Created: June 4-5, 2009
Modified: June 11-12, 2009
Modified: June 18-19, 2009
Modified: July 2, 2010 (Matrix multiplication)
Last update: July 8, 2010

S.D. Peckham

The I2PY 0.2.0 User's Guide

Introduction

I2PY version 0.1.0 was written in 2005 by Christopher J. Stawarz <chris@pseudogreen.org>. The I2PY project website can be found at: http://code.google.com/p/i2py/.
It can convert most of the low-level syntax and reserved words of the IDL language, including array subscripting, assignments, expressions, all types of loops, conditionals (if-then-else) and so on to Python. I2PY 0.1.0 represents the grammar of the IDL language as an "abstract syntax tree" or AST. The grammar specification is given as "productions" and "precedence rules" at the beginning of a file called "parser.py". I2PY uses PLY (Python Lex-Yacc), a Python port of two well-known parsing tools called "lex" and "yacc". (Note that "lex" is a "lexical analyzer" and "yacc" is a "parser generator".) More information on PLY can be found at: http://www.dabeaz.com/ply/. A function called "build_productions" in "parser.py" creates the functions that "yacc" needs to generate an IDL language parser. It then calls the "yacc" function defined in "yacc.py".

NOTE: If one were to provide a similar grammar specification for MatLab as an AST, then much of the machinery of I2PY could be used to create a tool (perhaps called M2PY?) for converting MatLab code to Python code.

I2PY 0.1.0 provides an API consisting mainly of 3 functions, "map_var()", "map_func()" and "map_pro()" that can be used to specify how either "built-in" or "user-written" IDL procedures and functions should be converted. These functions are defined in the file "map.py", and arguments are explained in the definition of the SubroutineMapping class. Unfortunately, however, version 0.1.0 can only convert a very small number of IDL's built-in procedures and functions (9 of them, with the API calls in "maplib.py"), so most IDL programs converted to Python with this version will not run as Python programs. Presumably, this is why it was given a low, alpha-level version number of 0.1.0.

NumPy or Numerical Python is is a Python package that provides Python with a fairly complete set of "array operators" similar to those in IDL or Matlab. It represents the culmination of similar but now obsolete packages called "Numeric" and "NumArray". Matplotlib is another Python package that provides a set of plotting functions with syntax similar to that of MatLab. IDL, MatLab and Python-with-NumPy are all "array-based", interpreted programming languages, also known as "very high level languages" or VHLLs. Each of them compiles to bytecode instead of machine code which gives them a very high

degree of portability across platforms In these languages, the "array operators" are actually programs that are written in low-level C "under the hood". This is why code written in IDL can run nearly as fast as C code, as long as the programmer consistently uses the array operators and avoids all "spatial loops" (e.g. over 2D array indices). In particular, models written in IDL that consist of a time loop wrapped around spatial calculations performed with array operators can run quite fast because the cost of the time loop is easy to amortize if the spatial arrays are not too small. There are a few additional IDL "array operators", such as CONVOL, that do not have a counterpart in NumPy itself, but that do have a counterpart in the SciPy (Scientific Python) package. It takes many separate Python packages to reproduce all of the functionality of IDL, but most if not all of them are free and open-source. The most important packages in this regard are NumPy, matplotlib (actually matplotlib.pyplot), wxPython (to create GUIs) and SciPy. Many other Python packages such as os, sys, time, glob, webbrowser, re and shutil are also needed but are included with a standard Python installation. Some sub-packages of NumPy are also needed, such as numpy.linalg and numpy.random. They are imported as needed. Additional Python packages (e.g. PIL) are needed for reading and writing to various file formats (e.g. BMP, netCDF and ESRI shapefiles) and for specialized tasks such as map projections. However, many such packages are available, partly because ESRI has adopted Python as the new scripting language for their GIS programs.

This document describes I2PY 0.2.0, a significant extension of I2PY 0.1.0 written by Scott D. Peckham. Part of this extension has to do with using the API to provide translations for a large number of IDL's built-in procedures and functions (126 of them so far), including most of their associated arguments and keywords. (See Appendix A for a list.) This adds about 90 pages of Python code to I2PY 0.1.0, in four new files called "idl_maps.py", "callfuncs.py", "callfunc_utils.py" and "idl_func.py". However, this version also contains numerous enhancements to the low-level language conversion (mainly in "ir.py" and "map.py"), such as added support for:

(1) explicitly-typed scalars
(2) structures
(3) pointers
(4) handling of IDL variables whose name is a reserved word in Python, and
(5) generation of more "Pythonic" code whenever possible.

The goal was to get I2PY to the point where it could be used to convert typical earth-science models written in IDL to functioning Python code, even if that code wasn't always pretty or "Pythonic". For example, IDL uses an EOF (End of File) function when reading data from files, and I2PY 0.2.0 defines a similar EOF function in "idl_func.py" that is used for translation. However, this is not the "Pythonic" way, and Python does not have an EOF function. When reading from a text file, the "read()" and "readline()" methods of the built-in "file" object return an empty string. So you would use something like this to break out of a while loop: "if (len(data) == 0): break" or use an iterator as in: "for line in file:". (Note: When reading from a binary file with "numpy.fromfile" a MemoryError is returned if you attempt to read past EOF. The "fread()" method in the "scipy.io.numpyio" package returns an empty array (i.e. len(array) == 0).)

During the course of this project it was also found that some IDL routines (i.e. built-in procedures and functions) have "complex" behavior that cannot be reproduced with just a few lines of Python/NumPy code. For example, they may:

(1) test the types of the arguments and then branch (e.g. BYTE) (That is, they are "overloaded" in the object-oriented sense and return different results based on the data types of the arguments.)

(2) return a structure (e.g. FSTAT, SIZE)

(3) be procedures that modify their arguments (e.g. READF, READS)

(4) have keywords that require many lines of code to handle properly (e.g. FORMAT keyword in STRING, PRINTF, READF, READS)

For commands like these, it turns out to be cleaner and easier to provide a function written in Python that has the same name and works the same as the corresponding IDL command. These functions can be found in a file called "idl_func.py" that can be copied into your project folder. I2PY will then translate these IDL commands as calls to same-named functions in "idl_func.py". Note that this approach to conversion was only used in a handful of cases.

How to Run I2PY 0.2.0

(1) Add "map_func()" or "map_pro()" calls in "user_lib.py" for all user-written IDL procedures and functions, in accordance with the information given in the previous section. If you don't do this, many of your routines may still work but I2PY will create a function called "_ret()" and embed it within the translated routine in order to process optional input and output parameters. (And this extra code isn't pretty.)

(2) Copy the file "idl_func.py" into the same folder as your IDL source code. The Python code generated by I2PY will be written to this same folder.

(3) Change to the "i2py-0.2.0" directory that contains the executable script called: "idl2python". (Or make sure it can be found in your path.) Note: On Windows computers the file you'll need will be called: "idl2python.py".

(4) Run I2PY by typing the following line at a Unix prompt:

```
% ./idl2python <path-to-your-idl-file>.pro
```

This will create a file with the same path/prefix and the extension ".py".

(5) Read the section on "Known Problems" below and go back through the new Python code to look for potential problems.

(6) If your IDL code is contained in multiple files, then you will most likely need to add appropriate "import" statements at the top of the new Python code files.

Optional Steps:

(1) Note that I2PY removes all of the line continuation characters, as indicated by the "$" symbol, from the input IDL code. In order to improve the readability of the resulting Python code you may want to scan the code for long lines and then insert backslashes (Python's line continuation character) at strategic locations.

(2) Search the Python code generated by I2PY for the string "EOF(" and edit the nearby code to use the Pythonic approach to reading data from files, which does not use an EOF (end-of-file) function.

(3) See the section below called "Translation of Boolean Keywords" and use the tip given there to further clean up your new Python code.

(4) Comment blocks that occur immediately after the definition of an IDL function or procedure do not get indented even though the code that follows does get indented. In the Python shell called IDLE, you can open your new '.py" file, highlight the comment block and then choose Format > Indent Region to change the indentation of comment blocks.

NOTE: There is a test file called "i2py_test.pro" included with I2PY 0.2.0 that you can use to see how it will translate a wide variety of different IDL commands. It is located in i2py-0.2.0/Tests.

NOTE: Python is case-sensitive, but IDL is not. I2PY 0.2.0 preserves the original case for many things, such as variable names. This is a good thing if you are consistent with the case(s) you use for your variables, but can be a problem if your code uses "a" and "A" for the same variable. This behavior can be toggled with minor changes to the pyname() function in the file "util.py" in the "i2py" folder. Note that the setting in "config.py" has been disabled (I think). ******

NOTE: When you try to run your converted Python code, it is possible that you will encounter the error message: "EOL while scanning single-quoted string." This typically occurs when you are defining a directory name as a quoted string for the Windows platform, e.g. "a = 'C:\'". Python sees the right-most backslash as its line continuation character and gets confused. The solution is to add another backslash character beside the first one, as in "a = C:\\".

NOTE: "Token Error: EOF in multi-line statement." (explain)

NOTE: "TypeError: Sequence item 0: expected string, NoneType found." (explain)

Testing the I2PY Translation Routines

There is a folder called "Tests" in the "i2py-0.2.0" folder. The file called "i2py_test.pro" contains a variety of different IDL constructs for testing and the file "i2py_test.py" shows how they were translated by I2PY 0.2.0.

Translation of "Built-In" IDL Procedures and Functions

I2PY 0.2.0 can translate many of the "built-in" procedures and functions that are part of the IDL language. Most of the code for doing this is located in a file calle "idl_maps.py" in the "i2py" folder. I2PY can be extended or customized by editing th code in that file. Several utility or "convenience" functions are included at the to of the file "idl_maps.py" to further simplify this job. A list of "built-in" IDL routine that I2PY 0.2.0 can translate are given in a separate section below.

Translation of IDL System Variables

IDL has a number of "system variables" that are used for built-in numerical value such as pi. I2PY 0.2.0 can now translate the following IDL system variables:

```
!DPI, !PI, !DTOR, !RADEG,
!VALUES.F_INF, !VALUES.F_NAN, !VALUES.D_INF, !VALUES.D_NAN,
!VERSION.OS, !VERSION.OS_FAMILY, !VERSION.ARCH,
!D.NAME
```

The ones that do not contain a dot in their name are translated with simple calls to "map_var()" in "maplib.py". Because of the dot, the others get identified by I2PY as a PostfixExpression (similar to referencing fields of a structure) and cannot be translated this way. However, in I2PY 0.2.0, the "pycode" method of the PostfixExpression class in "ir.py" has been modified to catch and translate IDL system variables with dots in their names.

Translation of User-Written IDL Procedures and Functions

I2PY can translate most of the "built-in" procedures and functions that are part of the IDL language. However, it cannot translate many user-written IDL procedures and functions properly unless it is provided with additional information about their arguments. This includes:

(1) Procedures (not functions) that return values. (Note that arguments to an IDL procedure may provide input values or return values.)

(2) Procedures or functions with optional arguments (that aren't keywords).

(3) Procedures or functions with keywords.

In each of these cases I2PY will issue an error message about missing arguments. In order to address this problem, I2PY version 0.2 allows additional information for user-written IDL procedures and functions to be provided in a file called: "user_lib.py". This file should be placed in the "i2py" folder and you should be careful not to have any other file in your PYTHONPATH with the same name. It is imported by "idl_maps.py" to assist with the translation process. The file "user_lib.py" contains a series of calls to either I2PY's "map_func()" or "map_pro()" function. Both "map_func()" and "map_pro()" take the procedure or function name (as a string) as their first argument. Both also accept the following keywords:

inpars = a list of integer-positions of args that are input parameters
(the first argument has an index of 1, not 0)
e.g. "inpars=[1,3]" or "inpars=range(1,5)"
inkeys = a list of strings (with capital letters) that are IDL keywords
that provide input
noptional = an integer (possibly zero) that indicates how many of the
arguments, starting from the last one, that are optional.
callfunc =
extracode =
readonly =

The "map_pro()" function also accepts two additional keywords, but "map_func()" does not:

outpars = a list of integer positions of args that are output parameters
outkeys = a list of strings (with capital letters) that are IDL keywords
that provide output (return values)

Note that "list" here refers to a Python list, comma-separated and enclosed with square brackets. It may also be an empty list.

Translation of IDL's WHERE Function

IDL's WHERE function presents a number of different challenges with regard to translation to Python/NumPy. These can be summarized as follows:

(1) IDL's WHERE function always returns a 1D array of array subscripts, which correspond to the long-integer indices you get if you number every element in the array in "calendar fashion", starting from the upper-left corner.

NumPy's WHERE function returns a tuple of 1D arrays, with one array for each dimension of its argument array. Consider this example:

```
>>> z = arange(9)
>>> z = z.reshape(3,3)
```

```
>>> print z
>>> w = where(z < 5)
>>> print w
        ( array([0,0,0,1,1]), array([0,1,2,0,1]) )
```

The first array in the 2-tuple contains the row indices and the second array contains the column indices for the elements that have "z < 5". The row and column of the z=3 element can be retrieved as "w[0][3]" and "w[1][3]".

In view of these facts, I2PY 0.1.0 would translate the IDL command:

```
w = where(z lt 5)
```

to

```
w = where(ravel(z < 5))[0]
```

Unfortunately, this translation will only work if z is a 1D array, because if you subscript a 2D or 3D NumPy array with a single integer, it is not interpreted as a "calendar index" but instead returns the corresponding row of values. Removing the "ravel" from the translated code solves this problem but unfortunately then leads to another problem. See Item 4 below.

(2) IDL's WHERE function returns "-1L" when there are no elements in the array argument for which the test is true.

NumPy's WHERE function returns a tuple of empty 1D arrays in this case.

(3) IDL's WHERE function has an optional second argument called COUNT and two optional keywords called COMPLEMENT and NCOMPLEMENT that return values. So, for example, IDL code could contain the following call to WHERE:

```
w1 = where(array lt 0, n1, COMPLEMENT=w2, NCOMPLEMENT=n2)
```

I2PY 0.2.0 translates this as the following 4 Python/NumPy commands:

```
w1 = where(array < 0)
n1 = size(w1)
w2 = where(invert(array < 0))
n2 = size(w2)
```

Note that the NumPy code is more costly because it requires 2 calls to the WHERE function. Notice that NumPy's "invert()" is used to get the complement set.

(4) "Nested" calls to IDL's WHERE function pose an additional challenge. This construct can make IDL code more efficient in the case where the result of some test determines whether we need to "pick through" and refine the results of a previous search. If the previous search resulted in an array that is much smaller than the original array, it is more efficient to apply a refined search to just these indices rather than perform another search of the original array. The following IDL code illustrates this basic idea, even though in this case it would be better to combine the two tests into one WHERE call.

```
a  = indgen(4,4) - 8
w1 = where(a lt 0, n1)
if (n1 ne 0) then begin
   w2 = where(a[w1] gt -3, n2)
   if (n2 ne 0) then begin
         b = a[w1[w2]]
         a[w1[w2]] = 99
   endif
endif
```

After this code runs we will have w1 = [0, 1, 2, 3, 4] and w2 = [3, 4]. IDL's WHERE always returns a 1D array, and IDL arrays of any dimension can be indexed with a 1D array, where 1D array contains "calendar style", long-integer indices. So this kind of "nested indexing" works for IDL arrays of any dimension. The IDL code above can be translated to Python-NumPy as follows:

```
a = arange(16).reshape(4,4) - 8
w1 = where(a < 0)
n1 = size(w1)
if (n1 != 0):
   w2 = where(a[w1] > -3)
   n2 = size(w2)
   if (n2 != 0):
         b = a[w1][w2]
         a[w1][w2] = 99    # (this doesn't do what we want)
```

After this code runs we will have:

```
w1 = (array([0, 0, 0, 0, 1, 1, 1, 1]), array([0, 1, 2, 3, 0, 1, 2, 3]))
```

and

```
w2 = (array([6, 7]),)
```

Notice that the indices in w2 now refer to 1D indices in the original array, "a", instead of 1D indices to a[w1] as was the case in IDL. The Python code snippet above runs without an error, but the very last line does not produce the intended result; the array "a" is not modified. Notice also that "b = a[w1][w2]" works but "b = a[w2]" would produce an error

because w2 is a 1D array. In NumPy, if we want to index a non-1D array with a 1D "calendar-index" array, then we have to do something extra. One option is to use an iterator object that NumPy arrays have "built-in" called "flat", as in: "b = a.flat[w2]" and "a.flat[w2] = 99". For an assignment, we can also use NumPy's "put()" function, as in: "put(a, w2, 99)". So to get the desired result in the code snippet above, we can replace the last line with any of these 3 options:

```
put(a, w2, 99)
a.flat[w2] = 99
a[ (w1[0][w2], w1[1][w2]) ] = 99
```

(5) On my MacPro running OS 10.5, with a standard install of Python and NumPy, NumPy's WHERE function is about 4.5 times slower than IDL's WHERE function. I don't know why this should be the case.

Translation of Array Creation and Concatenation via Square Brackets

Square brackets in IDL are "overloaded" in the sense that their meaning depends on the context. On the one hand, they are used to create arrays as in these examples:

```
IDL> a = [1,2,3]
IDL> a = [[1,2], [3,4]]
IDL> a = ['this', 'and', 'that']
```

However, square brackets are also used for array concatenation as in these 2 examples:

```
IDL> a = [1,2,3]
IDL> b = [4,5,6]
IDL> c = [a,b]

IDL> a = [1,2,3]
IDL> d = 0
IDL> c = [d, a]
```

NumPy uses "numpy.array()" for creating arrays and "numpy.concatenate()" for array concatenation. I2PY 0.2.0 takes great pains (in "ir.py") to determine when it should use one vs. the other. It also recognizes when an array is created from explicitly-typed scalars and simplifies the resulting code. For example, "a = [0b, 1b]" gets translated to: "a = uint8([1,2])"; notice that "array()" is not needed in this case for "a" to become a NumPy ndarray.

Whenever an IDL expression translated by I2PY 0.2.0 contains NumPy's "concatenate()", it issues the following warning:

> "Warning: Make sure all args to concatenate() are arrays."

This is because:
(1) if some of the arguments are named variables, it cannot tell if they are arrays or scalars, and
(2) "numpy.concatenate()" will issue the following error message:

"ValueError: arrays must have same number of dimensions"

if its arguments include both arrays and scalars, and

"ValueError: 0-d arrays can't be concatenated"

if all of its arguments are scalars.

All of these test cases (included in "Tests/i2py_test.pro") get translated correctly, but there may still be cases that won't.

```
a = [1,2,3, 4]
a = [0b, 1b, 2b,3B]
a = [0s, 1S, 2s,3S]
a = [0L, 1L, 2L,3L]
a = [0d, 1d, 2D,3d]
a = [0LL, 1ll, 2LL]
a = [0.0, 1, 2.]
a = ['x','y','z']
a = ["x", "y", "z"]
a = ['x', "y"]
msg = ['Error message.']
a = [a, 0]
a = [0, a]
a = [[1,2,3], 0]
a = [0, [1,2,3]]
a = [[1,2], 0, [3,4]]
a = [[1,2,3], b]
a = [b, [1,2,3]]
a = [[1,2,3], [4,5,6]]
a = [1,2,3]  &  b = [4,5,6]  &  c = [a,b]
```

Translation of Pointers

Python does not have or need pointers, because all variable names are references to the stored values. This makes them behave in essentially the same way as pointers. Because of this, the asterices that would indicate a pointer variable can simply be removed. In some cases it is also desirable to remove an unneeded set of parentheses. I2PY version 0.1.0 was already able to identify pointers appearing in IDL code (as distinct from a multiplication,

for example) using the PointerExpression class in the file "ir.py". I2PY version 0.2.0 now removes the asterix and extra parentheses and the resulting code should work as expected. (Could use more testing.)

Translation of Structures

I2PY version 0.1.0 was already able to identify structures appearing in IDL code using the PrimaryExpression class in the file "ir.py". In particular, primary expressions starting with a left brace (curly bracket) are identified as IDL structure definitions. These definitions are translated by I2PY 0.2.0 as a call to a special function called "bunch" in the file "idl_func.py", along with necessary changes to the syntax. Changes to the "pycode" method of the PrimaryExpression class also cause "inner expressions" -- used to assign values to the fields of the structure -- to be translated as well. IDL code that refers to the fields of a structure by tag index instead of field name are now translated as well. For example, "y = x.(k)" translates to "y = x.__dict__.values()[k]".

Translation of Explicitly-Typed Variables

Even though IDL and Python are dynamically-typed languages, both are what is often called "strongly typed". Advanced IDL programmers will often assign an explicit type to a variable when it is first created and will take advantage of the "upward conversion" (or upcasting or data type precedence) rules for IDL expressions. Being sure that a variable has a given type is also important when writing values to binary files. For scalars, explicit typing is often indicated by adding a letter after the assigned numerical value, as is also done in C.

For example, scalar values of zero with IDL data types of BYTE, INTEGER, LONG, LONG64, FLOAT and DOUBLE would be given as: 0b, 0, 0L, 0LL, 0.0 and 0d. There are benefits to having scalar variables treated as NumPy objects in expressions that involve both scalars and arrays, such as the ability to query the "dtype". Because of this, I2PY 0.2.0 translates scalars that are explicitly typed using letter code suffixes as follows:

```
nD   -> numpy.float64(n)    (D is for "double")
nS   -> numpy.int16(n)      (S is for "short integer")
nL   -> numpy.int32(n)      (L is for "long integer")
nLL  -> numpy.int64(n)
nB   -> numpy.uint8(n)
nU   -> numpy.uint16(n)
nUS  -> numpy.uint16(n)
nUL  -> numpy.uint32(n)
nULL -> numpy.uint64(n)
```

In addition, if a number contains a decimal point or "E" (scientific notation), then it is translated as "numpy.float32(n)". If a number contains "X" then it is interpreted and translated as hexadecimal.

However, wrapping every 16-bit integer with "int16()" creates a lot of clutter, for example when integer values are assigned to keywords. To prevent this clutter, I2PY 0.2.0 examines the right-hand side of expressions via changes to the "pycode" method of the AssignmentStatement class in the file "ir.py".

Explicitly-typed IDL arrays, such as those created with the BYTARR or LONARR function are translated using "numpy.zeros()" or "numpy.empty()" and using the "dtype" keyword.

Translation of I/O Commands

When reading and writing to files in IDL, one first uses the OPENR, OPENW, or OPENU procedure to open a file for reading, writing or updating (both). A unit number is either provided as the first argument or the GET_LUN keyword can be set, in which case the first argument is passed as an undefined variable that becomes set to the first available unit number. GET_LUN can also be used as a stand-alone procedure. (LUN = Logical Unit Number) This unit number is subsequently used as a means of referring to that file for all subsequent I/O operations. The READF and PRINTF procedures are used to read and write to text files (the F is for Formatted). Similarly, the READU and WRITEU procedures are used to read and write to binary files (the U is for Unformatted). When I/O is complete, files are closed with the CLOSE or FREE_LUN procedure. For example,

```
IDL> openr, unit, 'my_file.txt', /get_lun
IDL> line=''
IDL> readf, unit, line
IDL> free_lun, unit
```

The FREE_LUN procedure frees the unit number obtained via GET_LUN; otherwise we could simply use "close, unit".

Python uses an object-oriented approach to file I/O, where either the FILE (or equivalent OPEN) command is used to create a "file object" with methods that are then used for subsequent I/O operations. The equivalent Python code for the preceding example is:

```
>>> file_unit = file('my_file.txt', 'r')
>>> line = file_unit.readline()
>>> file_unit.close()
```

The second argument to Python's "file" function indicates the mode of access; this file is open for reading a text file. It would be 'rb' for a binary file. Here we have chosen "file_unit" as the name of the file object. Notice that the first argument to an IDL I/O procedure is always the unit number. I2PY 0.2.0 therefore translates IDL's I/O commands by taking the

first argument, as a string, and then prepending "file_" to it to get a locally unique name for Python's file object. Note that this works even if the unit number was provided as an actual number instead of a variable name.

Note that IDL's "OPEN" procedures do not specify the mode of access. So it is not clear until we perform an I/O operation whether the file is text or binary. As a result, I2PY 0.2.0 is unsure about the mode of access and currently assumes that the file is a text file unless the SWAP_ENDIAN keyword to the "OPEN" procedure has been set. I2PY 0.2.0 also introduces a variable into the translated code called I2PY_SWAP_ENDIAN, set to either True or False, and subsequently uses this variable to decide whether byteswapping is necessary when using "numpy.fromfile()" or the "tofile()" method of NumPy data objects. (Note that this includes scalars since they become "NumPy" data objects as a result of I2PY's explicit typing of scalars created via assignments.)

Translation of Plotting Commands

IDL has a number of "direct graphics" plotting commands such as CONTOUR, PLOT and SURFACE. Each accepts numerous optional keywords for setting various attributes of the plot such as the x-axis title or limits. There is a Python package called "matplotlib" that provides much of the same functionality, except there is a separate command for each of the plot attributes instead of a keyword argument. I2PY 0.2.0 can properly translate most IDL plotting commands along with their optional keyword arguments. However, the translated code will not work unless the "matplotlib" package has been installed correctly.

Translation of Commands for Creating GUIs

IDL also allows users to write platform-independent code for GUIs. The wxPython package (as well as PyQT) provide Python with a similar capability except the syntax is very different. I2PY does not yet convert IDL commands for GUIs (which usually have names starting with "WIDGET_") to wxPython.

Translation of Variable Names that are Reserved Words in Python/NumPy

There are a number of "reserved words" in Python and/or NumPy that may easily appear as variable names in IDL code. Some examples include:

> as, chr, del, dir, file, from, global, in, is, lambda, len, list,
> local, map, ord, pass, pow, str, type and yield.

In order to prevent conflicts, I2PY 0.2 adds a leading underscore to any IDL variable name that is in this list.

I2PY has a "map_var()" function for indicating how variable names should be translated. I2PY 0.2 tries to use the variable names from the IDL code "as is" whenever possible, including preservation of upper and lower case. These mappings can be viewed and edited in the file "maplib.py" in the "i2py" folder.

Note that some IDL system variables can also be mapped using "map_var()", such as "!DPI", "!DTOR", "!PI", "!RADEG" and "!PATH". However, IDL system variables that include a dot in their name cannot be translated this way. Examples include "!VALUES.F_INF" and "!VALUES.D_NAN". In order to properly translate IDL system variables with dots, code must be added in the "PostfixExpression" section in the file "ir.py".

Translation of Boolean Keywords

It is common in IDL to use boolean keywords as "flags" or "toggles" that turn a given feature on or off. For example, an IDL program could have boolean keywords such as SILENT, VERBOSE, REPORT or PLOT that would by default be unset, or off. These keywords are often introduced in the definition of an IDL procedure or function as in this example:

```
function My_Function, arg1, arg2, OPTION1=OPTION1, REPORT=REPORT
```

Notice that the keyword name is repeated before and after an equals sign. IDL keywords have the advantage that the order in which they are provided or set doesn't matter and they are optional. When calling an IDL function or procedure, there are several different ways to set a boolean keyword, such as:

```
result = My_Function(arg1, arg2, /REPORT)
result = My_Function(arg1, arg2, REPORT=1)
result = My_Function(arg1, arg2, REPORT=1b)
result = My_Function(arg1, arg2, REPORT=Do_Report())
```

The last example sets the keyword to the return value of a function called "Do_Report". I2PY 0.2.0 currently translates these four examples as follows:

```
result = My_Function(arg1, arg2, REPORT=True)
result = My_Function(arg1, arg2, REPORT=1)
result = My_Function(arg1, arg2, REPORT=uint8(1))
result = My_Function(arg1, arg2, REPORT=Do_Report())
```

Within an IDL routine such as My_Function, you would typically process the keyword in one of two ways. One way would be with a line such as

```
"REPORT = keyword_set(REPORT)".
```

This mechanism assigns a boolean value to REPORT even if it wasn't set by the caller. This allows for something called "keyword inheritance" where the same keyword can be passed

on with the same setting to any function that is called by My_Function. It also allows there to be a block in My_Function something like:

```
if (REPORT) then begin
   print, 'This is report line 1.'
   print, 'This is report line 2.'
endif
```

The second typical way to process a boolean keyword in IDL is to use a statement like this:

```
if not(keyword_set(REPORT)) then REPORT = 0b
```

I2PY 0.2.0 translates IDL's KEYWORD_SET function as:

```
"REPORT not in [0,None]"
```

in order to accomodate all of these cases and should now produce working code in each case. Note that the line above would then translate to:

```
if logical_not(REPORT not in [0,None]):
```

but I2PY 0.2.0 (in "ir.py") reduces this to the simpler and equivalent:

```
if (REPORT in [0,None]):
```

Notice that boolean expressions evaluate to "True" or "False" in Python, as opposed to 1 and 0, as in IDL. However, if 1 or 0 is encountered in a boolean test in Python, they are evaluated as True and False. See the next section for more details on this issue.

If you'd like to clean up your Python code after translation you can change "REPORT=REPORT" in the Python definition of My_Function to "REPORT=False". You can then delete lines like these from your code:

```
REPORT = keyword_set(REPORT)
if not(keyword_set(REPORT)) then REPORT = 0b
```

This is because unlike IDL keywords, Python keywords can be provided with a default value when a function is defined.

Translation of Bitwise and Logical OR, AND and NOT

In ir.py, the classes "LogicalExpression" and "BitwiseExpression" determine how IDL's OR, AND and NOT are converted. Recall that "logical and/or/not" evaluates all args and returns 0 or 1, but "bitwise and/or/not" does an AND operation on integers and returns one of the 2 args for other types.

Most of the time, translating IDL's "and/or/not" to NumPy's "logical_and/or/not" will give the desired result. However, the latter functions are NumPy "ufuncs". While they can also be passed scalar arguments, using them in tests that require a scalar boolean result, such as IF-THEN-ELSE statements and the start of WHILE loops will produce an error. I2PY 0.2.0 attempts to check for and correct this. Instead of returning array elements of 1 or 0, they return array elements of True or False. However, Python will evaluate 1 and 0 in tests as True and False.

In I2PY 0.2.0 the BitwiseExpression class in ir.py has been changed accordingly. However, in some cases it might be better for users to search and replace occurrences of:

"and", "or" and "not"

in their IDL code with:

"&&", "||" and "~"

and revert back to the previous code for BitwiseExpression in "ir.py".

In IDL:
(1) The 4 BITWISE operators are: AND, NOT, OR and XOR.

(2) The 3 LOGICAL operators are: "&&", "||" and "~".

(3) When dealing with logical operators, non-zero numerical values, non-null strings and non-null heap variables (pointers and object references) are considered TRUE; all else is FALSE.

(4)
```
false = 0b
true  = 1b
print, not(false)      (255)
print, not(true)       (254)
print, ~false          (1)
print, ~true           (0)
```

In Python:
(1) The and/or/not operators cannot operate on arrays, so for arrays we must use NumPy ufuncs called "logical_and/or/not" or "bitwise_and/or/not".

(2)
```
false = numpy.uint8(0)
true  = numpy.uint8(1)
not(false)                    (True)
not(true)                     (False)
numpy.logical_not(false)      (True)
numpy.logical_not(true)       (False)
```

```
numpy.bitwise_not(false)        (255)
numpy.bitwise_not(true)         (254)
numpy.logical_or(false, true)    (True)
numpy.logical_and(false, true)   (False)
numpy.bitwise_or(false, true)    (1)
numpy.bitwise_and(false, true)   (0)
a = numpy.array([0,1,0])
b = numpy.array([1,1,0])
a and b                    (error)
a or  b                   (error)
numpy.logical_or(a,b)          [True, True, False]
numpy.logical_and(a,b)         [False, True, False]
numpy.logical_not(a,b)         [1, 0, 1]
numpy.bitwise_or(a,b)          [1, 1, 1]
numpy.bitwise_and(a,b)         [0, 0, 0]
numpy.bitwise_not(a,b)         [-1, -2, -1]
```

Translation of STRARR with Initial Value

In IDL, it is possible to initialize a string array with a given scalar string, as in this example:

```
IDL> s = strarr(5) + 'this'
```

While NumPy supports string arrays, they cannot be added to a scalar string. So while this is okay:

```
>>> s = zeros([5], dtype='|S100')
```

the following line results in an error:

```
>>> s = zeros([5], dtype='|S100') + 'this'
```

However, the following two commands produce the desired result:

```
>>> s = zeros([5], dtype='|S100')
>>> s.fill('this')
```

However, I2PY 0.2.0 can only handle this case in the context of an assignment statement. This required changes to the "pycode()" method of a class called "AssignmentStatement" that is defined in "ir.py".

Translation of Matrix Multiplication Operators

From the IDL docs on the # operator:

"Computes array elements by multiplying the columns of the first array by the rows of the second array. The second array must have the same number of columns as the first array has rows. The resulting array has the same number of columns as the first array and the same number of rows as the second array."

Example 1 (IDL):

a = [[1,2,1], [2,-1,2]]
b = [[1,3], [0,1], [1,1]]
a # b = [[7, -1, 7], [2, -1, 2], [3, 1, 3]

From the IDL docs on the ## operator:

"Computes array elements by multiplying the rows of the first array by the columns of the second array. The second array must have the same number of rows as the first array has columns. The resulting array has the same number of rows as the first array and the same number of columns as the second array."

Example 2 (IDL):

a ## b = [[2, 6], [4, 7]] (same a and b as last example)

The "numeric" Python package that preceded "numpy" had a "matrixmultiply" function which has since been deprecated in favor of an improved "dot" function. The above examples look like this in numpy:

Example 1 (numpy):

a = array([[1,2,1], [2,-1,2]])
b = array([[1,3], [0,1], [1,1]])
numpy.dot(b, a) = [[7, -1, 7], [2, -1, 2], [3, 1, 3]

Example 2 (numpy):

numpy.dot(a, b) = [[2, 6], [4, 7]]

I2PY 0.1 translated the # and ## operators in the MultiplicativeExpression class in the file "ir.py" as follows:

a # b => transpose(matrixmultiply(transpose(a), transpose(b)))
a ## b => matrixmultiply(a, b)

In I2PY 0.2, they are now translated as:

a # b => dot(b, a)
a ## b => dot(a, b)

However, if a and b are 1D arrays (i.e. a.shape = (n,) instead of (n,1), then the # operator in IDL computes an outer product. This twist has not been implemented in I2PY 0.2.0 yet.

How Does I2PY Work?

The folder named "i2py" contains Python code in the following files, most of which were mentioned in previous sections. Most of the code required for translating IDL's built-in procedures and functions is contained in the files "idl_maps.py" and "idl_func.py". The code for translating IDL statements, including assignment statements, various types of loops and if-then-else constructs can be found in the file "ir.py".

These are from I2PY 0.1.0.

__init__.py (no changes)
config.py (no changes)
error.py (no changes)
ir.py (many changes in I2PY 0.2.0)
lex.py (no changes)
lexer.py (no changes)
map.py (a few changes in I2PY 0.2.0, search on "SDP")
maplib.py (superseded by idl_maps.py)
parser.py (no changes)
util.py (minor change to pyname() method to preserve case)
yacc.py (no changes)
ytab.py (no changes)

These are new in I2PY 0.2.0.

callfunc_utils.py (609 lines)
callfuncs.py (3232 lines)
idl_func.py (456 lines)
idl_maps.py (706 lines)
user_lib.py

List of Convenience Functions Used by I2PY 0.2.0

The "map_func()" and "map_pro()" functions in the API each have a "callfunc" keyword which is used to specify a "call function" that will be used to perform the translation. A set of "call functions" for translating built-in IDL routines are included in the file "callfuncs.py". These call functions often contain internal logic for (1) branching, (2) dissecting the IDL routine arguments to create corresponding Python/NumPy arguments and (3) for processing keywords and optional arguments. These call functions, in turn, depend on a set of utility or convenience functions in the file "callfunc_utils.py". The file "idl_maps.py" in the "i2py" folder imports "callfuncs.py", which in turn imports "callfunc_utils.py".

Note: I2PY "call functions" are often given as Python "lambda forms" -- essentially in-line function definitions. However, Python lambda forms may not contain statements, so they cannot themselves contain conditionals to determine branching. Starting with Python 2.5, lambda forms may contain "conditional expressions" which do allow branching. This capability is used in I2PY 0.2.0 (in idl_maps.py) for translating certain IDL routines, such as

ATAN, but is not especially useful when more complex branching logic is required. It is easier to define a new "call function" that is invoked by a lambda form in "idl_maps.py".

arrgen()
rampgen()
typeconv()
idl_key_set()
idl_arg_list()
idl_key_list()
idl_key_index()
reverse_arg_str()
arg_product_str()
keyword_var()
remove_chars()
get_window_index_string()
numpy_type_name()
idl_type_code()
idl_appended_graphics_keywords()
idl_color_mapping()
idl_graphics_keyword_commands()

List of Built-In IDL Procedures and Functions that I2PY can Translate

A total of 124 routines as of June 5, 2009.

Time-related: SYSTIME, WAIT

Pointer-related: PTR_FREE, PTR_NEW

Error handling: MESSAGE, STOP

Operating System Calls: CD, EOF, FILE_CHMOD, FILE_COPY, FILE_DELETE, FINDFILE, FILE_SEARCH, FILE_TEST, FSTAT, PATH_SEP, SPAWN

File Manipulation: CLOSE, FREE_LUN, GET_LUN, OPENR, OPENU, OPENW, POINT_LUN, PRINT, PRINTF, READF, READS, READU, WRITEU

Type Conversion: BYTE, FIX, LONG, LONG64, FLOAT, DOUBLE, UINT, ULONG, ULONG64, REAL_PART, IMAGINARY, COMPLEX, DCOMPLEX

Array Initialization: BYTARR, INTARR, LONARR, LON64ARR, FLTARR, DBLARR, UINTARR, ULONARR, ULON64ARR, PTRARR, STRARR, COMPLEXARR, DCOMPLEXARR, MAKE_ARRAY

"Ramp" Functions: BINDGEN, INDGEN, LINDGEN, FINDGEN, DINDGEN, UINDGEN,

ULINDGEN, UL64INDGEN, CINDGEN, DCINDGEN, SINDGEN

Math Functions: ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, CEIL, FINITE, FLOOR, MIN, MAX, RANDOMN, RANDOMU, ROUND, TOTAL
(Note: Many others, such as COS, SIN, TAN are the same in both IDL and NumPy.)

String Manipulation: EXECUTE, STRING, STRLEN, STRLOWCASE, STRMID, STRPOS, STRPUT, STRSPLIT, STRTRIM, STRUPCASE

Image Manipulation: BYTSCL (partial), REBIN (partial ??)

Array Manipulation: DETERM, HIST_2D, HISTOGRAM, INVERT, N_ELEMENTS, NORM, REFORM, REPLICATE, REVERSE, ROTATE, SHIFT, SIZE, SORT, UNIQ, WHERE
(Still need: BILINEAR, CONJ, CONVOL, INTERPOL, ISHFT, LUSOL, SMOOTH, TEMPORARY, TRANSPOSE, TRISOL)

Device and Window Manipulation: DEVICE, ERASE, WDELETE, WINDOW, WSET

Plotting (via matplotlib): CONTOUR, PLOT, PLOT_FIELD, SHADE_SURF, SURFACE, TV, TVSCL, XYOUTS
(Still need: PLOTS and many of the keywords)

Native GUI: DIALOG_PICKFILE

Miscellaneous: ONLINE_HELP (via the "webbrowser" package)

List of Important Built-In IDL Procedures and Functions that I2PY Cannot Translate Yet But That Should be Added Soon

a = ASSOC(unit, array_structure, offset, /PACKED)
a = numpy.memmap(file_unit, dtype=numpy.int16, shape=(m,n), offset=0)

CONVOL (use scipy.signal.convolve, convolve2d, etc.)
REBIN (need to write something for this?)
LOADCT
BILINEAR, INTERPOLATE, SMOOTH, etc.

Some Known Issues with I2PY Version 0.2

(*) There is no "master list" of unsupported IDL functions and procedures. For some, such as CONVOL, REBIN and SMOOTH, I2PY 0.2.0 will print an error message when they are

encountered in the IDL code. For many others, however, it will remain silent and you won't know that they weren't translated until you try to run the Python code.

(*) IDL's "widget_*" routines are not supported yet and may be difficult to convert automatically to wxPython commands.

(*) I2PY can't always tell for sure whether to open a file for binary vs. ASCII I/O, unless the SWAP_ENDIAN keyword to OPENR, OPENW or OPENU is used.

(*) IDL's WHERE returns -1 when there isn't a match, but the I2PY-converted code doesn't do this, so subsequent lines that test for this won't work. If the IDL code is edited to use the optional COUNT argument instead, which is set to zero when there isn't a match, everything works fine.

(*) FILE_SEARCH returning a null string when there's no match. Solution is to use the COUNT keyword instead.

(*) All IDL line continuation characters (LCCs) are removed, so users need to go back and add Python LCCs "\" manually to avoid long lines.

(*) IDL code should use REVERSE vs. ROTATE(y,2) to reverse the elements in a 1D array. Otherwise, converted code won't work. NumPy's rot90(y,-2) does not work on 1D arrays. NumPy's flipud() works on both 1D and 2D arrays, and gives same result as REVERSE for 1D arrays. However, for 2D arrays flipud() does not do same thing as rot90(y,-2).

(*) Arguments to READF, etc. can't be arrays yet, at least not when the FORMAT keyword is used.

(*) The ALL keyword to CLOSE. (see idl_func.close_all_files().)

(*) The HISTOGRAM function is not very well supported yet. It looks as though numpy.digitise() can be used to translate the REVERSE_INDICES keyword.

(*) Interpolation functions are not supported yet, such as INTERPOL, INTERPOLATE, BILINEAR, REBIN and SMOOTH.

Future Work

Python code to reproduce IDL's REBIN function:
http://www.scipy.org/Cookbook/Rebinning
http://fossplanet.com/python.numeric.general/thread-5184897-rebin/
http://mail.scipy.org/pipermail/numpy-discussion/2004-August/015782.html
(See all Previous Messages from last URL above.)

Mention the construct: "if (n_elements(a) eq 0) then a=0L"

Does the translated SORT command (and subsequent use) work as expected?