Documentation

OMPC is finally at a stage when you can start playing with it. At first you might want to have a look at the Examples page. It explains how and why OMPC works. The programs in the Examples section do not require anything but a working Python environment.

The stage of the development still doesn't allow you to use it as a replacement for MATLAB(R). This documentation is targeted at developers. It is more like something that is usually called a **hacking guide**.

Contents

Installation

First you will need a Python interpreter. The development and most of the testing is done with Python 2.5 (get it here).

Although my setup.py script is almost ready, I am still thinking about using alternative ways of installing OMPC, other than distutils and setuptools. OMPC will be changing rapidly now and I want to avoid issues with files left over from previous versions. I seriously recommend installing Mercurial.

To get the latest version of the package you could always get it in a zip file , tar.gz file or tar.bz2 file.
Downloading this package is equivalent to getting the newest version from the Mercurial repository.

```
hg clone https://www.bitbucket.org/juricap/ompc/
```

At the moment I am suggesting to work from within the directory that contains the **ompc** and **ompclib** directories. This means that after uncompressing the downloaded archive you should change into ompc-XXXXXXXXXXX.zip (ompc-2f62b3a16cd5.zip in my case). For example:

```
> wget https://www.bitbucket.org/juricap/ompc/get/tip.bz2
> tar xvfj tip.bz2
> cd ompc-2f62b3a16cd5
ompc-2f62b3a16cd5/.hg_archival.txt
ompc-2f62b3a16cd5/LICENSE
...
ompc-2f62b3a16cd5/test.py
> python
Python 2.5.1 (r251:54863, Jul 10 2008, 17:24:48)
```

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

```
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ompc
>>> print sum(reshape(mslice[1:30], 5,3,2),1)

ans =

(:, :, 1)

   15.0,   40.0,   65.0

(:, :, 2)

   90.0,  115.0,  140.0
```

You can look at the currently supported functions this way:

```
>>> from ompclib import __ompc_all__
>>> __ompc_all__
['addpath', 'tic', 'toc', 'mhelp', 'ohelp', 'pause', 'end', 'mslice', 'mstring',
'OMPCSEMI', 'OMPCException', 'elmul', 'elpow', 'error', 'isempty', 'disp',
'empty', 'zeros', 'ones', 'mcat', 'whos', 'size', 'rand', 'randn', 'reshape',
'sum', 'find', 'inv', 'eig', 'svd', 'poly', 'roots', 'conv', 'round', 'sqrt', 'set',
'xlabel', 'ylabel', 'zlabel', 'plot', 'bar', 'axis', 'grid']
```

This is not much but **numpy** functions can be used after a small wrapper is written. It should be not difficult to start writing a wrapper after you look in the ompclib/ompclib_numpy.py file. Otherwise read the next section. Such a wrapper can look like this

```
@_ompc_base
def round(X):
    return _marray('double', X.msize, np.around(X._a))
```

But more about it in the section **Adapting numpy functions for OMPC**.

The marray object

To start hacking the OMPC code it is better if you learn something about it's internals. All objects that fly around when a m-code is being interpreted are supposed to have the following

._a - a numerical object
.msize - MATLAB(R) compatible size (shape) vector
.dtype - MATLAB(R) compatible typecode (class in MATLAB(R))

It is essential that the data of the **._a** member are stored in the memory in the **FORTRAN** order. This is for the sake of reusing **MEX** extensions. **Numpy** can be made to work with almost any order but the **MEX** extensions and other **C**-code that works with MATLAB(R) compatible arrays assumes one single way of memory storage.

One would say that it is best to use an existing popular **numpy.ndarray** as a numerical object of OMPC. I agree. The **numpy.matrix** class is an example how to do it. **Numpy** is a great module but some of its features make it very difficult to simply inherit from its base numerical object. One reason is that the numpy's **FORTRAN** order is not MATLAB(R)'s **FORTRAN** order (at least for arrays with more than 2 dimensions). This is because the order of dimensions in **numpy** and MATLAB(R) is different. To get 5 matrices each with 3 rows and 2 columns in **numpy** you do **zeros((5,3,2))**, in MATLAB(R) **zeros(3,2,5)**. Notice that **(3,2)** has the same order in both packages but the rest of the dimension are named on different side of the dimension vector. This means that in numpy we have **(..., d4, d3, rows, cols)** but in MATLAB(R) **(rows, cols, d3, d4, ...)**.

```
>>> import ompc
>>> import numpy as np
>>> a = np.zeros((3,2),'f8','F')
>>> a[:,0] = 1
>>> from ctypes import *
>>> cast(a.ctypes, POINTER(c_double))[:6]
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0]
```

In **C** order this would have been [1.0, 0.0, 1.0, 0.0, 1.0, 0.0]. Then

```
>>> a = zeros(3,2)
>>> a(mslice[:],1).lvalue = 1
>>> cast(a._a.ctypes, POINTER(c_double))[:6]
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0]
```

Correct, but for 2 matrices :

```
>>> a = np.zeros((2,3,2),'f8','F')
>>> a[:,:,0] = 1
>>> cast(a.ctypes, POINTER(c_double))[:12]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

While this is the correct order

```
>>> a = zeros(3,2,2)
>>> a(mslice[:],1,mslice[:]).lvalue = 1
cast(a._a.ctypes, POINTER(c_double))[:12]
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0]
```

Keeping the shape the MATLAB(R) way while incompatible with **numpy** solves the problem:

```
>>> a = np.zeros((3,2,2),'f8','F')
>>> a[:,0,:] = 1
>>> cast(a.ctypes, POINTER(c_double))[:12]
[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0]
```

But printing such an array reveals what **numpy** thinks of such an array:

```
>>> print a
array([[[ 1.,  1.],
        [ 0.,  0.]],

       [[ 1.,  1.],
        [ 0.,  0.]],

       [[ 1.,  1.],
        [ 0.,  0.]]])
```

However, the main reason why one cannot derive **marray** from **ndarray** is that **ndarray** does not support in-place resizing. Growing of **ndarray**s is possible only by recreating the whole object and proper interpretation. This is a design feature, **numpy** is very efficient in storage of slices of an array, upon slicing it creates views that share data with their parent bigger arrays. Should such parent array change itself there is I believe currently no way of telling all children to quickly copy the current data because their parent is going to change.

For the above reasons the presented **marray** does not inherit from another numerical object directly. Rather it holds its data in the **._a** memebr variable. The **_a** member of an **marray('double', (5,3,2))** currently simply holds an **ndarray** with shape **(2,3,5)** stored in the **C**-order. All the the numpy functions that are going to operate on the should therefore assume that the axes are reversed.

The following shows how such **marray** can be converted to a **numpy** array:

```
>>> a = zeros(3,2,2)
>>> a(mslice[:],1,mslice[:]).lvalue = 1
>>> a._a.swapaxes(-1,-2)
array([[[ 1.,  0.],
        [ 1.,  0.],
        [ 1.,  0.]],

       [[ 1.,  0.],
        [ 1.,  0.],
        [ 1.,  0.]]])

>>> a._a.swapaxes(-1,-2)[0]
array([[ 1.,  0.],
       [ 1.,  0.],
       [ 1.,  0.]])

>>> print a

ans =

(:, :, 1)

     1.0,    0.0
     1.0,    0.0
     1.0,    0.0

(:, :, 2)

     1.0,    0.0
     1.0,    0.0
     1.0,    0.0
```

Holding data in a variables allows very fast adaptation of the numerical object to another numerical library. The pure Python version's **_a** holds a Python's **array** object from the **array** standard module.

Adapting numpy functions for OMPC

It will be better to explain this on a slightly more complicated function.

```
@_ompc_base
def find(cond):
    a = mpl.find(cond._a.reshape(-1)) + 1
    msize = (len(a), 1)
    if len(cond.msize) == 2 and cond.msize[0] == 1:
        msize = (1, len(a))
    return _marray('double', msize, a.astype('f8').reshape(msize[::-1]))
```

First I need to explain a number of new expressions:

- **_ompc_base -** is a decorator I use to make the function an OMPC top-level function. This means that it will be loaded into the workspace upon the **import ompc** statement. Using the decorator is equivalent to stating **__ompc_all__ += ['find']**.
- **mpl -** is a reference to **matplotlib** (**import pylab as mpl**), I am reusing here a ready implementation of **find** from **pylab**.
- cond.**_a -** in this version the **marray** object is based on **numpy**'s **ndarray** (more in the previous section).

A number of issues have to be taken care of by an m-function. In our example, the **pylab**'s **find** returns 0-based indices, therefore we use **mpl.find(...)+1**.

MATLAB(R) "squeeze"s the last dimension. So a correct implementation of the m-function **sum** should be:

```
>>> a = reshape(mslice[1:30], 5, 3, 2)
>>> sum(a,1).msize
[1, 3, 2]
>>> sum(a,3).msize
[5, 3]
```

Numpy squeezes all 0-length dimensions.

```
>>> np.sum(a._a,2).shape[::-1]
(3, 2)
>>> np.sum(a._a,1).shape[::-1]
(5, 2)
```

Some functions respect the original shape of vector, thus correctly:

```
>>> find(zeros(2,1)==0)
marray('double', (2, 1))
>>> find(zeros(1,2)==0)
marray('double', (1, 2))
>>> find(zeros(1,2,1)==0)
marray('double', (2, 1))
```

Also as you have probably noticed, **marray** never has less than two dimensions. **Numpy** however.

```
>>> import pylab as mpl
>>> mpl.find(zeros(1,2,1)._a==0).shape
(2,)
```

Taking care of these issues is often enough to make the function close to 100% MATLAB(R) compatible.

Importing m-files

Although there still may be small problems it should be possible to import m-files as in this example.

```
>>> import ompc
>>> addpath('examples/mfiles')
>>> import add
Importing m-file: "examples/mfiles\add.m"
>>> add(1,2)
marray('double', (1, 1))
```

```
>>> print add(1,2)
ans =
    3.0
```

The import statement initiates a search for files including fiels with **.m** extension. OMPC translates scripts to **.pym** files automatically and tries to import them. The Examples page has 2 sections explaining how is it possible.

Translating scripts

Use the following command to translate a file called **myscript.m**:

```
ompc/ompcply.py myscript.m > myscript.pym
```

The generated **myscript.pym** is a perfectly valid Python script that can be run by:

```
python myscript pym
```

It requires only that import ompc is called before the script is interpreted and that the implementation of ompclib has the function used in this script.

Comments

You do not have permission to add comments.

You do not have permission to add comments.