SearchAThing.Sci

A library to deal with scientific data and basic geometric processing in C#

# Measure Units

## Introduction

Physical quantity can be measured by through comparison with other homogeneous quantities.

For each physical quantity can exists one or more quantity assumed as unit in order to establish a measurement that will results in a number (scalar) associated with that unit. When that unit can be uniquely identified in the time and space we can say that a number multiplied the measure unit represents an equivalent physical quantity of the one we measured.

If exists more than one measure unit for a physical quantity we can establish a conversion factor between these measure unit if they share the zero reference. For example each length measure unit of a null length results 0 while different temperature measure unit not share the zero reference ( ie. 0C not requals 0F ).

## Arranging a conversion table

|  |  |  |  |
| --- | --- | --- | --- |
|  | **mm** | **cm** | **m** |
| **mm** | 1 | 0.1 | 0.01 |
| **cm** | 10 | 1 | 0.1 |
| **m** | 100 | 10 | 1 |

To ensure high speed computation we need a pre-compiled measure unit conversion table where scalar conversion is allowed simply using a factor. For example assuming that m = 100 \* mm :

when we write (7 \* m) we can express that physical quantity in an equivalent but with different measure unit as (7 \* (100 \* mm)) = 700 mm

In the above conversion table we have N\_mu=3 (mm, cm, m) and a table with 3\*3=9 factors, but only some of that numbers are “independent” while other are “dependent” and other “obvious” ( those 1 on the diagonal ).

In general we’ll need only the first column factors values ( except the obvious 1 ) so in general for N measure unit we need (N-1) conversion factor to fill a NxN mu conversion table as explain the follow diagram:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **A** | aa=1 | 1/ba | 1/ca |
| **B** | ba | bb=1 | 1/cb |
| **C** | ca | cb=ca/ba | cc=1 |

Explanation:

* Independent conversion factors
  + ba = factor to convert B mu to A mu
  + ca = factor to convert C mu to A mu
  + In other words for 3 mus ( A, B, C ) said A the common reference measure unit we need N-1={ ba, ca } conversion factor that converts between mus combinations
* Dependent conversion factors
  + [yellow] : cb is the factor that convert C measure unit to B but this is the same as if we convert C to A and then A to B and considering that the conversion A to B is the inverse of the conversion from B to A we have all of the known independent factor to complete the conversion.
  + [green] : the upper triangle of the table can be obtained inverting values of the lower triangle, so that for example the ab factor = 1/ba

## Measure comparision

Measuring the same physical quantity sample with an instrument can subjected to error that came from who take the measure and from the limit of decimals we can gather from the sampled measure itself deriving from the instrument limit. In other words we can take measure of “exactly the same” object two times and obtain two different measures. But the object is the same!

In order to establish correctness of comparisons operators in our Net library we need to consider a tolerance that let us to consider equals two different numerical values when this difference not exceed the tolerance associated with the measure.

### Error propagation

When a measure born from an instrumentation measurement we can say that the physical quantity measure is the number obtained with associated measure unit and with the instrument tolerance. The same object measured with another instrument produces another measure with a different tolerance. All this to clarify that the tolerance can’t be the same by a measure unit itself. Another thing to note is that the measure unit itself not contains additional errors: for example the length 10mm can contains error for the measure instrument for example +/- 0.1mm but the “mm” itself not add other errors in the measure cause that sample exists in the real and “should” be immutable in the space and in the time and we no need to measure the measure unit sample, we just refer to it as a real reference thing.

But what happens when we do some math with our measurements ? Answer, error propagates.

Consider as an example the measurement of the length of a stick:

* 1 single measurement with an instrument that can sample the entire object
* 2 measurement with an instrument that can sample 2/3 of the object. In this last case we need to measure the object two times and then we can say that the total length is the sum of measured lengths but the error we gave in the first measure not replace the second resulting errors add up.

Note: In the implementation of this library we’ll not consider error propagation in order to maintain faster execution. This is a limitation in absolute terms but we’ll consider a global tolerance for each reference measure unit. For example in a certain application domain we can say that our length reference measure [mm] has a tolerance of 0.1 [mm] or that our force reference measure [N] has a tolerance of 1e-3 [N]. In short more operations will be done with these measurements without considering the correct error propagation but a fixed one and more underestimated value of the real propagated error results. Depending on the type of application that can be irrelevant or fatal.

We’ll specify the tolerance only for one measure unit for each different physical quantity while the tolerance of other N-1 measure unit for each one will be computed as scalar through the conversion factor of the measure unit itself. For example if we states that our [mm] reference measure unit has a tolerance of 0.1 [mm] the consequent tolerance for the [m] will be 0.1 \* (1e-3 [m]) = 1e-4 [m].

## Measure unit domain

In order to maintain efficiency and speed in code execution this library implementation express measurements of a physical quantity without explicitly associate a measure unit. This behavior I refer as measure unit domain. In other words when a measure enter the domain it will be converted from its measure unit to a reference measure unit ( domain reference measure unit ) and for all subsequent operations we not need to decorate that number with a measure unit because that are already defined from the domain. Measure unit domain simply is a selection of one reference measure unit for each available physical quantity.

As an example follow function compute distance between two 3d points:

double Dst(Vector3D a, Vector3D b) { return Sqrt(Pow(b.x-a.x,2)+Pow(b.y-a.y,2)+Pow(b.z-a.z,2)); }

we’ll think to a.{x,y,z} and b.{x,y,z} and the result of Dst() function as measurements with homogeneous measure unit equals to the domain reference measure unit ( for example [mm] ).

When we want to keep track of an associated measure unit, for example coming from a GUI where the user inserted a specific measure unit, we’ll keep track of that by encapsulating this additional information in a specific value+mu object. But when we pass these measure to the domain-inside functions we’ll convert these to the domain reference measure unit resulting in a number (double) with implicit domain measure unit associated.

In order to allow the developer to customize its own preferred Measure Unit Domain we’ll need to keep track of the list of domain reference measure unit used in every serialized data file.

# Linear algebra

Linear algebra methods can extensively used to solve many geometric problems like for example find the intersection between lines and/or planes; In this implementation we’ll not use a generic approach when solving linear system of equations because that will results in a slower computation of 2d, 3d system of equations. In fact a general approach to solve n-dimensional linear equation systems imply to factorize the correspondent matrix (LU decomposition) where for simple 2d, 3d processing direct formula approach is preferred.

# Types

## 2D and 3D specifics

Working with a set of 3D points which measure all the same distance from a given plane appears like we are working with 2D points ( for example p1=(10,12,5) and p2=(30,33,5) and p3=(3,4,5) are three 3d points all within the plane xy at height z=5 ). From this point of view we can implement 3D points processing algorithm and use them to process our points as 2D simply preconditioning these points with the same Z ( for example z=0 ).

But there are some important and useful functions to implement for the 2D (cfr example <https://en.wikipedia.org/wiki/Point_in_polygon> ) where use of classes that handles third coordinate may results in unnecessary storage and incompleteness of prerequisites ( all points on the same plane ) for those an assert checking slow down the processing itself. For these considerations we’ll implement specific types and algorithm for 2d and separately for 3d.

## Point, vector, segment, line, planes, coordinate systems

The type Vector will be used and interpreted as:

* **point** ( the vector components are the coordinate of the point )
* **segment** or **line** is a type that contains two vector p1, p2 that are points intersected by the line itself. We can construct this type in at least two manner
  + Line(double x1, double y1, double z1, double x2, double y2, double z2) or

Line(Vector v1, Vector v2, LineInitMode = PointAndVector )

In the latest type of constructor we can construct a segment either by default init mode passing a point as v1 and a direction using v2. Example

var l = new Line(new Vector(1,2,3), new Vector(1,0,0))

will create an 3d line that intersect the (1,2,3) point and that is parallel to the given direction vector (1,0,0)

The second type of initialization can be done by specifying two points that the constructed line will intersects and explicitly the type of initialization. For example:

var l = new Line(new Vector(1,2,3), new Vector(2,2,3), VectorInitMode.ByTwoPoints)

This line equals the previous but its constructed using two points.

* from this point of view the Line type does not contains information about its extension. In other words the Line type not explain if it’s a Segment or an infinite Line. All of this to avoid additional storage requirements and to allow easy polymorphism of the line type. The meaning of extension for a line will be treated by the processing functions. For example the Intersect as default will treat lines as infinite elements:

var p = l1.Intersect(l2);

While another method will check if the intersection point is contained in either Line type treated as segments:

var p = l1.SegmentIntersect(l2);

* **Plane** type is a triple of points (or vector) where we store in one the Origin, and in the other two the XAxis and YAxis directions respectively.
* A 3d **coordinate system** can be constructed in a way similar to the one used to build a plane with some considerations:
  + to construct a 3d coordinate system we need at least one origin point O, and two non parallel direction vectors
  + in the initialization process of the coordinate system the given vx, vy plane vector will be ortho-normalized in a way such that vx will be perpendicular to vy modifying eventually vy direction and a third base vector vz will be computed using the “right-hand” rule ( vz = vx x vy )
* From this point of view the Plane3D will be merely a 3d coordinate system in order to increase the type polymorphism.