Examples (Supplemental data to the article OMPC: An Open-source MATLAB® -to-Python Compiler.)

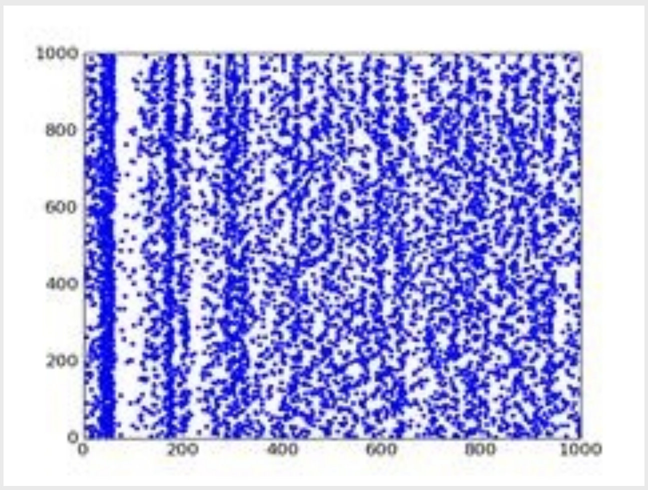
OMPC is a platform that makes it easy to move from MATLAB® to Python. MATLAB® can cost considerable amount of money, and for somebody using only its basic functionality without toolboxes, this seems like a waste. As we would like to prove with OMPC, MATLAB® is not irreplaceable. the usual reason to stick with it is reuse of already existing code. OMPC shall allow people to use their old MATLAB® code transparently, from Python. One should not have to rewrite all previously written functions or modules to switch to Python.

There is a number of applications that allow you to enjoy freedom while keeping your old MATLAB® code running, for example Octave and Scilab. These application are complete rewrites and required a lot of work before they became useful, not only the engine had to be developed, but much of the standard (non-toolbox) library of MATLAB(R) had to be rewritten. Python has its own numerical packages and a very stable engine that, as we show, can be used to run programs written in other languages.

You can download the current examples in a zip file.
They are also available at http://www.bitbucket.org/juricap/ompc/src/tip/examples/.

Look at the Izhikevich Simple spiking neural network model running in 100% pure Python.

(numpy example is included as well)



Contents

Navigation
An Open-Source MATLAB®-to-Python® Compiler
Request for features
Sitemap

Recent site activity
Top level
attachment from Peter Jurica
An Open-Source MATLAB®-to-Python® Compiler
edited by Peter Jurica
Examples (Supplemental data to the article OMPC: An Open-source MATLAB® -to-Python Compiler.)
edited by Peter Jurica
An Open-Source MATLAB®-to-Python® Compiler
edited by Peter Jurica
Request for features
edited by Peter Jurica
View All

Introduction

Example file: introduction.py

We start with an example from the introduction of the paper. One possible way of initializing the OMPC in Python is as follows:

```
>>> import ompc
>>> addpath('mfiles')
>>> import add
>>> add(1,2)
(3,)
```

The source code of the m-file imported is in the ***mfiles/add.m***:

```
function out = add(a, b)
% adds values of a and b
out = a + b;
```

The Python adapted version of the above m-file is automatically generated by OMPC. The statement "import ompc" extends the Python interpreter, and enables the execution of m-files. In this example, the statement "import add" will cause that a file "add.m" (that resides somewhere on the interpreters path) will be read by OMPC and translated, so that it can be executed in a Python interpreter. A successful execution of this demo will result in a file named ***add.pym***, with the following contents:

```
# This file was automatically translated by OMPC
(http://ompc.juricap.com)

from ompc import *

@mfunction("out")
def add(a=None, b=None):
    # adds values of a and b
    out = a + b
```

This small demo runs thanks to a number of concepts, each demonstrated and discussed in further sections. The concept are summarized by the source code:

1. **import ompc** - demonstrated on its own in ompc_ihooks.py, read a section below called **Automatic translation of ".m" files after Python import statement**.
2. **addpath('mfiles')** - mfiles is the location of the add.m file within the examples.
3. **import add** - besides being imported, thanks to to import hook installed by the preceding import statement, the add function is automatically compiled. This compilation involves 1) translation described in the section names **OMPC translator**, 2) modification of the function by the

mfunction decorator. The mfunction decorator is demonstrated in **The poor man's implementation of the mfunction decorator** and **Special variables nargin, nargout, varargin, varargout**.

4. **add(1,2)** - at this point the function is ready for execution in Python and should behave like a MATLAB(R) function. Notice that the OMPC translation does not contain the return a statement, however the function returns it's result anyway. This is because the **mfuction** decorator inserts byte-code for the correct return statement upon functions import.

Source code for all the demos resides in the "examples" archive and folder under the ***ompc/*** directory.

Growing cell-array

Example file: ***growing_array.py***

The cell array of MATLAB(R) is very similar to Python's list. One of the special features of MATLAB(R)'s arrays is that they are growable on demand. This means that one can assign to a non-existent index, the array will be first extended to contain the requested element and the position will be subsequently filled with a new value. The following demonstrates such feature in MATLAB(R):

```
>>> import ompc

>>> class mcellarray(list):
    def __setitem__(self,i,v):
        if i >= len(self):
            self.extend([None]*(i-len(self)) + [v])

>>> m = mcellarray()
>>> tic()
>>> for i in xrange(100000): m[i] = 12
>>> toc()
Elapsed time is 0.372690 seconds.
```

The time measurement is there for a reason. Let's look at MATLAB(R)'s performance on a similar example:

```
>> m = {}; tic, for i=1:100000, m{i} = 12; end, toc
Elapsed time is 9.637410 seconds.
```

Python's implementation of list is very good.

Online OMPC compiler

Example files: ***appengine/****
This is a Google Appengine application that demonstrating the OMPC compiler. This example shows how Python can be used for online demonstrations. A running instance of the small application is located at http://ompclib.appspot.com/ .
This application saves all submited m-files in the database and will serve as a bug-submission system.

OMPC translator

Example file: ***ompc/ompcply.py***

OMPC parser is written using the PLY package (http://www.dabeaz.com/ply/). The grammar was implemented to the best knowledge of the author. It is very possible that the compiler fails in some untested cases. If you find an incompatible statement please use the online compiler at http://ompclib.appspot.com/ . The submitted source file will be compiled and in case it fails it will be stored in the database, so the incompatibilities can be addressed.

Run the "ompcply.py" from within its directory. OMPC console will translate submited statements and print their MATLAB® equivalents. A captured session could look as follows:

```
Welcome to OMPC compiler test console!

ompc> a = 1;
a = 1
ompc> disp 12
disp("12")
ompc> a = 1
a = 1; print a
ompc> for x = 1:10,
for x in mslice[1:10]:
ompc> a(12:14) = [ class(a) rand(3).' ; 'test']
   a(mslice[12:14]).lvalue = mcat([mclass(a), rand(3).T, OMPCSEMI,
mstring('test')]); print a
ompc>
```

You can test your own statements. You can see that the parser obeys the semicolon ';' verbosity rules. Also notice that MATLAB® names which are a reserved word in Python are adjusted, for example *class* -> *mclass* and *print* -> *_print*.
The parser is easy to extend or change, it is a set of syntactical rules similar to the following examples:

```
def p_statement_list(p):
    '''statement_list : statement
                  | statement COMMA
                  | statement SEMICOLON'''
    p[0] = _print_statement(p[1], len(p) > 2 and p[2] or None, p[0])

def p_expression_lambda(p):
    '''expression : LAMBDA LPAREN name_list RPAREN expression'''
    p[0] = 'lambda %s: %s'%(p[3], p[5])
```

OMPC's approach is to generate a Python compatible source code and let the Python interpreter do the more complex task of executing the code. Another possibility would be to translate m-files directly to Python's byte-code.

Special variables nargin, nargout, varargin, varargout

***Example file:* arguments.py**

This example demonstrates the how special variables **nargin, nargout, varargin, varargout** can be emulated. At first the following code is tested:

```
@mfunction_arguments("out", "varargout")
def example1(a=None, b=None, *varargin):
    print "NIN: %d, NOUT: %d"%(nargin, nargout)
    print "VARARGIN", varargin, "VARARGOUT:", varargout
    out = 12
    return nargout > 0 and range(nargout) or None
```

The test cases and their corresponding output are here:

```
>>> example1()
NIN: 0, NOUT: 0
VARARGIN () VARARGOUT: []
>>> example1(1)
NIN: 1, NOUT: 0
VARARGIN () VARARGOUT: []
>>> example1(1,2)
NIN: 2, NOUT: 0
VARARGIN () VARARGOUT: []
>>> example1(1,2,4)
NIN: 3, NOUT: 0
VARARGIN (4,) VARARGOUT: []
>>> example1(1,2,4,'a')
NIN: 4, NOUT: 0
VARARGIN (4, 'a') VARARGOUT: []

>>> a = example1(1,2,4,'a')
NIN: 4, NOUT: 0
VARARGIN (4, 'a') VARARGOUT: []
>>> [a, b] = example1(1,2,4,'a')
NIN: 4, NOUT: 2
VARARGIN (4, 'a') VARARGOUT: [None]
>>> [a, b, c] = example1(1,2,4,'a')
NIN: 4, NOUT: 3
VARARGIN (4, 'a') VARARGOUT: [None, None]
>>> [a, b, c, d] = example1(1,2,4,'a')
NIN: 4, NOUT: 4
VARARGIN (4, 'a') VARARGOUT: [None, None, None]
>>> print 'OUT:', a
OUT: 12
```

Finally an example from the MATLAB online help on the varargin/varargout was used:

```
@mfunction_arguments("s", "varargout")
def mysize(x=None):
    nout = max(nargout, 1) - 1
    s = size(x)
    for k in m_[1:nout]:
        varargout(k).lvalue = mcellarray([s(k)])
```

The output is equivalent to the output in the web page:

```
>>> [s,rows,cols] = mysize(rand(4,5))
>>> print 's = %s, rows = %r, cols = %r'%(s, rows, cols)
s = [4, 5], rows = [4], cols = [5]
```

Calling mexFunctions from Python

Example file: mex/mex.py

This is an example of calling a *mexFunction* from Python. Only a very small subset of the mex API is implemented, this demo serves as a proof of concept.

The file ***mex.py*** contains a function called ***mexFunc***. This Python function wraps any dynamic-link library that exports an "extern ***mexFunction***"(as specified by the mex API). The wrapper automatically provides the following functionality:

- determines the number of input and output arguments
- create C-arrays emulating a regular mexFunction call. This example does not need MATLAB®! The comiled dynamic-link libraries can be called even from C or FORTRAN.

To run the example, first compile the C-filesby running the batch file "***compile***" in the "**examples/mex**" " directory. A succesful compilation should result in a file called **mexlib.lib** (a MEX library substitute) and the compiled mexFunctions in the same directory. At the moment, only batch files for Windows and Visual Studio are provided, if you would like to try this demo on Linux or a Mac, contact me (Peter Jurica).

Run the file ***mex.py*** .

You can try your own mex-files, but remember, only a small subset of mex API was implemented to prove the concept.

Emulating nargin/nargout, assigning to a function call

Example file: ompc_narginout.py

This example shows that Python allows emulation of the ***nargin/nargout*** variables. It also shows how values can be assigned to a function call. The following function is used for the demonstration:

```
def a(b=None,c=None):
    out1, out2 = None, None
    nargin, nargout = _get_narginout()
    print '  nargin = %s, nargout = %d'%(nargin, nargout)
    if nargin == 2:
        out1 = b + c
    elif nargin == 1:
        k = marray()
        out2 = '---'
    k = locals().get('k', marray())
    k(10).lvalue = 12
    return (out1, out2)[:nargout]
```

Assignment to a function call is enabled by the following minimal implementation of 'marray' object:

```
class marray:
    def __init__(self, val=[]):
        self.val = []
    def __call__(self, *args):
        print "  return elements %r"%args
        return self
    def __setattr__(self, name, val):
        if name == "lvalue":
            print "  I am an L-value ('%s') being set to %r."%(name, val)
            self.val = val
```

Function ***a***, in the first code snippet, first prints the number of submitted **in-**put

and **out-**put parameters. Next, the output parameters are computed.
The rest of the function demonstrates an assignment to a variable ***k***, which exists (if *nargin == 1)* or does not (if *nargin != 1*), before it is used. The statement k = locals().get('k', marray()) demonstrates how to ensure that a variable is initialized in the current scope before it is used. The following line, k(10).lvalue = 12, demonstrates an assignment to a function call. Finally, the last statement makes sure that only the requested number of output arguments is returned. The execution of this demo should result in output similar to this:

```
>>> a()
  nargin = 0, nargout = 1
  return elements 10
  I am an L-value ('lvalue') being set to 12.
(None,)
>>> a(1)
  nargin = 1, nargout = 1
  return elements 10
  I am an L-value ('lvalue') being set to 12.
(None,)
>>> b = a(1)
  nargin = 1, nargout = 2
  return elements 10
  I am an L-value ('lvalue') being set to 12.
None ---
>>> b, c = a(12, 11)
  nargin = 2, nargout = 2
  return elements 10
  I am an L-value ('lvalue') being set to 12.
23 None
>>> a, b, c, d = a('++', '--')
  nargin = 2, nargout = 4
  return elements 10
  I am an L-value ('lvalue') being set to 12.
Traceback (most recent call last):
  File "c:\devel\hg\ompc\src\ompc_narginout.py", line 60, in <module>
    a, b, c, d = a('++', '--')
ValueError: need more than 2 values to unpack
```

Automatic translation of ".m" files after Python import statement.

This demo shows how to invoke the OMPC compiler automatically on every Python import statement and how ***documentation strings*** (the first comment in the function definition) can be converted to Python *docstrings*. Automatic import of m-files is achieved by implementing import hooks and calling the OMPC compiler implemented in ***ompcply.py***.

Example file:* *ompc_ihooks.py

The **__*main*__** section of this file contains:

```
import sys
sys.path += ['mfiles']
import add
print add
help(add)
```

The class implemented within the same file implement an import hook. On the command import add, the Python interpreter looks besides the regular Python modules also for files with the extension ***.m***. The file ***mfiles/add.m*** is imported and a file add.pyc is generated. The ***.pyc*** file is created from a Python source code generated by ompc (saved as ***add.pym***) by the line:

```
m_compile.compile(filename, cfile)    # m-file compilation
```

The result if running this example should contain the following:

```
<function add at 0x00C22CF0>
Help on function add in module ompc:

add(*args)
    adds values of a and b
```

The import add results in a function object, this is different from the default Python import and can be only done, because m-files contain only 1 function callable from outside.

Doc-strings from comments

Example file: ***doc_comments.py***

This file contains a function called coinflip, with documenation in the first comment after the function declaration. This example shows how it is possible to extract this comments and convert them to a function doc-string.

Input:

```
@mfunction("x")
def coinflip(ndraws=None, p=None):
    # Generate a list x of zeros and ones according to coin flipping (i.e.
    # Bernoulli) statistics with probability p of getting a 1.
    # 1/20/97  dhb  Delete obsolet rand('uniform').
    # 7/24/04  awi  Cosmetic.

    # Generate ndraws random variables on the real interval [0,1).
    unif = rand(ndraws, 1)
    # Find all of the ones that are less than p.
    # On average, this prop
...
```

Output:

Help on function coinflip in module __main__:

```
coinflip(*args)
    Generate a list x of zeros and ones according to coin flipping (i.e.
    Bernoulli) statistics with probability p of getting a 1.
    1/20/97  dhb  Delete obsolet rand('uniform').
    7/24/04  awi  Cosmetic.
```

Poor man's implementation of the mfunction decorator

Example file: ***ompc_mfunction_poormans.py***

This example demonstrates the full functionality of the mfunction decorator implemented in an un-Pythonic way. Maybe this way interpreted lanuages were misused 10 years ago. The decorator looks up the function's source code and modfies it. Then this new source code is compiled and a new function object is created. This approach would fail without a source file available.

The function add in this example is executed at the end of the program.

Input:

```
@mfunction("out1, out2")
def add(a=None, b=None):
    # adds 2 numbers
    if nargin == 2:
        out2 = (a, b)
    out1 = a + b
```

Output:

```
def add(a, b):
    """adds 2 numbers"""
    nargin, nargout = _get_narginout()
    if nargout > 2:
        error("Too manu output arguments!")
    if nargin == 2:
        out2 = (a, b)
    out1 = a + b

    return (out1, out2,)[:nargout]
```

The output shows what the actual source code of a decorated function looks like. The real mfunction decorator however works directly with bytecode. It does not require the source code of the function.

Element-wise operations

Example file: *elementwise_operations.py*

MATLAB(R) offers a number of operators that are unique to the language. The operators are **.***, **./**, **.^** and **.**, they are used for element-wise operations on arrays while their simpler version *****, **/**, **^** and **** are used for matrix operations. Python does support these special operations. Operator overloading allows one to choose only one of the two possible operations. However with the help of a third object whose presence implies an element-wise operation we can emulate a number of new operators.

For example MATLAB(R)'s **a*b** can be preserved while the **a.*b** will be translated into Python's **a*elmul*b**. Run the example and look inside to see how this can be implemented. The result of running should look like the following:

```
--------------------------------------------------------------------
--------------------------------------------------------------------
SCALAR*SCALAR
>>> 1.*2
times 1 2
2
>>> 1*2
2
```

```
------------------------------------------------------------------------
VECTOR*SCALAR
>>> a = [1,2,3,4]
>>> a.*2
times [1, 2, 3, 4] 2
[2, 4, 6, 8]
>>> a*2
times [1, 2, 3, 4] 2
[2, 4, 6, 8]
>>> 2*a
times 2 [1, 2, 3, 4]
[2, 4, 6, 8]
------------------------------------------------------------------------
VECTOR*VECTOR
>>> b = [2,3,4,5]
>>> a.*b
times [1, 2, 3, 4] [2, 3, 4, 5]
[2, 6, 12, 20]
>>> a*b
mtimes [1, 2, 3, 4] [2, 3, 4, 5]
40
------------------------------------------------------------------------
```

examples.zip (152k) Peter Jurica, Dec 21, 2008 v.14

Comments

You do not have permission to add comments.

You do not have permission to add comments.