Optymalizacja metodami niedeterministycznymi	
Data wysłania:	23.01.2021
Autorzy:	Emil Kobyłecki Bartosz Kozłowski Jakub Kopeć

1. Parametry wejściowe algorytmu ewolucyjnego:

solution EA (int N, matrix limits, double epsilon, int Nmax, matrix O), gdzie:

N - wymiar problemu do rozwiązania

limits - przestrzeń punktów dopuszczalnych, z których algorytm będzie generował populację początkową

epsilon - dokładność obliczeń,

Nmax - maksymalna liczba wywołań funkcji celu,

O - wartość odchylenia standardowego

Dla testowej funkcji celu wywołanie funkcji ma postać:

wynik = EA (2, limits, 0.0001, 10000, sigma), gdzie:

limits ogranicza punkty startowe tak, by należały do przedziału <-5;5>,

sigma przyjmuje wartości ze zbioru {0.01, 0.1, 1, 10, 100}.

Dla problemu rzeczywistego wywołanie funkcji ma postać:

wynik = EA (2, limits, 0.001, 5000, sigma), gdzie:

limits ogranicza punkty startowe tak, by należały do przedziału <0.1,3>,

sigma przyjmuje wartość 1

2. Dyskusja wyników oraz wnioski:

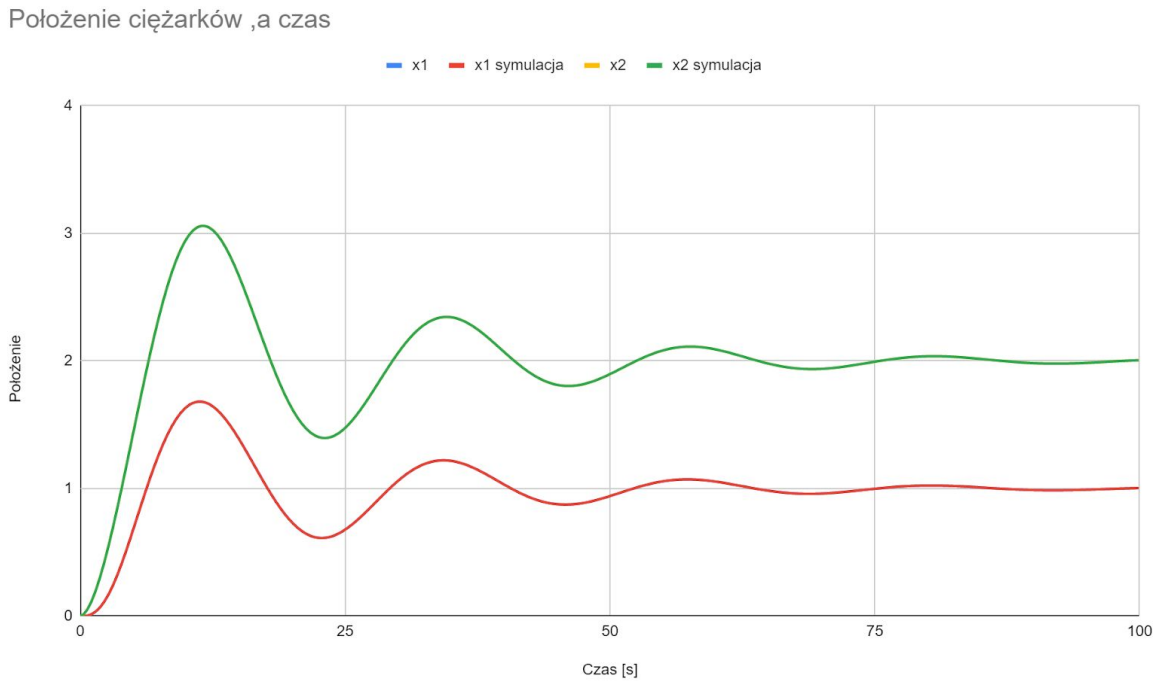
W przeciwieństwie do wcześniej poznanych metod optymalizacji nie podajemy w argumencie funkcji implementującej algorytm ewolucyjny punktu startowego poszukiwania minimum. Algorytm sam sobie generuje rozwiązania początkowe (populację), na podstawie której generuje kolejne rozwiązania w oparciu o populację bazową. Populacja potomków poddawana jest procesowi mutacji i krzyżowania by następnie wybrać najlepsze z rozwiązań. Rozwiązania te nadpisują populację bazową i proces się powtarza.

Zawartość tabeli z arkusza nr 2:

Początkowa wartość zakresu mutacji	x1*	x2*	y*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,01	2,96E-05	1,98E-04	4,96E-05	429	72
0,1	1,70E-05	2,76E-05	4,82E-05	597	90
1	-2,91E-05	-2,89E-05	4,87E-05	1077	100
10	5,05E-05	-8,72E-05	5,07E-05	1650	99
100	-2,05E-04	-2,78E-04	7,65E-05	6134	91

Na podstawie danych z tabeli nr. 2 nie można stwierdzić korelacji pomiędzy początkową wartością zakresu mutacji oraz znalezioną liczbą minimów globalnych funkcji testowej. Metoda EA działa na losowych punktach startowych oraz losowo dobiera populację, dlatego każdorazowe uruchomienie metody będzie zwracało różne rozwiązania. Należy wspomnieć, że zastosowanie tej metody w celu znalezienia minimów globalnych jest bardzo skuteczne w porównaniu do metod bezgradientowych

(metody Rosenbrocka oraz Hooke'a-Jeevesa). Liczba wywołań funkcji celu jest ściśle zależna od początkowej wartości zakresu mutacji, Z tego powodu nie powinno się przyjmować wysokich wartości zakresu mutacji w przypadku gdy istotny jest czas wykonywania obliczeń. Jednakże kosztem takiej decyzji jest ryzyko zapętlenia się algorytmu a następnie zakończenia jego działania w przypadku znalezienia minimum lokalnego.



3. Pozostałe kody źródłowe:
funkcja EA:

```
solution EA(int N, matrix limits, double epsilon, int Nmax, matrix O)
	{
		int mi = 20, lambda = 40;
		solution* P = new solution[mi + lambda];
		solution* Pm = new solution[mi];
		random_device rd;
		default_random_engine gen;
		gen.seed(static_cast<unsigned
int>(chrono::system_clock::now().time_since_epoch().count()));
		normal_distribution<double> distr(0.0, 1.0);
		matrix IFF(mi, 1), temp(N, 2);
		double r, s, s_IFF;
		double tau = 1.0/sqrt(2*N) , tau1 = 1.0/sqrt(2*sqrt(N));
		int j_min;
```

```
        //wylosowanie populacji poczatkowej
        //uniform_real_distribution<double> distr1(-5.0, 5.0);
        uniform_real_distribution<double> distr1(0.1, 3.0);
        for (int i = 0; i < mi; ++i)
        {
            P[i].x = matrix(N, 2);
            for (int j = 0; j < N; ++j)
            {
                P[i].x(j, 0) = distr1(rd);
                P[i].x(j, 1) = O(0,0);
            }
            P[i].fit_fun();
            if (P[i].y < epsilon)
                return P[i];
        }
        while (true)
        {
            //obliczenie funkcji przystosowania sigma i sumy sigm
            s_IFF = 0;
            for (int i = 0; i < mi; ++i)
            {
                IFF(i) = 1 / P[i].y(0);
                s_IFF += IFF(i);
            }
            //kolo ruletka mi osobnikow w lambda osobnikow
            for (int i = 0; i < lambda; ++i)
            {
                P[mi + i].x = matrix(N, 2);
                uniform_real_distribution<double> distr2(0.0, s_IFF);
                r = distr2(rd);
                s = 0;
                for (int j = 0; j < mi; ++j)
                {
                    s += IFF(j);
                    if (r <= s)
                    {
                        P[mi + i] = P[j];
                        break;
                    }
                }
            }
            //mutacja
            for (int i = 0; i < lambda; ++i)
            {
                r = distr(gen);
                for (int j = 0; j < N; ++j)
                {
                    P[mi + i].x(j, 1) *= exp(tau1*r + tau*distr(gen));
                    P[mi + i].x(j, 0) += P[mi + i].x(j, 1) * distr(gen);
                }
```

```
            }
            //krzyzowanie
            for (int i = 0; i < lambda; i += 2)
            {
                uniform_real_distribution<double> distr3(0.0, 1.0);
                r = distr3(rd);
                temp = P[mi + i].x;
                P[mi + i].x = r*temp + (1.0-r)*P[mi+i+1].x;
                P[mi + i + 1].x = r* P[mi + i + 1].x + (1.0-r)*temp;
            }
            //sprawdzenie czy jest rozwiazanie
            for (int i = 0; i < lambda; ++i)
            {
                P[mi + i].fit_fun();
                if (P[mi + i].y < epsilon)
                    return P[mi + i];
            }
            //znalezienie mi najlepszych
            for (int i = 0; i < mi; ++i)
            {
                j_min = 0;
                for (int j = 1; j < mi+lambda; ++j)
                    if (P[j_min].y > P[j].y)
                        j_min = j;
                Pm[i] = P[j_min];
                P[j_min].y = 1e10;
            }
            //przepisanie do nowej populacji
            for (int i = 0; i < mi; ++i)
                P[i] = Pm[i];
            if (solution::f_calls > Nmax)
                return P[0];
        }
    }
```

fit_fun:

```
void solution::fit_fun(matrix O)
{
    //y = pow(x(0), 2) + pow(x(1), 2);
    //y = pow(x(0), 2) + pow(x(1), 2) - cos( 2.5 * 3.14159265358 * x(0) ) - cos(
2.5 * 3.14159265358 * x(1) ) + 2;
    //PROBLEM RZECZYWISTY
    int N=1001;
    static matrix X(N,2);
    if(solution::f_calls==0) //Szczytywanie z pliku przy pierwszym wywołaniu
funkcji
    {
        std::ifstream s("polozenia.txt");
        s>>X;
        s.close();
```

```
    }
    matrix Y0(4,1);
    //początek,krok czasowy,koniec,wartości początkowe, x zawierający b1 && b2
    matrix *Y=solve_ode(0,0.1,100,Y0,x);
    //Wyznaczenie różnicy między szczytanymi z pliku połączeniami ,a symulacją
    y(0)=0;
    for (int i=0;i<N;++i)
    {
        // fabs() ?
        cout<<Y[1](i,0)<<" "<<Y[1](i,2)<<endl;
        y=y+abs( X(i,0)-Y[1](i,0) )+abs( X(i,1)-Y[1](i,2) );
    }
    y(0)=y(0)/(2*N);
    ++f_calls;
}
```

diff:

```
matrix diff(double t, const matrix &Y, matrix P)
{
#elif LAB_NO == 7 //matrix diff(double t, const matrix &Y, matrix P)
    //masy ciężarków,współczynniki sprężystości,siła
    double m1=5,m2=5,k1=1,k2=1,F=1;
    //przesyłana macierz P zawiera opory ruchu
    //double b1=P(0),b2=P(1);
    //wyznaczone optymalne wartości
    double b1=1.43958;
    double b2=1.08438;
    matrix dY(4,1);
    //wyznaczenie x1'
    dY(0)=Y(1);
    //wyznaczenie x1''
    dY(1)=( ( -b1*Y(1)-b2*( Y(1)-Y(3) ) ) -k1*Y(0)-k2*( Y(0)-Y(2) ) )/m1;
    //wyznaczenie x2'
    dY(2)=Y(3);
    //wyznaczenie x2''
    dY(3)=( F+b2*( Y(1)-Y(3) ) +k2*( Y(0)-Y(2) ) )/m2;
    return dY;
#endif
}
```

main:

```
///funkcja testowa:
ofstream x1("x1.txt");
ofstream x2("x2.txt");
ofstream y("y.txt");
ofstream fcalls("fcalls.txt");
matrix limits(2, 2);
limits(0, 0) = limits(1, 0) = -5;   //zakres punktow startowych
limits(0, 1) = limits(1, 1) = 5;
matrix sigma(1, 1);
sigma(0, 0) = 100;
```

```
solution wynik;
for(int i = 0; i < 100; i++)
{
wynik = EA(2, limits, 0.0001, 10000, sigma);
cout << wynik << endl << endl;
x1 << wynik.x(0, 0)<<endl;
x2 << wynik.x(1, 0)<<endl;
y << wynik.y<<endl;
fcalls << wynik.f_calls << endl;
solution::clear_calls();
}
x1.close();
x2.close();
y.close();
fcalls.close();
///problem rzeczywisty:
matrix limits(2,2);
limits(0,0)=limits(1,0)=0.1;
limits(0,1)=limits(1,1)=3;
matrix sigma(1,1);
sigma (0,0)=100;
solution wynik;
// ilość wymiarów problemu,granice (bn należy do 0.1-3),dokładność,ilość
iteracji,współczynnik mutacji
wynik=EA(2,limits,0.0001,5000,sigma);
cout<<wynik<<endl;
```