void backwardEulerMethod(std::vector<double> values){

  double err = 1e-6;
  int step = 1;
  std::vector<double> result, values0(values.size()),
      values1(values.size());

  std::ofstream outfile("implicit_euler.csv");

  while(step <= max_steps){

    result = ode.getDerivatives(values);
    for(int i=0; i<result.size(); i++){
      values0[i] = values0[i] + result[i] * dt;
    }

    result = ode.getDerivatives(values0);
    for(int i=0; i<result.size(); i++){
      values1[i] = values1[i] + result[i] * dt;
    }

    while(!isClose(values0, values1, err)){

      values0 = values1;

      result = ode.getDerivatives(values0);
      for(int i=0; i<result.size(); i++){
        values1[i] = values[0] + result[i] * dt;
      }
    }

    values = values1;

    for(int i=0; i<values.size(); i++){
      outfile << values[i];
      if(i != values.size()-1) outfile << ",";
      else outfile << std::endl;
    }

    step++;
  }

}
```

Midpoint method:

```
void midpointMethod(std::vector<double> values){

  int step = 1;
  std::vector<double> result, tmp(values.size());

  std::ofstream outfile("midpoint.csv");

  while(step <= max_steps){

    result = ode.getDerivatives(values);
    for(int i=0; i<result.size(); i++){
      tmp[i] = values[i] + result[i] * dt/2;
    }

    result = ode.getDerivatives(values);
    for(int i=0; i<result.size(); i++){
      values[i] = values[i] + result[i] * dt;
    }

    for(int i=0; i<values.size(); i++){
      outfile << values[i];
      if(i != values.size()-1) outfile << ",";
      else outfile << std::endl;
    }

    step++;
  }

}
```

Runge-Kutta method:

```
void rungeKuttaMethod(std::vector<double> values){

  int step = 1;
  std::vector<double> k1, k2, k3, k4, tmp(values.size());

  std::ofstream outfile("runge_kutta.csv");

  while(step <= max_steps){

    k1 = ode.getDerivatives(values);
    for(int i=0; i<k1.size(); i++){
      k1[i] *= dt;
      tmp[i] = values[i] + k1[i]/2;
    }

    k2 = ode.getDerivatives(tmp);
    for(int i=0; i<k2.size(); i++){
      k2[i] *= dt;
      tmp[i] = values[i] + k2[i]/2;
    }

    k3 = ode.getDerivatives(tmp);
    for(int i=0; i<k3.size(); i++){
      k3[i] *= dt;
      tmp[i] = values[i] + k3[i];
    }

    k4 = ode.getDerivatives(tmp);
    for(int i=0; i<k4.size(); i++){
      k4[i] *= dt;
    }

    for(int i=0; i<values.size(); i++){
      values[i] = values[i] + (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])/6;
    }

    for(int i=0; i<values.size(); i++){
      outfile << values[i];
      if(i != values.size()-1) outfile << ",";
      else outfile << std::endl;
    }

    step++;
  }

}
```

Zadanie 1 Proszę na podstawie informacji nabytych w zadaniu 5 poprzedniego laboratorium zaimplementować algorytm Verleta (leap-frog). Może być to zrealizowane w postaci zwyczajnej funkcji, która przyjmuje stosowne parametry.

Algorytm Verleta w wersji zsynchronizowanej:

```
class Leapfrog
{
  public:
    Leapfrog(IODE& ode, double dt, int max_steps) :
        ode(ode), dt(dt), max_steps(max_steps)  {}

    void calculate(std::vector<double> x, std::vector<double> v){

      int step = 1;
      std::vector<double> a = ode.getDerivatives(x);
      std::vector<double> tmp(a.size());
      std::ofstream outfile("leapfrog.csv");

      while(step <= max_steps){

        for(int i=0; i<x.size(); i++){
          x[i] = x[i] + v[i] * dt + (a[i] * dt * dt)/2;
        }

        tmp = a;
        a = ode.getDerivatives(x);

        for(int i=0; i<v.size(); i++){
          v[i] = v[i] + (tmp[i] + a[i])*dt/2;
        }

        // format t,x1,v1,a1,x2,v2,a2...
        outfile << dt*step << ",";
        for(int i=0; i<x.size(); i++){
            outfile << x[i] << "," << v[i] << "," << a[i];
            if(i != x.size()-1) outfile << ",";
            else outfile << std::endl;
          }

        step++;
      }
  }
  private:
    IODE& ode;
    double dt;
    int max_steps;
};
```

Zadanie 2-5 Proszę z zamieszczonego w treści zadań pdf-a wybrać dowolne 4 z 6 zadań dot. symulacji komputerowych prostych układów fizycznych. Wykorzystując implementacje z Lab8 proszę rozwiązać te problemy oraz przedstawić STOSOWNE opracowanie.

A. Wahadło matematyczne:

Za wahadło matematyczne uznajemy punkt materialny (pomijalna masa i objętość) zawieszony na nici. Wyprawiając wahadło z pozycji równowagi możemy zauważyć sinusoidalną zależność wychylenia od czasu.
Często dla małych kątów stosuje się przybliżenie $sin(x) = x$, które sprowadza równanie do równania oscylatora harmonicznego (tutaj nie korzystamy z tego uproszczenia).

Równanie ruchu:

$$\frac{d^2x}{dt^2} + \frac{g}{l}sin(x) = 0$$

gdzie: g - przyspieszenie ziemskie, l - długość wahadła

Sprowadzenie do postaci wygodnej dla algorytmu:

$$\frac{d^2x}{dt^2} = -\frac{g}{l}sin(x)$$

Dla równania w takiej postaci mogę już użyć algorytmu Verleta.

Klasa reprezentująca równanie:

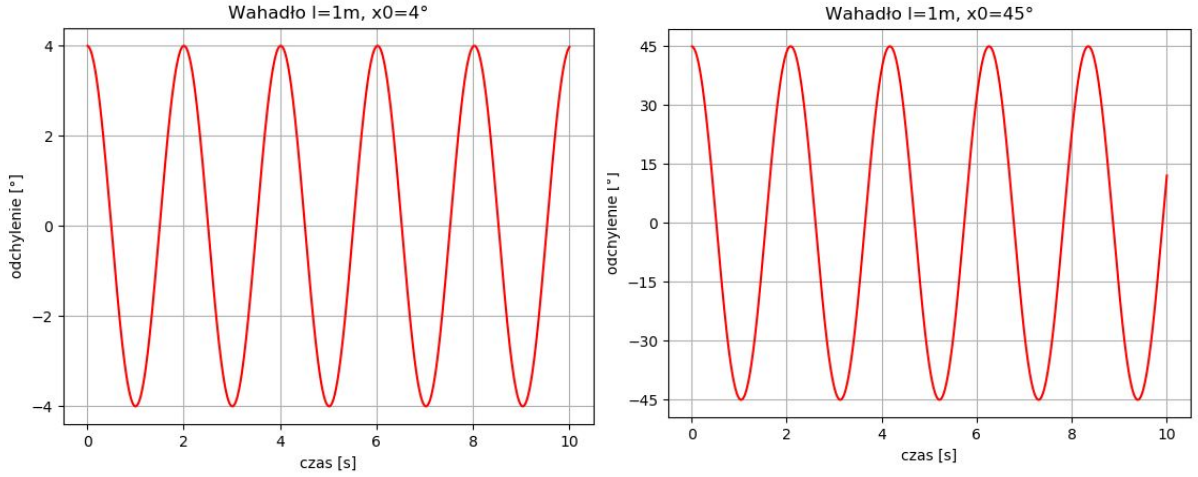
```
class SimplePendulum : public IODE
{
 public:
   SimplePendulum(double l) : l(l) {}

 private:
   std::vector<double> getDerivatives(std::vector<double> values){
     std::vector<double> result;
     result.push_back(-9.80665/l * sin(values[0]));
     return result;
   }

 private:
   double l;
};
```

Do symulacji ustalmy długość wahadła $l = 1m$.
Algorytm Verleta wymaga podania początkowej wartości wychylenia i prędkości.
Jako że w symulacji wychylamy wahadło i je puszczamy $v_0 = 0$.
Przetestujemy wahadło dla dwóch wartości początkowych wychylenia: 4° i 45°.

Do algorytmu podajemy wartości w radianach, dla czytelności na wykresie przedstawimy wartości w stopniach.



Niezależnie od wartości początkowej otrzymaliśmy wykresy sinusoidalne, czego oczekiwaliśmy, ponieważ równanie nie zawiera żadnych tłumień. Dodatkowo można zauważyć, że te wahadła mają różne okresy drgań (mniej szczytów na drugim wykresie), co oznacza że wahadło matematyczne jest oscylatorem anharmonicznym - okres drgań jest zależny od amplitudy.

B. Sprężyna:

Tutaj mamy jakąś masę m przyczepioną do sprężyny, leżącej w poziomie na równej powierzchni, którą wychylamy z położenia równowagi o x. Tutaj również pomijamy wszelkie tłumienia, więc spodziewamy się podobnego wykresu co w poprzednim przypadku.

Wychodząc z równania siły, która działa na ciało wychylone na sprężynie:

$$F = -kx$$

$$ma = -kx$$

gdzie m - masa ciała, k - współczynnik sprężystości sprężyny
Sprowadzenie do postaci wygodnej dla algorytmu:

$$a = \frac{d^2x}{dt^2} = -\frac{k}{m}x$$

Tutaj również skorzystamy z algorytmu Verleta.

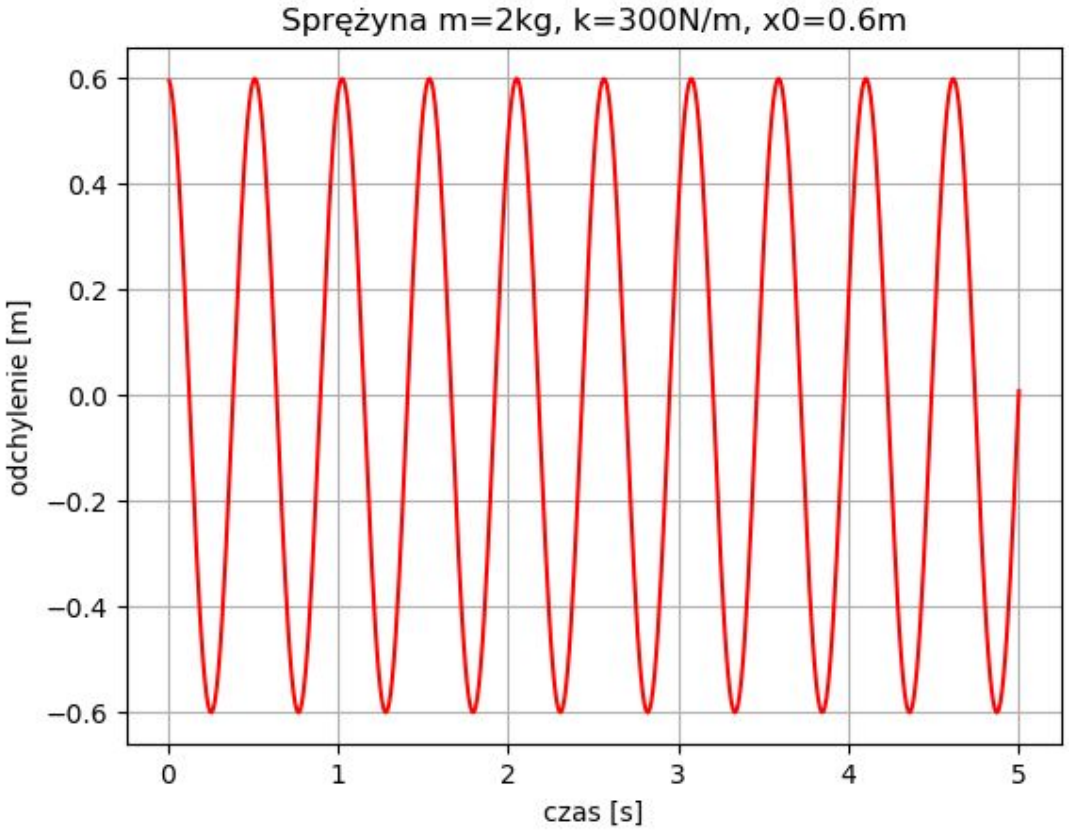
Reprezentacja równania:

```
class Spring : public IODE
{
 public:
   Spring(double m, double k) :
   m(m), k(k) {}

 private:
   std::vector<double> getDerivatives(std::vector<double> values){
     std::vector<double> result;
     result.push_back(-k * values[0] / m);
     return result;
   }

 private:
   double m;
   double k;
};
```

Przyjmijmy, że na sprężynie wisi ciało o masie $m = 2kg$, a współczynnik sprężystości sprężyny wynosi $k = 300N/m$.
W symulacji rozciągamy sprężynę i ją puszczamy, więc ponownie: $v_0 = 0$, a jako wychylenie początkowe (o ile rozciągnęliśmy sprężynę) ustalmy $x_0 = 0.6m$.



Jako że w modelu nie uwzględniamy np. siły tarcia to ponownie otrzymujemy ponownie wykres sinusoidalny czego oczekiwaliśmy. Sprężyna wracając do położenia równowagi rozpędza się wystarczająco, żeby przebyć odległość równą początkowemu wychyleniu w drugą stronę od położenia równowagi, gdzie proces się powtarza.

C. Model Lotki-Volterry:

Przyjmijmy:

x(t)- populacja drapieżników

y(t) -populacja ofiar

Wówczas:

$$\frac{dx}{dt} = -b * x + a * b * x * y$$

$$\frac{dy}{dt} = c * y - d * x * y$$

Gdzie:

a- częstość narodzin drapieżników

b- częstość umierania drapieżników

c - częstość narodzin ofiar

d- częstość umierania ofiar na skutek drapieżnictwa

Na logikę spodziewamy się naprzemiennie obserwować wzrosty i spadki obu wartości:

- dla drapieżników: zależnie od tego czy jest wystarczająco dużo pożywienia (ofiar) dla obecnej populacji
- dla ofiar: zależnie od tego czy ile drapieżników istnieje obecnie i poluje na nie

Skorzystamy z podanych równań i klasycznej metody Rungego-Kutty.

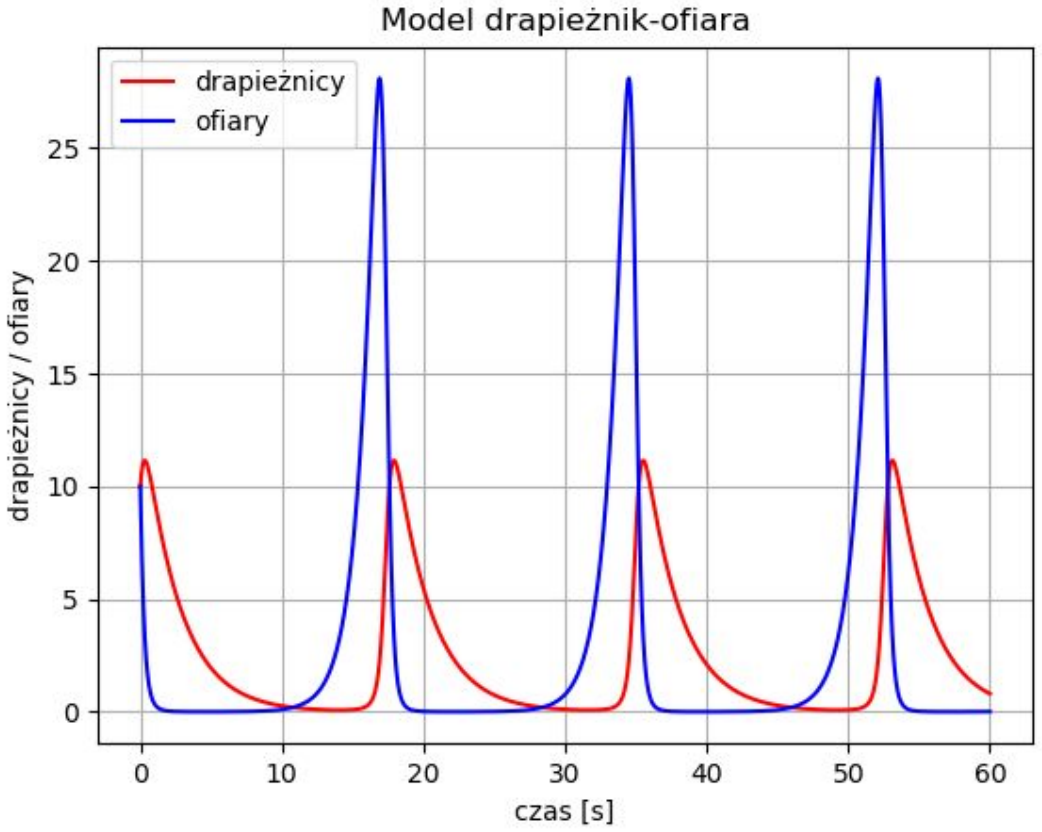
Klasa przedstawiająca model:

```
class LotkaVolterra : public IODE
{
 public:
   LotkaVolterra(double a, double b, double c, double d) :
   a(a), b(b), c(c), d(d) {}

 private:
   std::vector<double> getDerivatives(std::vector<double> values){
     std::vector<double> result;
     result.push_back(-b*values[0] + a*b*values[0]*values[1]);
     result.push_back(c*values[1] - d*values[0]*values[1]);
     return result;
   }

 private:
   double a;
   double b;
   double c;
   double d;
};
```

Ustalmy początkowe populacje: $x_0 = y_0 = 10$
Parametry układu: $a = 0.3$, $b = 0.4$, $c = 0.9$, $d = 0.4$



Otrzymaliśmy spodziewany rezultat pokazujący zachowanie równowagi w przyrodzie.

D. Rozpad promieniotwórczy

Prawo rozpadu promieniotwórczego:

$$\frac{du}{dt} + \frac{u}{\tau} = 0\,,\ u(0) = 1$$

Równoważnie:

$$\frac{du}{dt} = -\frac{u}{\tau}$$

Lub:

$$\frac{du}{dt} = -\lambda * u$$

Gdzie:

λ - stała rozpadu,

τ - średni czas życia (czas po którym pozostaje 1/e początkowej ilości cząstek)

Oczekujemy otrzymać malejącą funkcję zbiegającą do 0.

Skorzystamy z ostatniego równania i klasycznej metody Rungego-Kutty.

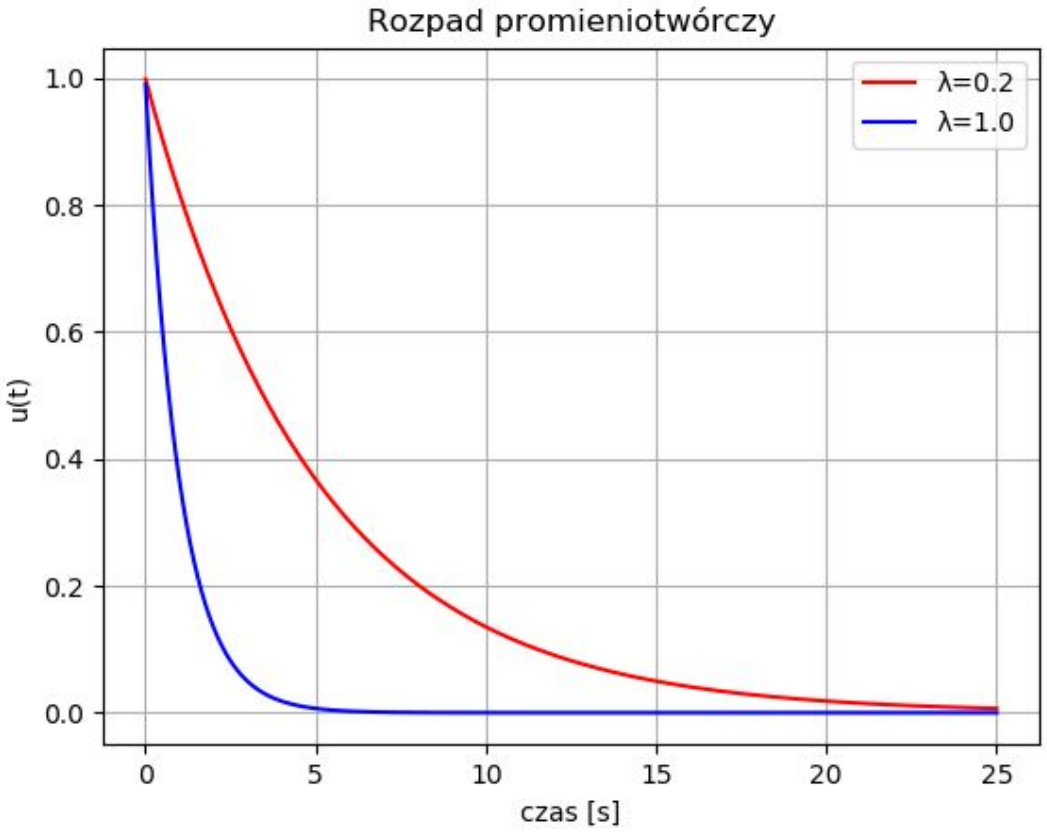
Klasa modelująca równanie:

```
class Decay : public IODE
{
 public:
   Decay(double lambda) : lambda(lambda) {}

 private:
   std::vector<double> getDerivatives(std::vector<double> values){
     std::vector<double> result;
     result.push_back(-lambda * values[0]);
     return result;
   }

 private:
   double lambda;
};
```

Wartością początkową jest 1.0.
Za stałą rozpadu przyjmijmy 0.2 oraz 1.0.



Otrzymaliśmy funkcję malejącą odpowiadającą równaniu:

$$u(t) = e^{-\lambda t}$$