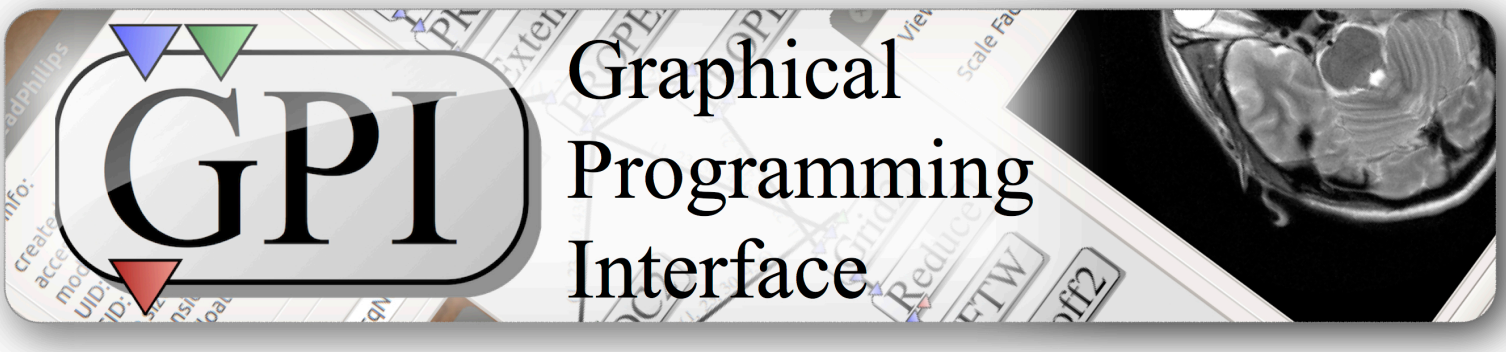

GPI Node Developer's Guide

Summary

This initial version of the Node Developer's Guide is a reference for widget and port attributes that can be set in a GPI node and various methods used to interact with the GPI infrastructure.

Contents

1. Widgets

The widget methods, types, and attributes described in this section are further clarified in the example code:
interfaces/GPI/Template_GPI.py

Widgets are visual interfaces associated with nodes to enter and retrieve a wide variety of values, e.g. floats, integers, strings, lists, images. Widgets have many attributes associated with them, which affect their behavior in a variety of ways. They are instantiated using self.addWidget() and modified using self.setAttr(). Their values and attributes are retrieved in using self.getVal() and self.getAttr().

1.1 Widget Methods

self.addWidget() is used in the initUI section to create the unique widget. It defines the widget type, the unique name, and then the desired options. For example, a DoubleSpinBox widget is instantiated with min and max settings, an initial value, and visibility turned on using:

```
self.addWidget('DoubleSpinBox','approximate_pi',min = 3.1, max = 4.0, val = 3.14159,
visible=True)
```

self.setAttr() is used in the validate and compute sections to define or change the value of one or more widget attributes. It references the unique name given by self.addWidget(). For example, the value and number of significant digits to display are set using:

```
self.setAttr('approximate_pi',val=3.14159265358979323846264338 3,decimals=27)
```

self.getAttr() is used in the validate and compute sections to retrieve the value of a single attribute from a widget. It references the unique name given by self.addWidget(). For example, The "number of significant digits" attribute is retrieved using:

```
precision = self.getAttr('approximate_pi', 'decimals')
```

self.getVal() is used in the validate and compute sections to retrieve the value associated with a unique widget. It references the unique name given by self.addWidget(). For example, the value of a widget is entered into an equation using:

```
circum = diameter*self.getVal('approximate_pi')
```

1.2 Widget Types and Associated Attributes

Generic Attributes:

These attributes are shared by all widgets:

attribute	possible values	description
collapsed	boolean (True, False)	Only hide the 'collapsable' elements of a widget within its groupbox.
id	integer	INTERNAL USE ONLY: used for saving network information that will aid in reinstantiation.
inport	boolean (True, False)	Turn the widget inport on/off
outport	boolean (True, False)	Turn the widget outport on/off
reset	boolean (True, False)	Automatically reset widget
val	see widget descriptions below	Actual Value represented by widget
quietval	same type as val option	Set the central value of a widget without triggering an event.
visible	boolean (True, False)	Set the widget groupbox's visibility within the node menu

ComboBox Widget

Provides a popup list for different labels. Value is string. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
items	list of strings, e.g. [‘apple’,‘banana’,‘orange’]	list of items to choose from. Only one item can be chosen from the list.
index	integer	gives the corresponding index of the chosen item, starting at 0. So, for example, self.getAttr(“myfruitbox”,index) would return a 2 if ‘orange’ were selected from the item list given in this table, while self.getVal(“myfruitbox”) would return the string ‘orange’.

DisplayBox Widget

A 2D QPixmap using a QLabel using built-in interpolation schemes. Value is Qimage(). Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
interp	boolean (True, False)	Interpolated scaling / nearest neighbor
line	---	TODO: create an image_annotations class
noscroll	boolean (True, False)	Turn display window scroll on/off
pixmap	QPixmap	A QPixmap to be displayed
points	---	TODO: create an image_annotations class
scale	float	Pre-defined image dimension scale

DoubleSpinBox Widget

Spin box for floating point values. Value is double. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
decimals	integer	The number of displayed decimal places
immediate	boolean (True, False)	Make the value changes immediate with scroll wheel etc
label	string	Set a text left of the spinbox
max	float	Maximum float value
min	float	Minimum float value
singlestep	float	The stepsize of one up/down button click
wrapping	boolean (True, False)	Allow up/down buttons to cause a wrap when exceeding max or min values

ExclusivePushButtons Widget

Provides a set of buttons for different labels. Buttons are placed side-by-side, and only one may be selected at a time. Returned Value is an integer corresponding to which button is pressed, starting at 0 for the leftmost button. Value is integer. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
buttons	A list of labels (e.g. ['b1', 'b2','b3','b4'])	The list of labels are placed on buttons. For the list given here, a value of 0 is returned for button b1, a value of 1 is returned for button b2, and so on.

ExclusiveRadioButtons Widget

Similar to ExclusivePushButtons widget except the that check boxes, stacked vertically, are used instead of side-by-side buttons. Returned Value is an integer corresponding to which button is pressed, starting at 0 for the topmost checkbox. Value is integer. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
buttons	A list of labels (e.g. ['b1', 'b2','b3','b4'])	The list of labels are placed next to vertically stacked checkboxes. For the list given here, a value of 0 is returned for checkbox b1, a value of 1 is returned for checkbox b2, and so on.

GenericWidgetGroup Widget

This is the base-class for all widgets. It provides abstract methods and default behavior. From the node-developer's perspective, this provides the widget-port, visibility, and collapsibility options. Attributes are given in the Generic Attributes section above.

NonExclusivePushButtons Widget

Similar to ExclusivePushButtons widget except that multiple buttons can be selected at once. Provides a set of check boxes for different labels. Buttons are stacked top-to-bottom, and only one may be selected at a time. Returned Value is a list of integers corresponding to which buttons are pressed, starting at 0 for the leftmost button (e.g. [0,3,4]). Value is list of integers. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
buttons	A list of labels (e.g. ['b1', 'b2','b3','b4'])	The list of labels are placed on buttons. The returned list of integers includes 0 if button b1 is pressed, 1 if button b2 is pressed, and so on.

OpenFileBrowser Widget

Provide a QFileDialog() at the push of a button. Value is string. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
button title	string	A title centered on the pushbutton
caption	string	Set browser title-bar
directory	string	Set the default directory
filter	string	Set the file extension filter (e.g. 'io_field (*.fld);;all (*)').

PushButton Widget

A simple single pushbutton box. Value is boolean. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
button title	string	A title centered on the pushbutton
toggle	boolean (True, False)	Make the pushbutton a toggle button on/off (otherwise a single-push)

SaveFileBrowser Widget

Provide a QFileDialog() at the push of a button. Value is a string. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
button title	string	A title centered on the pushbutton
caption	string	Set browser title-bar
directory	string	Set the default directory
filter	string	Set the file extension filter (e.g. 'io_field (*.fld);;all (*)').

Slider Widget

Basic slider widget. Value is an integer. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
max	integer	maximum allowed value
min	integer	minimum allowed value

SpinBox Widget

Basic spin box. Value is integer. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
immediate	boolean (True, False)	Make the value changes immediate with scroll wheel etc
label	string	Set a text left of the spinbox
max	integer	Maximum integer value
min	integer	Minimum integer value
singlestep	integer	The stepsize of one up/down button click
wrapping	boolean (True, False)	Allow up/down buttons to cause a wrap when exceeding max or min values

String Widget

A simple single line string box. Value is string.

TextBox Widget

Provides a multi-line plain-text display. Value is string. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
wordwrap	boolean (True, False)	Turn off/on wrap

TextEdit Widget

Provides an editable text window with scrollbar and python code syntax highlighting. Value is string.

WebBox Widget

For loading web urls. Value is string. Available attributes in addition to the Generic Attributes above:

attribute	possible values	description
passwd	string	user password for login sites (plain text, no privacy)
username	string	user ID for login sites

2. Ports

The methods described in this section are further clarified in the example code: interfaces/GPI/Template_GPI.py

Input and output ports are used to pass data into and out of nodes, respectively. They can pass different types of data (e.g. numpy arrays, dictionaries, etc.) and can limit accepted data types using attributes. Ports are created self.addInPort() and self.addOutPort(). Data are retrieved from input ports using self.getData() and sent to output ports using self.setData().

2.1 Port Methods

self.addInPort() is used in the initUI section to create an input port. It defines the unique port name, the data type, and the desired options. For example, after importing numpy, one can specify a 4-byte float numpy array that is either 2 or 3 dimensions using:

```
self.addInPort('kspacefilter','NPYArray',ndim=[2,3], dtype=numpy.float32)
```

self.addOutPort() is used in the initUI section to create an output port. It defines the unique port name, the data type, and the desired options. For example, one can specify an output port that will contain a dictionary using:

```
self.addOutPort('filteredDataDesc','DICT')
```

self.getData() is used in the validate and compute sections to retrieve the data from an input port. It defines the unique port name, and returns the data. For example, one can assign the data from an input port to a variable kfilt using:

```
kfilt = self.getData('kspacefilter')
```

The method returns NULL if no data are present at the port. This can be used to check if data are present at input ports set to gpi.optional

self.setData() is used in the compute section to assign data to an output port. It defines the unique port name, and the data. For example, one can assign the a dictionary contained in oxfordDict to an output port using:

```
self.setData('filteredDataDesc', oxfordDict)
```

2.2 Port Data Types and Associated Attributes

For self.addInPort and self.addOutPort defined in section 2.1, the 2nd argument is the type of data associated with the port. The possible types are listed below, along with the attributes that can be associated with them.

General Attributes

These attributes apply to all data types below.

attribute	possible values	description
obligation	gpi.REQUIRED gpi.OPTIONAL	determines whether a port must have data before the validate or compute methods will be executed in a node. Default is gpi.REQUIRED.
cyclic	boolean (True, False)	This is a special port type that allows cyclic connections, e.g. for the "iteration" class nodes. Default is False.

GLOList data type

Allows passing gpi-gl object definitions to be passed in lists.

NPYarray data type

The NPYarray type provides port enforcement for numpy multidimensional arrays (ndarray). The enforcement parms are type (ndarray), dtype, ndim, dimension range (drange), shape, and vec (the len of the last dim). Enforcement priority for aliased parms goes ndim->drange->shape. Although shape and vec can overlap, they are enforced independently. Attributes in addition to the general attributes above include:

attribute	possible values	description
dtype	NPY type or tuple or list, e.g. numpy.uint8 (numpy.float32, numpy.float64) [numpy.float32, numpy.float64]	Required array data type.
ndim	integer or tuple or list, e.g. 3 (2,5,11) [1,2]	Required number of dimensions.
shape	Integer tuple or list, e.g. (3, 4) [12,3,5]	Required dimension lengths.
vec	integer	Length of the last (most varying) dimension (i.e. shape[-1]).

COMPLEX data type

Standard python-complex number.

DICT data type

Standard python dictionary type.

FLOAT data type

Standard python float type. Attributes in addition to the general attributes above include:

attribute	possible values	description
range	float tuple or list, e.g. (0.5, 2.6) [-5.2,-2.003]	Specifies allowed data range (min, max).

INT data type

Standard python integer type. Attributes in addition to the general attributes above include:

attribute	possible values	description
range	integer tuple or list, e.g. (-10, 10) [0,2]	Specifies allowed data range (min, max).

LIST data type

Standard python list type.

LONG data type

Standard python long integer type. Attributes in addition to the general attributes above include:

attribute	possible values	description
range	integer tuple or list, e.g. (-10000000000, 71) [43,44]	Specifies allowed data range (min, max).

STRING data type

Standard python string type.

PASS data type

Allows any data type.

TUPLE data type

Standard python tuple type.

3. Node Interface Methods (Python)

The methods described in this section are further clarified in the example code:
interfaces/GPI/Template_GPI.py

3.1 "Top Level" Methods

These abstract methods are defined by the user and the three major sections of a GPI node. initUI() is required, validate() and compute() are optional.

***def initUI(self)**:* This part of the node will run in the constructor at instantiation (i.e. when the node is placed on the Canvas). It is used to define widgets and ports, which are displayed in the order they are defined (widgets top down, ports left to right).

***def validate(self)**:* This part of the node will run every time an event is being processed, always prior to the compute method. It is typically used to check/enforce compatibility of data and widget values, and set widget attributes such as min/max and visibility.

***def compute(self)**:* This part of the node will run every time an event is being processed, always after to the validate method. It is where (e.g.) the actual data computation occurs.

3.2 Widget Methods

See Section 1 for more details.

addWidget

addWidget(self, wdg=None, title=None, **kwargs)
wdg = (str) corresponds to the widget class name
title = (str) is the string label given in the node-menu
kwargs = corresponds to the set_<arg> methods specific to the chosen wdg-class.

getVal

getVal(self, title)
Returns get_val() from wdg-class (see getAttr()).

getAttr

getAttr(self, title, attr)
title = (str) wdg-class name
attr = (str) corresponds to the get_<arg> of the desired attribute.

setAttr

setAttr(self, title, **kwargs)
title = (str) the corresponding widget name.
kwargs = args corresponding to the get_<arg> methods of the wdg-class.

3.3 Input/Output Methods

See Section 2 for more details.

addInPort

addInPort(self, title=None, type=None, obligation= gpi.REQUIRED, cyclic=False, **kwargs)
title = (str) port-title shown in tooltips
type = (str) class name of extended type
obligation = gpi.REQUIRED or gpi.OPTIONAL (default REQUIRED)
kwargs = any set_<arg> method belonging to the GPIDefaultType derived class.

addOutPort

addOutPort(self, title=None, type=None, obligation= gpi.REQUIRED, **kwargs)
title = (str) port-title shown in tooltips
type = (str) class name of extended type
obligation = dummy parm to match function footprint
kwargs = any set_<arg> method belonging to the GPIDefaultType derived class.

getData

getData(self, title)
title = (str) the name of the InPort.

setData

setData(self, title, data)
title = (str) name of the OutPort to send the object reference.
data = (object) any object corresponding to a GPIType class.

3.4 Timing Methods

Frame code with starttime() and endtime() to measure wall time of computation. Optional text can be inserted.

self.starttime() # time your code, NODE level log

self.endtime('You can put text here if you want') # endtime w/ message

3.5 Logging Methods

Print messages (e.g. error messages) in the terminal/console window. GPI main menu (Debug -> Log Level) controls what level of log is printed. Text can be inserted as desired.

self.log.debug()
self.log.info()
self.log.node("hello from node level logger, running validation()")
self.log.warn("this is a bad code area")
self.log.error()
self.log.critical()

3.5 Event Checking Methods:

These methods allow the node to perform selective computation based on what activated the node (e.g. a widget event vs. a port event).

self.portEvents()

Returns the name of a port that received new data, or Null if no port has received new data since the last node execution.

self.widgetEvents()

Returns the name of a port that was activated, or Null if no port has received new data since the last node execution.

self.getEvents() # super set of events

Returns either the name of the last port to receive data or the last widget to have been changed (whichever occurred last)

Note on current behavior: Only the latest event for a node is kept. This means that if the following occurs for a given node (in the specified temporal order):

1) a user changes a widget
2) new data comes to an input port
3) The node executes

At this point,

a) the value of the widget is changed
b) the new data is at the input port
c) self.widgetEvent() is Null
d) self.portEvent() returns the port that received data.

This is because the data came after the widget was set. A future version of GPI will keep a list of all pending events since the last execution.

4. Extending/Embedding Python with PyFI (C++)

PyFI, or “Python Function Interface”, is a collection of macros and interface classes that simplify exposing C++ functions to the Python interpreter. The macros also reduce the amount of code needed to translate Numpy arrays in Python to the PyFI Array class in C++ (and vice versa).

PyFI can be used to extend or embed Python. Most of the time PyFI is used to speed up algorithms by moving them from Python to C/C++, extending Python. However, the vast Python library can still be leveraged from within C++ code by embedding Python, allowing the developer to make the occasional Python function call from C++ when something can be more easily accomplished through Python. The PyFI interface is separate from GPI and can be used to extend or embed Python in other C++ applications.

PyFI is located in the ‘core’ GPI library and can be included in a cpp file via:

```
#include “core/PyFI/PyFI.h”
```

The macros described in this section are demonstrated in the example code:

```
/opt/gpi/lib/core/PyFI/template_PyMOD.cpp
```

4.1 Extending Python (PyFunction Macros)

These macros are required to successfully compile a Python/C++ extension module (http://docs.python.org/2/extending/extending.html).

1) ***PYFI_FUNC(name), PYFI_START(), PYFI_END()****:* These macros are used to declare the function that will be available to the Python interpreter. PYFI_FUNC takes a function name as its argument. This is the name used in the PYFI_FUNCDESC and will be the name of the function available in Python. The PYFI_START and PYFI_END handle the Python input and output of the function (e.g. memory management and exception handling).

Ex:

```
PYFI_FUNC(myFunc)
{
        PYFI_START();

        /* your code goes here */

        PYFI_END();
}
```

2) ***PYFI_LIST_START_, PYFI_LIST_END_, PYFI_DESC(name, string)****:* These macros define the list of functions available within the compiled module. The list is made up of

PYFI_DESC() calls placed between the PYFI_LIST_START_ and PYFI_LIST_END_ macros. This group must be the last set of macro calls in the module file.

Ex:

```
PYFI_LIST_START_
    PYFI_DESC(myFunc, "Brief info about myFunc().")
PYFI_LIST_END_
```

4.2 Extending Python (Input/Output Macros)

PYFI_POSARG(*type, ptr****):*** This macro declares a pointer of the given type and converts the input args from the Python interface to the corresponding C++ variables. Valid types are *double*, *int64_t* (*long* depending on the OS), *std::string*, *Array<float>*, *Array<double>*, *Array<int32_t>*, *Array<int64_t>, Array<complex<float>* >, *Array<complex<double>* >.

Ex:

PYFI_POSARG(*double*, myInput1);

PYFI_KWARG(*type, ptr, default****):*** This macro declares a pointer of the given type and converts the input keyword argument (http://docs.python.org/2/tutorial/controlflow.html#keyword-arguments) to the pointed C++ variable, if it was passed. If the keyword arg is not used, then the default arg is set.

Ex:

```
double myDefault1 = 1.0;
PYFI_KWARG(double, myInput1, myDefault1);
```

PYFI_ERROR(*string****):*** This macro raises a Python *Runtime* exception and passes the error message contained in the string.

PYFI_SETOUTPUT(*ptr****):*** The output arguments are set using this macro. If more than one output exists, then all are packaged in a tuple. This macro will create and copy PyFI arrays (passed as *ptr*) to Python Numpy arrays in the Python session.

PYFI_SETOUTPUT_ALLOC(*type, ptr, dims****):*** If the output array size is known, before the algorithm code, this macro can be used to generate an output Numpy array that is accessible within the C++ code as a PyFI array. This is more time and memory efficient than using PYFI_SETOUTPUT with PyFI arrays. This macro only applies to PyFI arrays. 'dims' can be a std::vector<uint64_t> or a PyFI::ArrayDimensions object.

deb*:* This macro can be placed in the code to print out the line number and file name of the executed code.

coutv(*var****):*** This macro prints the name and contents of the variable 'var' passed to it.

4.3 PyFI Arrays

PyFI contains a simple array class that supports multi-dimensional indexing, overloaded operators (for simple math operations), a few common function interfaces (e.g. pseudo inverse and fft), index debugging and wrapping Numpy array objects.

The arrays support up to 10 dimensions. N-dimensional arrays support indexing as an ND array or as a 1D array. The arrays are initialized by default to a value of zero. The 'Array' class is a templated class that allows any type to be a basis element of the array. However, the types supported for export (by PyFI) between Python and C++ are: Array<float>, Array<double>, Array<complex<float> >, Array<complex<double> >, Array<int64_t>, Array<int32_t> and Array<uint8_t>.

An array wrapper to FFTW library is included in the PyFI::FFTW namespace. The implementation details can be found in:

```
/opt/gpi/lib/core/PyFI/PyFIArray_WrappedFFTW.cpp
```

Array Methods:

Constructors:

Array(std::vector<uint64_t> dims)
: Construct an array using a standard vector class containing the dimension sizes. This is the recommended way for dynamic dimensionality.

Array(uint64_t i, uint64_t j, ...)
: Construct arrays with integer arguments for the size of each dimension. The number of arguments determines the dimensionality.

Array Information:

ndim()
: The number of dimensions as a uint64_t type.

dimensions_vector()
: Returns a standard vector with the dimension sizes.

size()
: The total number of elements as a uint64_t type.

data()
: Returns a pointer to the contiguous data segment.

isWrapper()

Returns a bool indicating whether the array wraps an external data segment (usually a Numpy data segment).

Operators:

Array(uint64_t i, uint64_t j, ...)
: The indexing operator calculates multi-dimensional indices given the input integer arguments and returns the dereferenced pointer to the location in the data segment. This is the usual way for accessing array memory. All N-D arrays can also be accessed as 1-D arrays.

=, *=, /=, +=, -=
: The right-hand-side arguments can be a single element of the same type as the array or an array of the same type. Arrays must be the same 'size()'. Operations are on an element-wise basis (not matrix math).

+, *, -, /
: Math operators that work on both arrays and single elements. All operations are on an element-wise basis (not matrix math).

==, !=, <=, >=, <, >
: Inequalities return an Array<bool> object containing a bit-mask evaluated with the condition for each element. Works with Arrays or single elements (for quick thresholding).

Builtins:

sum()
: The sum of all elements returned as a datum of the base array type.

prod()
: The product of all elements returned as a datum of the base array type.

min(), max()
: The min or max of all elements returned as a datum of the base array type.

abs()
: Calculates the fabs() on an element-wise basis (operates on the array in-place)

any(T val)
: Returns true if any of the elements are equal to val.

any_infs(), any_nans()
: Checks for infs or nans respectively. Returns a bool.

clamp_max(T thresh), clamp_min(T thresh)
Sets arrays > or < thresh equal to thresh. Operates in-place.

mean(), stddev()
Calculates sample mean and standard-deviation of the array elements. Returns as a datum of the base array type.

as_ULONG(), as_FLOAT(), as_CFLOAT(), as_DOUBLE(), as_CDOUBLE(), as_LONG(), as_INT(), as_UCHAR()
Returns a copy of the array as the selected base type.

insert(Array<T> arr)
Insert the elements (centered in each dimension) of 'arr' into THIS array. If 'arr' is larger then the extra elements are cropped.

get_resized(std::vector<uint64_t>), get_resized(uint64_t), get_resized(std::vector<double>), get_resized(double)
Return a copy of THIS array inserted into a new array of a different size. Integer arguments indicate specific dimension sizes (isotropic for single value) and double arguments indicate a scale size of the original array dimensions.

reshape(std::vector<uint64_t)
Change the dimensionality of THIS array. The total size must not change.

4.4 Build Setup & Example

A PyFI Python extension module can be easily built using the 'gpi_make' command from a terminal shell. PyFI extensions are compiled into a library object file (.so for unix based platforms) via the 'distutils' module which part of the Python standard module library. PyFI modules should be placed in the GPI node library directory structure under the library specific to the modules function. For example a 'core' library module, used by the GPI node 'SpiralCoords' would be located in the 'spiral' sub-library:

```
/core/__init__.py                  # python pkg file
/core/spiral                       # sub-library
/core/spiral/__init__.py           # python pkg file
/core/spiral/spiral_PyMOD.cpp      # C++ extension module
/core/spiral/spiral.so             # compiled extension module
/core/spiral/GPI/SpiralCoords_GPI.py  # GPI node
```

The gpi_make script identifies extension modules by checking for the '_PyMOD.cpp' extension; other supporting .cpp files will be ignored as make targets.

A simple Python extension module 'mymath' might look like this:

Example: bni/math/mymath_PyMOD.cpp

```
#include “core/PyFI/PyFI.h”
using namespace PyFI;

PYFI_FUNC(add_one)
{
      PYFI_START();
      PYFI_POSARG(Array<float>, arr);

      Array<float> out_arr(*arr);
      out_arr += 1.0;


      PYFI_SETOUTPUT(&out_arr);
      PYFI_END();
}

PYFI_LIST_START_
      PYFI_DESC(add_one, “Adds one to each element in the array.”)
PYFI_LIST_END_
```

The mymath_PyMOD.cpp module is compiled by invoking the gpi_make from a terminal shell:

```
$ gpi_make mymath
```
or
```
$ gpi_make mymath_PyMOD.cpp
```

A debug flag can be set to compile the PyFI arrays in a debug mode, where all indexing will be checked against the array dimensions:

```
$ gpi_make --debug mymath
```

The gpi_make is configurable through the ~/.gpirc file (which can be generated from the GPI ‘Config’ menu). Under the ***[PATH]*** section there is a variable ***LIB_DIRS*** that can be configured to point to new GPI libraries. All libraries pointed to by ***LIB_DIRS*** will be included as searchable code and library paths in the gpi_make. NOTE: it is recommended that node developers create their own library for development and leave the ‘core’ library clean. This way new GPI releases won’t overwrite a developer’s development directory.

The Python code that uses this function would then look like this:

Example: test.py (placed in the same directory as ‘bni’)

```
import bni.math.mymath as bm
```

```
import numpy as np

x = np.array([1,2,3,4], dtype=np.float32)
y = bm.add_one(x)

print 'x: ', x
print 'y: ', y
```

Output: (run 'python test.py')

```
x: [1. 2. 3. 4.]
y: [2. 3. 4. 5.]
```

This example can be found in the PyFI library directory:

```
/opt/gpi/lib/core/PyFI
```

4.5 Embedding Python (PyCallable)

PyFI also includes a class called 'PyCallable' that simplifies the process of embedding Python. For the purposes of GPI, this allows the PyMOD developer to use Python libraries for functionality that is not yet available as a C++ solution (whether its not available as a library or it is not interfaced with PyFI arrays).

PyFI arrays that are sent to Python via PyCallable are wrapped by Numpy arrays so that the data are accessed directly by the interpreter. The PyCallable interface is threadsafe, however, it will block when executing internal Python calls. The PyCallable class is available in the PyFI namespace. The PyCallable object can be constructed in two ways:

1) Module & function:

```
/* use the numpy isnan() function */
PyCallable("numpy", "isnan");

or

/* use a python script that is loadable from the python path */
PyCallable("myScript", "myFunc");
```

2) Python code from std::string:

```
std::string myCode = "def func(x, y):\n"
                     "    print x, y\n";
PyCallable(code);
```

In the second case, the function defined in the inline code must define a function called ‘func’. This is what PyCallable looks for in the imported python code. ‘func’ may pass and return any number of arguments.

The PyCallable interface is used to wrap the Numpy implementation of the pseudo inverse ‘pinv()’ and the fft interface for 1D ffts. These examples can be found in:

```
/opt/gpi/lib/core/PyFI/PyFIArray_WrappedNUMPY.cpp
```

Other simple examples can be found in the template_PyMOD.cpp.

The PyCallable operation is similar to the PyFunction interface in that function arguments are parsed in the order in which they are given, in python its left to right, in PyFI its top to bottom. Regardless of how it is constructed, arguments are passed and returned to and from the Python function by the method functions. The passing functions are:

PyCallable::SetArg_Array(*ptr****)***
‘ptr’ is a pointer to a PyFI::Array<T> object.

PyCallable::SetArg_String(*string****)***
Takes a std::string.

PyCallable::SetArg_Long(*long****)***
Takes a long integer (i.e. *int64_t*)

PyCallable::SetArg_Double(*double****)***
Takes a double precision float.

The return functions are:

PyCallable::GetReturn_Array(*ptr_ptr****)***
‘ptr_ptr’ is a reference to a pointer to a PyFI::Array<T> object. This modifies the input pointer given. This is a templated function.

PyCallable::GetReturn_String()
Returns a std::string.

PyCallable::GetReturn_Long()
Returns a long (*int64_t)*.

PyCallable::GetReturn_Double()
Returns a double.

Once all the arguments are set, the ***Run()*** method can be called. If any of the GetReturn_ functions are called, then ***Run()*** is automatically invoked for the first GetReturn_.

NOTE: PyCallable() currently doesn't handle exceptions. This means the executed code cannot contain try-except clauses.

4.6 Converting from Philips Recon. Arrays to PyFI Arrays

The PyFI Array class is meant to be a distributable interface that closely resembles the Philips Reconstruction Platform (PRP) arrays. This allows PyFI module developers to develop code that can be easily converted to the PRP. The two Array classes are similar in most respects; the differences are as follows:

1) PyFI doesn't support negative bounds indexing and a a consequence there is not an ArrayBounds object nor does the Array class have a bounds() method. An analogous object would be the PyFI::ArrayDimensions object which holds an Array's number of dimensions 'ndim' and dimensions array 'dimensions'. The PyFI array class has a dims_object() method which will return an object with the array's dimension information.

   NOTE: PyFI arrays can be constructed using a std::vector<uint64_t> object from the standard C++ library. This is the preferred method since the vector class manages its own memory and provides a clean and concise mechanism for constructing array parameters. A dimensions vector can be obtained with the method 'dimensions_vector()'.

2) Inequality operations (==, >=, <=, !=, >, <), in PyFI, return a bit-mask array that represents the element-wise comparison of two arrays or an array and a constant.

3) Basic array math operations are part of the array class (so they are also templated). This means that each array has the methods min(), max(), mean(), and stddev().

4) There is no resize() method. If you want a centered (zeropadded or cropped) copy of an array you can use the get_resized() method with an isotropic scale argument (double), a std::vector<double> array of dimension scales, an isotropic dimension length argument (uint64_t), or a std::vector<uint64_t> array of dimension lengths.

5) Arrays can be recast to common types by using the methods as_ULONG(), as_FLOAT(), as_CFLOAT(), as_DOUBLE(), as_CDOUBLE(), as_LONG(), as_INT(), and as_UCHAR() in place of recast functions.

The PyFI interface to the PRP array library still exists and can be used by excluding the PyFI namespace (i.e. don't use the line "using namespace PyFI;") and using the '--R2' flag when compiling with gpi_make. The gpi_make can be configured to point to the location of the intel libraries and PRP libraries necessary for compilation in the ~/.gpirc file under the [MAKE] section.