SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

SP&E

Automated computation and consistency checking of physical dimensions and units in scientific programs

Grant W. Petty*,†

Atmospheric and Oceanic Sciences, University of Wisconsin-Madison, Madison, WI 53706, USA

SUMMARY

Physical dimensions and units form an essential part of the specification of constants and variables occurring in scientific programs, yet no standard compilable programming language implements direct support for automated dimensional consistency checking and unit conversion. This paper describes a conceptual basis and prototype implementation for such support within the framework of the standard Fortran 90 language. This is accomplished via an external module supplying appropriate user data types and operator interfaces. Legacy Fortran 77 scientific software can be easily modified to compile and run as "dimension-aware" programs utilizing the proposed enhancements.

KEY WORDS: dimensions and units, scientific programming, Fortran 90

Introduction

Despite the growing popularity of C++, Java and other modern object-oriented languages in the broader programming community, Fortran remains the predominant language for the physical sciences. The reasons are partly historical: almost all physical scientists and engineers in mid-career or older, and many younger scientists as well, cut their teeth on Fortran and perceive no compelling reason to learn a new language. Moroever, there exists a huge legacy of scientific and numerical routines in Fortran-77 that would be time consuming and expensive

*Correspondence to: Atmospheric and Oceanic Sciences, University of Wisconsin-Madison, Madison, WI 53706, USA
†E-mail: gpetty@aos.wisc.edu
Contract/grant sponsor: National Aeronautics and Space Administration; contract/grant number: NAG-1233 and NAG5-7741

to convert to C++ or some other more modern language. But the problem is not merely one of inertia: most languages other than Fortran seem to have been designed with little regard for the unique needs of scientific programmers, especially as they pertain to common floating point operations, such as exponentiation, and operations involving multidimensional arrays. An excellent summary of some of the shortcomings of C (for example) as a scientific programming language is given by Press et al. [1].

For all of the above reasons, and despite its other significant shortcomings (many of which have been corrected in Fortran 90) Fortran is unlikely to be abandoned any time soon as the programming language of choice for physical scientists. Nevertheless, even Fortran exhibits a glaring deficiency in the context of modern scientific computing: lack of built-in support for physical dimensions and units.

Like all other standard computer languages, Fortran, by default, treats floating point variables as if they involved "pure" (physically dimensionless) numbers. In reality, the variables in most scientific and engineering problems have values that represent not pure numbers but rather physical quantities, such as distance, speed, electric charge, etc.

For example, the velocity of light is commonly represented in abstract form by the variable c. It is the distance that light travels per unit time in a vacuum. As such, it has physical dimensions of length divided by time. It is important to understand that, as a physical quantity, this velocity is a constant – light always travels the same distance in a given time interval. However, in some books one may read that c has a value of 2.998×10^8 meters per second, in others (mostly older) one finds 186,300 miles per second, etc. Both values represent the same physical velocity; the only difference is the choice of units for expressing the physical dimensions of length divided by time.

Thus, the value of the speed of light, like that of any physical constant or variable, has both a dimensional part and a "pure" magnitude, not unlike the magnitude of a vector. The *numerical representation* of that magnitude depends on the choice of units used to represent length and time. No matter which set of units is chosen, however, the *product* of the quantity's magnitude and its dimensions represents an invariant physical quantity. The conversion between different systems of units is nothing more than an adjustment of the numerical representation of the magnitude to compensate for the use of larger or smaller units. In summary, *to be complete and unambiguous, any representation of the value of a physical variable must contain not only a numerical magnitude but also either an explicit or implicit statement of the units employed, which in turn must be consistent with the underlying dimensions.*

The issue is more fundamental than merely a concern about ambiguity. The existence of physical dimensions in scientific problems also imposes a strong, and extremely useful, constraint on the types of computations which are even physically admissible. Violations of dimensional consistency invariably reveal a conceptual or computational error, to be ignored or overlooked at one's peril.

In particular, there is no physical problem for which it is ever physically meaningful to attempt any of the following:

1. Add, subtract, or equate two values having different physical dimensions. For example, it is never physically meaningful to add a scalar variable with dimensions of time to one having dimensions of distance. Any expression which violates this rule is simply incorrect.

2. Supply anything but a pure dimensionless number as an argument to any transcendental functions, such as sine, logarithm[†], exponent, etc. For example, it is never meaningful to take the sine of a length, but it is meaningful to take the sine of ratio of lengths (e.g., the distance coordinate x divided by a wavelength λ).

All standard programming languages today, by default, place the burden on the programmer and, possibly, the program user to (1) keep track of which units are being assumed or expected by any given routine, (2) ensure that mathematical operations on physical variables are sensible, (3) explicitly (and correctly) program any necessary conversions between various systems of units during input and/or output, and (4) evaluate the reasonableness of the output of a particular code fragment based solely on the values of the "pure" numbers returned by that fragment.

It has been the author's experience that many of the most tedious programming and debugging problems in scientific software are direct consequences of of this burden, especially in connection with the passing of arguments between off-the-shelf routines written by different authors. The central thesis of this paper is that it is an unnecessary burden, because *provisions for proper consideration of physical dimensions can be incorporated both naturally and generally into the definition of any computer language intended for scientific or engineering computation.* Moreover, through the utilization of defined data types and operator overloading, it is even possible to add this capability to some standard languages, such as Fortran 90, without the need for compiler-dependent language extensions.

This is not a new idea. The popular commercial software package *MathCad* (by MathSoft, Inc.) used for WYSIWYG-type mathematical calculations rigorously enforces dimensional consistency and performs implicit unit conversions, making it far more difficult for errors of physical reasoning or unit conversion to occur and go undetected. Unfortunately, the same philosophy has yet to make significant inroads into widely used, compilable scientific programming languages.

A few proposals attempting to address these concerns have been published. Van Delft [2] in particular gives an excellent overview of the basic issues of physical dimensions in scientific programming and correctly notes significant conceptual shortcomings in the treatment of this problem by previous authors. He further demonstrates an extension to the Java language that successfully implements many operations involving physical dimensions.

Van Delft's proposal comes the closest of any known to this author to effectively and generally implementing the use of physical dimensions within a programming language. However, some significant additional improvements seem possible. The present paper therefore departs from Van Delft in four important respects:

1. Java seems extremely unlikely to displace Fortran as the programming language of choice for most physical scientists and engineers. Therefore this paper focuses on solutions compatible with Fortran. Among other things, our proposal allows legacy Fortran 77

[†]In fact, the prohibition against supplying a physically dimensioned value as an argument to a logarithm function is frequently violated. However, such cases invariably entail an implicit division by a standard reference unit, or else arise in cases where an arbitrary additive constant in the result may be ignored.

code, with relatively minor modifications, to be compiled under standard Fortran 90 so as to take advantage of a external module conferring "dimension-awareness".

2. Van Delft advocates the definition of multiple distinct data types to embody all different combinations of dimensions for variables utilized in a given program. Among other things, this requires the physical dimensions of each variable to be known *a priori*, thus again placing the burden on the programmer to pre-compute by hand, and declare, the dimensions of *all* variables, even those intended only to hold intermediate results. Also, under his system, all dimensional typing and checking is performed at compile time, precluding the development of software that can operate on variables whose physical dimensions are known only at run time. By contrast, we advocate the definition of *one generic data type* (which may have more than one "kind") capable of dynamically storing *any* physically dimensioned value (within the limits imposed by the selected set of base dimensions – see below), allowing the actual physical dimensions of a variable to be computed and/or modified at run time if desired.
3. Van Delft's proposed extension apparently cannot readily accommodate variables with dimensions raised to non-integer powers, even though such variables routinely occur in the physical sciences.
4. Van Delft's proposal entails an actual language extension, requiring modification of existing compilers in order to accommodate the extension. By contrast, the present proposal demonstrates that many of the most desirable objectives can be achieved through operator overloading and defined data types without actually extending a language or modifying a compiler. In particular, we describe and demonstrate a prototype external module implementing physical units and dimensions that can be invoked by any program compiled with any standard Fortran 90 compiler.

In the remainder of this paper, we discuss implementation issues potentially applicable to any modern language incorporating dimension-awareness. However, concrete illustrations are given in the form of Fortran 90 code fragments.

Conceptual Implementation

Data Types

At the most basic level, the implementation of dimension-awareness requires two steps: (1) the definition of a new generic data type which is used in much the same way as the existing type `REAL` but which allocates additional internal storage to dimensional information, and (2) the overloading of standard operators so as to recognize the new data type.

Let us call the new data type `preal` (i.e., "Physical REAL"). Variables of type `REAL` and of type `preal`[‡] may coexist in a single program; `REAL` variables would simply be treated

[‡]Here and throughout this paper, upper and lower case, respectively, are utilized to distinguish standard Fortran keywords from user-defined symbol names

computationally as a special case of type `preal` in which the stored values are by definition dimensionless. Existing software which does not make use of type `preal` would continue to compile and run without change. The relationship is analogous to the existence of type `REAL` as a special case of type `COMPLEX`.

All standard algebraic operators (`+`, `-` , `/`, `*`, `**`) would apply in the usual way to expressions of type `preal`. However, at either compile time or run time as appropriate, operations violating the fundamental rules given earlier would be flagged as fatal errors, not unlike the case for division by zero.

The actual internal representation of dimensions in a variable is straightforward, as there are only a few distinct generic physical dimensions in wide use in the sciences. For the present discussion, we restrict our attention to the base dimensions defined by the International System of Units (SI)[3]: length (L), mass (m), time (t), temperature (T), electric current (i), luminous intensity (I), and amount of substance (n), where the letters in parenthesis indicate the particular shorthand that will be used in this paper. Although not an SI base dimension per se, it is convenient to also include a measure of angle (a), for a total of 8 base dimensions.

The vast majority of derived physical units in the sciences involve products and powers of these base dimensions. For example, the following table shows how a few common quantities are represented in terms of integer powers of the base dimensions:

	L	m	t	T	i	I	n	a
Distance	1	0	0	0	0	0	0	0
Speed	1	0	-1	0	0	0	0	0
Acceleration	1	0	-2	0	0	0	0	0
Force	1	1	-2	0	0	0	0	0
Pressure	-1	1	-1	0	0	0	0	0
Energy	2	1	-2	0	0	0	0	0
Electric charge	0	0	1	0	1	0	0	0
Molecular Weight	0	1	0	0	0	0	-1	0
Angular velocity	0	0	0	-1	0	0	0	1

As the examples above illustrate, the powers involved for each dimension are usually small for most real-world physical problems. If one restricts one's attention for the moment to integer powers falling between -8 and $+7$, one could theoretically encode the dimensional part of the vast majority of common derived units with a total of 28 bits of storage, representing each of the 8 base physical dimensions raised to a four-bit signed power. Thus, a single-precision variable of type `preal` could be implemented with a total of 64 bits of storage, 32 for the numerical part and 32 for the dimensions.

In practice, computation on 4-bit entities usually entails considerable overhead at the hardware level, as do unaligned data accesses in core memory. Consequently, in our prototype Fortran 90 implementation of the defined data type `preal`, we allocate a full byte to each dimension, so that the entire data object requires three 32-bit words of storage. Therefore, our prototype external Fortran module begins with the following lines:

```
MODULE physunits
  IMPLICIT NONE
  PUBLIC

TYPE preal
  SEQUENCE
  REAL :: val
  INTEGER(KIND=1) :: dims(8)
END TYPE preal
```

Operators

Algebraic operations on type `preal` values are straightforward. In Fortran 90, they are easily implemented in the external module via operator overloading.

The primary cases of interest are the following three (recall that transcendental functions can only operate on non-dimensional numbers):

1) `C = A + B` or `C = A - B`

This case is simple because the operation is only permitted if the dimensions of A and B are identical, in which case the result C has the same dimensions.

2) `C = A*B` or `C = A/B`

In the case of multiplication or division of physical quantities, we are also multiplying or dividing the respective dimensions, which is equivalent to addition or subtraction of the powers of each base dimension. Thus, an operation involving the product of a force and a distance can be represented as the addition of the respective exponents:

```
                      L  m  t  T  i  I  n

Force                 1  1 -2  0  0  0  0
Distance              1  0  0  0  0  0  0
                     ---------------------

Energy                2  1 -2  0  0  0  0
```

3) `C = A**B`

Exponentiation (including root-taking) is similarly straightforward. To start with, the operation above is only permitted at all for non-dimensional `B`. The resulting dimensions for `C` are then simply `B` times the numerically encoded dimensions for `A`. Thus, if `area = d**2`, where `d` has dimensions of length (1 0 0 0 0 0 0), then the computed dimensions of `area` are represented by $2 \cdot (1\ 0\ 0\ 0\ 0\ 0\ 0) = (2\ 0\ 0\ 0\ 0\ 0\ 0)$.

A minor complication arises in the case of non-integer values of `B`, which are not only physically admissible but also occur regularly in scientific problems — e.g., when taking square roots, in which case `B` = 0.5. In such cases, the dimensions of the result may be represented via low-precision real values requiring somewhat greater storage than 8-bit integers. For example, 16 bits per base dimension would permit fixed-point representations of powers between −8.00000 and +7.99976 to a precision of 0.00024, which should be adequate for most common applications.

Thus, one might introduce a second instance of generic type `preal` which requires more memory but which specifically accommodates non-integral physical dimensions. (This generalization is not yet implemented in the prototype module described herein.) With the more general instance of `preal` it would no longer be desirable to insist on exact dimensional consistency in mathematical expressions, as minor discrepancies in dimensions will be introduced by round-off error into otherwise valid expressions. Rather, it is necessary to specify some level of tolerance in testing for equality of non-integral powers and an algorithm for resolving minor discrepancies.

Systems of units and unit conversions

We have addressed the internal binary representation of the *dimensions* of a quantity but not the role of *units*. Clearly the numerical value of a length, for example, depends on whether the length is expressed in millimeters or miles. The solution is to store all units and physical magnitudes internally referenced to a single standard unit of measure for each base dimension. For example, the SI system uses meters for (L), kilograms for (m), seconds for (t), degrees Kelvin for (T), amperes for (i), moles for (n), candlepower for (I), and radians for (a). In our Fortran 90 module `physunits.f90`, we therefore define a set of eight constants with descriptive names (beginning with the prefix `u_'` to distinguish them from other variable names), each constant representing one of the base units.

```
! SI base units
TYPE(preal), PARAMETER  :: u_meter       = preal(1.0, (/ 1,0,0,0,0,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_kilogram    = preal(1.0, (/ 0,1,0,0,0,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_second      = preal(1.0, (/ 0,0,1,0,0,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_kelvin      = preal(1.0, (/ 0,0,0,1,0,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_ampere      = preal(1.0, (/ 0,0,0,0,1,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_mole        = preal(1.0, (/ 0,0,0,0,0,1,0,0 /) )
TYPE(preal), PARAMETER  :: u_candlepower = preal(1.0, (/ 0,0,0,0,0,0,1,0 /) )
TYPE(preal), PARAMETER  :: u_radian      = preal(1.0, (/ 0,0,0,0,0,0,0,1 /) )
```

The above-defined units may then be used in expressions exactly like any other variable, as illustrated in the following simple program:

```
PROGRAM EXAMPLE
USE physunits
```

```
TYPE (preal) ::  speed, power, force

speed = 10.0*u_meter/u_second    ! 'speed' takes on value of 10
                                   !  meters per second
  ...
```

Other derived units may also be predefined in the module `physunits.f90`, as follows,

```
TYPE(preal), PARAMETER  :: u_foot   = preal(0.3048, (/ 1,0,0,0,0,0,0,0 /) )
TYPE(preal), PARAMETER  :: u_minute = preal(60.0,  (/ 0,0,1,0,0,0,0,0 /) )
```

or else they may be computed by the programmer at run-time:

```
TYPE(preal) :: u_mile,u_joule

u_mile = 5280.0*u_foot
u_joule = u_kilogram*(u_meter**2)/(u_second**2)
```

after which the new symbols may be utilized in exactly the same way as any of the pre-defined unit symbols.

Since the internal system of units affects only the binary representation of a value and is not directly visible to the programmer or the program user, the actual choice is arbitrary, provided only that the binary floating point representation of the numerical part of the value can span the full range of the possible orders of magnitudes encountered in any reasonable application without the danger of overflow or underflow. This is not necessarily a trivial concern. For example, plausible energies encountered in scientific programs can span well over a hundred orders of magnitude, ranging from those associated with subatomic particles to those released by a supernova.

This problem can be dealt with either by simply allocating enough bits to store a sufficently wide range of exponents in the floating point format to cover any reasonable contingency or else by allowing the set of internal base units for a given application to be specified by the programmer at compile time. For example, atomic physicists could reduce the risk of run time underflows by employing internal base units of nanometers for length, while astrophysicists might do better to use light years. In Fortran 90, this potential need to customize is easily addressed by creating slightly different "flavors" of the external physical units module in which the base units (and possibly the set of pre-defined derived units) are optimized for different disciplines.

Once the set of units required by a particular program module have been defined, the operator and assignement interfaces written into the `physunits` module are responsible for ensuring that only valid expressions are executed. Thus, the following fragment executes without error:

```
  v = 10.0*u_meter/u_second
  m = 5.0*u_kilogram
  g = 9.807*u_meter/(u_second**2)

  kinetic_energy = 0.5*m*v**2
  potential_energy = m*g

  total_energy = kinetic_energy + potential_energy
```

while this fragment does not:

```
  x = 1.0*u_meter
  y = 1.0*u_second

  z = x + y   ! generates run-time error
```

Note that there is no need to use consistent sets of *units* in expressions, since all available units are merely symbols whose value is defined in terms of the base units. Thus, the following line substituted for the previous assignment to `g` would be acceptable and would yield approximately the same results for the value of `potential_energy`:

```
  g = 32.175*u_foot/(u_second**2)
```

Regardless of the internal representation, the user will usually wish to display or store the final results of a series of calculations in a preferred set of units. To do this, divide the value in question by the preferred combination of pre-defined or user-defined units. The result will be a non-dimensional value giving the correct magnitude of the value expressed in those units. This value can be "cast" to type `REAL` and subsequently output to a file or terminal or passed to another traditional (dimension-unaware) Fortran subroutine in the usual way:

```
  TYPE preal :: u_erg, u_joule
  TYPE REAL :: energy_in_ergs, energy_in_joules

! following definitions may be hidden in PHYSUNITS module
  u_erg = u_gram*(u_centimeter**2)/(u_second**2)
  u_joule = u_kilogram*(u_meter**2)/(u_second**2)

! following statements generate run-time error if expressions on right
! are not non-dimensional

  energy_in_ergs = total_energy/u_erg
  energy_in_joules = total_energy/u_joule
```

```
WRITE(*,'(''Total Energy in ergs: '',e12.5)energy_in_ergs
WRITE(*,'(''Total Energy in joules: '',e12.5)energy_in_joules
```

Comments on storage requirements and computational overhead

One practical objection to the scheme described is the significant increase in storage required for variables of type `preal` (12 bytes at single precision), relative to what is required for the traditional type `REAL` (4 bytes). However, recall that the memory requirements of most large scientific programs are traceable to large arrays, not to large numbers of individually declared scalar variables. For many if not most scientific applications, it is reasonable to expect numerical arrays to represent a vector or matrix of values all having the same physical dimensions, in which case a single binary dimension field of eight bytes would be appended to the total storage requirement for a floating point array. The efficient and transparent implementation of such a solution in the case of Fortran 90 has not been investigated and might require an actual extension to the language definition, unlike the case described for scalar physical variables.

A second possible objection is the increase in computational overhead required by operations on the new data types. In some scientific programming applications, such as fluid dynamical simulations, numerical efficiency is of paramount concern. In many others, the programming effort that goes into writing and debugging a particular library of routines may be comparable to or even far greater than the actual expected execution time over the useful life of the routine. In the latter case, the added execution time may be more than compensated by a significant reduction in programming and debugging effort.

Even for the case that efficient routine execution of a numerically intensive program is required, there is a surprisingly simple solution:

1. Write the software so as to utilize the physically dimensioned data type.
2. Compile and run the software in a debugging mode so as to verify that no fatal errors arise pertaining to inconsistent physical dimensions.
3. Leaving the program code unmodified, recompile it with a "dummy" Fortran 90 module in which type `preal` reverts internally to the standard scalar type `REAL` and dimensional information is silently discarded. The program will run with normal scalar efficiency but will nevertheless benefit from the physical "sanity checking" of the previous step and it will continue to benefit from automated unit conversion in the recompiled program.

Concluding Remarks

In this paper, a basis is described for implementing automated computation and consistency checking of physical dimensions and units in scientific programs. The main features described herein do not require language extensions or compiler modifications. Rather, a reasonably

compact external module written in standard Fortran 90 can implement the relevant type definitions and operator interfaces, making it possible to immediately begin utilizing the proposed enhancements at any installation equipped with a standard Fortran 90 compiler. Furthermore, legacy Fortran 77 code can be upgraded with relatively little effort to utilize the enhancements.

A prototype version of the module and sample programs are available for download as `http://meso.aos.wisc.edu/~gpetty/physunits.tar`. It is the author's hope that this prototype will serve as the nucleus for future community development of a complete, rigorously tested, and efficient implementation of the concepts discussed herein.

ACKNOWLEDGEMENTS

The development of the prototype Fortran 90 physical units module described herein was undertaken in support of the author's scientific research funded by NASA Grants NAG-1233 and NAG5-7741. The author thanks R. Pincus for helpful discussions.

Appendix - Sample Program `physunits_demo.f90`

```
!-----------------------------------------------------------------------
! compile and run under Unix with any standard F90 compiler as follows:
!
!   f90  physunits_demo.f90 physunits.f90
!   a.out
!
! Example of a simple program to compute the total mass of the
! atmosphere, given the radius of the earth and the average atmospheric
! surface pressure.  It is deliberately written to use Fortran 77-style
! syntax wherever possible in order to highlight the relative ease of
! converting legacy code so as to utilize the Fortran 90 PHYSUNITS
! module.
!
! The key point to recognize in this example is that the programmer is
! able to use completely arbitrary units to specify the values of
! various constants used in the calculation and let the computer do all
! of the internal conversions to a consistent set of units.  The
! programmer can then choose an arbitrary set of units for displaying
! the results of the computation.  Also, there is run-time checking for
! illegal operations involving physically dimensioned quantities.
!-----------------------------------------------------------------------

      program units_demo

      use physunits
      implicit none

      type (preal)  radius,grav,sfcpres,sfcarea,atmos_mass
      real acres1,acres2,pi
```

```
      parameter (pi=3.14159)

! Note:  By convention, all variable names with prefix 'u_' identify
! physical units or constants predefined in module PHYSUNITS

! Radius of the earth specified in kilometers
      radius = 6370.*u_kilometer

! Acceleration due to gravity specified in feet per second squared
      grav = 32.2*u_foot/(u_second**2)

! Average surface air pressure specified as one standard atmosphere
      sfcpres = 1.0*u_atmosphere

! Calculate earth's surface area using generic formula
      sfcarea = 4.*pi*radius**2

! Print out the Earth's surface area in units of acres.
! Because 'acres1' is  an ordinary scalar 'real' variable, this
! assignment would generate a run-time error if the ratio
! sfcarea/u_acre were for some reason found not to be dimensionless
! due to a previous programming error.

      acres1 = sfcarea/u_acre
      write(*,*)' Earth surface area = ',acres1,' Acres'

! calculate mass of the atmosphere using generic formula based on
! hydrostatic balance

      atmos_mass = sfcarea*sfcpres/grav

! Print out the mass of the atmosphere in units of kilograms.
! Function 'nondim' casts variable of type PREAL to REAL, assuming that
! it is non-dimensional; otherwise generates run-time error.

      write(*,*)' Atmospheric mass = ', nondim(atmos_mass/u_kilogram) , ' kg'

! An attempt to print out the mass of the atmosphere in inappropriate
! units.  If not commented out, this line would generate a run-time error
! because the ratio is not dimensionless and therefore cannot be assigned
! to 'FACRES', which is of standard type REAL
!
      acres2 = atmos_mass/u_acre
      write(*,*)'Atmospheric mass = ',acres2,' Acres'

      stop
      end
```

REFERENCES

1. Press W, Teukolsky S, Vetterling W, Flannery B. *Numerical Recipes in C: The Art of Scientific Computing (2nd Ed.)* Cambridge Univ Pr, 1993.
2. van Delft, A. A Java extension with support for dimensions. *Software—Practice and Experience* 1999; **29**(7):605–616.
3. SI units page.
http://imartinez.etsin.upm.es/ot1/Units_uk.html [10 July 2000].