Research Inventy: International Journal of Engineering And Science
Vol.7, Issue 6 (June 2017), PP -11-21
Issn (e): 2278-4721, Issn (p):2319-6483, www.researchinventy.com

Fortran 90 Programming With Physical Unit Annotations

Albert S. Kim[1,*], Man Hin Leung[1], Jung-Hyun Moon[2], Sung Woo Kim[2]
[1]*Civil and Environmental Engineering, University of Hawaii at Manoa, 2540 Dole Street Holmes 383, Honolulu, Hawaii 96822, USA*
[2]*Seawater Utilization Plant Research Center (SUPRC), Korea Research Institute of Ships & Ocean Engineering, 124-32, Simcheungsu-gil, Jukwang-myeon, Goseong-gun, Gangwon-do 219-822, Republic of Korea*
Corresponding author: Albert S. Kim

Abstract: Scientific computation aims to analyze and predict natural and engineered phenomena for human understanding. Specific quantification of physical variables requires annotations of physical units, either basic, or derived. State-of-the-art computation uses the object-oriented programming (OOP) features using a number of processing cores. In translating scientific governing equations to computer programming, developers are still responsible to use and convert physical units to provide meaningful results. In this paper, we indicate some incoherency of SI units and propose possible ways to minimize unintended numerical errors. We use standard Fortran 90 as a programming language and discuss how physical units can be specified and checked during code compilations and executions.

Date of Submission: 17-02-2017 Date of acceptance: 20-9-2017

I. Introduction

Computational research in scientific disciplines often requires computer programming using advanced languages. Coding refers to translation of mathematical and logical representations using the programming languages as interfaces between humans and computers. These programming languages can be classified into compilers and interpreters. Compiling languages such as C/C++ and Fortran (FORmular TRANslator) 77/90 create executable files, which run faster than script languages, but are not usually compatible between different operating systems (OSs) [1]. Script languages such as python, Java script, R, Perl and tcl/tk interpret and process written instructions in their kernels sequentially. Advanced script languages such as Mathematica, MATLAB/Octave, Maple/Maxima, and python, use a wide variety of prebuilt numerical and graphical libraries for efficient and flexible simulations. A language developed by Google Inc., called "go", is a new interpreter, which can compile go-scripts to make executable files. Usage of "go" in the scientific community is limited due to the lack of prebuilt libraries. These script languages provide a convenient programming environment in which programmers can test and revise local codes intermittently. Coding statements, checking values, and debugging lines can be done almost simultaneously in the development stage. However, script languages are often limited to small, serial applications using a single process unit. For large-scale parallel computation, compiling languages are still predominantly used in parallel computing communities because compilers provide deterministic memory handling and invariant data types.

Parallel programming uses an open multi-processing interface (OMP) and a message passing interface (MPI), which are executed in shared and distributed memory systems, respectively. OMP-MPI hybrid programming is possible using distributed hardware consisting of multi-core CPUs. Parallel communication requires robust specification of data types and sizes for cooperative execution of distributed tasks among computing cores. Changing a data type of a variable using dynamic memory allocation is highly discouraged in parallel programming although it is considered as an advantage in serial script programming. In this case, global data should be well structured and assigned for processors to access, modify and share. Even distribution of tasks among processors, called load balance, is a key for successful parallel computing, which requires data specification. The conventional compilers such as C and/or F77 for numerical computation are designed for structured programming using the basic data types (e.g., integers, float or real, double) and structures (e.g., scalars, vectors, and matrices). In the past decades, object-oriented programming (OOP) became a critical programming feature in developing large-scale, high-performance, parallel programs for supercomputers. OOP developers can customize data structure as a combination of standard types to enhance speedy development and scalable execution [2].

In computational research, governing equations are solved for physically meaningful variables using adequate algorithms. Initial and boundary conditions need to be properly set prior to simulation runs. Fixed parameters are often stored separately and retrieved from customized libraries. While numerical computations consist of a series of basic operations such as addition/subtraction and multiplication/division of variables, their numerical values have specific physical units (i.e., dimensions). If the variables containing physical units, for example momenta, forces/torques, and energies in classical or statistical mechanics, are compared and operated, they should have the same physical units. Use of consistent units can significantly minimize any unintended errors and noticeably increase the computational reliability. If two variables of different units are multiplied, then the product should have a new physical meaning, for example, mass times a linear velocity gives the linear momentum ($p = mv$). A perfectly inelastic collision makes two individual objects one combined body, having a total mass equal to the sum of two masses. In general, only variables of the same unit can be added or subtracted. Although computational speed has been enhanced tremendously in the last few decades, handling units of physical variables still remains in a burgeoning stage, and is still subject to the responsibility of programmers and end-users. This is primarily because the programs are developed to only compute numerical and logical values in the mathematical aspect, and more importantly, most programming languages do not have intrinsic methods to deal with standard physical units for scientific and engineering calculations.

There have been a number of programming languages used for computational purposes for sciences and engineering. As both hardware and software develop quickly to meet needs and requirements for specific applications, no single programming language is predominantly used. From early 1970s to the present, compilers such as C/C++ and Fortran 77/90 have been widely used in communities of computer and software engineering. Fortran 77 was standardized in 1977, and Fortran 90 includes some advanced feature of C/C++, which are dynamic memory allocation and free format. Object oriented programming (OOP) features are officially included in Fortran 003 [3,4]. In general, C/C++ has much wider scope and functionalities than the latest Fortran, but there are several reasons that Fortran will continue to be used [5]. This is because, in our opinion, standard Fortran compilers continually incorporate advanced programming features such as OOP and polymorphism, some applications previously developed using Fortran do not require any updates, and more importantly legacy libraries [6] built using Fortran in the 1980s and 1990s can be called into C/C++ programs using cross-compiling. In our opinion, Fortran is a language, in which code lines are very close to pure mathematical formula. In this paper, we aim to develop a programming method to use and check physical units during compilation and execution phases using Fortran90 as an OOP compiling language for computational research and education. Our approach can be, however, readily applied to other OOP-featured languages.

II. Background

Software using physical units

Basic units used in physics and engineering are listed in Table 1 from the National Institute of Standards and Technology (NIST) [7]. The seven basic quantities are length, mass, time, (electric) current, temperature, substance number, and luminous intensity. To the best of our knowledge, there are only a limited number of software programs which incorporate physical units during simulations. OpenFOAM (OF), an open source software for computational fluid dynamics (CFD), is one of them and it became a popular package recently for multi-physics simulations in various engineering disciplines [8]. Basic units employed in OpenFOAM are identical to those in Table 1 with a slightly different sequence. Since OpenFOAM was developed originally for CFD, quantities for chemical, electrical, and optical phenomena are less frequently used. Table 3 includes units and their unit arrays of frequently used physical variables in science and engineering. Physical quantities are classified by their nature.

Related work

To the best of our knowledge, the idea of implementing physical units into numerical computation was proposed by Petty [9]. A new data type, preal ("Physical REAL") was introduced as a combination of a REAL (float) variable and an integer array of seven elements (let us call it a unit array). All derived units such as speed, acceleration, force, and energy are represented using the seven integers. Three types of operations are considered: addition/subtraction, multiplication/division, and exponentiation of the real variables. When two real variables have the same unit(s), addition and subtraction of the two variables requires checking the identicalness of their unit arrays. For multiplication and division of two real variables, the unit arrays are added or subtracted since the seven elements represent the power of the basic units. When one real variable is squared, each of the elements needs to be multiplied by 2. In addition to standard unit handling, unit conversion between two different units of the same variables was also considered, e.g., feet vs. meters, and meters per second vs. miles per hour.

Recently, some SI units were thoroughly investigated for periodic phenomena by Mohr and Phillips [10]. By convention, the angular displacement is measured using degrees or radians. Due to the direct connection between the angular displacement and the perimeter length, radians are more frequently used in computational research, while degrees are used mainly for educational purposes. Hertz (Hz) is defined as the number of cycles per second. Accurate representation of Hz is

$$1\,\text{Hz} = 1\,\text{cycle/s} = 2\pi\,\text{rad/s} \quad (1)$$

by implicitly defining 1 cycle equal to 2π in radians. A potential confusion occurs when Hz is defined as 1/s, in which 1 can be considered either in cycles or radians. They concluded that the unit Hz cannot be regarded as a coherent SI unit, because if cycles are omitted then Hz may be replaced by 1 (rad)/s. In our opinion, clarification of the angular frequency or angular velocity, often denoted as ω, can be made using units of either rev. (revolution)/s or rad/s. If a certain object performs a periodic motion for a long duration, rev./s is an intuitive, convenient unit. On the other hand, if relative motion of two colliding bodies is of interest, then the angular displacement $\Delta\theta$ must be much smaller than 2π so that rad/s can be a proper unit. The primary reason that radian is often omitted in the unit handling is because it is defined as the ratio of an arc length divided by the radius of a circle. To eliminate any confusion and reduce numerical errors, it must be better to, in our opinion, include radian as a standard unit.

Orchard et al. [11] claimed that the concept of inferring units of measure have been established in the research literature for a long time, but has not been actively adopted in scientific computing. They extended the Fortran language which allows automatic verification of units and more importantly developed a technique for reporting to the user a set of critical variables which should be explicitly annotated with physical units. Once these features are officially implemented in the Fortran standard or other languages, computational research on physical phenomena will be ~~in~~ a more robust programming framework. Contrastin et al. [12] further emphasized the importance of using units in scientific computation because unit annotation can significantly reduce the debugging effort, and programmers can be more confident in consistency and robustness of their codes. The next revision of the language (Fortran 2015) is planned for release in mid-2018 [13], to include further ~~interoperability between Fortran and C,~~ additional parallel features, for interoperability between Fortran and C. It is proposed that unit annotations are included in Fortran 2015 [14].

III. Programming Methods

3.1. Modules for unit annotation

Fig. 1 shows the first part of the module file containing conventional data types with specific unit annotations. A type, variable_real (line 2–7), is for a real number with a specific physical unit. In this type, there are four members: unt(1:7), dim(1:7), str, and val. The first member, unt, is an integer vector consisting of the seven elements, having the same sequence as shown in Table 2. The second and third members, dim and str, represent the physical unit in an easily recognizable form. Physically meaningful value is stored in the fourth member, val. In line 9–14, the data type of an integer variable has the same member structure. In mechanics, a vector of three components is frequently used in 3D spaces. The data type, variable_vec3, is especially for these 3-element vectors such as position, velocity, momentum and so forth. In comparison to variable_real, variable_vec3 has three physical values in a vector form. The length of a vector is arbitrarily determined by users or developers as needed. Dynamic memory allocation provides a flexible use of a customized vector by assigning the vector length (i.e., the number of elements) after the physical variable is declared. variable_vecn contains an allocatable vector vecn(:). Because each of the vector elements must have the same unit, the first three members of unit, dim, and str do not need to change in line 24–26 of Fig. 1. The same approach is used to define a new data type, variable_matn, for a matrix of arbitrary lengths of rows and columns, i.e., matn(:,:). Fig. 2 shows a function chkunts, which compares units of two variables. The returned results of this function is a logical value, denoted as ucheck, which is either true (T) of false (F). Two integer arrays are inputted as Aunit and Bunit, and their difference of each element is calculated as Cunit = Aunit - Bunit. If all the seven elements of Cunit are zero, then two variables associated to Aunit and Bunit have the same unit.

Fig. 3 is an example main program that shows how to assign a value and unit of a real, scalar variable. Initially, myMass is set as 10 kg with the unit array of (/1,0,0,0,0,0,0/) in lines 11 and 12, and x-, y-, and z-directional components of myVelocity are set as 1.0, 2.0 and 3.0 m/s, respectively, with the unit array of (/0,1,-1,0,0,0,0/). The numerical values of myMass and myVelocity are printed to the screen with specific formats in lines 18–24. Units of these variables are compared and inequality of these units is presented using a logical value false, F, in lines 26–31. A scalar myMass and a 3D vector myVelocity are multiplied to calculate myLinearMomentum as a 3D vector. The unit of myLinearMomentum is derived by adding myMass%unt and myVelocity%unt element by element implicitly. The final element-values and unit of myLinearMomentum are printed to the screen. Specific outcomes by executing the main program of Fig. 3 are shown in Fig. 4.

3.2. Modules for basic operations

Fig. 5 shows another module containing exemplary mathematical operations with unit annotations for addition of two real scalar variables, plusReal, and dot-product of two vectors of arbitrary length, dot_vectors. Usage of these operators is shown in a main program of Fig. 6. In the main program, basic_app.f90,

- Scalar values of myMass [kg] and myTime [s] are set to be 10.0 and 5.0 in lines 10 and 14, respectively.

- In line 19, these two heterogeneous variables are added using the customized operator (.plusReal.), and, as expected, the operation is failed to calculate mySum1.
- In line 27, two scalar variables having both a dimension of time ~~time dimension~~ are added and its sum, mySum2, is properly calculated.
- In lines 34 – 39, myVelocity1 is set to be (1.0, 2.0, 3.0) [m/s] and the member myVelocity2%vec3 is set to be 2.0*myVelocity1%vec3.
- In line 51, units of myVelocity1 and myVelocity2 are checked.
- In line 54, the two velocities are multiplied using customized operator (.dot.) and the magnitude is calculated as mySpeedSq%val = $(1.0,2.0,3.0) \cdot (1.0,2.0,3.0)^T = 28$. mySpeedSq has the unit of speed squared, which are calculated as the sum of unit-arrays of myVelocity1 and myVelocity2 in line 55.

Specific outputs to the screen are captured and shown in Fig. 7. More functions can be developed and included in basicop module. Subtraction of real B from real A can be easily calculated by replacing B by -B and adding -B to A in function plusReal in Fig. 5. Note that lengths of two input vectors A and B to dot_vectors are not specified. As long as the sizes of vectors A and B ~~vectors~~ are the same, .dot. operation will proceed. One can extend dot_vectors to dot_matrices for multiplication of two matrices, having the first matrix's column number equal to the second matrix's row number. But, f90 already has an intrinsic function of matmul, requiring the same structural condition for two multiplied matrices. In this case, only units can be compared and/or added. Exponentiation operation can be done similarly to multiplication. The unit-array needs to be multiplied by the exponent power.

If the future Fortran release has a functionality that data type and its member structure of an input variable can be detected inside functions and subroutines, then the current approach of having three additional members, %unt, %dim, and %str, can be much more efficient. To the best of our knowledge, this is not possible in general. Pointers can be used for the same purpose, but it still requires pointers be declared ~~pointers~~ with specific data types (either basic or customized).

IV. Concluding Remarks

Fortran is a programming language, originally developed for numerical computations. Recent releases of Fortran compilers include OOP features for advanced programming. A physical variable can be treated as an object having at least two members, i.e., the inter array of seven elements of unit exponents and a data member (of scalar, vector, or matrix). Using the unit-array, developers can easily check the consistency of physical units during a series of arithmetic operations. For addition and subtraction between two physical quantities, the identicalness of two quantities needs to be confirmed. For multiplication and division of two quantities, the resulting quantity should have a unit as an addition and subtraction of two unit-arrays, respectively.

We believe that the unit annotation will be included in advanced programming languages as such annotation is highly needed for coherent and robust computation. As described above, the next release of Fortran 2015 in 2018 or a later version may have intrinsic ways to include the unit annotations by declaring variables or using functions (or subroutines). Conversion between two units of the same variables is also suggested such as between meters [m] and feet [ft], meter per second [m/s] and miles per hour [mile/h], and pounds per squared inch [psi] and Pascal [Pa]. If these unit-handling features are included in the future version of Fortran, it ~~must~~ would be the first compiling language, ~~which can~~ that could check and convert units of physical variables. There are seven basic units from which conventional units are derived. In our opinion, adding radian as the eighth basic unit may enhance coherency of the unit annotations, especially for applications in rotational dynamics and wave mechanics.

GNU released version 2.13 of a software package, Units, which can convert quantities expressed in one measurement system to its equivalent in another system [15]. GNU Units has an annotated, extendable database of more than two hundred prebuilt measurement units. In a Unix/Linux terminal, executing 'units' in a command line will initiate its usage. GNU Octave has an interface to GNU Units [16], which allows programmers to declare both value and unit of a quantity. The Octave syntaxes with Units are very similar to OOP programming in C++. The main object and its members are separated by a dot (.), while the separator of Fortran is %. GNU gfortran uses % only and Intel Fortran (ifort) allows the use of either dot (.) or %. GNU Units is written in C and easily compiled using gcc. There are only five .c files with a total number of lines of about nine thousand. We believe that incorporating GNU gfortran and Units can be the first consistent step to handle unit annotations in Fortran programming. This may include forceful declaration of units even for numbers in pure mathematics.

One of biggest differences between f77 and f90 is the dynamic memory allocation. f77 has, however, implicit dynamic memory allocation if an array size is small. One can pass a matrix name and its sizes as arguments of a subroutine while the matrix sizes are not predetermined in the subroutine. It is also possible to pass a 2D matrix to a subroutine and receive it as a packed 1D array in the subroutine. These sophisticated features of f77 are clearly included in f90 as standard functionalities. In f90, if a variable is declared an intrinsic data type such as a real array, its size N can be measured inside the subroutine. If developers define new customized data types as objects consisting of a number of members of different intrinsic data types, then the specific internal structure of the object cannot be easily retrieved. This means the full information of an objects data type and structure should be passed to functions and subroutines in f90. If an arbitrary variable can be passed to functions and subroutines and its internal data structure can be retrieved using an intrinsic function such as an extension of SIZE, units can be much more easily handled during computation.

In this work, we demonstrated a possible way to handle units in Fortran 90 programming, taking advantages of OOP features. Currently, programmers are fully responsible to handling and converting units in computational and software engineering. Without OOP, it is a formidable task to coherently handle units of variables in coding and executing. Three promising ways are (1) adding intrinsic unit annotation features, (2) incorporating GNU Units, and (3) developing size/structure detecting functionalities to future standard Fortran releases. Before any of these possibilities come to reality, the method presented in this work may provide partial solutions to minimize unintended human error, not directly related to compilers and OSs, and calculate correct scientific numerical answers of specific physical meanings.

Acknowledgement

This work was financially supported by the National R&D project of "Development of 1MW Ocean Thermal Energy Conversion Plant for Demonstration" from the Korean Ministry of Oceans and Fisheries.

References

[1] Matthew G. Knepley. Programming languages for scientific computing. *Encyclopedia of Applied and Computational Mathematics*, 2012, September 2012.
[2] Damian Rouson, Jim Xia, and Xiaofeng Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, 2011.
[3] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. Expressing object-oriented concepts in Fortran 90. *ACM SIGPLAN Fortran Forum*, 16(1):13–18, Apr 1997.
[4] Damian W.I. Rouson, Jim Xia, and Xiaofeng Xu. Object construction and destruction design patterns in Fortran 2003. *Procedia Computer Science*, 1(1):1495–1504, May 2010.
[5] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V.W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, Sep 1997.
[6] http://www.netlib.org/.
[7] http://physics.nist.gov/cuu/units/units.html.
[8] https://openfoam.org and https://openfoam.com.
[9] Grant W Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience*, 31(11):1067–1076, 2001.
[10] Peter J Mohr and William D Phillips. Dimensionless units in the SI. *Metrologia*, 52(1):40–47, Dec 2014.
[11] Dominic Orchard, Andrew Rice, and Oleg Oshmyan. Evolving Fortran types with inferred units-of-measure. *Journal of Computational Science*, 9:156–162, Jul 2015.
[12] Mistral Contrastin, Andrew Rice, Matthew Danish, and Dominic Orchard. Units-of-measure correctness in Fortran programs. *Computing in Science & Engineering*, 18(1):102–107, Jan 2016.
[13] Steve Lionel. Doctor Fortran in "One Door Closes", https://software.intel.com/en-us/blogs/2015/09/04/doctor-fortran-in-one-door-closes.
[14] W. V. Snyder. ISO/IEC JTC1/SC22/WG5 N1969. Technical report, International Organization for Standardization, 2013.
[15] GNU Units. https://www.gnu.org/software/units/.
[16] Octave interface to GNU Units. http://octave.sourceforge.net/miscellaneous/function/units.html.

No.	Base quantity	Name	Symbol
1	length	meter	m
2	mass	kilogram	kg
3	time	second	s
4	electric current	ampere	A
5	thermodynamic temperature	kelvin	K
6	amount of substance	mole	mol
7	luminous intensity	candela	cd

Table 1. SI base units

No.	Base quantity	Name	Symbol
1	mass	kilogram	kg
2	length	meter	m
3	time	second	s
4	thermodynamic temperature	kelvin	K
5	amount of substance	mole	mol
6	electric current	ampere	A
7	luminous intensity	candela	cd

Table 2. SI base units used in OpenFOAM

Translational motion		
Displacement	(/ 0, 1, 0, 0, 0, 0, 0 /)	m
Velocity	(/ 0, 1,-1, 0, 0, 0, 0 /)	m/s
Acceleration	(/ 0, 1,-2, 0, 0, 0, 0 /)	m/s^2
Momentum	(/ 1, 1,-1, 0, 0, 0, 0 /)	kg m/s
Force	(/ 1, 1,-2, 0, 0, 0, 0 /)	kg m/s^2
Moment of force	(/ 1, 2,-2, 0, 0, 0, 0 /)	kg m^2/s^2
Rotational motion		
Displacement	(/ 0, 0, 0, 0, 0, 0, 0 /)	rad [-]
Angular velocity	(/ 0, 0,-1, 0, 0, 0, 0 /)	rad/s
Angular acceleration	(/ 0, 0,-2, 0, 0, 0, 0 /)	rad/s^2
Momentum	(/ 1, 0,-1, 0, 0, 0, 0 /)	kg rad/s
Torque	(/ 1, 0,-2, 0, 0, 0, 0 /)	kg rad/s^2
Plane angle	(/ 0, 0, 0, 0, 0, 0, 0 /)	
Solid angle	(/ 0, 0, 0, 0, 0, 0, 0 /)	
Mechanics		
Area	(/ 0, 2, 0, 0, 0, 0, 0 /)	m^2
Volume	(/ 0, 3, 0, 0, 0, 0, 0 /)	m^3
Pressure	(/ 1,-1,-2, 0, 0, 0, 0 /)	kg/m s^2
Energy density	(/ 1,-1,-2, 0, 0, 0, 0 /)	kg/m s^2
Energy (work)	(/ 1, 2,-2, 0, 0, 0, 0 /)	kg m^2/s^2
Molar energy	(/ 1, 2,-2, 0, 0, 0, 0 /)	kg m^2/s^2 mol
Power (radian flux)	(/ 1, 2,-3, 0, 0, 0, 0 /)	kg m^2/s^3
General transport		
Wave Number	(/ 0,-1, 0, 0, 0, 0, 0 /)	1/m
Frequency	(/ 0, 0,-1, 0, 0, 0, 0 /)	1/s
Mass transport		
Mass Density	(/ 1,-3, 0, 0, 0, 0, 0 /)	kg/m^3
Specific Volume	(/-1, 3, 0, 0, 0, 0, 0 /)	m^3/kg
Molar Concentration	(/ 0,-3, 0, 0, 1, 0, 0 /)	mol/m^3
Diffusivity	(/ 0, 2,-1, 0, 0, 0, 0 /)	m^2/s
Mass Fraction	(/ 0, 0, 0, 0, 0, 0, 0 /)	
Catalytic Activity	(/ 0, 0,-1, 0, 1, 0, 0 /)	mol/s
Catalytic concentration	(/ 0,-3,-1, 0, 1, 0, 0 /)	mol/m^3 s
Surface tension	(/ 1, 0,-2, 0, 0, 0, 0 /)	kg/s^2
Heat Transfer		
Celsius temperature	(/ 0, 0, 0, 1, 0, 0, 0 /)	K
Heat flux density	(/ 1, 0,-3, 0, 0, 0, 0 /)	kg/s^3
Heat capacity	(/ 1, 2,-2,-1, 0, 0, 0 /)	kg m^2/s^2 K
Specific heat capacity	(/ 0, 2,-2,-1, 0, 0, 0 /)	m^2/s^2 K
Molar Heat capacity	(/ 1, 2,-2,-1,-1, 0, 0 /)	kg m^2/s^2 K mol
Specific energy	(/ 0, 2,-2, 0, 0, 0, 0 /)	m^2/s^2
Thermal conductivity	(/ 1, 1,-3,-1, 0, 0, 0 /)	kg m/s^3 K
Momentum Transfer		
Dynamic Viscosity	(/ 1,-1,-1, 0, 0, 0, 0 /)	kg/m s
Kinematic Viscosity	(/ 0, 2,-1, 0, 0, 0, 0 /)	m^2/s
Electro-magnetism		
Electric charge (quantity of electricity)	(/ 0, 0, 1, 0, 0, 1, 0 /)	s A
Electric charge density	(/ 0,-3, 1, 0, 0, 1, 0 /)	s A/m^3
Current Density	(/ 0,-2, 0, 0, 0, 1, 0 /)	A/m^2
Magnetic Intensity (Magnetic field strength)	(/ 0,-1, 0, 0, 0, 1, 0 /)	A/m
Luminous flux	(/ 0, 0, 0, 0, 0, 0, 1 /)	cd
Luminance	(/ 0,-2, 0, 0, 0, 0, 1 /)	cd/m^2
Electric potential difference	(/ 1, 2,-3, 0, 0,-1, 0 /)	kg m^2/s^3 A
Capacitance	(/-1,-2, 4, 0, 0, 2, 0 /)	s^4 A^2/kg m^2
Electric resistance	(/ 1, 2,-3, 0, 0,-2, 0 /)	kg m^2/s^3 A^2
Electric conductance	(/-1,-2, 3, 0, 0, 2, 0 /)	s^3 A^2/m^2 kg
Magnetic Flux	(/ 1, 2,-2, 0, 0,-1, 0 /)	kg m^2/s^2 A
Magnetic Flux density	(/ 1, 0,-2, 0, 0,-1, 0 /)	kg/s^2 A
Inductance	(/ 1, 2,-2, 0, 0,-2, 0 /)	kg m^2/s^2 A^2
Electric field strength	(/ 1, 1,-3, 0, 0,-1, 0 /)	kg m/s^3 A
Electric flux density	(/ 0,-2, 1, 0, 0, 1, 0 /)	s A/m^2
Permittivity	(/ 0,-3, 4, 0, 0, 2, 0 /)	s^4 A^2/m^3 kg
Permeability	(/ 1, 1,-2, 0, 0,-2, 0 /)	kg m/s^2 A^2
Radiant intensity	(/ 1, 2,-3, 0, 0, 0, 0 /)	kg m^2/s^3
Radiance	(/ 1, 0,-3, 0, 0, 0, 0 /)	kg/s^3
Radioactivity		
Becquerel	(/ 0, 0,-1, 0, 0, 0, 0 /)	1/s
Absorbed dose	(/ 0, 2,-2, 0, 0, 0, 0 /)	m^2/s^2
Absorbed dose rate	(/ 0, 2,-3, 0, 0, 0, 0 /)	m^2/s^3
Dose equivalent	(/ 0, 2,-2, 0, 0, 0, 0 /)	m^2/s^2
Exposure	(/-1, 0, 1, 0, 0, 1, 0 /)	s A/kg

Table 3. A list of frequently used physical units

```fortran
module f90units
  type variable_real
     integer      :: unt(7) ! kg m s K mol A cd
     character(9) :: dim(7)
     character(63):: str
     real         :: val
  end type variable_real

  type variable_int
     integer      :: unt(7) ! kg m s K mol A cd
     character(9) :: dim(7)
     character(63):: str
     integer      :: int
  end type variable_int

  type variable_vec3
     integer      :: unt(7) ! kg m s K mol A cd
     character(9) :: dim(7)
     character(63):: str
     real         :: vec3(3)
  end type variable_vec3

  type variable_vecn
     integer      :: unt(7) ! kg m s K mol A cd
     character(9) :: dim(7)
     character(63):: str
     real, allocatable  :: vecn(:)
  end type variable_vecn

  type variable_matn
     integer      :: unt(7) ! kg m s K mol A cd
     character(9) :: dim(7)
     character(63):: str
     real, allocatable  :: matn(:,:)
  end type variable_matn

  public :: make_dim

contains

  logical function makeDim (unt,dim,str) &
       result (message)
    implicit none
    integer, intent(in) :: unt(7)
    character(9) :: dim(7)
    character(63):: str
    integer:: i
    character(3):: dummyChar
    dim(1) = " kg^(  ) "
    dim(2) = "  m^(  ) "
    dim(3) = "  s^(  ) "
    dim(4) = "  K^(  ) "
    dim(5) = "mol^(  ) "
    dim(6) = "  A^(  ) "
    dim(7) = " cd^(  ) "
    do i =1,7
       write(dummyChar,"(SP,I2)") unt(i)
       dim(i)(6:7) = dummyChar
       str((i-1)*9+1:i*9) = dim(i)
    end do
    message = .true.
  end function makeDim

```

Figure 1. The first part of module file: f90units.f90.

```fortran
  logical function chkunts (Aunit,Bunit) result (ucheck)
    implicit none
    integer, dimension(7) :: Aunit, Bunit
    integer, dimension(7) :: Cunit
    integer :: Cval, i
    Cunit = Aunit - Bunit
    do i = 1, 7
       Cval = Cval + Cunit(i)**2
    enddo

    if (Cval .eq. 0) then
       ucheck = .TRUE.
    else
       ucheck = .FALSE.
    endif
  end function chkunts

end module f90units
```

Figure 2. The second part of module file: f90units.f90.

```fortran
program dim_check
  use f90units
  implicit none
  logical :: myMessage
  type (variable_real), target  :: myMass
  type (variable_vec3), target  :: myVelocity
  type (variable_vec3), target  :: myLinearMomentum
  logical :: ucheck

  ! Set up myMass value and dimension
  myMass%val = 10.0
  myMass%unt = (/1,0,0,0,0,0,0/)
  myMessage  =  makeDim (myMass%unt,myMass%dim,myMass%str)

  ! Set up myVelocity values and dimension
  myVelocity%vec3 = (/1.0, 2.0, 3.0/)
  myVelocity%unt  = (/0,1,-1,0,0,0,0/)
  myMessage       =  makeDim (myVelocity%unt,myVelocity%dim,myVelocity%str)

  write(*,*)
  write(*,"('myMass is ',F6.3, ' kg ')")                          myMass%val
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')")  myMass%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) myMass%dim
  write(*,*)
  write(*,"('myVelocity is (',3(1X,SP,F6.2),') [m/s]')")    myVelocity%vec3
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')")  myVelocity%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) myVelocity%dim
  write(*,*)
  ucheck =  chkunts  (myMass%unt,myVelocity%unt)
  if( ucheck .eqv. .true.) then
     write(*,"('myMass and myVelocity have the same unit: ',L1)")  ucheck
  else
     write(*,"('myMass and myVelocity have different units: ',L1 )") ucheck
  end if

  ! Calculate myLinearMomentum values and dimension
  myLinearMomentum%vec3 = myMass%val * myVelocity%vec3
  myLinearMomentum%unt  = myMass%unt + myVelocity%unt
  myMessage       =  &
       makeDim (myLinearMomentum%unt,myLinearMomentum%dim,myLinearMomentum%str)
  write(*,*)
  write(*,"('myLinearMomentum is (',3(1X,SP,F6.2),') [kg m/s]')") myLinearMomentum%vec3
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')")        myLinearMomentum%unt
  write(*,"(' and so with a unit of ',A32)")
  write(*,*) myLinearMomentum%dim

end program dim_check
```

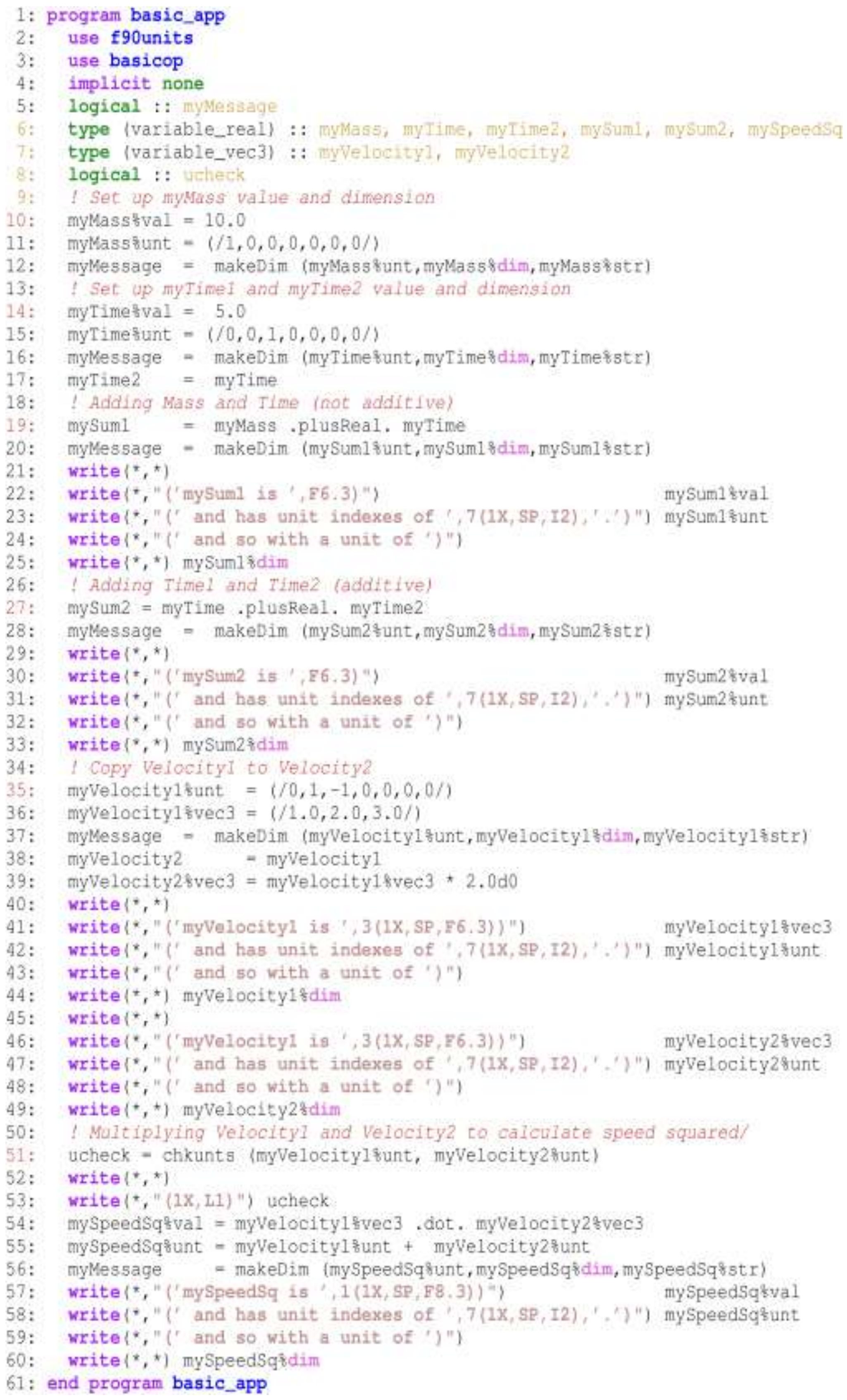
Figure 3. Contents of main program file, dimcheck.f90.

```
./dimcheck.exec

myMass is 10.000 kg
 and has unit indexes of  +1 +0 +0 +0 +0 +0 +0.
 and so with a unit of
  kg^(+1)   m^(+0)   s^(+0)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)

myVelocity is (  +1.00  +2.00  +3.00) [m/s]
 and has unit indexes of  +0 +1 -1 +0 +0 +0 +0.
 and so with a unit of
  kg^(+0)   m^(+1)   s^(-1)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)

myMass and myVelocity have different units: F

myLinearMomentum is ( +10.00 +20.00 +30.00) [kg m/s]
 and has unit indexes of  +1 +1 -1 +0 +0 +0 +0.
 and so with a unit of
  kg^(+1)   m^(+1)   s^(-1)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)
```

Figure 4. Contents of output message by execution of main program of Fig. 3.

```fortran
module basicop
  implicit none

  interface operator (.dot.)
     module procedure OP_DOT
  end interface operator (.dot.)

  interface operator (.plusReal.)
    module procedure plusReal
  end interface operator (.plusReal.)

contains

  function plusReal (A,B) result (C)
    use f90units
    implicit none
    type (variable_real), intent(in) :: A, B
    type (variable_real)             :: C, D
    D%unt = A%unt - B%unt
    IF( dot_product(D%unt, D%unt) == 0 ) then
       C%val = A%val + B%val
    else
       write(*,*) "Dimension error ... "
    end IF
  end function plusReal

  function dot_vectors (A,B) result (C)
    ! A dot B = C
    implicit none
    real, dimension(:), intent(in)    :: A, B
    real   :: C
    integer :: na, nb, i
    na = SIZE(A)
    nb = SIZE(B)
    if (na .eq. nb) then
       C = 0.0d0
       do i = 1, na
          C = C + A(i) * B(i)
       enddo
    else
    endif
  end function OP_DOT

end module basicop
```

Figure 5. Module basicop for basic operations with unit annotations.



```fortran
program basic_app
  use f90units
  use basicop
  implicit none
  logical :: myMessage
  type (variable_real) :: myMass, myTime, myTime2, mySum1, mySum2, mySpeedSq
  type (variable_vec3) :: myVelocity1, myVelocity2
  logical :: ucheck
  ! Set up myMass value and dimension
  myMass%val = 10.0
  myMass%unt = (/1,0,0,0,0,0,0/)
  myMessage  =  makeDim (myMass%unt,myMass%dim,myMass%str)
  ! Set up myTime1 and myTime2 value and dimension
  myTime%val =  5.0
  myTime%unt = (/0,0,1,0,0,0,0/)
  myMessage  =  makeDim (myTime%unt,myTime%dim,myTime%str)
  myTime2    =  myTime
  ! Adding Mass and Time (not additive)
  mySum1     =  myMass .plusReal. myTime
  myMessage  =  makeDim (mySum1%unt,mySum1%dim,mySum1%str)
  write(*,*)
  write(*,"('mySum1 is ',F6.3)")                            mySum1%val
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')") mySum1%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) mySum1%dim
  ! Adding Time1 and Time2 (additive)
  mySum2 = myTime .plusReal. myTime2
  myMessage  =  makeDim (mySum2%unt,mySum2%dim,mySum2%str)
  write(*,*)
  write(*,"('mySum2 is ',F6.3)")                            mySum2%val
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')") mySum2%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) mySum2%dim
  ! Copy Velocity1 to Velocity2
  myVelocity1%unt  = (/0,1,-1,0,0,0,0/)
  myVelocity1%vec3 = (/1.0,2.0,3.0/)
  myMessage  =  makeDim (myVelocity1%unt,myVelocity1%dim,myVelocity1%str)
  myVelocity2      = myVelocity1
  myVelocity2%vec3 = myVelocity1%vec3 * 2.0d0
  write(*,*)
  write(*,"('myVelocity1 is ',3(1X,SP,F6.3))")              myVelocity1%vec3
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')") myVelocity1%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) myVelocity1%dim
  write(*,*)
  write(*,"('myVelocity1 is ',3(1X,SP,F6.3))")              myVelocity2%vec3
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')") myVelocity2%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) myVelocity2%dim
  ! Multiplying Velocity1 and Velocity2 to calculate speed squared/
  ucheck = chkunts (myVelocity1%unt, myVelocity2%unt)
  write(*,*)
  write(*,"(1X,L1)") ucheck
  mySpeedSq%val = myVelocity1%vec3 .dot. myVelocity2%vec3
  mySpeedSq%unt = myVelocity1%unt +  myVelocity2%unt
  myMessage     = makeDim (mySpeedSq%unt,mySpeedSq%dim,mySpeedSq%str)
  write(*,"('mySpeedSq is ',1(1X,SP,F8.3))")                mySpeedSq%val
  write(*,"(' and has unit indexes of ',7(1X,SP,I2),'.')") mySpeedSq%unt
  write(*,"(' and so with a unit of ')")
  write(*,*) mySpeedSq%dim
end program basic_app
```

Figure 6. Module basicop for basic operations with unit annotations.

```
./basic_app.exec
  Dimensional error in .plusReal.

mySum1 is  0.000
 and has unit indexes of  +1 +6 ** +0 ** +0 **.
 and so with a unit of
  kg^(+1)   m^(+6)   s^(**)   K^(+0) mol^(**)   A^(+0)  cd^(**)

mySum2 is 10.000
 and has unit indexes of  +0 +0 +1 +0 +0 +0 +0.
 and so with a unit of
  kg^(+0)   m^(+0)   s^(+1)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)

myVelocity1 is  +1.000 +2.000 +3.000
 and has unit indexes of  +0 +1 -1 +0 +0 +0 +0.
 and so with a unit of
  kg^(+0)   m^(+1)   s^(-1)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)

myVelocity1 is  +2.000 +4.000 +6.000
 and has unit indexes of  +0 +1 -1 +0 +0 +0 +0.
 and so with a unit of
  kg^(+0)   m^(+1)   s^(-1)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)

 T
mySpeedSq is   +28.000
 and has unit indexes of  +0 +2 -2 +0 +0 +0 +0.
 and so with a unit of
  kg^(+0)   m^(+2)   s^(-2)   K^(+0) mol^(+0)   A^(+0)  cd^(+0)
```

Figure 7. Output of the main program of Fig. 6.