Scripting Guide

Version 11.0, Document Number: HERSCHEL-HSC-DOC-0517
10 April 2017

Scripting Guide

Table of Contents

Preface xvii
1. Scripting and Jython basics 1
    1.1. Getting started 1
        1.1.1. Why scripting with HIPE? 1
        1.1.2. Testing commands 1
        1.1.3. Writing your first script 1
        1.1.4. Running your first script 2
        1.1.5. Where to go from here? 2
    1.2. Jython, Python and Java 2
    1.3. Writing commands interactively 3
    1.4. Writing a script 4
    1.5. Variables and variable types 4
        1.5.1. More on complex numbers 5
        1.5.2. Java variable types 5
        1.5.3. The range of Jython numeric types 6
        1.5.4. Other variable types 7
    1.6. Getting help on variables 7
    1.7. Defining and modifying strings 8
        1.7.1. Java string types 9
    1.8. Formatting strings 9
    1.9. Converting between variable types 10
        1.9.1. Converting between Java and Jython types 11
    1.10. Lists, dictionaries and tuples 12
    1.11. Creating and modifying lists 12
    1.12. Concatenating lists and tuples 13
    1.13. Accessing lists and tuples 13
    1.14. Creating and modifying dictionaries 14
    1.15. Accessing dictionaries 14
    1.16. Nesting dictionaries 15
    1.17. Code blocks 15
    1.18. Writing branching code: if/elif/else 16
    1.19. Writing loops: for and while 16
    1.20. Controlling loops: break and continue 17
    1.21. Writing loops in the Console view 18
    1.22. Printing to the screen 18
    1.23. Writing strings to file 19
    1.24. Reading strings from file 19
    1.25. Writing numeric values to file 20
    1.26. Reading numeric values from file 20
    1.27. Functions 20
    1.28. Executing HIPE tasks from your scripts 22
    1.29. Classes 23
    1.30. Creating and using classes 23
        1.30.1. Printing objects 24
    1.31. Naming conventions for classes and variables 25
    1.32. Creating aliases for class and function names 25
    1.33. Importing modules 25
        1.33.1. Importing, reloading and unimporting your own modules 27
    1.34. Understanding pipeline scripts 29
    1.35. Accessing files and directories 29
    1.36. Adding simple dialogue windows 30
        1.36.1. Dialogue box with message 30
        1.36.2. Dialogue box with text input field 31
        1.36.3. Dialogue box asking yes/no question 32
    1.37. Pausing and debugging scripts 33

1.38. Interoperating with external software ... 33
1.39. Developing version-aware scripts ... 34
1.40. Sharing scripts ... 35
1.41. IDL to HIPE command mapping ... 36
1.41.1. Idl to Jython mapping ... 36
2. Arrays, datasets and products ... 40
2.1. HIPE-specific data structures ... 40
2.2. Numeric arrays ... 40
2.2.1. Creating an array ... 41
2.2.2. Inspecting an array ... 41
2.2.3. Inspecting a complex array ... 42
2.2.4. Modifying an array ... 42
2.2.5. Ordering of array elements ... 43
2.2.6. Numeric array arithmetic ... 44
2.2.7. Selecting and filtering array values ... 44
2.2.8. Using logical operators with arrays ... 46
2.2.9. Removing infinite and NaN values from arrays ... 47
2.2.10. Advanced tips for improved performance ... 47
2.2.11. Type conversions ... 48
2.3. Array datasets ... 49
2.3.1. Creating an array dataset ... 49
2.3.2. Modifying an array dataset ... 50
2.3.3. Inspecting an array dataset ... 50
2.4. Table datasets ... 50
2.4.1. Creating a table dataset ... 50
2.4.2. Modifying a table dataset ... 51
2.4.3. Copying a table dataset into another ... 52
2.4.4. Inspecting a table dataset ... 52
2.5. Composite datasets ... 52
2.5.1. Creating a composite dataset ... 52
2.5.2. Modifying a composite dataset ... 53
2.5.3. Inspecting a composite dataset ... 53
2.6. Measurement units ... 53
2.6.1. Creating and assigning units ... 54
2.6.2. Obtaining derived units ... 55
2.6.3. Converting units to and from strings ... 55
2.6.4. Converting units to other units ... 56
2.6.5. Comparing units for compatibility ... 56
2.6.6. Comparing units for equivalence ... 56
2.6.7. Obtaining physical and mathematical constants ... 57
2.7. Metadata ... 57
2.7.1. Modifying metadata ... 57
2.7.2. Inspecting metadata ... 58
2.8. Products ... 58
2.8.1. Creating a product ... 58
2.8.2. Modifying a product ... 59
2.8.3. Setting date and time in product metadata ... 59
2.8.4. Inspecting a product ... 60
2.8.5. Product contexts ... 60
2.8.6. Observation contexts ... 60
2.8.7. Product history ... 60
3. Spectra and spectral cubes ... 62
3.1. Spectrum containers and segments ... 62
3.2. Spectrum1d ... 62
3.2.1. Creating a Spectrum1d ... 63
3.2.2. Accessing data from a Spectrum1d ... 64
3.3. Spectrum2d ... 66
3.3.1. Creating a Spectrum2d ... 67

3.3.2. Accessing data from a Spectrum2d 68
3.4. SimpleSpectrum 69
3.5. SpectralSimpleCube 70
3.5.1. Creating a SpectralSimpleCube 70
3.5.2. Accessing data from a SpectralSimpleCube 70
3.6. Instrument-specific spectral products 72
4. The World Coordinate System 74
4.1. Assigning a World Coordinate System to images and cubes 74
4.2. Correcting the astrometry of your data 77
5. The Numeric library 79
5.1. Numeric functions and lambda expressions 79
5.2. Basic functions 80
5.3. Integral transforms 81
5.3.1. FFT 82
5.3.2. FFT_PACK 82
5.3.3. Selecting the right Fourier transform 84
5.3.4. Inverse Fourier transforms 84
5.3.5. Normalization 85
5.4. Power spectrum 86
5.5. Convolution 87
5.6. Boxcar and Gaussian filters 88
5.7. Interpolation 88
5.8. Fitting data 89
5.8.1. General approach 89
5.8.2. Available linear models 91
5.8.3. Available non-linear models 92
5.8.4. Compound and mixed models 93
5.8.5. Available fitters 93
5.8.6. Setting the fitter tolerance 94
5.8.7. 1D fit example 95
5.8.8. 2D fit example 96
5.8.9. Additional documentation 97
5.9. Masks 97
5.10. Matrices 97
5.11. Random numbers 100
5.12. Numeric integration 101
5.12.1. Integrating functions 101
5.12.2. Integrating discrete values 102
5.13. Interpolating discrete data 103
5.14. Statistics 104
5.15. Wavelet transforms 104
5.15.1. Continuous wavelet transform 105
5.15.2. Example 105
5.15.3. Modulo Maxima Line 106
5.15.4. The wavelet library 106
5.15.5. Discrete wavelet transform 107
5.15.6. Stationary wavelet transform 109
5.15.7. Tools 110
5.15.8. Wavelet toolbox overview 111
6. Running tasks 113
6.1. Running a task 113
6.2. Task parameters 114
6.2.1. Output parameters 114
7. Storing and accessing data products 116
7.1. Pools and storages 116
7.1.1. Creating a storage and registering pools 116
7.1.2. Saving and loading products 117
7.1.3. Deleting products 117

7.1.4. Tagging products ... 117
7.2. Local pools ... 118
7.2.1. The local pool directory ... 118
7.2.2. Repairing a local pool ... 119
7.2.3. Importing a directory of FITS files into a local pool ... 119
7.2.4. Troubleshooting ... 119
7.3. Querying ... 119
7.3.1. Inspecting query results ... 120
7.4. Product versioning ... 121
7.4.1. Querying product versions ... 121
7.5. Advanced querying ... 121
7.5.1. Querying for parts of a string ... 123
7.5.2. Querying for metadata in products ... 123
7.6. Tips and pitfalls ... 123
7.6.1. Changes to a product in a pool disappear ... 124
7.6.2. Minimising memory usage ... 124
7.6.3. Testing if two products are equal ... 124
7.6.4. Copying a product or context to a different storage ... 125
7.6.5. Tags may point to wrong product after renaming a pool ... 125
7.6.6. IndexError or IllegalArgumentException when querying ... 126
7.6.7. A query takes a long time to execute ... 126
7.7. Pools for remote data ... 127
7.7.1. The HSA pool ... 127
7.7.2. The HTTP pool ... 128
7.7.3. The cached pool ... 128
7.7.4. Metadata used in the HSA pool ... 129
8. Overview of data processing packages ... 134
8.1. Browsing the list of packages ... 135
8.2. Browsing the contents of a package ... 136
8.3. Viewing the details for a class or interface ... 137
8.4. Displaying alternative views of the Developer's Reference Manual ... 139
8.5. DP packages ... 139
8.5.1. herschel.ia.dataflow ... 139
8.5.2. herschel.ia.dataset ... 139
8.5.3. herschel.ia.document ... 140
8.5.4. herschel.ia.gui ... 140
8.5.5. herschel.ia.io ... 140
8.5.6. herschel.ia.numeric ... 140
8.5.7. herschel.ia.obs ... 141
8.5.8. herschel.ia.pal ... 141
8.5.9. herschel.ia.pg ... 142
8.5.10. herschel.ia.qcp ... 142
8.5.11. herschel.ia.spg ... 142
8.5.12. herschel.ia.task ... 142
8.5.13. herschel.ia.toolbox ... 143
8.5.14. herschel.ia.vo ... 143
8.5.15. herschel.share.fltdyn ... 144
9. Time and astronomical measurements ... 145
9.1. Time Definitions ... 145
9.1.1. System time in HIPE ... 145
9.1.2. International Atomic Time (TAI) and `FineTime` ... 146
9.1.3. Coordinated Universal Time (UTC) ... 146
9.1.4. DecMec Time [PACS only] ... 146
9.2. Time in Instrument House-Keeping (HK) Data ... 147
9.3. Time conversion ... 147
9.3.1. Time conversion in HIPE ... 147
9.3.2. CucConverter ... 148
9.4. Great circle and position angle calculations ... 148

A. Jython operators 150
B. Naming conventions 152
B.1. Naming Conventions 152
B.1.1. Jython code example 154
B.1.2. Java code example 155
Index 156

List of Figures

1.1. The window that appears calling the Swing `showMessageDialog` method. 30
1.2. Customising the icon and the window title. 30
1.3. The window that appears calling the Swing \`showInputDialog` method. 31
1.4. A more complex window with a combo box. 31
1.5. Using the Swing `showConfirmDialog` method. 32
1.6. The Debug window 33
3.1. An example of a Spectrum1d product from SPIRE (in the HIPE Dataset Viewer). In this case, the wave column contains wavenumbers in cm^{-1}, there is no segment number column (as only one segment is contained) and there are additional columns for error and mask. 63
3.2. Example of a HIFI Spectrum2d viewed in the SpectrumExplorer in HIPE. Different spectra appear as different rows, and in this case each spectrum has 4 subbands. Each subband is plotted in the colour shown in the boxes on the lower left. The other columns give further information about each spectrum. 66
3.3. Example of a SPIRE Spectrum2d viewed in the SpectrumExplorer in HIPE. Different spectra appear as different rows, but in this case each spectrum only has one subband. There are fewer columns for additional information than in the HIFI example in Figure 3.2. 66
3.4. Example of a HIFI Spectrum2d viewed in the SpectrumExplorer in HIPE, showing the metadata describing the different subbands (to display the metadata, right click in the plot and select Dialogs - Metadata). In this dataset, there is one spectrum with 4 subbands. 67
5.1. Execution times for FFT, FFT_PACK and RealDoubleFFT. 84
5.2. Illustration of various forms of interpolation functions. 89
5.3. Fitting data iteratively with tolerance set too high. 94
5.4. Illustration of polynomial fit. 96
5.5. Effects of modulo maxima line. 106
5.6. Principles of discrete wavelet transform. 107
5.7. Signal decomposed: Russian dolls view. 108
5.8. Formula of universal threshold. 110
8.1. View of SPIRE packages after opening up and clicking on SPIRE Developer's Reference Manual (API) in the HIPE help window. 135
8.2. Web browser page of JavaDocs top level frame. 136
8.3. Navigation bar on the *class view* of JavaDocs. 136
8.4. Package description page in Developer's Reference Manual. 137
8.5. The class view of TableDataset showing a brief description and a short example of its usage. 138
8.6. Page showing the constructor mechanism (how to create a TableDataset) and the associated set of methods (what you can do with the TableDataset you created). 138

List of Tables

1.1. Conversion types for string formatting. 10
2.1. Types of numeric array (N = 1...5) 40
3.1. Spectrum1d columns and access 64
3.2. Spectrum2d columns and access 68
3.3. SpectralSimpleCube content and access 70
3.4. Instrument-specific spectral products 72
5.1. Forward Fourier transforms for input of length N. 81
5.2. Options for the inverse Fourier Transforms. Note that the output of `RealDoubleFFT#bt` depends on the value of N with which the `RealDoubleFFT` object was created. 85
5.3. For the following normalizations, assume that the signal has N elements. 85
5.4. Algorithms 111
5.5. Tools 111
5.6. Signal dimensions 111
5.7. Border management 111
5.8. Supported signal types 111
5.9. Available wavelets 112
A.1. Jython unary arithmetic operators 150
A.2. Jython binary arithmetic operators 150
A.3. Jython shifting operators 150
A.4. Jython binary bitwise operators 151
A.5. Jython comparison operators 151
A.6. Jython boolean operators 151

List of Examples

1.1. Creating a simple image with fake data. 1
1.2. Assigning multiple values to multiple variables in one line. 4
1.3. Assigning the same value to several variables at once. 4
1.4. Deleting variables. 4
1.5. Deleting all variables. 5
1.6. Printing the type (class) of a variable. 5
1.7. How to represent complex numbers in Jython. 5
1.8. Printing complex numbers to the console. 5
1.9. Printing numbers (with default formatting) to the console. 6
1.10. Comparing variables from Java classes is different to Jython comparisons. 6
1.11. Checking the range limits of types. 6
1.12. Demonstrating the ranges for built-in types in Jython. 7
1.13. Printing the methods available for a variable in the console. 8
1.14. Concatenating strings with + in Jython does not have the performance impact it has in Java. 8
1.15. Converting numbers to string using the builtin `backquote` method. 9
1.16. Creating a Java string from a Jython string. 9
1.17. Creating Java Character variables. 9
1.18. Jython and Java string incompatibilities. 9
1.19. Printing text to the console with variable substitution. 9
1.20. How to change the floating-point precision when printing numbers. 10
1.21. Converting between Jython types. 10
1.22. Complex to float conversion is impossible. 10
1.23. String to float conversion. 11
1.24. Decimal to integer conversion is impossible. 11
1.25. Implicit Java to Jython numeric conversion. 11
1.26. Printing the title of a variable. 11
1.27. Printing the class of a Java variable. 11
1.28. Another example of Java and Jython numeric type incompatibilities. 12
1.29. Converting the Java numeric variable to a Jython value. 12
1.30. the Jython type to Java will not work. 12
1.31. Defining dictionaries. 12
1.32. Nesting lists does not require that all lists are of the same type. 13
1.33. Appending values to the end of a list, using different methods. 13
1.34. Appending several values with the help of a for loop. 13
1.35. Concatenating two lists or tuples. 13
1.36. Concatenating a list and a tuple will not work. 13
1.37. Accessing elements or ranges using list slice notation. 13
1.38. Accessing elements of nested lists or tuples. 14
1.39. Defining a dictionary. 14
1.40. Accessing a dictionary value using a key. 14
1.41. Accessing and printing a dictionary value using a key. 14
1.42. Printing all the keys of a dictionary. 15
1.43. Using a dictionary as a value of another dictionary. 15
1.44. Accessing a dictionary value is the same as any other value. 15
1.45. Accessing nested dictionaries is the same as with multidimensional arrays/lists. 15
1.46. Checking if a number belongs to several ranges using comparison operators. 16
1.47. Basic loop printing the index values. 16
1.48. Constructing loops with the help of the range function. 16
1.49. Adding an else code block to a for loop that is executed when the loop finishes. 17
1.50. Exiting the loop with break will not execute the else block. 17
1.51. Branching and looping structures in Jython. 17
1.52. Writing a while loop block. 17
1.53. How to break out from a while loop. 18
1.54. Exiting an infinite loop with the use of the continue keyword. 18

1.55. Using the range utility function to generate index values. 19
1.56. Printing content from dictionaries. 19
1.57. Printing text to file. 19
1.58. Reading text from files. 19
1.59. How to use the well-known pickle library. 20
1.60. Load data from file serialised using the pickle library. 20
1.61. Defining functions in Jython. 20
1.62. Declaring functions for reuse. 21
1.63. Global variables in Jython. 21
1.64. Declaring a global variable can be dangerous in an interpreted language like Jython. 21
1.65. Passing global variables as function arguments. 21
1.66. Setting default values for function arguments. 22
1.67. Passing functions as arguments is allowed in Jython. 22
1.68. Declaring a function without arguments. 22
1.69. How to create a Jython class. 23
1.70. Instantiating and using methods from a class. 24
1.71. Passing parameters to the class constructor. 24
1.72. Overriding the default behaviour of the __str__ method. 24
1.73. Naming conventions for objects. 25
1.74. Creating aliases. 25
1.75. Instantiating/aliasing an imported task. 26
1.76. Executing methods from non-imported modules. 26
1.77. Import statement. 26
1.78. Using names after importing a whole module. 26
1.79. 26
1.80. Importing several names from a module. 26
1.81. Aliasing imports to avoid name clashes. 27
1.82. Self-contained module with documentation. 27
1.83. Demonstrating that modules imported at start-up are available in HIPE. 28
1.84. Creating a file with just imports for HIPE start-up. 28
1.85. How to modify the Jython classpath at runtime. 28
1.86. Reloading updates the module loaded in memory with the latest changes from the source files. 28
1.87. Removing a module from memory using del. 28
1.88. How to list the contents of the current directory. 29
1.89. Using the glob module to recursively list files with wildcard matching. 29
1.90. Using pure Java to display dialogues to the user. 30
1.91. Configuring the style of Java dialogues. 30
1.92. Using input dialogues to retrieve data from the user. 31
1.93. Customising the style of input dialogues. 31
1.94. Providing a default value for input dialogues. 31
1.95. Using lists to restrict the values to use in an input dialogue. 31
1.96. Displaying a confirmation dialogue. 32
1.97. Using constants instead of the automatic indexes to improve readability. 32
1.98. Adding title and button type to the confirmation dialogue. 32
1.99. Pausing a script to allow debugging or printing text to console. 33
1.100. Outline on how to create a pipeline executing tasks sequentially. 33
1.101. Executing platform binaries from within HIPE. 34
1.102. Importing the Jython module that allows communication with the operating system. 34
1.103. Listing the contents of a directory. 34
1.104. Mixed processing with external commands and HIPE tasks. 34
1.105. Getting the version number of a user release (method 1). 35
1.106. Getting the version number of a user release (method 2). 35
1.107. Getting the build number of a developer build. 35
2.1. Declaring an array of doubles. 41
2.2. Declaring a two-dimensional array of doubles. 41
2.3. Creating Jython jagged arrays. 41
2.4. It is impossible to create Numeric jagged arrays. 41

2.5. Accessing array elements using the indices. 41
2.6. Using array slices to access ranges from Jython lists. 42
2.7. Accessing ranges of indices using slices. 42
2.8. Checking the differences between Jython arrays and numeric arrays. 42
2.9. Inspecting and manipulating a Complex Numeric array. 42
2.10. Appending values to an array. 43
2.11. Assigning values with the use of indices and slice notation. 43
2.12. Assign arrays to arrays using slice notation. 43
2.13. Declaring multidimensional Numeric array. 43
2.14. Accessing elements in multidimensional arrays. 43
2.15. Applying multiplication and addition to all elements of an array. 44
2.16. Concatenating numeric arrays. 44
2.17. Applying relational operators to a Numeric array. 44
2.18. Filtering array elements with the where method. 44
2.19. More complex filtering using the where method. 44
2.20. Accessing the array values with a filter. 45
2.21. Adding two arrays with the same set of filtered array indices. 45
2.22. Assigning values with the results of the where method. 45
2.23. The output list of where is not accessible by index. 45
2.24. Converting the output of where to a normal array that you can manipulate. 45
2.25. Converting the output of where makes the resulting object iterable. 45
2.26. Differences between Jython and Numeric arrays. 47
2.27. Removing infinite and NaN values from an array. 47
2.28. Creating a filter (mask) with a function ro remove NaNs. 47
2.29. Avoiding unnecessary array allocation for the addition operation. 48
2.30. Using Java array utility methods to avoid wasteful array allocation. 48
2.31. Grouping scalar multiplication avoids costly array multiplication. 48
2.32. Using arithmetic operations on arrays to avoid loops. 48
2.33. Converting types explicitly requires the creation of a numeric array of a specific type. 48
2.34. Converting types implicitly in Jython. 49
2.35. Dividing by zero will generate NaN or Infinity as appropriate. 49
2.36. Declaring an array dataset. 49
2.37. How to modify an array dataset. 50
2.38. Accessing relevant data of an array dataset. 50
2.39. Creating a table dataset. 50
2.40. Creating isolated columns. 50
2.41. Adding data columns to table datasets. 51
2.42. Avoiding references to the same data from two different columns. 51
2.43. How to correctly create independent columns in a table dataset. 51
2.44. Adding columns directly to a table dataset. 51
2.45. Exercising some of the most useful methods of a table dataset. 51
2.46. Copying table datasets. 52
2.47. Exercising the most useful methods of an array dataset. 52
2.48. Creating a composite dataset. 52
2.49. Adding a table dataset to a composite dataset. 53
2.50. Manipulating a composite dataset. 53
2.51. Exercising the most useful methods of a composite dataset. 53
2.52. Assigning units to variables. 54
2.53. Assigning units to columns or datasets. 55
2.54. Creating a new, derived unit. 55
2.55. Converting units between standard SI prefixes. 55
2.56. Printing unit names for ASCII output or dialog output that includes symbols and Greek characters. 55
2.57. Parsing the string representation of the unit to assign it to a variable. 55
2.58. Assigning units to variables. 56
2.59. Retrieving the conversion factor to SI units. 56
2.60. Using the to method to explicitly convert units. 56
2.61. Checking if two different units refer to the same physical quantity. 56

2.62. Checking if two units are the same but expressed differently. 56
2.63. Using physical constants provided within the Constant class. 57
2.64. Adding and modifying metadata associated to a table dataset. 57
2.65. Inspecting the metadata of a dataset. 58
2.66. Creating an empty product with some metadata. 58
2.67. Overwriting an array within a dataset. 59
2.68. The most common metadata have attributes defined. 59
2.69. Some of the time attributes are instances of FineTime. 59
2.70. Creating TAI or UTC time strings to set time metadata. 59
2.71. Inspecting an object from a subclass of Product. 60
2.72. Saving the product history to a script file. 61
2.73. Checking if a task has been executed on a product (either locally or as part of standard processing). 61
3.1. Adding data arrays as columns to a spectrum dataset. 63
3.2. Setting the units of various spectral metadata. 63
3.3. Adding spectral segments to a one-dimensional spectrum dataset. 64
3.4. Plotting the spectrum with the wave and the flux as axes. 65
3.5. Turning off automatic conversion of errors to weight. 65
3.6. Converting weights to errors and errors to weights. 65
3.7. Manipulating spectral segments. 65
3.8. Creating a two-dimensional dataset containing four spectra. 67
3.9. Creating a multiband spectrum dataset. 67
3.10. Plotting some columns selected using slice notation. 69
3.11. Accessing subbands using specific methods 69
3.12. Plotting the first spectrum of a subband using indices. 69
3.13. Plotting the first spectrum of a subband using the get method. 69
3.14. Inspect subbands. 69
3.15. Converting instrument specific spectral datasets to SimpleSpectrum. 69
3.16. Creating cubes from Numeric arrays. 70
3.17. Creating a weight cube. 70
3.18. Extracting the image metadata (including Wcs information). 71
3.19. Accessing the image (flux) data from a cube. 71
3.20. Extracting a single spectrum from a cube. 71
3.21. Extracting a single image plane for a specific frequency. 72
3.22. Printing the cube dimensions. 72
4.1. Creating a WCS object from scratch. 76
4.2. Getting the world coordinates from a screen pixel position. 76
4.3. Adding a third axis to a WCS structure to define an image index. 77
4.4. Printing if the third axis is regularly sampled. 77
4.5. Transforming between world coordinates and 77
5.1. Taking the square root of a numeric array of doubles. 79
5.2. Numeric functions are applied to each element of an array. 79
5.3. Converting values to double as it is the type of the numeric arrays. 79
5.4. Using lambda expression to apply new functions to arrays in the same way as Numeric functions. 79
5.5. For simple functions it is much more readable to use the built-in operators. 80
5.6. The SIN function works for arrays and scalars. 80
5.7. Finding the minimum value of an array. 80
5.8. Differences between the lower-case Jython functions and the upper-case Numeric functions. 81
5.9. FFT of a modulated signal, with and without HAMMING smoothing 82
5.10. Transforming a signal into the modulus of its spectrum. 82
5.11. Transforming a real signal into a spectrum. 83
5.12. Transforming a real signal with even symmetry into a spectrum. 83
5.13. Transforming a real signal with odd symmetry into a spectrum. 83
5.14. Example of the use of the convolution algorithm. 87
5.15. Importing the Convolution module. 87
5.16. Create a convolution function with zeroes beyond the edges. 87

5.17. Create a convolution function with circular wrapping beyond the edges. 87
5.18. Create a convolution function with value repetition beyond the edges. 88
5.19. Create a centred convolution function with zeroes beyond the edges. 88
5.20. Create a centred convolution function with circular wrapping beyond the edges. 88
5.21. Create a centred convolution function with value repetition beyond the edges. 88
5.22. Creating different filtering functions using the Convolution module. 88
5.23. Interpolation functions in DP 88
5.24. Defining some X-Y data points. 89
5.25. Fitting data with a polynomial model (linear). 90
5.26. Fitting data with a gaussian model (non-linear). 90
5.27. Executing the fit with or without parameters. 90
5.28. Printing the results of the fitting. 90
5.29. Re-sampling the fit data according to the model. 90
5.30. Retrieving the statistical indicators of the goodness of fit. 91
5.31. Retrieving the unscaled standard deviation from the fit. 91
5.32. Creating a custom non-linear fitting model. 92
5.33. Using a custom fitting model. 93
5.34. Fitting a line using two models at the same time. 93
5.35. Plotting the results of a polynomial fitting. 94
5.36. Setting the tolerance for the LevenbergMarquardt fitter. 95
5.37. How to get the dot product of two vectors or matrices. 97
5.38. Transposing a matrix. 98
5.39. Finding the determinant of a matrix. 98
5.40. Inverting a matrix. 98
5.41. Multiplying matrices this way returns a matrix. 98
5.42. Multiplying a matrix by a vector with matrix multiplication. 98
5.43. Decomposing a matrix to lower and upper matrices. 99
5.44. Verifying the results of a LU decomposition. 99
5.45. Getting the eigenvalues of a matrix after decomposing it. 99
5.46. Generating random numbers with this utility class. 101
5.47. Setting a seed for a random number generator. 101
5.48. Integrating numerically using the Romberg method. 102
5.49. Integrating numerically using the Simpson method. 102
5.50. Integrating tabular data using Newton-Cotes method. 103
5.51. Creating a fitter function with different fitters and models. 103
5.52. Customising the fitter even setting the simplex. 103
5.53. Creating a cubic spline interpolator. 103
5.54. Transforming a signal with a continuous wavelet. 105
5.55. Selecting one continuous wavelet. 106
5.56. Transforming a signal using a discrete wavelet. 106
5.57. Discrete wavelet transformation of a one dimensional signal. 107
5.58. Discrete wavelet transformation of a bidimensional signal. 108
5.59. Discrete Wavelet transformation manually handling the coefficients. 108
5.60. Stationary wavelet transformation of a one-dimensional signal. 109
5.61. Stationary wavelet transformation of a bidimensional signal. 109
5.62. Use of the wavelet thresholding tool. 110
5.63. Applying a threshold for wavelets using the visitor mechanism. 110
6.1. Printing the documentation of a task. 113
6.2. Executing the clear task with one parameter. 113
6.3. Retrieving the output value from a task. 113
6.4. Printing the status message of a task. 114
6.5. Naming the parameters to omit optional ones or pass them in any order. 114
6.6. Mixing named and positional parameters. 114
6.7. Wrong mix of mixed and named parameters. 114
6.8. Assigning output values to variables using list slicing. 114
6.9. Assigning output values to variables filtering using list comprehension syntax. 115
6.10. Assigning output values to variables filtering with lambda expression. 115
6.11. Assigning output values to variables filtering using list comprehension syntax (II). 115

6.12. Assigning output values to variables using the utility method outToIndex. 115
7.1. Registering many pools at once during storage definition. 116
7.2. Registering pools after storage creation. 117
7.3. Printing a map of all registered pools. 117
7.4. Removing products from a storage. 117
7.5. Tagging a product and adding it to a storage. 118
7.6. Loading a tagged product as a product reference. 118
7.7. Loading a tagged product. 118
7.8. Tagging an existing product. 118
7.9. Tagging a product with several tags. 118
7.10. Removing tags from a product. 118
7.11. Checking tag existence (in a storage) before tagging. 118
7.12. Rebuilding the index of a pool. 119
7.13. Using keyword queries to retrieve products from a storage. 120
7.14. Querying a storage with several keywords. 120
7.15. Retrieving references to all products in a storage. 120
7.16. Finding all products matching a class. 120
7.17. Querying by class and keywords at the same time. 120
7.18. Inspecting the results of a query. 121
7.19. Versioning products within a storage. 121
7.20. Retrieving the latest version of a product. 121
7.21. Printing version and tag information for each product. 121
7.22. Using a default query returns the latest version. 121
7.23. Returning all versions of a product in a query. 121
7.24. Creating simple, attribute, metadata and full (or data mining) queries. 122
7.25. Creating a simple query. 122
7.26. Creating an attribute query. 122
7.27. Creating a metadata query. 122
7.28. Creating a full query. 122
7.29. Creating a metadata query with SQL-like wildcards for values. 123
7.30. Querying by metadata requires the keyword to exist in all filtered products. 123
7.31. First step filtering the products containing the keyword. 123
7.32. Second step filtering by keyword value. 123
7.33. Changes to products should be done in memory before saving them to a pool. 124
7.34. Loading the product back from the pool, changing and saving to persist the change. 124
7.35. Checking if a reference is loaded in memory. 124
7.36. Loading specific parts of a product. 124
7.37. Saving a context to a pool without the leaf products in memory. 124
7.38. Comparing products in memory. 125
7.39. Comparing product references with hash codes. 125
7.40. Comparing product URNs using hash codes. 125
7.41. Saving a context. 125
7.42. Filtering directly on metadata values. 126
7.43. Filtering products in the archive that contain a specific metadata. 126
7.44. Filtering by value a set of products that contain the metadata. 126
7.45. Using a full query to filter by data values. 127
7.46. Creating a read-only pool connected to the archive. 127
7.47. Adding cache behaviour to the HSA read pool. 127
7.48. Creating an HTTP client pool. 128
7.49. Creating a cached pool from a URL. 128
7.50. Creating a cached pool from an already created remote pool. 128
8.1. Importing a complete package. 134
9.1. How to obtain the current time by various methods. 145
9.2. Different ways of formatting time variables. 146
9.3. Creating a date object. 146
9.4. Time conversion between `Date` and `FineTime` 147
9.5. Creating FineTime variables from other time formats. 148
9.6. Calculating the angle between vectors. 149

A.1. Boolean and operation between integers is also valid. 151
A.2. Boolean or operation between integers is also valid. 151
A.3. Boolean not operation between integers is also valid. 151

Preface

This manual is intended for advanced users interested in developing scripts and tools within HIPE. It complements the *cookbook* approach, based mostly on graphical interfaces, followed by the *Data Analysis Guide*.

Chapter 1. Scripting and Jython basics

1.1. Getting started

1.1.1. Why scripting with HIPE?

HIPE comes with plenty of graphical tools for your data analysis (head to the Data Analysis Guide to find out more about them). While you can go a long way by using these point-and-click tools, scripting can dramatically improve your efficiency:

- You can automate tedious tasks and have HIPE do them for you, as many times as you like.
- You can record a procedure that worked particularly well and store it as a script for future reference, including the best values of any parameters.
- Some HIPE components, such as plotting, rely heavily on scripting.

Note that many graphical tools in HIPE echo all their actions to the command line in the *Console* view, so that effectively HIPE writes a script for you as you point and click.

1.1.2. Testing commands

If you are unsure about the syntax or the behaviour of a command, the best thing to do is to experiment with it. Just back up your data before trying anything potentially risky!

The *Console* view of HIPE is where you try commands for immediate execution. Write your command at the prompt and press **Enter**.

There are plenty of commands for you to try in the rest of this chapter. For more information about executing commands interactively, see Section 1.3.

If you are familiar with IDL and want to see what the most common commands look like in HIPE, see Section 1.41.

1.1.3. Writing your first script

The *Editor* view of HIPE is where you write your scripts. This is a full code editor with features such as syntax highlighting, automatic indentation, incremental search and many more.

In HIPE, choose *File* → *New* → *Jython Script*. A new blank document opens in the *Editor* view. Copy and paste the following script:

```
data = Int2d(100, 100, 10)
myImage = SimpleImage()
myImage.image = data
Display(myImage)
```

Example 1.1. Creating a simple image with fake data.

Tip

Jython is case sensitive. This means that `SimpleImage` is not the same as `simpleImage` or `simpleimage`.

This script does nothing too exciting: it defines an image of 100x100 pixels and a constant value of 10, then displays it with the default HIPE image viewer. Do not worry if you do not understand the syntax of the script. The point is for you to become comfortable with running scripts.

For more information about editing scripts in HIPE, see the *HIPE Owner's Guide*: Section 11 in *HIPE Owner's Guide*.

1.1.4. Running your first script

Now that you have your script ready, you can execute it in three ways:

- **Line by line.** Note the little arrow icon next to the first line of the script. That shows the line that is going to be execute when you click the icon in the HIPE toolbar. HIPE executes just that line, and the icon moves to the following line. Click the icon again, and HIPE executes that line, and so on. You can also click to the left of any line to have the icon point there, so that in theory you could execute single script lines in any order you want.
- **All at once.** If you click the icon in the toolbar, HIPE executes the whole script, from beginning to end. It does not matter where the icon on the left is, or whether you executed part of the script before.
- **Just one section.** Click and drag your mouse pointer to select part of the script, so that the text appears in white on blue background. If you click the icon, HIPE only runs what you have selected, all at once. Note that, if you have selected just part of a line, HIPE tries to execute just that part, whether it makes sense or not!

For more information about running scripts in HIPE, see again the *HIPE Owner's Guide*: Section 9 in *HIPE Owner's Guide*.

1.1.5. Where to go from here?

Now you know the basics of writing and running scripts in HIPE. You may want to explore the following topics next:

- **The scripting language.** Read the rest of this chapter to learn about Jython, the scripting language used in HIPE.
- **Example scripts.** You will find many examples in the rest of this chapter. Furthermore, you can look at the *Scripts* menu in HIPE. This menu lists useful scripts you can run, or just open and study. You can also look at the *Pipeline* menu, which gives you access to the pipeline scripts for the three instruments. All these scripts, usually well commented, are a good source to explore more advanced scripting techniques.
- **For IDL users.** If you are familiar with IDL, you will feel right at home after looking at the comparison tables in Section 1.41.
- **Sharing scripts.** Once you have written your scripts, you can easily share them with colleagues as HIPE plug-ins. Go to Section 1.40 to discover how.

1.2. Jython, Python and Java

The language you use to write scripts in HIPE is called Jython. Jython is an implementation of the Python (please open this link in a new window/tab) language, already heavily used for scientific purposes (see for example http://www.scipy.org).

Jython is written using the Java programming language, thus combining the power of Java and Python. HIPE currently uses version 2.5 of Jython. The original Python implementation is written in the C programming language, and for this reason is sometimes called *CPython*.

Note

HIPE 7 and earlier versions used Jython 2.1 instead of 2.5. If you have scripts written for those versions, you may have to make changes for them to work under HIPE 8 and newer. See this page in the public Herschel wiki for a list of issues related to the upgrade to Jython 2.5.

If you want to try Python and Jython examples from external sources such as books and tutorials, please keep in mind the following caveats:

- HIPE includes Jython version 2.5, which corresponds to version 2.5 of CPython. Scripts written for more recent versions of CPython (for instance, 3.0 or newer) may not work in HIPE.
- HIPE includes the core Jython engine only. Additional libraries that are part of the full Jython installation are excluded. This may cause example from Jython textbooks to fail when executed in HIPE.

  The best way to fix this is to download the full Jython 2.5 installation from the Jython website, and set the property `hcss.jython.user.path` to include the `Lib` subdirectory of the full Jython installation.

  For more information on setting properties, see the *HIPE Owner's Guide*: Section 4 in *HIPE Owner's Guide*.
- If a library is only available for Python (based on the C programming language) and not for Jython (based on the Java programming language) you will not be able to use it in HIPE. This is the reason why a library such as **NumPy** cannot be used in HIPE.

For more information about the difference between Jython and Python, see http://wiki.python.org/jython/JythonFaq/GeneralInfo (please open this link in a new window/tab).

Tip

If you are familiar with IDL, see Section 1.41 for tables with common IDL commands and their equivalents in HIPE.

1.3. Writing commands interactively

You can write any Jython command in the *Console* view of HIPE to have it executed immediately.

For example you can use the *Console* view as a calculator. Use `+`, `-`, `*` and `/` for the four basic operations, and parentheses for grouping, as in the following example. Note the use of the hash mark # for inserting comments:

```
HIPE> print 2+2
4
HIPE> # This is a comment and is ignored by the interpreter
HIPE> print 2+2
4
HIPE> print 2+2  # A comment on the same line as the code
4
HIPE> print (50-5*6)/4
5
HIPE> print 7/3 # Integer division returns the floor
2
HIPE> print 7/-3
-3
```

You can write a command spanning multiple lines by adding a backslash \ at the end of any intermediate line. When you add a backslash at the end of a line and press **Enter**, the prompt in the *Console* view changes from `HIPE>` to `......`, indicating that you can add another line to the command. Write the last line without a backslash at the end and press **Enter** to execute the multiline command:

```
HIPE> print 5 + \
......(3 * 4) - \
......7
# 10
```

Multiline commands are useful to make long and complicated commands more legible, or to define multiline strings (see Section 1.7).

1.4. Writing a script

In the *Editor* view of HIPE you can write *scripts*, that is, series of Jython instructions that you can save and execute. You can execute a script all at once or line by line, or execute just a portion of a script.

You can turn most code examples in this chapter into scripts. Choose *File → New → Jython Script* . An empty window opens in the *Editor* view. Copy and paste a code example, then choose *File → Save as* to save it. HIPE saves scripts with a `.py` extension.

For more examples of Jython scripts, see entries in the *Pipeline* menu of HIPE.

For more information on opening and running scripts, see the *HIPE Owner's Guide*: Section 9 in *HIPE Owner's Guide*.

For more information on editing scripts, see again the *HIPE Owner's Guide*: Section 11 in *HIPE Owner's Guide*.

Non-ASCII characters. If you include non-ASCII characters in a Jython script, add the following line at the top of the file:

```
# encoding=utf-8
```

Failing to do so could cause errors later.

Script length. Each Jython script cannot exceed a certain size limit, usually 65536 bytes. If your Jython script is very long (more than a few thousands lines) then it is advisable to split it into separate scripts.

1.5. Variables and variable types

A *variable* is a name corresponding to a value. A variable can refer to anything from a number to an entire Herschel observation.

To create a variable, or modify an existing value, use the = operator. You can create several variables at once. The following command creates three variables called `x`, `y` and `z`, holding the values 1, 2 and 3 respectively:

```
x, y, z = 1, 2, 3
```

Example 1.2. Assigning multiple values to multiple variables in one line.

The following command creates two variables, `a` and `b`, both with the value 123.

```
a = b = 123
```

Example 1.3. Assigning the same value to several variables at once.

To delete some of your variables, use the **del** command:

```
del(x,y,z)  # Deletes three variables
```

Example 1.4. Deleting variables.

To delete all your variables, use the **clear** command:

```
clear()  # Deletes all variables
```

Example 1.5. Deleting all variables.

Any variable has a *type* associated to it. To find out the type of a variable, use the `type` function:

```
a = 5
print type(a)  # <type 'int'>
```

Example 1.6. Printing the type (class) of a variable.

Numeric variables (that is, variables representing numbers, can be of the following types in Jython:

- *Integer (`int`):* `a = 3`
- *Long integer (`long`)*, denoted by the `l` or `L` suffix: `a = 3L`
- *Float (`float`):* `a = 3.0`
- *Complex (`complex`):* `a = (3 + 1j)`
- *Boolean (`bool`):* `a = True` or `a = False`

1.5.1. More on complex numbers

Imaginary numbers are written with a `j` or `J` suffix. Complex numbers with a nonzero real component are written as `(`*`real`*` + `*`imag`*` j)`, or can be created with the `complex(real, imag)` function:

```
print 1j * 1J  # (-1+0j)
print 1j * complex(0,1)  # (-1+0j)
print 3+1j*3  # (3+3j)
print (3+1j)*3  # (9+3j)
print (1+2j)/(1+1j)  # (1.5+0.5j)
```

Example 1.7. How to represent complex numbers in Jython.

To extract the real and imaginary parts from a complex number `z`, use `z.real` and `z.imag`:

```
z = 1.5+0.5j
print z.real  # 1.5
print z.imag  # 0.5
```

Example 1.8. Printing complex numbers to the console.

1.5.2. Java variable types

In addition to native Jython numeric variable types, the following Java numeric types are also available in HIPE:

- *Byte:* signed 8-bit integer. Values from -128 to 127.
- *Short:* signed 16-bit integer. Values from -32,768 to 32,767.
- *Integer:* signed 32-bit integer. Values from -2,147,483,648 to 2,147,483,647.
- *Long:* signed 64-bit integer. Values from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.
- *Float:* single-precision 32-bit floating point. Values from 1.40129846432481707e-45 to 3.40282346638528860e+38, either positive or negative.

- *Double:* double-precision 64-bit floating point. Values from 4.94065645841246544e-324 to 1.79769313486231570e+308, either positive or negative.

- *Boolean:* `true` or `false`.

You can use Java types are used as follows:

```
a = Integer(3)  # Create an Integer with value 3
print a  # 3
b = Double(3)
print b  # 3.0
c = Boolean(0)
print c  # false
```

Example 1.9. Printing numbers (with default formatting) to the console.

Warning

Use only Jython primitive types in your scripts. If you use Java types like `Double` or `Integer`, Jython will silently and automatically convert back to native Jython types like `float` or `int` every time that an implicit cast is required (on assignment or comparison), which could result in strange errors when you try to operate on variables of incompatible types. See Section 1.9.1 for more information on these automatic conversions.

Another problem is apparently wrong results in comparisons between numbers. This may happen if you use Java number classes as in the following example:

```
    a = Integer(2)
b = Integer(3)
print a < b
# 0
```

Example 1.10. Comparing variables from Java classes is different to Jython comparisons.

It would seem that, according to HIPE (more precisely, to Jython), two is *not* smaller than three. This is because Jython is comparing the two *objects* `a` and `b`, using criteria that have nothing to do with the numeric values they represent.

1.5.3. The range of Jython numeric types

It is safe to apply the same ranges of Java numeric types to Jython numeric types, according to this list:

- Jython integer: same as Java `Integer`, from -2,147,483,648 to 2,147,483,647.

- Jython long: same as Java `Long`, -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

- Jython float: same as Java `Double`, from 4.94065645841246544e-324 to 1.79769313486231570e +308, either positive or negative.

- Jython complex: real and imaginary part have the same range of their variable type (integer, long or float).

If you go beyond the range of the integer type, a variable is automatically converted to long:

```
i = 2147483647  # Integer range limit
print type(i)
# <type 'int'>
i = i + 1  # Adding 1 to the variable
print type(i)
# <type 'long'>
```

Example 1.11. Checking the range limits of types.

Note that Jython allows you to go beyond the range of the long type, but you won't be able to use the variable in HIPE-specific data structures such as Numeric arrays (see Section 2.2 for more information on Numeric arrays). Going beyond the range limit can cause errors like in the following example:

```
i = 92233720368547758072147483647  # Long range limit
print type(i)
# <type 'long'>
i = i + 1  # Adding 1 to the variable
print i
# 9223372036854775808L
print type(i)
# <type 'long'>
a = Long1d([i]) # Trying to create a Numeric array
# java.lang.IllegalArgumentException: No implicit conversion for:
# 9223372036854775808L: too big for java.Long
```

Example 1.12. Demonstrating the ranges for built-in types in Jython.

1.5.4. Other variable types

This section deals with numeric variable types, but there are many more types you will encounter while working in HIPE. Some of the most important HIPE-specific variable types you are likely to encounter are the following:

- **Numeric arrays.** Numeric arrays are HIPE-specific data structures of up to five dimensions. They are described in Section 2.2.
- **Array, table and composite datasets.** These datasets are used to organise numeric arrays and add *metadata* to describe their contents. The table dataset is by far the most used. These datasets are described in Section 2.3, Section 2.4 and Section 2.5.
- **Products.** Products are the main building blocks of Herschel data. Products contain one or more datasets plus additional metadata. They are described in Section 2.8.
- **Contexts.** Context are special types of products that act as containers for other products. The most important type of context is the *observation context*. Contexts are described in Section 2.8.5 and observation contexts in Section 2.8.6.
- **Spectra and spectral cubes.** There are many variable types used to represent spectra and spectral cubes. These are described in Chapter 3.
- **Images.** The `SimpleImage` type is the most common variable type used to describe images. See the *Data Analysis Guide* for more information on images in HIPE: Chapter 4 in *Data Analysis Guide*.

1.6. Getting help on variables

While working in HIPE you come into contact with variables of many different types. Besides the primitive numeric types described in Section 1.5, there are many more variable types coming from the Java language, and other HIPE-specific variable types, such as those described in Chapter 2.

The following are some ways in which you can obtain help on a variable:

- Right click on the variable name in the *Variables* view and choose, if available, *Help in URM*. This opens in your default browser the corresponding entry in the *User's Reference Manual*.

  The equivalent command from the *Console* view is the following:

  ```
  help(myVariable)
  ```

- Right click on the variable name in the *Variables* view and choose, if available, *Help in DRM*. This opens in your default browser the corresponding entry in the *Developer's Reference Manual*.

This manual is intended primarily for developers, but you can find information on the *methods* (functions) you can apply to your variable.

- Right click on the variable name in the *Variables* view and choose *Show methods*. This prints a list of available methods in the *Console* view.

  The equivalent command from the *Console* view is the following:

```
print dir(myVariable.__class__)
```

  Example 1.13. Printing the methods available for a variable in the console.

- Issue the following command in the *Console* view:

```
print myVariable.__doc__
```

  This prints a short help text, when available.

The *User's Reference Manual* ans *Developer's Reference Manual* include help for HIPE-specific variable types. For help on Jython types, see the online Jython standard library reference.

1.7. Defining and modifying strings

Strings in Jython can be within single or double quotes:

```
print 'spam eggs'  # spam eggs
print "doesn't"  # doesn't
```

String literals can span multiple lines in several ways. A backslash as the last character of a line indicates that the next line is a logical continuation of the previous one:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is \
significant."

print hello
```

Note that newlines still need to be embedded in the string using `\n`; the newline following the trailing backslash is discarded. The previous example would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

You can access individual characters like this:

```
print hello[2]  # Third character: i
print hello[10:16]  # 11th to 16th character: rather
```

Note that numbering of the characters starts at 0.

Counting the characters in a string. To obtain the number of characters in a string, including any white spaces, use the following command:

```
print len(hello)  # 158
```

Concatenating strings. Use the + operator to concatenate strings:

```
a = "Blah Blah "
b = "Woof Woof"
```

```
print a + b  # Blah Blah Woof Woof
```

Example 1.14. Concatenating strings with + in Jython does not have the performance impact it has in Java.

Converting numbers to strings. Use backquotes (`) to convert a numeric variable to its string representation:

```
a = 42
print type (a)  # <type 'int'>
print type (`a`)  # <type 'str'>
```

Example 1.15. Converting numbers to string using the builtin `backquote` method.

1.7.1. Java string types

As with numeric types, you can use Java strings in addition to Jython native strings:

```
s1 = "Blah blah"  # Jython string
s2 = String("Woof woof")  # Java string
```

Example 1.16. Creating a Java string from a Jython string.

Java also has the `Character` type representing a single character. Note that it is not available by default within HIPE, but it has to be explicitly imported (see Section 1.33 for more information about importing):

```
c = Character("a")
#  <type 'exceptions.NameError'>: name 'Character' is not defined
from java.lang import Character
c = Character("a")  # No error this time
print c  # a
```

Example 1.17. Creating Java Character variables.

Always use Jython strings in your scripting. If you come across a Java string, you will not be able to use it together with Jython strings:

```
a = "Blah Blah "  # Jython string
b = "Woof Woof"  # Jython string
c = String("Woof Woof")  # Java string
print a + b  # Concatenating Jython strings
# Blah Blah Woof Woof
print a + c  # Jython and Java string
# <type 'exceptions.TypeError'>: cannot concatenate 'str' and 'java.lang.String'
 objects
print a + str(c)  # Convert Java string to Jython
# Blah Blah Woof Woof
```

Example 1.18. Jython and Java string incompatibilities.

1.8. Formatting strings

You can insert variable values in a string, either numeric values or other strings, with the `%` operator. The following example shows how to insert other strings in a string:

```
a = "Herschel"
b = "HIPE"
print "I like %s." % (a)
# I like Herschel.
print "I like %s and %s." % (a, b)
# I like Herschel and HIPE.
```

Example 1.19. Printing text to the console with variable substitution.

The `%s` means that the variable contents must be formatted as a string.

The following example shows how to insert numeric values:

```
value = 4.0/3.0
print "Four thirds is approximately %f" % (value)
# Four thirds is approximately 1.333333
print "Four thirds is approximately %4.2f" % (value)
# Four thirds is approximately 1.33
```

Example 1.20. How to change the floating-point precision when printing numbers.

The `%f` means that the variable contents must be formatted as a floating point number. In the second command, `%4.2f` means that the number must be written with at least four characters, with two characters reserved for digits after the decimal point.

The following table shows other conversion types you can used in addition to `s` and `f`.

Table 1.1. Conversion types for string formatting.

Type	Description
d	signed integer decimal
o	unsigned octal
u	unsigned decimal
x	unsigned hexadecimal (lowercase)
X	unsigned hexadecimal (uppercase letters)
E	floating point exponential format (uppercase ‘E’)
e	floating point exponential format (lowercase ‘e’)
f	floating point decimal format (lowercase)
g	floating point exponential format if exponent < -4, otherwise float
G	floating point exponential format (uppercase) if exponent < -4, otherwise float
c	single character
s	string

1.9. Converting between variable types

Use the following functions to convert variables to different Jython types: `float()`, `int()`, `long()` and `complex()`.

```
a = 1
print a  # 1
print float(a)  # 1.0
print long(a)  # 1 - No visible change
print complex(a)  # (1+0j)
```

Example 1.21. Converting between Jython types.

These conversions do not work with complex numbers, even if they have zero imaginary part:

```
a = 1 + 0j
print float(a)
# <type 'exceptions.TypeError'>: can't convert complex to float; use e.g. abs(z)
```

Example 1.22. Complex to float conversion is impossible.

You can also convert from string to numeric values:

```
s = "01234.56"
print float(s)  # 1234.56
```

Example 1.23. String to float conversion.

Note that with this method when you try to convert a string representation of a floating point to integer you will get an error:

```
s = "01234.56"
print int(s)
# <type 'exceptions.ValueError'>: invalid literal for int() with base 10:
# 01234.56
```

Example 1.24. Decimal to integer conversion is impossible.

1.9.1. Converting between Java and Jython types

When it gets a Java numeric type, or a Java string, Jython automatically converts it into one of its primitive types. Take for example the following code, which generates a random number between 0 and 1 using a Java function.

```
from java.util import Random
a = Random().nextDouble()
print a
# 0.7865746478405673 (You will get a different number!)
```

Example 1.25. Implicit Java to Jython numeric conversion.

The output of `nextDouble()` is a Java `Double`, but if you inspect the type using `print type(a)` you get something different:

```
print type(a)
# <type 'float'>
```

Example 1.26. Printing the title of a variable.

Another way to see whether a variable is a Java or Jython type is to check for the `.class` attribute. This is only available for Java types and gives an error for Jython types:

```
a = 1  # Jython type
print a.class
# <type 'exceptions.AttributeError'>: 'int' object has no attribute 'class'
b = Integer(1)  # Java type
print b.class
# <type 'java.lang.Integer'>
```

Example 1.27. Printing the class of a Java variable.

Java types are converted to Jython types according to the following table:

Java type	Jython type
Byte	Integer
Short	Integer
Integer	Integer
Long	Long
Float	Float
Double	Float
Boolean	Integer (`False` = 0, `True` = 1)

Java type	Jython type
Character	String (length 1)
String	String

Incompatible types

Java and Jython numeric types do not mix well:

```
a = 123.45
print type(a)  # <type 'float'>
b = Float(123.45)
print type(b)  # <type 'java.lang.Float'>
print a + b
# <type 'exceptions.TypeError'>: unsupported operand type(s) for +: 'float' and
 'java.lang.Float'
```

Example 1.28. Another example of Java and Jython numeric type incompatibilities.

Although the two variables *look* the same, to HIPE they are just two different things for which no addition has been defined. For the addition to succeed, you have to convert the Java type to Jython (you may get a slightly different result because of rounding errors):

```
print a + b.floatValue()  # 246.9
```

Example 1.29. Converting the Java numeric variable to a Jython value.

Converting the Jython type to Java will not work:

```
print Float(a) + b
# <type 'exceptions.TypeError'>: unsupported operand type(s) for +:
 'java.lang.Float' and 'java.lang.Float'
```

Example 1.30. the Jython type to Java will not work.

To apply math operators to variables of Java numeric types, you *always* have to convert them to Jython types. This is another very good reason to use Jython primitive types in the first place.

1.10. Lists, dictionaries and tuples

Lists, dictionaries and tuples are important data structures available in Jython.

Lists are arrays of values written in a specific order.

```
myList = ["one", "two", "three"]
```

Tuples are just like lists, but they cannot be modified once they are created.

```
myTuple = ("one", "two", "three")
```

The only difference in syntax is round parentheses for tuples instead of square brackets for lists.

Dictionaries are lists of key-value pairs. To access each value you must specify the corresponding key.

```
person = {"Alice": 111, "Boris": 112, "Clare": 113, "Doris": 114}
```

Example 1.31. Defining dictionaries.

In the previous example, `Alice` is a key and `111` the corresponding value.

1.11. Creating and modifying lists

You can mix different variable types and nest lists within other lists:

```
name = ["Isaac", "Newton"]  # List of strings
a = [1, 2, "Herschel", 4.5]  # Mixing different types
y = z = 5
x = [[1,2,3],[y,z],[1,[2,[3,4]]]]  # Nested lists
print x
print x[0]
print x[2]
print x[2][1]
print x[2][1][1]
```

Example 1.32. Nesting lists does not require that all lists are of the same type.

To append values to the end of a list, you can use the `+=` operator or the `append` function.

```
b = [1]  # List made of one element
b += [2] # Now b = [1, 2]. Note that the result is not b = [3]!
b.append(3) # Now b = [1, 2, 3]
```

Example 1.33. Appending values to the end of a list, using different methods.

You can use the `+=` operator or the `append` function to fill a list one element at a time through a `for` loop. See Section 1.19 for more information on loops.

```
a = []  # Create empty list
for i in [0.1, 0.2, 0.3]:
 a.append(SIN(i))
```

Example 1.34. Appending several values with the help of a for loop.

1.12. Concatenating lists and tuples

Use the + operator to *concatenate* (that is, join) lists or tuples to form a resultant third list or tuple:

```
a = (1,2,3,4)
c = ("x","y","z")
b = a + c
print b  # (1, 2, 3, 4, 'x', 'y', 'z')
```

Example 1.35. Concatenating two lists or tuples.

You cannot concatenate a list and a tuple:

```
a = (1,2,3)  # Tuple
b = ["x", "y"]  # List
print a + b
# <type 'exceptions.TypeError'>: can only concatenate tuple (not "list") to tuple
```

Example 1.36. Concatenating a list and a tuple will not work.

1.13. Accessing lists and tuples

You can access individual list and tuple elements or range of elements, as shown in the following example:

```
a = [0, 1, 2, 3, 4, 5]
print a[0]  # Accessing first element
# 0
print a[1:3]  # Accessing second and third element
```

```
# [1, 2]
print a[:3]  # From beginning to third element
# [0, 1, 2]
print a[3:]  # From fourth to last element [3, 4, 5]
```

Example 1.37. Accessing elements or ranges using list slice notation.

Accessing list and tuple elements follow these rules:

- Each elements is identified by an integer *index* starting from 0. For example, in a list or tuple of ten elements indexes go from 0 to 9 (list length minus one). So, `a[0]` returns the first element, `a[1]` the second element, and so on.
- `a[:3]` means "take every element from the beginning of the list or tuple up to the fourth element *not included*."
- `a[3:]` means "take every element from the fourth element *included* to the end".
- Negative numbers mean counting from the end of the list or tuple. `a[-3]` returns the third element from the end.

You can access elements of lists nested within other lists (or tuples nested within other tuples) as shown by the following example:

```
x = [[1,2,3],[5,6],[1,[2,[3,4]]]]  # Nested lists (tuples work the same way)
print x  # Entire list
print x[0]  # [1, 2, 3]
print x[2]  # [1, [2, [3, 4]]]
print x[2][1]  # [2, [3, 4]]
print x[2][1][1]  # [3, 4]
```

Example 1.38. Accessing elements of nested lists or tuples.

1.14. Creating and modifying dictionaries

A dictionary has a set of {key: value} pairs. You can create a dictionary as follows:

```
person = {"Alice": 111, "Boris": 112, "Clare": 113, "Doris": 114}
```

Example 1.39. Defining a dictionary.

The keys of this dictionary are `Alice`, `Boris`, `Clare` and `Doris`. The corresponding values are `111`, `112`, `113` and `114`.

To change a value associated to a key:

```
person['Alice'] = 222
```

Example 1.40. Accessing a dictionary value using a key.

The value associated with `Alice` in the dictionary called `person` has been changed to the number 222.

1.15. Accessing dictionaries

By providing a key to a dictionary, you can access the corresponding value:

```
print person['Alice']  # 111
```

Example 1.41. Accessing and printing a dictionary value using a key.

To obtain a list of all the keys and values in a dictionary:

```
print person.keys()
# ['Clare', 'Alice', 'Boris', 'Doris']
print person.values()
# [113, 111, 112, 114]
```

Example 1.42. Printing all the keys of a dictionary.

1.16. Nesting dictionaries

Dictionaries can hold other dictionaries. You can use this feature to create advanced data structures.

The following example puts a dictionary called `abc` into another dictionary called `dict`:

```
abc = {"John": 12345, "Jerry" : 23456, "Joe" : 34567}
dict = {"Alice" : 111, "Boris" : abc, "Charlie" : "angel"}
```

Example 1.43. Using a dictionary as a value of another dictionary.

In this case the value corresponding to key `Boris` is not a number or a string, but an entire dictionary:

```
print dict["Boris"]
#  {'Joe': 34567, 'John': 12345, 'Jerry': 23456}
```

Example 1.44. Accessing a dictionary value is the same as any other value.

You can get a value from the nested dictionary as follows:

```
print dict["Boris"]["John"]  # 12345
```

Example 1.45. Accessing nested dictionaries is the same as with multidimensional arrays/lists.

This is the same syntax you use for nested lists and tuples: see Section 1.13.

1.17. Code blocks

A *code block* is a portion of your Jython script that is set apart from the rest of the code. You use code blocks to make your scripts more readable and maintainable. The following are examples of code blocks:

- Portions of code executed only if a specific condition is true. According to the condition, the execution of the script *branches* to different code blocks. See Section 1.18.
- Portions of code executed many times. The execution of the script *loops* over the code block. See Section 1.19.
- Functions, that is, portions of code that you can *call* and execute from other points of your script. See Section 1.27.
- Classes, that is, bundles of variables and associated functions called *methods*. See Section 1.29.

In Jython, code blocks are indicated *only* through line indentation. No begin/end braces or other special characters are required. See the following sections for examples.

Warning

Do not leave blank lines in your code, especially within code blocks. This could lead to errors or to different behaviours when executing the script line by line or in one go. If you want to leave blank space between statements, use a single comment character # with the correct indentation.

1.18. Writing branching code: if/elif/else

The `if/elif/else` statement executes blocks of commands depending on given conditions. The syntax is:

```
if condition1:
  block1
elif condition2:
  block2
else:
  block3
```

The following example shows how to construct an `if/elif/else` statement, including the use of the `and` and `or` operators to combine logical conditions:

```
      x = 13

if x < 5 or (x > 10 and x < 20):
    print "The value is OK"

if x < 5 or 10 < x < 20:
   print "This value is OK"

if 0 <= x <= 10:
     print "The value is in the range [0,10]"
elif 10 < x < 20:
     print "The value is in the range [10,20]"
else:
     print "The value is not in the range [0,20]"
```

Example 1.46. Checking if a number belongs to several ranges using comparison operators.

1.19. Writing loops: for and while

The for loop. The `for` loop is used to execute a block of code a given number of times.

The syntax of the `for` loop is the following:

```
for variable in list:
  block
```

where list can be an array of values, sequence of dictionary keywords, tuples, strings.

Some examples:

```
for i in [1,2,3]:
   print i
```

Example 1.47. Basic loop printing the index values.

The above `for` loop goes through the values in the `[1,2,3]` list. A simpler way to create lists of increasing integer numbers is using the `range` function:

```
for value in range(100):
  print value
```

Example 1.48. Constructing loops with the help of the range function.

In the previous example, `range(100)` creates a list with values increasing from 0 to 99.

The `for` loop can have an `else` clause, like in the following example:

```
for i in range(3):
    print i
    if i == 1:
        print 'YES!'
else:
    print 'NO.'
```

Example 1.49. Adding an else code block to a for loop that is executed when the loop finishes.

The `else` block is executed after the `for` block has terminated normally. The `else` block is *not* executed if the `for` loop terminates prematurely with a `break` statement, like in the following example:

```
for i in range(3):
    print i
    if i == 2:
        break
else:
    print 'NO.'
```

Example 1.50. Exiting the loop with break will not execute the else block.

The `break` statement is described in Section 1.20.

The following example uses a `for` loop and an `if/elif/else` block together. Note the indentation of the different blocks:

```
        person = {"Alice" : 111, "Boris": 112, "Clare": 113, "Doris": 114}
# Get the list of people's names
list = person.keys()
# For each name in the list, get the associated value. This
# could be a test score, for example.
for i in list:
  pval=person.get(i)
  # Check if the person is on the cutoff, and print the name
  if pval == 112:
    print i, "is at the cutoff"
  # Below the cutoff
  elif pval < 112:
    print i, "is below the cutoff"
  # Or else, above the cutoff
  else:
    print i, "is above the cutoff"
```

Example 1.51. Branching and looping structures in Jython.

The while loop. The `while` loop executes a code block *while* a given condition is true. The syntax is the following:

```
while condition:
   block
```

The condition can be any expression which returns a value: zero is treated as `False`, as are empty strings, tuples or lists. Any other value is `True`.

```
      x = 0
while x <= Math.PI:
  y = SIN(x)
  x += 0.1
```

Example 1.52. Writing a while loop block.

1.20. Controlling loops: break and continue

Use the `break` command to immediately exit from a loop, and `continue` to jump to the next iteration of the loop without executing the rest of the block.

An example for their usage is given below.

```
      x = 0
while 1:
  y = TAN(x)
  if y < 0:
    break
  print x,y
  x += 0.1
```

Example 1.53. How to break out from a while loop.

The above example shows an infinite `while` loop (the condition is always true). Inside the loop block a condition is checked to exit the loop at the first negative tangent.

```
for i in range(100):
   if i % 2: continue
   print i
```

Example 1.54. Exiting an infinite loop with the use of the continue keyword.

The above example shows how you can skip the printing of the odd numbers (`i % 2` is i modulus 2 and it is zero for all even numbers).

1.21. Writing loops in the Console view

Multi-line code blocks are not only possible as part of a script, but you can also create and execute them in the *Console* view of HIPE.

Write the following in the *Console* view and press **Enter**:

```
for i in (1,2,3):
```

This returns `......` instead of the usual `HIPE>` prompt. This means that the interpreter is waiting for the rest of the block. Input for instance a `print i` command *indented by at least one space*. A further `......` is returned. Press **Enter** once more to signal that the block is complete, and the interpreter executes the command.

The whole session should look like this (again, not that `print i` is indented by one space):

```
HIPE> for i in (1,2,3):
...... print i
......
1
2
3
HIPE>
```

Of course you could have added more commands in the block after `print i`.

1.22. Printing to the screen

Use the **print** command to print values and variable contents to screen:

```
print 1, 2, 1+2
# 1 2 3
print a
```

```
# (1, 2, 3, 4)
```

The printout can be formatted in the same way as with the C `sprinf` format codes. Some examples:

```
print "When %s is %i years old then PI will be %8.10f" %("John",23,Math.PI)
# When John is 23 years old then PI will be 3.1415926536
print "When %8s is %04i years old then PI will be %016.12f" %("John",23,Math.PI)
# When     John is 0023 years old then PI will be 003.141592653590
```

To print lists or arrays it is necessary to make a loop:

```
a = [1,1,2,3,5,8,13,21,34]
for i in range(len(a)):
  print "Line: %3i" %(a[i])
```

Example 1.55. Using the range utility function to generate index values.

Another useful usage of formatted printout is with dictionaries as shown in the following example:

```
record = {"name": "John", "Room": 112, "class": "manager", "age": 27}
print "Extracted record\n Name: %(name)10s Room: %(Room)4i" % record
# Extracted record
#  Name:       John Room:  112
```

Example 1.56. Printing content from dictionaries.

1.23. Writing strings to file

You can print to file in the following way:

```
a = "example string" # A Jython string variable
file = open("output.txt", 'w') # 'w' allows write access overwriting
                               # previous contents.
                               # 'a' would append at the end of the file.
print >> file, 2 # Puts the number 2 into output.txt
print >> file, a # Puts the contents of the variable "a" into output.txt
```

Example 1.57. Printing text to file.

Note that it is not necessary to close access to a file within your HIPE session. To overwrite the original text file, reopen the file. Reopening the file will remove the contents.

Tip

Why does Jython complain when I use the backslash (\) in a file name ? The backslash is the *escape character* in Java/Jython. For instance \a is interpreted as Control-A while \t is interpreted as the tab character. If you want to use a backslash, either *escape* it, that is, write it twice as `\\`, or replace it with a forward slash (`/`).

For example, change the expression `\Documents\demo_data.txt` to either `\\Documents\\demo_data.txt` or `/Documents/demo_data.txt`.

1.24. Reading strings from file

You can read strings of characters from a file as shown in the following example:

```
file = open("input.txt", 'r')
a = file.read()  # Reads entire file into variable a
a = file.read(5)  # Reads first five bytes of the file
a = file.readline() # Reads one line of the file
a = file.readlines() # Reads all the lines of the file into an array
```

Example 1.58. Reading text from files.

Note that everything will be read as a string, even numeric values. If you want to read numeric values in tabular form from a text file, see the *Data Analysis Guide*: Chapter 2 in *Data Analysis Guide*.

1.25. Writing numeric values to file

You can write numeric Jython data (single numbers, lists, tuples, dictionaries) to file with the `pickle` module. Use this method only if you want to write native Jython data and read it back with HIPE or with another Jython/Python interpreter. The `pickle` module has the following limitations:

- Data are written to the file in binary format, so you cannot read and change the file with a text editor.
- The `pickle` command can be used only with native Jython data. To save the HIPE-specific data described in Chapter 2, refer to the *Data Analysis Guide*: Chapter 2 in *Data Analysis Guide*.
- Do not use `pickle` if you want to exchange data with other applications. Instead, use table datasets, explained in Section 2.4, and save them to FITS or text files via a right click.

You can write data using `pickle` as in the following example:

```
import pickle  # Importing module, you need to do it only once
a = [1,2,3]  # Creating two lists
b = [3,4,5]
file = open("output.txt", 'wb')
pickle.dump(a, file)  # Saving lists to file
pickle.dump(b, file)
```

Example 1.59. How to use the well-known pickle library.

See Section 1.26 for information on how to read back into HIPE data saved with `pickle`.

1.26. Reading numeric values from file

If you have written data to file using `pickle` (see Section 1.25) you can read them back as in the following example:

```
import pickle  # Importing module, you need to do it only once
file = open("output.txt", 'rb')
a = pickle.load(file) # Read first record into a
b = pickle.load(file) # Read second record into b
# And so on until the end of the file
```

Example 1.60. Load data from file serialised using the pickle library.

Records can be simple numeric values or more complex Jython data structures such as lists, dictionaries or classes. The HIPE-specific data structures described in Chapter 2 cannot be read from file using `pickle`. For information on how to read data from text files into HIPE-specific data structures, see the *Data Analysis Guide*: Chapter 2 in *Data Analysis Guide*.

1.27. Functions

A function is a code block with a *name*. You use the function name to *call* the function, that is, to execute its code block.

A function may have a set of *input parameters* and one *return parameter*. Input parameters are values that the function needs to execute its code. The output parameter is the result of the code execution.

You create a function with the keyword `def`:

```
def square (x):
```

```
    return x*x
```

Example 1.61. Defining functions in Jython.

The `square` function takes one input parameter, called `x`. The following line calls the function passing `2` as the value for the input parameter:

```
print square(2) # 4
```

The arguments of the functions are passed *by value*. This means that input arguments are not changed outside the function:

```
def myfunc(a):
  a = a + 1
  return a
#
x = 4.0
print myfunc(x)  # Passing x as input parameter
# 5.0
print x  # The function returned 5.0, but x was not changed.
# 4.0
```

Example 1.62. Declaring functions for reuse.

Note that variables from the main HIPE session have *global scope*: they are accessible inside functions but *cannot* be changed. The example below gives an error:

```
def myfunc(a):
  a = a + 1
  x = x + 5
  return a
#
x = 4.0
print myfunc(x)
# <type 'exceptions.UnboundLocalError'>: local variable 'x' referenced before
 assignment
```

Example 1.63. Global variables in Jython.

Using global variables. The following example shows a dangerous effect of relying on global variables:

```
def myfunc(a):
  return a*z + 1
#
x = 4.0
z = 10.0
print myfunc(x)
# 41.0
```

Example 1.64. Declaring a global variable can be dangerous in an interpreted language like Jython.

The function works before the global variable `z` has been defined. However, there is no guarantee that `z` will be defined in future HIPE sessions, which could cause the function to give an error. The advice is to always pass global variables as function arguments, as in the following example:

```
def myfunc(a, z):
  return a*z + 1
#
x = 4.0
z = 10.0
print myfunc(x, z)
```

```
# 41.0
```

Example 1.65. Passing global variables as function arguments.

Default argument values. Input arguments may have default values. This is illustrated by the following example:

```
def myfunc(x,y=1.0,verbose=True):
  z = x*x + y
  if (verbose):
    print "The input is %f %f and the output is %f" %(x,y,z)
  return z
#
myfunc(5.0) # Using default values for y and verbose
# The input is 5.000000 1.000000 and the output is 26.000000
print myfunc(5.0,y=5.0,verbose=False)
# 30.0
print myfunc(5.0,5.0,False) # The same as the previous
# 30.0.
print myfunc(5.0,5.0)
# The input is 5.000000 5.000000 and the output is 30.000000
# 30.0
```

Example 1.66. Setting default values for function arguments.

Functions as function arguments. The arguments of a function can be functions themselves, like in the following example:

```
def func1(x):
  return x*x
def func2(x):
  return x/2.0
def myfunc(f1,f2,x):
  return f1(x) + f2(x)
#
x = 3.0
print myfunc(func1,func2,x)
# 10.5
# Even the user can input any available function of one argument
print myfunc(SIN,func1,x)
# 1.6411200080598671
```

Example 1.67. Passing functions as arguments is allowed in Jython.

Functions without input arguments. To define and call a function without input arguments, then the `()` brackets are still required:

```
def myfunc():  # No arguments
  print "This function just prints a message."
#
myfunc()  # Calling the function
# This function just prints a message.
```

Example 1.68. Declaring a function without arguments.

1.28. Executing HIPE tasks from your scripts

HIPE *tasks* are prepackaged functions that take one or more *input parameters* and return results as *output parameters*. You can execute tasks from your scripts, thus taking advantage of HIPE advanced features.

All tasks have a graphical interface you can open from the *Tasks* view of HIPE. The graphical interface offers an easy way to find out the correct syntax to execute a task from a script:

1. Open the task graphical interface by double clicking on the task name in the *Tasks* view.
2. Fill the task parameters with values and press *Accept* to run the task.
3. The command corresponding to the task execution appears in the *Console* view. Copy the command to a new line of your script.
4. Make any changes to adapt the command to your script. For example, you may want to substitute a literal value with a variable name.

For more information on running tasks, see Chapter 6.

If you want to create a task from scratch, or to modify an existing task, see these tutorials on the *HIPE community* website:

- Creating your first Jython task
- Modifying an existing Jython task

1.29. Classes

A *class* is a bundle of variables and special functions called *methods*.

From a class you create, or *instantiate*, *objects*. You can think of a class as a blueprint from which you can create many objects.

Each object has a *status* and a *behaviour*. The status of an object is defined by its variables (also called *instance variables*), and its behaviour is defined by its *methods*.

See the following section for an example of creating and using a class in Jython.

1.30. Creating and using classes

The following code creates a class called `Basket`:

```
class Basket:
   'A basket that can contain many items.'  # <co id="Basket.doc-co"
 linkends="Basket.doc" />
   def __init__(self, contents=None):  # <co id="Basket.constr-co"
 linkends="Basket.constr" />
     self.contents = contents or []  # <co id="Basket.__init__-co"
 linkends="Basket.__init__" />
   def add(self, element):
     self.contents.append(element)  # <co id="Basket.add-co" linkends="Basket.add" /
>
   def print_me(self):
     result = ""
     for element in self.contents:
       result = result + " " + `element`  # <co id="Basket.print_me-co"
 linkends="Basket.print_me.add" />
     print "Basket contains: "+result
```

Example 1.69. How to create a Jython class.

Basket.doc-co ??? This line is a *documentation string*, a short description of the class that you can see by invoking the command `print Basket.__doc__` (note the two underscore characters before and after `doc`).

Basket.constr-co ??? This line declares a *constructor*, that is, a special method called `__init__` that is called when you want to create an object from a class. Note the two underscore characters before and after `init`, and the `self` parameter, which must always be present.

Basket.__init__-co ??? This line defines the *instance variable* `contents`, which holds the contents of the basket. You can pass some contents when you create a `Basket` object, otherwise you get an empty basket.

Basket.add()?? The add methods adds an element to the contents of the basket (that is, to the `self.contents` variable).

Basket.print_me()??? This line, part of the `print_me` method, prints the contents of the basket. Note the use of the inverted commas around `element`.

The class has one variable `contents` and two methods, `add` and `print_me()` (following `def` in the above example).

You can put the above class definition in a script in the *Editor* view and run it all so that HIPE "knows" about the `Basket` class. Now you can *instantiate* the class, that is, create a particular basket from the `Basket` "blueprint":

```
a = Basket()  # ❶
a.add("saw")  # ❷
a.add("hammer")  # ❸
a.print_me()  # ❹
```

Example 1.70. Instantiating and using methods from a class.

❶ Sets up an empty basket called `a`.
❷ Adds the item `saw` to the basket. It runs the `add` method on the object `a`.
❸ Adds the item `hammer` to the basket.
❹ Prints the contents of the basket called `a`, which should be `saw` and `hammer`. The command runs the `print_me` method on the object `a`.

You could equally have created your basket with one item:

```
a = Basket(["saw"])
```

Example 1.71. Passing parameters to the class constructor.

Note

If you had written `a = Basket("saw")` (without the square brackets) the `print_me()` method would have returned this: `Basket contains: 's' 'a' 'w'`.

The general syntax for calling methods is `object.method(arg1, arg2)`.

In the previous example `a` is the object, while `add` and `print_me` are the methods. Note that you do not explicitly call the `__init__` method when you construct an object. Instead, you call the class name, with or without arguments, depending on the class: in the previous examples, `a = Basket()` or `a = Basket(["saw"])`.

1.30.1. Printing objects

You can use the special `__str__` method to define a string representation of an object, that is displayed when you use the `print` command on the object.

In the `Basket` class script, replace the `print_me` method with the following method:

```
   def __str__(self):
     result = ""
     for element in self.contents:
       result = result + " " + `element`
     result = "Basket contains: " + result
     return result
```

Example 1.72. Overriding the default behaviour of the __str__ method.

Now you can use `print str(a)` instead of `a.print_me()` to display the contents of the basket.

1.31. Naming conventions for classes and variables

The following rules are used to name classes and objects in HIPE. You are advised to follow the same rules when creating your own.

- *Classes*

  Class names consist of words with capitalised initials:

```
MyOwnClass
TableDataset
HifiProduct
```

- *Class instances (objects)*

  Objects (variables) of a particular class have names that start with the first letter in lower case. In general, this translates to the following:

```
myOwnClass = MyOwnClass(....)
table = TableDataset()
a = 2
```

Example 1.73. Naming conventions for objects.

- *Constants*

  Constants have names with all their letters capitalised and words separated by an underscore '_'. This rule also applies to so-called *static instances*. An example is `SIN`: it is the only allowed instance of class `Sin`, since it does not make sense to have multiple instances (they all would compute the sine of an angle in the same way). Some examples are the following:

```
VARIANCE
IS_FINITE
ALL_PRESENT
```

1.32. Creating aliases for class and function names

The names of classes and functions in HIPE are generally long and descriptive, such as `TableDataset` rather than `TDset`.

Longer, descriptive names tend to make scripts more readable. However, you can create shorter *aliases* to save typing as shown by the following example:

```
TDset = TableDataset
```

Example 1.74. Creating aliases.

You can now use `TDset` as you would use the original name `TableDataset`. For more information on what a `TableDataset` is, see Section 2.4.

1.33. Importing modules

Most useful classes and functions are put into Jython *modules* or Java *packages*. These are then imported into a given environment or program with the `import` statement.

In Python and Jython, source code is automatically organised in *modules* by means of each file being a module. Additionally, you can create *Jython packages*, which are directories with script files that also contain a special file called `__init__.py`. When importing other modules or packages, take special care in identifying the visibility of anything you have imported. For example, HCSS tasks usually have an alias (or short name, see the URM for reference), e.g.: `calcAttitude` is an alias for `CalcAttitudeTask`. If you want to use the alias of a task when using command-line *hipe* or when several packages are involved, the best way is to re-create the alias by instancing the task (see Section 1.31):

```
from herschel.ia.toolbox.pointing import CalcAttitudeTask
calcAttitude = CalcAttitudeTask()
def function():
   calcAttitude(...)
```

Example 1.75. Instantiating/aliasing an imported task.

This instantiation statement will be imported along with your package or module and will be accessible when Jython executes `function` in the context of the caller module, avoiding a `name not found` error.

Try issuing the following command from within HIPE:

```
print localtime()
```

Example 1.76. Executing methods from non-imported modules.

You get an error:

```
<type 'exceptions.NameError'>: name 'localtime' is not defined
```

This is because, although the `localtime` function is part of the software distribution, it has not been *imported* into your session. The `localtime` function is part of the `time` Jython module, which you can import by issuing this command:

```
import time
```

Example 1.77. Import statement.

This imports the entire module, but forces you to use the *qualified name* of the function (that is, including the module name):

```
print time.localtime()
# (2011, 9, 2, 12, 19, 46, 4, 245, 1)
```

Example 1.78. Using names after importing a whole module.

The following syntax allows you to use the `localtime` function without the qualified name:

```
from time import localtime
print time.asctime(localtime())
# Fri Sep  2 12:20:31 2011
```

Example 1.79.

Note that `asctime`, which converts the time into a human-friendly format, still needs the qualified name. To import more than one name from a module, use a comma-separated list like in the following example:

```
from time import localtime, asctime
print asctime(localtime())
```

```
# Sun May 17 10:44:35 2009
```

Example 1.80. Importing several names from a module.

Note that some of the names imported from the module could overwrite names you defined locally. To see all the names contained in a module, use the following command (here for the `time` module):

```
print dir(time)
```

To avoid name clashes, you can define a different name from what you import:

```
from time import localtime as ltime
print ltime()
# (2011, 9, 2, 12, 19, 46, 4, 245, 1)
```

Example 1.81. Aliasing imports to avoid name clashes.

Importing Java packages works in exactly the same way as importing Jython modules.

Most of the functions you are likely to need are imported automatically when HIPE starts, so you won't need to use the `import` statement very often.

What modules can you import? The list of modules you can import into HIPE is virtually endless. Moreover, you will need only a small fraction of all the modules you can theoretically import. When a module is not imported automatically by HIPE, the code examples in the documentation show the necessary import statements.

If you want to have a look at what is available, these are some reference resources:

- **HIPE-specific modules and packages.** Chapter 8 has a list of the main packages that come with your HIPE installation. For a more comprehensive list see the *Developer's Reference Manual*.
- **Jython native modules.** See the online Jython standard library reference.
- **Java native packages.** See the online Java reference documentation. This resources is likely to be useful only if you are familiar with Java and know what you are looking for.

Warning

Starting from HIPE 8, Jython was upgraded from version 2.1 to version 2.5. Some import rules have changed, which could cause old scripts to stop working. For more information see the Jython upgrade page on the Herschel public TWiki.

1.33.1. Importing, reloading and unimporting your own modules

Suppose you have written the module `myModule.py` and placed it into `/home/user/some/folder`. The module file contains the following:

```
"""This module contains one simple function"""

def simpleFunc():
    print "Simple message."
```

Example 1.82. Self-contained module with documentation.

To have your module imported automatically when HIPE starts, you can modify the property `hcss.interpreter.imports`. To do so, follow these steps:

1. Choose *Edit* → *Preferences*. The *Preferences* window opens.

2. Click the *Advanced* button and answer *Yes* to the warning. The *Properties* window opens.

3. In the *Filter by property name* text field, write `hcss.interpreter.imports`. This property is the only one left in the list. If you have not modified it previously, its value should be `{}`.

4. Double click the *Value* cell to modify it. List all the files you want to import automatically, like in the following example (but *all on the same line*):

```
{ /home/user/some/folder/myModule.py,
    /home/user/another/folder/anotherModule.py }
```

5. Click *Save* and then *Close*. Click *OK* to close the *Preferences* window.

6. Restart HIPE. Your modules are now imported. In the case of the `simpleFunc()` example, the function will be immediately available in the *Console* view:

```
simpleFunc()
# Simple message.
```

Example 1.83. Demonstrating that modules imported at start-up are available in HIPE.

With this property you can also execute a list of custom `import` statements when HIPE starts. Just add them to a file, for instance `imports.py`:

```
from some.module import Foo
from another.module import Bar
```

Example 1.84. Creating a file with just imports for HIPE start-up.

Then add the `imports.py` file to the `hcss.interpreter.imports` property:

```
   { /home/user/some/folder/myModule.py,
     /home/user/another/folder/anotherModule.py,
     /home/user/yet/another/folder/imports.py }
```

If your module is not imported automatically, you can import it on the fly within HIPE. First you have to add the directory containing the module to the list of paths searched by Jython:

```
import sys
sys.path.append('/home/user/some/folder')
import myModule
myModule.simpleFunc()
# Simple message.
```

Example 1.85. How to modify the Jython classpath at runtime.

Reloading a module. If you modify your module and want to apply the changes to your current HIPE session, use the **reload** command:

```
reload(myModule)
```

Example 1.86. Reloading updates the module loaded in memory with the latest changes from the source files.

Unimporting a module. To unimport a module so that it is no longer available in your HIPE session, use the **del** command:

```
del(myModule)
```

Example 1.87. Removing a module from memory using del.

Of course this command does not delete the `myModule.py` file.

1.34. Understanding pipeline scripts

Rather than writing scripts from scratch, you may have to understand and possibly modify scripts written by others. This is especially true for *pipeline scripts*, available from the *Pipelines* menu in HIPE. These are the scripts you run when you want to reprocess your data.

Pipeline scripts are essentially a series of task calls. For specific details on running and understanding pipeline scripts for a given instrument and observing mode, see the instrument manuals:

- HIFI.
- PACS photometry.
- PACS spectroscopy.
- SPIRE photometry.
- SPIRE spectroscopy.

To fully understand a pipeline script you must also understand the tasks it calls. You can see the documentation and the source code of each task by following these steps:

1. Find the task in the *Tasks* view and double click on it. The task dialogue window opens in the *Editor* view of HIPE.
2. Click *Help* to open the task entry in the *User's Reference Manual*.
3. Click *Source* to open the source code of the task in the *Editor* view. For this to work you must have chosen to include the source code when installing HIPE.

   Note that most HIPE tasks are written in Java, so you will need some familiarity with this language to understand their source code.

1.35. Accessing files and directories

It is possible to print the file contents of the current working directory using the following in a console window.

```
import os

# Print the working directory
print os.getcwd()
# Print the names of the files in the working directory
print os.listdir(os.getcwd())
# Any directory name can be placed in the brackets
print os.listdir('anotherDirectory')
```

Example 1.88. How to list the contents of the current directory.

The **listdir** provides an unsorted listing of all the files and subdirectories within a directory. Use the *glob* module to filter the file list. You can use wildcards such as "*", "?", "[]" and so on:

```
import glob
fitsfiles = glob.glob("/my/directory/*.fits")
# fitsfiles is a list of the files that were found.
print fitsfiles  # Print the list
```

Example 1.89. Using the glob module to recursively list files with wildcard matching.

1.36. Adding simple dialogue windows

This section explains how you can make your scripts more interactive, having them asking the user for input and reacting accordingly.

The scripts in the following subsections use *Swing*, a Java library used to create graphical interfaces. For more information about using Swing in Jython scripts, a possible source is the first chapter of the *Jython Book*.

1.36.1. Dialogue box with message

The following example shows how to display a message in a window, together with an OK button:

```
from javax.swing import JOptionPane
print "Let's stop for a while"
JOptionPane.showMessageDialog(None, "Press OK to continue")
print "Well done."
```

Example 1.90. Using pure Java to display dialogues to the user.

The first line imports the swing package (note that it is `javax` rather than `java`). Then we have the line creating the window, embedded between two lines printing text messages to demonstrate that the script will not advance until we press the OK button.

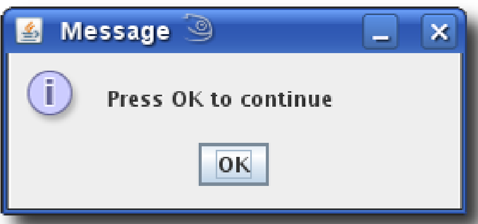


Figure 1.1. The window that appears calling the Swing `showMessageDialog` method.

In the above example, the first parameter of `showMessageDialog` is set to `None`. This parameter indicates the "parent" element of the dialogue box. For creating single dialogue boxes, you do not need to set this parameter to anything else.

You can add two more parameters to customise the window title and the icon:

```
JOptionPane.showMessageDialog(None, "Press OK to continue", "Title bar text", \
JOptionPane.ERROR_MESSAGE)
```

Example 1.91. Configuring the style of Java dialogues.

Note that the third and four parameters must both be present. You cannot specify only the title or the icon.

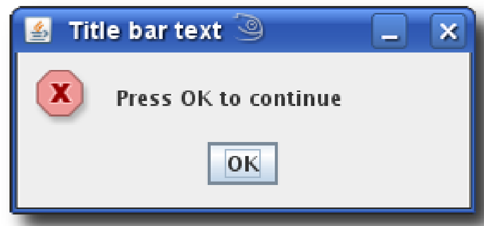


Figure 1.2. Customising the icon and the window title.

Besides `ERROR_MESSAGE`, other available icons are `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` and `PLAIN_MESSAGE`.

1.36.2. Dialogue box with text input field

The following command creates a window for entering text, just like the `raw_input` function:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything")
```

Example 1.92. Using input dialogues to retrieve data from the user.

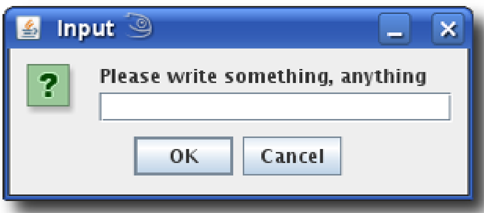


Figure 1.3. The window that appears calling the Swing \\`showInputDialog` method.

Additional options are available for this method as well:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything", \
"Big question", JOptionPane.QUESTION_MESSAGE)
```

Example 1.93. Customising the style of input dialogues.

You can also put a default string of text in the box, like this:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything", \
"Default text")
```

Example 1.94. Providing a default value for input dialogues.

If you want the user to choose from a predefined set of options, you can use `showInputDialog`, as the following script demonstrates:

```
from javax.swing import JOptionPane
myAnswer = ""
possibleAnswers = ["HIFI", "PACS", "SPIRE", "No clue", "All three"]
while myAnswer == "":
  myAnswer = JOptionPane.showInputDialog(None, "Favourite Herschel instrument?", \
  "Test", JOptionPane.QUESTION_MESSAGE, None, possibleAnswers, possibleAnswers[4])
if myAnswer == None:
  myAnswer = ""
print "Your answer is: " + myAnswer
```

Example 1.95. Using lists to restrict the values to use in an input dialogue.

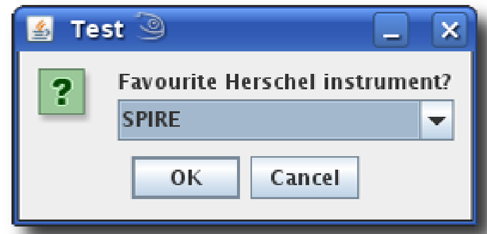


Figure 1.4. A more complex window with a combo box.

Let us go through the parameters one by one:

1. `None`: the "parent" element.
2. `"Favourite Herschel instrument?"`: the window text.
3. `"Test"`: the window title text.
4. `JOptionPane.QUESTION_MESSAGE`: the type of window.
5. `None`: the custom icon. We choose to provide no one and stick with the default one.
6. `possibleAnswers`: the array of possible answers.
7. `possibleAnswers[4]`: the default answer.

1.36.3. Dialogue box asking yes/no question

The `showConfirmDialog` method can be used to display a window asking the user to confirm or block a certain action:

```
from javax.swing import JOptionPane
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?")
if myAnswer == 0:            # Now myAnswer is an integer variable
    print "You agree"
elif myAnswer == 1:
    print "You disagree"
else:
    print "You have no opinion on this"
```

Example 1.96. Displaying a confirmation dialogue.

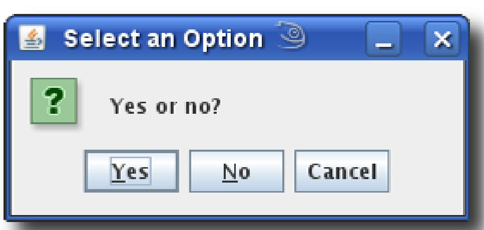


Figure 1.5. Using the Swing `showConfirmDialog` method.

You can use predefined constants to make the code easier to understand:

```
from javax.swing import JOptionPane
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?")
if myAnswer == JOptionPane.YES_OPTION:
    print "You agree"
elif myAnswer == JOptionPane.NO_OPTION:
    print "You disagree"
elif myAnswer == JOptionPane.CANCEL_OPTION:
    print "You have no opinion on this"
elif myAnswer == JOptionPane.CLOSED_OPTION:
    print "You closed the window. How rude!"
```

Example 1.97. Using constants instead of the automatic indexes to improve readability.

You can add another two parameters to provide a title for the window and the type of buttons you want:

```
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?", "Question", \
```

```
        JOptionPane.YES_NO_OPTION)
```

Example 1.98. Adding title and button type to the confirmation dialogue.

Other possible options are `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`, both self-explanatory, and `DEFAULT_OPTION`, which just displays an OK button.

1.37. Pausing and debugging scripts

You can pause a script at any point using the `pause()` command. This allows you to inspect and change variable values while the script is paused, and to resume execution with the new values. This is especially useful to diagnose problems in a script.

Run the following example script:

```
a = 10
pause() # Pause here. You can change the value of a in the debugger.
print a
pause() # Pause again.
print a
```

Example 1.99. Pausing a script to allow debugging or printing text to console.

When the first `pause()` statement is reached, the following window appears:

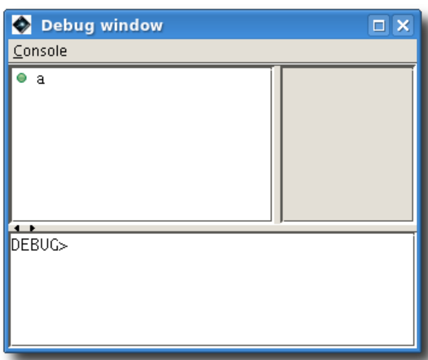


Figure 1.6. The Debug window

Click on a variable name to display a summary in the right-hand area of the window. You can also change the value of a variable at the `DEBUG>` prompt. For example, try changing the value of `a` by writing `a = 100` at the `DEBUG>` prompt and pressing **Enter**. Then resume the script execution by choosing *Console* → *Resume*, and see the new value for `a` reflected in the `print` statements.

Another way to resume the execution of the script is to issue the `resume()` command at the `DEBUG>` prompt. Note that using `resume()` elsewhere, for instance in a script or in the *Console* view, has no effect.

1.38. Interoperating with external software

HIPE offers a complete solution for reducing, visualising and analysing your data. However, for a variety of reasons you may want to do some processing with other astronomical or data analysis software, such as IDL or IRAF. This section explains how to do that.

Any data processing, whether done through an official pipeline or a custom script, is a series of *tasks* applied on *products*, like in the code fragment below. For more information on tasks, see Chapter 6; for more information on products, see Section 2.8.

```
...
```

```
product_2 = TaskA()(product_1)
product_3 = TaskB()(product_2)
...
```

Example 1.100. Outline on how to create a pipeline executing tasks sequentially.

Any task can output a product representing the state of processing up to that point. For example, `product_2` is the result of processing by `TaskA`, before `TaskB` is applied.

To continue processing outside HIPE, you only have to export a product to FITS format, as explained in the *Data Analysis Guide* in *Data Analysis Guide*. See also the `simpleFitsWriter` entry in the *User's Reference Manual*: Section 1.383 in *HCSS User's Reference Manual*.

You can start processing outside HIPE with the `system` instruction. For example, to launch the **myCommand** command insert the following in your script:

```
os.system('myCommand')
```

Example 1.101. Executing platform binaries from within HIPE.

For this to work you need to import the `os` module first:

```
import os
```

Example 1.102. Importing the Jython module that allows communication with the operating system.

Note

With `os.system` you execute external software *outside* HIPE, which means that you will not get feedback from the external command in the *Console* view of HIPE. For example, if you are working on Linux and try to list files in the current directory with the following command, you will not get any result:

```
os.system('ls')
```

However, if you started HIPE from a command line window, you will see the output of the **ls** command in that window.

By the way, you can use this command to obtain a list of files in the current directory from within HIPE:

```
print os.listdir('.')
```

Example 1.103. Listing the contents of a directory.

The **myCommand** executable could be, for instance, a shell script with further processing instructions. Whatever your external processing, it *must* accept as input the FITS file produced by HIPE, and *must* output another FITS file that can then be loaded into HIPE again. For more information on how to load a FITS file into HIPE, see the `fitsReader` entry in the *User's Reference Manual*: Section 1.147 in *HCSS User's Reference Manual*.

A script with part of the processing carried out outside HIPE would look something like this:

```
import os
aProduct = aTask()(inputProduct)
simpleFitsWriter(product = aProduct, file = "aProduct.fits")
os.system('myCommand') # Reads aProduct.fits, produces output.fits
outputProduct = fitsReader(file = "output.fits")
```

Example 1.104. Mixed processing with external commands and HIPE tasks.

1.39. Developing version-aware scripts

There are times when you require your scripts to check for the current HIPE version in order to use features that were introduced in a particular version or build. To do that you have two options, depending on the type of HIPE release that you want to check.

- *User releases:* These releases are downloaded from the HIPE official page and use a Java-based installer that sets some properties in several configuration files. One of these properties is `hcss.release.user` and can be retrieved in Jython using the standard Java method:

```
version = java.lang.System.getProperty("hcss.release.user")
```

Example 1.105. Getting the version number of a user release (method 1).

Note that you can also use the `Configuration` class to retrieve this property:

```
version = Configuration.getProperty("hcss.release.user")
```

Example 1.106. Getting the version number of a user release (method 2).

The string returned by both methods contains the version number in `"major.minor"` format.

- *Developer builds (also known as CIB builds):* These releases are generated each time a new change in HIPE is integrated and can be downloaded from the Continuous Integration Build (CIB) system pages.

Note

These builds are not tested as much as the user releases and may not be suitable for you if you are not a developer.

These builds do not set any property in the configuration system, but there are static methods (in the `Configuration` class) to obtain the version of a particular developer build. You can see an example using these methods below:

```
pi = Configuration.getProjectInfo()
pi.getBuild() # Retrieves the build number
pi.getTrack() # Retrieves the version (in "major.minor" format)
pi.getVersion() # Retrieves the complete version string (in "major.minor.build"
 format)
```

Example 1.107. Getting the build number of a developer build.

1.40. Sharing scripts

If you have a small number of files you want to share with your colleagues, you can just send the `.py` files by whatever means you prefer, for example by email.

If you have larger collections of scripts, you may want to pack them into a HIPE *plug-in*. This allows other users to easily install, update and remove all your scripts at once.

To share Jython scripts as a HIPE plug-in, follow these steps:

1. Put all the scripts you want to share into a directory and rename this directory `scripts`.
2. Compress the directory into a zip file. The zip file must contain the `scripts` directory with the scripts, not just the scripts.
3. Rename the zip file as follows: plug-in name, followed by an underscore, followed by a version number. The version number can be any number of digits separated by dots. An example of a valid file name is `MyScripts_0.1.zip`.
4. The zip file is the plug-in itself. You can now share it, for example by linking to it from a web page.

For information on installing HIPE plug-ins see the *HIPE Owner's Guide*: Section 6 in *HIPE Owner's Guide*.

1.41. IDL to HIPE command mapping

1.41.1. Idl to Jython mapping

The following tables contain the HIPE equivalents of the most common IDL commands and functions.

Warning

HIPE uses `[row, column]` as notation for two-dimensional arrays, while IDL uses `[column, row]`. In addition, HIPE uses *row major* ordering when allocating array elements into memory: elements in the first row are allocated first, followed by elements in the second row, and so on. IDL uses *column major* ordering instead. This difference also holds for arrays of more than two dimensions.

This means that IDL scripts involving loops on large arrays will suffer big performance penalties unless they are adapted to row major ordering. Write your loops so that elements are accessed in the same order they are located in memory:

```
x = [[1,2,3], [4,5,6]]  # Two rows, three columns
# Order of elements in memory:  1  2  3  4  5  6
# Correct way of looping:
for i in range (2):  # Loop on rows
  for j in range (3):  # Loop on columns
    print x[i][j]
```

You should also use array operations instead of loops whenever possible. Look here for more information: Section 2.2.10.

Basic commands	IDL	HIPE equivalent
Create a variable	a = 5	a = 5
Get info on a variable type	help, a	print a.__class__
Print value of variable	print, a	print a
Create an array	a = [2., 3.]	a = Float1d([2., 3.])
Create a list	-	a = [2., 3.]
Create an automatic array	a = findgen(10)	a = Float1d.range(10)
Get info on array variable	print, a	print a
Get one element of array	print,a(1)	print a[1]
Define new 1D array of 10 elements	a = fltarr(10)	a = Float1d(10)
Assign value inside an array	a(4) = 219	a[4] = 219
Define new 2D array of 5 rows, 10 columns	a = fltarr(10,5)	a = Float2d(5, 10)
First element index number	0	0
Pause time	wait	time.sleep()
Execute script	execute()	exec()

Plot commands	IDL	HIPE equivalent
Open a plotting window	window,retain = 2	p = PlotXY()
Plot two numeric arrays a & b	plot,a,b	p = PlotXY(a,b)

Plot commands	IDL	HIPE equivalent
Define axis ranges and styles	plot,a,b,[xy]range = [0.,10.], [xy]title = "lambda"	PlotXY(a,b,[xy]range = [0.,10.], [xy]title = "λ")
Define line style	plot,a,b,linestyle = 1	p.style.line = 2
Define plotting symbol	plot,a,b,psym = 2	p.style.symbol = 5
Define plot title	plot,a,b,title = 'title'	PlotXY(a,b,titleText = 'title')
Overplot	oplot,a,c	p[1] = LayerXY(a,c)
Make annotations	xyouts,0.2,0.7,Label'	d.addAnnotation("Label", 0.2, 0.7)
Save as postscript	set_plot,'ps' device,filename = "file.ps" device,/close set_plot,'X'	p.saveAsEPS("file.ps")
Save as JPG	-	p.saveAsJPG("file.jpg")
Save as PNG	-	p.saveAsPNG("file.png")
Save as PDF	-	p.saveAsPDF("file.pdf")
Further customisations	-	(right-click on plot and select *Properties*)

Data import/export commands	IDL	HIPE equivalent
Read a text table	readcol,'file.dat',a,b,c	t = simpleAsciiTableReader(file = "file.dat")
Plot read data	plot,a,b	p = PlotXY(t["c0"].getData(), t["c1"].getData()) or right-click on `t` and choose *Open with TablePlotter*
Read a comma separated table (.csv) text file	readcol,'file.csv', DELIMITER = ','	t = asciiTableReader(file = "file.csv")
Read a image FITS file	im = mrdfits("image.fits")	im = fitsReader(file = "image.fits")
Display an image	tvcsl,im	right-click on "im" and Open with "ImageViewer"
Read a cube FITS file	cube = mrdfits("cube.fits")	im = fitsReader(file = "cube.fits")
Display a cube	-	right-click on "cube" and Open with "CubeAnalysisToolbox"
Read a spectrum FITS file	sp = mrdfits("spec.fits")	sp = fitsReader(file = "spec.fits")
Display a spectrum	plot,wave,flux	right-click on "sp" and Open with "SpectrumExplorer"
Write to FITS	mwrfits,image,'image.fits'	simpleFitsWriter(product = image, file = "image.fits")
Write a text table (csv by default)	get_lun,u	asciiTableWriter(table = t, file = "file.csv")

Data import/export commands	IDL	HIPE equivalent
	openw,u,'file.csv' printf,u,a,b close,u free_lun,u	

Arithmetics commands	IDL	HIPE equivalent
Add	3 + 4	3 + 4
Multiply	3. * 4.	3. * 4.
Powers	3^4	3**4
Absolute value	abs()	absolute(), fabs()
Arc cosine	acos()	arccos()
Natural logarithm	alog()	log()
Base 10 logarithm	alog10()	log10()
Arc sine	asin()	arcsin()
Arc tangent	atan()	arctan()
Ceil	ceil()	ceil()
Conjugate	conj()	conjutage()
Cosine	cos()	cos()
Hyperbolic cosine	cosh()	cosh()
Exponential	exp()	exp()
Floor	floor()	floor()
Invert (matrix)	invert()	Matrix (module)
Bit shift operations	ishft()	right_shift(),left_shift()
Sine	sin()	sin()
Hyperbolic sine	sinh()	sinh()
Square root	sqrt()	sqrt()
Tangent	tan()	tan()
Hyperbolic tangent	tanh()	tanh()
Random 0-1 generator	randomu()	random()
Reverse array 'a'	reverse(a)	a[::-1]
Collapse array	total(a)	sum(a)
Number of elements	n_elements()	len(), size()
Number of parameters	n_params()	len(*args)
Extra parameters	_extra	**kwargs
Array size	size()	shape(),arrayvar.type()

These are external resources that you may find useful:

- IDL to Python
- Jython homepage

HIPE is based on Jython 2.5. External examples for different Jython/Python versions might not always work.

Chapter 2. Arrays, datasets and products

2.1. HIPE-specific data structures

The previous chapter described some data structures available in Jython, such as tuples and dictionaries. This chapter introduces other data structures available in HIPE, but that are not part of Jython:

- **Numeric arrays:** arrays of boolean, integer, floating point or complex values, from one to five dimensions. One of their major advantages with respect to Jython lists is the ability to do array arithmetic in single line commands rather than having to loop through arrays.
- **Datasets:** sets of arrays with additional information, such as a description of the content of each array, measurement units, and other *metadata*. Metadata consist of *keyword-value* pairs and have the same role as keywords and their values in the header of a FITS file.

  There are three types of datasets:

  - **Array datasets:** an array dataset contains a single numeric array.
  - **Table datasets:** a table dataset contains one or more numeric arrays organised in *columns*. The arrays can be of different types and sizes.
  - **Composite datasets:** a composite dataset is a container for other datasets, including other composite datasets.
- **Products:** sets of one or more datasets with additional metadata.
- **Contexts:** special products acting as containers for other products, including other contexts.

2.2. Numeric arrays

Numeric arrays available in HIPE are shown in the following table. A numeric array can have from one to five dimensions.

For completeness the following table also shows the `String1d` array type, which is not a numeric array. String arrays can only be one-dimensional.

Table 2.1. Types of numeric array (N = 1...5)

Name	Type
BoolNd	boolean
ByteNd	byte
ShortNd	short
IntNd	integer
LongNd	long
FloatNd	float
DoubleNd	double
ComplexNd	complex
String1d	string

Differences with Jython native arrays. Numeric arrays are optimised for holding Herschel data and for working with HIPE tasks and tools. Many tasks and functions accept Numeric arrays as their input, but not native Jython arrays. You are advised to use Numeric arrays when manipulating Herschel data.

2.2.1. Creating an array

See the following example for various ways in which you can create a one-dimensional array:

```
y = Double1d() # Create an empty array
y = Double1d([3.0, 5.5, 2.1, 6.0]) # Create from a Jython array
y = Double1d(4) # [0.0, 0.0, 0.0, 0.0]
y = Double1d(4, 42.0) # [42.0, 42.0, 42.0, 42.0]
y = Double1d.range(4) # [0.0, 1.0, 2.0, 3.0]
```

Example 2.1. Declaring an array of doubles.

You can create a complex array, with the same commands. In addition, you can specify real and imaginary parts separately:

```
y = Complex1d() # Create an empty array
y = Complex1d([1+4j, 2+3j, 5+8j]) # Create from a Jython array
y = Complex1d([1, 2, 5], [4, 3, 8]) # Same result as previous command
y = Complex1d(4) # [0j, 0j, 0j, 0j]
y = Complex1d(3, 1+4j) # [(1+4j),(1+4j),(1+4j)]
y = Complex1d.range(4) # [0j,(1+0j),(2+0j),(3+0j)
```

The following example shows how to create a two-dimensional array. You can create arrays of more dimensions in the same way:

```
y = Double2d() # Create an empty array
y = Double2d([[3.0, 5.5], [2.1, 6.0]]) # Create from a Jython array
y = Double2d(4, 4) # Creates a 4x4 array whose values are all zero.
y = Double2d(4, 4, 42.0) # Creates a 4x4 array whose values are all 42.
```

Example 2.2. Declaring a two-dimensional array of doubles.

Rectangular and jagged arrays. Rectangular arrays are multidimensional arrays which always have the same number of elements in each row or column. Jython and Java allow you to create *jagged arrays*. Jagged arrays are multidimensional arrays where each row can have a different number of elements. The following example creates a two-dimensional jagged array, with two rows of two and three elements, respectively:

```
x = [[1,2], [3,4,5]]
```

Example 2.3. Creating Jython jagged arrays.

However, you cannot create Numeric jagged arrays:

```
x = Double2d([[1,2], [3,4,5]]) # Gives an error
```

Example 2.4. It is impossible to create Numeric jagged arrays.

2.2.2. Inspecting an array

Array elements along each dimension are defined by an *index* running from zero to the array length, minus one:

```
x = Double1d([3.0, 5.0, 9.0, 4.3])  # Four elements
print x[0]  # Prints the first element, 3.0
```

```
print x[3]  # Prints the fourth element, 4.3
```

Example 2.5. Accessing array elements using the indices.

To access ranges of elements, or *slices*, use a notation like `[i:j]`:

```
print x[1:3]  # From index 1, included, to 3, excluded
# [5.0,9.0]
print x[2:]  # From index 2, included, to end of array
# [9.0,4.3]
print x[:3]  # From start of array to index 3, excluded
# [3.0,5.0,9.0]
```

Example 2.6. Using array slices to access ranges from Jython lists.

To access elements in multi-dimensional arrays, separate indices or ranges along each dimension with commas:

```
x = Int2d([[1,2,3],[4,5,6]])
#   1 2 3
#   4 5 6
print x[1]                # 2 (second element of the first row)
print x[0,:]              # Row 0: [1,2,3]
print x[1,1]              # Individual element: 5
print x[:,:]              # Print entire array. Same as print x
print x[:,1]              # Column 1: [2,5]
```

Example 2.7. Accessing ranges of indices using slices.

Multi-dimensional arrays are conceptually arrays of lower-dimensional arrays. For a two-dimensional array, the first subscript selects a row and the second subscript selects an element within that row (the column).

Note

This is the opposite order to some other computer languages, but it is the same behaviour as in the Java programming language.

Note the difference in syntax when inspecting native Jython arrays and numeric arrays:

```
# Jython array:
x = [[1,2,3,4],[5,6,7,8]]
print x[1][2]       # 7
print x[1][1:3]     # 6, 7
# Numeric array:
y = Int2d([[1,2,3,4],[5,6,7,8]])
print y[1,2]        # 7
print y[1,1:3]      # 6, 7
```

Example 2.8. Checking the differences between Jython arrays and numeric arrays.

2.2.3. Inspecting a complex array

Complex arrays offer the following additional commands:

```
z = Complex1d([1,2,3,4],[4,3,2,1]) # Set up complex array
print z.real # [1.0,2.0,3.0,4.0]
print z.imag # [4.0,3.0,2.0,1.0]
print z.conjugate() # [(1.0-4.0j),(2.0-3.0j),(3.0-2.0j),(4.0-1.0j)]
```

Example 2.9. Inspecting and manipulating a Complex Numeric array.

2.2.4. Modifying an array

You can append single values or entire arrays to an existing array:

```
y = Double1d()
y.append(2.0)  # Append a single value
y.append(Double1d([3.0, 7.5, 2.8]))  # Append a whole array
```

Example 2.10. Appending values to an array.

Individual elements or slices can be set as follows:

```
x[1,2] = 22 # Set an element in place
x[0,1:3] = 42
print x # [
        # [2.0,42.0,42.0],
        # [1.0,3.0,22.0]
        # ]
```

Example 2.11. Assigning values with the use of indices and slice notation.

It is possible to set a row to a copy of a 1d array of the same length:

```
x[0,:] = [5,6,7,8]        # Set a row to (a copy of) a Jython array
y[1,:] = Int1d([9,7,6,5]) # Set a row to a Double1d array
```

Example 2.12. Assign arrays to arrays using slice notation.

2.2.5. Ordering of array elements

The following line of code creates an array of two rows and three columns:

```
x = Double2d([[2,4,6],[1,3,5]])
```

Example 2.13. Declaring multidimensional Numeric array.

You can access the element corresponding to the *i*-th row and *j*-th column like this:

```
x[i, j]
```

Example 2.14. Accessing elements in multidimensional arrays.

The values are stored sequentially in memory as follows:

```
[2 4 6 1 3 5]
```

This means that, if you go through the array elements as they are stored in memory, their indices would vary as follows:

```
x[0,0] x[0,1] x[0,2] x[1,0] x[1,1] x[1,2]
```

That is, index *j* varies *more rapidly* than index *i*. This can be generalised to more than two dimensions by saying that *the rightmost index varies most rapidly*. This is called *row-major* ordering, and is the convention followed by languages such as Java and C, *but not Fortran*.

This has an implication on performance. When looping through a multidimensional array, it is more efficient to read its elements in the order they are stored in memory.

Confusion may arise when dealing with images, which are stored as two-dimensional arrays. If you visualise the array with horizontal rows and vertical columns, then the number of rows and columns represents the size of the vertical (y) and horizontal (x) side of the image, respectively. When accessing a particular pixel (array element), you have to specify the *y* coordinate *before* the *x* coordinate:

```
myImage(y, x)
```

2.2.6. Numeric array arithmetic

HIPE numeric arrays support arithmetic operations that are applied element-by-element. For example:

```
y = Double1d.range(5)                   # [0.0,1.0,2.0,3.0,4.0]
print y * y * 2 + 1                     # [1.0,3.0,9.0,19.0,33.0]
```

Example 2.15. Applying multiplication and addition to all elements of an array.

This is much simpler (and runs much faster) than writing an explicit loop in Jython. **The '+' operator does not concatenate arrays, as it does with Jython arrays.** For example:

```
# Adding Jython arrays
print [0,1,2,3] + [4,5,6,7]                   # [0, 1, 2, 3, 4, 5, 6, 7]

# Adding numeric arrays
print Double1d([0,1,2,3]) + Double1d([4,5,6,7])       # [4.0,6.0,8.0,10.0]

# Concatenate two numeric arrays
print Double1d([0,1,2,3]).append(Double1d([4,5,6,7]))
# [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]

# Adding Jython arrays to numeric arrays
print [0,1,2,3] + Double1d([4,5,6,7])         # [4.0,6.0,8.0,10.0]
print Double1d([0,1,2,3]) + [4,5,6,7]         # [4.0,6.0,8.0,10.0]
```

Example 2.16. Concatenating numeric arrays.

All arrays support the following arithmetic operators:

```
+, -, *, /, %, **
```

Note that the 'modulo' operator '%' provides the normal Jython semantics for this operation, which is not the same as that of the Java '%' operator. The Jython definition is more consistent with the mathematical notion of congruence for negative values.

The following relational operators return a `Bool1d` array:

```
<, >, <=, >=, ==, !=
```

For example:

```
y = Double1d([0,1,2,3,4])
print y > 2                          # [false,false,false,true,true]
```

Example 2.17. Applying relational operators to a Numeric array.

2.2.7. Selecting and filtering array values

Use the `where` function to select the elements of an array that satisfy a given condition:

```
y = Double1d([2,6,3,8,1,9])
print y.where(y > 4) # [1,3,5] Indices of elements greater than four
```

Example 2.18. Filtering array elements with the where method.

Note that the result contains the *indices* of the elements that satisfy the condition, not the elements themselves. You can combine more conditions with the `&` (and), `|` (or) and ~ (not) operators:

```
print y.where((y > 4) & (y < 9))  # [1,3]
print y.where((y > 8) | (y < 2))  # [4,5]
```

```
print y.where(~(y > 4))  # [0,2,4]
```

Example 2.19. More complex filtering using the where method.

You can obtain the actual array elements instead of the indices as follows:

```
print y[y.where(y > 4)]  # [6.0,8.0,9.0]
print y.get(y > 4)  # [6.0,8.0,9.0]
```

Example 2.20. Accessing the array values with a filter.

Obtaining the indices rather than the actual values can be useful, for instance, when you want to select the same elements from different arrays:

```
x = Double1d([5,6,7,8,9,10])
s = y.where(y > 4)
print x[s] + y[s]  # [12.0,16.0,19.0]
```

Example 2.21. Adding two arrays with the same set of filtered array indices.

You can also use the `where` function to set values:

```
s = y.where(y > 4)
y[s] = 0 # Set all matching elements to 0
print y  # [2.0,0.0,3.0,0.0,1.0,0.0]
y[s] = [9,8,7] # Set matching elements using an array
print y  # [2.0,9.0,3.0,8.0,1.0,7.0]
```

Example 2.22. Assigning values with the results of the where method.

You cannot use the `where` function like this:

```
a = Double1d.range(10)
b = a.where(a < 3)
print b[0]  # AttributeError: __getitem__
print b[0:2]  # AttributeError: __getitem__
print a[b[0]]  # AttributeError: __getitem__
```

Example 2.23. The output list of where is not accessible by index.

The commands fail because `b` is a `Selection` object rather than a Jython or Numeric array. For the above to work you need to convert it to `Int1d`:

```
c = b.toInt1d()
print c[0]  # 0
print c[0:2]  # [0,1]
print a[c[0]]  # 0.0
```

Example 2.24. Converting the output of where to a normal array that you can manipulate.

By converting to `Int1d` you can also loop over all the selected elements:

```
for i in c:
  print a[i]
```

Example 2.25. Converting the output of where makes the resulting object iterable.

Another useful function is `get`, with which you can extract individual elements or a subset of element values from an array. There are four ways to use it:

- Get a single value:

```
HIPE> print y.get(0)
```

```
2.0
```

This is the same as `print y[0]`.

- Retrieve elements based on a `Bool1d` array (in other words, a mask):

```
HIPE> mask = Bool1d([0,0,1,0,1,1])
HIPE> print y.get(mask)
[3.0,1.0,9.0]
```

The mask array can be shorter than the target array (in which case the remaining elements in the target array will be ignored), but not longer.

- Retrieve elements based on a `Selection` object:

```
HIPE> indices = Selection([1,2,4])
HIPE> print y.get(indices)
[6.0,3.0,1.0]
```

An out of bounds index will cause an error.

- Retrieve elements based on a `Range` object:

```
HIPE> range = Range(2,4)
print y.get(range)
[3.0,8.0]
```

Note how the lower boundary (second element) is *included*, while the upper boundary (fourth element) is *excluded*. An out of bounds value will cause an error.

You can combine `get` calls to perform the same operation as a compound IDL `WHERE` execution such as the one shown in this example:

```
IDL> a = [1, 2, 3, 4, 5, 6]
IDL> b = [2, 3, 4, 5, 6, 7]
IDL> c = [3, 4, 5, 6, 7, 8]
IDL> q =  WHERE(a ge 2 and b lt 6 and c gt 5)
IDL> x = [a[q],b[q],c[q]]
IDL> print, x
       4       5       6
```

This is the equivalent in HIPE:

```
q = (a >= 2) & (b < 6) & (c > 5)
x = a.get(q),b.get(q),c.get(q) # x == ([4.0], [5.0], [6.0])
```

```
HIPE> a = Int1d([1, 2, 3, 4, 5, 6])
HIPE> b = Int1d([2, 3, 4, 5, 6, 7])
HIPE> c = Int1d([3, 4, 5, 6, 7, 8])
HIPE> q = (a >= 2) & (b < 6) & (c > 5)
HIPE> x = a.get(q),b.get(q),c.get(q)
HIPE> print x
    ([4], [5], [6])
```

2.2.8. Using logical operators with arrays

The **Jython logical operators** `and`, `or` and `not` work like normal Boolean operators (see Appendix A for more details), but using them with arrays (both the native Jython arrays and HIPE numeric arrays) can give unexpected results. The reason is that these operators do *not* work on an element-by-element basis when applied to arrays, but they evaluate the entire array at once.

The Jython **bitwise operators**, represented by the symbols `&`, `|` and `^` (see Appendix A for more details) can be used with numeric arrays (`Int1d`, `Bool3d` and so on), but what you get is *not* a

bitwise comparison. Instead, these operators perform the usual boolean comparisons, but this time working element by element. Precisely what `and`, `or` and `not` do *not* do.

Finally, Numeric array classes have the `and`, `or` and `xor` *methods* acting like boolean *operators* working element by element. An example will clarify the differences among all these operators:

```
jythonOne = [1, 0, 0, 1]
jythonTwo = [0, 0, 1, 1]
numericOne = Bool1d(jythonOne)
numericTwo = Bool1d(jythonTwo)
print jythonOne and jythonTwo
## [0, 0, 1, 1] # jythonOne is not empty so it is treated as true, which means that
                # jythonTwo is evaluated and returned
print numericOne and numericTwo
## [false,false,true,true] # Same thing as with the Jython native arrays
print jythonOne & jythonTwo
## Here an error is returned
print numericOne & numericTwo
## [false,false,false,true] # Here the operator works element by element
print numericOne.and(numericTwo)
## [false,false,false,true] # Same thing as the & operator
```

Example 2.26. Differences between Jython and Numeric arrays.

2.2.9. Removing infinite and NaN values from arrays

To remove infinite and NaN values from an array, use the `IS_FINITE`, `IS_INFINITE` and `IS_NAN` functions as shown in the following examples.

First example:

```
# Set up an array with non-finite elements
x = Double1d.range(5)+1
x[0] = Double.NaN
x[3] = Double.POSITIVE_INFINITY
print x     # [NaN,2.0,3.0,Infinity,5.0]

# Keeping finite values
q = x.where(IS_FINITE)
print q     # [1,2,4]
print x[q]  # [2.0,3.0,5.0]
```

Example 2.27. Removing infinite and NaN values from an array.

Second example:

```
# Creating a mask
mask = IS_FINITE(x)
print mask  # [false,true,true,false,true]

q = x.where(mask)
print q     # [1,2,4]        (as before)
print x[q]  # [2.0,3.0,5.0]  (as before)

# Asking to filter the non-finite numbers:
q = x.where(~mask)
print q     # [0,3]
print x[q]  # [NaN,Infinity]

# Replacing selected values
x[q] = 0
print x     # [0.0,2.0,3.0,0.0,5.0]
```

Example 2.28. Creating a filter (mask) with a function ro remove NaNs.

2.2.10. Advanced tips for improved performance

The underlying array operations and functions are very fast, as they are implemented in Java. The overhead of invoking them from Jython is relatively small for large arrays. However, the advanced user may find the following tips useful to improve performance in cases where it becomes a problem.

The arithmetic operations, such as '+', have versions that allow in-place modification of an array without copying. For example:

```
y = Double1d.range(10000)
y = y + 1 # The array is copied
y += 1 # The array is modified in place
```

Example 2.29. Avoiding unnecessary array allocation for the addition operation.

Copying an array is slow as it involves allocating memory (and subsequently garbage collecting it). For simple operations, such as addition, the copying can take longer than the actual addition.

Function application also involves copying the array. This can be avoided by using the Java API instead of the simple prefix function notation. For example:

```
x = Double1d.range(10000)
x = SIN(x) * COS(x) # This operation involves three copies
x = x.apply(SIN).multiply(x.apply(COS)) # Only one copy
```

Example 2.30. Using Java array utility methods to avoid wasteful array allocation.

When writing array expressions, it is better to group scalar operations together to avoid unnecessary array operations. For example:

```
y = Double1d([1,2,3,4])
print y * 2 * 3 # 2 array multiplications
print y * (2 * 3) # 1 array multiplication
print 2 * 3 * y # 1 array multiplication
```

Example 2.31. Grouping scalar multiplication avoids costly array multiplication.

It is better to avoid explicit loops over the elements of an array. It is often possible to achieve the same effect using existing array operations and functions. For example:

```
sum = 0.0
for i in y:
  sum = sum + i * i # Explicit iteration

sum = SUM(y * y) # Array operations
```

Example 2.32. Using arithmetic operations on arrays to avoid loops.

2.2.11. Type conversions

Since the numeric library supports different types it would be very convenient to be able to convert an array from one type to another. The numeric library supports both implicit conversion from within Jython for all supported dimensions and explicit conversion from one data type to another.

2.2.11.1. Explicit conversion

Explicit conversion is supported for all data types by constructing a numeric array from another numeric array of the same or a different type. Note however that some explicit conversions may result in rounding and/or truncation of the values e.g. an explicit conversion from Long1d to Double1d will reduce the number of significant digits.

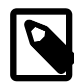
```
i = Int1d([1,2,3])                    # [1,2,3]
r = Double1d(i)                       # [1.0,2.0,3.0]
```

```
c = Complex1d(r)                          # [(1.0+0.0j),(2.0+0.0j),(3.0+0.0j)]
b = Byte1d(r)                             # [1,2,3]
```

Example 2.33. Converting types explicitly requires the creation of a numeric array of a specific type.

2.2.11.2. Implicit conversion

Implicit conversions are conversions that can be done automatically, provided that such a conversion is a widening operation, for example. from Int1d to Double1d. Implicit narrowing conversions are not allowed and result in an error message.



Note

A widening operation is when the result is stored in more bits than the source, thus not losing any accuracy. A narrowing operation is the opposite and data loss is possible as the result is stored using fewer bits than the source.

The library supports implicit conversions in the following cases:

- access: [...]
- operators: +, -, *, /, ^ and %
- in-line operators: +, -, *, /, ^ and %

The examples below show allowed implicit conversions.

```
d = Double1d(5)                           # [0.0,0.0,0.0,0.0,0.0]
d[1] = 3                                  # [0.0,3.0,0.0,0.0,0.0]
d[1:4] = [-5, 0, 5]                       # [0.0,-5.0,0.0,5.0,0.0]
```

Example 2.34. Converting types implicitly in Jython.

HIPE considers the conversion from int to float and from long to float/double as an automatic widening operation, but some of the least significant digits of the value may be lost during the conversion. You will not be notified of this loss of significant digits.

Another thing to notice is that floating point operations will never throw an exception or error. As shown in the following example, a division by zero results in NaN or Infinity.

```
d = Double1d.range(5)
l = Long1d.range(5)
print d/l                            # [NaN,1.0,1.0,1.0,1.0]
print d/SHIFT(l, 1)                  # [0.0,Infinity,2.0,1.5,1.3333333333333333]
```

Example 2.35. Dividing by zero will generate NaN or Infinity as appropriate.

2.3. Array datasets

2.3.1. Creating an array dataset

The following example shows the available commands to create an array dataset:

```
x = ArrayDataset() # Create an empty dataset
# Create an array dataset with an embedded array
array = Double1d([1.0, 5.3, 2.1])  # Create the array first
x = ArrayDataset(array)  # Create the array dataset
# Create an array dataset with measurement units and a description
from herschel.share.unit import *
x = ArrayDataset(array,  Power.WATTS, "My measurements")
```

Example 2.36. Declaring an array dataset.

See Section 2.6 for more information about measurement units.

2.3.2. Modifying an array dataset

The following example shows various ways of modifying an array dataset:

```
# Change the description
x.description = "New description"
# Change the measurement unit
from herschel.share.unit import *
x.unit = Mass.GRAMS
# Change the data
x.data = Int1d([2,5,6,4])
# Change one element of the data
x.data[1] = 7
```

Example 2.37. How to modify an array dataset.

See Section 2.7.1 for how to change the array dataset metadata.

2.3.3. Inspecting an array dataset

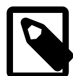
The following example shows various ways of inspecting an array dataset:

```
print x.description # Print the description
print x.unit  # Print the measurement unit
print x.hasData()  # Check if the array dataset contains data (returns 1 if yes)
print x.data  # Print only the data contained in the dataset
print x.data[2] # Print a single element of the data
```

Example 2.38. Accessing relevant data of an array dataset.

2.4. Table datasets

A table dataset is made up of a number of columns. Each column contains an array (data), a description and a measurement unit.



Note

For reasons of flexibility, memory consumption and performance, this class is not checking whether all columns are of the same length: this is your responsibility.

2.4.1. Creating a table dataset

The following example shows how to create a table dataset:

```
t = TableDataset()  # Empty table dataset, no description
t = TableDataset(description = "My dataset")  # Empty table dataset, with
 description
```

Example 2.39. Creating a table dataset.

You can only create empty table datasets and fill them later with columns.

Creating columns is very similar to creating array datasets:

```
c = Column() # Create an empty dataset
# Create a column with an embedded array
array = Double1d([1.0, 5.3, 2.1])  # Create the array first
c = Column(array)  # Create the array dataset
# Create a column with measurement units and a description
```

```
from herschel.share.unit import *
c = Column(array, Power.WATTS, "My measurements")
```

Example 2.40. Creating isolated columns.

See Section 2.6 for more information about measurement units.

The following example shows how to add columns to a table dataset:

```
# Add column c with name Measurements
t["Measurements"] = c
# You can create and add a column in one step
t["secondColumn"] = Column(Int1d([1,2,3,4,5]))
```

Example 2.41. Adding data columns to table datasets.

Warning

Do not use the same column multiple times in a table dataset:

```
col = Column(Double1d(10))  # Column holding ten double numbers
t = TableDataset()
t["First"] = col
t["Second"] = col  # Using "col" twice
```

Example 2.42. Avoiding references to the same data from two different columns.

The two columns in the table dataset have different names, but they both refer to the `col` variable, so any change to one column will appear in the other one.

To properly define independent columns, create a different variable for each column:

```
col1 = Column(Double1d(10))  # Column holding ten double numbers
col2 = Column(Double1d(10))
t = TableDataset()
t["First"] = col1
t["Second"] = col2  # Using different variables
```

Example 2.43. How to correctly create independent columns in a table dataset.

You can also create the columns inside the table dataset without defining separate variables:

```
t = TableDataset()
t["First"] = Column(Double1d(10))  # Creating and adding columns
 in one step
t["Second"] = Column(Double1d(10))
```

Example 2.44. Adding columns directly to a table dataset.

2.4.2. Modifying a table dataset

The following example shows how to modify a table dataset:

```
# Changing the description
t.description = "New description"
# Substituting a column (same as adding a column)
t["Measurements"] = Column(myOtherArray)
# Changing column data
t["Measurements"].data[0] = 2 # First element set to 2
# Changing a specific data element
t.setValueAt(12, 0, 1) # Set the value 12 at row = 0, column = 1
# Removing a column by name
t.removeColumn("myColumn") # Remove column called myColumn
# Removing a column by index
t.removeColumn(0) # Remove the first column
```

```
# Adding a row, assuming the table dataset has two integer
# and one floating point column
t.addRow([5, 9, 3.2])
# Removing a row
t.removeRow(1) # Remove the second row
```

Example 2.45. Exercising some of the most useful methods of a table dataset.

2.4.3. Copying a table dataset into another

You can append data from one table dataset to another, provided that they have the same number of columns and each column in either dataset is of the same type. The following example adds `t2` as rows to table `t1`.

```
# Create two compatible table datasets
t1 = TableDataset()
t1["x"] = Column(Int1d.range(5))
t1["y"] = Column(Double1d.range(5))
t2 = TableDataset()
t2["a"] = Column(Int1d.range(10))
t2["b"] = Column(Double1d.range(10))

# Append the data in t2 to the data in t1
t1.addRowsByIndex(t2)
```

Example 2.46. Copying table datasets.

2.4.4. Inspecting a table dataset

The following example shows various ways of inspecting an array dataset:

```
print t.description # Print the description
print t.columnCount # Print the number of columns
print t.rowCount # Print the number of rows
print t.getColumnName(i) # Print the name of the column at integer position i
print t.columnNames # Print a vector with the names of all the columns
print t[i] # Print the column at integer position i
print t["name"] # Print the column called name
print t["name"].data # Print only the data inside the column called name
print t["name"].data[2] # Print a specific value of the data
print t["name"].unit # Print the unit of the column called name
print t["name"].description # Print the description of the column called name
print t.getRow(i) # Print a vector containing the values of the row at integer
 position i
print t.getValueAt(0,1) # Print the value contained in row = 0, column = 1
```

Example 2.47. Exercising the most useful methods of an array dataset.

If the table dataset has columns of different lengths, `getRow` gives an error if some of the row values are undefined.

2.5. Composite datasets

A composite dataset is a container in which you can place other datasets (array or table), including other composite datasets.

2.5.1. Creating a composite dataset

This example shows how to create a composite dataset:

```
# Create an empty composite dataset
c = CompositeDataset()
# Create an empty composite dataset with a description
```

```
c = CompositeDataset("My dataset")
```

Example 2.48. Creating a composite dataset.

Adding datasets to a composite dataset follows the same syntax as adding columns to a table dataset:

```
# Assume that t is a table dataset and c is a composite dataset
c["measurements"] = t  # Adding t to c with the name measurements
```

Example 2.49. Adding a table dataset to a composite dataset.

2.5.2. Modifying a composite dataset

This example shows how to modify a composite dataset:

```
# Set the description
c.description = "My composite dataset"
# Replace the dataset called coordinates with the one held in variable t
c["coordinates"] = t
# Remove a dataset
c.remove("coordinates")
# Extract a copy of a dataset to a variable
c["measurements"] = myData  # Dataset called measurements now held in myData
# Changes to myData will also affect the composite dataset
```

Example 2.50. Manipulating a composite dataset.

To change a dataset name, remove it and add it again with the new name.

2.5.3. Inspecting a composite dataset

This example shows how to inspect a composite dataset:

```
print c.description # Print the description
print c.keySet() # Print a vector with the names of all the datasets
print c["name"] # Print the dataset called name
print c["name"].data # Print only the data inside the dataset called name
print c["name"].description # Print the description of the dataset called name
```

Example 2.51. Exercising the most useful methods of a composite dataset.

2.6. Measurement units

You can assign measurement units to array datasets and columns in a table dataset. Different columns in the same table dataset may have different units.

To use units in your scripts, you have to manually import the unit package with the following command:

```
from herschel.share.unit import *
```

The following table shows the measurement units available in HIPE.

Note

Constants describing the units are given in American English spelling (e.g.: meter versus metre) because that is the way they are stored within the software.

Type	VALUES
Acceleration	METERS_PER_SECOND_SQUARED
Angle	RADIANS, DEGREES, MINUTES_ARC, SECONDS_ARC

Type	VALUES
AngularMomentum	JOULE_SECOND
AngularSpeed	RADIANS_PER_SECOND, DEGREES_PER_SECOND
Area	SQUARE_METERS, SQUARE_KILOMETERS
Constant	H_PLANCK, K_BOLTZMANN, ELECTRON_CHARGE, SPEED_OF_LIGHT
Duration	SECONDS, MINUTES, HOURS, DAYS
ElectricCapacitance	FARADS, MILLIFARADS, MICROFARADS, NANOFARADS, PICO-FARADS
ElectricCharge	COULOMBS
ElectricConductance	SIEMENS
ElectricCurrent	AMPERES, MILLIAMPERES
ElectricInductance	HENRIES
ElectricPotential	VOLTS, MILLIVOLTS
ElectricResistance	OHMS
Energy	JOULES, ERGS, ELECTRON_VOLTS
Entropy	JOULES_PER_KELVIN
Flux density	JOULES_PER_SQUARE_METER, JANSKYS, MILLIJANSKYS, MI-CROJANSKYS
Force	NEWTONS, DYNES
Frequency	HERTZ, KILOHERTZ, MEGAHERTZ, GIGAHERTZ, TERAHERTZ
Length	METERS, ANGSTROMS, KILOMETERS, CENTIMETERS, MIL-LIMETERS, MICROMETERS
Mass	GRAMS, KILOGRAMS
NEP (Noise Equivalent Power)	WATTS_PER_SQRT_HERTZ
Power	WATTS, KILOWATTS, MEGAWATTS
Pressure	PASCALS, BARS, MILLIBARS
Scalar	This class represents scalar units and provides some constants:ONE, PER-CENT,DECIBELS
SolidAngle	STERADIANS, SQUARE_MINUTES_ARC, SQUARE_SECOND-S_ARC
Speed	KILOMETERS_PER_SECOND, METERS_PER_SECOND
Temperature	CELSIUS, KELVIN
ThermalConductivity	WATTS_PER_METER_KELVIN
TimeInstant	TAI, UTC
WaveNumber	RECIPROCAL_METERS, RECIPROCAL_CENTIMETERS

2.6.1. Creating and assigning units

To create a variable representing a unit, specify the type and value of the unit, separated by a dot, as shown in the following example:

```
a = Angle.DEGREES  # Type Angle, value DEGREES
b = Temperature.KELVIN
```

Example 2.52. Assigning units to variables.

You can then assign the unit to an array dataset or to a column of a table dataset. You can use a variable representing the unit, or specify the unit definition:

```
myTableDataset["x"].unit = Angle.DEGREES  # Assign to a column, specify unit
 definition
myArrayDataset.unit = b  # Assign to array dataset, use existing variable
```

Example 2.53. Assigning units to columns or datasets.

Warning

If you change the unit assigned to a dataset, this will have no effect on its values. For example, changing a unit from metres to centimetres will not result in the values being multiplied by 100. For a discussion on how to properly convert the units of a quantity, affecting the values, see this chapter in the Data Analysis Guide in *Data Analysis Guide*.

2.6.2. Obtaining derived units

You can obtain derived units with simple expressions, as shown in the following example:

```
N   = Force.NEWTONS
m   = Length.METERS
m2  = m**2              # Square meters
Pa  = N / m2            # Pascals
J   = N * m             # Joules
```

Example 2.54. Creating a new, derived unit.

You can also obtain multiples and fractions of units by using functions such as `.milli()`, `.mega()`, `.kilo()`:

```
s   = Duration.SECONDS
us  = s.micro()  # microseconds
ns  = s.nano()   # nanoseconds
```

Example 2.55. Converting units between standard SI prefixes.

All SI prefixes from *atto* (10^{-18}) to *exa* (10^{18}) are available. Fr a list of SI prefixes, see for example http://www.bipm.org/en/measurement-units/prefixes.html.

2.6.3. Converting units to and from strings

You can convert a unit variable to various string representations:

```
A = Length.ANGSTROMS
print A.name            # angstrom. ASCII characters only.
print A.dialogName      # Å.
um = Length.MICROMETERS
print um                # micrometer [1.0E-6 m]. Includes factor
                        # with respect to SI unit
print um.name           # micrometer. ASCII characters only.
print um.dialogName     # µm.
```

Example 2.56. Printing unit names for ASCII output or dialog output that includes symbols and Greek characters.

You can also convert a string to a unit variable:

```
myUnit  =  Unit.parse("km s-1")
myUnit  = (Unit.parse("km") / Unit.parse("s"))
myUnit  = Unit.parse("km s-1")
myUnit  = Unit.parse("arcsec")
myUnit  = Unit.parse("eV")
```

```
myUnit  = Unit.parse("cm")
myUnit  = Unit.parse("mm")
myUnit  = Unit.parse("microm")
```

Example 2.57. Parsing the string representation of the unit to assign it to a variable.

If you give an invalid string to the `parse` method, a unit is created anyway, but it will give an error if you try any operation on it.

2.6.4. Converting units to other units

To obtain the SI equivalent of a unit, use `.asSI`:

```
a = Angle.DEGREES # a is an angle in degrees
b = a.asSI  # b is an angle in radians
v1 = Speed.KILOMETERS_PER_HOUR # v1 is a speed in km/h
v2 = v1.asSI  # v2 is a speed in m/s
```

Example 2.58. Assigning units to variables.

To obtain the conversion factor between a unit and its SI equivalent, use `.toSI`:

```
a = Length.ANGSTROMS  # a is a length in angstroms
print a.toSI  # 1e-10
print Duration.HOURS.toSI            # 3600.0
print FluxDensity.MILLIJANSKYS.toSI  # 1.0E-29
print Unit.parse("g cm s-2").toSI    # 1.0E-5
```

Example 2.59. Retrieving the conversion factor to SI units.

To obtain conversion factor between two units, use `to`:

```
min = Duration.MINUTES
ms  = Duration.MILLISECONDS
print min.to(ms)       # 60000.0
```

Example 2.60. Using the to method to explicitly convert units.

2.6.5. Comparing units for compatibility

You can compare units to see if they are of compatible types.

```
kg = Mass.KILOGRAMS
g  = Mass.GRAMS
m  = Length.METERS
print kg.isCompatible(g)                           # true
print kg.isCompatible(m)                           # false
print kg.isCompatible(Mass)                        # true
print kg.isCompatible(Area)                        # false
print Unit.parse("g cm s-2").isCompatible(Force)  # true
print Unit.parse("g cm s-2").isCompatible(Power)  # false
```

Example 2.61. Checking if two different units refer to the same physical quantity.

2.6.6. Comparing units for equivalence

You can use the `.isEquivalent` method to determine if two unit types are the same.

```
kg  = Mass.KILOGRAMS
s   = Duration.SECONDS
m   = Length.METERS
N   = Force.NEWTONS
dyn = Force.DYNES
print N.isEquivalent(dyn)            # false
```

```
print N.isEquivalent(kg * m / s**2)  # true
```

Example 2.62. Checking if two units are the same but expressed differently.

2.6.7. Obtaining physical and mathematical constants

You can obtain the following physical constants, complete with units:

```
h = Constant.H_PLANCK
print h.value  # 6.62606896E-34
print h.unit   # J s
print h        # 6.62606896E-34 J s
k = Constant.K_BOLTZMANN
c = Constant.SPEED_OF_LIGHT
e = Constant.ELECTRON_CHARGE
```

Example 2.63. Using physical constants provided within the Constant class.

The following mathematical constants are also available:

```
from java.lang.Math import PI
print PI # 3.141592653589793
from java.lang.Math import E
print E # 2.718281828459045
```

2.7. Metadata

Metadata contain additional information about a dataset. Metadata consist of couples of *keys* and *values*. A key describes a piece of information, while the value is the actual piece of information. For example, in the metadata entry `instrument = HIFI`, the key is `instrument` and the value is `HIFI`.

Metadata are automatically created whenever you create a dataset. For products, the metadata set is filled with a few compulsory entries. For other datasets, the metadata set is initially empty.

Metadata entries, or *parameters*, can be of the following types:

- `StringParameter`: strings of text.
- `BooleanParameter`: true/false values.
- `LongParameter`: integer numbers.
- `DoubleParameter`: floating point numbers.
- `DateParameter`: date values.

2.7.1. Modifying metadata

The following example shows how to add or modify the metadata of a dataset called `s`. If a metadata entry with a given key value already exists in the dataset, it is modified. If not, it is added.

```
s.meta["observation"] = StringParameter("NGC 4151")
s.meta["principal investigator"] = StringParameter("William Herschel")
# Set "date" as the current date and time
s.meta["date"] = DateParameter(FineTime(java.util.Date()))
s.meta["ra"] = DoubleParameter(182.836)
# Add a description
s.meta["dec"] = DoubleParameter(39.405, "Declination")
# Add a description and a unit
# Units are only available to DoubleParameter and LongParameter
from herschel.share.unit import *
```

```
s.meta["ra"] = DoubleParameter(182.836, "Right Ascension", Angle.DEGREES)
# Remove a key-value pair
s.meta.remove("dec")
```

Example 2.64. Adding and modifying metadata associated to a table dataset.

Date parameters are expressed as `FineTime` objects. For more information see Chapter 9.

2.7.2. Inspecting metadata

The following example shows how to inspect the metadata of a dataset called `s`:

```
# Print all the metadata
print s.meta
# Print a list of all the parameters in the metadata
print s.meta.keySet()
# Print a specific parameter.
print s.meta["observation"]
# Print the value of a specific parameter
print s.meta["observation"].value
# Print the description of a specific parameter
print s.meta["observation"].description
# Print the unit of a specific parameter (numeric parameters only).
print s.meta["wavelength"].unit
# Check if a key value is present. Returns True or False.
print s.meta.containsKey("obsid")
# Check if the metadata is empty. Returns True or False.
print s.meta.empty
```

Example 2.65. Inspecting the metadata of a dataset.

If you want to search the metadata of datasets held in a pool, use the Product Browser perspective. See the *Data Analysis Guide* for more information: Section 1.7 in *Data Analysis Guide*.

For more information on searching metadata from the command line, see Section 7.3.

2.8. Products

A product is an object containing a set of metadata entries and one or more datasets.

With respect to composite datasets, products offer more features, such as a history recording all the changes (see Section 2.8.7) and a set of metadata parameters added automatically whenever a product is created.

The automatic metadata values are `type`, `creator`, `creationDate`, `description`, `instrument`, `modelName`, `startDate`, `endDate` and `formatVersion`.

2.8.1. Creating a product

The following example shows how to create a product:

```
# Create an empty product
myProduct = Product()
# Create a product with a given description
myProduct = Product("This is my product")
# Create a product by specifying some of the compulsory metadata values
myProduct = Product(creator="Myself", instrument="SPIRE", \
description="An empty product", modelName="PFM", type="ABC")
```

Example 2.66. Creating an empty product with some metadata.

Adding datasets or other products to a product follows the same syntax as adding datasets to a composite dataset:

```
# Assume that t is a table dataset and p is a product
p["measurements"] = t  # Adding t to p with the name measurements
```

2.8.2. Modifying a product

The following example shows how to modify a product:

```
# Substituting an array/product (same as adding an array/product)
p["Measurements"] = myOtherDataset
# Removing an array/product
p.remove("Measurements")
```

Example 2.67. Overwriting an array within a dataset.

To change a dataset name, remove it and add it again with the new name.

There is a simplified syntax to set the compulsory metadata values:

```
myProduct.type = "ABC"
myProduct.creator = "Myself"
myProduct.description = "An empty product"
myProduct.instrument = "SPIRE"
myProduct.modelName = "PFM"  # And so on
```

Example 2.68. The most common metadata have attributes defined.

2.8.3. Setting date and time in product metadata

The `startDate`, `endDate` and `creationDate` are mandatory metadata parameters and are set to the current date and time when the product is created. To set any of these parameters to the system date and time, use these commands:

```
myProduct.creationDate = FineTime(java.util.Date())
myProduct.startDate = FineTime(java.util.Date())
myProduct.endDate = FineTime(java.util.Date())
```

Example 2.69. Some of the time attributes are instances of FineTime.

To set these parameters to an arbitrary date and time, expressed as UTC or TAI, use the following commands:

```
formatter = SimpleTimeFormat(TimeScale.UTC)
timeUtc = formatter.parse("2008-01-31T12:35:00.0Z")    # Z at the end is mandatory
 for UTC

formatter = SimpleTimeFormat(TimeScale.TAI)            # or just SimpleTimeFormat()
timeTai = formatter.parse("2008-01-31T12:35:00.0TAI")  # TAI at the end is mandatory
 for TAI

myProduct.creationDate = timeUtc # or
myProduct.creationDate = timeTai
```

Example 2.70. Creating TAI or UTC time strings to set time metadata.

Note that the two previous dates, represented as `FineTime`, are different:

```
HIPE> print timeUtc
2008-01-31T12:35:33.000000 TAI (1580474133000000)
HIPE> print timeTai
2008-01-31T12:35:00.000000 TAI (1580474100000000)
```

See Section 9.1 for more information on representing time in the Herschel software.

2.8.4. Inspecting a product

The following example shows how to inspect a product:

```
print myProduct.description  # Print the description of a product
# You can use the above syntax for any of the compulsory metadata. For example:
print myProduct.type
print myProduct.creationDate  # And so on
print myProduct.isEmpty()  # Check whether a product is empty. Returns True or
 False.
print myProduct.sets.size()  # Print the number of dataset/products in the product.
print myProduct.keySet() # Print a vector with the names of all the datasets and
 products in this product
print myProduct.default # Print the first inserted dataset.
print myProduct["anotherDataset"]  # Print the dataset called anotherDataset.
```

Example 2.71. Inspecting an object from a subclass of Product.

Instead of just printing out the datasets you get, you can assign them to variables and execute other operations on them. To see how to explore the contents of datasets please refer to the previous sections of this chapter.

See the *HIPE Owner's Guide in HIPE Owner's Guide* for ways of inspecting a product via a graphical interface.

2.8.5. Product contexts

Contexts are special types of products that contain references to other products. Contexts are used to organise related products into a coherent structure. For example, the observation context in *HCSS User's Reference Manual* contains all the data related to an observation.

There are two standard types of context products provided: ListContext in *HCSS User's Reference Manual*, for grouping products into sequences or lists, and MapContext in *HCSS User's Reference Manual*, for grouping products into containers with access to each by key. The observation context is a map context.

Most likely you will not have to create your own contexts, but just access those you retrieve from the Herschel Science Archive.

2.8.6. Observation contexts

Every observation you download from the Herschel Science Archive is an *observation context*. An observation context contains other contexts and many data products. While the high-level structure of observation contexts is broadly the same, the detailed structure and product types vary according to the observation type. See the following resources for more information on observation contexts:

- For a brief introduction to observation contexts see the *Products Definitions Document*: Section 2.9 in *Product Definition Document*.
- For more information on HIFI observation contexts, see the *HIFI Data Reduction Guide*.
- For more information on PACS observation contexts, see the *PACS Products Explained*.
- For more information on SPIRE observation contexts, see the *SPIRE Data Reduction Guide*.

2.8.7. Product history

The Product history is generated or updated whenever a task is run on a product. It contains the tasks which have been run to generate the product (including used parameters), as well as the used calibration files and the track and build number of the used build.

You can retrieve the history of a product in Jython as follows:

```
history = product.history
```

A simple `print` shows which tasks, build numbers and calibration files have been used:

```
HIPE> print history
```

Additional interesting functionalities of the history list are the following:

- Get the history as a Jython script:

  ```
  script = product.history.script
  product.history.saveScript("script.py")
  ```

  Example 2.72. Saving the product history to a script file.

- Find out if a certain task has been run (useful in a task which depends on another task):

  ```
  if not product.history.isTaskPerformed("someTask"):
        print "You have to run someTask first!"
        exit
  ```

  Example 2.73. Checking if a task has been executed on a product (either locally or as part of standard processing).

Calibration file appear in the history with identifiers constructed from meta keywords. How this is done depends on the instrument:

- For PACS the identifier is constructed from the meta keywords `calFileId`, `modelName` and `calFileVersion` as follows:

  ```
  Common|Photometer|Spectrometer_calFileId_modelName_calFileVersion
  ```

  Also, if you want to introduce your own handcrafted PACS calibration file you should change especially the `calFileId` keyword to make sure that this is visible in the history.

- For SPIRE the `fileName` meta keyword is used to identify calibration files.

- For HIFI no standard has been implemented yet.

Chapter 3. Spectra and spectral cubes

3.1. Spectrum containers and segments

Within HIPE there are several types of *datasets* and *products* that are used to contain spectra. These consist of the data and metadata that are necessary for correct handling of spectral display and combination (e.g. spectral arithmetic).

The available spectral *datasets* are `Spectrum1d` and `Spectrum2d`. These are extensions of the generic Herschel TableDataset, and are made up of columns containing arrays of numbers. A Product is a wrapper which can contain one or more Datasets collected together with metadata (header) - see Section 2.1 for more details on datasets and products. Examples of spectral *Products* are `SimpleSpectrum` and `SpectralSimpleCube`.

In general, most of the datasets and products dealing with spectra inside HIPE make use of the "SpectrumContainer" interface. This interface specifies the way that the spectral data are accessed so that the same tasks (e.g. those in the Spectrum Toolbox) can operate on all of the different types of spectral datasets and products.

A "SpectrumContainer" is built up of several layers. It can contain one or several "PointSpectrum" objects, with each PointSpectrum consisting of one or several "SpectralSegments".

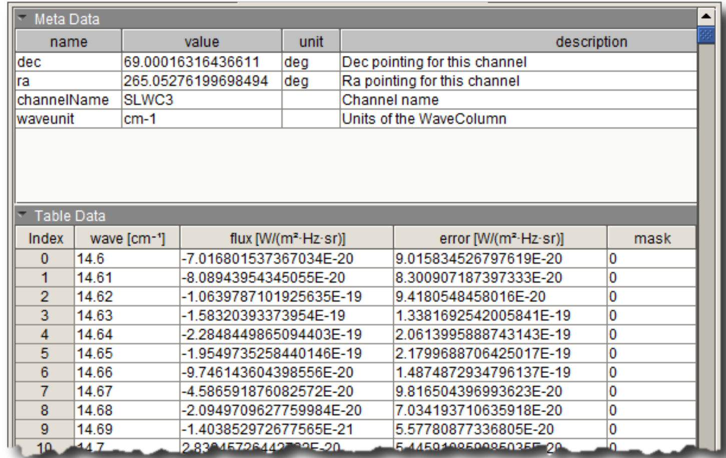
- The **SpectralSegment** is the most basic unit of spectral data, with a frequency, wavelength or velocity array, a flux array and, optionally, weights, errors, flags and masks.
- The **PointSpectrum** contains individual segments, and may also contain additional parameters that hold for all of its segments. Examples are integration time, observation time, position on sky or other instrument specific information (e.g. house-keeping data).

In general, the full complexity of SpectrumContainers, PointSpectra and SpectralSegments is only used by HIFI, where a SpectralSegment physically corresponds to a single HIFI subband. For PACS and SPIRE, a SpectrumContainer usually contains a single PointSpectrum with one SpectralSegment.

The most basic SpectrumContainers for Herschel data are the `Spectrum1d` (see Section 3.2) and `Spectrum2d` (see Section 3.3) datasets and the `SpectralSimpleCube` product (see Section 3.5). They form the basic building blocks and can be wrapped into various kinds of other products (i.e. with added metadata) or extended to form instrument-specific products.

3.2. Spectrum1d

A `Spectrum1d` is a one-dimensional representation of a spectrum. It is a TableDataset (see Section 2.4) that has columns for wave, flux, flag, weights and segment number. In this case, the column called wave can contain any of wavelength, frequency, wavenumber, or velocity information. If the dataset has more than one segment, they are contained within the same columns, and distinguished by the segment number. For PACS and SPIRE, an error column is used instead of weights. A SPIRE example is shown in Figure 3.1. Details of the `Spectrum1d` can also be found in the User Reference Manual in *HCSS User's Reference Manual*.



Meta Data

name	value	unit	description
dec	69.00016316436611	deg	Dec pointing for this channel
ra	265.05276199698494	deg	Ra pointing for this channel
channelName	SLWC3		Channel name
waveunit	cm-1		Units of the WaveColumn

Table Data

Index	wave [cm⁻¹]	flux [W/(m²·Hz·sr)]	error [W/(m²·Hz·sr)]	mask
0	14.6	-7.016801537367034E-20	9.015834526797619E-20	0
1	14.61	-8.08943954345055E-20	8.300907187397333E-20	0
2	14.62	-1.0639787101925635E-19	9.4180548458016E-20	0
3	14.63	-1.58320393373954E-19	1.3381692542005841E-19	0
4	14.64	-2.2848449865094403E-19	2.0613995888743143E-19	0
5	14.65	-1.9549735258440146E-19	2.1799688706425017E-19	0
6	14.66	-9.746143604398556E-20	1.4874872934796137E-19	0
7	14.67	-4.586591876082572E-20	9.816504396993623E-20	0
8	14.68	-2.0949709627759984E-20	7.034193710635918E-20	0
9	14.69	-1.403852972677565E-21	5.57780877336805E-20	0
10	14.7	[illegible]	[illegible]	0

Figure 3.1. An example of a Spectrum1d product from SPIRE (in the HIPE Dataset Viewer). In this case, the wave column contains wavenumbers in cm^{-1}, there is no segment number column (as only one segment is contained) and there are additional columns for error and mask.

In addition to the actual data, each column can also contain the units and a description, which is used to label the axes in plots generated by spectral tools within HIPE. Metadata can also be added to a `Spectrum1d` to describe the target, RA, Dec. etc. (see Section 2.7.1).

Several related spectra, or sub-spectra, can be kept together in a single `Spectrum1d` dataset by storing them as different spectral segments. Different spectral segments can be stored in the same dataset even if they do not have the same number of datapoints or an identical wave array (but they must have the same units). The spectral tools in HIPE operate on all of the segments - in general applying the selected function to all segments separately. However, the spectrumFitterGUI works on one specified segment at a time.

3.2.1. Creating a Spectrum1d

To make a `Spectrum1d` from scratch the data for each column must be created as an array and added to the dataset - the following is a simple example with a single segment:

```
# Define the spectrum
#(in this case in frequency and brightness temperature)
wave = Double1d([1000.0, 1000.2, 1000.4, 1000.6, 1000.8,\
                 1001.0, 1001.2, 1001.4, 1001.6, 1001.8])
flux = Double1d([12.2, 12.5, 13.0, 11.8, 11.9, \
                 12.6, 12.2, 12.8, 12.2, 15.2])
# Flags all set to zero (by default)
flag = Int1d(10)
# Weights all set to one
# (note that weight of zero means the sample is irrelevant)
weight = Double1d(10) + 1
# Put all the data into a single segment (i.e. all segment Ids to zero)
segs = Int1d(10)
# Create the Spectrum1d
mySpectrum1d = Spectrum1d(flux, wave, weight, flag, segs)
```

Example 3.1. Adding data arrays as columns to a spectrum dataset.

This simple example can now be viewed and operated upon by spectral tools in HIPE. More information can be added to define the units (see Section 2.6) and descriptions of the axes:

```
# Import the Herschel unit classes
from herschel.share.unit import *
# Set the wave and flux units and descriptions
mySpectrum1d.waveUnit = Frequency.GIGAHERTZ
mySpectrum1d.waveDescription = "Frequency"
```

```
mySpectrum1d.fluxUnit = Temperature.KELVIN
mySpectrum1d.fluxDescription = "Brightness Temperature"
```

Example 3.2. Setting the units of various spectral metadata.

Multiple spectral segments can be added by changing the segment ID in the segments column (note that if several segments are present, they do not necessarily need to have the same number of data points, or the same wave array):

```
# Define the spectrum
#(in this case in frequency and brightness temperature)
wave1 = Double1d([1000.0, 1000.2, 1000.4, 1000.6, 1000.8,\
                 1001.0, 1001.2, 1001.4, 1001.6, 1001.8])
wave2 = Double1d([453.0, 453.2, 453.4, 453.6, 453.8,\
                 454.0, 454.2, 454.4])
flux1 = Double1d([12.2, 12.5, 13.0, 11.8, 11.9, \
                 12.6, 12.2, 12.8, 12.2, 15.2])
flux2 = Double1d([6.2, 6.5, 5.0, 6.8, 7.9, \
                 6.6, 5.2, 5.8])
# Flags all set to zero (by default)
flag1 = Int1d(10)
flag2 = Int1d(8)
# Weights all set to one
# (note that weight of zero means the sample is irrelevant)
weight1 = Double1d(10) + 1
weight2 = Double1d(8) + 1
# Set the segment IDs
seg1 = Int1d(10)
seg2 = Int1d(8) + 1
# Create the Spectrum1d
flux   = flux1.append(flux2)
wave   = wave1.append(wave2)
weight = weight1.append(weight2)
flag   = flag1.append(flag2)
segs   = seg1.append(seg2)
mySpectrum1d = Spectrum1d(flux, wave, weight, flag, segs)
```

Example 3.3. Adding spectral segments to a one-dimensional spectrum dataset.

3.2.2. Accessing data from a Spectrum1d

The following table defines the standard `Spectrum1d` columns and how these are accessed on the command line in HIPE.

Table 3.1. Spectrum1d columns and access

Column Name	Contents	Direct data access	Java access method	Jython shortcut	Type
wave	Wavelength, frequency, wavenumber, or velocity	`['wave'].data`	`.getWave()`	`.wave`	Double1d
flux	Flux density, brightness temperature, brightness, etc.	`['flux'].data`	`.getFlux()`	`.flux`	Double1d
weight	Weight for each datapoint	`['weight'].data`	`.getWeight()`	`.weight`	Double1d
flag	Flag for each datapoint	`['flag'].data`	`.getFlag()`	`.flag`	Int1d
segments	Spectral segment number	`['segment'].data`	`.getSegment()`	`.segment`	Int1d

The direct data access (using square brackets) is the generic way to access columns inside Herschel datasets (see Section 2.1). The Java access method and related Jython shortcut provide shorter ways to access specific columns of data, and so are preferred by some people in their scripts. These methods are explicitly coded into the class definition of the `Spectrum1d` dataset and so only exist for the columns included in that definition (this is one of the things that can be added in instrument-specific extensions of the `Spectrum1d`).

As an example, the following code plots the spectrum from a `Spectrum1d` in PlotXY (see the chapter on plotting in the *Data Analysis Guide* in *Data Analysis Guide*) using the Jython shortcut access:

```
pl = PlotXY()
pl.addLayer(LayerXY(mySpectrum1d.wave, mySpectrum1d.flux))
```

Example 3.4. Plotting the spectrum with the wave and the flux as axes.

The weights column is used by HIFI, and by default, these are computed based on receiver temperature. A weight of zero means that sample was irrelevant.

For PACS and SPIRE, errors are (or will be, for PACS) used rather than weights. Therefore, PACS and SPIRE products contain an error column instead of the weights column. The errors are converted into weights on-the-fly when either `.getWeight()` or `.weight` are called. The conversion follows this formula:

weight = $1/\text{error}^2$

The HIPE Spectrum Toolbox tasks work solely with weights, and so they apply the above conversion automatically if only errors are present. The task then uses/propagates the weights and automatically converts them back to errors in the output. This automatic conversion in the Spectrum Toolbox tasks can be turned off by setting a Boolean parameter in the `Spectrum1d`:

```
# Turn off automatic conversion of errors to weights
# (default is True)
mySpectrum1d.setAutoConversion(False)
```

Example 3.5. Turning off automatic conversion of errors to weight.

The conversion can be carried out explicitly by calling the following methods:

```
error  = mySpectrum1d.weight2Error(mySpectrum1d.weight)
weight = mySpectrum1d.error2Weight(mySpectrum1d.error)
```

Example 3.6. Converting weights to errors and errors to weights.

The flag column contains an integer flag per sample. These are used by HIFI to show which data samples are affected by particular issues such as saturation (see the HIFI *Data Reduction Guide*: Flags in HIFI data). The flags are used by some of the Spectrum Toolbox tasks to mask out bad samples.

More details of the Spectrum Toolbox tasks can be found in the *Data Analysis Guide* in *Data Analysis Guide* and in the User Reference Manual.

Some useful commands related to manipulating spectral segments in a script are:

```
# The total number of segments in this spectrum1d
segmentCount = mySpectrum1d.segmentCount
# The valid segment indices in the spectrum1d
segmentIndices = mySpectrum1d.segmentIndices
# Extract the segment using the segment ID
# (as given in the segment column)
segment0 = mySpectrum1d.getSegment(0)
# Access the wave and flux data of this segment
wave0 = segment0.wave
flux0 = segment0.flux
# Extract the segment using a counter over segment IDs
# (i.e. the number given refers to the order of
# the IDs in the segment column, starting at zero)
```

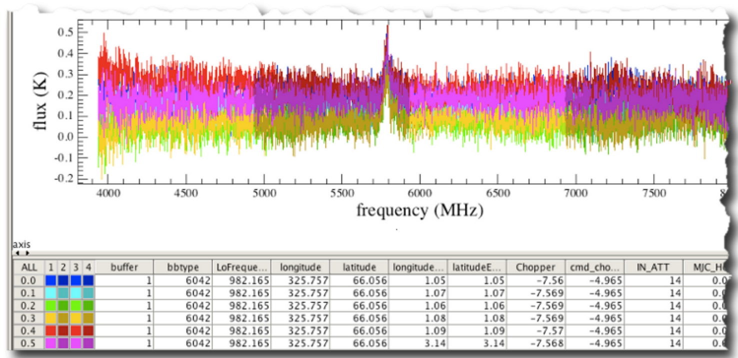
```
segment0 = mySpectrum1d.getSpectralSegment(0)
```

Example 3.7. Manipulating spectral segments.

3.3. Spectrum2d

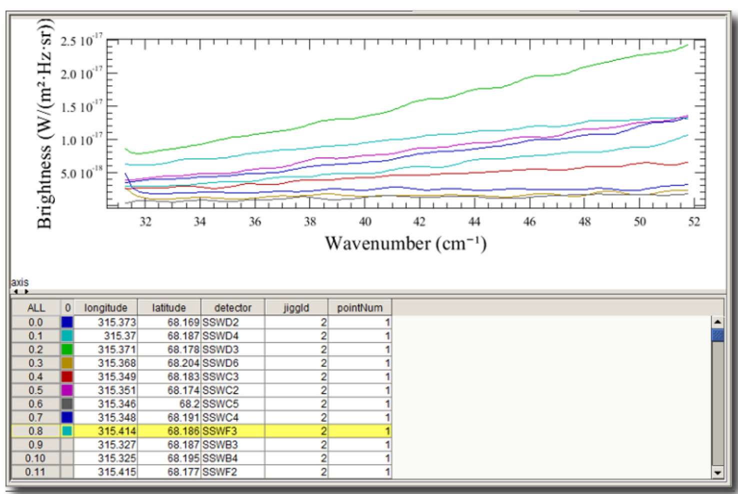
For multiple spectra taken in an observation, a two-dimentional structure is required. The components of a `Spectrum2d` dataset are similar to the `Spectrum1d` dataset (see Section 3.2), except for the provision of a second dimension where each spectrum is stored horizontally in a new row of the table. The same column names exist as in the `Spectrum1d`, but each element in the column is an array containing a spectrum. Details of the `Spectrum2d` can also be found in the User Reference Manual in *HCSS User's Reference Manual*.

The `Spectrum2d` dataset is used by HIFI to store all of the data from different data frames from a single observation (see Figure 3.2). For SPIRE, it is used to collect together the spectra from different sky positions before gridding them into a spectral cube (see Figure 3.3). The advantage of this structure over many separate `Spectrum1d` datasets is that a group of related spectra can be contained in a single dataset that is easier to display and manipulate than a bunch of individual datasets.



ALL	1 2 3 4	buffer	bbtype	LoFreque...	longitude	latitude	longitude...	latitudeE...	Chopper	cmd_cho...	IN_ATT
0.0		1	6042	982.165	325.757	66.056	1.05	1.05	-7.56	-4.965	14
0.1		1	6042	982.165	325.757	66.056	1.07	1.07	-7.569	-4.965	14
0.2		1	6042	982.165	325.757	66.056	1.06	1.06	-7.569	-4.965	14
0.3		1	6042	982.165	325.757	66.056	1.08	1.08	-7.569	-4.965	14
0.4		1	6042	982.165	325.757	66.056	1.09	1.09	-7.57	-4.965	14
0.5		1	6042	982.165	325.757	66.056	3.14	3.14	-7.568	-4.965	14

Figure 3.2. Example of a HIFI Spectrum2d viewed in the SpectrumExplorer in HIPE. Different spectra appear as different rows, and in this case each spectrum has 4 subbands. Each subband is plotted in the colour shown in the boxes on the lower left. The other columns give further information about each spectrum.



ALL	0	longitude	latitude	detector	jiggId	pointNum
0.0		315.373	68.169	SSWD2	2	1
0.1		315.37	68.187	SSWD4	2	1
0.2		315.371	68.178	SSWD3	2	1
0.3		315.368	68.204	SSWD6	2	1
0.4		315.349	68.183	SSWC3	2	1
0.5		315.351	68.174	SSWC2	2	1
0.6		315.346	68.2	SSWC5	2	1
0.7		315.348	68.191	SSWC4	2	1
0.8		315.414	68.186	SSWF3	2	1
0.9		315.327	68.187	SSWB3	2	1
0.10		315.325	68.195	SSWB4	2	1
0.11		315.415	68.177	SSWF2	2	1

Figure 3.3. Example of a SPIRE Spectrum2d viewed in the SpectrumExplorer in HIPE. Different spectra appear as different rows, but in this case each spectrum only has one subband. There are fewer columns for additional information than in the HIFI example in Figure 3.2.

An additional component of the `Spectrum2d`, which is important for HIFI data, is the ability to split each spectral row into sub-spectra (analogous to the segments in the `Spectrum1d` - see Section 3.2).

The output from the HIFI spectrometers contains subbands, where several CCD or autocorrelator readouts lead to several "chunks" (subbands) of spectra in one data frame. This functionality is only used by HIFI, and so these chunks, or segments, in the `Spectrum2d` are (officially) referred to as "subbands".

Inside the `Spectrum2d`, subbands are vertical splits in the flux (and weight, flag etc.) columns, equivalent to the functionality of the segment column in the `Spectrum1d` . The flux etc. columns are replaced with flux_1, flux_2, ... columns, depending on how many subbands were defined. The definition of the subbands in terms of the starting index (e.g. the channel number in the HIFI CCD) and the length (the number of datapoints in the subband) are stored in metadata items called `subbandstart` and `subbandlength` (see Figure 3.4). The `subbandstart` refers to the index in the original spectrum before it was split into subbands, and for HIFI this refers to the physical channel inside the instrument. It is an important parameter because some data samples might have been discarded from the beginning/end of the spectrum when it was split into subbands (i.e. the first `subbandstart` is not necessarily equal to zero).

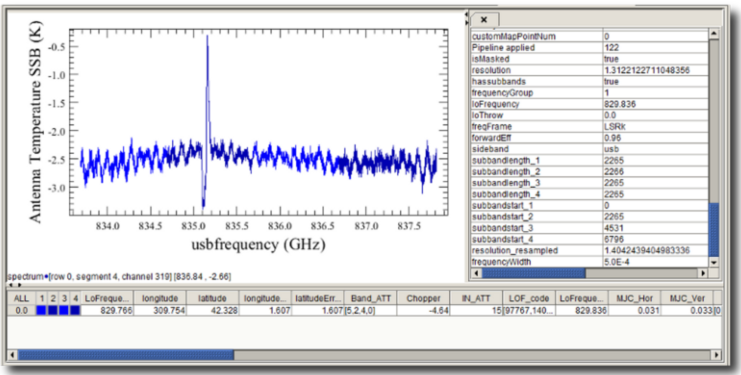


Figure 3.4. Example of a HIFI Spectrum2d viewed in the SpectrumExplorer in HIPE, showing the metadata describing the different subbands (to display the metadata, right click in the plot and select Dialogs - Metadata). In this dataset, there is one spectrum with 4 subbands.

3.3.1. Creating a Spectrum2d

An example of creating a simple `Spectrum2d` containing 4 spectra, from scratch, is given below. The inputs to create the `Spectrum2d` are 2D arrays of wave, flux etc., and so every spectrum must have the same number of datapoints.

```
wave2d    = Double2d([[1000.0,1000.2,1000.4,1000.6],\
                      [1000.0,1000.2,1000.4,1000.6],\
                      [1000.0,1000.2,1000.4,1000.6],\
                      [1000.0,1000.2,1000.4,1000.6]])
flux2d    = Double2d([[12.2,12.5,13.6,12.8],\
                      [12.8,12.2,13.3,12.9],\
                      [10.2,14.5,12.5,11.4],\
                      [12.2,12.5,13.6,12.8]])
flag2d    = Int2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
weight2d = Double2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
mySpectrum2d = Spectrum2d(flux2d, weight2d, flag2d)
mySpectrum2d.setWave(wave2d)
```

Example 3.8. Creating a two-dimensional dataset containing four spectra.

A further example to set up a basic `Spectrum2d` with associated subbands is given below. This example sets up a dataset which holds 2 spectra, each with 2 subbands. Each subband can cover a different wavelength range if necessary (e.g., as for the individual subbands of the HRS spectrometer of HIFI). This forms the basis of how HIFI observations, which are typically made up of many frames, are stored in HIPE.

```
# This example creates a Spectrum2d that contains
# two rows of spectra, each containing two subbands
```

```
#
# Initialise the Spectrum2d
mySpectrum2d = Spectrum2d()
#
# Define the two spectra - each will be split into two subbands
wave2d = Double2d([[1000.0,1000.2,1000.4,1000.6,1000.8,1001.0,1001.2,1001.4],\
                   [1000.0,1000.2,1000.4,1000.6,1000.8,1001.0,1001.2,1001.4]])
flux2d = Double2d([[12.2,12.5,13.6,12.8,10.2,14.5,12.5,11.4],\
                   [12.8,12.2,13.3,12.9,12.2,12.5,13.6,12.8]])
#
# Indicate the number of subbands that each spectrum will have
mySpectrum2d.setSubbands(2)
#
# Set the index at which each subband starts and its length.
# Here, the intial spectrum is split into two equal length
# subbands without discarding any data.
mySpectrum2d.setSubbandStart(Int1d([0, 4]))
mySpectrum2d.setSubbandLength(Int1d([4, 4]))
#
# Now set the (2D) data for each subband according to the
# start and length above.
# Subband 1
mySpectrum2d.set("wave_1", wave2d[:, 0:4])
mySpectrum2d.set("flux_1", flux2d[:, 0:4])
#
# Subband 2
mySpectrum2d.set("wave_2", wave2d[:, 4:8])
mySpectrum2d.set("flux_2", flux2d[:, 4:8])
```

Example 3.9. Creating a multiband spectrum dataset.

3.3.2. Accessing data from a Spectrum2d

The following table defines the standard `Spectrum2d` columns and how these are accessed on the command line in HIPE.

Table 3.2. Spectrum2d columns and access

Column Name	Contents	Direct data access	Java access method	Jython shortcut	Type
wave	Wavelength, frequency, wavenumber, or velocity	`['wave'].data`	`.getWave()`	`.wave`	Double2d
flux	Flux density, brightness temperature, brightness, etc.	`['flux'].data`	`.getFlux()`	`.flux`	Double2d
weight	Weight for each datapoint	`['weight'].data`	`.getWeight()`	`.weight`	Double2d
flag	Flag for each datapoint	`['flag'].data`	`.getFlag()`	`.flag`	Int2d

The direct data access (using square brackets) is the generic way to access columns inside Herschel datasets (see Section 2.1). The Java access method and related Jython shortcut provide shorter ways to access specific columns of data, and so are preferred by some people in their scripts. These methods are explicitly coded into the class definition of the `Spectrum2d` dataset and so only exist for the columns included in that definition (this is one of the things that can be added in instrument-specific extensions of the `Spectrum2d`).

If subbands are not used (e.g. for SPIRE data), all the access methods above work to return 2D arrays. For example, to plot (see the chapter on plotting in the *Data Analysis Guide* in *Data Analysis Guide*) the first spectrum using the Jython syntax:

```
pl = PlotXY()
pl.addLayer(LayerXY(mySpectrum1d.wave[0,:], mySpectrum1d.flux[0,:]))
```

Example 3.10. Plotting some columns selected using slice notation.

For HIFI, where different subbands can be specified, each column is split into sub-columns (appended with the subband number - e.g. flux_1, flux_2..), and only the Java-like access methods (`getFlux()` etc.) work, with the subband number specified inside the brackets. For example:

```
# To access (e.g.) subband number 3
subbandNum = 3
subbandWave = mySpectrum2d.getWave(subbandNum)
subbandFlux = mySpectrum2d.getFlux(subbandNum)
```

Example 3.11. Accessing subbands using specific methods

In the above example, `subbandWave` and `subbandFlux` are 2D arrays. There are two (equivalent) ways to access the individual spectra - either specifying the indices (the first index is for the subband number), as in the previous plot example above:

```
# Plot the first spectrum for the subband
pl.addLayer(LayerXY(subbandWave[0,:], subbandFlux[0,:]))
```

Example 3.12. Plotting the first spectrum of a subband using indices.

Or by using a "get()" method, specifying the subband index number:

```
# Plot the first spectrum for the subband
pl.addLayer(LayerXY(subbandWave.get(0), subbandFlux.get(0)))
```

Example 3.13. Plotting the first spectrum of a subband using the get method.

Some other useful commands to investigate subbands are:

```
# To determine how many subbands there are
print mySpectrum2d.subbandCount
print mySpectrum2d.numberOfSubbands
# To determine the start element and length of each subband
# (returns an array of indices/lengths taken from the metadata)
print mySpectrum2d.subbandStart
print mySpectrum2d.subbandLength
```

Example 3.14. Inspect subbands.

3.4. SimpleSpectrum

The `SimpleSpectrum` is a product containing a single `Spectrum1d` (see Section 3.2) dataset and metadata describing the associated observation. It is designed to allow a `Spectrum1d` to be written out to a FITS file and easily shared between collaborators, and also to allow simple spectra from other instruments to be read into HIPE with meaningful metadata preserved. Details of the `SimpleSpectrum` can also be found in the User Reference Manual in *HCSS User's Reference Manual*. Note that also, a `Spectrum1d` object can be written directly to a FITS file without the `SimpleSpectrum` product wrapper.

There are tasks for each instrument to extract a `SimpleSpectrum` product from any of the more complicated instrument products:

```
simpleSpectrum = convertSingleHifiSpectrum()
simpleSpectrum = spireProduct2SimpleSpectrum()
simpleSpectrum = convertPacsProduct2SimpleSpectrum()
```

Example 3.15. Converting instrument specific spectral datasets to SimpleSpectrum.

Details of how to run these tasks can be found in the User Reference Manual for HIFI, SPIRE and PACS.

3.5. SpectralSimpleCube

The `SpectralSimpleCube` is a product containing three-dimensional datasets with axes denoting the longitude, latitude and spectral dimension of the data. The three dimensions are interpreted as spectral stacks of images, each with the same spatial grid - i.e. the `SpectralSimpleCube` contains data that has been projected (gridded) onto a regular square grid. It is the format used by all three instruments as the output of their gridding/projection tasks for spectral mapping observations.

The spatial grid of the cube is specified in the **World Coordinate System** with a WCS object defining *cdelt*, *crpix*, *crval* etc. in the same way as Herschel images (see Chapter 4). The individual spatial grid squares are referred to either as "pixels" (e.g. by HIFI) or as "spaxels" (e.g. by PACS).

The spectral axis values are provided as a `Double1d` array. It is assumed that the spectral axis applies to each spatial position.

The product can contain cubes for the flux, weight, error, flag, coverage, depending on the gridding algorithm used (`SpectralSimpleCubes` need not contain all of these datasets).

3.5.1. Creating a SpectralSimpleCube

You can create a `SpectralSimpleCube` out of a three-dimensional array, like in the following example:

```
# Create a 100x100x100 cube with all values equal to 1.0
data = Double3d(100, 100, 100, 1.0)
# Create a SpectralSimpleCube
myCube = SpectralSimpleCube(image=data)
```

Example 3.16. Creating cubes from Numeric arrays.

You can then set other cube components among the ones shown in Table 3.3. For instance, you can set the weight for each data point as in the following example:

```
# Create a dummy weights cube with all weights equal to 1.0
weight = Double3d(100, 100, 100, 1.0)
# Assign weights to cube
myCube.weight = weight
```

Example 3.17. Creating a weight cube.

You can set components in any order, with one exception: before assigning a wavelength array to the cube (`wave` component) you must have assigned a valid WCS (`wcs` component). For information on how to assign a WCS to a cube, see Section 4.1.

3.5.2. Accessing data from a SpectralSimpleCube

The standard datasets that can be contained in a `SpectralSimpleCube` are described in the following table (not all of them must be present) with how they are accessed on the command line in HIPE.

Table 3.3. SpectralSimpleCube content and access

Column Name	Contents	Direct data access	Java access method	Jython shortcut	Type
wave	Wavelength, frequency,	`['ImageIndex']['DepthIndex'].data`	`.getWave()`	`.wave`	Double1d

Column Name	Contents	Direct data access	Java access method	Jython shortcut	Type
	wavenumber, or velocity	*(Remove space between] and [ )*			
image	Flux density, brightness temperature, brightness, etc.	`['image'].data`	`.getImage()`	`.image`	Double3d
wcs	World Coordinate System parameters	*	`.getWcs()`	`.wcs`	Wcs
weight	Weight for each datapoint	`['weight'].data`	`.getWeight()`	`.weight`	Double3d
error	Error for each datapoint	`['error'].data`	`.getError()`	`.error`	Double3d
flag	Flag for each datapoint	`['flag'].data`	`.getFlag()`	`.flag`	Short3d
coverage	Coverage for each datapoint from gridding task	`['coverage'].data`	`.getCoverage()`	`.coverage`	Double3d

* the WCS information is stored in the "image" metadata. The values can be accessed in the form of metadata items with the square bracket notation as follows:

```
# Extract the "image" metadata
wcsMeta = cube['image'].meta
# Wrap this into a Wcs object
wcs = Wcs(wcsMeta)
```

Example 3.18. Extracting the image metadata (including Wcs information).

The direct data access (using square brackets) is the generic way to access columns inside Herschel datasets (see Section 2.1). The Java access method and related Jython shortcut provide shorter ways to access specific columns of data, and so are preferred by some people in their scripts. These methods are explicitly coded into the class definition of the `SpectralSimpleCube` and so only exist for the data included in that definition (this is one of the things that can be added in instrument specific extensions of the `SpectralSimpleCube`).

An example of extracting the 3D array of fluxes from a cube is:

```
flux3d = cube.image
```

Example 3.19. Accessing the image (flux) data from a cube.

The most basic operations that might be useful for scripting with a cube are to extract a spectrum from a particular location (one spaxel/pixel), or to extract a single image plane as a `SimpleImage` (see the chapter on image analysis in the *Data Analysis Guide* in *Data Analysis Guide*). More complicated operations can be achieved using the Cube Spectral Analysis Toolbox (see the *Data Analysis Guide* in *Data Analysis Guide*) or with the basic Spectrum Toolbox (see the *Data Analysis Guide* in *Data Analysis Guide*).

```
# To extract a single spectrum
# Define the RA and Dec of the position to extract
# e.g. taking the nominal (commanded) source position from the metadata
ra  = cube.meta['raNominal'].value
dec = cube.meta['decNominal'].value
# First find the pixel/spaxel closest to the desired RA/Dec
```

```
pixCoord = cube.nearestPixels(ra, dec)
#
# Extract the flux as a Double1d specifying RA, Dec
# in decimal degrees
extractedFlux = cube.getFlux(ra, dec)
# or, specifying pixel/spaxel position (row, column)
extractedFlux = cube.getFlux(pixCoord[0], pixCoord[1])
#
# Extract the flux as a Spectrum1d at pixel/spaxel position
extractedSpectrum = cube.getSpectrum1d(pixCoord[0], pixCoord[1])
```

Example 3.20. Extracting a single spectrum from a cube.

Note: the `nearestPixels` and `getFlux(ra, dec)` methods work by calculating the exact position in fractions of a pixel from the WCS (using `wcs.getPixelCoordinates(ra, dec)` and then rounding this to the nearest integer with `Math.round()`).

```
# To extract an image plane
# Find the nearest plane to a particular
# wave-scale value (e.g. 1000.0 GHz)
freq = 1000.0
planeIndex = cube.nearestPlane(freq)
#
# Extract a single plane as a Double2d specifying the frequency
extractedFlux = cube.getFlux(freq)
# or, specifying the plane index
extractedFlux = cube.getFlux(planeIndex)
#
# Wrap the extracted Double2d dataset into a
# SimpleImage so that it can be displayed in HIPE
extractedImage       = SimpleImage()
extractedImage.image = extractedFlux
extractedImage.wcs   = cube.wcs
```

Example 3.21. Extracting a single image plane for a specific frequency.

Note: the `nearestPlane` and `getFlux(freq)` work by starting at the zero element in the wave array and looping through all planes in the cube to find the smallest separation with the entered value. This applies even if the entered value is outside of the cube range.

Displaying cube dimensions. You can display the dimensions of the cube with the following command:

```
print myCube.dimensions
#array('i', [100, 70, 30])
```

Example 3.22. Printing the cube dimensions.

The first number (100 in the example above) is the size in the wavelength dimension, while the second and third number are the y and x dimensions, respectively, or in other words the number of rows and columns. For more information about cube dimensions see the *Data Analysis Guide*: Section 6.3 in *Data Analysis Guide*.

3.6. Instrument-specific spectral products

The following table summarises the spectral products used by each instrument and how these relate to the basic building blocks defined in the previous sections.

Table 3.4. Instrument-specific spectral products

Instrument	Product/Dataset	Basic building block	Description
SPIRE (Level-1)	SpectrometerDetectorSpectrum	Spectrum1d	Composite dataset containing spectra from all detectors and scans from one

Instrument	Product/Dataset	Basic building block	Description
			observation. Uses **SpireSpectrum1d** datasets which are extensions of **Spectrum1d** for SPIRE containing an error column and ra, dec and channelName metadata.
SPIRE (Level-2)	Spectrometer-PointSourceSpectrum	Spectrum1d	Composite dataset containing point source calibrated spectra from the central detectors. Uses **SpireSpectrum1d** datasets.
SPIRE	SpirePreprocessed-Cube	Spectrum2d	Data from all sky positions prepared for gridding into a cube.
SPIRE (Level-2 maps)	SpectralSimple-Cube	SpectralSimple-Cube	**SpectralSimpleCube** used directly without SPIRE specific extension.
HIFI	HifiSpectrum-Dataset	Spectrum2d	Extension of **Spectrum2d** for HIFI - contains additional data specific to HIFI (e.g. housekeeping , flags).
HIFI	HrsSpectrum-Dataset	Spectrum2d	Extension of a **HifiSpectrumDataset** - contains additional information for the HRS (e.g. correlation functions).
HIFI	WbsSpectrum-Dataset	Spectrum2d	Extension of a **HifiSpectrumDataset** - contains additional information for the WBS.
PACS (Level 1)	PacsCube	-	PACS specific format - it is a Spectrum-Container, but does not extend any of the basic building blocks described in this chapter. Does not contain flag or weight information, rather it has PACS-specific Masks.
PACS (Level 2)	SpectralSimple-Cube	SpectralSimple-Cube	The so-called PACS "projectedCube". Currently does not contain flag or weight information. This uses the **SpectralSimpleCube** without PACS specific extension.
PACS (Level 2)	PacsRebinnedCube	SpectralSimple-Cube	PACS extension of the **SpectralSimpleCube**. Currently does not contain flag or weight information.

Chapter 4. The World Coordinate System

4.1. Assigning a World Coordinate System to images and cubes

You can assign WCS information to images and cubes. The World Coordinates System (WCS) describes the coordinates of a `SimpleImage` or `SimpleCube`. It makes it possible to convert image coordinates to world coordinates and the other way around. The WCS can have a lot of parameters, as defined in the FITS standard:

- naxis: the number of axes
- crval1: First coordinate of the centre
- crval2: Second coordinate of the centre
- crpix1: Reference pixel X coordinate
- crpix2: Reference pixel Y coordinate
- cdelt1: Pixel scale of axis 1. Step per pixel or number of degrees per pixel along x-axis when converting to Sky Coordinates. These parameters are no longer used in modern Wcs definition, but are included in the CDi_j matrix.
- cdelt2: Pixel scale axis 2. Step per pixel or number of degrees per pixel along y-axis when converting to Sky Coordinates. These parameters are no longer used in modern Wcs definition, but are included in the CDi_j matrix.
- ctype1, ctype2: Projection type name. This can be "LINEAR", "PIXEL" or the FITS convention. The default value for ctype1 and ctype2 is "LINEAR". When using the FITS convention, the first four characters are:

  - RA-- and DEC- for equatorial coordinates
  - GLON and GLAT for galactic coordinates
  - ELON and ELAT for ecliptic coordinates

  The next four characters describe the projection. Possibilities are:

  - -AZP: Zenithal (Azimuthal) Perspective
  - -SZP: Slant Zenithal Perspective
  - -TAN: Gnomonic = Tangent Plane
  - -SIN: Orthographic/synthesis
  - -STG: Stereographic
  - -ARC: Zenithal/azimuthal equidistant
  - -ZPN: Zenithal/azimuthal PolyNomial
  - -ZEA: Zenithal/azimuthal Equal Area

  - -AIR: Airy
  - -CYP: CYlindrical Perspective
  - -CAR: Cartesian
  - -MER: Mercator
  - -CEA: Cylindrical Equal Area
  - -COP: COnic Perspective
  - -COD: COnic equiDistant
  - -COE: COnic Equal area
  - -COO: COnic Orthomorphic
  - -BON: Bonne
  - -PCO: Polyconic
  - -SFL: Sanson-Flamsteed
  - -PAR: Parabolic
  - -AIT: Hammer-Aitoff equal area all-sky
  - -MOL: Mollweide
  - -CSC: COBE quadrilateralized Spherical Cube
  - -QSC: Quadrilateralized Spherical Cube
  - -TSC: Tangential Spherical Cube
  - -NCP: North celestial pole (special case of SIN)
  - -GLS: GLobal Sinusoidal (Similar to SFL)
- Other types are also possible (for example TEMP for temperature.)
  - cunit1: The Unit of Axis 1.
  - cunit2: The Unit of Axis 2.
  - epoch: Epoch of coordinates.
  - Radesys: The reference frame, default value is "ICRS".
  - pc1_1: Element (1,1) of the linear transformation matrix. The pc1 and pc2 parameters are no longer used in modern Wcs definition, but are together with CDELT1 and CDELT2 included in the CDi_j matrix.
  - pc1_2: Element (1,2) of the linear transformation matrix.
  - pc2_1: Element (2,1) of the linear transformation matrix.
  - pc2_2: Element (2,2) of the linear transformation matrix.
  - cd1_1: Element (1,1) of the corrected linear transformation matrix.

  - cd1_2: Element (1,2) of the corrected linear transformation matrix.
  - cd2_1: Element (2,1) of the corrected linear transformation matrix.
  - cd2_2: Element (2,2) of the corrected linear transformation matrix.

With a third dimension the following also applies:

- ctype3: Description of what the 3rd axis represents, for instance Wavelength, Time, M1 Temperature, and so on.
- cunit3: The Unit of Axis 3.
- crval3: [Optional - in case of equidistant 3rd dimension]. Wavelength, time, ... of reference layer; unit : length, time, ...
- crpix3: [Optional - in case of equidistant 3rd dimension] Reference layer index
- cdelt3: [Optional - in case of equidistant 3rd dimension] Scale in 3rd dimension - unit: length, time, ...
- PC elements:
  - pc1_3: Element (1,3) of the linear transformation matrix.
  - pc2_3: Element (2,3) of the linear transformation matrix.
  - pc3_1: Element (3,1) of the linear transformation matrix.
  - pc3_2: Element (3,2) of the linear transformation matrix.
  - pc3_3: Element (3,3) of the linear transformation matrix.

To create a WCS object that can be assigned to an image you can use something like the following.

```
# Create the WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5, \
   cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = "DEGREES", \
   cunit2 = "DEGREES", ctype1 = "RA---TAN", ctype2 = "DEC--TAN")
# Check whether the WCS is valid
print myWcs.valid
# Assign the world coordinates to our image
myImage = SimpleImage(description = "Mock image", wcs = myWcs)
# You can then obtain the world coordinates at any pixel
print myImage.getWcs().getWorldCoordinates(31,31)
# This provides an array of sky coordinates in degrees.
# You can get the intensity at a given WCS position
# Assign a mock image with all intensity values set to 1.0
myImage.image = Double2d(100, 100, 1.0)
# Get the intensity at a given WCS position.
print myImage.getIntensityWorldCoordinates(30.0012, -22.498)
```

Example 4.1. Creating a WCS object from scratch.

For the `SimpleCube` and `SpectralSimpleCube` objects you can do this almost identically. Using the `d3` cube defined in a previous example:

```
# Create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5, \
   cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = "DEGREES", \
   cunit2 = "DEGREES", ctype1 = "RA---TAN", ctype2 = "DEC--TAN")
# Create a mock cube with all intensity values set to 1.0
myCube = SimpleCube(description="Mock cube", image=Double3d(100, 100, 100, 1.0),
 wcs=myWcs)
# Add third axis (WCS is created with two axes by default)
```

```
myWcs.NAxis = 3
# Add quantities related to the third axis
myWcs.crval3 = 300.0
myWcs.crpix3 = 0
myWcs.cdelt3 = 0.001
myWcs.ctype3 = "Wavelength"
myWcs.cunit3 = "MICROMETERS"
# You can obtain the world coordinates at any pixel on the image.
print myCube.getWcs().getWorldCoordinates(31,31)
# Get the intensity at a given WCS position. We need three
# arguments now, with the first argument being the layer number (depth)
# from which we want the intensity measure. Count starts from 0.
print myCube.getIntensityWorldCoordinates(0,30.0012, -22.498)
```

Example 4.2. Getting the world coordinates from a screen pixel position.

If the third axis of the cube is irregularly sampled, you can define an `imageIndex` array with the sampling values of each layer along the axis. Such array would replace the values of the `crval3`, `crpix3` and `cdelt3` parameters:

```
# Create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5, \
   cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = "DEGREES", \
   cunit2 = "DEGREES", ctype1 = "RA---TAN", ctype2 = "DEC--TAN")
# Create a mock cube
myCube = SimpleCube(description="Mock cube", image=Double3d(100, 100, 100, 1.0),
 wcs=myWcs)
# Add third axis (WCS is created with two axes by default)
myWcs.NAxis = 3
# Add quantities related to the third axis
myWcs.ctype3 = "Wavelength"
myWcs.cunit3 = "MICROMETERS"
# Add the imageIndex array
from herschel.share.unit.Length import MICROMETERS
wavelengths = Double1d([20.0, 45.0, 100.0])
myWcs.setImageIndex(wavelengths, MICROMETERS)
```

Example 4.3. Adding a third axis to a WCS structure to define an image index.

To check whether the third axis is regularly sampled, the following will return 1 if true, 0 if false:

```
print myWcs.equidistantInZ
```

Example 4.4. Printing if the third axis is regularly sampled.

A summary of queries on the wcs of the cube (replace the specific numbers here with those appropriate for your cube):

```
# You can obtain the world coordinates at any spaxes/pixel
print myCube.getWcs().getWorldCoordinates(31,31)

# You can obtain the spectral coordinate at any index position
print myCube.getWcs().getWorldCoordinateZ(0)

# Or the spaxel/pixel and wavelenth/frequency coordinate
# for a world coordinate position
print myCube.wcs.getPixelCoordinates(30.0012, -22.498)
print myCube.wcs.getPixelCoordinateZ(300.023)

# Get the intensity at a given WCS position. We need three
# arguments now, with the first argument being the layer number (depth)
# from which we want the intensity measure. Count starts from 0.
print myCube.getIntensityWorldCoordinates(0,30.0012, -22.498)
```

Example 4.5. Transforming between world coordinates and

4.2. Correcting the astrometry of your data

With the `astrometryFix` task you can correct the astrometry of your data to make it consistent with the astrometry of some other data. To learn more about the `astrometryFix` task, see the corresponding entry in the *User's Reference Manual*: Section 1.29 in *HCSS User's Reference Manual*.

If you are a SPIRE user, there is a helper script to correct the astrometry of your observations. In HIPE, choose *Scripts* → *SPIRE Useful scripts* → *Photometer Astrometry Correction*.

Chapter 5. The Numeric library

This chapter describes the numeric functions available within HIPE. For information on numeric arrays and other datasets, see Chapter 2. For reference information about all the numeric functions in HIPE, see the *User's Reference manual* in *HCSS User's Reference Manual*.

5.1. Numeric functions and lambda expressions

You can apply functions as follows:

```
print SQRT(16) # 4.0 (applied to a scalar)
y = Double1d([1,4,9,16])
print SQRT(y) # [1.0,2.0,3.0,4.0] (applied to a DP numeric array)
```

Example 5.1. Taking the square root of a numeric array of doubles.

As shown by this example, functions on scalars (such as `SQRT`) are implicitly mapped over each element of an array. You can combine functions with arithmetic operators to perform complex operations on each element of an array:

```
t = Double1d([1,2,3,4])
print SIN(1000 * t * (1 + .0003 * COS(3 * t)))
# [0.6260976237441638,0.5797470124743422,0.8629107307631398,
#-0.9811675382238753]
```

Example 5.2. Numeric functions are applied to each element of an array.

The **names of functions in the numeric library have ALL LETTERS** capitalised. This is to avoid ambiguity, as Jython already defines certain functions, such as '`abs`', which are not applicable to our DP numeric arrays.

There are various types of functions in the numeric library:

```
y = Double1d([1,2,3,4])

print SQRT(4) # double->double
print SQRT(y) # double->double (mapped)
print REVERSE(y) # Double1d->Double1d
print MEAN(y) # Double1d->double
```

Example 5.3. Converting values to double as it is the type of the numeric arrays.

You can define **new functions** as *lambda expressions* in Jython and apply them to numeric arrays. For example:

```
y = Double1d([1,2,3,4])

f = lambda x: x*x + 1 #take the given array, call it 'x' and
#return the value x^2 +1 to an array called f.

print f(y) #[2.0,5.0,10.0,17.0]. Each element of y was
#taken -> x then each element was squared
#plus 1 added.
```

Example 5.4. Using lambda expression to apply new functions to arrays in the same way as Numeric functions.

However, in this case, it is much easier and faster to do this with array operations.

```
print y * y + 1
```

Example 5.5. For simple functions it is much more readable to use the built-in operators.

Lambda expressions are not as fast as the standard Java functions provided by the numeric library, but this is often not a problem.

More complex functions (equivalent to subroutines) can be created using the **def** command, which is discussed in Section 1.27.

5.2. Basic functions

Basic functions applicable to scalars or arrays, and returning scalars or arrays of the same size:

ABS in *HCSS User's Reference Manual*	ARCCOS in *HCSS User's Reference Manual*	ARCSIN in *HCSS User's Reference Manual*	ARCTAN in *HCSS User's Reference Manual*
CEIL in *HCSS User's Reference Manual*	COS in *HCSS User's Reference Manual*	EXP in *HCSS User's Reference Manual*	FIX in *HCSS User's Reference Manual* (not applicable to scalars)
FLOOR in *HCSS User's Reference Manual*	LOG in *HCSS User's Reference Manual*	LOG10 in *HCSS User's Reference Manual*	ROUND in *HCSS User's Reference Manual*
SIGNUM in *HCSS User's Reference Manual*	SIN in *HCSS User's Reference Manual*	SQRT in *HCSS User's Reference Manual*	SQUARE in *HCSS User's Reference Manual*
TAN in *HCSS User's Reference Manual*			

These are applied in the form

```
b = SIN(a)
```

Example 5.6. The SIN function works for arrays and scalars.

`b` will be an array of the same dimension as `a`, or a scalar if `a` is a scalar.

Array functions on `Double<n>d` returning a `double`:

MIN in *HCSS User's Reference Manual*	MAX in *HCSS User's Reference Manual*	SUM in *HCSS User's Reference Manual*	PRODUCT in *HCSS User's Reference Manual*
MEAN in *HCSS User's Reference Manual*	MEDIAN in *HCSS User's Reference Manual*	STDDEV in *HCSS User's Reference Manual*	

Warning

Remember to strip your arrays of any `NaN` (Not a Number) values before using these functions, or the result will always be a `NaN`. See Section 2.2.9 for more information.

```
b = MIN(a) # 'b' is the minimum value of the array 'a'.
```

Example 5.7. Finding the minimum value of an array.

These functions can also be used to reduce the dimensionality of an array (for instance, three- to two-dimensional).

Functions applicable to one-dimensional arrays and returning an array of the same size:

- REVERSE in *HCSS User's Reference Manual*

Functions applicable to arrays and returning an array of increased rank (number of dimensions):

- REPEAT in *HCSS User's Reference Manual*

Warning

Many of these functions have lower case equivalents built-in in Jython. Be aware of which one you are using, because their behaviour could differ in some cases, as shown by the example below which creates a table with NaN values in it.

```
tt=Double1d.range(10)
tt[0]=Double.NaN
print max(tt)
# NaN
print min(tt)
# NaN
tt[1]=Double.NaN
tt[0]=1.0
print max(tt) # By using the built-in Jython functions
# 9.0
print min(tt)
# 1.0
print MAX(tt) # By using the DP Numeric functions
# NaN
print MIN(tt)
# NaN
```

Example 5.8. Differences between the lower-case Jython functions and the upper-case Numeric functions.

5.3. Integral transforms

HIPE has two options for performing fast Fourier transforms: FFT and FFT_PACK. These two classes are comparable in terms of their accuracy. Both FFT and FFT_PACK transform data from a complex array to a complex array. They differ in execution time. Additional classes, which are related to FFT_PACK, add the options of transforming real data and of taking advantage of symmetry for shorter execution run-times. Table 5.1 gives an overview of the available transformation options:

Table 5.1. Forward Fourier transforms for input of length N.

Name	Input type	Output type	Output length	Notes
FFT	Complex1d	Complex1d	N	
	$X_k=\sum_{n=0}^{N-1} x_n e^{-ikn\frac{2\pi}{N}} \qquad k=0,\ldots,N-1.$			
FFT_PACK	Complex1d	Complex1d	N	
	$X_k=\sum_{n=0}^{N-1} x_n e^{-ikn\frac{2\pi}{N}} \qquad k=0,\ldots,N-1.$			
RealDoubleFFT#ft	double[]	Complex1D	floor(N/2) +1	Calculates FFT of real input.
	$a_k=\sum_{n=0}^{N-1} x_n \cos kn\frac{2\pi}{N}, \qquad k=0,\ldots,\lfloor N/2\rfloor.$ $b_k=-\sum_{n=1}^{N-1} x_n \sin kn\frac{2\pi}{N}, \qquad k=1,\ldots,\lfloor N/2\rfloor.$			

<table>
<tr><th>Name</th><th>Input type</th><th>Output type</th><th>Output length</th><th>Notes</th></tr>
<tr><td rowspan="2">FFT_PACK_EVEN</td><td>Double1d</td><td>Double1d</td><td>N</td><td>Discrete cosine transform.</td></tr>
<tr><td colspan="4">$c_k = x_0 + 2\sum_{n=1}^{N-2} x_n \cos kn\frac{\pi}{N-1} + (-1)^k x_{N-1}, \qquad k=0,\ldots,N-1.$</td></tr>
<tr><td rowspan="2">FFT_PACK_ODD</td><td>Double1d</td><td>Double1d</td><td>N</td><td>Discrete sine transform.</td></tr>
<tr><td colspan="4">$c_k = 2\sum_{n=1}^{N-1} x_n \sin(k+1)(n+1)\frac{\pi}{N}, \qquad k=0,\ldots,N-1.$</td></tr>
</table>

5.3.1. FFT

The `FFT` class offers a Discrete Fourier Transform for `Complex1d` arrays. It uses a radix-2 FFT algorithm for array lengths that are powers of 2 and a Chirp-Z transform for other lengths.

Example 5.9 shows the generation of a frequency modulated signal, followed by a FFT both with and without windowing:

```
ts = 1E-6           # Sampling period (sec)
fc = 200000         # Carrier frequency (Hz)
fm = 2000           # Modulation frequency (Hz)
beta = 0.0003       # Modulation index (Hz)
n = 5000            # Number of samples

pi = java.lang.Math.PI   # define pi

t = Double1d.range(n) * ts
# t is a 5000 element array holding time values

signal = SIN(2 * pi * fc * t * (1 + beta * COS(2 * pi * fm * t)))
#create the modulated signal with modulation frequency fm and carrier
#frequency fc, t is the array we created above for the time elements.

spectrum = ABS(FFT(Complex1d(signal)))
#spectrum holds the absolute value (ABS) of the FFT of the signal.
#We need to handle these arrays as Complex1d rather than Double1d.

freq = Double1d.range(n) / (n * ts)
#The frequency values for the spectrum.

# Repeat with apodizing
spectrum2 = ABS(FFT(Complex1d(HAMMING(signal))))
```

Example 5.9. FFT of a modulated signal, with and without HAMMING smoothing

5.3.2. FFT_PACK

HIPE offers a Java implementation of FFT_PACK for the fast calculation of the Discrete Fourier Transform for `Complex1d` arrays and for `Double1d` arrays. Specialized classes are available to operate on `Double1d` arrays that contain data of even or odd symmetry. FFT_PACK aims to make use of the data properties (real only, symmetry) in order to reduce execution run-time.

FFT_PACK (Discrete Fourier Transform for complex data) in *HCSS User's Reference Manual*

`FFT_PACK` transforms complex input into complex output. For the numbers from 2 to 10,000, `FFT_PACK` is faster than `FFT` in more than 55% of all cases. `FFT_PACK` is slower than `FFT` if the length of the input array is a power of 2. `FFT_PACK` is also slower than `FFT` if the prime factor decomposition of the length of the input array contains a large value.

```
# Handle these arrays as Complex1d rather than Double1d.
```

```
spectrum = ABS(FFT_PACK(Complex1d(signal)))
# spectrum now holds the absolute value (ABS) of the FFT of signal.
```

Example 5.10. Transforming a signal into the modulus of its spectrum.

RealDoubleFFT#ft (Discrete Fourier Transform for real data) in *HCSS User's Reference Manual*

`RealDoubleFFT#ft` transforms real input into complex output. It makes no assumption about the symmetry of the input data. To speed up the transform, `RealDoubleFFT#ft` only calculates values which cannot be determined from conjugate symmetry (recall that for real input, $X_k = (X_{N-k})^*$). Given an input of length N, `RealDoubleFFT#ft` outputs only the first `floor(N/2)+1` values of the FFT.

```
from herschel.ia.numeric.toolbox.xform.util import RealDoubleFFT
# Initialization of the RealDoubleFFT to a certain length N must be performed
# only once for each new array length
N = 10
signal = Double1d.range(10)
rdfft = RealDoubleFFT(N)
# Create a Complex1d to store the FFT of signal.
spectrum = Complex1d()
# Execute the real double FFT with an explicit copy as the calculation
# invalidates the input signal. Handle the signal array as a Java array.
rdfft.ft(signal.copy().array, spectrum)
# spectrum now holds the FFT of signal
spectrum = ABS(spectrum)
# spectrum now holds the absolute value (ABS) of the FFT of signal.
# Forward transform of real signal of length N
rdfft = RealDoubleFFT(N)
spectrum = Complex1d()
rdfft.ft(signal.toArray(), spectrum)
# Inverse transform of complex spectrum of length N/2+1
recreatedSignal = Double1d(N)
rdfft.bt(spectrum, recreatedSignal.array)
# Normalize signal
signal = recreatedSignal/N
```

Example 5.11. Transforming a real signal into a spectrum.

FFT_PACK_EVEN (Discrete Cosine Transform for real data with even symmetry) in *HCSS User's Reference Manual*

`FFT_PACK_EVEN` implements the Discrete Cosine Transform. It transforms real input into real output, assuming data with even symmetry of the form $(x_0, x_1, \ldots, x_{N-2}, x_{N-1}, x_{N-2}, \ldots, x_1)$. `FFT_PACK_EVEN` is fast for input lengths for which the input array length *minus 1* can be decomposed into small prime factors.

```
N_extended = signal.size
spectrum = FFT_PACK_EVEN(signal[0:N_extended/2+1])
# spectrum now holds the positive real frequencies of the FFT of signal
# (assuming that signal has even symmetry)
```

Example 5.12. Transforming a real signal with even symmetry into a spectrum.

FFT_PACK_ODD (Discrete Sine Transform for real data with odd symmetry) in *HCSS User's Reference Manual*

`FFT_PACK_ODD` implements the Discrete Sine Transform. It transforms real input into imaginary output (the imaginary part of a complex array) assuming data with odd symmetry of the form $(0, x_0, x_1, \ldots, x_{N-1}, 0, -x_{N-1}, \ldots, -x_0)$. `FFT_PACK_ODD` is fast for input lengths for which the input array length *plus 1* can be decomposed into small prime factors.

```
N_extended = signal.size
```

```
spectrum = FFT_PACK_ODD(signal[1:N_extended/2])
# spectrum now holds the positive imaginary frequencies of the FFT of signal
# (assuming that signal has odd symmetry)
```

Example 5.13. Transforming a real signal with odd symmetry into a spectrum.

5.3.3. Selecting the right Fourier transform

If run-time efficiency is critical, it is necessary to be careful about selecting the Fourier transform suitable for the job at hand. The guidelines below help select the right Fourier transform to minimize execution run-time, based on N, i.e. the length of the input array:

- If N is a power of 2, i.e. $N = 2^m$ for an integer m, then use `FFT`.
- If the largest value in the prime number decomposition of N is relatively large, then compare execution run-time between `FFT` and `FFT_PACK`.
- If the input is real, use `RealDoubleFFT` instead of `FFT_PACK`.
- If input is real and multiple transforms need to be performed on different inputs of the same length, create a single `RealDoubleFFT` object and perform all transforms using the same object.
- If the input is real and has even symmetry, use `FFT_PACK_EVEN`. Note that for fast execution, the length of the input to `FFT_PACK_EVEN` minus one should decompose into small factors.
- If the input is real and has odd symmetry, use `FFT_PACK_ODD`. Note that for fast execution, the length of the input to `FFT_PACK_ODD` plus one should decompose into small factors.

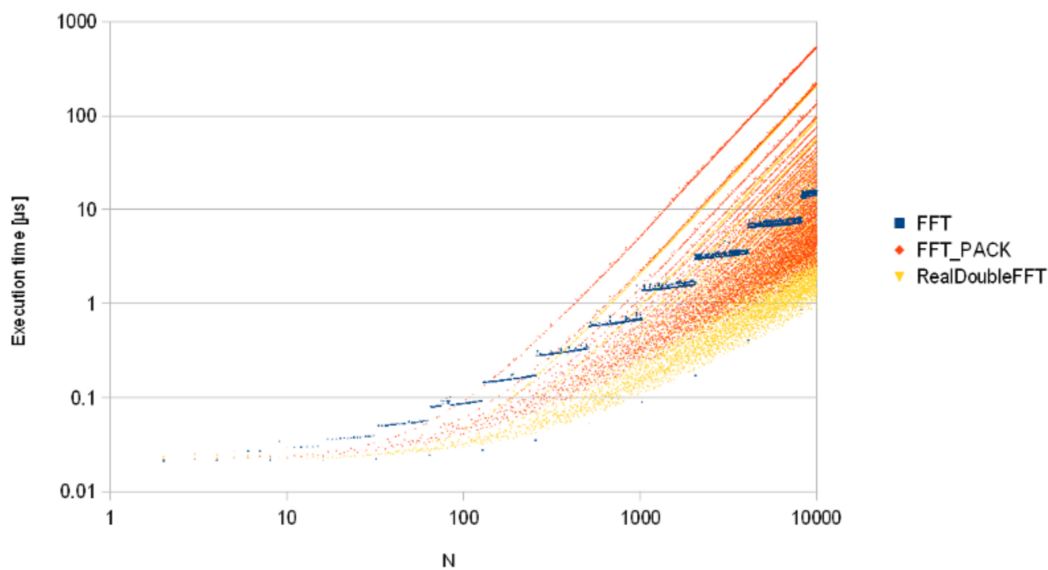


Figure 5.1. The speed of `FFT_PACK` is dependent on input length: if the input length can be factored into small numbers, `FFT_PACK` is faster than `FFT`; if the input length contains a large prime factor, `FFT` is faster than `FFT_PACK`. `FFT` is also faster when input length is a power of two. For strictly real input, `RealDoubleFFT` is always faster than `FFT_PACK`.

5.3.4. Inverse Fourier transforms

HIPE offers inverse Fourier transforms which correspond to each forward Fourier transform.

Table 5.2. Options for the inverse Fourier Transforms. Note that the output of `RealDoubleFFT#bt` depends on the value of N with which the `RealDoubleFFT` object was created.

Name	Input type	Output type	Output length	Notes
IFFT	Complex1d	Complex1d	N	
	$x_n = \frac{1}{N}\sum_{k=1}^{N-1} X_k e^{ikn\frac{2\pi}{N}}, \quad n=0,\ldots,N-1.$			
IFFT_PACK	Complex1d	Complex1d	N	
	$x_n = \sum_{k=1}^{N-1} X_k e^{ikn\frac{2\pi}{N}}, \quad n=0,\ldots,N-1.$			
RealDoubleFFT#bt	Complex1D	Double1d	2L-2 (N even) 2L-1 (N odd)	L is the length of the input to RealDoubleFFT#bt. N is the length of input to RealDoubleFFT#ft and is fixed when the RealDoubleFFT object is created. Outputs strictly real data.
	N even: $x_n = a_0 + 2\sum_{k=1}^{N/2-1}\left(a_k \cos kn\frac{2\pi}{N} - b_k \sin kn\frac{2\pi}{N}\right) + a_{N/2}(-1)^n, \quad n=0,\ldots,N-1.$ N odd: $x_n = a_0 + 2\sum_{k=1}^{(N-1)/2}\left(a_k \cos kn\frac{2\pi}{N} - b_k \sin kn\frac{2\pi}{N}\right), \quad n=0,\ldots,N-1.$			
IFFT_PACK_EVEN	Double1d	Double1d	N	Inverse discrete cosine transform.
	$x_n = a_0 + 2\sum_{k=1}^{N/2-1}\left(a_k \cos kn\frac{2\pi}{N} - b_k \sin kn\frac{2\pi}{N}\right) + a_{N/2}(-1)^n, \quad n=0,\ldots,N-1.$			
IFFT_PACK_ODD	Double1d	Double1d	N	Inverse discrete sine transform.
	$x_n = 2\sum_{k=1}^{N-1} c_k \sin kn\frac{\pi}{N}, \quad n=1,\ldots,N-1.$			

5.3.5. Normalization

The application of the forward transform followed by the application of the inverse transform is an identity operation if the output of the inverse transform is normalized.

Table 5.3. For the following normalizations, assume that the signal has N elements.

Forward and inverse transforms	Normalization to identity
FFT, IFFT	`# Automatically normalized` `signal = IFFT(FFT(signal))`
FFT_PACK, IFFT_PACK	`signal = IFFT_PACK(FFT_PACK(signal))/N`
RealDoubleFFT#ft,	`# Forward transform of real signal of length N` `rdfft = RealDoubleFFT(N)`

<table>
<tr><th>Forward and inverse transforms</th><th>Normalization to identity</th></tr>
<tr><td>RealDoubleFFT#bt</td><td><pre>c = Complex1D()
rdfft.ft(signal.toArray(), c)
# Inverse transform of complex spectrum of length N/2+1
recreatedSignal = Double1d(N)
rdfft.bt(c, recreatedSignal.array)
# Normalize signal
signal= recreatedSignal/N</pre></td></tr>
<tr><td>FFT_PACK_EVEN,
IFFT_PACK_EVEN</td><td><pre>signal = IFFT_PACK_EVEN(FFT_PACK_EVEN(signal))/(2*N-2)</pre></td></tr>
<tr><td>FFT_PACK_ODD,
IFFT_PACK_ODD</td><td><pre>signal = IFFT_PACK_ODD(FFT_PACK_ODD(signal))/(2*N+2)</pre></td></tr>
</table>

5.4. Power spectrum

With the `PowerSpectrum` class you can create the power spectrum of each column of a Table Dataset. Table dataset that are suitable for power spectrum conversion typically contain a column bearing units of time, plus other columns of quantities from which to compute power spectra. Since real signals sometimes contain unwanted strong excursions, called glitches or spikes, that will dominate the power spectrum, the Task includes a simple deglitcher, that detects and removes such events from the data stream, replacing them with an average of the surrounding data.

The *Power Spectrum Viewer*, a graphical interface wrapping the functionality of this class, is described in the *Data Analysis Guide*.

You can obtain your power spectra by invoking the `getPowerSpectrum` method on the `PowerSpectrum` class. The method takes the following arguments:

- *`table`*: the input Table Dataset.
- *`flimit`*: the inverse cut-off frequency (default 0.1).
- *`sigma`*: the deglitcher threshold (default 4).
- *`deglitch`*: boolean, activates the deglitcher if `true` (default).
- *`timeColumn`*: a `Column` containing time information.

The inverse cut-off frequency determines the length of the intervals into which the data timeline is subdivided before performing the FFT. Each of these datasets is Fourier transformed individually, and the resulting power spectra are quadratically co-added to yield a power spectrum with a better S/N ratio, that is, a higher cut-off frequency will yield a better S/N for the resulting power spectrum.

The sigma value controls a simple sigma/kappa deglitcher, that eliminates all datapoints that are more than sigma (default = 4) times the standard deviation away from the mean. After eliminating these data points the procedure is repeated iteratively until no more data can be discarded.

The `getPowerSpectrum` method has the following variants:

- `getPowerSpectrum(table)`
- `getPowerSpectrum(table, timeColumn)`
- `getPowerSpectrum(flimit, table)`
- `getPowerSpectrum(flimit, table, timeColumn)`

- `getPowerSpectrum(flimit, sigma, table)`
- `getPowerSpectrum(flimit, sigma, table, timeColumn)`
- `getPowerSpectrum(flimit, sigma, deglitch, table)`
- `getPowerSpectrum(flimit, sigma, deglitch, table, timeColumn)`

5.5. Convolution

Convolution is currently supported for `Double1d` arrays. A direct convolution algorithm is used, although a future release might implement Fourier convolution to improve the speed for large arrays and large kernels. An example of its use is given in Example 5.14.

```
x = Double1d.range(100)
# Create array [0.0, 1.0, 2.0 ... 99.0]
kernel = Double1d([1,1,1])
#provide kernel for the convolution
f = Convolution(kernel)
#create the convolution
y = f(x)
#apply it to the array x. The result is in array y
```

Example 5.14. Example of the use of the convolution algorithm.

This illustrates a general approach with the numeric library i.e. general function objects may be instantiated using parameters to create a customised function which can then be applied to one or more sets of data.

The constructor of the `Convolution` class allows optional *keyword* arguments to be specified, to further customise the function:

- The '`center`' parameter allows selection of a causal asymmetric filter for time domain filtering or a symmetric filter for spatial domain filtering.
- The '`edge`' parameter controls the handling of edge effects, as well as allowing a choice between periodic (circular) and aperiodic convolution.

The following examples show construction of filters using these options:

Note

Make sure you have input the following import line before trying these out.

```
from herschel.ia.numeric.toolbox.filter.Convolution import *
```

Example 5.15. Importing the Convolution module.

Use zeroes for data beyond edges, causal:

```
f = Convolution(kernel, center=0, edge=ZEROES)
```

Example 5.16. Create a convolution function with zeroes beyond the edges.

Circular convolution, causal:

```
f = Convolution(kernel, center=0, edge=CIRCULAR)
```

Example 5.17. Create a convolution function with circular wrapping beyond the edges.

Repeat edge values, causal:

```
f = Convolution(kernel, center=0, edge=REPEAT)
```

Example 5.18. Create a convolution function with value repetition beyond the edges.

Use zeroes for data beyond edges with centred kernel:

```
f = Convolution(kernel, center=1, edge=ZEROES)
```

Example 5.19. Create a centred convolution function with zeroes beyond the edges.

Circular convolution with centred kernel:

```
f = Convolution(kernel, center=1, edge=CIRCULAR)
```

Example 5.20. Create a centred convolution function with circular wrapping beyond the edges.

Repeat edge values with centred kernel:

```
f = Convolution(kernel, center=1, edge=REPEAT)
```

Example 5.21. Create a centred convolution function with value repetition beyond the edges.

5.6. Boxcar and Gaussian filters

Finite Impulse Response (FIR) filters and symmetric spatial domain filters can be defined by instantiating the `Convolution` class with appropriate parameters. *In addition, special filter functions are provided for Gaussian filters and box-car filters* :

```
from herschel.ia.numeric.toolbox.filter.Convolution import *

f = GaussianFilter(5, center=1, edge=ZEROES)
f = BoxCarFilter(5, center=0, edge=ZEROES)
```

Example 5.22. Creating different filtering functions using the Convolution module.

These filters are subclasses of `Convolution` and hence inherit the use of similar keyword arguments.

5.7. Interpolation

Interpolation functions are provided for a variety of common interpolation algorithms.

Example 5.23 illustrates the use of the currently available interpolation functions.

```
# Create the array x [0.0, 1.0, 2.0, ..., 9.0]
x = Double1d.range(10)
print x # [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
# Create an array y which contains the sine of each element in x
y = SIN(x)
# u contains the values at which to interpolate
u = Double1d.range(80) / 10 + 1
print u #[1.0,1.1,1.2,1.3....8.6,8.7,8.8,8.9]
# Linear interpolation
# This sets up the interpolation, linear x-y fit
# Interpolate at specified values
interp = LinearInterpolator(x,y)
# Prints out the values interpolated at each position noted in array u
```

```
print interp(u) #[0.8414709848,0.848253629....0.5275664375,0.4698424613]

# NearestNeighbour and CubicSpline interpolation may be performed
# in the same way:

# Cubic-spline interpolation
interp = CubicSplineInterpolator(x,y)

# Nearest-neighbour interpolation
interp = NearestNeighborInterpolator(x,y)
```

Example 5.23. Interpolation functions in DP

The result of the interpolations used in the above example is illustrated in Figure 5.2.

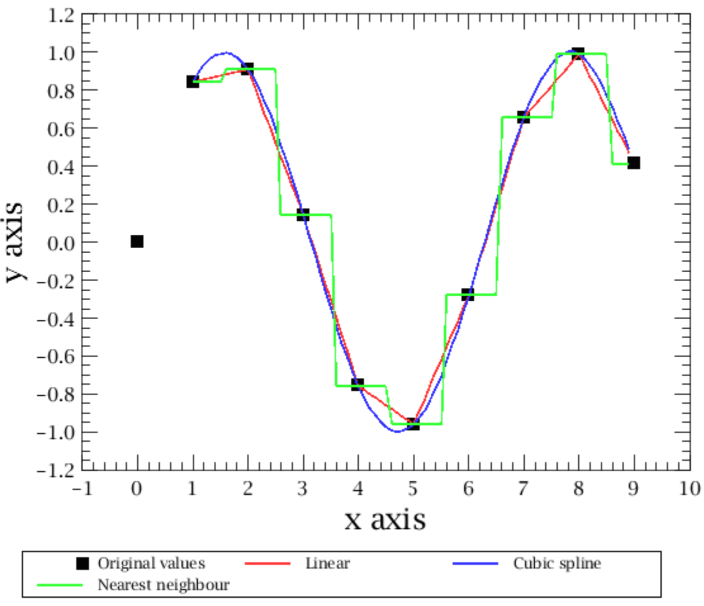


Figure 5.2. Illustration of various forms of interpolation functions.

5.8. Fitting data

Here we provide information on the basic linear and non-linear fitting routines available in HIPE.

5.8.1. General approach

Input Data: The fitter package expects your data to be in two datasets that are related to each other. Typically, these are `Double1d` arrays, e.g.,

```
# Data points: each element in x and y define a data point
x = Double1d.range(12)    # Make x vector (the data positions/channels)
y = Double1d([1.0,1.2,0.9,2.2,3.3,\
 4.5,3.6,2.7,1.8,1.2,1.0,1.1])   # Make y vector (the data values)
```

Example 5.24. Defining some X-Y data points.

Model Selection: Fitting means adjusting the parameters of a known function, called *model* , so that it best matches the input data. This toolbox provides some pre-defined linear models as well as non-linear models. Viewing your data will hopefully give you some hints about what function model would reflect your input data. For example, if it seems to be polynomial of a certain degree, you would choose a `PolynomialModel`.

Note

For the case of non-linear fitters (e.g., used with Gaussians) it is also necessary to provide initial guesses in the form of a parameter set to the model before invoking a fitter. The closer the initial guess for the parameter set to the true values the higher the likelihood that the minimisation will not find a local minimum with wrong/unrealistic parameter estimation.

An example of the use of a linear fitter:

```
# Choose a model: 4th degree polynomial
myModel = PolynomialModel(4)
# Create a fitter and feed it your positions/channels along the array
# (x, a Double1d array) and your model
myFitter = Fitter(x, myModel)
```

Example 5.25. Fitting data with a polynomial model (linear).

Or for a non-linear fitter applied to our array 'x':

```
myModel = GaussModel()
peak = 4.5
channel = 5.5
sigma = 1.0  # Note that sigma is not the FWHM
initialvalues = Double1d([peak, channel, sigma])
# Apply the initial estimates: peak height, channel position and
# sigma of gaussian
myModel.setParameters(initialvalues)
# Choose non-linear fitter to use
myFitter = AmoebaFitter(x, myModel) # see later section on available fitters
```

Example 5.26. Fitting data with a gaussian model (non-linear).

Fit Execution (with and without weights):

```
# Now actually fit the data values at each x position (the y array) to the model
fitresults = myFitter.fit(y)
# Or with associated weights array
fitresults = myFitter.fit(y, yWeights)
```

Example 5.27. Executing the fit with or without parameters.

Results Now the fitter has done its job. We can print the results (`fitresults`) to see the parameters fitted.

```
print fitresults  # from using the polynomial fitter
# [1.0993589743591299,-1.1096331908843398,0.8923489704745665,
# -0.14688390313399513,0.006825466200470528]
# provides coefficients of the polynomial fit
print fitresults  # from using the Gaussian fitter
# [3.751009700481534,5.353351564022887,2.5098951536394383]
# Peak of fit, channel of Gaussian peak, sigma of Gaussian
```

Example 5.28. Printing the results of the fitting.

The fit parameters model are computed and we can start using that model to e.g. re-sample your model fit data:

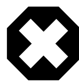
```
# Re-sample with equally spaced x data points and a finer grid:
xs = Double1d.range(1200) / 100     # Re-sampled x positions
ys = myModel(xs)                    # Computed y data points
# a plot of xs versus ys plots out 1200 points with the fit.
```

Example 5.29. Re-sampling the fit data according to the model.

Statistical Information The above procedure demonstrates how to use the fit package to fit your data against a certain model. However, it does not tell you how good the fit actually is. The fitters provide ways to extract such information from the fit.

```
# After fitting
print myFitter.getChiSquared()        # Goodness of the fit
# e.g., 2.5765684980727577 for Gaussian fit
print myFitter.autoScale()            # How well does the data fit the model.
# e.g., 0.5350564350372312 for Gaussian fit
print myFitter.getStandardDeviation() # Standard deviations for the parameters.
# e.g., [0.30907540430060004,0.24531121048289006,0.2525757390634412]
# for Gaussian fit parameters
print myFitter.getHessian()           # Retrieve the Hessian matrix
es = myFitter.monteCarloError(xs)     # Errors on the resampled datapoints
# es is now an error array with a length the same as "xs" -- 1200 samples
```

Example 5.30. Retrieving the statistical indicators of the goodness of fit.



Warning

If you use `getStandardDeviation` (as in the example above) to obtain the standard deviation array, please note that the results are scaled by a noise estimation value as stated in the method description in the URM in *HCSS User's Reference Manual* . You can use the method `myFitter.getScale()` to know this value or:

```
covMat = myFitter.getCovarianceMatrix()
dimension = covMat.getDimension(0) # A square matrix
stdDev = Double1d(dimension)
for i in range(dimension):
   stdDev[i] = SQRT(covMat[i,i])
```

Example 5.31. Retrieving the unscaled standard deviation from the fit.

to obtain the unscaled value of the standard deviations from the covariance matrix (the inverse of the Hessian Matrix).

5.8.2. Available linear models

There are several models that can be used for linear fitting.

In the descriptions below, the models provide parameter fit values $p_0, p_1 \dots p_k$.

Note

In the following examples the parameter subscripts match the position of the parameter in the output array (`fitsresult` in the previous section). So p_0 will be the first element of the `fitsresult` array, p_1 the second one, and so on.

BinomialModel in *HCSS User's Reference Manual*, which allows for the fitting of a binomial model with two variables: $f(x,y:p) = \Sigma\, p_k\, x^k\, y^{(d-k)}$, where *d* is the degree. *Usage: BinomialModel(4)* – provides a binomial model of degree 4.

PolynomialModel in *HCSS User's Reference Manual*, which allows for the least squares fitting of a polynomial to the data: $f(x:p) = \Sigma\, p_k\, x^k$. *Usage: PolynomialModel(3)* – provides a third order polynomial fitting of the data.

SineAmpModel in *HCSS User's Reference Manual*, which allows for the fitting of cosine and sine waves of a given frequency to get amplitudes – $f(x:p) = p_0 \cos(2\,\pi\, f\, x) + p_1 \sin(2\,\pi\, f\, x)$, where *x* is the data. *Usage: SineAmpModel(f)* – which provides cosine/sine fits with a frequency, *f*.

PowerModel in *HCSS User's Reference Manual*, which allows for the fitting of a power law of order k: $f(x:p) = p_0\, x^k$. *Usage: PowerModel(3)* – provides a third-order power-law fit.

SplinesModel in *HCSS User's Reference Manual*, which allows for the fitting of a cubic splines with arbitrary knots settings. *Usage: SplinesModel(Double1d([12.5, 15.8, 17.7]))* – provides a cubic splines fit with three knots.

5.8.3. Available non-linear models

There are a number of models that can be used for non-linear fitting. For fitting of these models we need initial values (guesses) for parameters labelled p_0, p_1 and p_2 (see example given in Section 5.8.1).

ArctanModel in *HCSS User's Reference Manual*, which allows for the fitting of a general arctan function – $f(x{:}p) = p_0 \arctan(p_1 (x - p_2))$. *Usage: ArctanModel()*

ExpModel in *HCSS User's Reference Manual*, which allows for the fitting of a general exponential function – $f(x{:}p) = p_0 \exp(p_1 x)$. *Usage: ExpModel()*

LorentzModel in *HCSS User's Reference Manual*, which allows for the fitting of a Lorentz function – $f(x{:}p) = p_0 (p_2^2/((x - p_1)^2 + p_2^2))$. *Usage: LorentzModel()*

PowerLawModel in *HCSS User's Reference Manual*, which allows for the fitting of a general power-law function – $f(x{:}p) = p_0 (x - p_1)^{p_2}$. *Usage: PowerLawModel()*

SincModel in *HCSS User's Reference Manual*, which allows for the fitting of a sinc function – $f(x{:}p) = p_0 \sin((x - p_1)/p_2)/(x - p_1)/p_2$. *Usage: SincModel()*

SineModel in *HCSS User's Reference Manual*, which allows for the fitting of a general cosine/sine wave – $f(x{:}p) = p_1 \cos(2 \pi p_0 x) + p_2 \sin(2 \pi p_0 x)$. *Usage: SineModel()*

GaussModel in *HCSS User's Reference Manual*, which allows for the fitting of a 1-D gaussian – $f(x{:}p) = p_0 \exp(-0.5 ((x - p_1) / p_2)^2)$, where p_0 is the amplitude, p_1 the x-shift (from zero) and p_2 the sigma of the fit, with initial values of 1.0, 0.0 and 1.0 respectively. Note that *Gauss2DModel* produces a fit to 2D data. *Usage: GaussModel()*

SincGaussModel in *HCSS User's Reference Manual*, which allows for the fitting of the convolution of a 1-D sinc function with a 1-D gaussian – $f(x{:}p) = p_0 \exp(-b^2) (\mathrm{erf}(a - ib) + \mathrm{erf}(a + ib))/ (2\, \mathrm{erf}(a))$, with $a = p_2 / (2^{0.5} p_3)$ and $b = (x - p_1) / (2^{0.5} p_3)$, where p_0 is the amplitude, p_1 is the x-shift (from zero), p_2 is the width of the sinc function (distance between first zero-crossings divided by 2π), and p_3 is the width of the Gaussian function (sigma), with initial values of 1.0, 0.0, 1.0 and 1.0 respectively. *Usage: SincGaussModel()*

User supplied non-linear function, which allows for fitting a function (linear or non-linear) constructed by the user. This function must be put in a Jython class and optionally the user could provide an analytical calculation of the partial derivatives with respect to the parameters (otherwise they are calculated numerically). This is shown in the following example for the following function of four parameters: $f(x{:}p) = p0/(1+(x/p1)^2)^{p2} + p3$ (the so called beta-profile):

```
from herschel.ia.numeric.toolbox.fit import NonLinearPyModel

class BetaModel(NonLinearPyModel):
# the full 4-parameter beta-model with partial derivatives
# f(x:p) = p0/(1+(x/p1)**2)**p2 + p3
#
  npar = 4
  def __init__(self):
    # Constructor
    NonLinearPyModel.__init__(self, self.npar)
    self.setParameters(Double1d([1,1,-1,1]))
  #
  def pyResult(self,x,p):
    model = p[0]/(1.0 + (x/p[1])**2)**p[2] + p[3]
    return model
  #
```

```
  def pyPartial(self,x,p):
    # the partial derivatives
    arg1 = 1.0 + (x/p[1])**2
    dp = Double1d(self.npar)
    #
    dp[0] = 1.0/arg1**p[2] # df/dp0
    dp[1] = 2.0*p[0]*p[2]*x*x/((p[1]**3)*arg1**(p[2]+1.0)) # df/dp1
    dp[2] = -p[0]*Math.log(arg1)/arg1**p[2] # df/dp2
    dp[3] = 1.0 # df/dp3
    return dp
  def myName( self ):
    # Return an explicatory name (String). Optional.
    return "beta-profile: f(x:p) = p[0]*{1 + (x/p[1])2}^p[2] + p[3]"
```

Example 5.32. Creating a custom non-linear fitting model.

Once we define the function as shown in the example then we can proceed as before and create a model and then perform the fitting using either the Lavenberg-Marquardt or Amoeba fitters:

```
bm = BetaModel()
bm.setParameters(Double1d([10.0,1.0,-2.0,5.0]))
myfit = LevenbergMarquardtFitter(x, bm) # see section on available fitters below
# or myfit = AmoebaFitter(x, bm)
result = myfit.fit(y)
print result
```

Example 5.33. Using a custom fitting model.

5.8.4. Compound and mixed models

It is possible to add two models, e.g. if one wants to fit a spectral line (a Gaussian) on a background (a Polynomial). The resulting model is non-linear.

```
myModel = GaussModel()          # Define a Gaussian
myModel += PolynomialModel(1)   # Add a Polynomial to it of order 1. Only with +=
print myModel.toString()        # Information about the model
```

Example 5.34. Fitting a line using two models at the same time.

More models can be added if wished.

5.8.5. Available fitters

All the following fitters are used as follows (the example uses `Fitter`):

```
myFitter = Fitter(xDataPoints, model)
```

Fitter in *HCSS User's Reference Manual*. Fitter for linear models. You create a fitter by providing the model assumption and the x points of the data. With that information you compute the parameters within the model by fitting the y data points. Once the computation of those parameters is done, you can extract statistical information from the fitter.

LevenbergMarquardtFitter in *HCSS User's Reference Manual*. Fitter for non-linear models. The LMFitter is a gradient fitter, which means that it goes downhill from the starting location until it cannot go down anymore. There is no guarantee that the minimum found is an absolute or global minimum. If the chisq-landscape is multimodal it ends in the first minimum it finds. See also Numerical Recipes, Ch 15.5.

AmoebaFitter in *HCSS User's Reference Manual*. Fitter for non-linear models. The AmoebaFitter implements the Nelder-Mead simplex method. It comes in 2 varieties, one where the simplex simply goes downhill (temperature = 0) and one which implements an annealing scheme. Depending on the temperature, the simplex sometimes takes an uphill step, while a downhill steps always is taken. This way it is able to escape from local minima and it has a better chance of finding the global minimum.

No guarantee, however. *AmoebaFitter* is also able to handle limits on the parameter range. Parameters stay within the limits when they are set. See also Numerical Recipes, Ch. 10.4 and 10.9.

SingularValueDecompositionFitter in *HCSS User's Reference Manual*. Linear fitter based on Singular Value Decomposition (described in the section called “Singular value decomposition”). Much more robust in case of (nearly) degenerated models, at the cost of more CPU use. See Numerical Recipes for more information.

5.8.6. Setting the fitter tolerance

The iterative fitters `LevenbergMarquardtFitter` and `AmoebaFitter` have a *tolerance* value, set by default to 0.01, against which the chi square value from each iteration is compared. When the chi square value is lower than the tolerance, the iteration stops.

When the noise level of your data is low compared to the tolerance, iterations stop too early. This is shown in the following example, using mock data with noise of order 0.001.

```
# The data
array = Double1d([1.001, 1.004, 1.005,1.002,1.006,1.007,1.007,1.009,1.01,1.011])
time = Double1d(range(10))
# Fitting data with the LevenbergMarquardtFitter
order = 1
poly = PolynomialModel(order)
poly.setParameters(Double1d(order+1))
fitter = LevenbergMarquardtFitter(time,poly)
lineFit = fitter.fit(array)
# Plotting the results
plot = PlotXY()
plot.addLayer(LayerXY(time, array, line=Style.MARKED, symbol=14, \
     symbolSize=5, color=java.awt.Color.BLUE))
plot.addLayer(LayerXY(time, poly(time), line=Style.MARKED, symbol=14, \
     symbolSize=5, color=java.awt.Color.RED))
# Fitting data with the Fitter
fitter = Fitter(time,poly)
linefit = fitter.fit(array)
plot.addLayer(LayerXY(time, poly(time), line=Style.MARKED, symbol=14, \
     symbolSize=5, color=java.awt.Color.GREEN))
plot.xtitle='Time'
plot.ytitle='Array'
```

Example 5.35. Plotting the results of a polynomial fitting.

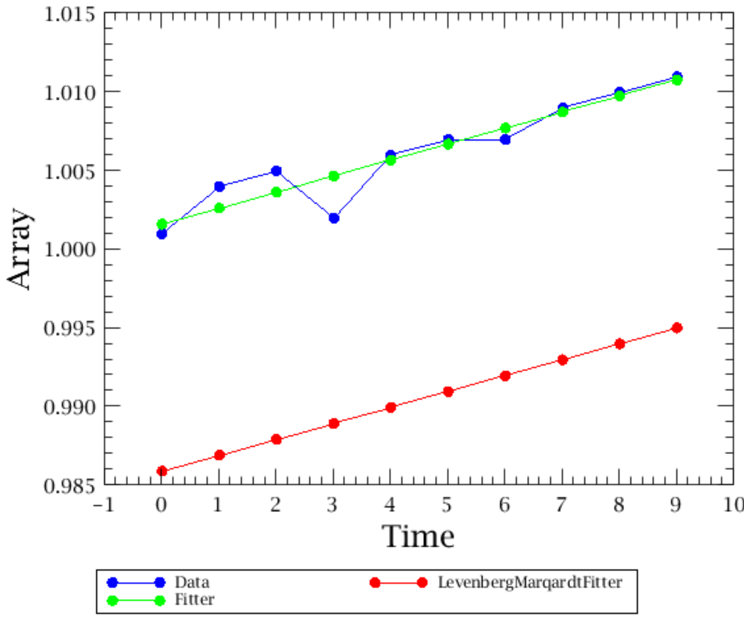


Figure 5.3. Fitting data iteratively with tolerance set too high.

Note how the fit given by `LevenbergMarquardtFitter` is offset with respect to the data. You can improve the fit by setting the tolerance to a lower value:

```
fitter = LevenbergMarquardtFitter(time, poly)
fitter.tolerance = 0.00001
```

Example 5.36. Setting the tolerance for the LevenbergMarquardt fitter.

However, if you have a linear model like a `PolynomiaModel`, as in the previous example, the preferred fitter is `Fitter` or `SingularValueDecompositionFitter`.

5.8.7. 1D fit example

The following example shows how a polynomial can be fitted to a set of 1D data.

```
# Create some data
x = Double1d([3,4,6,7,8,10,11,13]) # These are the positions of the 1D data
y = Double1d([2,4,5,6,5,6,7,9]) # These are the data values at each position
# The created arrays are:
print x  # [3.0,4.0,6.0,7.0,8.0,10.0,11.0,13.0]
print y  # [2.0,4.0,5.0,6.0,5.0,6.0,7.0,9.0]

# Decide that we will fit it with a polynomial

model = PolynomialModel(3)

# The Fitter class expects the 'x' data point positions and the model.
# In the binomial case, a Double2d array of x,y values is required.
# The Fitter class deals with non-iterative models only.
# [Note: For non-linear models the fitter toolbox provides
# the AmoebaFitter and the LevenbergMarquardtFitter]

fitter = Fitter(x, model)

# Now we fit the data values(y); the returned array contains the parameters
# that make up a 3rd degree polynomial.
# Note: the model that we fed into the fitter is modified along the
# way, such that it contains the computed parameters of the polynomial.

poly = fitter.fit(y)

# Printing the fit results (truncate to 3 decimal places to fit in line)

print poly # [-6.921,4.463,-0.543,0.022]

# ..and also getting the Chi-squared. The fitter has already been applied
# and we can use the getChiSquared() method to determine the fit

print "Chi-Squared = ", fitter.getChiSquared()
# Chi-Squared =  0.9933079890409999

# The fitted polynomial can then be applied as a function to interpolate
# between fitted points. Interpolate at 'n' uniformly spaced x values

n = 100
u = MIN(x) + Double1d.range(n + 1) * ((MAX(x) - MIN(x)) / n)

# Apply the model
umodel = model(u)

# Now we can plot the data (x vs y) and the polynomial fit (u vs umodel)
# Set up the plot space
plot = PlotXY()
# Plot x against y in blue.
plot[0] = LayerXY(x, y, name = "Data")
# Overlay a second plot showing the polynomial fit in green.
plot[1] = LayerXY(u, umodel, name = "Fit", color = java.awt.Color.green)
```

The final plotted display should look like Figure 5.4:

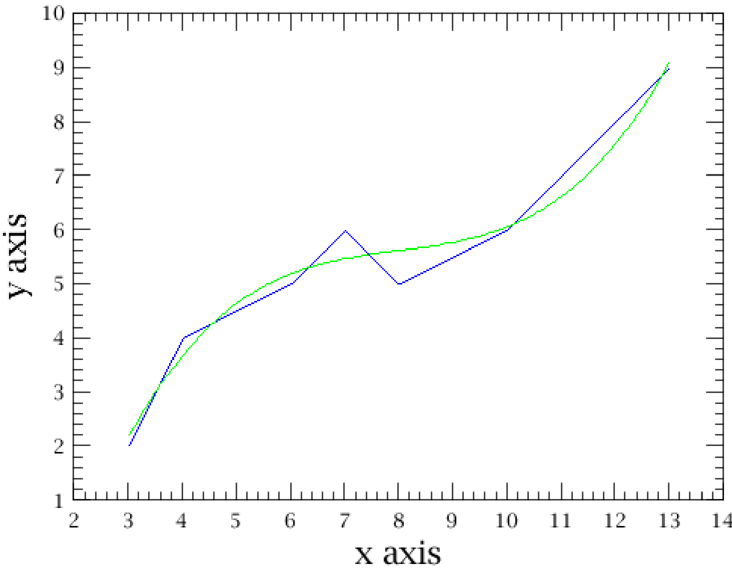


Figure 5.4. Illustration of polynomial fit.

5.8.8. 2D fit example

For 2D data we express the positions at which we have data by a Double2d array. This is a list of x, y positions at which we have known data values that we will fit a 2D Gaussian to. So the `x` array in our previous example is now replaced by a 2D array of data positions. The `y` array has the data values at those positions.

In the following example, an array with values that provide a Gaussian with random noise added is fitted by the Gauss2D model.

```
# We start by making a little routine that creates the data for us.
# The output contains the 'xy' positions as a Double2d array and the data
# values are held in in the Double1d array 'y2'.
def makeData():
# Define some constants
        N = 9                  # We will create an array that is NxN
        a0 = 10.0              # Amplitude of gaussian
        x0 = 0.7               # x position of gaussian
        y0 = -0.3              # y position of gaussian
        s0 = 0.4               # Width
# Make data with an underlying gaussian model.
        x = Double1d.range(N) / 2.0 - 2 # create x values
        NN = N * N # the number of x and y positions (NxN)
        xy = Double2d(NN, 2) # Create empty array of xy positions
        ym = Double1d(NN)    # Create empty array for amplitude of pure Gaussian
        y2 = Double1d(NN)    # Create empty array for Gaussian with noise (our
 data).
# These have amplitude values only.
        rng = java.util.Random( 12345 ) #provide a random amplitude (noise)
# To add to our model Gaussian with a seed value.
        si = 1.0 / s0 #just inverse of Gaussian width to be used
        for i in Int1d.range(NN):
          xy[i,0] = x[i / N] # Fills x positions for our data array
          xy[i,1] = x[i % N] # Fills y positions for our data array
          xx = ( xy[i,0] - x0 ) * si
          yy = ( xy[i,1] - y0 ) * si
          ym[i] = a0 * EXP(-0.5 * xx * xx) * EXP(-0.5 * yy * yy)
          # Fills 1d array with amplitude values...
          y2[i] = ym[i] + 0.2 * rng.nextGaussian()  # ...and adds noise to it
        return xy,y2

# Create the array with a 2D gaussian in it using the above routine.
a = makeData()
```

```
# The first item in "a" has the xy positions in it
xy=a[0]
# The second item has the data values
y2=a[1]

# Define the model to be used in the fit
gaus2d = Gauss2DModel()

# Define the fitter: LevenbergMarquardt, a non-linear fitter is needed for
# a gaussian fit. We could use an AmoebaFitter here also -- user preference.
fitter = LevenbergMarquardtFitter(xy, gaus2d)

# A useful way to make data formats prettier for the printout of our results
F = DataFormatter()
# Find the parameters
param = fitter.fit(y2)
print "Parameters %s" % F.p(param)
# Parameters [ 9.645  0.694  -0.300 0.413]
print "Parameters are:  gaussian height, x position, y position, width"
#Parameters are:  gaussian height, x position, y position, width
# Find the standard deviations of the all four parameters...
stdev = fitter.getStandardDeviation();
print "Stand Devs %s" % F.p(stdev)
#Stand Devs [    0.218     0.009     0.009     0.007]

# ...and the chi-squared for the fit
print "ChiSq      %s" % F.p(fitter.getChiSquared())
#ChiSq          3.552
```

5.8.9. Additional documentation

The following additional documentation is available, maintained directly by the developer of the fit toolbox:

- The Fit Toolbox in Jython.
- Fitter reference documentation. In particular, see the troubleshooting section if you encounter problems.
- Jython examples for the Fit Toolbox.

5.9. Masks

The Numeric library offers two classes for handling data masks:

- `FixedMask` represents a *traditional* mask definition, with different masks (up to 64) defined at different bit offset positions. Note that this class only stores mask *definitions*, with mask data stored in different arrays. For more information and several examples, see the entry in the *User's Reference Manual*: Section 1.150 in *HCSS User's Reference Manual*.
- `PackedMask` instead stores the mask data itself. There is in principle no limit on the number of masks that can be stored in a single `PackedMask` object. For more information and several examples, see the entry in the *User's Reference Manual*: Section 1.292 in *HCSS User's Reference Manual*.

5.10. Matrices

Most of the utilities for dealing with matrices are provided by the numeric.toolbox.matrix package. However, we must not forget that simple vectors are just matrices with just one row (or one column), so even vector classes like `Double1d` provide tools like a `dotProduct` method for scalar multiplication of vectors:

```
x = Double1d([1,2,3,4])
```

```
y = Double1d([1,3,5,7])
print x.dotProduct(y) # 50.0
```

Example 5.37. How to get the dot product of two vectors or matrices.

We now take a closer look at the numeric.toolbox.matrix package and its classes and function objects for matrix manipulation.

Transpose

To transpose a matrix do the following:

```
A = Int2d([[1,2],[3,4],[5,6]])
print TRANSPOSE(A) # [[1,3,5],[2,4,6]]
```

Example 5.38. Transposing a matrix.

Determinant

Use this function to find the determinant of a square matrix given by a `Double2d` array.

```
A = Double2d([[1,2],[3,4]])
print DETERMINANT(A) # -2.0
```

Example 5.39. Finding the determinant of a matrix.

Note: This currently does not work for complex matrices.

Inverse

You can find the inverse of a square matrix as follows:

```
A = Float2d([[1,2],[3,4]])
print INVERSE(A) # [[-2.0,1.0],[1.5,-0.5]]
```

Example 5.40. Inverting a matrix.

Note: This currently does not work for complex matrices.

Matrix multiplication

Use `MatrixMultiply` for matrix multiplication:

```
x = Double2d([[2,4,6],[1,3,5]])
y = TRANSPOSE(x)
z = MatrixMultiply(y)(x)
print z
```

Example 5.41. Multiplying matrices this way returns a matrix.

It is important **not** to use the Jython `*` operator for matrix multiplication. However, the `+` operator performs element-wise addition as expected.

It is also possible to multiply a matrix by a vector as follows (since, as we already pointed out, a vector is nothing more than a matrix with just one row or column):

```
a = Double2d([[1,2,3],[7,5,4],[7,4,9]])
b = Double1d([4,1,7])
print MatrixMultiply(b)(a) # [27.0,61.0,95.0]
```

Example 5.42. Multiplying a matrix by a vector with matrix multiplication.

Warning

The correct syntax to multiply matrix `a` by matrix `b` is `MatrixMultiply(b)(a)`.

LU decomposition

For an *m x n* matrix *A*, LU decomposition returns matrices *P*, *L* and *U* so that *PA* = *LU*:

- *P* is a *permutation matrix*, so that the product *PA* results in a permutation of *A*'s rows. In the class described below, *P* is replaced by an equivalent *permutation vector* ***p***.
- *L* is a unit lower triangular matrix.
- *U* is an upper triangular matrix.

The `LUDecomposition` class provides this functionality. The following example shows how it is used:

```
A = Double2d( [ [1,1,1],[1,2,3],[1,3,6] ] )
print A
# [
# [1.0,1.0,1.0],
# [1.0,2.0,3.0],
# [1.0,3.0,6.0]
# ]
d = LUDecomposition(A)
print d.l # Getting L
# [
# [1.0,0.0,0.0],
# [1.0,1.0,0.0],
# [1.0,0.5,1.0]
# ]
print d.u # Getting U
# [
# [1.0,1.0,1.0],
# [0.0,2.0,5.0],
# [0.0,0.0,-0.5]
# ]
print d.pivot # Getting the permutation vector
# [0,2,1]
```

Example 5.43. Decomposing a matrix to lower and upper matrices.

You can easily verify that the result is correct:

```
print MatrixMultiply(d.u)(d.l)
# [
# [1.0,1.0,1.0],
# [1.0,3.0,6.0],
# [1.0,2.0,3.0]
# ]
```

Example 5.44. Verifying the results of a LU decomposition.

LU gives *A* with the row order changed as described by the permutation vector: row 0, then row 2, then row 1.

Eigenvalue decomposition

The `EigenvalueDecomposition` class provides eigenvalues and eigenvectors of a real matrix. The following examples shows how it can be used:

```
A = Double2d( [[1,1,1],[1,2,3],[1,3,6]] )  # Creating matrix
```

```
evd = A.apply(EigenvalueDecomposition())   # Performing decomposition
D = evd.d  # Obtaining the block diagonal eigenvalue matrix
V = evd.v  # Obtaining the eigenvector matrix
print evd.imagEigenvalues  # Printing the imaginary parts of the eigenvalues
print evd.realEigenvalues  # Printing the real parts of the eigenvalues
print evd.vcond  # Printing the condition (2-norm) of the matrix, defined as
                 #  the ratio of the highest and smallest singular value
```

Example 5.45. Getting the eigenvalues of a matrix after decomposing it.

If A is symmetric, then $A = V D V^T$, where the eigenvalue matrix D is diagonal and the eigenvector matrix V is orthogonal.

If A is not symmetric, then the eigenvalue matrix D is block diagonal with the real eigenvalues in 1-by-1 blocks and any complex eigenvalues, $\lambda + i\,\mu$, in 2-by-2 blocks, $[\lambda, \mu; -\mu, \lambda]$. The columns of V represent the eigenvectors in the sense that $A\ V = V D$. The matrix V may be badly conditioned, or even singular, so the validity of the equation $A = V D V^{-1}$ depends upon `vcond`.

Singular value decomposition

For an $m \times n$ matrix A with $m >= n$, the singular value decomposition is an $m \times n$ orthogonal matrix U, an $n \times n$ diagonal matrix S, and an $n \times n$ orthogonal matrix V so that $A = U S V'$.

The singular values, $\sigma_k = S_{kk}$, are ordered so that $\sigma_0 >= \sigma_1 >= ... >= \sigma_{n-1}$.

The singular value decomposition always exists, so the constructor will never fail. The matrix condition number and the effective numerical rank can be computed from this decomposition.

The `SingularValueDecomposition` class provides this functionality. For more information see the *User's Reference Manual in HCSS User's Reference Manual*.

Matrix equations

Use `MatrixSolve` to solve matrix equations. For example, if you wanted to solve the matrix equation: `A.X = B`:

```
x = MatrixSolve(b)(a)
print x  # [-0.9838709677419352,0.5322580645161287,1.3064516129032258]
```

A note on naming conventions

You might find a bit confusing that some names, like `dotProduct`, start with a lowercase letter and have all the other initials capitalised, while other names, like `MatrixMultiply`, have *all* initials capitalised, and yet other names like `TRANSPOSE` are all in upper case. You can find more about naming conventions in Section 1.31.

5.11. Random numbers

To create pseudorandom numbers you first have to instantiate a *generator*. Three generators are available:

- `RandomUniform`: generates random numbers in the range 0 <= x < 1 if invoked without parameters, like this:

  ```
  myGenerator = RandomUniform()
  ```

  It is also possible to give a maximum value different from 1 to have random numbers created in the range 0 <= x < max:

```
myGenerator = RandomUniform(max)
```

- `RandomGauss`: generates random numbers following a Gaussian distribution.
- `RandomPoisson`: generates random numbers following a Poisson distribution of specified mean value greater than zero. It is instantiated like this:

```
myGenerator = RandomPoisson(mean)
```

  It can only produce integer-type random numbers (`int`, `short` and `long`).

In all cases what is being used under the hood is the Donald Knuth generator (see *The Art of Computer Programming*, Volume 2, Section 3.2.1) as implemented in the `java.util.Random` class.

Once we have a generator in place, how do we create random numbers? The handy feature is that we can create a single scalar random number or an array of any size and dimension we like (as long as it fits in memory). Just put the type of numeric value you want as input, and the output will be the same thing, but populated with random numbers. A few examples:

```
myGenerator = RandomUniform()   # Generating random numbers between 0 and 1
print myGenerator(0.0)          # We want a floating point random number...
# 0.8754230073094597            # ...and there it is (don't expect to get the
                                # same number)
x = Double1d(10)                # Now for an array of ten doubles...
print myGenerator(x)            # We leave it to you to see the result
print myGenerator(Double1d(10)) # Of course you can create the input on the fly
print myGenerator(Int1d(100))   # What's the result of this one? Does it make sense?
```

Example 5.46. Generating random numbers with this utility class.

You might have been puzzled to see a hundred zeroes scroll on your screen after executing the last command of the example. It's not so surprising if we think that we asked the computer to produce *integer* random numbers between zero and one, *excluding one*. The choice of possible values was pretty limited.

If we want to change the *seed* of the random number generator we can do so by the `setSeed` method, which takes a long parameter as an input:

```
myGenerator.setSeed(54653856L)
```

Example 5.47. Setting a seed for a random number generator.

5.12. Numeric integration

In HIPE you can integrate a function defined analytically or a set of *(x, y)* values sampling a function.

5.12.1. Integrating functions

The function to be integrated has to be declared as a class of a `RealFunction` type, containing a method called *calc* which takes one argument, the independent variable.

The following integrators for a standard integration interval [a,b] are available:

- RectangularIntegrator
- RombergIntegrator
- SimpsonIntegrator
- TrapezoidalIntegrator

- GaussianQuad4Integrator
- GaussianQuad5Integrator
- GaussLegendreIntegrator

All these integrators have two arguments for initialisation: the lower limit of integration (a) and the upper limit (b). Once the integrator is initialised and the user function is defined then to perform the integration a method called *integrate()* is executed with an argument the user function. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction

class MyFunction(RealFunction):
  def calc(self,x):
    return x*x

f = MyFunction()
a = -3.0
b = 3.0
i = RombergIntegrator(a, b)
print i.integrate(f) # 18.0
print "Analytical answer: ",(b**3 - a**3)/3.0
```

Example 5.48. Integrating numerically using the Romberg method.

The following special cases of numeric integration are also implemented:

- GaussHermiteIntegrator: for integration with limits (-Inf,+Inf) of a special class of functions

  $$\int_{-\infty}^{+\infty} e^{-x^2} f(x)\,dx$$

- GaussLaguerreIntegrator: for integration with limits [0,+Inf) of a special class of functions

  $$\int_{0}^{+\infty} x^{\alpha} e^{-x} f(x)\,dx$$

  The input for the integrator initialisation is α.

- GaussJacobiIntegrator: for integration with limits [-1,1] for a special class of functions

  $$\int_{-1}^{1} (1-x)^{\alpha}(1+x)^{\beta} f(x)\,dx$$

  The input for the integrator initialisation are α and β.

5.12.2. Integrating discrete values

If you want to integrate a set of *(x, y)* values, you have two choices.

Interpolate first and then apply a suitable integrator. You use interpolation to create a function based on your data set, and then apply one of the integrators described in the previous section. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction

x = 0.1 + 1.9*Double1d.range(11)/10.0 # 11 points between 0.1 and 2.0
```

```
y = 1.0/x

f = CubicSplineInterpolator(x,y) # interpolate first.
a = 0.1
b = 2.0
integrator = SimpsonIntegrator(a, b) # use Simpson's rule

res = integrator.integrate(f) #
print "Result: ",res
print "Analytical result: ",LOG(b) - LOG(a)
```

Example 5.49. Integrating numerically using the Simpson method.

For more information on interpolating discrete values see Section 5.13.

Use the `IntTabulated` function. This function integrates a tabulated set of *(x, y)* data using a five-point Newton-Cotes integration formula:

```
x = Float1d([0.0, .12, .22, .32, .36, \
   .40, .44, .54, .64, .70, .80])
y = Float1d([0.200000, 1.30973, 1.30524, 1.74339, 2.07490, \
   2.45600, 2.84299, 3.50730, 3.18194, 2.36302, 0.231964])
print IntTabulated(x)(y)
# 1.62323157271
```

Example 5.50. Integrating tabular data using Newton-Cotes method.

This function is equivalent to the `INT_TABULATED` function in IDL. For more information on `IntTabulated`, see the *User's Reference Manual*: Section 1.226 in *HCSS User's Reference Manual*.

5.13. Interpolating discrete data

If the objective is to integrate discrete data, this can be done by means of a `FitterFunction`, which is a function that interpolates the given data, with a specific model. For example:

```
from herschel.ia.toolbox.fit import FitterFunction

# x, y are Double1d that represent the abscissas and values of our data
f = FitterFunction(x, y, PolynomialModel(3))                  # Uses a Fitter
g = FitterFunction(x, y, PolynomialModel(2), FitterFunction.AMOEBA)
# Uses an AmoebaFitter
```

Example 5.51. Creating a fitter function with different fitters and models.

If more precise fitting is needed, you can do it by yourself, and then pass the already built fitter (or the model) to the FitterFunction:

```
# x, y are Double1d that represent the abscissas and values of the data
model = PolynomialModel(x)
fitter = AmoebaFitter(x, model)
fitter.setSimplex(params, range) # customize the fitter as you want
fitter.fit(y)
f = FitterFunction(fitter) # or f = FitterFunction(model)
```

Example 5.52. Customising the fitter even setting the simplex.

If one of the defined interpolators suits your needs, it can be used directly, instead of a `FitterFunction`. For example:

```
# x, y are Double1d that represent the abscissas and values of the data
f = CubicSplineInterpolator(x, y)
```

Example 5.53. Creating a cubic spline interpolator.

5.14. Statistics

The following statistics functions are available. Follow the links to go to the corresponding entries in the *User's Reference Manual*.

- ChiSquared in *HCSS User's Reference Manual*: Performs the chi-square statistical test.
- Correlate in *HCSS User's Reference Manual*: Returns the linear Pearson correlation coefficient between two arrays.
- CorrelateMatrix in *HCSS User's Reference Manual*: Returns the linear Pearson correlation coefficients of an M x N matrix.
- Covariance in *HCSS User's Reference Manual*: Returns the covariance between two arrays.
- CovarianceMatrix in *HCSS User's Reference Manual*: Returns the covariance matrix of an M x N matrix.
- Erfc in *HCSS User's Reference Manual*: Returns the complementary error function of an array.
- Erf in *HCSS User's Reference Manual*: Returns the error function of an array.
- GAMMALN in *HCSS User's Reference Manual*: Computes the natural log of the Gamma function.
- GammaP in *HCSS User's Reference Manual*: Computes the incomplete Gamma function P(a,x).
- GammaQ in *HCSS User's Reference Manual*: Computes the complement of the incomplete Gamma function Q(a,x).
- GEOMEAN in *HCSS User's Reference Manual*: Returns the geometric mean value of an array.
- KURTOSIS in *HCSS User's Reference Manual*: Returns the kurtosis excess of an array.
- MEAN in *HCSS User's Reference Manual*: Returns the mean of an array.
- MedianAbsoluteDeviation in *HCSS User's Reference Manual*: Returns the median standard deviation of an array.
- MEDIAN in *HCSS User's Reference Manual*: Returns the median of an array.
- MODE in *HCSS User's Reference Manual*: Returns the mode of an array.
- Sigclip in *HCSS User's Reference Manual*: Returns array values more than *n* sigma from a comparator.
- SKEWNESS in *HCSS User's Reference Manual*: Returns the skewness of an array.
- StatWithNaN in *HCSS User's Reference Manual*: Removes NaN values from arrays for some statistics functions.
- STDDEV in *HCSS User's Reference Manual*: Returns the standard deviation of an array.
- VARIANCE in *HCSS User's Reference Manual*: Returns the variance of an array.
- WeightedMean in *HCSS User's Reference Manual*: Returns the quadratically weighted mean of an array.

5.15. Wavelet transforms

Fourier transforms offer a convenient way to convert any signal to frequencies. The signal is decomposed on a sinusoidal function. Unfortunately, it is difficult to locate in time the frequencies. To avoid

this problem, the signal can be decomposed inside a window of length L, this window being moved along the time axis. If we replace this window by a function converging (rapidly) to zero and select this function so that it forms a base, we have defined a wavelet. We can dilate the wavelet and build a family of wavelets. Each of them catches a band of frequencies. The frequency and time resolutions are linked by the Heisenberg uncertainty principle, which means that we cannot have an infinite resolution in time and frequency, but only a compromise.

This wavelet toolbox implements signal decomposition and synthesis. You can analyse your signal and filter the coefficients, then rebuild your signal.

5.15.1. Continuous wavelet transform

This decomposition is a redundant decomposition useful to visualise how the signal energy is distributed and evolves in a time-scale scalogram. Elementary steps of decomposition are fixed by the number of voices per octave. Performing a decomposition consists of decomposing the signal on octave number + voices/number of voices per octave. Here are the formulae used for synthesis and decomposition.

Formula of an individual wavelet, derived from a "mother wavelet" by scaling and translation, with *a* being the scaling factor and *b* the translation factor:

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{a}}\Psi(\frac{t-b}{a})$$

Formula of continuous wavelet transform of a function *f(x)*:

$$Wf(a,b) = \frac{1}{\sqrt{a}}\int_{-\infty}^{+\infty} f(x)\overline{\Psi\left(\frac{x-b}{a}\right)}dx$$

Formula of inverse wavelet transform:

$$f(x) = \frac{1}{C_\psi}\int_0^{+\infty}\int_{-\infty}^{+\infty} Wf(a,b)\frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right)db\frac{da}{a^2}$$

Formula of the admissibility condition, that must be satisfied for a successful inverse transform:

$$C_\psi = \int_0^{+\infty}\frac{\left|\hat{\psi}(w)\right|^2}{w}dw < +\infty$$

5.15.2. Example

Worked Jython example:

```
from herschel.ia.numeric.toolbox.wavelet import ContinuousWavelet
from herschel.ia.numeric.toolbox.wavelet.wutil import WBorder
# Create dummy signal
from herschel.ia.numeric.toolbox.wavelet.wdemo import WSigGenerator
sigGen = WSigGenerator()
sig = sigGen.getPredefinedRealFunction1()
# Continuous wavelet transform
algo = ContinuousWavelet("MexicanHat")
coefs = algo.decompose(sig, WBorder.SYMMETRIC)
# Change data found in octave 2 and voice 10
octave = 2
voice = 10
data = coefs.getNodeForOctaveAndVoice(octave, voice)
data.multiply(0.25)
# Many functions can be called: add, subtract, multiply, divide, reset
# Rebuild the signal
res = algo.synthesis(coefs)
# Plot the difference
res.subtract(sig)
print MIN(res), MAX(res), STDDEV(res), MEAN(res)
```

```
plot = PlotXY(res)
```

Example 5.54. Transforming a signal with a continuous wavelet.

5.15.3. Modulo Maxima Line

Signal irregularities can be identified with the modulo maxima line or how the irregularities is propagated in the time - scale scalogram. This functionality is available in the wavelet package. First of all, you should activate this functionality, then decompose the signal.

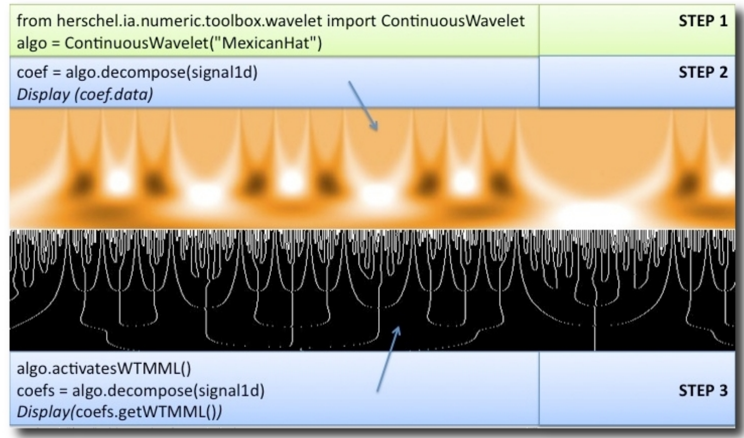


Figure 5.5. Effects of modulo maxima line.

Note

After `algo.activatesWTMML()`, you cannot run `algo.synthesis()`. Instead, you should run `algo.deactivatesWTMML()` and rerun `algo.decompose()`.

5.15.4. The wavelet library

With the exception of the wavelets used with the CWT algorithm, all wavelets are defined in an XML file stored in herschel.ia.numeric.toolbox.wavelet.wibrary.wavelets.xml. You can define another location via the user property file.

The following examples show how to obtain information on the wavelets used in the transformations:

Continuous wavelet

```
from herschel.ia.numeric.toolbox.wavelet.wlibrary import WaveletLoader
wloader = WaveletLoader()
wloader.setCwtPreference()
wavelet = wloader.get("MexicanHat")
realPart = wavelet.getData(10.).real
plot = PlotXY(realPart)
plot.setSubtitleText("Mexican Hat wavelet at scale 10")
plot.getXaxis().setTitleText("Support")
plot.getYaxis().setTitleText("")
```

Example 5.55. Selecting one continuous wavelet.

Discrete wavelet

```
from herschel.ia.numeric.toolbox.wavelet.wlibrary import WaveletLoader
wloader = WaveletLoader()
wloader.setDwtPreference()
wavelet = wloader.get("db2")
```

```
print wavelet
```

Example 5.56. Transforming a signal using a discrete wavelet.

```
db2  # Wavelet name
daubechie  # Family name
2  # Wavelet order
4  # Wavelet length
false  # Applicability of the wavelet: no continuous transform
true  # Applicability of the wavelet: discrete yes
-1  # Vanish moment of the scaling function, -1 means undefined
2  # Vanish moments of the wavelet
orthogonal  # The wavelet is orthogonal
asymmetric  # The wavelet is asymmetric
# Last four lines: filter coefficients of the wavelet
Synthesis low pass filter (LoR)=
  [0.48296291314469025,0.836516303737469,0.22414386804185735,-0.12940952255092145]
Synthesis high pass filter (HiR)=
  [-0.12940952255092145,-0.22414386804185735,0.836516303737469,-0.48296291314469025]
Decomposition low pass filter (LoD)=
  [-0.12940952255092145,0.22414386804185735,0.836516303737469,0.48296291314469025]
Decomposition high pass filter (HiD)=
  [-0.48296291314469025,0.836516303737469,-0.22414386804185735,-0.12940952255092145]
```

5.15.5. Discrete wavelet transform

A continuous wavelet transform produces redundant information and too much data. We can perform an efficiency decomposition if we halve the signal according to the size of the wavelet. Thanks to this dyadic decomposition, the Nyquist condition is respected and the signal can be reconstructed. Two filters will be convolved with our signal:

- A high pass filter, our wavelet catching the detail of the signal.
- A low pass filter, a scaling function getting the approximation of the signal.

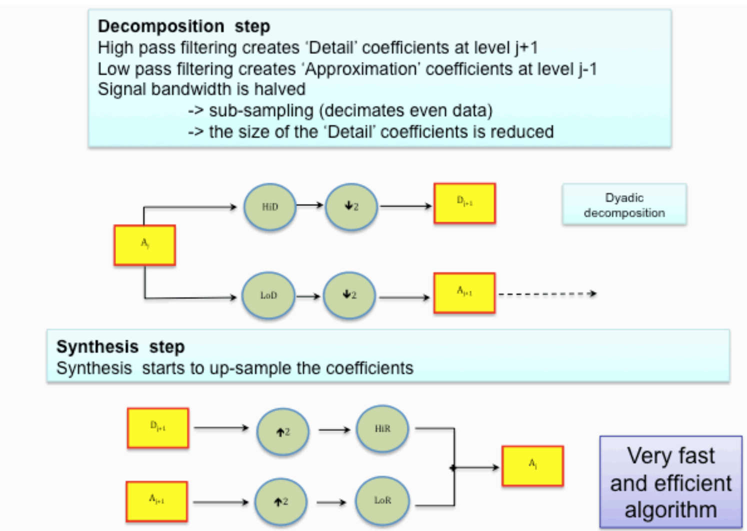


Figure 5.6. Principles of discrete wavelet transform.

Dealing with a one-dimensional image:

```
from herschel.ia.numeric.toolbox.wavelet import DiscreteWavelet
algo = DiscreteWavelet("db3") # Select db3 as wavelet.
# Create dummy signal
from herschel.ia.numeric.toolbox.wavelet.wdemo import WSigGenerator
sigGen = WSigGenerator()
signal1d = sigGen.getPredefinedRealFunction1()
coefs = algo.decompose(signal1d) # Performs wavelet decomposition.
```

```
Display(coefs.data) # display details at each level
PlotXY(coefs.getApproximation().data) # Get and plot approximation.
data = coefs.getDetailForLevel(2) # Get detail for level 2.
data.add(0.25) # Add value 0.25. Other functions: multiply, divide, subtract and
 reset
res = algo.synthesis(coefs) # Rebuilt signal taking into account user changes.
```

Example 5.57. Discrete wavelet transformation of a one dimensional signal.

Dealing with two-dimensional data:

```
from herschel.ia.numeric.toolbox.wavelet import DiscreteWavelet
algo = DiscreteWavelet("db3") # Select discrete wavelet algorithm and db3 wavelet.
# Load the standard test image included with the package
from herschel.ia.numeric.toolbox.wavelet.wdemo import WReadImage
signal2d = WReadImage.loadImage("Lena.png")
coefs = algo.decompose(signal2d) # Perform wavelet decomposition.
res = algo.synthesis(coefs) # Rebuild signal from coefficients.
Display(coefs.compose()) # Show coefficients as Russian dolls.
```

Example 5.58. Discrete wavelet transformation of a bidimensional signal.

Figure 5.7. Signal decomposed: Russian dolls view.

At the top, right hand corner, there are horizontal details.

At the bottom, right hand corner, there are diagonal details.

At the bottom, left hand corner there are vertical details.

The previous figure shows only two scales of decomposition. The signal is halved at each scale.

At the top, left hand corner we found the approximation.

The following example shows how to obtain and change the coefficients:

```
from herschel.ia.numeric.toolbox.wavelet import DiscreteWavelet
algo = DiscreteWavelet("db5") # Select db5 as wavelet
```

```
# Load the standard test image included with the package
from herschel.ia.numeric.toolbox.wavelet.wdemo import WReadImage
signal2d = WReadImage.loadImage("Lena.png")
coefs = algo.decompose(signal2d) # Decompose two-dimensional signal
level = 2 # Selects level 2
horizontal = coefs.getHorizontalForLevel(level) # Get horizontal detail for level 2
horizontal.multiply(4.0) # Multiply horizontal detail by 0.25
Display(horizontal.data)
vertical = coefs.getVerticalForLevel(level) # Get vertical detail for level 2
vertical.add(-0.25) # Subtract 0.25 from vertical detail
Display(vertical.data)
diagonal = coefs.getDiagonalForLevel(level) # Get diagonal detail for level 2
diagonal.multiply(0.25) # Multiply diagonal detail by 0.25
Display(diagonal.data)
# Level has no sense here unlike the Stationary Wavelet transform
approximation = coefs.getApproximation()
Display(approximation.data)
res = algo.synthesis(coefs) # Rebuild the signal
```

Example 5.59. Discrete Wavelet transformation manually handling the coefficients.

5.15.6. Stationary wavelet transform

DWT does not perform a signal translation-invariant transform. SWT resolves that. When the signal is decimated, we can decimate either the even data, or the odd data. DWT decimates even data, while the SWT algorithm will decimate in each side. We add some redundant data and keep an efficient decomposition. Unlike the continuous wavelet transform, only a dyadic decomposition is performed.

Dealing with a one-dimensional image:

```
from herschel.ia.numeric.toolbox.wavelet import StationaryWavelet
algo = StationaryWavelet("db3") # Select stationary wavelet algorithm and db3 as
 wavelet
# Create dummy signal
from herschel.ia.numeric.toolbox.wavelet.wdemo import WSigGenerator
sigGen = WSigGenerator()
signal1d = sigGen.getPredefinedRealFunction1()
# Perform a wavelet decomposition till the level 3 and use ZERO border management
from herschel.ia.numeric.toolbox.wavelet.wutil import WBorder
coefs = algo.decompose(signal1d, WBorder.ZERO, 3)
Display(coefs.data) # Display details at each level
PlotXY(coefs.getApproximation().data) # Get and plot approximation
data = coefs.getDetailForLevel(2) # Get detail for level 2
data.add(0.25) # Add value 0.25. Other functions: multiply, divide, subtract and
 reset
res = algo.synthesis(coefs) # Rebuild signal taking into account user changes
```

Example 5.60. Stationary wavelet transformation of a one-dimensional signal.

Dealing with a two-dimensional image:

```
from herschel.ia.numeric.toolbox.wavelet import StationaryWavelet
algo = StationaryWavelet("db5") # Select stationary wavelet algorithm and db5
 wavelet
# Load the standard test image included with the package
from herschel.ia.numeric.toolbox.wavelet.wdemo import WReadImage
signal2d = WReadImage.loadImage("Lena.png")
coefs = algo.decompose(signal2d) # Decompose two-dimensional signal
level = 2 # Selects level 2
horizontal = coefs.getHorizontalForLevel(level) # Get horizontal detail for level 2
horizontal.multiply(4.0) # Multiply horizontal detail by 0.25
Display(horizontal.data)
vertical = coefs.getVerticalForLevel(level) # Get vertical detail for level 2
vertical.add(-0.25) # Subtract 0.25 from vertical detail
Display(vertical.data)
diagonal = coefs.getDiagonalForLevel(level) # Get diagonal detail for level 2
diagonal.multiply(0.25) # Multiply diagonal detail by 0.25
Display(diagonal.data)
# Get approximation for level 2
```

```
approximation = coefs.getApproximationForLevel(level)
Display(approximation.data)
res = algo.synthesis(coefs) # Rebuild the signal
```

Example 5.61. Stationary wavelet transformation of a bidimensional signal.

5.15.7. Tools

Once your signal is decomposed, you have three tools to threshold your coefficients or to evaluate the noise of your data. These tools are described in the following subsections.

Gaussian noise estimator

This tool evaluates the noise contained in the coefficients. It computes : `median(|dc1(i)|)/0.675`, dc1 for detail coefficient at first level

Thresholding tool

This tool cuts (sets to zero) the coefficients below the threshold.

Example:

```
# Load the standard test image included with the package
from herschel.ia.numeric.toolbox.wavelet.wdemo import WReadImage
signal2d = WReadImage.loadImage("Lena.png")
from herschel.ia.numeric.toolbox.wavelet import StationaryWavelet
algo = StationaryWavelet("db5") # Select stationary wavelet algorithm and db5
 wavelet
from herschel.ia.numeric.toolbox.wavelet.wutil import WGaussianNoiseVisitor
from herschel.ia.numeric.toolbox.wavelet.wutil import WThresholdingVisitor
coefs = algo.decompose(signal2d) # Decompose two-dimensional signal
estimator = WGaussianNoiseVisitor() # Create the gaussian noise visitor
thresholding = WThresholdingVisitor() # Create the thresholding visitor
coefs.accept(estimator) # Apply the visitor to the coefficients
print "noise=", estimator.noiseEstimated # Now the visitor contains the estimated
 noise
# Initialise the 'thresholding' visitor with the gaussian noise level computed
 previously
thresholding.threshold = estimator.noiseEstimated
# Apply the visitor. All coefficients below the threshold will be considered as
 noise and removed
coefs.accept(thresholding)
res = algo.synthesis(coefs) # Rebuild the signal
```

Example 5.62. Use of the wavelet thresholding tool.

Universal threshold

This threshold has been defined by Donoho and Johstone. In the figure below, *n* is the signal length.

Universal threshold (Donoho & Johnstone)

$$threshold = \sigma\sqrt{2\log(n)}$$

Figure 5.8. Formula of universal threshold.

```
# Load the standard test image included with the package
from herschel.ia.numeric.toolbox.wavelet.wdemo import WReadImage
signal2d = WReadImage.loadImage("Lena.png")
```

```
from herschel.ia.numeric.toolbox.wavelet import StationaryWavelet
algo = StationaryWavelet("db5") # Select stationary wavelet algorithm and db5
 wavelet
from herschel.ia.numeric.toolbox.wavelet.wutil import WUniversalThresholdVisitor
from herschel.ia.numeric.toolbox.wavelet.wutil import WGaussianNoiseVisitor
coefs = algo.decompose(signal2d) # Decompose two-dimensional signal
estimator = WGaussianNoiseVisitor() # Create the gaussian noise visitor
thresholding = WUniversalThresholdVisitor() # Create the thresholding visitor
coefs.accept(estimator) # Apply the visitor to the coefficients
print "noise=", estimator.noiseEstimated # Now the visitor contains the estimated
 noise
# Initialise the 'thresholding' visitor with the gaussian noise level computed
 previously
thresholding.sigma = estimator.noiseEstimated
# Apply the visitor. All coefficients below the threshold will be considered as
 noise and removed
coefs.accept(thresholding)
res = algo.synthesis(coefs) # Rebuild the signal
```

Example 5.63. Applying a threshold for wavelets using the visitor mechanism.

5.15.8. Wavelet toolbox overview

The following tables give an overview of the features supported by the wavelet package.

Table 5.4. Algorithms

CWT	Continuous Wavelet Transform
DWT	Discrete Wavelet Transform
SWT	Stationary Wavelet Transform

Table 5.5. Tools

Algorithm//Tool	Gaussian noise estimator	Thresholding	Universal thresholding
CWT	no	no	no
DWT	yes	yes	yes
SWT	yes	yes	yes

Table 5.6. Signal dimensions

Algorithm/signal	One-dimensional	Two-dimensional
CWT	yes	-
DWT	yes	yes
SWT	yes	yes

Table 5.7. Border management

Algorithm/padding	Zero	Symmetric	Constant	Periodic
CWT	no	yes	no	no
DWT	yes	yes	yes	no
SWT	yes	yes	yes	no

Table 5.8. Supported signal types

Algorithm/Signal type	Double1d	Double2d	Complex1d	Complex2d
CWT	yes	-	yes	-

Algorithm/Signal type	Double1d	Double2d	Complex1d	Complex2d
DWT	yes	yes	-	-
SWT	yes	yes	-	-

Table 5.9. Available wavelets

Algorithm/Wavelet	Daubechie	Symlet	Coiflet	MexicanHat	Morlet
CWT	1	-	-	yes	no
DWT	1-10	1-10	1-5	-	-
SWT	1-10	1-10	1-5	-	-

For more information on functions for wavelet transforms, follow the links to go to the corresponding entries in the *User's Reference Manual*.

- DiscreteWavelet in *HCSS User's Reference Manual*: Performs a discrete wavelet transform.
- Continuous Wavelet in *HCSS User's Reference Manual*: Performs a continuous wavelet transform.
- CWavelet in *HCSS User's Reference Manual*: A continuous wavelet (along with an example of the creation of custom continuous wavelets using only Jython).

Chapter 6. Running tasks

Tasks are a standardised format for data reduction routines. A task provides consistent conventions for input and output parameters, processing history and help information. Tasks can be executed from the command line or via a standard graphical interface in HIPE.

With tasks you can create modular and reusable code for data reduction and analysis, easier to use and to distribute.

This chapter shows how to execute tasks from the command line. For information on executing tasks via the HIPE graphical interface, see the *HIPE Owner's Guide* in *HIPE Owner's Guide*.

For information on how to develop tasks in Jython or Java, see the HIPE Community website.

6.1. Running a task

This section describes how to run a task from the *Console* view of HIPE.

Imagine you want to run the `clear` task, which deletes one or more variables in the HIPE session.

Printing task parameters. You can print a list of parameters, with all their properties such as type and default value, by printing the name of the task:

```
print clear
```

Printing task help. You can print a short help text about a task as follows:

```
print clear.__doc__
```

Example 6.1. Printing the documentation of a task.

Note that there are two underscore characters before and after `doc`.

Executing the task. Execute the task by passing parameter values within brackets:

```
clear(variable="myVar")
```

Example 6.2. Executing the clear task with one parameter.

The `clear` task has no output parameter. For tasks with an output parameter, assign its value to a variable when executing the task. In the following example, the task *`myTask`* is executed and the value of the output parameter is passed to the *`myResult`* variable:

```
myResult = myTask(parameter="value")
```

Example 6.3. Retrieving the output value from a task.

See Section 6.2 for more information on task parameters.

Tip

If you have doubts about the syntax of a task command, try executing the task via its dialogue window. You can open the dialogue window by double clicking on the task name in the *Tasks* view. When you execute the task, the corresponding command appears in the *Console* view.

Checking execution result. To make sure that the task executed successfully, you can look at the `statusMessage`:

```
print clear.statusMessage
```

Example 6.4. Printing the status message of a task.

6.2. Task parameters

Tasks can have input, output and input/output parameters. Input/output parameters are passed as input and their values are modified by the task.

You can identify parameters by their position or by their name. Look at the following code:

```
# Positional arguments
result = myTask(param1, param2)
# Named arguments
result = myTask(first=param1, second=param2)
```

Example 6.5. Naming the parameters to omit optional ones or pass them in any order.

Here the task `myTask` has two parameters called `first` and `second`.

You can mix positional and named modes, but only if all positional arguments come first. For example:

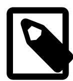
```
result = myTask()(param1, second=param2)
```

Example 6.6. Mixing named and positional parameters.

The following line would cause an error instead:

```
result = myTask()(first=param1, param2)
```

Example 6.7. Wrong mix of mixed and named parameters.



Note

Identifying parameters by their name, rather than their position, is strongly recommended. This will make your scripts much more maintainable.

Once a task is executed, parameters are reset to their default values.

6.2.1. Output parameters

When a task has multiple output parameters, you can call it in the following ways:

- ```
  myVar = myTask(...)
  ```

  In this case (in HIPE 12 and later), *`myVar`* becomes a list and all outputs are returned as elements of the list.

  Jython/Python lists are very powerful and allow advanced functionality like:

  - List slicing: Reference and retrieve specific ranges of values using the list indices. Example of assigning the output parameters from the third to the fifth position after running task `myTask()` (remember that indices start at zero):

    ```
    outPar2, outPar3, outPar4 = myTask(...)[2:4]
    ```

    Example 6.8. Assigning output values to variables using list slicing.
    ```

- List filtering: Use an anonymous function (a lambda function, see below) to include only values that match a condition. Example of excluding from the list myVar all the values lesser than zero using list comprehension syntax:

```
myVar = [x for x in myVar if x >= 0 ]
```

Example 6.9. Assigning output values to variables filtering using list comprehension syntax.

- Lambda functions: They are anonymous constructions that take any input (including other functions) and apply a function to them. Lambdas are often used with iterators or the `map()` function. The previous example is actually a lambda function in shorthand notation. Example of getting the squares of each item of the list myVar using a lambda definition:

```
myVar = map (lambda x: x**2, myVar)
```

Example 6.10. Assigning output values to variables filtering with lambda expression.

- List comprehension: These are expressions that make applying lambdas to lists easier. Example of getting the squares of each item of the list myVar using list comprehension syntax:

```
myVar = [ x**2 for x in myVar ]
```

Example 6.11. Assigning output values to variables filtering using list comprehension syntax (II).

Another option to deal with named output parameters is using the utility method `outToIndex()`, that returns the index of the parameters which names are passed as a list. For example (with the help, again, of the list comprehension syntax):

```
myVar = myTask(...)
# print myVar
outPar2, outPar1 = [myVar[i] for i in myTask.outToIndex(["parname2","parname1"])]
del(myVar)
```

Example 6.12. Assigning output values to variables using the utility method outToIndex.

The parameter names (`parname1` and `parname2` in the example) have to be clearly described in the task documentation.

- ```
  myVar1, myVar2, ..., myVarN = myTask(...)
  ```

  In this case, each output is returned to a different variable. If the number of variables is different from the number of outputs, HIPE gives an error.

An introductory guide of working with lists in Jython can be found beginning in Section 1.10 of this manual. For the more advanced topics outlined above you should refer to the Jython book chapter on Data Types or the Python tutorial chapter on Data Structures.

Tip

To view more information on the parameters of a task, click on the task name in the *Tasks* view. A table appears in the *Outline* listing the task parameters and their properties. In particular, you can see at a glance which parameters are input (IN), output (OUT) and input/output (INOUT).
  ```

Chapter 7. Storing and accessing data products

This chapter describes how to store, retrieve and search for data products, either on your local system or on remote locations such as the Herschel Science Archive.

Before reading this chapter you should be familiar with the basic concepts explained in the *Data Analysis Guide*: Section 1.3 in *Data Analysis Guide*.

7.1. Pools and storages

A *product storage* is the front-end interface that allows you to communicate with products stored in pools.

By *registering* a pool to your storage, you can access the products in that pool.

A product storage provides mechanisms to load, save and query products in the registered pools. When doing so you receive a reference to a product (returned by the `load()` and `save()` commands) or a set of product references (when querying). This functionality of a product reference is provided by the `ProductRef` class; it allows to fetch information of the product, such as metadata, without loading the product in memory.

A *urn* or *URN* is part of the product reference and stands for *Uniform Resource Name*. It represents the address of the product. A typical urn looks like this:

```
urn:myPool:herschel.ia.dataset.Product:5
```

A urn consists of four fields separated by colons:

1. A fixed string `urn`.
2. The name of the pool.
3. The name of the Java class representing the product.
4. A count of the number of products that have been created of that class. Do *not* think of it as a version — it is not.

7.1.1. Creating a storage and registering pools

Since a storage without registered pools is useless, you usually create a storage and register a pool at the same time. You can pass a string with the name of the pool to be registered, or a variable representing the pool itself:

```
storageName = ProductStorage("poolName")
storageName = ProductStorage(pool)
```

You can also register many pools at once, by passing an array of names or pools:

```
storageName = ProductStorage(["poolName1", "poolName2", ...])
storageName = ProductStorage([pool1, pool2, ...])
```

Example 7.1. Registering many pools at once during storage definition.

If a name does not correspond to an existing pool, a new pool is created.

Use the `register` method to register pools after you have created the storage:

```
storageName.register(pool)
```

Example 7.2. Registering pools after storage creation.

To get a list of the currently registered pools:

```
print PoolManager.getPoolMap()
```

Example 7.3. Printing a map of all registered pools.

7.1.2. Saving and loading products

Use the `save` and `load` methods to save and load products:

```
HIPE> myReference = myStorage.save(myProduct)
HIPE> print myReference.urn
urn:simple.default:herschel.ia.dataset.Product:0
```

```
HIPE> myReference
 = myStorage.load("urn:simple.default:herschel.ia.dataset.Product:0")
```

In both cases you obtain a `ProductRef` object. A reference provides access to parts of the product as well as access to the product itself:

```
HIPE> print myReference.urn
urn:simple.default:herschel.ia.dataset.Product:0

HIPE> print myReference.type
herschel.ia.dataset.Product

HIPE> myMeta = myReference.meta  # Getting metadata

HIPE> myProduct = myReference.product  # Getting the product
```

Note that, if you have multiple pools registered to a storage, only the *first* registered pool is write-enabled. In other words, all `save` operations write products to the first pool only. If you want to write products to another pool, you have to register it as first pool to another storage.

7.1.3. Deleting products

To remove a product from a storage, use the `remove` method:

```
 myStorage.remove(urn)
```

Example 7.4. Removing products from a storage.

note that you need to know the urn of a product to delete it.

Remember that you have write permission only on the first pool registered to a storage. If you try to remove a product from another pool, which is read-only, you get an error. You must first register the pool as *first pool* to another storage.

In the case of a local pool, you can delete a product by deleting the corresponding FITS file from the local pool directory. Then you need to rebuild the pool index (see Section 7.2.2).

7.1.4. Tagging products

Tags are keywords or phrases you can associate to a product, to better describe and remember its contents. For example, you could assign to a product the tag `"to be completed"` to remember that you have not finished processing it. When defining tags, you are free to use the keywords and phrases that work best for you.

To save a product with a given tag:

```
myStorage.saveAs(myProduct, "myTag")
```

Example 7.5. Tagging a product and adding it to a storage.

You can then use the tag to load a reference to the product:

```
myProductRef = myStorage.load("myTag")
```

Example 7.6. Loading a tagged product as a product reference.

To load the product itself, instead of a reference, add a `.product` to the previous command:

```
myProductRef = myStorage.load("myTag").product
```

Example 7.7. Loading a tagged product.

To assign a tag to an existing product already in the storage:

```
myStorage.setTag("myTag", productUrn)
```

Example 7.8. Tagging an existing product.

You can assign multiple tags to the same product by invoking the `setTag` method multiple times. However, a given tag can only be assigned to a single product. In the following example, assigning `myTag` to the product identified by `urn2` removes the same tag from the product identified by `urn1`:

```
myStorage.setTag("myTag", urn1)
myStorage.setTag("myTag", urn2)
```

Example 7.9. Tagging a product with several tags.

To remove a tag:

```
myStorage.removeTag("myTag")
```

Example 7.10. Removing tags from a product.

To check if a given tag exists:

```
print myStorage.tagExists("myTag")
# Returns 1 if the tag exists, 0 otherwise
```

Example 7.11. Checking tag existence (in a storage) before tagging.

7.2. Local pools

The *local pool*, also known as *local store* for historical reasons, is the most commonly used type of pool.

7.2.1. The local pool directory

By default, data is stored in a directory with the user-supplied store name in the following directory:

```
home/.hcss/lstore/
```

To change the local pool directory, follow these steps:

1. In HIPE, choose *Edit → Preferences*. The *Preferences* dialogue window opens.
2. Click *Local Store* under *Data Access* in the left-hand list. The *Local Store* panel opens in the right-hand area.

3. Change the directory in the *Local Store directory* field.

4. Click *Apply*.

Tip

The local store directory can also be a link to another directory. This is useful if you want to store your products in a different hard disk with more space.

You can rename a local pool by renaming the corresponding directory, but *only if the pool was created with HCSS 4.0 or newer*.

7.2.2. Repairing a local pool

A local pool index can become inconsistent, for example if you add or delete files manually in the pool directory. In this case, you must rebuild the pool index as follows:

```
myPool.rebuildIndex()
```

Example 7.12. Rebuilding the index of a pool.

Do not access the pool during the operation, which can take a while depending on pool size.

7.2.3. Importing a directory of FITS files into a local pool

To place all FITS files from a directory into a local pool, use the following commands:

```
myPool.ingest(java.io.File("path_to_directory"), 0)
```

If the second parameter is set to zero, the FITS files are copied into the local pool directory. If the parameter is set to one, only references to the original files are created in the local pool directory.

7.2.4. Troubleshooting

You may experience problems with local pools in the following cases:

- When saving data to NFS-mounted disks (you may get an `IllegalMonitorStateException`).
- When saving data to a FAT32 filesystem from a Mac (you may get an `OverlappingFileLockException`).
- In any case involving large observations or the use of the `bg` command (you may get an `org.apache.lucene.store.LockObtainFailedException`).

Both problems are solved by setting the following property:

```
hcss.ia.pal.pool.lstore.lock = simple
```

See the HIPE Owner's Guide in *HIPE Owner's Guide* for information on how to set properties. See the Known Issues page for information on other issues affecting HIPE.

If this solution does not work, please try removing the lock files (`write.lock`) present in the local store directory. This bash one-liner script can speed up the process on UNIX-based systems (including OS X):

```
find <pool> -name 'write.lock' -exec rm {} \;
```

7.3. Querying

To find out the contents of a storage, you execute a *query* on it. This sections introduces the syntax for command line queries.

Note

The syntax for querying the Herschel Science Archive differs slightly from what is explained in this section. See Section 7.7.1 for more information.

The following example looks for products with *ThatsMe* as creator:

```
query1 = Query("creator == 'ThatsMe'")
res = myStorage.select(query1)
print res
```

Example 7.13. Using keyword queries to retrieve products from a storage.

You can query any metadata and combine more keywords (note that == and = are equivalent):

```
query1 = Query("creator == 'ThatsMe' and instrument = 'SPIRE'")
res = myStorage.select(query1)
print res
```

Example 7.14. Querying a storage with several keywords.

Now `res` contains a list of references to the products that satisfy the query. Printing `res` will give a list of URN values:

```
HIPE> print res
[urn:default:herschel.ia.dataset.Product:0,
urn:default:herschel.ia.dataset.Product:1,
urn:test:herschel.ia.dataset.Product:0]
```

If you want to execute an unconditional query to find all products in a storage, you can use the following:

```
query2 = Query(1)
res2 = myStorage.select(query2)
print res2
```

Example 7.15. Retrieving references to all products in a storage.

To find all the products of a given class (here `ObservationContext`):

```
res3 = myStorage.select(Query(ObservationContext))
```

Example 7.16. Finding all products matching a class.

Note that the above example puts query and selection on the same line.

You can query metadata and limit the query to a given class at the same time. This is usually a good idea, because it speeds up the query:

```
res3 = myStorage.select(Query(ObservationContext, "creator == 'ThatsMe'"))
```

Example 7.17. Querying by class and keywords at the same time.

For information on how to query pools via the *Product Browser* perspective in HIPE, see the *Data Analysis Guide*: Section 1.7 in *Data Analysis Guide*.

7.3.1. Inspecting query results

The results of a query come as a list of product references. You can inspect the results as follows, assuming they are held in a variable called `res`:

```
# Printing the number of results.
print res.size()
# Checking whether the list of results is empty.
print res.isEmpty()  # Returns True or False.
# Assigning the first result to variable myProduct.
myProduct = res[0].product
```

Example 7.18. Inspecting the results of a query.

7.4. Product versioning

To save a set of versions of a particular product:

```
myStorage.save(myProduct) # Version 0 of myProduct saved
# After modifying myProduct...
myStorage.save(myProduct) # Version 1 of myProduct saved
```

Example 7.19. Versioning products within a storage.

To get the latest version of a product, or the list of versions for that product, you need to have available at least one, arbitrary, version. With this, you can recover the latest version of the product, and the list of all versions of the product in the storage:

```
latest = myStorage.getHead(productRefOfAnyVersion)
versions = myStorage.getVersions(productRefOfAnyVersion)
```

Example 7.20. Retrieving the latest version of a product.

You can get information on the current version of each product, as well as tag information, as follows:

```
print myStorage.versioningInfo
```

Example 7.21. Printing version and tag information for each product.

Note that versioning is a property of a pool, not of a storage.

7.4.1. Querying product versions

Querying by default searches for just the latest version of a product:

```
query = Query(Product, "p", "1")
storage.select(query) # Just the latest version
```

Example 7.22. Using a default query returns the latest version.

If you want to get all versions of products that match a query, add a fourth argument set to 1:

```
query = Query(Product, "p", "1", 1)
storage.select(query) # All versions of matching products
```

Example 7.23. Returning all versions of a product in a query.

(Note that with this extended query, the special products containing versioning information, VersionTrackProduct and TagsProduct, are also returned if they match the query.)

7.5. Advanced querying

There are three types of queries:

- Attribute query is a (fast) query on meta data that all products contain: *creator, creationDate, startDate, endDate, instrument, modelName*. This is akin to querying a standard set of FITS header keywords.

- Meta data query is a (semi fast) query on meta data that can be different from product to product. This is akin to doing a query on any FITS keywords (if present).

- Full query is a data mining query that allows querying on *all* data elements in products.

All query types have the same syntax, but a different purpose as described above. Setting up a query is as follows:

```
# Simple query
query = Query(expression)
# More advanced queries
query = AttribQuery(product-class, variable, expression)
query = MetaQuery(product-class, variable, expression)
query = FullQuery(product-class, variable, expression)
```

Example 7.24. Creating simple, attribute, metadata and full (or data mining) queries.

The parameters are explained below:

- `product-class`: restricts the query to a family of products. All product classes have `herschel.ia.dataset.Product` as the base class. You can restrict the query to a sub-family of products. For example, if all HIFI Calibration Product classes stem from `HifiCalProduct`, you can limit your search by specifying that class.

- `variable`: a string denoting the variable name of the product that will be used in the expression.

- `expression`: a string holding the query expression, which is limited to the query type.

- **Query Example**

  ```
  query = Query("instrument == HIFI and band == 1a")
  # A simple query should be enough in most cases.
  ```

  Example 7.25. Creating a simple query.

- **AttribQuery Example**

  ```
  query = AttribQuery(Product, 'product', \
          'product.creator=="Me" and product.instrument="HIFI"')
  ```

  Example 7.26. Creating an attribute query.

- **MetaQuery Example**

  This type of query allows to inspect any part of the meta data of the product specified in the first argument.

  ```
  query = MetaQuery(HifiCalProduct, 'h', 'h.meta["key1"].value < 123 and \
       h.meta["key2"].value == "Hello world"')
  ```

  Example 7.27. Creating a metadata query.

  Note

  To obtain a numerical value (rather than, for instance, the string equivalent) it is necessary to stipulate that the meta key "value" is required, hence the need for the stipulation of query on 'h.meta["key1"].value' rather than 'h.meta["key1"]'

- **FullQuery Example**

  A data mining query exploits the full interface of the product in question. Numeric functions defined in the basic toolbox are allowed:

  ```
  query = FullQuery(Product, 'p', 'p.creator=="Me" and (ANY(p.spectrum.data < 2) \
  ```

```
        or ALL(p["myTable"]["myColumn"].data > 5)')
```

Example 7.28. Creating a full query.

The ANY function used above is one of the standard numerical function provided in HIPE, and checks whether the expression provided in its argument is true for any of the elements in that argument.

7.5.1. Querying for parts of a string

Use the % character when you want to query for parts of a string. For example, the following command looks for all products of type ObservationContext in which the aorLabel metadata value begins by ObsCal:

```
query = MetaQuery(ObservationContext,'p','p.meta["aorLabel"].value == "ObsCal%"')
```

Example 7.29. Creating a metadata query with SQL-like wildcards for values.

To look for ObsCal anywhere within the string, rather than just at the beginning, put a % character at either end: %ObsCal%.

7.5.2. Querying for metadata in products

One thing you need to watch out when performing a meta or full query, is when you try to query for a metadata that does not exist in one or more products that you are applying the query to.

For example, consider the following MetaQuery:

```
query = MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
resultset = storage.select(query)
```

Example 7.30. Querying by metadata requires the keyword to exist in all filtered products.

The query first starts creating a shortlist of all products in the storage matching type Product. It then runs the query string on each product in that shortlist. If any of those products don't contain the information referenced in the query string, an error is raised.

There are two ways to avoid this:

- Be as specific as you can when it comes to specifying the product type in a query. If you know the product type you want to query is of type CalHrsQDCFull, then specify that. Running queries using the most general product type of Product is not recommended, unless the products you have saved are of this type only.
- Run a two-stage query, using the containsKey() operator to check whether a component exists first. For example, first get a sub-set of products that contain the metadata 'temperature':

```
queryOne = MetaQuery(Product, 'p', 'p.meta.containsKey("temperature")')
resultsetOne = storage.select(queryOne)
```

Example 7.31. First step filtering the products containing the keyword.

Then run the original query on this subset:

```
queryTwo = MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
resultsetTwo = storage.select(queryTwo, resultSetOne)
```

Example 7.32. Second step filtering by keyword value.

7.6. Tips and pitfalls

7.6.1. Changes to a product in a pool disappear

While product *contexts* saved in a pool are cached in memory, *leaf* products are not cached for performance reasons. This can lead to problems as illustrated by the following example:

```
ref = ProductRef (Product (creator="me", instrument="SPIRE"))
print ref.product.instrument
# SPIRE
ref = myStorage.save (ref.product)
ref.product.instrument = "PACS"
print ref.product.instrument
# SPIRE
```

Example 7.33. Changes to products should be done in memory before saving them to a pool.

The change does not have any effect because the product has been saved to a pool and is no longer cached in memory.

The correct behaviour is obtained by *dereferencing* the product, that is, by creating a variable that represents the product itself, not a reference to it:

```
p = ref.product  # Load from pool
p.instrument = "PACS" # Modify
print p.instrument
# PACS
ref = myStorage.save (p)  # Save back to pool
print ref.product.instrument
# PACS
```

Example 7.34. Loading the product back from the pool, changing and saving to persist the change.

After saving the product back to the pool, even the reference correctly shows the modified information.

To check whether the product corresponding to a reference is loaded into memory, use the `isLoaded` method:

```
print ref.isLoaded()
# 0  # Product not loaded
# 1  # Product loaded
```

Example 7.35. Checking if a reference is loaded in memory.

7.6.2. Minimising memory usage

You can look at a product's metadata without loading the product into memory:

```
ref = myStorage.save(myProduct)
ref = myStorage.load(ref.urn)
print ref.meta
```

Example 7.36. Loading specific parts of a product.

The following example saves a context to a pool without having more than one product in memory at any time. This is vital to avoid running out of memory with large contexts:

```
myContext = ListContext(description = "A very big context")
for i in range(10):  # Saving ten products
    p = Product(description = "Dummy product no. " + str(i))
    myContext.refs.add(myStorage.save(p))
```

Example 7.37. Saving a context to a pool without the leaf products in memory.

7.6.3. Testing if two products are equal

If you have two products in memory, you can use the `equals` method:

```
print product1.equals(product2)
```

Example 7.38. Comparing products in memory.

This method returns 1 if the two products are equal, 0 otherwise.

If you just have a reference to a product, you can compare *hash values* as shown in this example:

```
from herschel.ia.pal.util import HashCoder
hash1 = HashCoder().getHash(myProduct)
hash2 = myProductRef.getHash()
print hash1 == hash2
```

Example 7.39. Comparing product references with hash codes.

The result is 1 if *`myProduct`* and the product referenced by *`myProductRef`* are equal, 0 otherwise.

If you have a product urn, represented by a variable of type `Urn`, you can retrieve the product hash code from the pool where the product is saved, like this:

```
hash3 = myPool.loadDescriptors(myUrn).get("hash")
```

Example 7.40. Comparing product URNs using hash codes.

You can then compare the hash values with the == operator as before.

7.6.4. Copying a product or context to a different storage

Use the `save` method to copy a product or an entire context from a storage to another:

```
myStorage.save(myContext)
```

Example 7.41. Saving a context.

If a product within the data tree already exists in the destination product storage, it is not copied. A product can exist in the destination storage if it belongs to a pool shared between the origin and destination storage.

Note that a storage may contain multiple versions of a context. A new version of a context is created when a context is saved, modified, then saved again. The older versions of the context are also copied. However, if that context has any descendants that are contexts, the local versions of those descendant contexts are not copied.

7.6.5. Tags may point to wrong product after renaming a pool

A tag may end up pointing to the wrong product, or to a non-existing product, in the following scenario:

1. A storage has two pools, `a` and `b`, of which `a` is writable. Remember that only one pool in a storage is writable.
2. You tag a product in pool `b`. This is written to pool `a`, since `b` is read-only.
3. You delete pool `b` and rename pool `a` to `b`. The tag you created now points to a wrong or non-existing product.

Note that this situation can only arise when managing pools and storages from the command line. The graphical tools provided with HIPE work with storages connected to a single pool.

7.6.6. IndexError or IllegalArgumentException when querying

When running a query on a storage, you may get an `IndexError` or `IllegalArgumentException: <query> could not be evaluated correctly` message. This can be due to one of the following reasons:

1. Your query string (the third argument of a query, for instance `'p.creator==..'`) is simply not consistent with the Jython syntax and could not be correctly interpreted. Check your query string by evaluating it on the Jython command line. If your query uses a *handle* to a product (for example the `p` in a query `p.meta[..]` is a handle), then create a dummy product of the type you want to query on the command line to test the query against.

2. It could be possible that the query references some data that does not exist in *any* of the products that match the product type you have passed in that query. If you see in the details of the error message something along the lines of '<something> does not exist', then this may be the case.

   For example, consider the following MetaQuery:

   ```
   query =MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
   resultset=storage.select(query)
   ```

   Example 7.42. Filtering directly on metadata values.

   The query first starts creating a shortlist of all products in the storage matching type `Product`. It then runs the query string on each product in that shortlist. If any of those products don't contain the information referenced in the query string, an error is raised.

   There are two ways to avoid this:

   a. Be as specific as you can when it comes to specifying the product type in a query. If you know the product type you want to query is of type `CalHrsQDCFull`, then specify that. Running queries using the most general product type of `Product` is not recommended, unless the products you have saved are of this type only.

   b. Run a two-stage query, using the containsKey() operator to check whether a component exists first, for example:

      i. Get a sub-set of products that contain the metadata 'temperature':

      ```
      queryOne = MetaQuery(Product, 'p', 'p.meta.containsKey("temperature")')
      resultsetOne = storage.select(queryOne)
      ```

      Example 7.43. Filtering products in the archive that contain a specific metadata.

      ii. Run the original query on this subset:

      ```
      queryTwo = MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
      resultsetTwo = storage.select(queryTwo, resultSetOne)
      ```

      Example 7.44. Filtering by value a set of products that contain the metadata.

7.6.7. A query takes a long time to execute

One of the possible reasons is that you are executing a `FullQuery`, and full queries by their very nature are the most intense of queries and are therefore the slowest.

`FullQuery` executions should be run as the last stage of a multi-stage query operation. Below is an example of how to search a storage for products of type `MyProduct` that are created by a developer called 'timo', but contain a specific value in the product data itself.

1. Find all products of type `MyProduct` with creator 'timo':

```
attquery = AttQuery(MyProduct, 'p', p.creator=='timo')
resultset = storage.select(attquery)
```

2. Find all products in selection generated from previous queries, # that has a value 10 in the column 'mycolumn' in dataset 'mydataset':

```
fullquery = FullQuery(Product, 'p', 'p["mydataset"]["mycolumn"].data[5]==10')
storage.select(fullquery, resultset)
```

Example 7.45. Using a full query to filter by data values.

There can be intermediate queries between the two steps involving `AttribQuery` or `MetaQuery`, but `FullQuery` should be left to last.

7.7. Pools for remote data

In addition to local pools, you may find these other pool types useful:

- With the *HSA pool* you can connect to the Herschel Science Archive via the command line.
- With the *HTTP client pool* you can access products on a remote server.
- With the *cached pool* you can cache everything retrieved from a pool. It is useful if the pool you are working with is a remote online pool, and you want to work offline.

All the content of the previous sections (except that specific to local pools) also applies to these pools.

7.7.1. The HSA pool

With the HSA pool you can connect to the Herschel Science Archive, query its contents and retrieve products and observations, all from the command line. The HSA pool is used behind the scenes by tasks such as `getObservation` (see the *Data Analysis Guide*: Data Analysis Guide).

You can create an HSA pool as follows:

```
hsa = HsaReadPool()
```

Example 7.46. Creating a read-only pool connected to the archive.

You can use the HSA pool together with a cached pool, so that retrieved data are cached (see Section 7.7.3):

```
hsa = CachedPool(HsaReadPool())
```

Example 7.47. Adding cache behaviour to the HSA read pool.

You can now use this pool to query the Herschel Science Archive using the query syntax described in Section 7.3 and Section 7.5. You need to be logged into the HSA to retrieve products. See the *Data Analysis Guide* for details: Section 1.4.1 in *Data Analysis Guide*.

Querying the HSA pool differs slightly from what is explained in Section 7.3:

- You can use the == operator but not the = one: `"creator == 'ThatsMe'"` is valid, `"creator = 'ThatsMe'"` is not.
- You can only query a subset of metadata, shown in Section 7.7.4.

Why would I want to use this pool? This pool is useful if you want to write scripts that automatically connect to the HSA and perform queries and other operations for which the syntax of `getO-`

bservation is not enough. You do not need to use this pool directly if you only access the HSA through its graphical interface.

7.7.2. The HTTP pool

With the HTTP pool you can query and retrieve data hosted on a remote server pool that can be accessed via a URL.

To access a remote HTTP pool, create an HTTP client pool as follows:

```
pool = HttpClientPool("http://url.of.remote.pool", "remotePoolName")
```

Example 7.48. Creating an HTTP client pool.

You must specify the URL and name of the remote pool. Ask the remote pool administrator for them.

You can use the HTTP pool together with a cached pool, so that retrieved data are cached (see Section 7.7.3):

```
pool = CachedPool(HttpClientPool("http://url.of.remote.pool", "remotePoolName"))
```

Example 7.49. Creating a cached pool from a URL.

The previous examples only describe how to set up a *client* pool to access remote data. Setting up a *server* pool to host the data is beyond the scope of this manual, but you can find instructions on the public Herschel TWiki: http://herschel.esac.esa.int/twiki/bin/view/Public/HttpPool.

If you encounter problems while setting up an HTTP server pool, please ask the Herschel Helpdesk (please open this link in a new tab or window) for assistance.

Why would I want to use this pool? You should use this pool to access remote data products made available on a server. If you only access products on the Herschel Science Archive and on your local disk, you do not need this pool.

7.7.3. The cached pool

The cached pool allows you to cache everything that is retrieved from any remote pool. You can cache any remote pool as follows:

```
pool = CachedPool(remotePool)
```

Example 7.50. Creating a cached pool from an already created remote pool.

Using a cached pool allows you to work offline once you have retrieved the data you need.

The cached pool set-up consists of a remote pool and a pool to store the locally cached products (which we call the *delegated* pool), held inside a directory with administrative data. We refer to the whole as just "the cached pool".

The delegated pool can be accessed independently as well. However, the cached pool views this as a private storage area, and explicitly assumes that nothing will be added or removed, unless it is through its own interface (by a call to `clearCache()`, for example). Do not modify the delegated pool by accessing it directly if you do not want to risk that the cached pool becomes corrupted and must be cleaned and restarted.

A cache is kept between HIPE sessions, and the cached pool identifies pools by their ID. If you create a new pool in the next HIPE session with the same ID, then it is assumed that this is the same pool as before and the cache will be reused. It is up to you to explicitly clear the cache if this is required (if it is a different pool than the one that the cached data corresponds to). Also, you should be aware of potential name conflicts between pools: if two HTTP client pools are created, connecting to different

hosts, but with the same ID, then if they are both cached in the same HIPE session (or in different HIPE sessions but simultaneously) a name conflict will arise.

For more information see the `CachedPool` entry in the *User's Reference Manual*: Section 1.51 in *HCSS User's Reference Manual*.

Why would I want to use this pool? You should use the pool with the HSA and HTTP pools, if you use them with any regularity, so that you can minimise network transfers and work offline.

7.7.4. Metadata used in the HSA pool

Name	Type	Description
acmsMode	String	ACMS mode
activeStrId	String	identification of the active STR
aorLabel	String	AOR label as entered in HSpot
aot	String	AOT Identifier
aperture	String	Instrument aperture in use
apid	long	Application Programme Identifier
arrayName	String	Name of Detector Array
author	String	Author of the data
averaging	String	Averaging operator
band	String	Band
baselineModel	String	Baseline Model
baselineParams	String	Parameters of Baseline model
bbCount	long	Building Block Count
bbid	long	Building Block Identifier
bbType	long	Building Block Type
bbTypeName	String	Building Block Type Name
biasFreq	double	Bias frequency
biasMode	double	Bias mode
biasVoltage	double	Bias voltage factor
bitPos	long	Bit position of this mask
calFileId	String	Calibration file ID
calFileVersion	int	Calibration file version
calThreshold	double	Specified position accuracy threshold for a plateaux in calibration
camera	String	Name Camera/ detector array
cameraModel	String	Model of the camera (CQM, FM, Sixpack,...)
cd_1_1	double	CD_1_1 element of CD matrix
cd_1_2	double	CD_1_2 element of CD matrix
cd_1_3	double	CD_1_3 element of CD matrix
cd_2_1	double	CD_2_1 element of CD matrix
cd_2_2	double	CD_2_2 element of CD matrix
cd_2_3	double	CD_2_3 element of CD matrix

cd_3_1	double	CD_3_1 element of CD matrix
cd_3_2	double	CD_3_2 element of CD matrix
cd_3_3	double	CD_3_3 element of CD matrix
cdelt1	double	pixel size in axis 1
cdelt2	double	pixel size in axis 2
changelog	String	Logging of changes
chopperPlateau	long	Indicates the chop plateau within sequence
constVelFlag	boolean	Constant velocity flag
conversionFactor	double	conversion factor from chopper deflection (degrees) to angle on sky
creationDate	FineTime	Date of product creation
creator	String	The name of the software that created the product
crota2	double	rotation angle
crpix1	double	CRPIX1 reference pixel of axis 1
crpix2	double	CRPIX2 reference pixel of axis 2
crval1	double	axis 1 coordinate at tangency
crval2	double	axis 2 coordinate at tangency
ctype1	String	type of coordinate axis eg RA---TAN
ctype2	String	type of coordinate axis eg DEC—TAN
cusMode	String	CUS observation mode
dataAnalyst	String	Name of data analyst
dec	double	Actual Declination of pointing
decNominal	double	requested declination of pointing
decObject	double	Declination of target object
deltaPix	double	Correction of output angle per pixel unit offset to central pixel
description	String	Full name of product
endDate	FineTime	End date of observation
endWavelength	double	End of wavelength interval
epoch	double	equinox of celestial coordinate system
equinox	double	equinox of celestial coordinate system
error	double	Error on signal
fileName	String	name of exported file
filter	String	Filter name [SHORT/LONG/none]
fineTime	long	Time of signal sampling
formatVersion	String	Version of product format
gyroPropQualIdx	double	Gyro-propagated quality index
instMode	String	Instrument mode

instrument	String	Instrument name
interpMethod	String	Recommended interpolation method to be applied
jiggleId	long	Jiggle Identifier
keyWavelength	double	Key Wavelength
level	String	Product Level
maxWavelength	double	Maximum wavelength
minWavelength	double	Minimum wavelength
missionConfig	String	Mission configuration
modelName	String	Instrument Model Name
naifId	String	SSO NAIF identifier
nodCycleNum	long	Switching/nodding cycle number
numChopCyc	long	Number of chopping cycles
numHifiSaa	long	Number of HIFI reference Solar Aspect Angles
numJigglePos	long	Number of jiggle positions
numNodCyc	long	Number of nodding cycles
numPacsSaa	long	Number of PACS reference Solar Aspect Angles
numRasterCol	long	Number of raster columns
numRasterLines	long	Number of raster lines
numScanLines	long	Number of scan lines
numSpectra	long	Number of Spectra
numSpireSaa	long	Number of SPIRE reference Solar Aspect Angles
object	String	target name
objectType	String	astronomical object type
observer	String	name of observer
obsid	long	Observation Identifier
obsMode	String	Observation mode name'
odNumber	long	operational day number
offPosFlag	boolean	Off-position flag
onTargetFlag	boolean	On-target flag
origin	String	site that created the product
outOfFieldFlag	boolean	Out-of-field flag
pc1_1	double	PC1_1 element of PC matrix
pc1_2	double	PC1_2 element of PC matrix
pc1_3	double	PC1_3 element of PC matrix
pc2_1	double	PC2_1 element of PC matrix
pc2_2	double	PC2_2 element of PC matrix
pc2_3	double	PC2_3 element of PC matrix
pc3_1	double	PC3_1 element of PC matrix
pc3_2	double	PC3_2 element of PC matrix

pc3_3	double	PC3_3 element of PC matrix
pixelRow	long	Pixel row index
plwBiasAmpl	double	PLW bias amplitude
pmwBiasAmpl	double	PMW bias amplitude
pointingMode	String	Pointing mode identifier
posAngle	double	Position Angle of pointing
proposal	String	proposal name
pswBiasAmpl	double	PSW bias amplitude
ptcBiasAmpl	double	PTC bias amplitude
qualityFlag	int	Quality flag
ra	double	Actual Right Ascension of pointing
raDeSys	String	Coordinate reference frame for the RA and DEC
raErr	double	Error on Right Ascension of actual pointin
raNominal	double	requested RA of pointing
raObject	double	RA of target object
rasterColumnNum	long	Raster column number
rasterLineNum	long	Raster line number
readouts	double	sample readouts for one ramp
references	String	References
refPixel	long	Reference Pixel
roll	double	Spacecraft roll angle
saa	double	Reference SAA value in the range 0-180 degrees
saturation	double	Fraction of saturated samples
satValuesSigned	long	Saturation values signed modes
satValuesUnsigned	long	Saturation values unsigned modes
scanLineNum	long	Scan line number
scope	string	Scope
sedVersion	String	Version of the SED
serendipityFlag	boolean	SPIRE serendipity mode flag
siamId	String	Reference to the applicable SIAM
skyResolution	double	Spatial resolution
slewFlag	boolean	Slew flag
slwBiasAmpl	double	SLW bias amplitude
sswBiasAmpl	double	SSW bias amplitude
source	String	Source packet
sourceDetector	String	Detector Source Packet
sourceSmec	String	SMEC Source Packet
specNum	long	Spectrum Number
spectralResolution	double	Spectral resolution of data

startDate	FineTime	Start date of observation
startWavelength	double	Begin of wavelength interval
status	String	Pixel status or channel status
strInterlacingStatus	boolean	STR interlacing status
strQualIdx	double	STR quality index
subinstrumentId	String	Sub-instrument identifier
subsystem	String	Instrument Subsystem
telescope	String	Name of telescope
temperature	double	
type	String	Product type identification
variability	String	Information on object variability
version	String	version of product
wavelengthId	long	Key Wavelength ID
wcsReference	String	Reference of Coordinate System
wcsType	String	Type of Coordinate System
wheelPos	long	Wheel position
zeroPointOffset	double	Zero point offset

Chapter 8. Overview of data processing packages

Software in the HCSS is organized into a tree or hierarchy of packages. Each package contains one or more classes. Java development automatically leads to documentation of packages, classes, and their methods in the *Javadoc*, known in the HCSS help system as the *Developer's Reference Manual (DRM)*. This chapter explains when, why and how to use the DRM, and provides an overview of the main DP packages only. A full listing of packages and classes available in your HCSS installation is given in the API documentation, which you can access by selecting *HCSS Developer's Reference Manual (API)* from the HIPE Help System table of contents.

When to use the DRM. Java developers will already be familiar with the *DRM* or *Javadoc*. Users of HIPE will need it only in certain specialised cases:

- When a class is not documented in the User's Reference Manual.
- When you want to view the exhaustive list of methods for a class.
- When you want to browse through all the classes in a given package.
- When you want to browse all the packages in your installation of HIPE.

Definitions. These Java development terms will be encountered in the following sections.

- *Class*. The (Java) class of a product defines the type of object it is. In object-oriented programming a class is a construct that is used as a blueprint to create objects, so all objects of a particular class will have the same organisation and definition.
- *Method* . In object-oriented programming a method is a group of (software) instructions that is given a unique name and can be called up at any point by simply quoting the name. In other languages a method is called a function, subroutine or procedure. Example: `> myTask.getSomething();` the getSomething() is a method for myTask and it returns an answer that depends on what is (or is not) in the ().
- *Constructor*. Constructors are methods that create objects of a particular class.
- *Attribute*. An attribute is a field that denotes a particular characteristic of the class, much like a property of a class. For example, an important attribute of an observation context is the observation ID.
- *Interface*. An interface is a named collection of method definitions (without implementations). An interface can also include constant declarations. A class that implements the interface agrees to implement all of the methods defined in the interface, thereby agreeing to certain behaviour.
- *Package*. Multiple classes of larger programs are usually grouped together into a package. Packages correspond to directories in the file system, and may be nested just as directories are nested.

Packages discussed in this chapter. A number of DP packages are discussed elsewhere in some detail. The *Numeric* package was discussed in Chapter 5, while the *Plot* and *Display* packages are discussed in the *Data Analysis Guide*. Illustrations of how to use parts of several other HCSS packages are also shown in other chapters.

To access functionality within HCSS packages you have to *import* it into your HIPE session. For many packages this is done automatically by default; if not you can do it manually via commands like the following:

```
from herschel.ia.numeric import *
```

Example 8.1. Importing a complete package.

8.1. Browsing the list of packages

The Javadoc is normally started up as three frames in a web browser as illustrated in Figure 8.2 The upper left frame contains the *packages index* which is a clickable list of all packages in the system. The title in that frame represents the HCSS build number for which this documentation is valid. The lower left frame contains the *classes index* which is a clickable list of all classes. The selection of classes shown in this frame depends on the package that was selected in the packages index frame. The *Main frame* contains overview information on the system and packages or shows the page for a selected class.

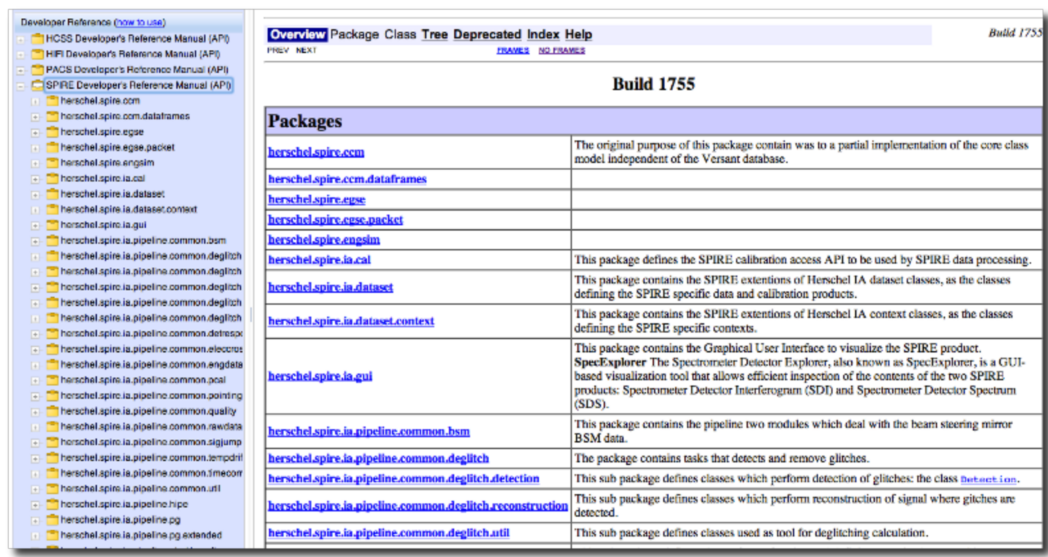


Figure 8.1. View of SPIRE packages after opening up and clicking on SPIRE Developer's Reference Manual (API) in the HIPE help window.

If you are comfortable with Javadoc documentation, you can access it from the HIPE Help System by clicking on any of the developer reference manuals listed in the Developer Reference section at the bottom of the table of contents. To obtain the traditional frame-based Javadoc layout, click on the FRAMES link on any Javadoc page. To get back to the HIPE Help System layout, use the Back button of your browser (clicking on the NO FRAMES link will not work). To have both layouts available, open the Javadoc layout in a new tab or window of your browser, by right-clicking on the FRAMES link.

Then you will see the Javadoc as three frames as illustrated in Figure 8.2 The upper left frame contains the *packages index* which is a clickable list of all packages in the system. The title in that frame represents the HCSS build number for which this documentation is valid. The lower left frame contains the *classes index* which is a clickable list of all classes. The selection of classes shown in this frame depends on the package that was selected in the packages index frame. The *Main frame* contains overview information on the system and packages or shows the javadoc for a selected class.

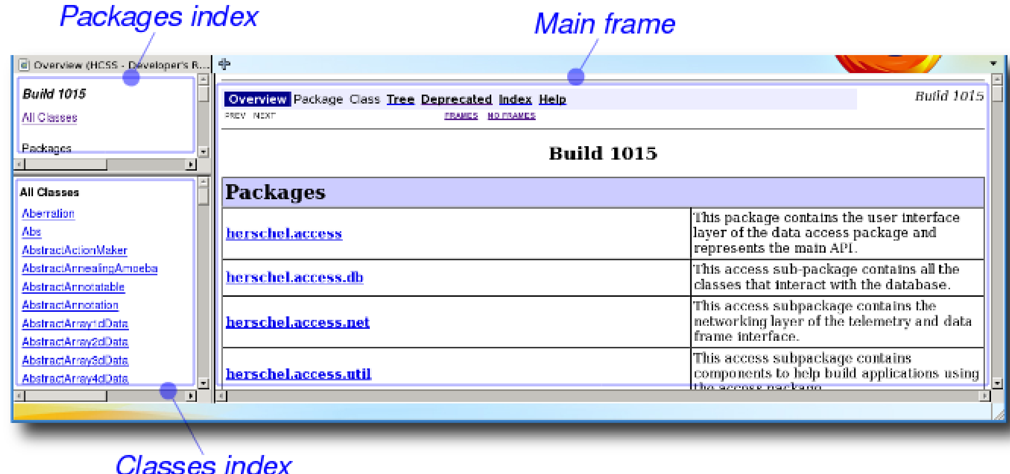


Figure 8.2. Web browser page of JavaDocs top level frame.

Click in the *Packages index* frame to select a package and update the *Classes index* frame to show those classes for the selected package. Click the *Classes index* frame to show the javadoc of a particular class in the *Main frame*.

The *Main frame* contains a kind of navigation bar at the top where the view in this frame can be selected. The figure above shows the overview of all the packages. Other views are: Package, Class, Tree, Deprecated, Index, and Help. These views will be explained in more detail below. In the overview the Package and Class views are disabled, they become available when a package or class is selected. Figure 8.3 shows the slightly expanded navigation bar for the Class view.

Figure 8.3. Navigation bar on the *class view* of JavaDocs.

Note that the navigation bar provides the possibility to browse through packages and classes with NEXT and PREVIOUS and provides direct access to the specific parts of the class documentation e.g. constructors (start class/program) or methods (which can be thought of as sub-routine components of programs that can be applied). It is also possible to switch between FRAMES and NO FRAMES. With NO FRAMES only the *Main frame* of the javadoc will be shown and index frames become unavailable.

8.2. Browsing the contents of a package

Each package has a page that contains a list of its classes and interfaces, with a summary for each. This page can contain four categories*: Interfaces summary, Classes summary, Exceptions and Error summary*. Not all categories are always present. At the end there is the package description and possible links to specific and/or related documentation.

Figure 8.4 shows the `herschel.ia.dataset` package which contains a number of interface and classes such as `Column` and `TableDataset`. The *Classes index* frame provides a clear separation of interfaces and classes and the *Main frame* shows the interface and class summaries and provides a brief package description with links to package specific info at the bottom. You can navigate to the interface and class detailed documentation by clicking the names in the summary tables or in the left sidebar.

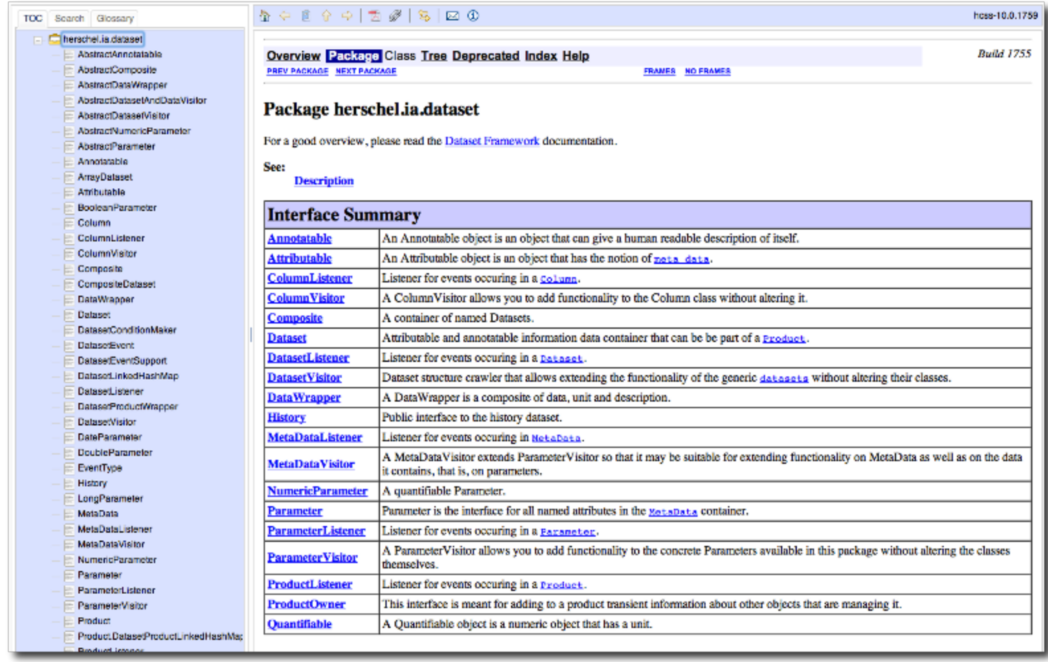


Figure 8.4. Package description page in Developer's Reference Manual.

8.3. Viewing the details for a class or interface

Each class and interface has its own separate page in the *DRM*. There are at least two ways to reach a class description page. From within HIPE, right-clicking on a variable and choosing will bring up the page for the corresponding class in the HIPE help system in your web browser. When viewing the contents of a package as described in Section 8.2, clicking on a class name in the sidebar or the main frame brings up the class description page.

Each of these pages has three sections consisting of a class/interface description, summary tables for constructors and methods, and detailed descriptions of constructors, methods and attributes. The information shown in the class view is restricted to the public *API (Application Programming Interface)*.

Each summary entry contains the first sentence from the detailed description for that item. The summary entries are alphabetical, while the detailed descriptions are in the order they appear in the source code. This preserves the logical groupings established by the programmer.

Figure 8.5 is taken from the *Main frame* of the *TableDataset* class and shows the class description together with its hierarchy. You can see that the *TableDataset* implements a number of interfaces and also has many subclasses. The second part of the figure shows a more detailed description of the class usage.

herschel.ia.dataset

Class TableDataset

java.lang.Object
└ herschel.ia.dataset.AbstractAnnotatable<Dataset>
  └ **herschel.ia.dataset.TableDataset**

All Implemented Interfaces:
Annotatable, Attributable, Dataset, MetaDataListener, TableModel

Direct Known Subclasses:
BitPatternDataset, CalAccumDataset, CalAcquisitionDataset, CommandsDataset, DeltaVDataset, DemoDataset, EventsLogDataset, GsDtcpDataset, HorizonsDataset, MonitorCCUADataset, MonitorCCUBDataset, ObservationBlockExecutionDataset, ObsTable, OolBaseDataset, OrbitEphemDataset, ParameterDataset, ParameterDataset, PointingDataset, PointingRequestsDataset, ProposalDataset, RawAccumDataset, RawAcquisitionDataset, RwlDataset, SourceFittingProduct, SourceListDataset, StrictTableDataset, TagsDataset, TeTable

```
public class TableDataset
extends AbstractAnnotatable<Dataset>
implements Dataset, TableModel, MetaDataListener
```

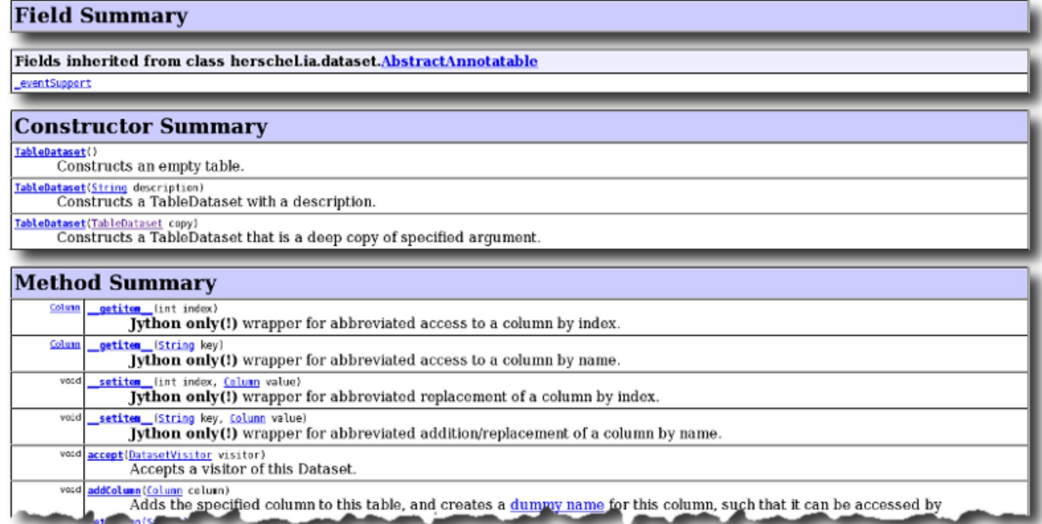
A TableDataset is a tabular collection of Columns. It is optimized to work on array data as specified in the herschel.ia.numeric package. The column-wise approach is convenient in many cases. For example, one has an event list, and each algorithm is adding a new field to the events (i.e. a new column, for example a quality mask).

Although mechanisms are provided to grow the table row-wise, one should use these with care especially in performance driven environments as this orthogonal approach (adding rows rather than adding columns) is expensive.

Examples of actual ArrayData objects can be found in the herschel.ia.numeric package, and therefore they will not be discussed here.

Figure 8.5. The class view of TableDataset showing a brief description and a short example of its usage.

Scrolling down in the *Main frame* brings you to the summary section which is shown in Figure 8.6. The constructor summary shows all public constructors for this class with their specific argument list. To see detailed information on the constructor click the name of the constructor that you need. Constructors are methods that create objects of a particular type. The code example in the description section above shows you how to create a *TableDataset* on the *jython* command line.



Field Summary

Fields inherited from class herschel.ia.dataset.AbstractAnnotatable
_eventSupport

Constructor Summary

Constructor
TableDataset() Constructs an empty table.
TableDataset(String description) Constructs a TableDataset with a description.
TableDataset(TableDataset copy) Constructs a TableDataset that is a deep copy of specified argument.

Method Summary

Return	Method
Column	__getitem__(int index) **Jython only(!)** wrapper for abbreviated access to a column by index.
Column	__getitem__(String key) **Jython only(!)** wrapper for abbreviated access to a column by name.
void	__setitem__(int index, Column value) **Jython only(!)** wrapper for abbreviated replacement of a column by index.
void	__setitem__(String key, Column value) **Jython only(!)** wrapper for abbreviated addition/replacement of a column by name.
void	accept(DatasetVisitor visitor) Accepts a visitor of this Dataset.
void	addColumn(Column column) Adds the specified column to this table, and creates a dummy name for this column, such that it can be accessed by

Figure 8.6. Page showing the constructor mechanism (how to create a TableDataset) and the associated set of methods (what you can do with the TableDataset you created).

The method summary shows all public methods for this class in alphabetical order. For detailed information on a specific method, click its name. The return values of the methods are in the left column while the method signature and a summary line is in the right column. The summary line can be preceded with a **deprecation** note. Deprecation means that this method should not be used anymore because it is marked to be removed from future releases. The deprecation comment normally provides the alternate method to be used instead. An overview of all deprecated methods in the whole system is available from the navigation bar at the top of the *Main frame*.

Sometimes method names can start and end with two underscore characters like in `__getitem__` above. These methods are special constructs which allow you to use the specific Jython syntax to access and manipulate objects from this class.

8.4. Displaying alternative views of the Developer's Reference Manual

- **Tree view**

  There is a Class Hierarchy page for all packages, plus a hierarchy for each package. Each hierarchy page contains a list of classes and a list of interfaces. The classes are organised by inheritance structure starting with *java.lang.Object*. The interfaces do not inherit from *java.lang.Object*. When viewing the Overview page, clicking on "Tree" displays the hierarchy for all packages. When viewing a particular package, class or interface page, clicking "Tree" displays the hierarchy for only that package.

- **Deprecated view**

  The Deprecated API page lists everything that has been *deprecated*. A deprecated item is not recommended for use, and a replacement is usually suggested.

  Deprecated items may be removed in future versions.

- **Index view**

  The Index contains an alphabetic list of all classes, interfaces, constructors, methods, and fields.

8.5. DP packages

The following short paragraphs outline the packages currently available within the Herschel DP system. For full details please see the Javadoc.

8.5.1. herschel.ia.dataflow

Handles processing threads. Particularly useful for Quick Look Analysis (QLA) and Standard Product Generation (SPG). It can be used in interactive sessions too. Allows the user to connect scripts from process modules as is typically required for a set of data reduction steps. Dataflow also supports event-based processing as well as threads.

Main subpackages:

- **herschel.ia.dataflow.data.process:** Classes for handling the processes used in a dataflow session.
- **herschel.ia.dataflow.example.indicator_control.monothread:** Classes used to illustrate the control of a dataflow.
- **herschel.ia.dataflow.example.indicator_control.multithread:** Same as above, but for multiple threads.
- **herschel.ia.dataflow.template:** Class to allow template dataflow to be created.
- **herschel.ia.dataflow.util:** Class for identifying dataflows.

8.5.2. herschel.ia.dataset

Contains *Table Datasets*, *Array Datasets*, *Composite Datasets*, *Products* and all auxiliary components such as columns, parameters and metadata. Datasets and products are described in Chapter 2.

Main subpackages:

- **herschel.ia.dataset.demo:** Contains classes that demonstrate the use of datasets.

- **herschel.ia.dataset.gui:** Contains the *Dataset Inspector* graphical interface.
- **herschel.ia.dataset.history:** Defines the *History Dataset*, which records the complete history of the tasks which were executed to produce a Product.
- **herschel.ia.dataset.image:** Provides a framework for defining images, cubes of images and stacks of images. Includes tools for adding World Coordinate System information.
- **herschel.ia.dataset.spectrum:** Contains tools for defining one- and two-dimensional spectra, and spectral cubes.

8.5.3. herschel.ia.document

Provides tools to generate documentation of dynamic as well as static DocBook documents in different formats.

8.5.4. herschel.ia.gui

Contains several subpackages related to graphical applications.

Main subpackages:

- **herschel.ia.gui.apps:** Contains the classes used to build graphical applications such as HIPE.
- **herschel.ia.gui.cube:** Graphical interfaces to analyse data cubes.
- **herschel.ia.gui.explorer:** Graphical interfaces to analyse datasets, such as TablePlotter and OverPlotter.
- **herschel.ia.gui.image:** Classes for handling images. The display capabilities from this package are discussed in the *Data Analysis Guide*.
- **herschel.ia.gui.plot:** Plotting utilities. For more details see the *Data Analysis Guide*.

8.5.5. herschel.ia.io

Provides a means of accessing local archives where Products can be saved or loaded from. Products are combinations of data and information and can be likened to the contents of a single FITS file.

Main subpackages:

- **herschel.ia.io.ascii:** Allows input and output of data to and from ASCII files.
- **herschel.ia.io.dbase:** Allows data/products to be put into objects that can be stored in databases (Versant databases are currently available for use with the HCSS).
- **herschel.ia.io.fits:** A FITS implementation that can write Products to a FITS file and read such FITS files back into the system. Allows the production of a FITS archive.

8.5.6. herschel.ia.numeric

Contains numeric and mathematical tools described in Chapter 2 and Chapter 5.

Main subpackages:

- **herschel.ia.numeric.toolbox:** Provides a large set of numeric classes. These include mathematical functions (trigonometric functions, polynomials), Fourier transforms, fitter functions, interpolation and matrix functions. Note that these classes are automatically loaded when starting HIPE.

  This package contains the following subpackages:

- **herschel.ia.numeric.toolbox.basic:** Provides classes that allow basic mathematical manipulation of numeric arrays: trigonometric functions, mathematical product, variance and so on.
- **herschel.ia.numeric.toolbox.filter:** Provides the classes `BoxCarFilter`, `Convolution` and `GaussianFilter`.
- **herschel.ia.numeric.toolbox.fit:** Provides classes that allow the fitting of data with numerous models (iterative fitters, sine model fitters, polynomial model fitters and so on).
- **herschel.ia.numeric.toolbox.integr:** Provides integrator functions for several integral models (Gauss-Jacobi, Gauss-Laguerre and so on).
- **herschel.ia.numeric.toolbox.interp:** Provides classes that allow the interpolation of data. These include `Interpolator` (a general interpolator), `LinearInterpolator`, `CubicSplineInterpolator` and `NearestNeighborInterpolator`.
- **herschel.ia.numeric.toolbox.mask:** Provides tools for creating and managing masks, in particular the two classes `FixedMask` and `PackedMask`.
- **herschel.ia.numeric.toolbox.matrix:** Provides classes that allow the manipulation of `Double2d` arrays holding matrices. It includes the classes `MatrixDeterminant`, `MatrixInverse` and `MatrixSolve`.
- **herschel.ia.numeric.toolbox.random:** Provides tools for generating pseudo-random numbers with uniform (`RandomUniform`), Gaussian (`RandomGauss`) and Poisson (`RandomPoisson`) distributions.
- **herschel.ia.numeric.toolbox.stat:** Provides statistical tools for arrays, to compute covariance (`Covariance` and `CovarianceMatrix`), geometric mean (`GeoMean`) and mode (`Mode`).
- **herschel.ia.numeric.toolbox.util:** Provides the classes `MoreMath`, which has methods for mathematical manipulation of single numerical elements (integers, doubles, bytes and so on), and `Util`, which has utilities for converting arrays.
- **herschel.ia.numeric.toolbox.wavelet:** Provides algorithms to perform continuous, discrete and stationary wavelet transforms.
- **herschel.ia.numeric.toolbox.xform:** Provides the classes `FFT`, `FFT_PACK`, `RealDoubleFFT`, `FFT_PACK_EVEN`, `FFT_PACK_ODD`, `Hamming` and `Hanning` for Fourier transforms and Hanning/Hamming smoothing of data.

8.5.7. herschel.ia.obs

Defines the *Observation Context*, a container for Products applicable to a specific obervation, and related classes.

Main subpackages:

- **herschel.ia.obs.auxiliary:** Defines the auxiliary Products related to an observation, and their container, the *Auxiliary Context*.
- **herschel.ia.obs.cal:** Calibration-related classes.
- **herschel.ia.obs.quality:** Defines the *Quality Context* and the flags used for quality control.

8.5.8. herschel.ia.pal

Defines the *Product Access Layer*, which allows storage and retrieval of Products both locally and remotely. The Product Access Layer is treated in detail in Chapter 7.

Main subpackages:

- **herschel.ia.pal.browser:** Defines the *Product Browser* graphical application.
- **herschel.ia.pal.io:** Defines classes for importing and exporting Products to FITS format.
- **herschel.ia.pal.pool:** Defines, in various subpackages, the available types of *Product Pools*.
- **herschel.ia.pal.query:** Defines the types of query that can be applied to a *Product Storage*.

8.5.9. herschel.ia.pg

Describes the *Product Generation Framework*, on which running of instrument pipelines is based.

Main subpackages:

- **herschel.ia.pg.od:** Defines the *Operational Day Plugin*, used to process an entire OD *before* processing its observations.
- **herschel.ia.pg.plugins:** Defines basic versions of other plugins that are applied during pipeline processing, such as `BasicLevel0Plugin` and `BasicQualityPlugin`.

8.5.10. herschel.ia.qcp

Defines components and utilities to handle Quality Control messages.

Main subpackages:

- **herschel.ia.qcp.example:** Provides an example Task for using the facilities of this package.
- **herschel.ia.qcp.flags:** Provides a hierarchical structure of Quality Control flags.
- **herschel.ia.qcp.gui:** Provides graphical components for displaying Quality Control messages.
- **herschel.ia.qcp.plugin:** Provides plugins for logging Quality Control messages during Operational Day and pipeline processing.
- **herschel.ia.qcp.tools:** Provides a standalone application for displaying Quality Control information.

8.5.11. herschel.ia.spg

Manages the execution of the data reduction process for all the instrument in the Herschel satellite. It is built upon the framework defined in the herschel.ia.pg package (see Section 8.5.9).

Main subpackages:

- **herschel.ia.spg.gui:** Contains the *Pipeline Manager* graphical interface.
- **herschel.ia.spg.kayako:** Contains a helper class for creating a ticket in the *kayako* system.
- **herschel.ia.spg.od:** Tools for scheduling and executing Operational Day processing.
- **herschel.ia.spg.ops:** Miscellaneous tools for configuring pipeline processing.
- **herschel.ia.spg.tools:** Classes for memory monitoring and the remote management of processing queues.

8.5.12. herschel.ia.task

Provides the tools needed to create a data processing *task* which you can then incorporate into your scripts. For more information on tasks please see Chapter 6.

Main subpackages:

- **herschel.ia.task.example:** Provides example Tasks that demonstrate some features of the package.
- **herschel.ia.task.gui:** Provides components used to build graphical interfaces for Tasks.
- **herschel.ia.task.history:** Provides a class for managing the history of a Task.
- **herschel.ia.task.mode:** Provides different execution modes for a Task (interactive, on demand, systematic and test).
- **herschel.ia.task.util:** Miscellaneous utility functions for Task development.

8.5.13. herschel.ia.toolbox

Provides tools for a wide range of data analysis needs. Tools are organized in thematic subpackages.

Main subpackages:

- **herschel.ia.toolbox.astro:** Astronomical utilities.
- **herschel.ia.toolbox.cube:** Tasks for importing and analysing data cubes.
- **herschel.ia.toolbox.fit:** Tasks for function fitting.
- **herschel.ia.toolbox.hsa:** Provides an interface for accessing the Herschel Science Archive.
- **herschel.ia.toolbox.image:** Tasks for image processing (cropping, smoothing and so on).
- **herschel.ia.toolbox.mapper:** Tasks for mapmaking.
- **herschel.ia.toolbox.pointing:** Provides a task for plotting pointing information.
- **herschel.ia.toolbox.spectrum:** Tasks for analysing spectra. This package contains several subpackages, among which are the following:
  - **herschel.ia.toolbox.spectrum.fit:** Tools for fitting spectra.
  - **herschel.ia.toolbox.spectrum.operations:** Tools for performing mathematical operations on spectra (divide, average, resample and so on).
  - **herschel.ia.toolbox.spectrum.projection:** Tools for projecting spectral data on the sky.
  - **herschel.ia.toolbox.spectrum.selections:** Tools for selecting and managing ranges and discrete values within spectra.
  - **herschel.ia.toolbox.spectrum.standingwaves:** Tools for fitting and removing fringes.
  - **herschel.ia.toolbox.spectrum.utils:** Other utilities, for example to integrate and interpolate spectra.
- **herschel.ia.toolbox.srcext:** Tools for point source extraction.
- **herschel.ia.toolbox.util:** Miscellaneous tools, among which are tasks for importing from and exporting to ASCII tables and FITS files.

8.5.14. herschel.ia.vo

Contains tools that implement the interface to the Virtual Observatory.

8.5.15. herschel.share.fltdyn

Contains classes and interfaces related to flight dynamics, such as time measurement and ephemerides.

Main subpackages:

- **herschel.share.fltdyn.constraint:** Classes to define time intervals and time constraints.
- **herschel.share.fltdyn.ephem:** Classes to define planetary ephemerides and spacecraft orbital ephemerides.
- **herschel.share.fltdyn.math:** Mathematical classes for handling spacecraft attitudes, rotations and coordinates.
- **herschel.share.fltdyn.time:** Time classes with microsecond resolution and handling of leap-seconds. For more information see Chapter 9.

Chapter 9. Time and astronomical measurements

The first part of this chapter describes how time is defined within HCSS and how to deal with it. Unfortunately, there are several ways in which time can be represented. The standard for the HCSS/DP is a `FineTime` - which is the number of microseconds since the beginning of 1 January 1958. This provides the kind of accuracy needed to represent time on a space mission.

However, there are several other time representations and it is often the case that conversions between times/dates is necessary. In particular, it is noted that the standard Java commands lead to date measurements with respect to 1 January 1970. This chapter indicates how to deal with times within DP and converting between the various times, particularly between dates and `FineTime`'s.

The last section of this chapter explains how to carry out great circle and position angle calculations.

9.1. Time Definitions

9.1.1. System time in HIPE

There are many ways to access the system time in HIPE. See also the description of the Java class "Date" for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

The Java `Date` class is deprecated and is being replaced by a more flexible `SimpleDateFormat` capability within Java that allows the user to express dates more conveniently. A `Date` object is still obtained and can be turned into a `FineTime` (see below) once created.

Two possibilities for creating a "Date" object are:

```
# To get the current time in milliseconds:
# The difference, measured in milliseconds, between the current
# time and midnight, January 1, 1970 UTC.
print java.lang.System.currentTimeMillis()
# To get the number of milliseconds since
# January 1, 1970, 00:00:00 GMT represented by a Date object.
d = java.util.Date()
#printing this gives the current time and date at the location of the
#system on which the java is being run.
print d
#We can also get the number of milliseconds since Jan 1, 1970 using
#this Java Date
print d.getTime()
```

Example 9.1. How to obtain the current time by various methods.

Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger.

If we want to get the number of milliseconds since 1 January 1970 for any other date then we can use a non-default form of the Java Date capability where the year, month, day, hour, minute and second are provided.

- Year format -- year (A.D.) - 1900. So the year 2006 = 2006 - 1900 = 106
- Month format -- number of the month, beginning from January = 0. E.g. March = 2.
- Day -- just day number in the month.

- Hours, minutes, seconds -- on the 24-hour clock.

NOTE: This is the time on our computer system.

```
#Format of date is year (in units of true year - 1900), month (number 0...11),
#day, hour, minute, second. So the following gives us the number of milliseconds
#between the beginning of 1 January 1970 and 3:15:00 pm on 23 October 2004.
d = java.util.Date(104, 9, 23, 15, 15, 0)
print d # should indeed show we have 3:15pm on 23 October 2004
print d.getTime() # provides the number of milliseconds between this
#date and 1 Jan 1970.
```

Example 9.2. Different ways of formatting time variables.

The following sample code shows how to use `SimpleDateFormat` to create a "Date" object.

```
simpleDate = java.text.SimpleDateFormat("yyyy.MM.dd HH:mm:ss z")
#set up how you want to set up your input Date format. In this
#case we could input "2006.01.14 01:00:00 CST" for 1a.m. on 14
#January 2006. z -- indicates the time zone (default is the zone for the
#computer system being used).

simpleDate.applyPattern("dd/MM/yy HH:mm")
#change the pattern to a different format

startTime = simpleDate.parse("13/01/06 14:06")
#create the data object "startTime"

print startTime
#...and see what this looks like
```

Example 9.3. Creating a date object.

Allowed choices for the data format are available from Java documentation of the SimpleDateFormat capability.

9.1.2. International Atomic Time (TAI) and `FineTime`

TAI is an international standard measurement of time based on the comparison of many atomic clocks. TAI is the basis for Coordinated Universal Time (UTC). `Finetime` is based on TAI as measured from 00:00:00 1 January 1958.

9.1.3. Coordinated Universal Time (UTC)

UTC, World Time, is the standard time common to every place in the world. UTC is derived from International Atomic Time (TAI) by the addition of a whole number of "leap seconds" to synchronise it with Universal Time 1 (UT1), thus allowing for the eccentricity of the Earth's orbit and the rotational axis tilt (23.5 degrees), but still showing the Earth's irregular rotation, on which UT1 is based.

9.1.4. DecMec Time [PACS only]

The commands DPUSelectTime and DPUWriteTime are selecting and setting a start time which is written to the TMP1 and TMP2 fields of the Dec/Mec headers. This is used in coordinating the activities of the mechanical devices on board PACS. It is possible to construct an absolute time by adding counters (CRDC) to the start time considering an offset between setting and writing the start time.

This offset is expected to be a number with an uncertainty depending on the system load. It might require a calibration file. Currently this offset is not considered.

In case the commands and are not given the TMP1 and TMP2 fields are zero. To avoid software confusions the time will be related to a fixed date (1.Jan 1970, 0:00).

During construction of the SpuBuffer the time is computed from the TMP1, TMP2 entries in the Dec/Mec header and the CRDC counter. This time is used during construction of the DataFrameSequence and the associated Tables holding the SPU science data.

Between the Dec/Mec time and the packet time (see the `PusTmBinStruct` class in the herschel.binstruct package) we have an offset. Therefore the association between HK and science data will be within an accuracy of 2 seconds.

9.2. Time in Instrument House-Keeping (HK) Data

The most convenient method of obtaining time stamped HK information is the use of the herschel.binstruct package.

When dealing with HK time information directly, it is important to know that telemetry packets contain the time as defined within the "PUS Data Field Header". The field represents the on-board reference time of the packet, referenced to TAI, expressed in spacecraft time units - CCSDS Unsegmented Time Code (CUC) units. CUC units are multiples of 1/65536 sec from 1 January 1958 in TAI time. CUC units cannot be expressed in whole microseconds but can be converted to the *FineTime* standard (see below).

CUC time is written for HK by the data processing unit (DPU).

Current `PusTmBinStruct` methods related to time:

long getTime()

Returns the packet time of the Pus telemetry packet.

boolean isTimeSynchronized()

Returns true if the telemetry packet is synchronized, false otherwise.

java.util.Date getTimeAsDate()

Returns the packet time as a Date object.

FineTime getTimeAsFineTime()

Returns the packet time of the Pus telemetry packet as FineTime.

9.3. Time conversion

9.3.1. Time conversion in HIPE

It can often be the case that we need to convert between FineTime (TAI) and Date (UTC). Coordinated Universal Time is expressed using a 24-hour clock and uses the Gregorian calendar. FineTime represents a TAI time (epoch 1958), whereas the Java Date class is used to represent UTC, by resetting the system clock whenever a leap second occurs and don't need to handle leap seconds. Converting between Java dates and the FineTime standard requires the use of the DateConverter() class. Long integers can also be directly converted to FineTimes and are interpreted as representing the number of microseconds since 00:00:00 1 January 1958. In Example 9.4 we illustrate how to create a FineTime from a long integer and convert back and forth between FineTime and Java Dates.

```
# FineTime to Date
# Enter a time in seconds (a long integer - put letter "l"
# at the end of the number)
c = FineTime(1436094449715400l) # convert to a FineTime
```

```
# Prints corresponding date and time
print DateConverter.fineTimeToDate(c)
# Date to a FineTime
d = java.util.Date() # gets today's date and time
# Prints corresponding FineTime
print DateConverter.dateToFineTime(d)
```

Example 9.4. Time conversion between `Date` and `FineTime`

9.3.2. CucConverter

Converts between Spacecraft Elapsed Time, in CCSDS Unsegmented Time Code (CUC) format and FineTime (TAI). This implementation is for the Herschel CUC format, which is corrected on-board the spacecraft to TAI (epoch 1 Jan 1958). This representation uses 32-bits for seconds and 16 bits for fractional seconds. CUC times are multiples of 1/65536 sec and cannot be expressed as an exact multiple of 1 microsecond (the resolution of FineTime). However, the following relations hold for 'coarse' and 'fine' values in the allowed range:

long coarse(FineTime t)

Return the number of whole seconds since the epoch 1 Jan 1958.

long cucValue(FineTime t)

Return the number of 1/65536 fractional seconds since the epoch 1 Jan 1958.

int fine(FineTime t)

Return the fractional part of the number of 1/65536 seconds since the epoch 1 Jan 1958.

FineTime toFineTime(long cuc)

Return a new FineTime constructed from a 48-bit CUC time.

FineTime toFineTime(long coarse, int fine)

Return a new FineTime constructed from CUC coarse & fine fields.

```
from herschel.share.fltdyn.time import *

d=CucConverter.toFineTime(50000000000000L)
#Converts the long integer - representative of a CUC time -
#into a FineTime. The FineTime is stored in d.
e = CucConverter.coarse(d)
#provides the number of whole seconds since 1 Jan 1958
#and stores it in e.
print e
```

Example 9.5. Creating FineTime variables from other time formats.

9.4. Great circle and position angle calculations

This functionality is available within the herschel.share.fltdyn package. This package is a low level library written for the mission planning system. It uses radians throughout, except where indicated in the method names (for instance, `Direction.fromDegrees(ra,dec)` and `Direction.getRaDegrees()`). Please see the following example:

```
HIPE> from herschel.share.fltdyn.math import *
HIPE> x = Direction(125./180*Math.PI, 80./180*Math.PI)
HIPE> y = Direction(125./180*Math.PI, 70./180*Math.PI)
```

```
HIPE> print x.distanceTo(y)
0.17453292519943298
HIPE> print x.distanceTo(y)/Math.PI*180
10.000000000000002
HIPE> z = Direction(100./180*Math.PI,80./180*Math.PI)
HIPE> print x.distanceTo(z)/Math.PI*180
4.307863243850451
```

Note the manual conversions between radians and degrees throughout the example. However, the `Direction` class has factory methods that accept degrees. Conversion of the result to degrees is most elegantly achieved using the `Math.toDegrees()` method. So, the previous example simplifies to the following:

```
HIPE> x = Direction.fromDegrees(125,80)
HIPE> y = Direction.fromDegrees(125,70)
HIPE> print Math.toDegrees(x.distanceTo(y))
10.000000000000002
```

The position angle is given by the following command:

```
pos = x.positionAngleTo(y)
```

Example 9.6. Calculating the angle between vectors.

The `Direction` class is recommended for calculations of any complexity. It does all the internal calculations using the `Vector3` and `Quaternion` classes, which are fast and avoid problems with singularities at the poles.

Appendix A. Jython operators

The following tables shows all the various operators you can use in Jython. HCSS and HIPE use Jython version 2.5.2.

This list and the associated operator descriptions have been largely taken from the Python Reference Manual, which you can find online at http://docs.python.org/reference/.

Table A.1. Jython unary arithmetic operators

Operator	Operator description	Example
+	**Unary plus**: yields its numeric argument unchanged.	`print +5` `#5`
-	**Unary minus**: yields the negation of its numeric argument.	`print -5` `#-5`
~	**Invert**: yields the bitwise invert of its plain or long integer argument.	`print ~5` `#-6`

Table A.2. Jython binary arithmetic operators

Operator	Operator description	Example
+	**Sum**: yields the sum of its arguments.	`print 2 + 2` `# 4`
-	**Subtraction**: yields the difference of its arguments.	`print 2 - 3` `# -1`
*	**Multiplication**: yields the product of is arguments.	`print 3 * 2` `# 6`
/	**Division**: yields the quotient of its arguments.	`print 5 / 2` `# 2` `print 5.0 / 2` `# 2.5`
//	**Floor division (Jython 2.2 alpha only)**: yields the result of the `floor()` function applied to the quotient of its arguments.	`print 5 // 2` `# 2` `print 5.0 // 2` `# 2.0`
%	**Modulo**: yields the remainder from the division of its arguments.	`print 5 % 2` `# 1`
**	**Power**: yields its left argument raised to the power of its right argument.	`print 5**2` `# 25`

Table A.3. Jython shifting operators

Operator	Operator description	Example
<<	**Left shift**: `a << b` shifts plain or long integer `a` by `b` bits.	`print 5 << 1` `# 10`
>>	**Right shift**: `a >> b` shifts plain or long integer `a` by `b` bits.	`print 5 >> 1` `# 2`

Table A.4. Jython binary bitwise operators

<table>
<tr><th>Operator</th><th>Operator description</th><th>Example</th></tr>
<tr><td>&</td><td>Bitwise AND: yields the bitwise AND of its plain or long integer arguments.</td><td><code>print 5 & 6</code>
<code># 4</code></td></tr>
<tr><td>^</td><td>Bitwise XOR: yields the bitwise exclusive OR of its plain or long integer arguments.</td><td><code>print 5 ^ 6</code>
<code># 3</code></td></tr>
<tr><td>|</td><td>Bitwise OR: yields the bitwise inclusive OR of its plain or long integer arguments.</td><td><code>print 5 | 6</code>
<code># 7</code></td></tr>
</table>

Table A.5. Jython comparison operators

Operator	Operator description	Example
`<`	**Less than**: `a < b` yields true if `a` is less than `b`.	`print 5 < 6` `# 1`
`>`	**Greater than**: `a > b` yields true if `a` is greater than `b`.	`print 5 > 6` `# 0`
`==`	**Equal to**: `a == b` yields true if `a` and `b` are equal.	`print 5 == 6` `# 0`
`>=`	**Greater or equal to**: `a >= b` yields true if `a` is greater than or equal to `b`.	`print 5 >= 6` `# 0`
`<=`	**Less or equal to**: `a <= b` yields true if `a` is less than or equal to `b`.	`print 5 <= 6` `# 1`

`!=` (preferred) or `<>`	**Not equal to**: `a != b` yields true if `a` is not equal to `b`.	`print 5 != 6`
		`# 1`
		`print 5 <> 5`
		`# 0`

Table A.6. Jython boolean operators

Operator	Operator description	Example

`and`	**Boolean AND**: yields `True` if both arguments are true, `False` otherwise.	`print 1 and 0`
		`# 0`
		Example A.1. Boolean and operation between integers is also valid.

`or`	**Boolean OR**: yields `True` if at least one argument is true, `False` otherwise.	`print 1 or 0`
		`# 1`
		Example A.2. Boolean or operation between integers is also valid.

`not`	**Boolean NOT**: yields `True` if the argument is false, `False` otherwise.	`print not 1`
		`# 0`
		Example A.3. Boolean not operation between integers is also valid.

Appendix B. Naming conventions

B.1. Naming Conventions

for Java and Jython users and developers. Version 0.3, 6th December 2006

Element	Description	Naming convention
Class	Defines the state and behaviour of something. Classes are defined as declaring variables (fields) and functions (methods) associated with the objects of that class.	Names should be nouns and written in mixed case starting with an upper case letter. Do not use underscores to separate words. `DataFrameGenerator, FitsArchive`
Interface	Defines a collection of method definitions and constant values. It can later be implemented by classes that define this interface with the `implements` keyword.	Names have the same convention as class names but are preferably adjectives. Try to end the names with -able or -ible: `Sortable, Accessible, Savable`
Variable	An item of data named by an identifier. Each variable has a type, such as `int` or `Frame`, and a scope.	Names should be mixed case starting with a lower case letter. Do not use underscores to separate words. `frameBufferCounter, nSamples, line, detectorNo`
Instance Variable	A variable that is part of an object.	Names should start with an underscore, otherwise follow the general conventions for variables (see above). `_packetType, _isVisible`
Local Variable	A variable that is part of a function or method.	Names follow the naming convention of normal variables. `counter, length, pixelName`
Constant	A variable whose value can not be changed during execution.	Names should be all uppercase using an underscore to separate words: `MAX_ITERATIONS`
Boolean variable and method	A logical type/function that can only have or return the values 'true' or 'false'. Methods have parentheses () while variables haven't.	Names should start with is-, has-, can-, or should-. `isVisible, hasChanged(), canHandle(), shouldAbort`
Parameter	An argument to a function or a method.	Names follow the naming convention of normal variables.

Element	Description	Naming convention
		`name, packet`
Property	A platform independent implementation of environment variables and settings.	Names should be all lower case and start with 'hcss'. The hierarchical parts should be separated with a dot. `hcss.binstruct.services`
Method	A function defined in a class.	Names should be verbs and written in mixed case starting with a lower case letter. Do not use underscores to separate words. `getName(), load()`
Function Section 1.27	A jython function is a collection of code lines to perform a specific task under one name. Functions take arguments and provide one output. They are like methods, except they are not inside a class. A function can also be an instance of the Task class.	Names follow the same convention as method names in classes. `resample(), readTm()`
Numeric function Section 5.1	Parameterless Java functions provided by the herschel.ia.numeric toolboxes. For these functions only one instance is needed. Other numeric functions follow the same convention as classes.	Names are in all uppercase with an underscore to separate words. `UNIQ, MEDIAN, IS_FINITE`
Task Chapter 6	A Task is a class which can be called as a function. Tasks do input and output parameter type checking and provide history to Products.	Names follow the same conventions as for classes. Task names should end with the word 'Task'. `DisplayDataFrameTask, ResampleTask`
Package	Defines a collection of related classes and interfaces in Java. Packages provide the namespace in Java and Jython.	Names should be in lower-case letters and digits, don't use underscores. `herschel.ia.numeric` Package names should be short so that the fully qualified package name doesn't become excessively long.

Abbreviations and acronyms should **not** be all uppercase when used as a name:

GOOD	BAD
`exportAsHtml()`	`exportAsHTML()`
`saveAsJpeg()`	`saveAsJPEG()`
`OolPacket`	`OOLPacket`

Using all uppercase for the abbreviations in base names will give conflicts with the naming conventions given above. A variable of this type would have to be named `hTML`, `jPEG` etc. which obviously is not very readable. Another problem is illustrated in the examples above: when the name is connected to another, the readability is seriously reduced, since the word following the acronym does not stand out as it should.

The term *compute* can be used in methods where something is computed and might take considerable time to execute.

`computeAverage(), matrix.computeInverse()`

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

The *'n'* prefix should be used for variables representing a number of objects, note that the names are plural.

`nPoints, nLines, nSamples`

The notation is taken from mathematics where it is an established convention for indicating a number of objects. Note that Sun uses the `num` prefix in the core Java packages for such variables. This is probably meant as an abbreviation of number of, but as it looks more like number it makes the variable name strange and misleading. If "number of" is the preferred phrase, `numberOf` prefix can be used instead of just `n`. The `num` prefix must not be used.

The *'No'* suffix should be used for variables representing an entity number.

`tableNo, employeeNo`

The notation is taken from mathematics where it is an established convention for indicating an entity number.

Reserved words: the following words are reserved by Java as language keywords and can not be used for variables, methods or class names in Java.

`abstract, continue, for, new, synchronized, assert, default, goto, package, this, boolean, double, if, private, throws, break, do, implements, protected, throw, byte, else, import, public, transient, case, enum, instanceof, return, try, catch, extends, interface, short, void, char, finally, int, static, volatile, class, final, long, super, while, const, float, native, switch.`

B.1.1. Jython code example

```
# herschel.ia.dataset.gui = PACKAGE; DatasetInspector = CLASS
from herschel.ia.dataset.gui import DatasetInspector
# PI = CONSTANT
from java.lang.Math import PI
# testName = VARIABLE
testName = "chop_freq_test_2909_1832_1902_"
# load = METHOD
t2 = fits.load(myDir+testname+"PHOTF.fits").default
# MAX = NUMERIC FUNCTION
maxStep = MAX(step[step.where(step < 0xffff)])
# startEndTimes = FUNCTION; step, maxStep, time... = FUNCTION PARAMETERS
def startEndTimes(step, maxStep, time, startTime, endTime):
  for i in range(0, maxStep): # i = LOCAL VARIABLE
    temp=(step.where(step == i+1))
    endTime[i] = time[MAX(temp.toInt1d())
  return endTime
# len = FUNCTION
```

```
upper = len(startarr)
```

B.1.2. Java code example

```
package herschel.ia.numeric; // herschel.ia.numeric: PACKAGE
public final class Complex1d // Complex1d: CLASS
    implements Serializable // Serializable: INTERFACE
{
    private transient double[][] _internal; // _internal: INSTANCE VARIABLE
    // writeObject: METHOD
    private void writeObject(ObjectOutputStream os) { // os = METHOD PARAMETER
        os.defaultWriteObject();
        os.writeInt(length());
        if (length()==0) return;
        for (int i=0,n=length();i<n;i++) { // i = LOCAL VARIABLE
            os.writeDouble(_re[i]); os.writeDouble(_im[i]);
        }
    }
}
```

Index

Symbols

& (Jython bitwise operator), 46
^ (Jython bitwise operator), 46
__doc__, 7
| (Jython bitwise operator), 46

A

ABS, 80
Aliases, 25
Amoeba fitter, 93
and (Numeric arrays method), 47
and logical operator, 16, 46
ARCCOS, 80
ARCSIN, 80
ARCTAN, 80
Arctan fitting model, 92
Arithmetic operators, 44
Array datasets, 49
  creating, 49
  inspecting, 50
  modifying, 50
Arrays
  rectangular and jagged, 41
  removing NaN and infinite values from, 47
Astrometry, 74
  correcting, 77
astrometryFix, 78
AttribQuery, 121

B

Backslashes
  in filenames, 19
Binomial fitting model, 91
Bitwise operators, 46
bool (Jython variable type), 5
Bool1...5d, 40
Boxcar filters, 88
break (Jython keyword), 17
Byte1...5d, 40

C

Cached pool, 128
  using with an HSA pool, 127
  using with an HTTP pool, 128
CEIL, 80
Character (Java variable type), 9
ChiSquared, 104
class (Jython keyword), 23
Classes, 23
  aliases, 25
  creating, 23
  Naming conventions, 25
Code blocks, 15
Columns (of table datasets), 50
complex (Jython variable type), 5
Complex numbers, 5
Complex1...5d, 40
  inspecting, 42
  real and imaginary parts, 41
Composite datasets, 52
  creating, 52
  inspecting, 53
  modifying, 53
Compound fitting models, 93
Console view, 3
Constants, 57
  Naming conventions, 25
Contexts (see Product contexts)
continue (Jython keyword), 18
Convolution, 87
Coordinated Universal Time, 146
Correlate, 104
CorrelateMatrix, 104
COS, 80
Covariance, 104
CovarianceMatrix, 104
CPython, 2
creationDate (product metadata), 59
Cubic splines fitting model, 92
CUC, 147
CucConverter, 148

D

Data fitting
  tolerance, 94
Date (Java class), 145
DecMec time, 146
def (Jython keyword), 20
Determinant of a matrix, 98
Dialogue windows, adding, 30
Dictionaries, 12
  accessing, 14
  creating, 14
  modifying, 14
  nesting, 15
Direction (HIPE class), 148
Directory
  finding current, 29
  listing files, 29
Double1...5d, 40

E

Eigenvalue decomposition, 99
elif (Jython keyword), 16
else (Jython keyword), 16
  in for loops, 16
endDate (product metadata), 59
Erf, 104
Erfc, 104
EXP, 80
Exponential fitting model, 92

External software, interoperating with, 33

F

FFT (discrete Fourier transform), 82
FFT_PACK, 82
Filenames
  backslashes in, 19
Files
  reading numeric values from, 20
  reading strings from, 19
  writing numeric values to, 20
  writing strings to, 19
FineTime, 146
FITS files
  importing into a pool, 119
FitterFunction, 103
Fitters, 93
  tolerance, 94
Fitting data, 89
  additional documentation, 97
  fit execution, 90
  model selection, 89
  one-dimensional example, 95
  statistical information, 90
  two-dimensional example, 96
FIX, 80
FixedMask, 97
float (Jython variable type), 5
Float1...5d, 40
FLOOR, 80
for loops, 16
FullQuery, 121
Functions, 20
  aliases, 25
  as function arguments, 22
  without input arguments, 22

G

GAMMALN, 104
GammaP, 104
GammaQ, 104
GaussHermiteIntegrator, 102
Gaussian filters, 88
Gaussian model, 92
GaussianQuad4Integrator, 101
GaussianQuad5Integrator, 101
GaussJacobiIntegrator, 102
GaussLaguerreIntegrator, 102
GaussLegendreIntegrator, 101
GEOMEAN, 104
get (Numeric array method), 45
getcwd (Jython function), 29
glob (Jython module and function), 29
Global variables, 21
Great circle, 148

H

hcss.interpreter.imports property, 27
Help, 7
herschel.ia.dataflow, 139
herschel.ia.dataset, 139
herschel.ia.document, 140
herschel.ia.gui, 140
herschel.ia.io, 140
herschel.ia.numeric, 140
herschel.ia.obs, 141
herschel.ia.pal, 141
herschel.ia.pg, 142
herschel.ia.qcp, 142
herschel.ia.spg, 142
herschel.ia.task, 142
herschel.ia.toolbox, 143
herschel.ia.vo, 143
herschel.share.unit package, 53
HifiSpectrumDataset, 72
History (of products), 60
HrsSpectrumDataset, 72
HSA pool, 127
  metadata, 129
HTTP pool, 128
HttpClientPool, 128

I

IDL, 3
IDL equivalents
  arithmetics commands, 38
  basic commands, 36
  data import/export commands, 37
  plot commands, 36
if (Jython keyword), 16
IllegalArgumentException, 126
Imaginary numbers, 5
import (Jython keyword), 25
IndexError, 126
Infinite values
  removing from arrays, 47
Input arguments of a function, 20
  default values, 22
int (Jython variable type), 5
Int1...5d, 40
INT_TABULATED (IDL function), 103
Integral transforms, 81
Integration, 101
  discrete values, 102
  functions, 101
Interactive prompt, 3
International Atomic Time, 146
Interpolation, 88
  of discrete data, 103
IntTabulated, 103
Inverse Fourier transforms, 84
Inverse of a matrix, 98

J

Java, 2
 ranges of numeric types, 5
 variable types, 5
Javadoc, 134
 class view, 137
 constructor summary, 138
 deprecated view, 139
 method summary, 138
 overview, 135
 package view, 136
 tree view, 139
JOptionPane, 30, 31, 32
Jython, 2
 operators, 150
  binary arithmetic, 150
  bitwise, 150
  boolean, 151
  comparison, 151
  shifting, 150
  unary arithmetic, 150
 ranges of numeric types, 6
 variable types, 5

K

KURTOSIS, 104

L

Lambda expressions, 79
Levenberg Marquardt fitter, 93
Linear fitting models, 91
List contexts, 60
listdir (Jython function), 29
Lists, 12
 accessing, 13
 appending values, 13
 concatenating, 13
 creating, 12
Local pools, 118
 directory, 118
  changing, 118
 importing FITS files, 119
 repairing, 119
Local stores (see Local pools)
LOG, 80
LOG10, 80
Logical operators, 46
long (Jython variable type), 5
Long1...5d, 40
Loops
 breaking, 18
 for, 16
 in the Console view, 18
 while, 17
Lorentz fitting model, 92
LU decomposition, 99

M

Map contexts, 60
Masks, 97
Matrices, 97
Matrix equations, 100
MAX, 80
MEAN, 80, 104
Measurement units (see Units of measurement)
MEDIAN, 80, 104
MedianAbsoluteDeviation, 104
Metadata, 57
 inspecting, 58
 modifying, 57
 querying, 120
MetaQuery, 121
Methods, 23
 calling, 24
MIN, 80
Mixed fitting models, 93
MODE, 104
Modules
 importing, 25
  your own, 27
 reloading, 28
 unimporting, 28
Modulo maxima line, 106
Multiple line commands, 3
Multiplying matrices, 98

N

Naming conventions, 25, 152
 Java example, 155
 Jython example, 154
NaN
 removing from arrays, 47
Non-ASCII characters in scripts, 4
Non-linear fitting models, 92
 user-supplied, 92
Normalization (Fourier transform), 85
not logical operator, 46
Numeric arrays, 40
 creating, 41
 differences from Jython arrays, 42
 differences with Jython arrays, 40
 element-by-element operations, 44
 improving performance, 47
 inspecting, 41
 modifying, 42
 ordering of elements, 43
 selecting and filtering values, 44
 type conversions, 48
  explicit, 48
  implicit, 49
Numeric library, 79
 basic functions, 80
  lower case equivalents, 81
 using functions, 79

numpy, 3

O

Objects, 23
- Naming conventions, 25
- printing contents, 24

Observation contexts, 60
open (Jython keyword), 19
Operators, 150
or (Numeric arrays method), 47
or logical operator, 16, 46
os (Jython module), 29, 34

P

PackedMask, 97
PacsCube, 72
PacsRebinnedCube, 72
pause(), 33
Photometer Astrometry Correction, 78
pickle (Jython module), 20, 20
Pipeline scripts, 29
Plug-ins, sharing scripts with, 35
PointSpectrum, 62
Polynomial fitting model, 91
Pools, 116
- creating, 116
- for remote data, 127
- local (see Local pools)
- minimising memory usage, 124
- product versioning, 121
- querying, 119
  - advanced, 121
  - inspecting results, 120
  - part of a string, 123
  - product metadata, 120, 123
  - product versions, 121
- registering to a storage, 116
- testing if two products are equal, 124
- tips and pitfalls, 123
- troubleshooting, 119
- wrong tag after renaming, 125

Position angle, 148
Power law fitting model, 91, 92
Power spectrum, 86
print (Jython keyword), 18
Printing
- formatting printouts, 18
- numeric values to file, 20
- strings to file, 19
- to the screen, 18

PRODUCT, 80
Product contexts, 60
- copying from a storage to another, 125

ProductRef, 117
Products, 58
- automatic metadata, 58
- copying from a storage to another, 125
- creating, 58
- deleting from a pool, 117
- history, 60
- inspecting, 60
- loading from a pool, 117
- modifying, 59
- saving to a pool, 117
- setting date and time in metadata, 59
- tagging in a pool, 117

Python, 2

Q

Query (HIPE class), 119

R

Random numbers, 100
RandomGauss, 100
RandomPoisson, 100
RandomUniform, 100
range function, 16
read(), 19
readline(), 19
readlines(), 19
RectangularIntegrator, 101
Relational operators, 44
reload (Jython function), 28
REPEAT, 80
resume(), 33
Return parameter of a function, 20
REVERSE, 80
RombergIntegrator, 101
ROUND, 80

S

Scripting
- getting started, 1

Scripts
- debugging, 33
- maximum length, 4
- menu in HIPE, 2
- pausing, 33
- resuming, 33
- running, 2
- sharing, 35
- third-party, 29
- writing, 4

Selection (Java class), 45
Short1...5d, 40
showConfirmDialog, 32
showInputDialog, 31
showMessageDialog, 30
Sigclip, 104
SIGNUM, 80
SimpleDateFormat (Java class), 146
SimpleSpectrum, 69
SimpsonIntegrator, 101
SIN, 80

Sinc fitting model, 92
Sinc function convolved with Gaussian model, 92
Sine wave fitting model
  linear, 91
  non-linear, 92
Singular value decomposition, 100
Singular value decomposition fitter, 94
SKEWNESS, 104
Slices, 42
Spectral datasets and products, 62
  instrument-specific, 72
SpectralSegment, 62
SpectralSimpleCube, 70, 72
  creating, 70
  dimensions, 72
  inspecting, 70
SpectrometerDetectorSpectrum, 72
SpectrometerPointSourceSpectrum, 72
Spectrum1d, 62
  creating, 63
  inspecting, 64
Spectrum2d, 66
  creating, 67
  inspecting, 68
SpectrumContainer, 62
SpirePreprocessedCube, 72
SQRT, 80
SQUARE, 80
startDate (product metadata), 59
Statistics functions, 104
StatWithNaN, 104
STDDEV, 80, 104
Storages, 116
  copying products and contexts, 125
  creating, 116
  listing registered pool, 117
  wrong tag after renaming pool, 125
String1d, 40
Strings, 8
  converting to numeric values, 10
  formatting, 9
  Java types, 9
  printing to file, 19
  printing to screen, 18
  reading from file, 19
SUM, 80
Swing (Java GUI library), 30
system() (Jython function), 34

T

Table datasets, 50
  copying, 52
  creating, 50
  inspecting, 52
  modifying, 51
Tags (for products in pools), 117
  checking if a tag exists, 118
  pointing to wrong product, 125
  removing, 118
TAI, 146
TAN, 80
Tasks, 113
  calling from a script, 22
  parameters, 114
    multiple outputs, 114
  printing help, 113
  printing parameters, 113
  running, 113
Time
  conversions, 147
    between CUC and TAI, 148
    between TAI and UTC, 147
  measurement, 145
Tolerance (data fitting), 94
Transposing a matrix, 98
TrapezoidalIntegrator, 101
Tuples, 12
  accessing, 13
  concatenating, 13

U

Units of measurement, 53
  comparing for compatibility, 56
  converting
    to and from strings, 55
    to other units, 56
  creating and assigning, 54
  derived units, 55
  multiples and fractions, 55
URN, 116
UTC, 146

V

Value, passing by, 21
Variables, 4
  converting between Java and Jython types, 11
  converting between Jython types, 10
  deleting, 4
  getting help, 7
  Java types, 5
  ranges of Java numeric types, 5
  ranges of Jython numeric types, 6
  types, 7
VARIANCE, 104
Version
  Developer builds, 34

W

Wavelet transforms, 104
  continuous, 105
    example, 105
  discrete, 107
  library, 106
  stationary, 109
  toolbox overview, 111

tools, 110
Gaussian noise estimator, 110
thresholding tool, 110
universal threshold, 110
WbsSpectrumDataset, 72
WCS (see World Coordinate System)
WeightedMean, 104
WHERE (IDL function), 46
where (Numeric array method), 44
while loops, 17
World Coordinate System, 74

X

xor (Numeric arrays method), 47