5-remaining-issues

June 3, 2024

1 Remaining issues

1.1 Only a subset of mathematic functions

Let's take this example : we want to compute the total energy `e` of an electron whose linear momentum `p` is known. We can deduce it from the equality `e^2 = m^2*c^4 + p^2*c^2`, with `c` the speed of light and `m` the mass of the electron.

[63]:
```
%%file tmp.remaining-issues.cpp

#include <iostream>
#include "phys/units/io.hpp"
#include "phys/units/quantity.hpp"

using namespace phys::units ;
using namespace phys::units::io ;
using namespace phys::units::literals ;

int main()
 {
  constexpr auto m = 9.109e-31_kg ;
  constexpr auto c = 299792458_m/second ;
  constexpr auto p = m*(0.75*c) ;

  auto e = sqrt(square(m)*nth_power<4>(c)+square(p)*square(c)) ;

  std::cout << e << std::endl ;
 }
```

```
Overwriting tmp.remaining-issues.cpp
```

[64]:
```
!rm -f tmp.remaining-issues.exe && g++ -I. -std=c++17 tmp.remaining-issues.cpp␣
 ↪-o tmp.remaining-issues.exe
```

[65]:
```
!./tmp.remaining-issues.exe
```

```
1.02335e-13 J
```

We were not able to use `std::pow`, which would have not correctly handle the unit of `c`.

We were not able to benefit from the `constexpr` of `e`, because the `sqrt` function used is not provided by the standard library, but is an alternative implementation from PhysUnits, which can handle units, but is not a `constexpr`.

The use of a library of units tends to **limit us to the numerical functions provided by this library**, with their limitations.

1.2 Irrelevant internal reference unit

Let's try our example with `float`.

[66]:
```
%%file tmp.remaining-issues.cpp

#include <iostream>

#include "phys/units/io.hpp"
#include "phys/units/quantity.hpp"

using namespace phys::units ;
using namespace phys::units::io;
using namespace phys::units::literals ;
using momentum_d = dimensions< 1, 1, -1 > ;

int main()
 {
  constexpr quantity<mass_d,float> m = 9.109e-31_kg ;
  constexpr quantity<speed_d,float> c = 299792458_m/second ;
  constexpr quantity<momentum_d,float> p = m*(0.75*c) ;

  auto e = sqrt(square(m)*nth_power<4>(c)+square(p)*square(c)) ;

  std::cout << e << std::endl ;
 }
```

```
Overwriting tmp.remaining-issues.cpp
```

[67]:
```
!rm -f tmp.remaining-issues.exe && g++ -I. -std=c++17 tmp.remaining-issues.cpp␣
 ↪-o tmp.remaining-issues.exe
```

[68]:
```
!./tmp.remaining-issues.exe
```

```
6.14677e-14 J
```

The result if very wrong. However, we can prove that this computation can be done with simple precision, provided we express our variables in a unit which is close to the scale of the problem, which is not the case of `kg` above, used internally by PhysUnits.

Below, the mass and linear momentum have been shift by `1e27`, computation is enforced in `float`, and the result is ok.

[75]:
```
%%file tmp.remaining-issues.cpp

#include <iostream>
#include <cmath>

int main()
 {
  constexpr float m = 0.0009109 ;
  constexpr float c = 299792458 ;
  constexpr float p = m*(0.75f*c) ;

  constexpr float m2 = m*m ;
  constexpr float c2 = c*c ;
  constexpr float c4 = c2*c2 ;
  constexpr float p2 = p*p ;
  constexpr float e = sqrt(m2*c4+p2*c2) ;

  std::cout << e*1e-27 << std::endl ;
 }
```

```
Overwriting tmp.remaining-issues.cpp
```

[76]:
```
!rm -f tmp.remaining-issues.exe && g++ -I. -std=c++17 tmp.remaining-issues.cpp␣
 ↪-o tmp.remaining-issues.exe
```

[77]:
```
!./tmp.remaining-issues.exe
```

```
1.02335e-13
```

In the domain of infinitely small and infinitely large, it is aberrant to store values in seconds, meters, kilograms, etc. **The units library must let you control the unit used for internal storage**, or you may suffer a huge loss of precision. About this topic, Martin Moene (the author of PhysUnits) indicates the work of Tony Pilz (which I did not explore yet) : * ScaledValue : https://github.com/tonypilz/ScaledValue * Units : https://github.com/tonypilz/units

1.3 Libraries of linear algebra

In scientific computing, we often use a lot of matrice computation, and deeply rely on libraries like Eigen… Can we make Eigen uses units provided by PhysUnits ? Let's try to compute the linear momentum of our electron, but in 3D space, with x/y/z coordinates.

[23]:
```
%%file tmp.remaining-issues.h

#include <iostream>

#include "phys/units/io.hpp"
#include "phys/units/quantity.hpp"
#include "Eigen/Dense"
```

```
using namespace phys::units ;
using namespace phys::units::io;
using namespace phys::units::literals ;

using momentum_d = dimensions< 1, 1, -1 > ;

using Speed = quantity<speed_d> ;
using Momentum = quantity<momentum_d> ;

using Speed3d = Eigen::Matrix<Speed,3,1> ;
using Momentum3d = Eigen::Matrix<Momentum,3,1> ;
```

```
Overwriting tmp.remaining-issues.h
```

[132]:
```
%%file tmp.remaining-issues.cpp

#include "tmp.remaining-issues.h"

int main() {

  constexpr quantity<mass_d> m = 0.0009109_yg ;
  constexpr quantity<speed_d> c = 299792458_m/second ;

  Speed3d speed3d(0.75*c,0.*c,0.*c) ;

  Momentum3d momentum = m*speed3d ;
  Momentum p = momentum.norm()*kilogram*meter/second ;
  auto e = sqrt(square(m)*nth_power<4>(c)+square(p)*square(c)) ;
  std::cout << speed << std::endl ;

}
```

```
Overwriting tmp.remaining-issues.cpp
```

[27]:
```
!rm -f tmp.remaining-issues.exe && g++ -I/usr/local/include/eigen3/ -I.␣
 ↪-std=c++17 tmp.remaining-issues.cpp -o tmp.remaining-issues.exe >␣
 ↪compiler_log.txt 2>&1
```

Actually, Eigen does not accept to multiply a scalar with a vector if all the numbers are not expressed in the same types. Actually, we should work the other way round : wrap the eigen objects within our strong types:

[88]:
```
%%file tmp.remaining-issues.h

#include <iostream>

#include "phys/units/io.hpp"
#include "phys/units/quantity.hpp"
#include "Eigen/Dense"
```

```
using namespace phys::units ;
using namespace phys::units::io;
using namespace phys::units::literals ;

using momentum_d = dimensions< 1, 1, -1 > ;

using Eigen31 = Eigen::Matrix<double,3,1> ;

using Speed3d = quantity<speed_d,Eigen31> ;
using Momentum3d = quantity<momentum_d,Eigen31> ;
```

```
Overwriting tmp.remaining-issues.h
```

[133]:
```
%%file tmp.remaining-issues.cpp

#include "tmp.remaining-issues.h"

int main() {

  constexpr quantity<mass_d> m = 0.0009109_yg ;
  constexpr quantity<speed_d> c = 299792458_m/second ;

  Speed3d speed(detail::magnitude_tag,Eigen31{0.75*c.magnitude(),0.,0.}) ;
  Momentum3d momentum = m*speed ;

  quantity<momentum_d> p = momentum.magnitude().norm()*kilogram*meter/second ;
  auto e = sqrt(square(m)*nth_power<4>(c)+square(p)*square(c)) ;
  std::cout << e << std::endl ;

}
```

```
Overwriting tmp.remaining-issues.cpp
```

[90]:
```
!rm -f tmp.remaining-issues.exe && g++ -I/usr/local/include/eigen3/ -I.␣
 ↪-std=c++17 tmp.remaining-issues.cpp -o tmp.remaining-issues.exe
```

[91]:
```
!./tmp.remaining-issues.exe
```

```
1.02335e-13 J
```

The author of phys/units has not anticipated such a use of non-scalar value type: - we must use a special constructor with `detail::magnitude_tag`, - all eigen methods such as `norm()` are hidden wothin the wrapper object.

1.4 Also...

I/O : When handling very large data sets, how to add the lacking units afterwards ? How to store the data with their units efficiently ?

Compilation time: provided there exists a units-friendly linear algebra library, what will be the compilation time, if every `Vector/Matrix` must be instanciated with all the possible physical units used in the program... And let's speak about the compilation error messages...

2 Take away

Until there exists some linear algebra library which is compatible with strong types, do not throw the baby out with the bathwater: * use strong types for integers: index, sizes, identifiers, ... ; * rely on user-defined literals for the handling of physical units multipliers ; * use physical units in code areas which do not require some incompatible external library ; * select reference storage units at the right scale for your problem.

3 Questions ?

4 Exercice

We reuse below our home made `SiUnit`, which has been reviewed so to support our energy computation, in the scalar flavor. Because C++ do not accept "user-defined literals" for something else than scalar numbers, and we want to later handle vectors and matrices, we drop such literal sand fall back on the good old constants approach. Compile and run.

Then, try to make our speed vectorial by using `Eigen::Matrix<double,3,1>` as internal type. What should be changed? Suggestions: 1. Make types Speed3d and Momentum3d which are wrapping `Eigen::Matrix<double,3,1>` instead of `double`. 2. Make `speed` and `p` instances of `Speed3d` and `Momentum3d`. 3. Initialize `speed` with `Eigen::Matrix<double,3,1> { 0.75*static_cast<double>(c), 0., 0. }`. 4. Overload `operator*( UT lhs, SiUnit<<Eigen::Matrix<UT,3,1>,...> )` 5. Overload `operator*( SiUnit<UT,...>, SiUnit<<Eigen::Matrix<UT,3,1>,...> )` 6. Overload `norm( SiUnit<Eigen::Matrix<UT,3,1>,s,m,kg> )`. 7. In the computation of the final energy, take the norm of `p`.

[22]:
```
%%file tmp.emc2.cpp

#include <iostream>
#include <cmath>

// main class, which supports any mix of duration,
// length and mass.

template< typename UnderlyingType, int s, int m, int kg >
class SiUnit {
  public :
    explicit SiUnit( UnderlyingType value ) : my_value{value} {}
    explicit operator UnderlyingType() { return my_value ; }
  private :
    UnderlyingType my_value ;
} ;
```

```
// operators

template< typename UT, int s, int m, int kg >
std::ostream & operator<<( std::ostream & os, SiUnit<UT,s,m,kg> obj )
 { return (os<<static_cast<UT>(obj)) ; }

template< typename UT, int s, int m, int kg >
auto operator+( SiUnit<UT,s,m,kg> lhs, SiUnit<UT,s,m,kg> rhs )
 { return SiUnit<UT,s,m,kg>(static_cast<UT>(lhs)+static_cast<UT>(rhs)) ; }

template< typename UT, int s1, int m1, int kg1, int s2, int m2, int kg2 >
auto operator*( SiUnit<UT,s1,m1,kg1> lhs, SiUnit<UT,s2,m2,kg2> rhs )
 { return␣
 ↪SiUnit<UT,s1+s2,m1+m2,kg1+kg2>(static_cast<UT>(lhs)*static_cast<UT>(rhs)) ; }

template< typename UT, int s, int m, int kg >
auto operator*( UT lhs, SiUnit<UT,s,m,kg> rhs )
 { return SiUnit<UT,s,m,kg>(lhs*static_cast<UT>(rhs)) ; }

template< typename UT, int s1, int m1, int kg1, int s2, int m2, int kg2 >
auto operator/( SiUnit<UT,s1,m1,kg1> lhs, SiUnit<UT,s2,m2,kg2> rhs )
 { return SiUnit<UT,s1-s2,m1-m2,kg1-kg2>(static_cast<UT>(lhs)/
 ↪static_cast<UT>(rhs)) ; }

// base units and constants

using Time = SiUnit<double,1,0,0> ;
Time S { 1. } ;

using Length = SiUnit<double,0,1,0> ;
Length M { 1. } ;

using Mass = SiUnit<double,0,0,1> ;
Mass KG { 1. } ;

// combined units

using Speed = SiUnit<double,-1,1,0> ;
using Momentum = SiUnit<double,-1,1,1> ;
using Energy = SiUnit<double,-2,2,1> ;

// math functions

template< int pn, int pd, typename UT, int s, int m, int kg >
auto power( SiUnit<UT,s,m,kg> value )
```

```
 { return SiUnit<UT,s*pn/pd,m*pn/pd,kg*pn/
 ↪pd>(pow(static_cast<UT>(value),static_cast<UT>(pn)/pd)) ; }

template< typename UT, int s, int m, int kg >
auto square( SiUnit<UT,s,m,kg> value )
 { return power<2,1>(value) ; }

template< typename UT, int s, int m, int kg >
auto sqrt( SiUnit<UT,s,m,kg> value )
 { return power<1,2>(value) ; }

// main

int main() {

  Mass m { 9.109e-31*KG } ;
  Speed c { 299792458.*M/S } ;
  Speed s { 0.75*c } ;
  Momentum p { m*s } ;

  Energy e { sqrt(square(m)*power<4,1>(c)+square(p)*square(c)) } ;
  std::cout << e << std::endl ;

 }
```

```
Overwriting tmp.emc2.cpp
```

[26]:
```
!rm -f tmp.emc2.exe && g++ -I/usr/local/include/eigen3/ -std=c++17 tmp.emc2.cpp␣
 ↪-o tmp.emc2.exe
```

[27]:
```
!./tmp.emc2.exe
```

```
1.02335e-13
```