PhysCalc: a Scilab package for physical calculation

Denis J.-Y. MARION (denis.marion@cea.fr)

November 19, 2009

Introduction

Any scientist has already experienced errors in their results or their predictions of physical parameters. These errors are very often due to a missing conversion between two initial data or formulas expressed in different systems of units. Yet the PhysCalc package doesn't ensure that your calculations will be correct, it still can be of great help when dealing with values and constants gathered from different bibliographical sources, hence possibly expressed in varied systems of units.

PhysCalc facilitates the declaration and the correct use of physical quantities in Scilab. Once all your constants and parameters are declared (with a very light syntax), any operation and Scilab function can be used to work on them. While you're concentrated on your work and the physical meaning of your results, PhysCalc:

- checks for you that your operations have a sense (e.g. that you are not adding or concatenating speeds and energies...)
- makes implicitly the appropriate conversions when adding two quantities of identical physical meaning (e.g., a speed in International Units and a speed in CGS units).
- calculates the unit of the result of every operation (addition, division, multiplication, exponentiation,...) based on physical operands.
- displays your results with their unit in a human-readable format. For instance:

```
--> E  = .5 * my_mass * my_speed^2
 E =
                       2  -2
    8.983733e+01  kg.m .s     (#International)
```

Installation and use of PhysCalc

For a comprehensive example, please refer yourself to the `Units.sci` file where all the following rules are synthesised in a working manner.

Install the package

PhysCalc consists in only one mandatory .sci file called `PhysCalc.sci`. To include it in your scope, just `chdir()` in the correct directory and then type

```
--> exec('PhysCalc.sci');
```

This file declares a new type called `"UNIT"`, a mlist representing a vector (or a n-dimension array) of physical quantities. It also defines the operators concerning this new type.

The two other files included in the .gz package are only examples showing how to declare your own systems of units and your units (file `Units.sci`) and your physical constants and parameters (file `Constants.sci`).

Declare different systems of units

In preamble to any work with PhysCalc, you must first initialize the global variable called `SystemList`, as shown in the first line of `Units.sci`. This variable will be used by PhysCalc to store the different unit systems you will declare.

```
--> global SystemList ; SystemList = list();
```

The first system created will be used as the "reference" system and will then have a privileged role. To declare a system of units, use the `defUnitSystem` function with the following syntax:

```
defUnitSystem(SystemName, ConversionFactors, UnitNames);
```

where:

- `SystemName` is a character string describing the new system, for instance `#International`, `#ISU`, `#CGS`, `#BTU`... The '`#`' character is not mandatory but helps differenciate the unit systems, the units and the physical constants themselves...
- `ConversionFactors` is a vector of real numbers. These represent the factors of conversion between the units of your new system and the "reference" system. Hence, when you declare the first system, `ConversionFactors` must be a vector of k ones, where k is the number of units you will be using in your work (length, mass, time, current, temperature,... up to seven in a full system of units).
- `UnitNames` is a string vector of length k. Each string represents the name of the unit which will be printed (exponentiated to the right power) when displaying a quantity expressed in the newly created sytem of units.

Here's an example of the call you may want to make if you desire to declare the International System of Units as the reference system:

```
defUnitSystem("\#International", [1, 1, 1, 1, 1, 1, 1], ...
                ['m', 'kg', 's', 'A', 'K', 'mol', 'cd']) );
```

and then the CGS system as an alternative system:

```
defUnitSystem("\#CGS", [1e-2, 1e-3, 1, 1, 1, 1, 1], ...
                ['cm', 'g', 's', 'A', 'K', 'mol', 'cd']) );
```

The second declaration stipulates that PhysCalc will apply, for instance, a conversion factor of 0.01 to convert a length expressed in CGS units in IS units.

NB: If the first system you declared owned k units, all subsequent systems must also own k units.

Declare units and constants

Units and quantities

Once you have declared all the systems of units you will need for your work, you can define units which will be "shortcuts" for the further declarations. Units are in fact defined as physical quantities with a numerical value equal to 1 in the reference system. To declare a unit (or any physical quantity), use the `defQuantity` function:

```
_NewUnit = defQuantity(SystemName, Value, UnitsExponentiation) );
```

where

- `_NewUnit` is the name of the new unit you want to declare. The underscore is not mandatory but, as already said, helps to differentiate units from other variables in the Scilab scope.
- `SystemName` is a string : the System of Units in which the unit is described.
- `Value` is a real number: the numerical value of the unit in the `SystemName` system.
- `UnitsExponentiation` is the power at which every unit is exponentiated to form the desired unit. For instance, an energy is a mass multiplied by the square of a velocity, thus the following definition of the joule "`_J`" (which you can find in `Units.sci`):

  ```
  _J = defQuantity("#International", [2, 1, -2, 0, 0, 0, 0] );
  ```

If you observe thoroughly the `Units.sci` file, you will see that the most common units are already defined there, all with their standard name[1], but you can of course define any unit you want. Good examples would be to define thermodynamical units, or units more closely related to your work (the electron-volt or the GeV for instance).

[1] as quoted for the American NIST, and with the exception of the *ohm* which is, of course, not named "_Ω" but `_Ohm`.

Constants and physical parameters for your work

This preliminary job of defining unit systems and units is meant to be done only once, in preliminary scripts like `Units.sci`. You should define the real quantities on which you work in a separate file, as it is done in `Constants.sci`. This file includes some very common physical constants[2]. Let us examine a line of this file, for instance the definition of the Boltzmann constant "`kB`":

```
kB = 1.3806504e-23 &  (_J * _K^(-1))  & #International ;
```

We see that the declaration of a parameter begins with its numerical value, followed by an ampersand character "`&`", followed then by a formation of units, once again followed by an ampersand and finally ending with the name of the unit system the numerical value is given in. When preceded by a real (scalar, vector, matrix, ...) and followed by a unit, the ampersand character "`&`" has the following meaning:

```
NewQuantity = x & U
```

returns the `UNIT`-type mlist `NewQuantity` formed of the same unit exponentiation as `U`, declared in the same System of units, and with a numerical value being equal to `x`. When preceded by a `UNIT`-type mlist (scalar, vector, matrix, ...) and followed by a unit System, the ampersand character "`&`" has the following meaning:

```
NewQuantity = U & US
```

returns the `UNIT`-type mlist `NewQuantity` where the numerical value it contains are "force-cast" as the ones in the `US` system of units[3].

Hence, the declaration of the Boltzmann constant could be read literally as:

> "k_B is a new quantity with the same units exponentiation as the unit $(J \cdot K^{-1})$ and with a numeric value of $1.3806504 \cdot 10^{-23}$ when expressed in the International system of units."

It is then exactly equivalent as to say that $k_B = 1.3806504 \cdot 10^{-23} J \cdot K^{-1}$.

Everyday use of PhysCalc

The PhysCalc package is meant to be as transparent as possible to its user. The best experience would be to almost never notice that it is here, except when one needs to check the unit of a quantity, to avoid inconsistent additions, or to convert from one system of units to one another. Once the data are correctly defined, they should behave exactly like regular `CONST` vectors, that's why, instead of writing a long list of function definitions, we will see the usage of the PhysCalc through a very short and comprehensive examples.

[2] all of them are also quoted from the American NIST.

[3] this syntax is somewhat redundant to the previous definition of "`&`" and is kept only to ensure that the user sees clearly in which system he is defining the numerical value of his constants.

Add a physical sense to your variables

Let's say you imported experimental data in a variable `I` which happens to be a milliampere current induced in a coil. For instance:

```
I =
    84.974524
    68.573102
    87.821648
    6.8374037
    56.084861
```

Just declare a new variable `I1` defined as a milliampere current:

```
I1 = I & (1e-3 * _A);
```

Convert, multiply, exponentiate, display quantities

If you know the inductance of the coil (86.7 mH) `L1 = 86.7 & (1e-3 * _H)`, you can easily calculate the electro-magnetic energy stored in your coil:

```
-->Energy = 1/2 * L1 * I1^2
 Energy =

     0.0003130
     0.0002038
     0.0003343
     0.0000020
     0.0001364

        2     -2
 Unit: m .kg.s     (#International)
```

You don't need to remember any more in which systems and submultiples of units your data are written before you make operations on them!

Notice that the unit of the result is always written at the end of the numerical values contained in the `Energy` vector when displayed in the Scilab prompt. Also, if, in an expression, you mix data declared in a system of units A and other data written in a system B, the result will always be declared in the system of the leftmost member of the expression.

If you want your result converted into another system of units, try the "\" operator:

```
-->Energy \ #cgs
 ans =

     3130.1603
     2038.4342
     3343.4303
```

```
    20.266164
    1363.5793

          2    -2
Unit:    cm .g.s     (#cgs)
```

The multiplication and division are considered piecewise (like regular vectors) if you used the `.*` and `./` operators instead of `*` and `/`.

Add, concatenate, extract, insert quantities

Just above, you see that `Energy` is not a scalar but resembles more a classical Scilab vector. It is possible with PhysCalc to declare scalars, vectors, matrices and even $n-$dimensions arrays of such physical quantities. You can add two vectors of identical size, proven that they have the same physical dimension[4]:

```
-->Energy + (Energy \ #cgs)
 ans =

    0.0006260
    0.0004077
    0.0006687
    0.0000041
    0.0002727

          2     -2
Unit:    m .kg.s     (#USI)

-->Energy + I1;
 !--error 10001
Unmatched dimensions.
at line       4 of function %UNIT_a_UNIT called by :
Energy + I1
```

You can as well concatenate and insert subsets of physical quantities, if they have the same physical dimension and if they obey the same row/column rules as for Scilab concatenation/insertion in a classical array.

```
-->[Energy, 2 * Energy]
 ans  =

    0.0003130    0.0006260
    0.0002038    0.0004077
    0.0003343    0.0006687
    0.0000020    0.0000041
    0.0001364    0.0002727
```

[4]"physical dimension" means here a given exponentiation of the units.

```
          2     -2
Unit:    m .kg.s     (#USI)

--> Energy + I1
 !--error 10001
Unmatched dimensions.
at line       4 of function %UNIT_a_UNIT called by :
Energy + I1

-->Energy(3) = Energy(2)
 E  =

    0.0003130
    0.0002038
    0.0002038
    0.0000020
    0.0001364

          2     -2
Unit:    m .kg.s     (#USI)

--> Energy(3) = Energy(1:3);
Incorrect assignment.

at line       9 of function %UNIT_i_UNIT called by :
Energy(3) = Energy(1:3)

-->Energy(3) = I1(3);
 !--error 10001
Unmatched dimensions.
at line       6 of function %UNIT_i_UNIT called by :
Energy(3) = I1(3)
```

This precaution for "homogeneous" addition and concatenation can be considered too cautious, but it is there to insure that one does not mix quantities which have nothing to do with each other, a particularity which takes all its sense in the physical calculus.

Use every other Scilab function

Most obviously, every Scilab function cannot be redefined for this new type of variable. But you can still use every Scilab function by taking the raw numerical data contained in your physical quantities by calling first the function `numeric()`:

```
data = numeric(PhysQuantity)
```

returns the numerical data of the PhysCalc object `PhysQuantity` in the same-size array `data`. Hence, you can for instance plot the energy stored in the coil very rapidly:

```
-->time = (0:0.1:0.4) & _s;              // define time in seconds
-->plot(numeric(time), numeric(Energy)); // plot Energy vs. time
```

Please notice that the philosophy of PhysCalc is to maintain the physical sense of the data being worked on as long as possible, so one should use `numeric()` only when working on functions. A very good way to ensure (at least in the body of a function or a script) that one works on non-dimensional quantities (quantities "equivalent" to the number 1) is to try/catch the comparison to the "unit" called "`_1`" (declared in the `Units.sci` file). For instance:

```
function yesNo = isNonDimensional(MyPhysQuantity)
try
  MyPhysQuantity(1) = _1;
  yesNo = %T;
  return;
catch
[str, n] = lasterror();
  if (n == 10001)
    yesNo = %F;
    return;
  else
    error(str, n);
  end;
end;
endfunction;

--> isNonDimensional(Energy)
 ans  =

  F

--> isNonDimensional(_1)
 ans  =

  T
```

Remarks, future improvements

- PhysCalc is still in its first stage of release, please be patient and contact its author if you encounter some difficulties and/or unexpected results. The author should obviously not be considered responsible for false results inducing damages or wrongs of any kind!

- PhysCalc includes a very simple definition of elementary fractions (and vectors of fractions), look at the end of the source file if you want to know more or try typing `fraction(229/27) + fraction (144/51)` for a rapid insight.
- There is no counter-indication against the definition and the use of complex (as opposed to real) data. This could be of some use for harmonic and electro-magnetic calculations.
- Please keep in mind that operations on physical quantities involve many more individual operations than their equivalents on classical arrays. To summarize, PhysCalc is not intended for high-throughput calculations.