Zadanie 0 – metoda siecznych (2pkt)

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:

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

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.

Zadanie 1 – podział projektu na kilka plików (1pkt)

Stwórz trzy pliki:

1. jk1_main.cpp
2. jk1.cpp
3. jk1.hpp

gdzie zamiast "jk" wstaw swoje inicjały. Skopiuj do "jk1_main.cpp" podaną niżej funkcję main

```
#include <iostream>//nie stawiaj spacji przed >
#include <cmath>

int main()
{
        TLorentzVector v(5, -2, -1, 3);
        std::cout << v.getX() << " " <<
                v.getY() << " " << v.getZ() << " "
```

```
                << v.getT() << std::endl;
        std::cout << v.Mag2() << " " << v.Mag()
                <<std::endl;

        double pi = 3.14159265359;
        v.setX(2*cos(pi/3)*sin(pi/2));
        v.setY(2*sin(pi/3)*sin(pi/2));
        v.setZ(2*cos(pi/2));
        v.setT(0);

        std::cout << v.getX() << " " <<
                v.getY() << " " << v.getZ() << " "
                << v.getT() << std::endl;
        std::cout << "phi=" << v.Phi() <<
        " theta=" << v.Theta() << std::endl;
        return 0;
}
```

W pliku nagłówkowym .hpp zdefiniuj klasę "TLorentzVector"[1]. Klasa ta ma reprezentować cztwerowektor, jakich używamy w teoriach relatywistycznych. Powinna mieć 4 prywatne składowe: x,y,z,t. Dla każdej ze składowych klasa ma mieć publiczne gettery i settery (trywialne!). Dodatkowo klasa ma dwa konstruktory:

```
TLorentzVector(); //ustawia wszedzie 0
TLorentzVector(double x, double y,
        double z, double t);
```

Konstruktor domyślny ustawia pola na 0. Działanie drugiego konstruktora jest (mam nadzieję) oczywiste. Dodatkowo klasa ma posiadać 4 metody:

```
double Mag2();
```

Metoda Mag2 liczy i zwraca długość czterowektora $s^2 = t^2 - x^2 - y^2 - z^2$.

```
double Mag();
```

Metoda Mag zwraca $\sqrt{s}$ jeśli $s >= 0$. W przeciwnym wypadku zwraca $-1 * \sqrt{-1 * s}$.

```
double Phi();
```

Metoda Phi zwraca kąt azymutalny (między współrzędnymi x i y) z przedziału $[0, 2\pi)$.

```
double Theta();
```

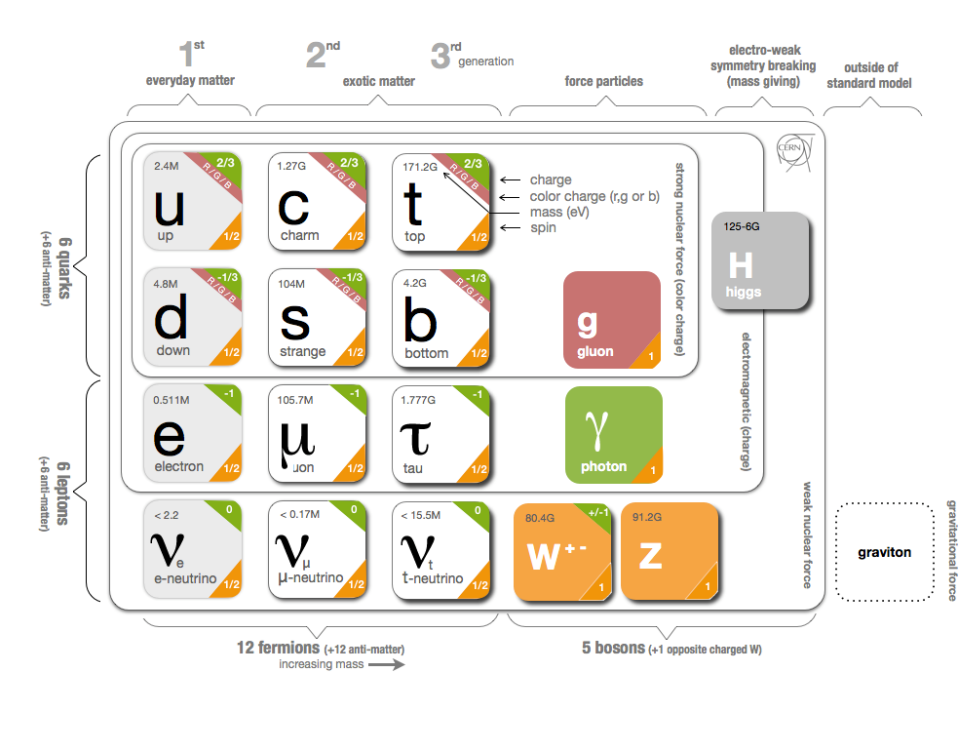
Metoda Theta zwraca kąt polarny, tj. kąt z przedziału $[-\pi, \pi]$, będący kątem pomiędzy osią OZ a kierunkiem wektora w 3D.
Uwaga: Wszystkie metody mają być zdefiniowane w pliku "jk1.cpp". W pliku nagłówkowym jedynie je deklarujemy, tzn. piszemy ich sygnaturę i po niej średnik, czyli pomijamy ciało metody w nawiasach klamrowych. W pliku .cpp musimy: 1) dołączyć plik nagłówkowy za pomocą include 2) definiując metody użyć odpowiedniej przestrzeni nazw, np. definiując gettera piszemy "double TLorentzVector::getX()return x;"
Podpowiedź: Krótki opis jak dzielić program na pliki oraz jak kompilować program z wielu plików jest w teoretycznym minimum. Jest tam również wyjaśnione, co to jest getter/setter.
Dla chętnych: Napisz i przetestuj globalną funkcję, która przyjmuje dwa czterowektory i zwraca ich iloczyn skalarny (w metryce Minkowskiego (+—)). Napisz i przetestuj kolejną funkcję, która zwraca interwał czasoprzestrzenny dla dwóch czterowektorów. **Output:**

[1] Wink wink!

5 -2 -1 3
-21 -4.58258
1 1.73205 -2.06823e-13 0
phi=1.0472 theta=1.5708

Zadanie 2 – Model Standardowy (2pkt)

Wykorzystaj dziedziczenie aby zaprogramować strukturę hierachiczną cząstek modelu standardowego. W tym celu należy stworzyć kilka klas opisanych poniżej, zdefiniować dla nich dziedziczenie (specyfikator dostępu public) oraz odpowiednie metody. Zadanie nie jest trudne, ale wymaga więcej kodu niż zazwyczaj. Ważne jest, żeby zachować porządek wcięć, klamer, nazw itd. inaczej zadanie zrobi się dużo trudniejsze.



Rysunek 1: Cząstki standardowe w Modelu Standardowym. Źródło: Google Image

Particle

Particle to klasa od której zaczniemy. Ma być abstrakcyjna, tzn. taka, której obiektu nie można utworzyć (patrz teoretyczne minimum). Powinna posiadać chronione pola:

```
protected:
        double m; //GeV
        double eCharge;
        long int PDG;
        bool red;
        bool green;
        bool blue;
        bool antiRed;
        bool antiBlue;
```

```
        bool antiGreen;
```

Gdzie kolejne pola zawierają informację o masie cząstki, jej ładunku w jednostkach ładunku elementarnego, kodzie cząstki. Dalej są zmienne, które opisują ładunek/antyładunek kolorowy[2]. Dla wszystkich pól należy zdefiniować metody typu get/set. Ponadto należy stworzyć 3 konstruktory:

```
        Particle();
        Particle(double mass, double charge, long int pDG);
        Particle(double mass, double charge, long int pdg,
                bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Pierwszy konstruktor (domyślny) ustawia wszystkie pola na 0. Drugi konstruktor ustawia wartości pól numerycznych, a pola odpowiadające kolorom na 0 (false). Ostatni konstruktor ustawia wszystkie pola. Ponieważ Particle ma być abstrakcyjna, to będzie potrzebować czysto wirtualnego destruktora

```
virtual ~Particle() = 0;
```

Boson

Boson to abstrakcyjna klasa reprezentująca bozony. Ma dziedziczyć (ze specyfikatorem public) po klasie Particle. Dodatkowo, ma posiadać zabezpieczone pole int spin, oraz odpowiedni getter i setter. Ma posiadać 3 konstruktory, podobnie do klasy Particle:

```
Boson();
Boson(double mass, double charge, long int pDG, int sp);
Boson(double mass, double charge, long int pdg, int sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu w konstruktorze domyślnym ma być 0.

Fermion

Boson to abstrakcyjna klasa reprezentująca fermiony. Ma dziedziczyć (ze specyfikatorem public) po klasie Particle. Dodatkowo, ma posiadać zabezpieczone pole double spin, oraz odpowiedni getter i setter. Ma posiadać 3 konstruktory, podobnie do klas Particle i Boson:

```
Fermion();
Fermion(double mass, double charge, long int pDG, double sp);
Fermion(double mass, double charge, long int pdg, double sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu w konstruktorze domyślnym ma być 1/2.

Lepton

Klasa Lepton reprezentuje leptony, fermiony nieoddziałujące silnie. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma tylko dwie metody-konstruktory.

```
Lepton();
Lepton(double mass, double charge, long int pdg, double sp);
```

Domyślną wartością spinu ma być 1/2. Wszystkie kolory mają być ustawione na 0.

[2]Nie jest to najbardziej elegancka implementacja, ale najprostsza.

Hadron

Klasa Hadron reprezentuje hadrony, fermiony oddziałujące silnie. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Hadron();
Hadron(double mass, double charge, long int pdg, double sp);
Hadron(double mass, double charge, long int pdg, double sp,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Domyślną wartością spinu ma być 1/2. Domyślnie kolory mogą być ustawione na 0.

Scalar

Klasa Scalar reprezentuje bozony skalarne, czyli o spinie 0. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Scalar()
Scalar(double mass, double charge, long int pdg);
Scalar(double mass, double charge, long int pdg,
 bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Wartość spinu ma zawsze być 0.

Vector

Klasa Vector reprezentuje bozony wektorowe, czyli o spinie 1. Klasa reprezentuje fizyczne cząstki, zatem nie jest abstrakcyjna. Ma trzy metody-konstruktory.

```
Vector();
Vector(double mass, double charge, long int pdg);
Vector(double mass, double charge, long int pdg,
        bool r, bool g, bool b, bool ar, bool ag, bool ab);
```

Wartość spinu ma zawsze być 1.

Przykładowy main

```
#include <iostream> //nie dawa spacji przed >
#include "zadanie2.hpp"//zmien nazwe na odpowiednia

using namespace std;

template<class T>
void print(T* p)
{
        cout << p->getM() << " " << p->getCharge() <<
                " " << p->getPDG() << " " << p->getSpin() <<
                " " << p->getRed() << " " << p->getGreen() <<
                " " << p->getBlue() << " " << p->getAntiRed() <<
                " " << p->getAntiGreen() << " " << p->getAntiBlue()
                << endl;
}
```

```
int main()
{
        //gauge vectors
        Vector gamma;
        gamma.setPDG(22);
        Vector Z(91.1876, 0, 23);
        print(gamma);
        print(Z);
        //leptons
        Lepton e(0.00051, -1, 11, 0.5);
        print(&e);
        Lepton mu(1.776, -1, 13, 0.5);
        print(&mu);
        //quarks
        Hadron u(0.00022, 2./3, 2, 0.5);
        u.setGreen(true);
        print(&u);
        Hadron ab(4.18, 1./3, -5, 0.5);
        ab.setAntiRed(true);
        print(&ab);
        //Higgs
        Scalar h(125, 0, 25);
        print(&h);
        //proton
        Hadron p(0.938, 1, 2212, 0.5);
        print(&p);
        //gravitino!
        Lepton gr(3000, 0, 1000039, 3./2.);
        print(&gr);


        return 0;
}
```

Uwaga: Klasy Particle, Boson i Fermion powinny być abstrakcyjne. Relacje dziedziczenie powinny być takie jak opisano, ze specyfikatorem dostępu public. Nie nadpisywać zdefiiowanych wyżej w hierarchi pól. Klasy najniższego rzędu, a więc Lepton, Hadron, Scalar i Vector nie mają nowych pól a jedynie konstruktory. Zadanie można zrobić w 2/3 plikach, ale można też w jednym.
Podpowiedź: Używaj kopiuj-wklej pisząc nowe klasy. W teoretycznym minimum jest opisane jak zrobić klasę abstrakcyjną.