C++ School 8-12 June, DESY

Our Small C++ Project

A simple MC generator to calculate Z production at Born level

Cross section

The Born level cross section is phase space integral of the matrix elements and the observable and it is convoluted to the parton distribution functions (PDFs):

Phase space

$$\sigma = \int_0^1 d\eta_a \int_0^1 d\eta_b \int d\Gamma(\eta_a, \eta_b; \{p, f\}_m)$$
$$\times f_{a/A}(\eta_a, \mu^2) f_{b/B}(\eta_b, \mu^2)$$
$$\times |M(\{p, f\}_m)|^2 F(\{p, f\}_m)$$

PDFs

Matrix element

Observables

The event is an array of *momenta* and *flavor* of the incoming and outgoing partons.

Need a Lorentz vector

Lorentz vector: Three vector

Lorentz vector has 3 space-like and 1 time-like component. The space-like part is the usual three vector with X, Y, Z component. Thus first we want to define a class that represents three vectors.

```
class threevector
{
protected:
  //  data member
  double _M_x, _M_y, _M_z;

  //  constructors
  threevector(const threevector&) = default;   //  defaulted copy constructor
  //   elements access

  //   aritmethic operators
  //  +=, -=, *=, /=

  double mag2 () const { return _M_x*_M_x + _M_y*_M_y + _M_z*_M_z;}
  double perp2() const { return _M_x*_M_x + _M_y*_M_y;}

  //        magnitude and the transverse component
  double mag () const { return std::sqrt(this -> mag2());}
  double perp() const { return std::sqrt(this -> perp2());}

  //        azimuth and polar angles
  double phi() const { return _M_x == 0.0 && _M_y == 0.0 ? 0.0 : std::atan2(_M_y,_M_x);}

  double theta() const {
    double p = this -> perp();
    return p == 0.0 && _M_z == 0.0 ? 0.0 : std::atan2(p, _M_z);
  }
};
```

- Write the header file `threevector.h`
- We *don't need* `.cc` file since every functions are simple and they can be inline.
- Play with, try the arithmetic operators with simple examples.

Three vector

At the end of the day you should be able to do something like this:

```
#include <iostream>
#include "threevector.h"

using namespace std;


int main()
{
  threevector a(1.0,2.0,3.0), b(5.0,6.0,7.0), c;

  c =a+b;
  cout<<"c = a+b = "<<c<<endl;

  c = a-b;
  cout<<"c = a+b = "<<c<<endl;
  cout<<"a*b = "<<a*b<<c<<endl;
  cout<<"a*2.0 = "<<a*2.0<<c<<endl;
  cout<<"a/2.0 = "<<a/2.0<<c<<endl;

  return 0;
}
```

class threevector (one solution)

```
#ifndef __SCHOOL_THREEVECTOR_H__
#define __SCHOOL_THREEVECTOR_H__ 1

//   Standard includes
#include <cmath>
#include <iostream>

namespace school {

  class threevector
  {
  protected:
    //          data member
    double _M_x, _M_y, _M_z;

    //.....
  };  //class threevector
}  // namespace school
#endif
```

- Class `threvector` with three double variables as data member (x, y, z).
- They are in protected field. Available for the inherited classes but not visible from outside

class threevector (one solution)

```
class threevector
{
protected:
  //         data members
  double _M_x, _M_y, _M_z;

public:
  //  constructors
  threevector(double x = 0.0, double y=0.0, double z=0.0)
    : _M_x(x), _M_y(y), _M_z(z) {}

  //  copy
  threevector(const threevector&) = default;
  threevector& operator=(const threevector&) = default;

  //  destructor
  ~threevector() = default;

  // ...
};
```

- The default constructor creates null vector.
- We have one no trivial constructor.
- Copy operators and destructor can be defaulted, since we have simple data members (no dynamic memory allocation in the class).

class threevector (one solution)

```
class threevector
{
protected:
  //        data member
  double _M_x, _M_y, _M_z;

public:
  //  elements access
  const double& X() const { return _M_x;}
  const double& Y() const { return _M_y;}
  const double& Z() const { return _M_z;}

  double& X() { return _M_x;}
  double& Y() { return _M_y;}
  double& Z() { return _M_z;}

  // ...
};
```

- Since the data members are protected we need functions to get access to the elements.
- Constant operators are READ-OLNY operations.
- Non-constant operators can READ-WRITE.

```
threvector v(1.,2.,3.);

v.X() = 12.0;  // changes v._M_x to 12.0
```

class threevector (one solution)

```
class threevector
{
protected:
  //         data member
  double _M_x, _M_y, _M_z;

public:
  //   computed assignments
  threevector& operator+=(const threevector& a) {
    _M_x += a._M_x; _M_y += a._M_y; _M_z += a._M_z;
    return *this;
  }

  threevector& operator*=(double a) {
    _M_x *= a; _M_y *= a; _M_z *= a;
    return *this;
  }

  // similarly the operators -= and /=
};
```

- The computed assignment operators are member function. The left argument is always the current object (`*this`) that owns the operator.
- They returns a reference of the object itself. It allows something like this:

```
threevector a(1,2,3),b(3,2,1);
threevector c = (a+=b);
```

It is equivalent to

```
threevector a(1,2,3),b(3,2,1);
a+=b;
threevector c = a;
```

class threevector (one solution)

```
inline
threevector operator+(const threevector& a, const threevector& b) {
  return threevector(a) += b;
}

inline
threevector operator*(const threevector& a, double b) {
  return threevector(a) *= b;
}

//    I/O operations
inline
std::ostream& operator<<(std::ostream& os, const threevector& q) {
  return os<<"("<<q.X()<<","<<q.Y()<<","<<q.Z()<<")";
}
```

- Operators outside of the class definition are usually binary operators, like the a+b operator.
- They always return value or reference to one of the argument. Never return reference to local or temporary variable.

class threevector (one solution)

```
inline
threevector operator+(const threevector& a, const threevector& b) {
  return threevector(a) += b;
}
```

This is equivalent to

```
inline
threevector operator+(const threevector& a, const threevector& b)
{
  threevector tmp(a);
  tmp += b;
  return tmp;
}
```

Lorentz vector

Lorentz vector also has time-like component. Define a class inherited from three vector. Define all the arithmetic operators plus some more functions

```
class lorentzvector  //   inherited from threevector
{

  //  member functions
  double plus () const { return _M_t + _M_z;}
  double minus() const { return _M_t - _M_z;}
  double rapidity() const { return 0.5*std::log(plus()/minus());}
  double prapidity() const { return -std::log(std::tan(0.5*theta()));}
  double mag2() const { return _M_t*_M_t - threevector::mag2();}

  threevector boostVector() const {
    return threevector(*this) /= _M_t;
  }

  //  Lorentz boost
  void boost(double, double, double);
  void boost(const threevector& a) { boost(a.X(), a.Y(), a.Z());}
};
```

- Write the header file `lorentzvector.h`
- The `boost(…)` function is implemented in the `lorentzvector.cc` file.
- Play with, try the arithmetic operators with simple examples.

Event record

```
#ifndef __SCHOOL_EVENT_H__
#define __SCHOOL_EVENT_H__ 1

#include "lorentzvector.h"

//   std includes
#include <vector>

namespace school {

  //   flavors
  enum flavor_type {nuebar = -12, positron,
    topbar=-6, bottombar, charmbar, strangebar, upbar, downbar,
    gluon, up, down, strange, charm, bottom, top,
    electron = 11, nue
  };

  //   structure for representing incoming and outgoing particles
  struct particle {
    //  flavor of the particle
    int flavor;

    //   momentum of the particle
    lorentzvector momentum;
  };


  class event
  {
  public:
    //   ….
  };
}   //  namespace school
#endif
```

- Protect your header file to avoid including it more than one.
- We have to label the flavors, use enum.
- The particle can be represented by its momenta and flavor.
- The event record is an array of particles.
- Indexing:
  -1, 0 => incomings
  1,2,…,n => outgoings

Event record

```
class event
{
public:
  double xa;
  double xb;

private:
  std::vector<particle> _M_array;

public:
  //  constructor
  //(we have always 2 incomming + n outgoing)
  explicit event(unsigned int n=1u);

  //  copy
  event(const event&) = default;
  event& operator=(const event&) = default;

  //   dectructor
  ~event() = default;
};
```

- Momentum fraction of the incoming partons.
- Array of particles
- Constructors and destructor.
- Indexing:
  -1, 0 => incomings
  1,2,…,n => outgoings

Event record

```
class event
{
public:
  double xa;
  double xb;

private:
  std::vector<particle> _M_array;

public:
  //   element access
  particle& operator[](int k);

  const particle& operator[](int k) const;
};
```

- Element access by subscript operators.
- Constant and non-constant access.
- Indexing:
  -1, 0 => incomings
  1,2,…,n => outgoings

Event record

```
 class event
 {
public:
   //   iterators
   typedef std::vector<particle> _Base;
   typedef _Base::iterator iterator;
   typedef _Base::const_iterator const_iterator;

   iterator begin();
   const_iterator begin() const;

   iterator end();
   const_iterator end() const;

   //  resize
   void resize(unsigned int n);

   //   structural information
   unsigned int number_of_outgoings() const;
 };
```

- Element access by iterators
- Number of the outgoing particles.

Good luck!!!