Podstawy Programowania Komputerów

Wykład 8

Algorytmy numeryczne i optymalizacja

Rok akademicki: 2022/2023

Dr inż. Łukasz Maliński

lukasz.malinski@polsl.pl

Politechnika Śląska

UCZELNIA BADAWCZA
INICJATYWA DOSKONAŁOŚCI

KATEDRA INFORMATYKI PRZEMYSŁOWEJ POLITECHNIKI ŚLĄSKIEJ

Plan wykładu

- Problemy dokładności numerycznej
- Kompensacja błędów numerycznych
- Wstęp do algorytmów numerycznych:
  - pierwszy przykład - poszukiwanie minimum funkcji
  - Rodzaje zbieżności
  - sterowanie zbieżnością algorytmów – kryteria i przykłady
  - Przykład użycia metody Monte-Carlo
- Szacowanie czasu wykonania kodu.
- Metody optymalizacji obliczeń iteracyjnych

Problem dokładności numerycznej

Liczby rzeczywiste w systemie binarnym:

Ujemne potęgi dwójki i ich wielokrotności:

$0.50_{(10)} = 1 * (2^{-1})_{(10)} = 0.10_{(2)}$;
$0.25_{(10)} = 1 * (2^{-2})_{(10)} = 0.01_{(2)}$;
$0.75_{(10)} = 3 * (2^{-2})_{(10)} = 0.11_{(2)}$;

Inne liczby rzeczywiste:

$0.10_{(10)} = 0.00011(0011)_{(2)}$;
$0.30_{(10)} = 0.010011(0011)_{(2)}$;

UWAGI:

- Bez względu na liczbę bitów przeznaczoną na zapis liczby, większość liczb rzeczywistych nie może być zapisana dokładnie.
- Ograniczenia w reprezentacji liczb rzeczywistych powodują znikome błędy wartości – tzw. błędy numeryczne.
- Problem polega na tym, że dla komputera dwie liczby są sobie równe tylko jeśli ich zapis binarny jest identyczny (co do jednego bitu)!
- Jeśli ta sama liczba jest otrzymywana na różne sposoby, to wyniki mogą się różnić. Dla nas może to być znikoma różnica, ale dla komputera jest bardzo istotna.

Problem dokładności numerycznej

Faktyczne wartości liczb zmiennoprzecinkowych:

```
double a = 0.1               =>  a = 0.10000000000000056
double b = 0.3               =>  b = 0.30000000000000444
double c = 0.1 + 0.1 + 0.1   =>  c = 0.29999999999999889
```

UWAGI:

- Wynikiem porównania: (0.3 == 0.3) jest zawsze prawda, ale wynikiem porównania: (0.3 == (0.1 + 0.1 + 0.1)), już nie!
- Należy zawsze liczyć się z możliwymi błędami numerycznymi, gdy obliczamy tą samą wartość na różne sposoby. Ponadto, błędy te zazwyczaj rosną wraz ze wzrostem złożoności obliczeń.
- Typ float przechowuje informację na mniejszej liczbie bitów, więc złożone operacje z jego użyciem skutkują znacznie większym błędem.
- Na dokładność numeryczną może też mieć wpływ architektura procesora, oraz proces automatycznej optymalizacji kodu w trakcie tłumaczenia z C++ do assemblera. Niektóre procesory do masowego przetwarzania równoległego mogą nawet nie gwarantować pełnej powtarzalności wyników uzyskiwanych z tego samego wzoru.

Problem dokładności numerycznej - przykład


```cpp
double a = 0.75, b = 0.25 + 0.25 + 0.25;
double c = 0.3, d = 0.1 + 0.1 + 0.1;

if (a == b) cout << "a = b\n";
else cout << "a != b\n";

if (c == d) cout << "c = d\n";
else cout << "c != d\n";

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;

cout << setprecision(20) << fixed << endl;

cout << "a: " << a << endl;
cout << "b: " << b << endl;
cout << "c: " << c << endl;
cout << "d: " << d << endl;
```

$0.25_{(10)} = 0.01_{(2)}$; $0.3_{(10)} = 0.0100110011(0011)_{(2)}$

WAŻNE:

+ wyniki są zawsze dokładne dla liczb dających się skomponować z potęg liczby 2 (0.25 to 2^{-2}),
+ pozostałe liczby (np. 0.3) cechują się nieskończonym zapisem bitowym,
+ błędy numeryczne są często widoczne dopiero na odległych miejscach dziesiętnych, co łatwo może umknąć uwadze przy domyślnej precyzji ich wyświetlania na ekranie.

C:\Windows\system32\cmd.exe

```
a = b
c != d
a: 0.75
b: 0.75
c: 0.3
d: 0.3

a: 0.75000000000000000000
b: 0.75000000000000000000
c: 0.29999999999999998890
d: 0.30000000000000004441
```

Pętla z licznikiem rzeczywistym

```
double pocz, kon, krok;
cout << "Podaj pocz: "; cin >> pocz;
cout << "Podaj krok: "; cin >> krok;
cout << "Podaj kon : "; cin >> kon;

cout << scientific << setprecision(20);
cout << endl;

for (double x = pocz; x <= kon; x += krok)
{
   cout << "x: " << x << endl;
}
cout << endl;
```

C:\WINDOWS\system32\cmd.exe

```
Podaj pocz: -1.2
Podaj krok: 0.3
Podaj kon : 1.2

x: -1.19999999999999995559e+00
x: -8.99999999999999911182e-01
x: -5.99999999999999866773e-01
x: -2.99999999999999877875e-01
x: 1.11022302462515654042e-16
x: 3.00000000000000099920e-01
x: 6.00000000000000088818e-01
x: 9.00000000000000133227e-01
```

Brak wartosci 1.2!

C:\WINDOWS\system32\cmd.exe

```
Podaj pocz: 1.5
Podaj krok: 0.3
Podaj kon : 3.6

x: 1.50000000000000000000e+00
x: 1.80000000000000004441e+00
x: 2.10000000000000008882e+00
x: 2.39999999999999991118e+00
x: 2.69999999999999973355e+00
x: 2.99999999999999955591e+00
x: 3.29999999999999937828e+00
x: 3.59999999999999920064e+00
```

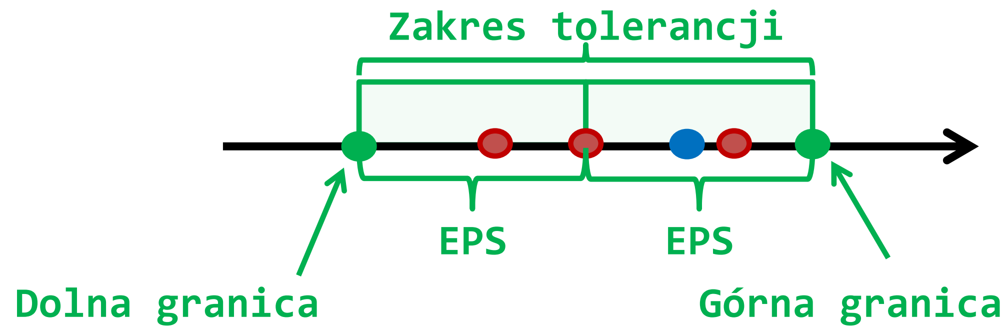
WAŻNE:

+ wartość 0, wyliczana wskutek sumowania niedokładnych liczb, też może być obarczona błędem, mimo że ma dokładny zapis bitowy.
+ błędy numeryczne zakłócają działanie inkrementację w pętli z licznikiem rzeczywistym do tego stopnia, że mogą skutkować pominięciem całych iteracji (to może zależeć też od granic).

Porównywanie liczb rzeczywistych - tolerancja

PROBLEM:

ROZWIĄZANIE:



UWAGI:

- Wprowadzenie tolerancji polega na określeniu niewielkiego zakresu wartości (tolerancji) . Jeśli wynik obliczeń się w nim zawrze, to uznawany jest za „równy" wynikowi prawdziwemu.
- Zakres tolerancji musimy budować wokół, jednego z wyników obliczeń, a nie wokół wartości prawdziwej, gdyż ta może być nieosiągalna.
- Wartość EPS to bardzo mała liczba, która powinna być o kilka rzędów mniejsza o spodziewanego wyniku (lub kroku) obliczeń.
- Granice przedziału uzyskujemy przez dodanie i odjęcie od jednego z wyników (tego, który uznajemy za najdokładniejszy) wartości EPS.

Implementacja tolerancji


```cpp
double a = 0.3, b = 0.1 + 0.1 + 0.1;
double eps = 0.001 * a;

if (a - eps < b && b < a + eps) cout << "a = b\n";
else cout << "a != b\n";
```

PORÓWNYWANIE LICZB

```cpp
double pocz, kon, krok;
cout << "Podaj pocz: "; cin >> pocz;
cout << "Podaj krok: "; cin >> krok;
cout << "Podaj kon : "; cin >> kon;
cout << endl;

eps = 0.01*krok;
```

WYLICZENIE TOERANCJI NA BAZIE KROKU

```cpp
cout << "Tolerancja: " << eps << endl;
cout << scientific << setprecision(20);
cout << "Wartosc brzegowa + tolerancja:\n"
     << kon + eps << endl << endl;

for (double x = pocz; x <= kon + eps; x += krok)
  cout << "x: " << x << endl;
```

KOREKCJA PĘTLI

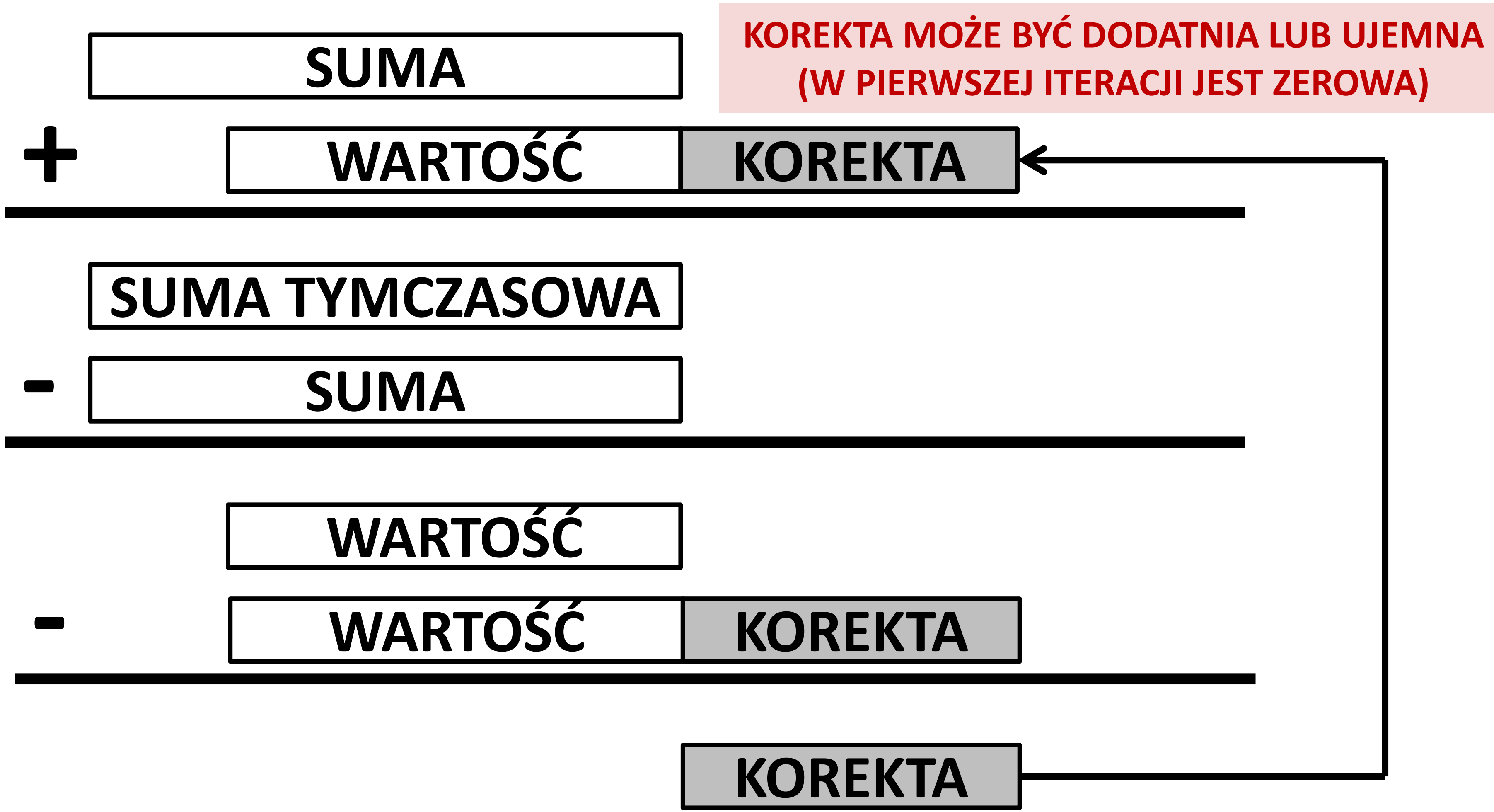
C:\Windows\system32\cmd.exe

```
a = b
Podaj pocz: -1.2
Podaj krok: 0.3
Podaj kon : 1.2

Tolerancja: 0.003
Wartosc brzegowa + tolerancja:
1.20299999999999984723e+00

x: -1.19999999999999995559e+00
x: -8.99999999999999911182e-01
x: -5.99999999999999866773e-01
x: -2.99999999999999877875e-01
x: 1.11022302462515654042e-16
x: 3.00000000000000099920e-01
x: 6.00000000000000088818e-01
x: 9.00000000000000133227e-01
x: 1.20000000000000017764e+00

Press any key to continue . . .
```

WAŻNE:

+ wprowadzenie tolerancji pomaga osiągnąć prawidłowy wynik, porównania ale należy pamiętać, żeby jej zakres nie był za duży, bo może się nagle okazać że 0.4 = 0.3.
+ to, że w przypadku pętli z licznikiem rzeczywistym wystarczy zwiększyć wartość końcową o eps, gdyż tylko obliczenia z nadmiarem stanowią problem.

Algorytm Kahana



Algorytm Kahana - implementacja


```
double suma_rz = 0.0, wartosc_rz = 0.001;
constexpr long N = 100000000;
cout << scientific << setprecision(12);

for (long i = 0; i < N; ++i)
{
  suma_rz = suma_rz + wartosc_rz;
}
```

ZWYKŁE SUMOWANIE

```
cout << "Suma:\t\t" << suma_rz << endl;
cout << "Prawidlowa:\t" << N*wartosc_rz;
cout << endl << endl;

suma_rz = 0.0;
double blad = 0.0;
for (long i = 0; i < N; ++i)
{
  double y = wartosc_rz - blad;
  double t = suma_rz + y;
  blad = (t - suma_rz) - y;
  suma_rz = t;
}
```

ALGORYTM KACHAN'A

```
cout << "Suma:\t\t" << suma_rz << endl;
cout << "Prawidlowa:\t" << N*wartosc_rz;
cout << endl << endl;
```

C:\WINDOWS\system32\cmd.exe **double**

```
Suma:           1.000000000261e+05
Prawidlowa:     1.000000000000e+05

Suma:           1.000000000000e+05
Prawidlowa:     1.000000000000e+05
```

C:\WINDOWS\system32\cmd.exe **float**

```
Suma:           3.276800000000e+04
Prawidlowa:     1.000000078125e+05

Suma:           1.000000078125e+05
Prawidlowa:     1.000000078125e+05
```

WAŻNE:

+ w algorytmie Kachana obliczamy wartość dodawaną już po jej dodaniu i odnosimy do faktycznej zadanej.
+ problem błędu sumowania jest bardzo widoczny dla typu float.

Algorytmy numeryczne

Algorytmem numerycznym będziemy nazywać metodę obliczeniową rozwiązującą problem matematyczny w dziedzinie liczb rzeczywistych.

<u>UWAGI:</u>

- Wyniki algorytmów numerycznych są zwykle obarczone błędem (dają rozwiązanie przybliżone).
- Algorytmy numeryczne są bardzo często algorytmami iteracyjnymi, które wyliczają kolejne przybliżenia rozwiązania problemu, wraz z kolejnymi jego iteracjami.
- Algorytm nazwiemy zbieżnym, gdy wraz ze wzrostem liczby iteracji błąd wyniku maleje (osiąga 0 przy nieskończonej liczbie iteracji).
- Liczba iteracji algorytmu może być z góry zadana lub uzależniona od przyjętego kryterium zbieżności.
- Głównymi czynnikami wpływającymi na wielkość błędu są:
  + błędy numeryczne (reprezentacji cyfrowej liczb rzeczywistych),
  + przyjęte kryteria zbieżności i parametry algorytmu.

Przykład algorytmu numerycznego


```cpp
double funMat(double x)
{
  return x*x + 2 * x + 1; // minumim w (-1,0)
}
```

FUNKCJA DLA, KTÓREJ SZUKAMY MINIMUM

```cpp
void main(void)
{
  double xPocz, xKon, dx, y, yMin, xMin;
  cout << "Podaj poczatek przedzialu: "; cin >> xPocz;
  cout << "Podaj krok poszukiwania: ";   cin >> dx;
  cout << "Podaj koniec przedzialu: ";   cin >> xKon;
  double eps = 0.01*dx;

  xMin = xPocz;
  yMin = funMat(xMin);
```

OKREŚLNIE OBSZARU POSZUKIWAŃ I ROZWIĄZANIA WSTĘPNEGO

```cpp
  for (double x = xPocz; x < xKon + eps; x += dx)
  {
    y = funMat(x);
    cerr << "y: " << y << " w pkt.: " << x << endl;
```

PRÓBKOWANIE FUNKCJI

```cpp
    if (y < yMin)
    {
      yMin = y;
      xMin = x;
    }
  }
  cout << "Minimum: " << yMin << " w pkt.: " << xMin << endl;
}
```

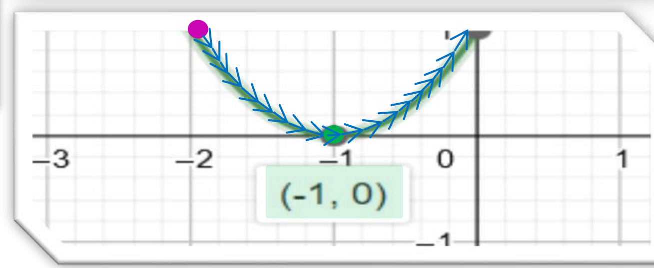
ZAPAMIĘTANIE NAJLEPSZEGO WYNIKU

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -2.0
Podaj krok poszukiwania: 0.3
Podaj koniec przedzialu: 0.0
y: 1 w pkt.: -2
y: 0.49 w pkt.: -1.7
y: 0.16 w pkt.: -1.4
y: 0.01 w pkt.: -1.1
y: 0.04 w pkt.: -0.8
y: 0.25 w pkt.: -0.5
y: 0.64 w pkt.: -0.2
Minimum: 0.01 w pkt.: -1.1
```

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -2.0
Podaj krok poszukiwania: 0.2
Podaj koniec przedzialu: 0.0
y: 1 w pkt.: -2
y: 0.64 w pkt.: -1.8
y: 0.36 w pkt.: -1.6
y: 0.16 w pkt.: -1.4
y: 0.04 w pkt.: -1.2
y: 0 w pkt.: -1
y: 0.04 w pkt.: -0.8
y: 0.16 w pkt.: -0.6
y: 0.36 w pkt.: -0.4
y: 0.64 w pkt.: -0.2
y: 1 w pkt.: -2.77556e-16
Minimum: 0 w pkt.: -1
```



WAŻNE:

+ błąd i liczba iteracji zależą tu od przyjętej wielkości kroku.

Sterowanie zbieżnością

Wybrane kryteria zbieżności:

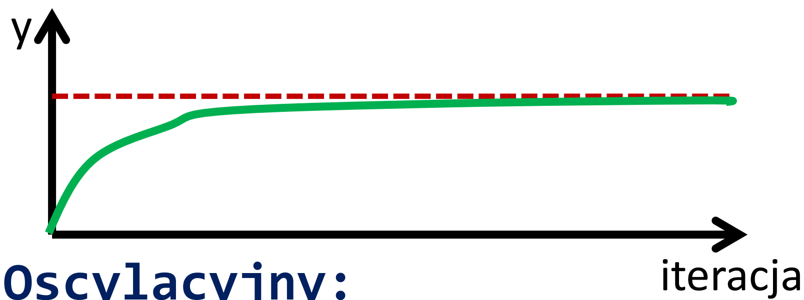
1) Uzyskanie gorszego wyniku
2) limit iteracji i/lub stagnacja
3) Dopuszczalny błąd wielkości powiązanej z wynikiem
4) Dopuszczalny przyrost oszacowania wyniku

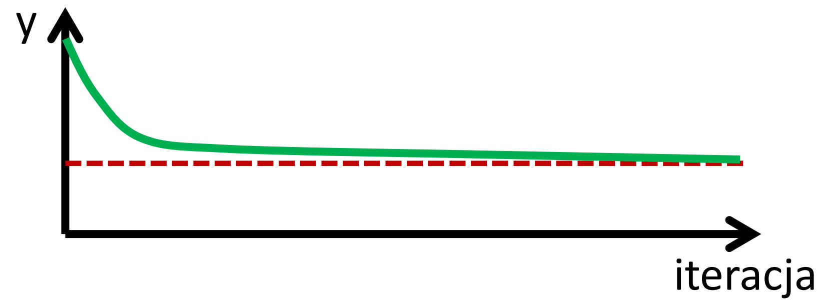
UWAGI:

- Kryterium zbieżności to warunek jaki musi zostać spełniony, aby uznać, że można już zakończyć obliczenia.
- Kryterium zbieżności musi być właściwie dobrane do rozwiązywanego problemu (są kryteria lepsze i gorsze, i często te gorsze mają szerszy zakres stosowalności).
- Kryteria zbieżności same często posiadają parametry, które także mają wpływ na dokładność algorytmu. Często im bardziej wymagające kryterium tym większa dokładność obliczeń.

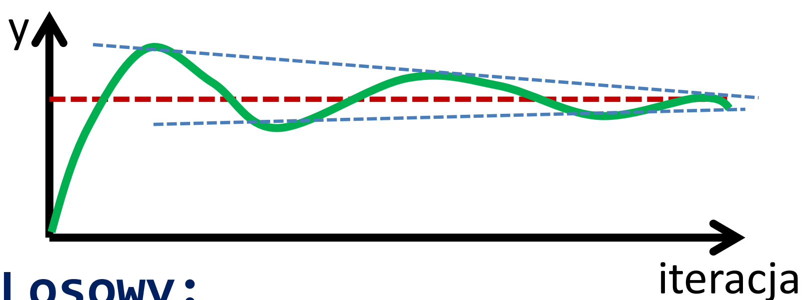
Charakter zbieżności

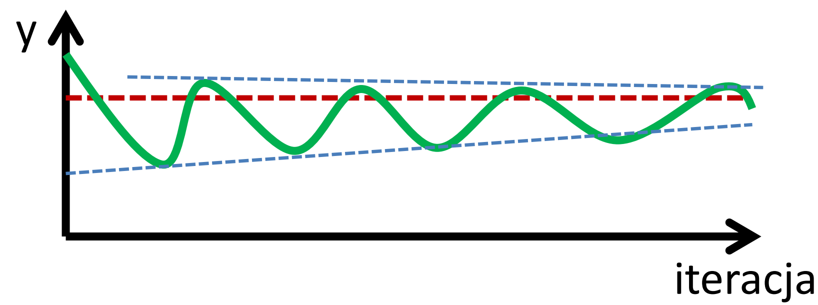
Asymptotyczny:



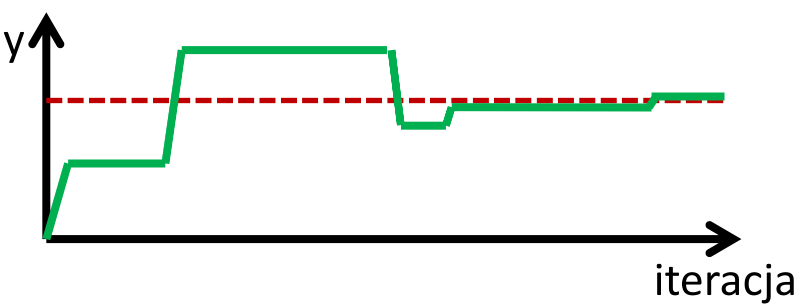


Oscylacyjny:





Losowy:



Wartość prawdziwa

Oszacowanie

Trend zanikania drgań

Kryterium uzyskania gorszego wyniku

Kryterium uzyskania gorszego wyniku polega na tym, że przerywamy obliczenia, gdy wynik uzyskany w iteracji bieżącej jest gorszy on aktualnie najlepszego.

UWAGI:

- Kryterium to jest bardzo proste w implementacji.
- Możemy je zastosować, gdy mamy możliwość określenia, czy rozwiązanie jest gorsze, czy lepsze.
- Aby kryterium było skuteczne musimy mieć pewność, że w chwili uzyskania gorszego wyniku, algorytm nie jest już w stanie znaleźć lepszego.

Przykład:

Poszukiwanie minimum funkcji kwadratowej, ze stałym krokiem i kierunkiem:

+ Funkcja ta posiada tylko jedno minimum.
+ Mniejsza wartość funkcji w danym punkcie, jest lepsza.
+ Uzyskanie wartości większej oznacza, że wspinamy się w górę paraboli (oddalamy od szukanego minimum).

Uzyskanie gorszego wyniku - przykład


```cpp
double funMat(double x)
{
  return x*x + 2 * x + 1; // minumim w (-1,0)
}

void main(void)
{
  double xPocz, xKon, dx, y, yMin, xMin;
  cout << "Podaj poczatek przedzialu: "; cin >> xPocz;
  cout << "Podaj krok poszukiwania: ";   cin >> dx;
  cout << "Podaj koniec przedzialu: ";   cin >> xKon;
  double eps = 0.01*dx;
  xMin = xPocz;
  yMin = funMat(xMin);

  for (double x = xPocz + dx; x < xKon + eps; x += dx)
  {
    y = funMat(x);
    cerr << "y: " << y << " w pkt.: " << x << endl;

    if (y < yMin)
    {
      yMin = y;
      xMin = x;
    }
    else break;
  }
  cout << "Minimum: " << yMin << " w pkt.: " << xMin << endl;
}
```

FUNKCJA, KTÓRA SPEŁNIA ZAŁOŻENIA

OKREŚLNIE OBSZARU POSZUKIWAŃ I ROZWIĄZANIA WSTĘPNEGO

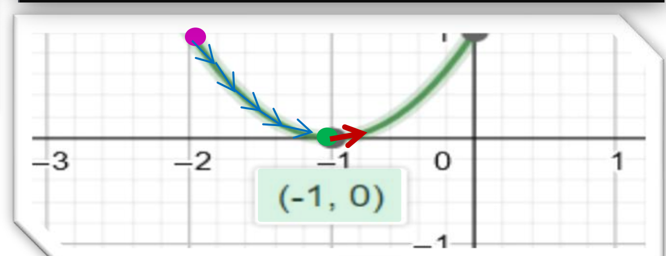
PRÓBKOWANIE FUNKCJI

ZAPAMIĘTANIE NAJLEPSZEGO WYNIKU

KRYTERIUM ZBIEŻNOŚCI

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -2.0
Podaj krok poszukiwania: 0.1
Podaj koniec przedzialu: 0.0
y: 0.81 w pkt.: -1.9
y: 0.64 w pkt.: -1.8
y: 0.49 w pkt.: -1.7
y: 0.36 w pkt.: -1.6
y: 0.25 w pkt.: -1.5
y: 0.16 w pkt.: -1.4
y: 0.09 w pkt.: -1.3
y: 0.04 w pkt.: -1.2
y: 0.01 w pkt.: -1.1
y: 0 w pkt.: -1
y: 0.01 w pkt.: -0.9
Minimum: 0 w pkt.: -1
Press any key to continue . . .
```



WAŻNE:

+ Dodanie przerwania obliczeń, gdy nie uzyskamy poprawy, przyspiesza znalezienie rozwiązania.

Uzyskanie gorszego wyniku – zły przykład

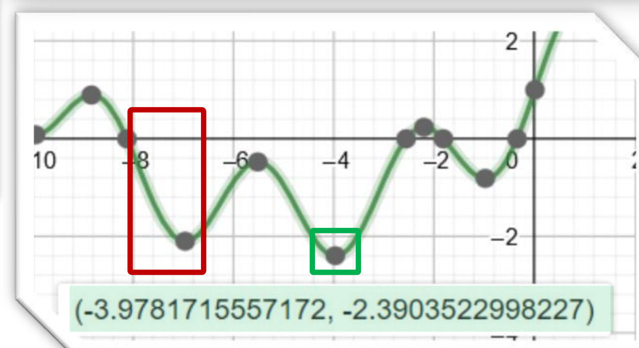

```
double funMat(double x)
{
  return 0.1 * x * x + x + 1 + sin(2 * x);
}

void main(void)
{
  double xPocz, xKon, dx, y, yMin, xMin;
  cout << "Podaj poczatek przedzialu: "; cin >> xPocz;
  cout << "Podaj krok poszukiwania: ";   cin >> dx;
  cout << "Podaj koniec przedzialu: ";   cin >> xKon;
  double eps = 0.01*dx;
  xMin = xPocz;
  yMin = funMat(xMin);

  for (double x = xPocz + dx; x < xKon + eps; x += dx)
  {
    y = funMat(x);
    cerr << "y: " << y << " w pkt.: " << x << endl;


    if (y < yMin)
    {
      yMin = y;
      xMin = x;
    }
    else break;
  }
  cout << "Minimum: " << yMin << " w pkt.: " << xMin << endl;
}
```

FUNKCJA, KTÓRA NIE SPEŁNIA ZAŁOŻENIA

KRYTERIUM ZBIEŻNOŚCI

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -8.0
Podaj krok poszukiwania: 0.1
Podaj koniec przedzialu: 0.0
y: -0.567093 w pkt.: -7.9
y: -0.823754 w pkt.: -7.8
y: -1.07412 w pkt.: -7.7
y: -1.3104 w pkt.: -7.6
y: -1.52529 w pkt.: -7.5
y: -1.71225 w pkt.: -7.4
y: -1.86579 w pkt.: -7.3
y: -1.98166 w pkt.: -7.2
y: -2.05703 w pkt.: -7.1
y: -2.09061 w pkt.: -7
y: -2.0827 w pkt.: -6.9
Minimum: -2.09061 w pkt.: -7
```



WAŻNE:

+ Algorytm kończy się przedwcześnie, ze względu na minima lokalne. Trzeba potrafić dobrać algorytm do zadania.

Limity iteracji

Kryteria limitu operacji polegają ustaleniu liczby iteracji, które mogą się odbyć w ogóle, i/lub bez uzyskania poprawy (stagnacja).

UWAGI:

- Odgórny limit iteracji jest zawsze użyteczny, gdyż gwarantuje, że obliczenia wykonane zostaną w skończonym czasie.
- Ograniczenie na liczbę iteracji bez poprawy (stagnacja), pozwala na skrócenie czasu obliczeń. Jest użyteczne głównie w algorytmach, które poszukują rozwiązania w sposób losowy (np. metoda Monte-Carlo) i wymaga możliwości określenia jakości rozwiązania.
- Stosując kryterium stagnacji, należy pamiętać o resetowaniu licznika stagnacji, w chwili znalezienia lepszego rozwiązania
- Oba przypadki mogą prowadzić do przedwczesnego zakończenia obliczeń, nie dając kontroli nad błędem wyniku. Więcej iteracji może, ale nie musi przełożyć się na mniejszy błąd wyniku końcowego.

  Przykład:

  Poszukiwanie minimum funkcji o wielu minimach lokalnych.

Limity iteracji - przykład


```cpp
double funMat(double x)
{
  return 0.1 * x * x + x + 1 + sin(2 * x);
}

void main(void)
{
  int maxIter, maxStag, liczStag = 0, iter = 0;
  double xPocz, xKon;
  cout << "Podaj poczatek przedzialu: ";     cin >> xPocz;
  cout << "Podaj koniec przedzialu: ";       cin >> xKon;
  cout << "Podaj limit iteracji: ";          cin >> maxIter;
  cout << "Podaj limit stagnacji: ";         cin >> maxStag;

  double xMin = xPocz, yMin = funMat(xPocz);
  srand(time(0));

  for (iter = 0; iter < maxIter && liczStag < maxStag; iter++ )
  {
    double x = xPocz + (1.0*rand() / RAND_MAX)*(xKon - xPocz);
    double y = funMat(x);

    if (y < yMin)
    {
      yMin = y; xMin = x;
      liczStag = 0;
    }
    else liczStag++;
  }
  cout << "Minimum: " << yMin << " w pkt.: " << xMin << endl;
  cout << "Iter: " << iter << " / " << maxIter << endl;
}
```

KRYTERIA ZBIEŻNOŚCI

OKREŚLANIE STAGNACJI

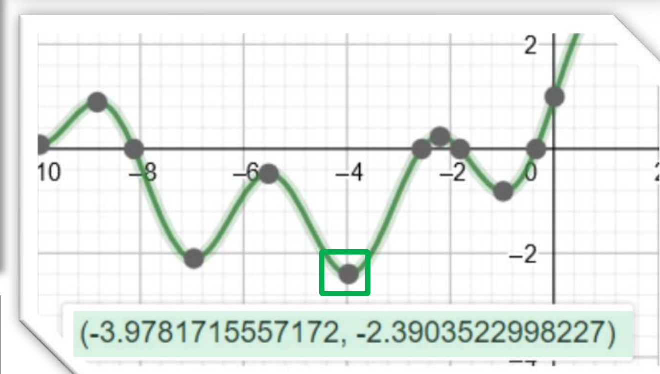
POSZUKIWANIE LOSOWE

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -10.0
Podaj koniec przedzialu: 0.0
Podaj limit iteracji: 1000
Podaj limit stagnacji: 10
Minimum: -2.33152 w pkt.: -3.81054
Iter: 20 / 1000
```

C:\Windows\system32\cmd.exe

```
Podaj poczatek przedzialu: -10.0
Podaj koniec przedzialu: 0.0
Podaj limit iteracji: 1000
Podaj limit stagnacji: 100
Minimum: -2.39013 w pkt.: -3.98846
Iter: 123 / 1000
```



WAŻNE:

+ Limit iteracji i kryterium stagnacji.

Błąd wielkości powiązanej

Kryterium oparte o dopuszczalny błąd wielkości powiązanej polega na obliczeniu faktycznego błędu znanej wartości powiązanej z szacowanym wynikiem i sprawdzeniu czy błąd ten jest dopuszczalny.

UWAGI:

- W rozwiązywanym problemie musi istnieć możliwość przeliczenia wyniku na wartość powiązaną, która jest znana.
- Zazwyczaj wartość powiązaną uzyskujemy przez odwrócenie pierwotnego problemu co wymaga, aby rozwiązywany problem był łatwo odwracalny.
- Kryterium wymaga określenia zakresu tolerancji dla błędu (podobnie jak przy porównywaniu wartości rzeczywistych).
- Kryterium to ma wąskie spektrum zastosowań, ale często daje najlepsze rezultaty.

  Przykład:

  Iteracyjne obliczając wartość pierwiastka liczby x, można wyliczone już oszacowanie łatwo podnieść do kwadratu i obliczyć błąd w odniesieniu do wartości x, którą znamy.

Błąd wielkości powiązanej - przykład


```cpp
void main(void)
{
  double x, eps; int i = 1;
  cout << "Podaj x: "; cin >> x;
  cout << "Podaj dokladnosc: "; cin >> eps;

  double error = x, sqrtX = x;

  while (error >= eps)
  {
    sqrtX = 0.5*(sqrtX + (x / sqrtX));

    error = fabs(x - sqrtX * sqrtX);

    cerr << i++ << ": " << sqrtX
         << " err: " << error << endl;
  }

  cout << "Wynik: " << sqrtX << endl;
  cout << "Prawidlowy: " << sqrt(x) << endl;
}
```

OSZACOWANIA POCZĄTKOWE

KRYTERIUM ZBIEŻNOŚCI

OSZACOWANIE WYNIKU

C:\Windows\system32\cmd.exe

```
Podaj x: 2.0
Podaj dokladnosc: 0.001
1: 1.5 err: 0.25
2: 1.41667 err: 0.00694444
3: 1.41422 err: 6.0073e-06
Wynik: 1.41422
Prawidlowy: 1.41421
```

C:\Windows\system32\cmd.exe

```
Podaj x: 2.0
Podaj dokladnosc: 0.00000001
1: 1.5 err: 0.25
2: 1.41667 err: 0.00694444
3: 1.41422 err: 6.0073e-06
4: 1.41421 err: 4.51061e-12
Wynik: 1.41421
Prawidlowy: 1.41421
```

WAŻNE:

+ należy zacząć od wstępnego oszacowania błędu (nawet jeśli jest bardzo zgrubne),
+ w tym przykładzie błąd szacujemy rozwiązując zadanie odwrotne (potęgowanie),
+ pierwiastek szacujemy jako średnią z poprzedniego oszacowania i wyniku podzielenia x przez nie.

Przyrost oszacowania wyniku

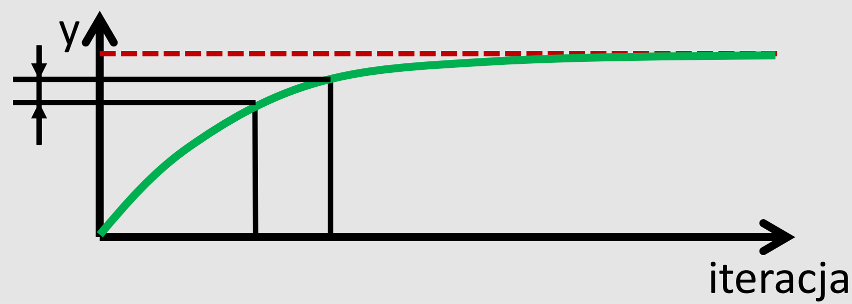
Kryterium oparte o dopuszczalny przyrost oszacowania wyniku polega na wyliczeniu różnicy oszacowania wyniku w dwóch (lub więcej) kolejnych iteracjach. Jeśli ta różnicą mieści się w dopuszczalnym zakresie to przerywamy obliczenia.

UWAGI:

- To kryterium może być szeroko stasowane, nawet gdy nie możemy ustalić czy nowe rozwiązanie jest lepsze od poprzedniego.
- Jest podobne do kryterium stagnacji tylko, że operuje na wartościach oszacowania, a nie na liczbie iteracji.
- Może prowadzić do przedwczesnego zakończenia obliczeń, szczególnie jeśli metoda wyznaczania oszacowań nie gwarantuje, że każde kolejne oszacowanie jest dokładniejsze.

Przykład:

Iteracyjne obliczanie wartości pierwiastka liczby x.



Przyrost oszacowania - przykład


```cpp
void main(void)
{
  double x, eps; int i = 1;
  cout << "Podaj x: "; cin >> x;
  cout << "Podaj dokladnosc: "; cin >> eps;
  double error = x, sqrtX = x, popSqrtX = x;

  while (error >= eps)
  {
    sqrtX = 0.5*(sqrtX + (x / sqrtX));

    error = fabs(popSqrtX - sqrtX );

    popSqrtX = sqrtX;

    cerr << i++ << ": " << sqrtX
         << " err: " << error << endl;
  }
  cout << "Wynik: " << sqrtX << endl;
  cout << "Prawidlowy: " << sqrt(x) << endl;
}
```

OSZACOWANIA POCZĄTKOWE

KRYTERIUM ZBIEŻNOŚCI

OSZACOWANIE WYNIKU

ZAPAMIĘTANIE OSZACOWANIA

C:\Windows\system32\cmd.exe

```
Podaj x: 3.0
Podaj dokladnosc: 0.1
1: 2 err: 1
2: 1.75 err: 0.25
3: 1.73214 err: 0.0178571
Wynik: 1.73214
Prawidlowy: 1.73205
```

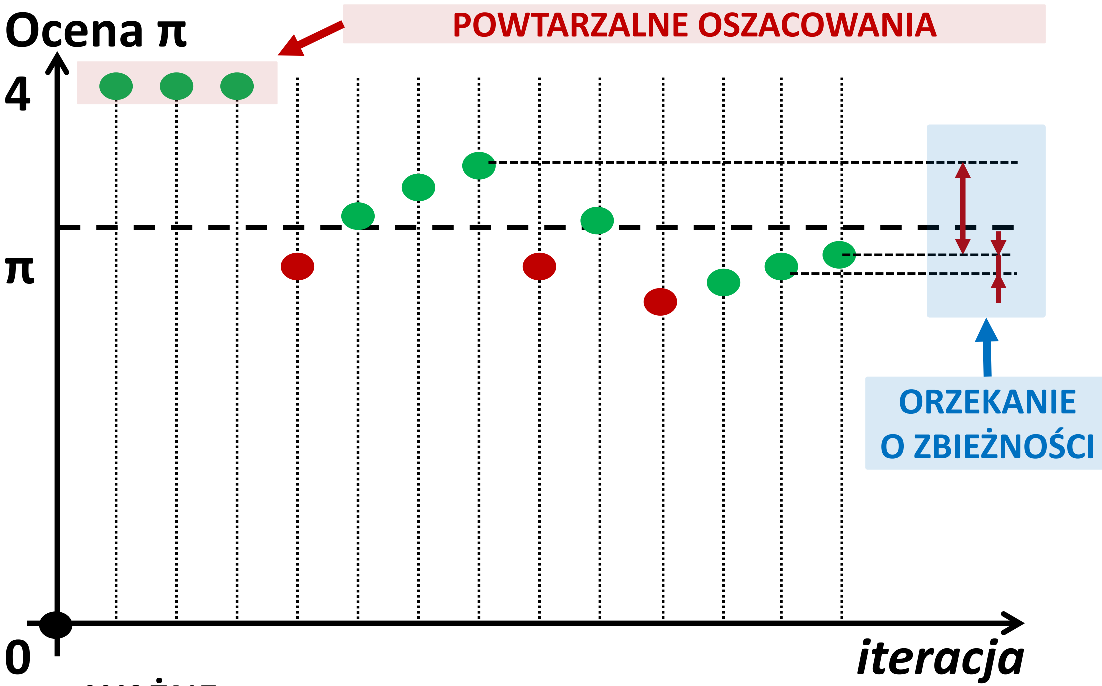
C:\Windows\system32\cmd.exe

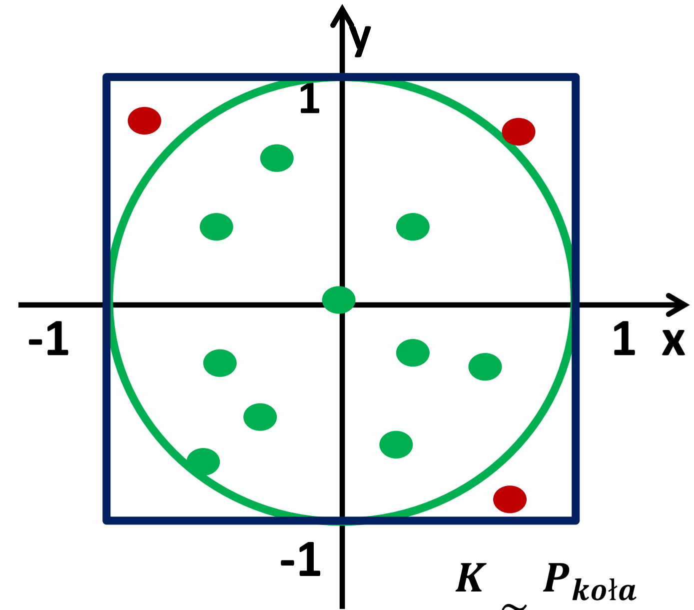
```
Podaj x: 3.0
Podaj dokladnosc: 0.001
1: 2 err: 1
2: 1.75 err: 0.25
3: 1.73214 err: 0.0178571
4: 1.73205 err: 9.20471e-05
Wynik: 1.73205
Prawidlowy: 1.73205
```

WAŻNE:

+ początkowy błąd trzeba założyć arbitralnie tak, aby spełnić warunek działania pętli,
+ błąd szacujemy jako różnicę między aktualnym oszacowaniem o poprzednim,
+ na koniec iteracji aktualne oszacowanie zapamiętujemy - nie wcześniej, bo zakłóci obliczenia!

Metoda Monte Carlo – poszukiwanie liczby π





$$\frac{K}{N} \approx \frac{P_{koła}}{P_{kw.}}$$

$$\frac{K}{N} \approx \frac{\pi}{4}$$

$$\pi \approx \frac{4K}{N}$$

WAŻNE:

+ metoda Monte Carlo pozwala oszacować wynik przez przeprowadzenie duże liczby losowych prób i klasyfikację wyników. Oszacowanie jest tym lepsze ile jest prób.
+ różnie można dobierać koncepcje orzekania o zbieżności algorytmu.
+ przy takiej klasyfikacji łatwo otrzymać dwa identyczne oszacowania (szczególnie) na początku, wiec warto błąd szacować dopiero po pewnej istotnej liczbie iteracji.

Metoda Monte Carlo – poszukiwanie liczby π

$$\pi \approx \frac{4K}{N}$$

```cpp
constexpr int MAKS_ITER = 10000;
constexpr double TOLERANCJA = 0.00100;
constexpr int CZEST_SPR = 100;

int liczPkt = 0, liczPktKola = 0;
double x,y, ocenaPI, popOcPI = 1.0;
srand(time(0));

 for (int i = 0; i < MAKS_ITER; ++i)
{
  x = 2.0 * (1.0 * rand() / RAND_MAX) - 1.0;
  y = 2.0 * (1.0 * rand() / RAND_MAX) - 1.0;
  liczPkt++;

  if ((x*x + y*y) <= 1.0)
    liczPktKola++;

  ocenaPI = 4.0 * liczPktKola / liczPkt;

  if (!(i % CZEST_SPR))
  {
    if (fabs(ocenaPI - popOcPI) < TOLERANCJA)
    break;
  else
    popOcPI = ocenaPI;
  }
}
cout << "Ocena PI: " << ocenaPI << "\nz toleran.: " << TOLERANCJA
     << "\npo: " << liczPkt << "/" << MAKS_ITER << " iter.\n";
```

WARTOŚĆ POCZĄTKOWA ROŻNA OD 0.0 i 4.0

TWARDY WAR. KOŃCA

LOSOWNIE PUNKTU W KWADRACIE

SPRAWDZENIE PRZYNALEŻNOŚCI DO KOŁA

OSZACOWANIE π

SPRAWDZENIE KRYTERIUM ZBIEŻNOŚCI

C:\WINDOWS\system32\cmd.exe

```
Ocena PI: 3.15646
z toleran.: 0.0001
po: 8801/10000 iter.
```

C:\WINDOWS\system32\cmd.exe

```
Ocena PI: 3.1438
z toleran.: 1e-07
po: 1000000/1000000 iter.
```

WAŻNE:

+ wartość początkową musimy dobrać tak, aby była odporna na powtarzalne oszacowania początkowe,
+ kryterium zbieżności sprawdzamy, co pewną liczbę iteracji, gdyż w tym przypadku niewielka różnica miedzy kolejnymi nie musi wcale świadczyć o bliskości do prawdziwego wyniku,
+ losowanie współrzędnych to dwa osobne losowania (x i y).

Zaawansowana parametryzacja pętli

```
double pocz, kon, krok;
cout << "Podaj pocz: "; cin >> pocz;
cout << "Podaj krok: "; cin >> krok;
cout << "Podaj kon : "; cin >> kon;


for (double x = pocz; x < kon + 0.01 * krok; x += krok)
{
  cout << "x: " << x << " x^2: " << x*x << endl;
}


for (double x = pocz, e = kon + 0.01 * krok; x < e; x += krok)
{
  cout << "x: " << x << " x^2: " << x*x << endl;
}


for (int i = 0,j = 0; (i - j) < 20; i++, j--)
{
  cout << "i-j: " << i - j << endl;
}
```

C:\WINDOWS\system32\cmd.exe

```
Podaj pocz: -0.3
Podaj krok: 0.3
Podaj kon : 0.3

x: -0.3 x^2: 0.09
x: 0 x^2: 0
x: 0.3 x^2: 0.09

x: -0.3 x^2: 0.09
x: 0 x^2: 0
x: 0.3 x^2: 0.09

i-j: 0
i-j: 2
i-j: 4
i-j: 6
i-j: 8
i-j: 10
i-j: 12
i-j: 14
i-j: 16
i-j: 18
```

WAŻNE:

+ obliczenie niezmiennej części złożonego warunku działania można wykonać w ramach instrukcji początkowych i przechować wynik w zmiennej (lokalnej dla pętli),
+ nie ma problemu aby w szczególnych przypadkach, sterować pętlą za pomocą dwóch zmiennych, jednocześnie (poszczególne instrukcje oddzielmy w tedy przecinkami).

Oszacowanie czasu wykonania kodu


```
1)    void main(void)
2)    {
3)      srand(time(0));
4)      constexpr size_t ROZMIAR = 1e3,
                         POWTORZENIA = 1e6;
5)      int tablica[ROZMIAR];

6)      clock_t start = clock();

7)      for (size_t iter = 0; iter < POWTORZENIA; iter++)
8)      {
9)        for (size_t elem = 0; elem < ROZMIAR; elem++)
10)         tablica[elem] = rand();
11)     }

12)     clock_t koniec = clock();
13)     clock_t cykle = koniec - start;
14)     double czas = (double) cykle / CLOCKS_PER_SEC;

15)     cout << fixed << setprecision(6);
16)     cout << "Calk. czas wykonania kodu: "
             << czas << "s" << endl;
17)     cout << "Sredni czas poj. iteracji: "
             << 1e6*czas / POWTORZENIA << "us" << endl;
18)   }
```

ROZPOCZĘCIE POMIARU CZASU

FRAGMENT BADANEGO KODU

ZAKOŃCZENIE POMIARU CZASU

WAŻNE:

+ czas wykonania pojedynczej instrukcji jest często ta mały że nie da się go zmierzyć przy dostępnej w komputerze rozdzielczości zegara. Dlatego mierzymy czas wielu jej powtórzeń i szacujemy średni czas wykonania.
+ takie oszacowanie, ogranicza też wpływ czynników zewnętrznych na pomiar (chwilowe obciążenie procesora),
+ pomiar wyrażony jest w liczbie cykli procesora, więc czas otrzymamy dzieląc tą liczbę, przez liczbę cykli na sekundę.

C:\Windows\system32\cmd.exe — **JEDNA ITERACJA**

```
Calk. czas wykonania kodu: 0.000000s
Sredni czas poj. iteracji: 0.000000us
```

C:\Windows\system32\cmd.exe — **DUŻO ITERACJI**

```
Calk. czas wykonania kodu: 5.448000s
Sredni czas poj. iteracji: 54.480000us
```

Prostowanie pętli


```
double suma = 0.0; double dx = 0.00000001;
clock_t start = clock(); double czas;

for (double x = dx; x < 1.000; x+=dx)
  suma += x;

czas = (double)(clock() - start) / CLOCKS_PER_SEC;
cout << "Suma: " << suma << " in: " << czas << endl;

start = clock(); suma = 0.0;

for (double x = dx; x < 1.000; x += 4*dx)
{
  suma += x;
  suma += x;
  suma += x;
  suma += x;
}

czas = (double)(clock() - start) / CLOCKS_PER_SEC;
cout << "Suma: " << suma << " in: " << czas << endl;
}
```

WYKONUJEMY 10’000’000 RAZY:
x += dx; suma += x; oraz x < 1.000

```
C:\Windows\system32\cmd.exe
Suma: 5e+07 in: 0.241
Suma: 5e+07 in: 0.225
Press any key to continue . . .
```

WYKONUJEMY 10’000’000 RAZY:
suma += x;
ALE TYLKO 2’500’000 RAZY:
x += 4 * dx; oraz x < 1.000;

WAŻNE:

+ prostowanie pętli polega na celowym łamaniu zasady DRY, gdy wydajność jest absolutnie kluczowa,

+ pozwala ono zmniejszyć liczbę inkrementacji i porównań n-razy, co trochę przyspiesza pracę.

Maszyna Duff’a


```
constexpr int SIZE = 777;
int srcTab[SIZE], dstTab[SIZE];
int *src = srcTab, *dst = dstTab;


int n = (SIZE + 7) / 8;
```

OBLICZENIE LICZBY ITERACJI PĘTLI

WYZNACZENIE PUNKTU STARTU PIERWSZEJ ITERACJI

```
switch (SIZE % 8)
{
case 0: do { *dst++ = *src++;
case 7:      *dst++ = *src++;
case 6:      *dst++ = *src++;
case 5:      *dst++ = *src++;
case 4:      *dst++ = *src++;
case 3:      *dst++ = *src++;
case 2:      *dst++ = *src++;
case 1:      *dst++ = *src++;
} while (--n > 0);
}
cout << boolalpha << "Takie same: "
     << !memcmp(dstTab,srcTab,SIZE*sizeof(int))
     << endl;
```

TO JEST PĘTLA do-while

WAŻNE:

- \+ switch pozwala wskoczyć we właściwe miejsce pierwsze iteracji pętli while,
- \+ dzięki temu liczba operacji nie musi być podzielna przez liczbę powtórzeń instrukcji w ciele pętli,
- \+ obliczanie liczby iteracji pętli odbywa się na bazie liczby operacji i liczby powtórzeń w ciele pętli,
- \+ dekrementacja licznika następuje przy sprawdzaniu warunku działania,
- \+ operacje wykonywane na wskaźnikach są niezależne od licznika pętli, więc nadają się do linearyzacji pętli.

C:\Windows\system32\cmd.exe

```
Takie same: true
Press any key to continue . . .
```

Wyłączenie operacji poza pętlę


```
double a = 2.0, yMax = -10e10, dx = 1e-7;
clock_t start = clock(); double czas;
for (double x = 1.0; x < 10.0; x += dx)
{
  if (a > 0.0)
  {
    double y = log2(a)*sin(2 * 3.1415*x) + 2;
    if (y > yMax) yMax = y;
  }
}
czas = (double)(clock() - start) / CLOCKS_PER_SEC;
cout << "yMax: " << yMax << " in: " << czas << endl;
```

WOLNE

```
yMax = -10e10; start = clock();
const double LOG2A = log2(a), OMEGA = 2 * 3.1415;
if (a > 0.0)
{
  for (double x = 1.0; x < 10.0; x += dx)
  {
    double y = LOG2A*sin(OMEGA*x) + 2;
    if (y > yMax) yMax = y;
  }
}
czas = (double)(clock() - start) / CLOCKS_PER_SEC;
cout << "yMax: " << yMax << " in: " << czas << endl;
```

SZYBKIE

WAŻNE:

- przed pętlę można wyłączyć te operacje które nie są zależne od jej zmiennej sterującej (tu x),
- jeśli w pętli jest if, który nie zależy od zmiennej sterujące, to lepiej jest zamienić kolejność zagnieżdżenia,
- tego typu przekształcenia często znacznie podnoszą wydajność, ale wymagają wiele uwagi.

C:\Windows\system32\cmd.exe — dx = 1e-5

```
yMax: 3 in: 0.076
yMax: 3 in: 0.016
```

C:\Windows\system32\cmd.exe — dx = 1e-6

```
yMax: 3 in: 0.624
yMax: 3 in: 0.156
```

C:\Windows\system32\cmd.exe — dx = 1e-7

```
yMax: 3 in: 6.254
yMax: 3 in: 1.553
```

Scalanie pętli


```
constexpr long long LIMIT = 1e7; int n, silnia, fib, fibPre;
clock_t start = clock(); double czas;
cout << "Podaj n: "; cin >> n; cout << "OSOBNO:\n";
```

```
for (long long i = 0; i < LIMIT; i++)
{
  silnia = 1, fib = 1, fibPre = 0;
  for (int i = 2; i <= n; i++)
    silnia *= i;
  for (int i = 2; i <= n; i++)
  {
    fib += fibPre;
    fibPre = fib - fibPre;
  }
}
czas = 1.0*(clock() - start) / CLOCKS_PER_SEC;
cout << "Silnia: " << silnia << " Fibo: " <<
     fib << "\nw czasie: " << czas << endl;
```

PĘTLA I

PĘTLA II

```
cout << "RAZEM:\n"; start = clock();
for (long long i = 0; i < LIMIT; i++)
{
  silnia = 1; fib = 1, fibPre = 0;
  for (int i = 2; i <= n; i++)
  {
    silnia *= i;
    fib += fibPre;
    fibPre = fib - fibPre;
  }
}
czas = 1.0*(clock() - start) / CLOCKS_PER_SEC;
cout << "Silnia: " << silnia << " Fibo: " <<
fib << "\nw czasie: " << czas << endl;
```

PĘTLA SCALONA: I + II

C:\Windows\system32\cmd.e

```
Podaj n: 4
OSOBNO:
Silnia: 24 Fibo: 3
w czasie: 0.093
RAZEM:
Silnia: 24 Fibo: 3
w czasie: 0.069
```

C:\Windows\system32\cmd.exe

```
Podaj n: 15
OSOBNO:
Silnia: 2004310016 Fibo: 610
w czasie: 0.477
RAZEM:
Silnia: 2004310016 Fibo: 610
w czasie: 0.316
```

WAŻNE:

+ gdy obie pętle mają dokładnie ten sam zakres zmian zmiennej sterującej, oraz końcowy wynik jednej nie jest potrzebny drugiej, to można je scalić w jedną,
+ eliminujemy w ten sposób część operacji, które służą tylko do obsługi działania pętli.

Zamiana kolejności zagnieżdżenia pętli


```
long long suma = 0, i, j;
const int HIGH = 1e5, LOW = 1e3;
clock_t start = clock(); double czas;
for (i = 1; i <= HIGH; i++)
{
  for (j = 1; j <= LOW; j++)
    suma += i % j;
}                                         PĘTLA I

czas = 1.0*(clock() - start)
       / CLOCKS_PER_SEC;
cout << "Suma: " << suma
     << " Czas: " << czas << endl;

suma = 0;
start = clock();
for (i = 1; i <= LOW; i++)
{
  for (j = 1; j <= HIGH; j++)
    suma += j % i;
}                                         PĘTLA II
czas = 1.0*(clock() - start)
       / CLOCKS_PER_SEC;
cout << "Suma: " << suma
     << " Czas: " << czas << endl;
```

WAŻNE:

+ zewnętrzna pętla (I) wykonuje się HIGH razy, a wewnętrzna HIGH * LOW razy:
  - 1 + HIGH inicjalizacji zmiennej sterującej,
  - HIGH + HIGH * LOW inkrementacji,
  - HIGH + HIGH * LOW sprawdzeń warunku,
+ zewnętrzna pętla (II) wykonuje się LOW razy, a wewnętrzna LOW * HIGH razy:
  - 1 + LOW inicjalizacji zmiennej sterującej,
  - LOW + LOW * HIGH inkrementacji,
  - LOW + LOW * HIGH sprawdzeń warunku,
+ zysk jest zauważalny tylko gdy HIGH jest znacząco większe niż LOW,
+ warunkiem użycia tej techniki jest niezależność kolejności pętli.

C:\Windows\system32\cmd.exe **1e7/10**

```
Suma: 224999994 Czas: 0.76
Suma: 224999994 Czas: 0.635
```

C:\Windows\system32\cmd.exe **1e5/1e3**

```
Suma: 24947345693 Czas: 0.642
Suma: 24947345693 Czas: 0.64
```

Quiz sprawdzający wiedzę

- **Z czego wynika błąd numeryczny?**

  Z niedokładności zapisu liczb na skończonej liczbie bitów w systemie binarnym.

- **Jak skompensować błąd numeryczny przy porównaniach?**

  Wprowadzając niewielką tolerancję na wartość. Inaczej dopuszczając aby wartość wyliczana znajdowała się w pewnym wąskim zakresie wokół wartości odniesienia.

- **Kiedy błąd numeryczny osiąga znaczne wartości?**

  Gdy sumujemy bardzo dużo liczb rzeczywistych. Szczególnie gdy używamy do ich reprezentacji typu float.

Quiz sprawdzający wiedzę

- Czy błędy numeryczne pojawiają się przy obliczaniach na liczbach całkowitych?

  Nie takie obliczenia są zawsze dokładne. Są oczywiście inne źródła błędów w takich obliczeniach.

- Co można zrobić, gdy pętla wykonuje za mało iteracji z powodu błędu numerycznego na zamiennej sterującej?

  Dodać do wartości końcowej (tej zapisanej w warunku) małą liczbę (mniejszą od kroku inkrementacji).

- Jakie są typowe rodzaje zbieżności?

  Asymptotyczna, Oscylacyjna i losowa.

Quiz sprawdzający wiedzę

- **Na czym polega kryterium gorszego wyniku?**

  Na przerwaniu obliczeń zaraz po tym gdy wynik obliczeń będzie gorszy od poprzedniego. Musi istnieć kryterium oceny, który wynik jest lepszy/gorszy.

- **Na czym polega kryterium stagnacji?**

  Gdy przez zadaną liczbę iteracji nie uda się wypracować lepszego wyniku. Licznik iteracji należy resetować, gdy jednak znajdziemy lepszy wynik.

- **Jaka jest ogólna zasada działania metody Monte-Carlo?**

  Na losowym próbkowaniu przestrzeni rozwiązań zadania.

Quiz sprawdzający wiedzę

- Co jest warunkiem stosowania linearyzacji pętli?

  Niezależność powielanej instrukcji od licznika pętli. W innych wypadkach konieczność przeliczania wartości licznika znosi cały zysk z linearyzacji.

- Która metoda optymalizacji pętli daje najbardziej obiecującą poprawę?

  Wyciąganie obliczeń poza pętlę. Jest to najtrudniejsza metoda, gdyż wymaga uważnej oceny co można, a co nie można wycinać przed pętlę. Jest ona także najbardziej efektywna obecnie, gdyż przez to, że nie jest łatwa w automatyzacji nie jest wykonywana przez kompilatory automatycznie jak na przykład linearyzacja.

Quiz sprawdzający wiedzę

- **Dlaczego przy szacowaniu czasu wykonania kodu trzeba powtarzać operację wiele razy?**

  Głównie dlatego, że czas jej wykonania jest krótszy niż takt zegara, którego używamy do pomiaru. Drugi powód to kompensacja zakłóceń pomiarów wynikająca np.: z ciągle zmiennego obciążania procesora.

- **Ile operacji można wykonać w instrukcji kroku pętli for?**

  Dowolną ilość. Operacje oddzielamy przecinkiem.

- **Jak optymalizujemy zagnieżdżone pętle?**

  Umieszczamy pętlę o największej liczebności iteracji jako najbardziej wewnętrzną, a tą o najmniejszej jako najbardziej zewnętrzną. Ma to tylko sens przy dużych dysproporcjach w liczbie iteracji.

PYTANIA?