AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Sprawozdanie 2

Optymalizacja funkcji wielu zmiennych metodami bezgradientowymi

Autorzy: *Paulina Grabowska*
Filip Rak
Arkadiusz Sala

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2024

Spis treści

Cel ćwiczenia

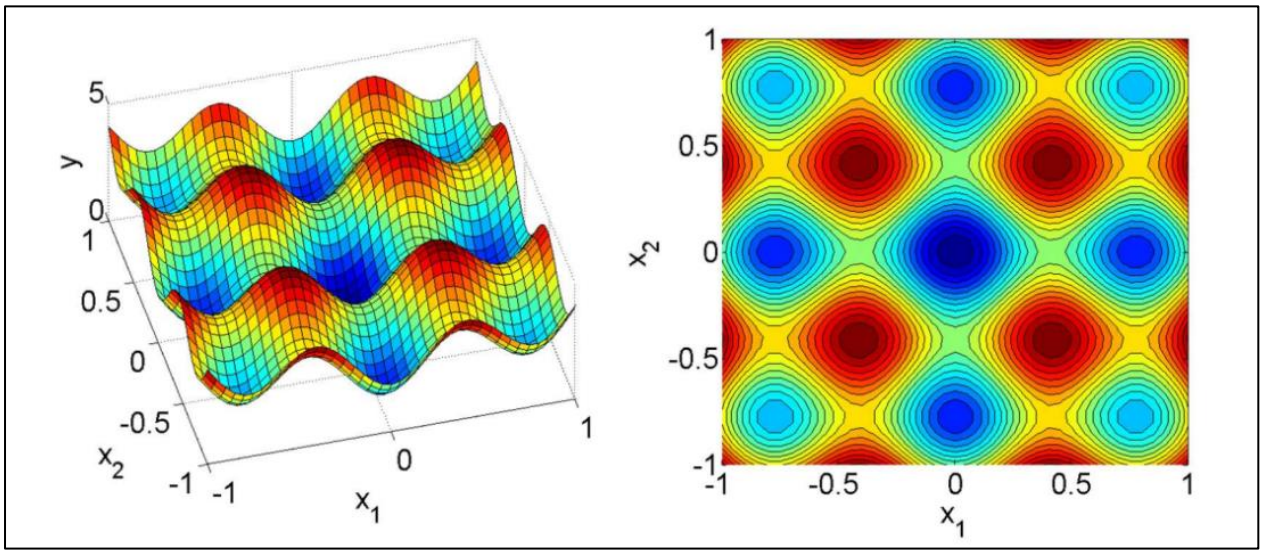
Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat bezgradientowych metod optymalizacji, w tym metody Hooke'a-Jeevesa oraz metody Rosenbrocka. Zostały one zaimplementowane i zastosowane do rozwiązywania problemów optymalizacyjnych dla funkcji wielu zmiennych, umożliwiając praktyczne zrozumienie ich działania oraz efektywności.

Optymalizowane problemy

Testowa funkcja celu

Celem praktycznego wykorzystania zaimplementowanych algorytmów badano funkcję celu podaną wzorem (1), której wykres został załączony na *Rys.1*. Minimum globalne badanej funkcji na przedziale $x_1^{(0)} \in [-1,1]$; $x_2^{(0)} \in [-1,1]$ znajduje się w punkcie (0,0). Minima lokalne o wartości 0,7 w punktach (0; -0,7); (0; 0,7); (-0,7; 0); (0,7; 0) natomiast minima lokalne o wartości powyżej 1 znajdują się w punktach (-0,7; 0,7); (-0,7; -0,7); (0,7; -0,7); (0,7; 0,7). Z powodu tego, że we wzorze funkcji znajdują się cosinusy, minima znajdują się też poza przedziałem [-1; 1].

$$f(x_1, x_2) = x_1^2 + x_2^2 - cos(2{,}5\pi x_1) - cos(2{,}5\pi x_2) + 2 \qquad (1)$$



Rys.1. Wykres funkcji (1)

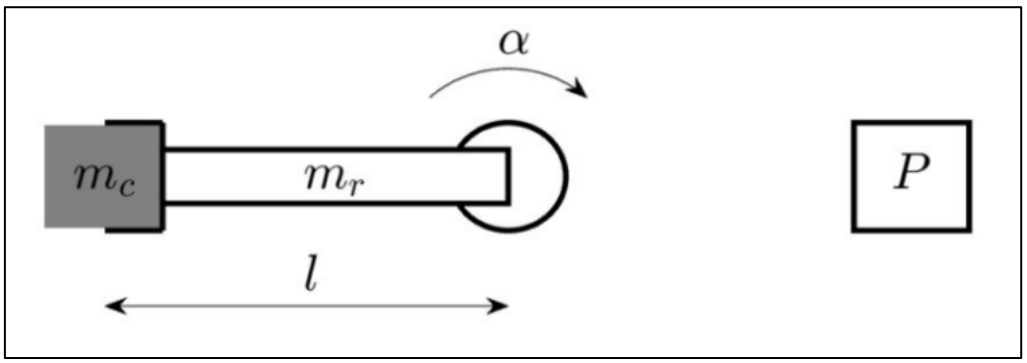
W pliku *user_funs.cpp* dodano implementację funkcji (1) z użyciem macierzy, jak przedstawiono we *Fragmencie kodu 1*.

```
1. matrix ff2T(matrix x, matrix ud1, matrix ud2)
2. {
3.         return matrix( pow( x(0) ) + pow( x(1) )
4.                 - cos( 2.5 * M_PI * x(0) ) - cos(2.5 * M_PI * x(1) ) + 2 );
5. }
```

Fragment kodu 1: funkcja licząca testową funkcję celu

Problem rzeczywisty

W analizowanym problemie mamy model sterowania robotycznym ramieniem o długości l = 1m oraz masie m = 1kg, którego zadaniem jest umieszczenie obciążenia m_c = 5kg na platformie *P* (*Rys. 2*). Aby zrealizować to zadanie, ramię musi wykonać obrót o kąt π radianów i następnie zatrzymać się w odpowiedniej pozycji. Ramię robota znajduje się w płaszczyźnie, dlatego siła grawitacji została pominięta w obliczeniach.



Rys.2. Ilustracja problemu rzeczywistego

Ruch ramienia jest opisany równaniem dynamicznym (2), gdzie b = 0,5*Nms* to współczynnik tarcia, a $I = \frac{1}{3}m_r l^2 + m_c l^2$ to moment bezwładności ramienia z obciążeniem.

$$I\frac{d\alpha}{dt^2} + b\frac{d\alpha}{dt} = M(t) \qquad (2)$$

Moment siły działający na ramię, *M(t)*, (3) jest zależny od różnicy między położeniem i prędkością kątową ramienia a wartościami referencyjnymi. Prędkość kątową oznaczono przez $\omega = \frac{d\alpha}{dt}$, a pozycję docelową przez zależność $\alpha_{ref} = \pi\, rad$. Pożądaną prędkość końcową oznaczono przez $\omega_{ref} = 0\frac{rad}{s}$, a współczynniki wzmocnienia regulatora opisują zmienne k_1 oraz k_2. Zakładamy, że masa ramienia jest równo rozłożona na całej długości l.

$$M(t) = k_1\left(\alpha_{ref} - \alpha(t)\right) + k_2\left(\omega_{ref} - \omega(t)\right) \tag{3}$$

Celem optymalizacji jest znalezienie takich wartości współczynników wzmocnienia zmienne k_1 oraz k_2, dla których funkcjonał jakości $Q(k_1, k_2)$ zilustrowany wzorem (4) osiąga najmniejszą wartość. Czas symulacji (t_{end}) wynosi 100s, z krokiem czasowym *dt* = 0,1s. Wzór (4) uwzględnia błąd kąta (różnicę od kąta referencyjnego); błąd prędkości (różnica od prędkości referencyjnej) oraz moment siły, która działamy. Najważniejszym w tym wzorze jest to, żeby ramię rzeczywiście się obróciło – stąd waga 10 przy zależności położenia.

$$Q(k_1, k_2) = \int_0^{t_{end}} \left(10\left(\alpha_r - \alpha(T)\right)^2 + \left(\omega_r - \omega(t)\right)^2 + \left(M(t)\right)^2\right) dt \tag{4}$$

Początkowe wartości współczynników k_1 oraz k_2 powinny należeć do przedziału [0,10] odpowiednio Nm lub Nms. W pliku *user_funs.cpp* dodano implementację funkcji dla problemu rzeczywistego z użyciem macierzy, jak przedstawiono we *Fragmencie kodu 2*. Funkcja *df2* oblicza pochodne prędkości i kąta ramienia; w tej funkcji zdefiniowane są fizyczne parametry ramienia robota.

```
1. matrix df2(double t, matrix Y, matrix ud1, matrix ud2)
2. {
3.       // Physical parameters
4.       double m1 = 1.0;              // Arm mass
5.       double m2 = 5.0;              // Weight Mass
6.       double l = 1.0;               // Arm length
7.       double b = 0.5;               // Friction coefficient
8.       double I = ((m1 / 3) + m2) * pow(l, 2);
9.
10.      // Control values
11.      double k1 = m2d(ud1);
12.      double k2 = m2d(ud2);
13.
14.      // Targets
15.      double alpha_target = M_PI;   // Target angle in radians
16.      double omega_target = 0.0;    // Target speed
17.
18.      // Getting the moment of force
19.      double alpha_diff = alpha_target - Y(0);
20.      double omega_diff = omega_target - Y(1);
21.      double M_t = k1 * alpha_diff + k2 * omega_diff;
22.
23.      // Get derivatives
24.      matrix dY(2, 1);
25.      dY(0) = Y(1);       // Derivative of an angle is speed
26.      dY(1) = (M_t - b * Y(1)) / I; // Derivative of speed
27.
28.      return dY;
29. }
```

Fragment kodu 2: funkcje *df2*

Natomiast w funkcji *ff2R*, która oblicza jakość regulacji ruchu ramienia w czasie, zadeklarowano niezbędne zmienne do przeprowadzenia symulacji oraz wzór (4).

```
matrix ff2R(matrix x, matrix ud1, matrix ud2)
{
	matrix k1 = x(0);
	matrix k2 = x(1);

	// Initial conditions
	double start_time = 0.0;
	double end_time = 100.0;
	double time_step = 0.1;

	// Initial arm properties
	double start_angle = 0.0;
	double start_speed = 0.0;
	matrix Y0 = matrix(2, new double[2] {start_angle, start_speed});

	// Arm motion simulation
	matrix* results = solve_ode(df2, start_time, time_step, end_time, Y0, k1, k2);

	// Get the quality function of Q
	double Q = 0.0;
	for (int i = 0; i < get_len(results[0]); i++)
	{
		double alpha_diff = results[1](i, 0) - M_PI;	// Target is pi rad
		double omega_diff = results[1](i, 1);		// Target is speed of 0

		double M_t = m2d(k1) * alpha_diff + m2d(k2) * omega_diff;

		Q += (10 * pow(alpha_diff, 2) + pow(omega_diff, 2) + pow(M_t, 2))* time_step;
	}

	// Memory cleanup
	delete[] results;

	// Return the result
	return matrix(Q);
}
```

Fragment kodu 3: funkcja *ff2R*

Implementacja metod optymalizacji

Metoda Hooke'a-Jeevesa

Zaimplementowano metodę poszukiwania optymalnych rozwiązań za pomocą algorytmu Hooka-Jeevesa w funkcji *solution HJ*. Algorytm ten iteracyjnie poszukuje minimum funkcji celu *ff* poprzez testowanie różnych rozwiązań, modyfikując ich wartości i wybierając te, które obniżają wartość funkcji celu. Proces ten jest kontrolowany przez parametry, takie jak krok poszukiwania *s*, współczynnik zmniejszania kroku *alpha*, tolerancję *epsilon*, oraz maksymalną liczbę wywołań *Nmax*.

```
 1. solution HJ(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha, double
epsilon, int Nmax, matrix ud1, matrix ud2)
 2. {
 3.       try
 4.       {
 5.                solution Xopt;
 6.                solution xs(x0);
 7.
 8.                //std::cout << xs.x << std::endl;
 9.
10.                xs.fit_fun(ff, ud1, ud2);
11.
12.                do
13.                {
14.                          solution xB = xs;
15.
16.                          xs = HJ_trial(ff, xB, s, ud1, ud2);
17.                          xs.fit_fun(ff, ud1, ud2);
18.                          if (m2d(xs.y) < m2d(xB.y))
19.                          {
20.                                    do
21.                                    {
22.                                              solution _xB = xB;
23.                                              xB = xs;
24.                                              xs = (xB.x * 2.0) - _xB.x;
25.                                              xs = HJ_trial(ff, xs, s, ud1, ud2);
26.                                              if (solution::f_calls > Nmax)
27.                                              {
28.                                                  throw string("Nie znaleziono przedzialu po
                                                    Nmax probach (f(x)<f(xB))");
29.                                              }
30.                                    } while (m2d(xs.y) < m2d(xB.y));
31.                                    xs = xB;
32.                          }
33.                          else
34.                          {
35.                                    s = alpha * s;
36.                          }
37.                          if (solution::f_calls > Nmax)
38.                          {
39.                               throw string("Nie znaleziono przedzialu po Nmax probach (L)");
40.                          }
41.
42.                } while (s >= epsilon);
43.                Xopt = xs;
44.                Xopt.flag = 0;
45.                return Xopt;
46.       }
47.       catch (string ex_info)
48.       {
49.                throw ("solution HJ(...):\n" + ex_info);
50.       }
51. }
```

Fragment kodu 4: funkcja *HJ*

Funkcja *HJ_trial* służy do obliczania etapu próbnego metody Hooke'a-Jeevesa, w którym do badane jest lokalne zachowanie się funkcji celu w otoczeniu pewnego punktu. Przeprowadzane jest to za pomocą wykonywania niewielkich kroków we wszystkich kierunkach ortogonalnej bazy. Jeśli znalezione rozwiązanie daje niższą wartość funkcji celu, jest przyjmowane jako nowe najlepsze rozwiązanie. W przeciwnym razie, kontynuuje się testowanie w przeciwnym kierunku.

W głównej pętli funkcji *HJ* algorytm iteracyjnie modyfikuje krok *s* zgodnie z wartością *alpha,* aż do spełnienia kryterium dokładności epsilon. Jeśli przekroczy maksymalną liczbę wywołań funkcji celu, rzuca wyjątek, sygnalizując brak znalezienia rozwiązania w zadanym limicie prób.

```
 1. solution HJ_trial(matrix(*ff)(matrix, matrix, matrix), solution XB, double s, matrix ud1,
matrix ud2)
 2. {
 3.       try
 4.       {
 5.                const int DIM = 2;
 6.
 7.                matrix dj(DIM, DIM);
 8.
 9.                for(int k = 0; k < DIM; k++)
10.                         for (int l = 0; l < DIM; l++)
11.                                  dj(k, l) = (l == k)? 1 : 0;
12.
13.               for (int j = 0; j < DIM; j++)
14.               {
15.                         solution xj = XB.x + (s * dj[j]);
16.                         xj.fit_fun(ff, ud1, ud2);
17.                         if ( xj.y < XB.y )
18.                         {
19.                                  XB = xj;
20.                         }
21.                         else
22.                         {
23.                                  xj = XB.x - (s * dj[j]);
24.                                  xj.fit_fun(ff, ud1, ud2);
25.                                  if (xj.y < XB.y)
26.                                  {
27.                                           XB = xj;
28.                                  }
29.                         }
30.               }
31.
32.
33.               return XB;
34.      }
35.      catch (string ex_info)
36.      {
37.               throw ("solution HJ_trial(...):\n" + ex_info);
38.      }
39. }
```

Fragment kodu 5: funkcja *HJ_trial*

Metoda Rosenbrocka

W zaimplementowanej metodzie Rosenbrocka funkcja *solution Rosen* rozpoczyna optymalizację z początkowym punktem *x0* i macierzą kierunków *d*. Algorytm iteracyjnie modyfikuje wektor kroków *s* i wektory *l* oraz *p* dla każdego wymiaru, zwiększając krok, gdy poprawia rozwiązanie, lub zmniejszając go w przeciwnym wypadku.

W przypadku poprawy funkcji celu wektor kierunku *d* jest aktualizowany na podstawie wektorów *l*, które wskazują, o ile przesunięto się w każdym kierunku. Aktualizacja kierunków

wykorzystuje ortogonalizację, aby zapewnić niezależność ruchu w różnych wymiarach, co poprawia efektywność poszukiwania minimum.

Algorytm kończy działanie, gdy wartość największego kroku jest mniejsza niż zadana tolerancja *epsilon* lub po osiągnięciu maksymalnej liczby iteracji *Nmax*. Jeśli minimum nie zostanie znalezione przed przekroczeniem *Nmax*, algorytm zwraca rozwiązanie z odpowiednią flagą błędu.

```
1. solution Rosen(matrix(*ff)(matrix, matrix, matrix), matrix x0, matrix s0, double alpha, double beta, double epsilon, int Nmax, matrix ud1, matrix ud2)
2. {
3.       try
4.       {
5.
6.                const int DIM = 2;
7.                //matrix: row column
8.                matrix d(DIM, DIM); // DIM x DIM square matrix
9.
10.               for (int w = 0; w < DIM; w++)
11.                         for (int k = 0; k < DIM; k++)
12.                                   d(w, k) = (w == k) ? 1 : 0;
13.
14.               matrix l(DIM, 1, 0.0); // vertical vector
15.               matrix p(DIM, 1, 0.0); // vertical vector non-essential
16.               matrix s(s0); // vertical vector
17.
18.               solution xB(x0); // x -> vertical vector; y -> scalar
19.               xB.fit_fun(ff, ud1, ud2);
20.               solution Xopt(xB);
21.               int max_s;
22.               do
23.               {
24.                         for (int j = 0; j < DIM; j++)
25.                         {
26.                                   solution _x(xB.x + s(j) * d[j]);
27.                                   if (_x.fit_fun(ff, ud1, ud2) <  xB.y)
28.                                   {
29.                                             xB = _x;
30.
31.                                             l(j) = l(j) + s(j);
32.                                             s(j) = s(j) * alpha;
33.
34.                                   }
35.                                   else
36.                                   {
37.
38.                                             s(j) = -s(j) * beta;
39.                                             p(j) = p(j) + 1;
40.
41.                                   }
42.                         }
43.                         Xopt = xB;
44.                         bool zero = false;
45.                         //std::cout << "----\n"<< Xopt.x << "\n----\n";
46.                         for (int j = 0; j < DIM; j++)
47.                         {
48.                                   if (p(j) == 0 || abs(l(j)) < epsilon ) {
49.                                             zero = true;
50.                                             break;
51.                                   }
52.                         }
53.
54.                         if (!zero)
55.                         {
56.                                   //change direction { square matrix d(DIM, DIM) }
57.
58.                                   matrix _D(d);
```

```
                                        matrix _lQ(DIM, DIM);
                                        for (int i = 0; i < DIM; i++) // row
                                                  for (int j = 0; j < DIM; j++) // column
                                                            _lQ(i, j) = (i >= j) ? l(i) : 0.0;

                                        _lQ = _D * _lQ;
                                        //std::cout << "_l\n";
                                        //std::cout << _lQ << std::endl;
                                        matrix v(DIM,DIM);
                                        //std::cout << "0:\nv:\n";
                                        //std::cout << _lQ[0] << std::endl;
                                        //std::cout << "norm: " << norm(_lQ[0]) << std::endl;
                                        v.set_col(_lQ[0]/(norm(_lQ[0])), 0);

                                        for (int _j = 1; _j < DIM; _j++) // v/Q matrix column
                                        {
                                                  matrix sigma(DIM,1);
                                                  matrix t_lQ(trans(_lQ[_j]));

                                                  for (int k = 0; k < _j; k++) // sigm column
                                                  {
                                                            sigma.set_col(
                                                                      sigma[0]+(t_lQ * d[k])*d[k],
                                                                      0);

                                                  }

                                                  matrix pk = _lQ[_j] - sigma[0];
                                                  //std::cout << _j <<":\nv:\n";
                                                  //std::cout << pk << std::endl;
                                                  //std::cout << "norm: " << norm(pk) <<std::endl;
                                                  v.set_col(pk/norm(pk), _j);

                                        }
                                        //std::cout << "d!\n";
                                        //std::cout << d << std::endl;
                                        d = v;
                                        //std::cout << d << std::endl;
                                        //end

                                        l = matrix(DIM, 1, 0.0);
                                        p = matrix(DIM, 1, 0.0);
                                        s = s0;
                              }
                              if (solution::f_calls > Nmax)
                              {
                                        Xopt.flag = -2;
                                        break;
                                       //throw string("Nie znaleziono przedzialu po Nmax probach
                                       (f_calls > Nmax)");
                              }
                              max_s = 0;
                              for (int j = 1; j < DIM; j++)
                              {
                                        if (abs(s(max_s)) < abs(s(j)))
                                        {
                                                  max_s = j;
                                        }
                              }

                    } while (abs(s(max_s)) >= epsilon);
                    return Xopt;

          }
          catch (string ex_info)
          {
                    throw ("solution Rosen(...):\n" + ex_info);
          }
}
```

Fragment kodu 6: funkcja *Rosen*

Opracowanie wyników

Zadeklarowano zmienne lokalne wykorzystywane podczas późniejszego testowania (*Fragment kodu 7*). Składają się na nie:

- *epsilon* – dokładność wynosząca 10^{-6};
- *n_max* – liczba maksymalnych wywołań funkcji celu;
- zmienna *contr* równa 0.5 – dla metody Hooke'a-Jeevesa jest to współczynnik *alpha* służący do zmniejszania długości kroku poszukiwań; a dla metody Rosenbrocka jest to współczynnik kontrakcji (*beta*)
- zmienna *expa* równy 2 – używany tylko w metodzie Rosnebrocka to współczynnik ekspansji *alpha*

```
1. // Common arguments
2.  double epsilon = 1e-06;
3.  int n_max = 1000;
4.  double contr = 0.5, expa = 2; // for HJ: contr -> alpha, for Rosen: expa -> alpha, contr -> beta
```

Fragment kodu 7: zmienne lokalne niezbędne dla działania badanych funkcji

Testowa funkcja celu

Obrano trzy długości kroku, kolejno: 0,75; 0,5 oraz 0,25. Badania przeprowadzono dla każdej z dwóch metod optymalizacji (metoda Hooke'a-Jeevesa oraz metoda Rosenbrocka), po 100 optymalizacji dla każdej długości. Wartości uzyskanych wyników zestawiono w *Tabeli 1*, a następnie w *Tabeli 2* zebrano ich uśrednione wartości.

Celem uzyskania danych do obu tych tabel, w funkcji *main* zawarto pętlę (*Fragment kodu 8*), w której dla 100 losowych punktów przeprowadzane są różne metody numeryczne w celu znajdowania minimum funkcji. Wyniki są zapisywane do pliku, w tym współrzędne punktu minimalnego, wartość funkcji celu, liczba wywołań funkcji celu oraz informacja, czy znaleziono minimum spełniające kryterium dokładności.

```
1.      // ---------- Table 1 and Table 2 ----------
2.
3.      std::cout << "Solving for Table 1 and Table 2...";
4.
5.      // Init random number generator
6.      srand(time(NULL));
7.
8.      // Range for defined local minimums
9.      double x_min = -1.0, x_max = 1.0;  // Boundries for random number generation
```

```
10.
11.       // Set step size for testing
12.       double step[3] = { .75, .5, .25 }; // Step size for each iteration
13.
14.       // File output
15.       const char delimiter = '\t';
16.       const string OUTPUT_PATH = "Output/lab_2/";
17.       ofstream tfun_file(OUTPUT_PATH + "out_1_tfun.txt");
18.
19.       // Check if the file has been correctly opened
20.       if (!tfun_file.is_open())
21.                 return;
22.
23.       bool found_candidate = false;
24.       solution H0, R0;
25.
26.       for (int j = 0; j < 3; j++)
27.       {
28.
29.                 for (int i = 0; i < 100; i++)
30.                 {
31.                           // Draw a 2D point
32.                           matrix x(2,1);
33.                           x(0)=x_min+static_cast<double>(rand()) /RAND_MAX * (x_max - x_min);
34.                           x(1)=x_min+static_cast<double>(rand()) /RAND_MAX * (x_max - x_min);
35.
36.                           // Save points in the file
37.                           tfun_file << x(0) << delimiter << x(1) << delimiter;
38.
39.                 // Calculate and write found answer to the same file, Hooke-Jeeves method.
40.                           solution y0 = HJ(ff2T, x, step[j], contr, epsilon, n_max);
41.                           tfun_file << y0.x(0) << delimiter << y0.x(1) << delimiter
42.                           << m2d(y0.y) << delimiter << solution::f_calls << delimiter
43.                                     << (abs(m2d(y0.y)) < epsilon) << delimiter;
44.                           solution::clear_calls();
45.
46.                 // Calculate and write found answer to the same file, Rosenbrock method.
47.                           solution y1 = Rosen(ff2T, x, matrix(2, 1, step[j]), expa, contr,
                              epsilon, n_max);
48.                           tfun_file << y1.x(0) << delimiter << y1.x(1) << delimiter
49.                                     << m2d(y1.y) << delimiter << solution::f_calls <<
                                        delimiter
50.                                     << (abs(m2d(y1.y)) < epsilon);
51.                           solution::clear_calls();
52.                    if (!found_candidate && (abs(m2d(y1.y)) < epsilon) && (abs(m2d(y0.y)) <
                       epsilon))
53.                           {
54.                                     found_candidate = true;
55.                                     H0 = y0;
56.                                     R0 = y1;
57.                           }
58.
59.                           tfun_file << std::endl;
60.                 }
61.
62.       }
63.       // Close file
64.       tfun_file.close();
65.
```

Fragment kodu 8: fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 1* oraz *Tabeli 2*

Analizując, w obu metodach średnie wartości x_1^* oraz x_2^* są bliskie zera, co sugeruje, że algorytmy zbliżają się do optymalnego rozwiązania. Wartość y^* jest niższa w metodzie Hooke'a-Jeevesa niż w metodzie Rosenbrocka przy każdej długości kroku. Metoda Hooke'a-Jeevesa wymaga mniejszej liczby wywołań funkcji celu niż metoda Rosenbrocka przy każdej

długości kroku. W miarę zmniejszania długości kroku, liczba wywołań funkcji celu maleje w obu metodach. Jednakże metoda Rosenbrocka jest bardziej kosztowna obliczeniowo niż metoda Hooke'a-Jeevesa przy tej samej długości kroku. W przypadku oby metod, w miarę zwiększania długości kroku rośnie liczba wywołąń funkcji celu, ale rośnie też liczba minimów globalnych.

Tabela 2. Wartości średnie wyników optymalizacji dla trzech współczynników ekspansji

Długość kroku	Metoda Hooke'a-Jeevesa				
	x_1^*	x_2^*	y*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,75	3,44853E-08	-1,38907E-08	1,06476E-11	225,3125	80
0,5	-1,51777E-07	-1,55258E-07	2,30326E-11	219,1428571	21
0,25	-2,12789E-07	2,25363E-07	2,60762E-11	208,25	12

Długość kroku	Metoda Rosenbrocka				
	x_1^*	x_2^*	y*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,75	-2,88233E-07	8,63796E-08	2,65676E-10	300,7931034	87
0,5	-1,08451E-07	2,93874E-07	2,58655E-10	271,8235294	17
0,25	-1,63391E-07	4,23951E-07	3,23776E-11	242,6666667	12

Kolejno zebrano dane potrzebne do narysowania wykres rozwiązania optymalnego uzyskanego po każdej iteracji naniesiony na wykres poziomic funkcji celu. Dla każdego punktu w trajektorii zapisuje jego współrzędne dla obu metod optymalizacji, a wynik zapisuje do pliku.

```
// ---------- Graph ----------
ofstream graph_file(OUTPUT_PATH + "out_2_tfun.txt");
      if (!graph_file.is_open())
                return;
      int* hl = get_size(H0.ud);
      for (int i = 0; i < hl[1]; i++)
                graph_file << H0.ud(0, i) << delimiter << H0.ud(1, i) << std::endl;

      graph_file << "\n-----\n";
      delete[] hl;
      hl = get_size(R0.ud);
      for (int i = 0; i < hl[1]; i++)
                graph_file << R0.ud(0, i) << delimiter << R0.ud(1, i) << std::endl;

      delete[] hl;
      graph_file.close();
      return;

```

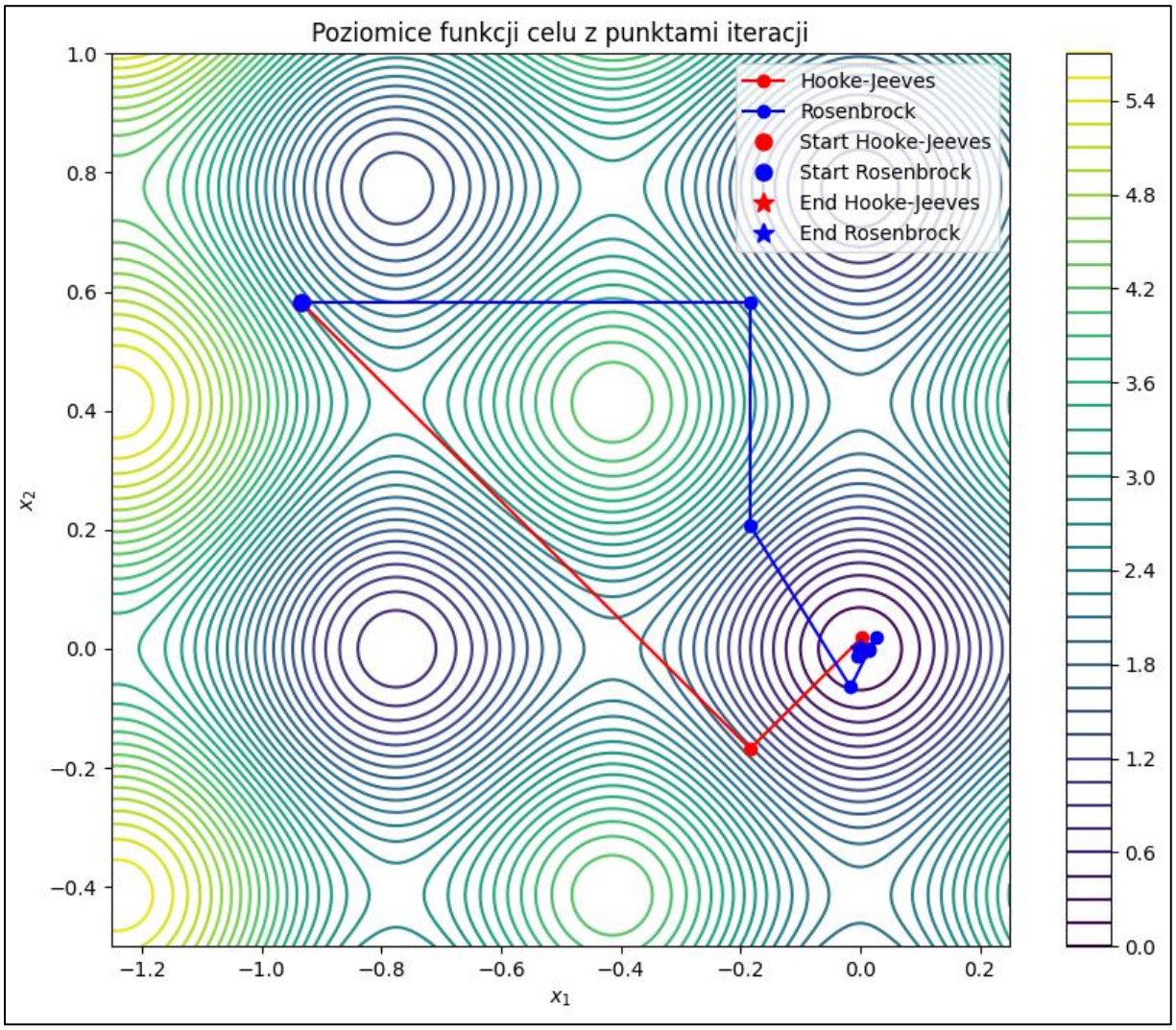
Fragment kodu 9: fragment funkcji *main* pozwalający na zebranie odpowiednich danych do utworzenia grafu funkcji długości przedziału od numeru iteracji.

Następnie naniesiono rozwiązania optymalne uzyskane po każdej iteracji, zarówno dla metody Hooke'a Jeevesa, jak i metody Rosenbrocka. Celem uzyskania takiego wykresu napisano pomocniczy kod w Pythonie, celem wykonania przejrzystej wizualizacji (*Fragment kodu 10*).

```
# funkcji celu
def objective_function(x1, x2):
    return x1**2 + x2**2 - np.cos(2.5 * np.pi * x1) - np.cos(2.5 * np.pi * x2) + 2

# zakres dla x1 i x2
x1_vals = np.linspace(-1.25, 0.25, 500) #(-1.5, 1.5, 500) dla obu dla równego wykresu
x2_vals = np.linspace(-0.5, 1.0, 500) # obecne parametry dla wyraźniejszego wykresu

# siatka wartości funkcji celu
X1, X2 = np.meshgrid(x1_vals, x2_vals)
Z = objective_function(X1, X2)

# współrzędne punktów iteracyjnych z algorytmu Hooke'a-Jeevesa
hooke_jeeves_points = np.array([
    [-0.933653, 0.582141], [-0.183653, -0.167859], [0.00384724, 0.0196406],
    [0.00384724, -0.00379693], [-0.00201214, 0.00206244], [0.000917552, -0.000867244],
    [-0.000547292, 0.0005976], [0.00018513, -0.000134822], [2.02476e-06, 4.82831e-05],
    [-8.36266e-07, -3.54292e-07], [5.94246e-07, -3.54292e-07]
])

# współrzędne punktów iteracyjnych z algorytmu Rosenbrocka
rosenbrock_points = np.array([
    [-0.933653, 0.582141], [-0.183653, 0.582141], [-0.183653, 0.207141],
    [-0.0159477, -0.064212], [0.0259786, 0.0196406], [0.015497, -0.00132257],
    [0.0147269, -7.64985e-05], [-0.0052102, -0.0123983], [-0.000153932, -0.00182651],
    [-0.00081467, -0.00151049], [-0.00100358, -0.000176136], [-0.000439404, 0.00029093],
    [-0.00029535, -4.57578e-05], [5.30734e-05, 5.94204e-05], [-3.4573e-05, 3.29627e-05],
    [-3.04291e-05, 1.04528e-05], [-1.06854e-05, 2.20309e-05], [-7.37814e-06, 1.10751e-05],
    [9.85086e-06, 1.5121e-06], [4.84782e-06, 4.28905e-06], [6.75948e-07, -1.12067e-06],
    [6.51838e-07, 3.09637e-07], [3.85296e-08, -5.83885e-08]
])

# poziomice funkcji celu
plt.figure(figsize=(10, 8))
contour = plt.contour(X1, X2, Z, levels=50, cmap='viridis')
plt.colorbar(contour)
plt.title("Poziomice funkcji celu z punktami iteracji")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

# naniesienie punktów iteracyjnych z algorytmu Hooke'a-Jeevesa
plt.plot(hooke_jeeves_points[:, 0], hooke_jeeves_points[:, 1], 'o-', color='red',
label='Hooke-Jeeves')

# naniesienie punktów iteracyjnych z algorytmu Rosenbrocka
plt.plot(rosenbrock_points[:, 0], rosenbrock_points[:, 1], 'o-', color='blue',
label='Rosenbrock')

# początkowe i końcowe punkty
plt.plot(hooke_jeeves_points[0, 0], hooke_jeeves_points[0, 1], 'ro', markersize=8,
label='Start Hooke-Jeeves')
plt.plot(rosenbrock_points[0, 0], rosenbrock_points[0, 1], 'bo', markersize=8, label='Start
Rosenbrock')
plt.plot(hooke_jeeves_points[-1, 0], hooke_jeeves_points[-1, 1], 'r*', markersize=10,
label='End Hooke-Jeeves')
plt.plot(rosenbrock_points[-1, 0], rosenbrock_points[-1, 1], 'b*', markersize=10, label='End
Rosenbrock')

plt.legend()
plt.show()
```

Fragment kodu 10: funkcja pomocnicza w języku Python umożliwiająca wizualizację wykresu poziomic funkcji celu wraz z rozwiązaniami optymalnymi po każdej iteracji

Metoda Hooke’a-Jeevesa (czerwona linia na *Rys.3*) podąża prostszą, bardziej bezpośrednią ścieżką do minimum globalnego, przechodząc przez kilka punktów pośrednich, natomiast metoda Rosenbrocka (niebieska linia przemieszcza się po bardziej złożonej trajektorii, wykonując większą liczbę zakrętów i korekt, zanim osiągnie końcowe rozwiązanie. Metoda Hooke’a-Jeevesa wykazuje bardziej bezpośredni ruch do celu, podczas gdy metoda Rosenbrocka przechodzi przez bardziej szczegółowy proces poszukiwania, z większą liczbą korekt w trajektorii.



Rys.3. Wykres funkcji długości przedziału od numeru iteracji
dla oby metod optymalizacji

Problem rzeczywisty

W przypadku problemu rzeczywistego program został uruchomiony jedynie raz, a jego wyniki zostały zebrane w *Tabeli 3*. Celem uzyskania wyników wykorzystano kod załączony na *Fragmencie kodu 11*, będący częścią funkcji *main*. Dla każdej z metod wywoływana jest funkcja dla zadanego zakresu, a wyniki zapisywane są do pliku. Za początkowe wartości $k_1$oraz k_2 obrano wartości 2 i 6.

```
1.      // ---------- Table 3 ----------
2.
3.      std::cout << "\nSolving for Table 3...";
4.
5.      // Open files
6.      ofstream hook_file(OUTPUT_PATH + "out_3_1_hook.txt");
7.      ofstream rosen_file(OUTPUT_PATH + "out_3_2_rosen.txt");
8.
9.      // Problem parameters
10.     double starting_step = 1.0;
11.     double k_values[2] = { 2.0, 6.0 };
12.     matrix x0 = matrix(2, k_values);
13.
14.     // Save the results of Hook's method
15.     solution hook_opt = HJ(ff2R, x0, starting_step, contr, epsilon, n_max);
16.     hook_file
17.               << hook_opt.x(0) << delimiter  << hook_opt.x(1) << delimiter
18.               << m2d(hook_opt.y) << delimiter << hook_opt.f_calls << delimiter
19.               << hook_opt.flag;
20.     solution::clear_calls();
21.
22.     // Save the results of Rosenbrock's method
23.     solution rosen_opt = Rosen(ff2R, x0, matrix(2, 1, starting_step), expa, contr, epsilon,
n_max);
24.     rosen_file
25.               << rosen_opt.x(0) << delimiter << rosen_opt.x(1) << delimiter
26.               << m2d(rosen_opt.y) << delimiter << rosen_opt.f_calls << delimiter
27.               << rosen_opt.flag;
28.     solution::clear_calls();
29.
30.     // Close the files
31.     hook_file.close();
32.     rosen_file.close();
```

Fragment kodu 11: fragment funkcji main zawierający funkcjonalności dla *Tabeli 3*

Zgodnie z danymi zestawionymi w *Tabeli 3*, wyniki dla k_1^*, k_2^* i minimalnej wartości funkcji celu Q^* są niemal identyczne w obu metodach, co oznacza, że obie metody skutecznie znalazły to samo minimum, wskazując na równą precyzję w znalezieniu optymalnych parametrów. Metoda Hooke'a-Jeevesa osiągnęła minimum funkcji celu po 233 wywołaniach funkcji, podczas gdy metoda Rosenbrocka wymagała 329 wywołań, co wskazuje, że metoda Hooke'a-Jeevesa była bardziej efektywna obliczeniowo, potrzebując mniejszej liczby iteracji do osiągnięcia tego samego wyniku.

Tabela 3. Zestawienie wyników optymalizacji dla problemu rzeczywistego

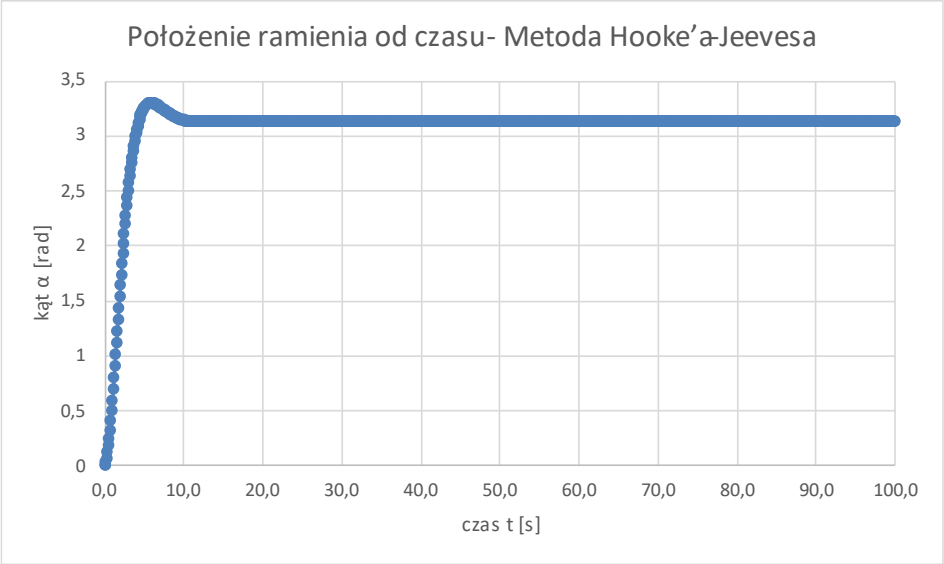
Długość kroku	Metoda Hooke'a-Jeevesa				Metoda Rosenbrocka			
	k_1*	k_2*	Q*	Liczba wywołań funkcji celu	k_1*	k_2*	Q*	Liczba wywołań funkcji celu
1	2,87318	4,8817	194,135	233	2,87319	4,88171	194,135	329

Następnie przygotowano kod do przeprowadzenia symulacji dla obu metod optymalizacji (*Fragment kodu 12*). Ustawiono warunki początkowe dla kąta i prędkości oraz parametry czasu – symulacja będzie trwała 100s z krokiem czasowym 0.1s. Zebrane dane zapisywane są do plików z których następnie będzie możliwe wykonanie wykresów położenia ramienia robota od czasu oraz prędkości ramienia robota od czasu dla obu metod optymalizacji.

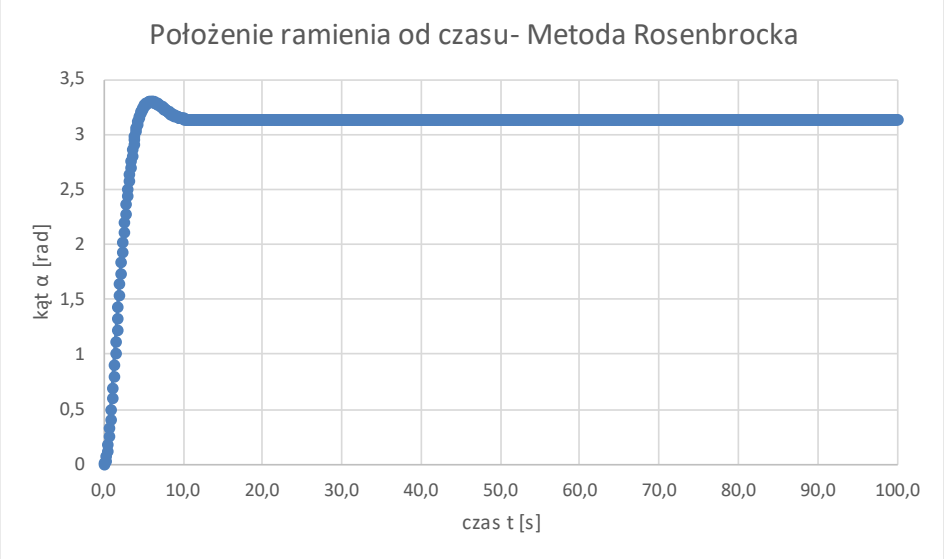
```
1.      // ---------- Simulation ----------
2.
3.      std::cout << "\nSolving for Simulation table...\n";
4.
5.      // Open files
6.      hook_file.open(OUTPUT_PATH + "out_4_1_hook.txt");
7.      rosen_file.open(OUTPUT_PATH + "out_4_2_rosen.txt");
8.
9.      // Initial condition
10.     matrix Y0 = matrix(2, new double[2] {0.0, 0.0}); // Start angle and speed
11.
12.     // Time parameters
13.     double start_time = 0.0;
14.     double end_time = 100.0;
15.     double time_step = 0.1;
16.
17.     // Solve for Hook's optimization
18.     matrix ud1 = hook_opt.x(0);   // K1
19.     matrix ud2 = hook_opt.x(1);   // K2
20.     matrix* hook_results = solve_ode(df2, start_time, time_step, end_time, Y0, ud1, ud2);
21.     hook_file << hook_results[1];
22.
23.     // Solve for Rosen's optimization
24.     ud1 = rosen_opt.x(0); // K1
25.     ud2 = rosen_opt.x(1); // K2
26.     matrix* rosen_results = solve_ode(df2, start_time, time_step, end_time, Y0, ud1, ud2);
27.     rosen_file << rosen_results[1];
28.
29.     // Close the files
30.     hook_file.close();
31.     rosen_file.close();
32.
```

Fragment kodu 12: fragment funkcji *main* pozwalający na zebranie odpowiednich danych do utworzenia odpowiednich wykresów

Na *Rys. 4a* oraz na *Rys. 4b* przedstawiono wykresy zamiany położenia ramienia robota w czasie dla obu metod optymalizacji. Wykresy obu metod są identyczne - kąt położenia ramienia stabilizuje się na poziomie około 3 rad. Przebiegi w obu wykresach mają charakter podobny: następuje szybkie wzrostowe przejście w pierwszych kilku sekundach, a następnie tłumienie oscylacji do ustabilizowanej wartości. Wykresy wskazują, że w obu metodach ramie osiąga stabilny kąt w czasie krótszym niż 10 sekund.

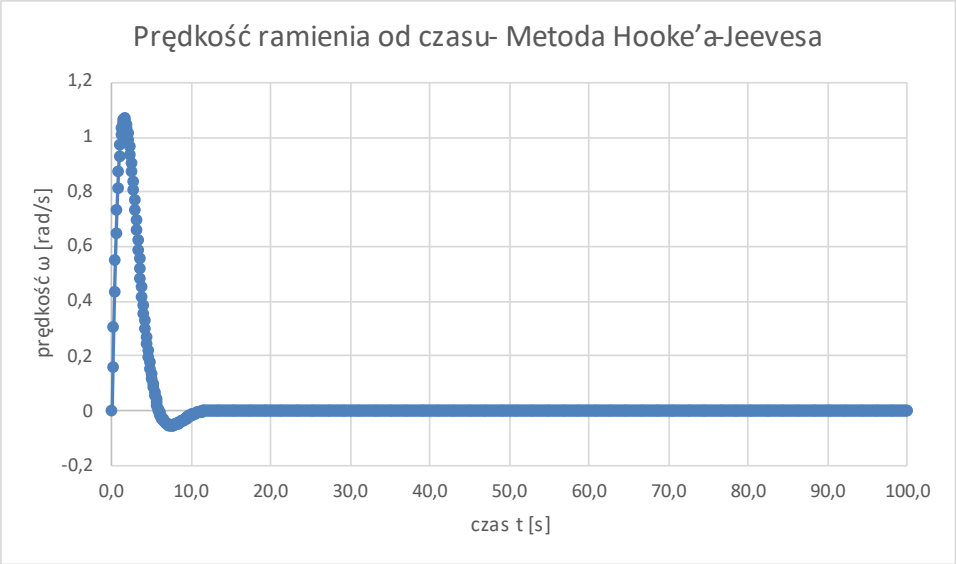


Rys. 4a. Wykres zmiany położenia ramienia w czasie uzyskany metodą Hooke'a-Jeevesa

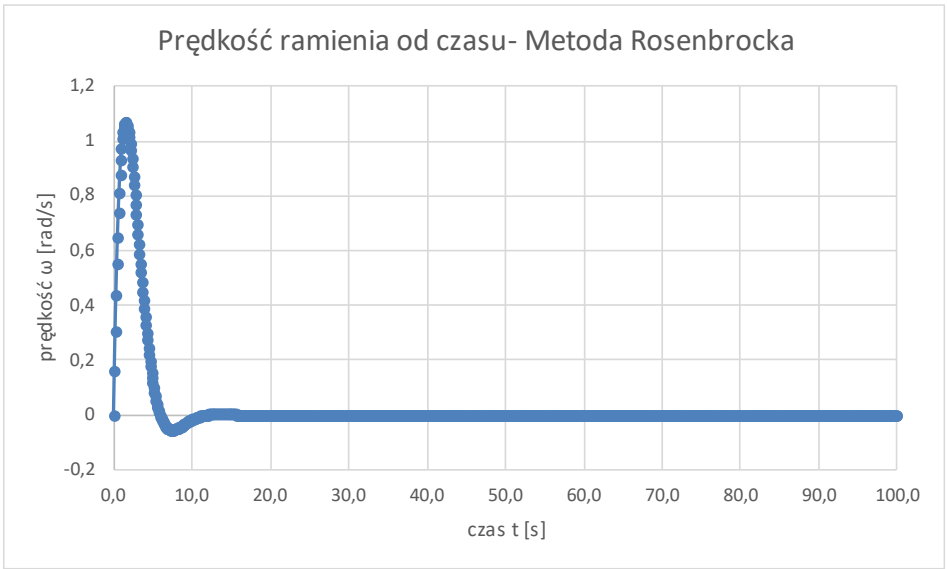


Rys. 4b. Wykres zmiany położenia ramienia w czasie uzyskany metodą Rosenbrocka

Na *Rys. 5a* oraz na *Rys. 5b* przedstawiono wykresy zamiany prędkości kątowej ramienia robota w czasie dla obu metod optymalizacji. Również w tym przypadku oba uzyskane wykresy są identyczne - obie metody początkowo wykazują gwałtowny wzrost prędkości kątowej, wskazując na szybki ruch ramienia. Obie metody również skutecznie zbliżają się do stanu ustalonego, w którym prędkość kątowa stabilizuje się blisko zera. Również w przypadku zbieżności nie widać znaczących różnic między metodami.



Rys. 5a. Wykres zmiany prędkości ramienia w czasie uzyskany metodą Hooke'a-Jeevesa



Rys. 5b. Wykres zmiany prędkości ramienia w czasie uzyskany metodą Rosenbrocka

Wnioski

Obie metody optymalizacji uzyskują podobne wartości dla parametrów optymalnych x_1^* oraz x_2^*, które są bliskie zera, co oznacza że obie metody skutecznie znajdują globalne minimum funkcji celu, wskazując na ich podobną precyzję. Metoda Hooke'a-Jeevesa okazuje się być bardziej efektywna obliczeniowo, gdyż potrzebuje mniejszej liczby wywołań funkcji celu niż metoda Rosenbrocka.

Zmniejszenie długości kroku prowadzi do zmniejszenia liczby wywołań funkcji celu w obu metodach, co może wynikać z dokładniejszego dopasowania do wartości minimalnej. Wzrost długości kroku sprawia, że obie metody wykrywają więcej minimów globalnych, co wskazuje na dokładniejsze przeszukiwanie przestrzeni rozwiązań.

Metoda Hooke'a-Jeevesa wybiera prostsze, bezpośrednie kierowanie w stronę minimum globalnego, co przejawia się na wykresie mniej zakrętów i korekt w porównaniu z metodą Rosenbrocka, która przyjmuje bardziej złożoną ścieżkę, przechodząc przez większą liczbę korekt i zakrętów, zanim osiągnie rozwiązanie końcowe.

Choć obie metody osiągają podobne wyniki w kontekście jakości rozwiązania, metoda Hooke'a-Jeevesa jest bardziej efektywna pod względem obliczeniowym, wymagając mniej iteracji do osiągnięcia optymalnego wyniku. Metoda Rosenbrocka wykazuje bardziej szczegółowy proces poszukiwania i jest bardziej kosztowna obliczeniowo, ale daje wynik o podobnej jakości.