Zadanie 0 – metody Newtona i Steffensena (1 pkt)

Napisz szablony funkcji

```
template <class T>
bool newton(double &x, T f, double x0, double eps, int n = 1000);

template <class T>
bool steffensen(double &x, T f, double x0, double eps, int n = 1000);
```

które znajdą pierwiastek x funkcji f, zaczynając od punktu x0. Szukanie ma odbywać się do osiągnięcia dokładności mniejszej niż eps ($|x_{n+1} - x_n| < \text{eps}$) lub do przekroczenia maksymalnej liczby iteracji n. W pierwszym przypadku funkcja ma zwrócić *true*, w drugim przypadku lub w razie błędnych danych funkcja ma zwrócić *false*. W obu przypadkach najlepsze przybliżenie pierwiastka powinno zostać przypisane do x.
Przydatne wzory:

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)} \tag{1}$$

$$g_{Newt}(x_n) = f'(x_n) \tag{2}$$

$$g_{Steff}(x_n) = \frac{f(x_n + f(x_n)) - f(x_n)}{f(x_n)} \tag{3}$$

Szablon:

```
#include <cmath>
#include <iomanip>
#include <iostream>

template <class T>
bool newton(double &x, T f, double x0, double eps, int n = 1000);

template <class T>
bool steffensen(double &x, T f, double x0, double eps, int n = 1000);

double f1(double x) { return x * x; }

double f2(double x) { return x * x - 2.; }

double f3(double x) { return exp(x) + x - 1; }

int main() {
  double x;
  std::cout<<"Newton :"<<std::endl;
   for (auto fx : {f1, f2, f3}) {
      for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
        if (newton(x, fx, 1.4, eps)) {
          std::cout << std::setprecision(8) << "\teps = " << eps
                    << "\troot = " << x << std::endl;
        } else {
          std::cout << "Unable to find root" << std::endl;
          break;
```

```
        }
      }
      std::cout << std::endl;
    }
    std::cout<<"Stefensen :"<<std::endl;
   for (auto fx : {f1, f2, f3}) {
      for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
        if (steffensen(x, fx, 1.4, eps)) {
          std::cout << std::setprecision(8) << "\teps = " << eps
                    << "\troot = " << x << std::endl;
        } else {
          std::cout << "Unable to find root" << std::endl;
          break;
        }
      }
      std::cout << std::endl;
    }
}
```

Output:

Newton :
eps = 0.1 root = 0.0875
eps = 0.01 root = 0.00546875
eps = 0.001 root = 0.00068359375
eps = 0.0001 root = 8.5449219e-05
eps = 1e-07 root = 8.3446503e-08
eps = 0.1 root = 1.4142857
eps = 0.01 root = 1.4142136
eps = 0.001 root = 1.4142136
eps = 0.0001 root = 1.4142136
eps = 1e-07 root = 1.4142136
eps = 0.1 root = 0.00129105
eps = 0.01 root = 4.1679256e-07
eps = 0.001 root = 4.3299152e-14
eps = 0.0001 root = 4.3299152e-14
eps = 1e-07 root = 4.5362831e-19
Stefensen :
eps = 0.1 root = 0.070547949
eps = 0.01 root = 0.0093676589
eps = 0.001 root = 0.00059063152
eps = 0.0001 root = 7.3867098e-05
eps = 1e-07 root = 7.2141161e-08
eps = 0.1 root = 1.4144928
eps = 0.01 root = 1.4142137
eps = 0.001 root = 1.4142137
eps = 0.0001 root = 1.4142136
eps = 1e-07 root = 1.4142136
eps = 0.1 root = 1.343199
eps = 0.01 root = 1.0932516e-07
eps = 0.001 root = 1.0932516e-07

eps = 0.0001 root = 8.8532142e-15
eps = 1e-07 root = -2.8570037e-17

Zadanie 1 – metoda siecznych (1,67 pkt)

a)

Napisz szablon funkcji

```
template <class T>
bool secant(double &x, T f, double x0, double x1, double eps, int n=1000);
```

znajdujący metodą siecznych pierwiastek funkcji f przy punktach poczatkowych x0 i x1. Wynik powinien być wpisany do zmiennej x, która przekaże go na zewnątrz poprzez referencję. Algorytm powinien się wykonywać aż do osiągnięcia dokładności eps ($|x_{n+1} - x_n| < eps$), ale nie więcej niż n iteracji. Jeżeli algorytm osiągnie wymaganą dokładność, funkcja powinna zwrócić wartość true. Jeżeli limit iteracji zostanie osiągnięty zanim uzyskamy żądaną dokładność, to funkcja powinna zwrócić false. Należy sprawdzić poprawność argumentów: eps > 0 oraz $n > 0$ oraz f nie jest nullptr. Jeżeli argumenty są niepoprawne to należy zwrócić false. Wyświetlenie komunikatu o złych parametrach jest opcjonalne.
Przydatny wzór:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} \tag{4}$$

Przykład:

```
#include <cmath>
#include <iomanip>
#include <iostream>

template <class T>
bool secant(double &x, T f, double x0, double x1, double eps, int n = 1000);

double f1(double x) { return x * x; }

double f2(double x) { return x * x - 2.; }

double f3(double x) { return exp(x) + x - 1; }

double f4(double x) {return log(x)-x+1.0;}

double f5(double x) {return sin(x+0.5);}


int main() {
  double x;
  for (auto fx : {f1, f2, f3, f4, f5}) {
    for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
      if (secant(x, fx, 0.1, 3, eps)) {
        std::cout << std::setprecision(8) << "eps = " << eps
                  << "\t root = " << x << std::endl;
```

```
      } else {
        std::cout << "Unable to find root" << std::endl;
      }
    }
    std::cout << std::endl;
  }
}
```

Przykładowy output:
eps = 0.1 root = 0.047619048
eps = 0.01 root = 0.047619048
eps = 0.001 root = 0.00065890622
eps = 0.0001 root = 9.6132278e-05
eps = 1e-07 root = 7.0478158e-08
eps = 0.1 root = 1.4142151
eps = 0.01 root = 1.4142136
eps = 0.001 root = 1.4142136
eps = 0.0001 root = 1.4142136
eps = 1e-07 root = 1.4142136
eps = 0.1 root = 0.00097475004
eps = 0.01 root = 3.1624215e-09
eps = 0.001 root = 3.1624215e-09
eps = 0.0001 root = 1.0333911e-14
eps = 1e-07 root = -1.0218562e-16
eps = 0.1 root = 1.0597106
eps = 0.01 root = 1.0084194
eps = 0.001 root = 1.0007553
eps = 0.0001 root = 1.0000681
eps = 1e-07 root = 1.0000001
eps = 0.1 root = 2.6415926
eps = 0.01 root = 2.6415927
eps = 0.001 root = 2.6415927
eps = 0.0001 root = 2.6415927
eps = 1e-07 root = 2.6415927

Uwaga: Nie tworzyć żadnych tablic ani kolekcji w funkcji liczącej punkt zerowy funkcji. Należy sprawdzić poprawność argumentów. Należy pamiętać o zakończeniu działania funkcji w momencie znalezienia pierwiastka.

Podpowiedź: Zadanie jest niemal identyczne z podobnymi zadaniami z poprzedniej serii, trzeba tylko zmienić wzór.

b)

Rozbuduj program dodając do niego szablon funkcji

```
template <class T>
bool secantExtr(double &x, T f, double x0, double x1, double eps, int n = 1000)
```

który znajduje punkt ekstremalny funkcji f za pomocą metody siecznych.
Dodaj do maina fragment kodu sprawdzający działanie nowej funkcji:

```
  std::cout << "Extremal points:" << std::endl;
  for (auto fx : {f1, f2, f4, f5}) {
```

```
    for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
      if (secantExtr(x, fx, 0.3, 2, eps)) {
        std::cout << std::setprecision(8) << "eps = " << eps
                  << "\t point = " << x << std::endl;
      } else {
        std::cout << "Unable to find extremum" << std::endl;
      }
    }
    std::cout << std::endl;
  }
```

Nowy output:
Extremal points:
eps = 0.1 point = -9.0127144e-23
eps = 0.01 point = -9.0127144e-23
eps = 0.001 point = -9.0127144e-23
eps = 0.0001 point = -9.0127144e-23
eps = 1e-07 point = -9.0127144e-23
eps = 0.1 point = 0
eps = 0.01 point = 0
eps = 0.001 point = 0
eps = 0.0001 point = 0
eps = 1e-07 point = 0
eps = 0.1 point = 0.99944145
eps = 0.01 point = 1
eps = 0.001 point = 1
eps = 0.0001 point = 1
eps = 1e-07 point = 1
eps = 0.1 point = 1.0707965
eps = 0.01 point = 1.0707963
eps = 0.001 point = 1.0707963
eps = 0.0001 point = 1.0707963
eps = 1e-07 point = 1.0707963
Podpowiedź: Ekstremum funkcji f to pierwiastek równania $f'(x) = 0$. Pochodną w punkcie można policzyć wg jednego ze znanych nam już wzorów.

Iteratory

Iteratory są specjalnymi obiektami, które umożliwiają poruszanie się po kolekcjach z biblioteki standardowej C++. Do tej pory poznaliśmy już iterator $std :: vector < T >:: iterator$, którego czasem używaliśmy w pętli for do iterowania po kolekcji **std::vector**.
Do zrobienia zadań z tej serii wystarczy wiedzieć, że będziemy używać iteratorów jednokierunkowych, a więc takich dla których zdefiniowane są operatory inkrementacji(++), tożsamości (==, !=) oraz dereferencji (*). Operator dereferencji użyty na iteratorze pozwala uzyskać dostęp do wskazywanego obiektu w kolekcji.
Niektóre z iteratorów użytych w przykładach posiadają więcej operatorów i dodatkowe metody, ale do zrobienia zadań wystarczy użycie wymienionych 4 operatorów.

Zadanie 2 – sortowanie bąbelkowe (1 pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void bubbleSort(ForwardIt begin, ForwardIt end)
```

wykonującej sortowanie bąbelkowe pomiedzy dwoma iteratorami jednokierunkowymi typu ForwardIt.
Do zamiany wartości dwóch iteratorów a i b możesz użyć **std::iter swap(a,b)** z < *algorithm* > lub użyć dereferencji **std::swap(*a,*b)**.
Pamietaj, żeby zakończyć szybko sortowanie, jeżeli w poprzedniej iteracji żadne elementy nie zostały zamienione miejscami.
Szablon:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void bubbleSort(ForwardIt begin, ForwardIt end);

int main() {
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  bubbleSort(v.begin(), v.end());
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  bubbleSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //list
  std::list<int> l = { 7, 5, 16, 8 };
  bubbleSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.