

A Design for Dimensional Analysis in Robotics

G.M. Biggs and B.A. MacDonald

Electrical & Computer Engineering, Private Bag 92019, University of Auckland, New Zealand
{g.biggs,b.macdonald}@auckland.ac.nz

Abstract

Robot programs typically manage a substantial amount of dimensioned data. However, existing robot programming tools do not directly support the description and manipulation of dimensioned quantities. A new system is presented for managing dimensioned data in robot software. The design requirements are described, along with a prototype design for object-oriented languages. An initial evaluation version is created by extending the Python interpreter. The dimensions and units commonly used in robot software are provided by default. The data type can be used in more complex structures such as paths and maps. The design removes much of the hard work of using and ensuring correctness in dimensional data and allowing mixing of units using dimensional analysis. The techniques can easily be integrated into existing programming systems.

1 Introduction

Robots have become increasingly complex and their controllers increasingly powerful, yet robotic programming tools have not advanced as rapidly [1]. Robots must be programmed both at the development stage, to create the functionality of the robot, and in the field, to customise the robot to applications, environments and tasks. It is important that robots become easier to program, so that their potential may be fully realised.

We believe robot programming systems must be targeted more closely to robotics, paying attention to the nature of typical robot programs, the typical skills of robot programmers, the interactions between humans and robots, and the programming constructs that prevail in robotic applications.

The C language, commonly used in low level robot programming and other embedded systems, suffers from a number of problems when evaluated for ease of programming in general, such as typing issues that limit the usefulness of libraries and admit program-

Table 1: Current methods for dealing with geometric data in some common robot programming systems.

<i>System</i>	<i>Units used</i>	<i>Representation</i>
Player [2]	Metres, radians	double
CARMEN [3]	Metres, radians	double or integer
Marie [4]	Millimetres, degrees	integer
Orca [5]	Metres, radians	double
Pyro [6]	Varies by robot	Python number
Kuka [7]	Varies by use	REAL type
ABB [8]	Varies by use	NUM type

mer errors, awkward manipulations of pointers, naming problems, and limitations of exception handling. Industrial robot languages are only suited for their specific problem domain and do not translate well into other areas of robotics such as mobile robots. Object-oriented programming languages provide more opportunities for code reuse and have become very popular. As robotic problems become more complex, and robot controllers become more powerful, we expect robot programming tools too will move further away from the embedded C style and increasingly adopt object-oriented programming styles.

There are several robot programming systems in use, as shown in Table 1. All these tools lack support for dimensioned data, which cannot be conveniently provided by an API. Robots operate in a physical environment rich in dimensioned data, including:

- Odometry data for tracking translation/rotation
- IR, sonar and laser scanner range data
- Motion control for mobile robots and limbs
- Geometric data for navigation systems
- Forces and torques for manipulations
- Geometric data in path plans and maps

A robot programming system should provide for dimensioned data ranging from fine control of primitive, atomic values such as metres, to the expressiveness needed for multidimensional poses and robot paths described in a configuration space.

Such a system should also support units, which give meaning to dimensioned data. Units support also

prevents inappropriate mixing of incompatible dimensioned values, for example distance and time.

Existing robot programming systems typically use fixed or floating point numeric types to represent dimensioned data, particularly geometric data, such as those in Table 1. Units are not specified, but an implied standard unit is assumed and the programmer is responsible for ensuring all necessary conversions.

It is not safe to rely on the programmer to ensure unit conversions, especially where different unit standards are used. In larger projects, the difficulties are compounded since many programmers are involved. It is difficult to change the implied standard units; the recent Player and Stage unit change from millimetres to metres generated much discussion and considerable reengineering of clients. Another, well-known incident was the failure of NASA’s Mars Climate Orbiter mission; the root cause was data with incorrect units [9].

To enhance programmability, dimensioned data in robot programs should appear and behave as programmers expect. Even when end users interact with robots in natural, human ways above the level of data description, robot developers and maintainers will still need to operate at the software level with dimensioned data.

The dimensional analysis technique can help make programs more robust against errors in units by verifying the consistency of the units given. It allows a new class of errors to be caught [10]. It can be applied by adding a new attribute to data types of the dimensions and units used, enabling verification of data compatibility and automatic conversions. Dimensional analysis systems have been proposed for Ada [11, 12, 13, 14], Pascal [10, 15, 16, 17], C++ [18, 19], and Java [20], among others, from 1977 to 2004. However, despite its potential in many common areas of programming and numerous proposals over the years, it has still not become a feature of the mainstream languages commonly used in robotics. Our work applies and extends some of these techniques for robotic programming systems.

The goal of this paper is to describe a possible design of dimensioned types for robot programming, suitable for extension with object-oriented techniques. It describes a low level, atomic data type, that can be extended to manage multi-dimensional data. While dimensional analysis has been a subject of research in generic programming language design, it has not been applied to robotic systems before.

An initial test prototype of the typing system has been implemented in Python to enable us to refine the design and explore improvements to the readability and syntax. While Python and other scripting languages may indeed be a suitable for robotic and other

real time programming applications (see for example [21] and the Python driver for Player), our purpose is simply to show a test implementation of the semantics, which should also be implementable in other object-oriented languages such as C++.

2 Design Requirements for a Dimensioned Data Type

An atomic item of dimensioned data has two aspects; a value and a unit of measure. The two must be inseparably bound together, both in the underlying representation, to ensure consistency, and in the visible program code, to ensure code clarity. The data type must be flexible, allowing programmers to define their own dimensions and units. It is not possible to anticipate all units that all programmers will require.

The units must be applicable to multiple numerical representations of values. In robotics, floating point numbers provide a large range of values such as laser data and outdoor environments, while fixed point numbers provide guaranteed accuracy, such as in grid-based representations.

It must also be possible to limit and wrap the range of values. For example, to wrap an angle to between zero and 360 degrees.

A key part of dimensional analysis is unit algebra: the interaction of the units when dimensioned values are combined. It prevents incompatible conversions (e.g. seconds to millimetres) while correctly transforming appropriate combinations (e.g. metres divided by seconds gives a value in metres per second). Unit algebra must be active for all dimensioned values.

3 Semantics and Design

The first issue of design is, where to store the unit information? The unit could be an attribute of the data or another part of the data’s value. Gehani proposed that units be represented as an additional attribute, separate from the type, as the unit has no bearing on the numerical representation [10]. However this affects every variable and expression in that all variables will carry a dimensional component no matter the type or the intent of the programmer.

Recent work using object-oriented concepts has shown that the unit may be more a part of the value [18, 19, 20]. Rather than changing the language itself, a separate type is provided for dimensioned data, using object abstraction. Often this requires some checking at run time as well as compile

time, potentially reducing efficiency. However, with modern computers this is less of a problem.

We chose to consider the unit as part of the data's value, albeit a part that helps define the numerical representation. This allows for run time use of units; for example units are also provided along with values in input and output operations. This is not possible in systems that use the unit information only at compile time, removing it from the final program.

We wanted to design a separate simple dimensioned object. However, object abstraction gains no special lexical representation to improve clarity. We have therefore decided to embed our dimensioned object as a new primitive data type, one with more information than common standard numerical types. This maintains compatibility with the unmodified language, maintaining its generality, while still gaining the clarity of a special syntax for dimensioned data.

Another aspect is the declaration of dimensions and units. Gehani's original proposal declared units upon use. House objected because the system had no knowledge of the units, and also it allowed the unit names to be reused elsewhere, for example as variable names, potentially confusing programmers [22]. House's system instead requires that units be declared before use, but allows declarations inside limited scopes. We go one step further, requiring all dimensions and units to be declared before use and not allowing limited scopes for unit declarations. Once a dimension or unit is declared it exists until the program exits. This ensures that a dimension or unit is the same thing wherever it is in the program, avoiding programmer confusion.

Unlike some other implementations of dimensional analysis, we do not limit dimensions to those in the SI system. A dimension is just a category used to discern compatible units. Programmers can declare any number of dimensions, and four dimensions commonly found in robotics are used as the defaults: distance, angle, time and mass, with several default units.

We do not allow aliases for units, for example "speed" for metres per second. This maintains both simplicity and clarity, preventing confusion for the developer about meanings of aliases.

Finally, there is the numerical representation of the data. Units can be declared to be either fixed point or floating point units (see section 2). The precision of a value is determined by its unit. This ensures that all values using a certain unit have the same precision. For multi-unit expressions such as metres per second, the precision is determined by the expression. If one unit in the expression is floating point then the whole unit expression is considered to be. This minimises

data loss in the simplest way possible and ensures consistent behaviour.

As both dimensions and units need to be specified before being used, there are some options to be set when defining them, to provide some of the functionality described in section 2. A dimension simply requires a name for reasons described earlier. The options for configuring a unit are:

- Dimension of the unit
- Name of the unit
- Precision (floating point or fixed point)
- Range limits (inclusive, exclusive or wrapped)
- A conversion ratio giving the unit size relative to others in the dimension

The conversion ratio is taken against units in the dimension defined with a conversion ratio of one. For example, in the "distance" dimension "metres" may have a ratio of one, and "kilometres" a ratio of 1000.

The *operators* available for the primitive dimensioned types are illustrated in Table 2:

Addition/subtraction: two dimensioned values can be added or subtracted if they both have equivalent dimensionality. For example, m/s and mm/hour are compatible. When the units are different (for example, metres and millimetres), the right operand is automatically converted to the left operand's units. The result is always in the left operand's units. It is not possible to add or subtract a dimensioned value and a non-dimensioned value.

Multiplication/division: multiplication and division introduce unit algebra. When two dimensioned values are multiplied or divided, their unit expressions must also be multiplied or divided to give the new unit expression. When two values in metres are multiplied together, the result is in metres². When dividing a distance value by a time value, such as metres by seconds, the result is clearly metres/second. Thus the units commonly used in robotics are provided automatically, given only predefined basic units. If two dimensionally identical unit expressions are divided, for example mm and m, the result is a scalar.

Comparisons: comparisons are performed on the physical value rather than the stored value. So, for example, 5mm < 2m would return true, while 6m/s < 15mm/hour would return false.

4 Implementation

We have created a custom version of the Python interpreter, named RADAR, to implement our new data

Table 2: Examples of operations on dimensioned data.

Operator	Example	Result
Addition	5m + 2cm	5.02m
	5m + 0.02	Error
Subtraction	5m - 2cm	4.98m
	5m - 2rad	Error
Multiplication	5m × 2	10m
	5m × 2m	10m ²
Division	5m ÷ 2	2.5m
	5m ÷ 2rad	Error
	5m ÷ 2m	2.5
Comparison	2m < 3cm	False
	2m > 3cm	True

type. The primitive dimensioned type has been implemented directly in the Python interpreter as a built-in type at the same level as integers. This adds new syntax for representing dimensioned values directly in program code. A dimensioned value is specified by:¹

```
dimensioned ::= (integer | float)~unit
              ((~* | ~/) unit)*
unit ::= [['~'] digit+] letter+ ['~' ['~'] digit+]
```

Table 3 shows the attributes of a dimensioned value, and some examples.

The Python interpreter consists of a tokenizer, a parser, a compiler and a code evaluator. To implement the dimensioned data type, changes were made to the tokenizer, to the grammar tree used by the parser, and to the compiler. The new tokens required by the type, specifically the “~” token and the “units” token, had to be added. Similarly, the grammar tree used by the parser had to be altered to add the new grammar shown above. This allows the correct nodes to be created in the parse tree used by the compiler. The compiler was altered to add support for these nodes. When a dimensioned data node is encountered in the parse tree, the compiler parses the subsequent nodes as appropriate and creates a dimensioned data type object.

Dimensioned values can also be created with the built-in `dimensioned()` function. The value is stored internally as either floating point (a C double) or integer (a C long), depending on the unit expression used (see section 3).

The unit for each dimensioned value is stored in a Python list of tuples. Each tuple contains a unit name, a multiplier and a power. For example, the unit expression m/s^2 is stored as $((1, 'm', 1), (1, 's', -2))$.

A built-in module `dimension` provides the functionality for managing dimensions and units. This neatly

¹The “~” prevents a conflict with existing Python syntax for specifying long integers and complex numbers.

Table 3: Attributes of a dimensioned value, and some examples.

Attribute	Examples		
Value	43.0	5	1.5
Unit	cm	cell	radians
Representation	float	fixed	float
Range	-10 to 120	0 to 10	0 to 2π
Code	43~cm	5~cell	1.5~rad

encapsulates the functions used separately from the standard namespace. The module includes functions for defining new dimensions and units and for getting the currently defined dimensions and units. The module stores the defined dimension and unit data in two linked lists of C structures.

During compilation, the compiler makes calls to the functions contained in this module to confirm that the requested units have been defined. If they have not, it raises an exception.

For example, a new dimension, “fasterthanlight,” can be added to the system and the unit “warp” defined with range limits of 0 to 9.9 and floating point precision:

```
import dimension
dimension.DefineDimension ('fasterthanlight')
dimension.DefineUnit ( 'fasterthanlight',
                       'warp',
                       UnitPrec_Floating,
                       1.0,
                       UnitRange_Exclude, 0,
                       UnitRange_Include, 9.9 )
```

To simplify units for robot programmers, defaults provide commonly used dimensions and units.

5 Integration with Player

Robot APIs must be able to extract and check the unit and dimension data used in robot programmes. As a first step towards this goal, a thin wrapper has been created around the Python bindings for the Player API. The wrapper does the unit checking necessary to ensure consistency of the data before it is passed into the API itself, when a function is called. Secondly the wrapper converts raw data into dimensioned data when retrieving values from the API. There is, as expected, a minor loss in efficiency in having to “flatten” the dimensioned data and recreate it when converting between program code and the API.

An entirely new Player client has also been written that directly supports the use of dimensioned data. Both this client and the wrapper are available for download, with the custom interpreter, at [23].

```

1 if motorsOn:
2     position.set_cmd_vel(0, 0, 0, 1)
3     position.enable(1)
4 else:
5     position.set_cmd_vel(0, 0, 0, 0)
6     position.enable(0)
7
8 newSpeed = 0
9 newTurnRate = 0
10 while 1:
11     robot.read()
12
13     if len (laser.Ranges()) == 0:
14         print 'No laser data'
15         continue
16     leftRanges = laser.Ranges()[len(laser.Ranges())/2]
17     rightRanges = laser.Ranges()[len(laser.Ranges())/2:]
18     minLeft = min(leftRanges)
19     minRight = min(rightRanges)
20     l = (minLeft * 10) / 5.0 - 1
21     r = (minRight * 10) / 5.0 - 1
22     if l > 1:
23         l = 1
24     if r > 1:
25         r = 1
26     newSpeed = (r + 1) / 10.0
27     newTurnRate = (r - 1) * 100.0
28     newTurnRate = min (newTurnRate, 40)
29     newTurnRate = max (newTurnRate, -40)
30     newTurnRate = math.radians (newTurnRate)
31     position.set_cmd_vel(newSpeed, 0, newTurnRate, 1)

```

(a) Using standard Python.

```

if motorsOn:
    position.set_cmd_vel(0~m/s, 0~m/s, 0~rad/s, 1)
    position.enable(1)
else:
    position.set_cmd_vel(0~m/s, 0~m/s, 0~rad/s, 0)
    position.enable(0)

newSpeed = 0~m/s
newTurnRate = 0~rad/s
while 1:
    robot.read()

    if len (laser.Ranges()) == 0:
        print 'No laser data'
        continue
    leftRanges = laser.Ranges()[len(laser.Ranges())/2]
    rightRanges = laser.Ranges()[len(laser.Ranges())/2:]
    minLeft = min(leftRanges)
    minRight = min(rightRanges)
    l = (minLeft * 10) / 5 - 1~m
    r = (minRight * 10) / 5 - 1~m
    if l > 1~m:
        l = 1~m
    if r > 1~m:
        r = 1~m
    newSpeed = ((r + 1) / 1~s) / 10
    newTurnRate = ((r - 1) / 1~m) * 100~deg/s
    newTurnRate = min (newTurnRate, 40~deg/s)
    newTurnRate = max (newTurnRate, -40~deg/s)

    position.set_cmd_vel(newSpeed, 0~m/s, newTurnRate, 1)

```

(b) Using our extended Python.

Figure 1: The Player laser obstacle avoidance example.

6 Examples

Three examples from the Player project have been ported from the original C code to our extended Python: the sonar and laser obstacle avoidance examples and the random walk example. The original C source for these examples can be found at [24]. The relevant parts of the laser example are shown in Figure 1. The full code for all examples can be found at [23].

The example code shows that the addition of a special syntax for dimensioned data makes it immediately obvious which variables and values correspond to real world values. One thing that is not immediately obvious is the automatic conversion of units. While the turn rates are specified in degrees per second, the Player client is expecting them in radians per second, and values are converted where necessary.

Also worth noting is that the number of lines of code has not significantly decreased. The benefit of the dimensioned data type is not in significantly shorter code but in shorter programming and debugging time and code that is easier to maintain.

When writing these examples we observed that it

required some effort to change our mindset from the common method of using plain numbers for real world data to thinking in terms of dimensioned data and manipulating dimensions directly in the language syntax. However, once we had made this transition programming became significantly easier and it became more natural to work with the dimensionality of the data.

The primitive dimensioned data type provides a powerful way to manage dimensioned data in robotic systems. Some potential uses are representing robot poses and representing paths.

A tuple of dimensioned values can be used to represent a robot's pose in a configuration space. A list of these could represent a path for a robot to follow. This concept can be extended further by including a value in the time dimension, producing a robot pose in time. By making a list of these a motion plan for the robot is produced. The plan can be executed by simply iterating over the list, using interpolation based on the current time to find the required action between poses. Other dimensional data could also be added, for example speeds to be attained at each waypoint.

7 Conclusions

A new system is presented for managing dimensioned data in robot software. The design requirements have been described as well as a test implementation using the Python language. The semantics have been designed with a view to implementation in any modern language with object-oriented concepts.

Support is provided for user-defined dimensions and units, allowing the system to be extended quickly and easily as needed by the programmer. Enforcing units is a simple task. Program clarity is improved by making dimensioned data explicit and visible in the code through the use of a special syntax. More complex structures can be created, such as lines, paths and maps. An interface has been created to allow the system to be used with the Player robot API.

The benefits of this dimensioned data system are that it greatly simplifies the management of dimensioned data within programs. It removes much of the programmer's hard work of using and ensuring correctness in dimensioned data. Furthermore, it can easily be integrated into existing programming systems. The system is available for download at [23].

References

- [1] G. Biggs and B. MacDonald, "A survey of robot programming systems," in *Proceedings of the Australasian Conference on Robotics and Automation*, CSIRO, Brisbane, Australia, December 1–3 2003, <http://www.araa.asn.au/acra/acra2003/papers/27.pdf>.
- [2] R. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS03)*, vol. 3, Las Vegas, Nevada, October 2003, pp. 2421–2427.
- [3] "Carnegie Mellon Robot Navigation Toolkit (CARMEN)," <http://www-2.cs.cmu.edu/~carmen/>, 2005.
- [4] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raïevsky, M. Lemay, and V. Tran, "Code reusability tools for programming mobile robots," in *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, Sendai, Japan, October 2004, pp. 1820–1825.
- [5] "ORCA," <http://orca-robotics.sourceforge.net>, 2005.
- [6] D. Blank, L. Meeden, and D. Kumar, "Python robotics: An environment for exploring robotics beyond legos," in *Proceedings of the Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education*. Reno, Nevada: ACM Press, February 2003.
- [7] "KUKA Automatisering + Robots N.V." <http://www.kuka.be/>, June 2005.
- [8] "The ABB group," <http://www.abb.com/>, June 2005.
- [9] A. G. Stephenson and group, "Mars Climate Orbiter Mishap Investigation Board Phase I Report," ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, November 10 1999.
- [10] N. Gehani, "Units of measure as a data attribute," *Computer Languages*, vol. 2, no. 3, pp. 93–111, 1977.
- [11] N. H. Gehani, "Ada's derived types and units of measure," *Softw. Pract. Exper.*, vol. 15, no. 6, pp. 555–569, 1985.
- [12] P. N. Hilfinger, "An Ada package for dimensional analysis," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 189–203, 1988.
- [13] P. Rogers, "Dimensional analysis in Ada," *Ada Lett.*, vol. VIII, no. 5, pp. 92–100, 1988.
- [14] D. W. Gonzalez and T. Peart, "Applying dimensional analysis," *Ada Lett.*, vol. XIII, no. 4, pp. 77–86, 1993.
- [15] M. B. Agrawal and V. K. Garg, "Dimensional analysis in Pascal," *SIGPLAN Not.*, vol. 19, no. 3, pp. 7–11, 1984.
- [16] A. Dreiheller, B. Mohr, and M. Moerschbacher, "Programming Pascal with physical units," *SIGPLAN Not.*, vol. 21, no. 12, pp. 114–123, 1986.
- [17] G. Baldwin, "Implementation of physical units," *SIGPLAN Not.*, vol. 22, no. 8, pp. 45–50, 1987.
- [18] R. Cmelik and N. Gehani, "Dimensional analysis with C++," *Software, IEEE*, vol. 5, no. 3, pp. 21–27, 1988.
- [19] Z. D. Umrigar, "Fully static dimensional analysis with C++," *SIGPLAN Not.*, vol. 29, no. 9, pp. 135–139, 1994.
- [20] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele, Jr., "Object-oriented units of measurement," in *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2004, pp. 384–403.
- [21] RTSJ, "RTSJ: Real Time Specification for Java," <https://rtsj.dev.java.net/>, June 2005.
- [22] R. T. House, "A proposal for an extended form of type checking of expressions," *Comput. J.*, vol. 26, no. 4, pp. 366–374, 1983.
- [23] G. Biggs, "Robot Programming Systems: Homepage of Geoff Biggs," <http://www.ece.auckland.ac.nz/~gbig005/>, 2005.
- [24] "The Player/Stage Project," <http://playerstage.sourceforge.net/>, 2005.