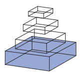
frontiers in
NEUROINFORMATICS

ORIGINAL RESEARCH ARTICLE
published: 10 February 2009
doi: 10.3389/neuro.11.005.2009



OMPC: an open-source MATLAB®-to-Python compiler

Peter Jurica* and Cees van Leeuwen

Perceptual Dynamics Laboratory, RIKEN Brain Science Institute, Wako-Shi, Saitama, Japan

Edited by:
Rolf Kötter, Radboud University Nijmegen, The Netherlands

Reviewed by:
Eilif Muller, Brain Mind Institute, EPFL, Switzerland
Dan Goodman, École Normale Supérieure, France

****Correspondence:***
Peter Jurica, Perceptual Dynamics Laboratory, RIKEN Brain Science Institute, Hirosawa 2-1, 351-0198 Wako-Shi, Saitama, Japan.
e-mail: pjurica@brain.riken.jp

Free access to scientific information facilitates scientific progress. Open-access scientific journals are a first step in this direction; a further step is to make auxiliary and supplementary materials that accompany scientific publications, such as methodological procedures and data-analysis tools, open and accessible to the scientific community. To this purpose it is instrumental to establish a software base, which will grow toward a comprehensive free and open-source language of technical and scientific computing. Endeavors in this direction are met with an important obstacle. MATLAB®, the predominant computation tool in many fields of research, is a closed-source commercial product. To facilitate the transition to an open computation platform, we propose Open-source MATLAB®-to-Python Compiler (OMPC), a platform that uses syntax adaptation and emulation to allow transparent import of existing MATLAB® functions into Python programs. The imported MATLAB® modules will run independently of MATLAB®, relying on Python's numerical and scientific libraries. Python offers a stable and mature open source platform that, in many respects, surpasses commonly used, expensive commercial closed source packages. The proposed software will therefore facilitate the transparent transition towards a free and general open-source *lingua franca* for scientific computation, while enabling access to the existing methods and algorithms of technical computing already available in MATLAB®. OMPC is available at http://ompc.juricap.com.

Keywords: technical computation, Python, Matlab, compiler

INTRODUCTION

Scientific progress is optimally served when everyone has access to the relevant information. No matter how effective commercial organizations, such as publishers or software houses, are in distributing information; their copyright and proper use requirements are often an impediment to information sharing. Open-access scientific journals attempt to remedy this problem; but this is only a first step, involving the free distribution of scientific results. The next step is to make auxiliary and supplementary materials that accompany scientific publications, such as methodological and data-analysis procedures, open and accessible to the scientific community in the form of freely downloadable software.

Sharing software tools requires a common platform. Currently one platform dominates the sciences: MATLAB®. As a commercial product, this language has successfully conquered the market for scientific communication (Moler, 2004, 2006) because it is easy to adopt for beginners as well as professionals, and because of its policy to offer licenses at reduced rates to educational institutions. However, it does not meet our criteria to be used as a common standard for free sharing of software tools. Using a method implemented in MATLAB® requires a full MATLAB® license. Moreover, its core software is closed source, preventing users from verifying, updating, and improving it.

While some MATLAB® users find the features of the language sufficient and see no reason to switch to an alternative, those who want to move to another platform feel the weight of code already written in MATLAB® impeding on their decision. Developers who have tried to offer an open-source alternative have made efforts to offer a level of compatibility with MATLAB®. Examples of such products are Octave and Scilab. None of these packages ever reached 100% compatibility and failed to meet the challenge of catching up with a platform with substantial financial support.

We propose OMPC as a possible alternative strategy to facilitate transition to an open-source platform. OMPC aims to offer a bridge between MATLAB® and Python. Development of the Python programming language project was started in late 1980s (http://www.artima.com/intv/python.html) at the National Research Institute for Mathematics and Computer Science in the Netherlands as an open-source scripting language for gluing components of an operating system. Today, powerful hardware allows Python to be used as a general purpose programming language. Over the years, the community contributing to the development of the Python language has grown considerably. Programmers and scientists alike are attracted by the simplicity of its syntax and its powerful set of features. Python is a good bet for a future free and open-source product that will develop far and fast enough to become the new *lingua franca* of technical computing (Fangohr, 2004; Langtangen, 2006).

Since the early stages there have been attempts to develop a Python package that offers certain features available in MATLAB®-compatible languages (http://matpy.sourceforge.net/). Scientific computation libraries were developed in the 1990s (Oliphant, 2006) and have been updated several times (Ascher et al., 2001; Oliphant, 2007), gaining in reliability, stability and versatility over years of development and use. The most important ones, especially in the context of our project, are *numpy*, *scipy* and *matplotlib* (http://numpy.scipy.org/, http://www.scipy.org/ and http://matplotlib.sourceforge.net/ respectively). The first two provide functions

largely equivalent to those of MATLAB®, while *matplotlib* is providing plotting functionality. Within the controlled development of Python, a proposal was made in 2000 to enhance Python with a feature that has been one of the major assets of MATLAB®: the availability of both matrix and element-wise operators (Zhu and Lielens, 2000). Another proposal has been to include a numerical array package *numpy* into the standard Python library, resulting in a revision of the buffer interface for the Python 3.0 (Oliphant and Banks, 2006). The new buffer interface facilitates the sharing of multi-dimensional data between different Python extension modules. All these developments point to an expanding role for Python in scientific computation.

The main problem with these packages is that each offers only a subset of MATLAB® features, but they lack a common, standardized interface. Our first aim, therefore, is to organize the available numerical libraries and provide them with a common interface. Our second aim is to provide 100% compatibility with MATLAB® syntax and with its dynamic interpreter (the MATLAB® engine). One advantage is that users will be able to download MATLAB® applications and run them for free. For programmers, OMPC offers the advantage of a free and open collaboration platform allowing reuse of code developed for the commercial MATLAB® platform without laborious rewriting.

OMPC is basically a translator of MATLAB® code to Python-compatible syntax. This paper discusses the compiler and the fundamental concepts that allow it to generate interpretable code; in particular code that will handle certain dynamic MATLAB® features not present in Python. For the generated code to work, OMPC needs to be complemented by a library that will ensure the proper interpretation of the translated code. We refer to this library as OMPClib. OMPClib contains, in particular, numerical objects that emulate the dynamical behavior of their MATLAB® counterparts. Proof-of-concept implementations of OMPClib that possess additional functionality just sufficient to reproduce the results of a spiking neural-network simulation (from Izhikevich, 2003) are presented in the Supplementary Material. OMPClib is a work in progress. A regularly updated version is found at the project's website (http://ompc.juricap.com). The current implementation of the OMPClib is an integral component of the OMPC package and is based on the extension modules *numpy*, *scipy* and *matplotlib*.

PROBLEM STATEMENT

In part, the translation of MATLAB® into Python code is a straightforward, technical problem. We need a compiler to generate Python compatible code from MATLAB® code (see The Compiler). In addition, there are four MATLAB® types (string, cell array, array, and slice) that have features not available in the corresponding Python objects. For these, we introduce Python objects that act as proxies for their MATLAB® equivalents (see Numerical Library).

The central, unique feature of the present translation problem is that both languages are interpreted languages, but have different dynamic features. Usually "dynamic" refers to a property of variable types and means that variables do not have to have a declared purpose or type – we refer to an object by its name and the interpreter decides at run-time if an operation on the variable is allowed. However, MATLAB® also adds dynamics to a number of other aspects of the language. The dynamic features of the MATLAB® engine differ from those of Python as well as most other general-purpose interpreters, because of the specific purpose for which MATLAB® was designed. These issues include: array slicing, on-demand updating of the variable namespace and populating it with implied variables such as *nargin/nargout*, element-wise operations, and implied returns. The dynamic feature of MATLAB® that is the most difficult to implement in languages other than Python is the *nargin/nargout* implied variable. The slicing syntax, although available in Python, differs in syntax. In subsequent Sections "Array Slicing, Index Base 1", "Dynamic Update of the Variable Name Space, Emulation of *nargin/nargout*", "Assignments to Novel Variables, Assignments to Slices", "Element-wise Operations" and "Implied Returns", we show how each of these particular problems can be solved. In Section "The *mfunction* Decorator" we mention how OMPC allows integration of these solutions with a minimum impact on the structure of the original MATLAB® code. Our approach illustrates that it is possible, given enough knowledge of the compiler of a particular language, to interpret code written in an arbitrary programming language, provided that the emulated language has a subset of the features of the emulating one. This translation maxim may apply universally between any pair of languages. However, as we argue, Python in addition is syntactically close, sufficiently dynamic, and has a large enough library to enable translation that leaves the original structure intact.

Any platform for technical and scientific computation should keep up to the standards of speed and quality of MATLAB®. This is only possible if such a platform is built on the base of standard numerical packages. Indeed at the base of all of currently competing scientific packages we find ATLAS (Automatically Tuned Linear Algebra Software). This is the reason why results of operations on matrices are bit-by-bit equivalent in MATLAB®, Python, Octave and many other tools. Also the speed of execution of operations defined in this library does not change significantly between different engines. There is no essential difference in speed of execution compared to compiled languages like C/C++; C/C++ code written by the average user can even be slower compared to implementations available from the ATLAS BLAS/LAPACK libraries used by *numpy/scipy*. This is because optimization of the elementary operations is done automatically at the time of compilation of the library and the speed of the result in not affected by the programming language from which this library is initiated (except for translation of parameters). The functionality of many toolboxes of MATLAB® is dependent on a number of other open-source packages as well. These are all available to Python users and probably have already been wrapped into a Python package. For custom made, non-standard packages (MEX extensions), we still need a way to allow OMPC to use them. This issue is discussed in Section "OMPC Extensions".

PROPOSED SOLUTION

An underappreciated aspect of Python, especially in scientific computing, is a feature known as introspection. Python offers built-in modules that allow run-time inspection of its own bytecode. Bytecode is the equivalent of the machine language in interpreted and just-in-time compiled languages. Introspection makes possible the run-time modification of the bytecode of a program, provided that the engine allows this. Python offers this facility. Where the

specific dynamic features of the MATLAB® engine have made it impossible for Python to interpret MATLAB® code directly, we show that with the help of introspection it is possible to emulate the remaining features. The following section presents specific features that together implement the proposed solution. Supplementary Material files on the project site include Python scripts that demonstrate the features presented in this section.

OMPC – A MATLAB®-TO-PYTHON COMPILER

OMPC is a compiler that translates MATLAB® code to functionally equivalent Python code. The design philosophy of OMPC is to enable seamless integration of existing MATLAB® code in Python programs. As a feature of convenience, OMPC allows automatic loading and translation of *.m* files using the Python *import* statement. Thus, assuming there is an m-function called *add* implemented in a file called *add.m*, an example Python session using this file would look as follows[1]:

```
>>> import ompc
>>> import add
>>> add(1,2)
ans = 3
```

The steps taken during execution are schematically illustrated in **Figure 1**. They are:

1. **import ompc** – OMPC installs a so-called import hook into the current instance of the interpreter. This allows OMPC to act at every import statement and compile *m-files* to Python code on demand. From this point on it is possible to import *.m* files.
2. **import add** – the OMPC import hook is called and searches for *add.m* on the current path (an equivalent to MATLAB®'s path variable). OMPC compiles *add.m* to a *.pym* file and submits this file to Python's built-in *__import__* function that will compile this file as any other regular Python file.
3. **add(1, 2)** – is a Python function call. It is running in the current Python instance as a Python function working with Python variables. In other words, MATLAB® is not involved at any stage of this process.

[1]The following sections contain listings of code in both programming languages. We adhere to the following convention: The mark >> at the beginning of a statement signifies a MATLAB® program, while the mark >>> signifies Python code. Each of the concepts introduced in the following subsections has a corresponding executable script that is part of the Supplementary Material.

OMPC is complemented by the module *OMPClib*. This module provides implementations of objects that act as proxies of dynamic features specific to MATLAB®.

Note that the mentioned enhancement of functionality is realized without any change to the Python language itself. It is absolutely important not to change the Python language in favor of a single package. Changes to the interpreter should only be made if they are met with general acceptance among the users of the language. Otherwise it would lead to the opposite of the unification aimed for. Moreover, a program translated by OMPC preserves the structure of the original MATLAB® program. The resulting program, in all but three cases (function declaration, switch statement, multiple statements on a single line), corresponds line by line to its MATLAB® source code. An example of equivalent MATLAB® and Python compatible codes can be found in the "Results" section.

THE COMPILER

To use MATLAB® code in Python, an intermediate step of MATLAB®-to-Python *syntax adaptation* is needed. The MATLAB® code must be parsed and translated into Python code that is functionally equivalent to its original. To parse MATLAB® source code we used a free 100% Python implementation of lex and yacc parsing tools called PLY (http://www.dabeaz.com/ply/). The compiler is implemented in a single Python file (examples/ompc/ompcply.py). This file is a collection of grammar definitions. Each definition is associated with a processing function for a specific language construct (keyword, number, assignment, index access and others). The grammatical rule for each construct is specified in the

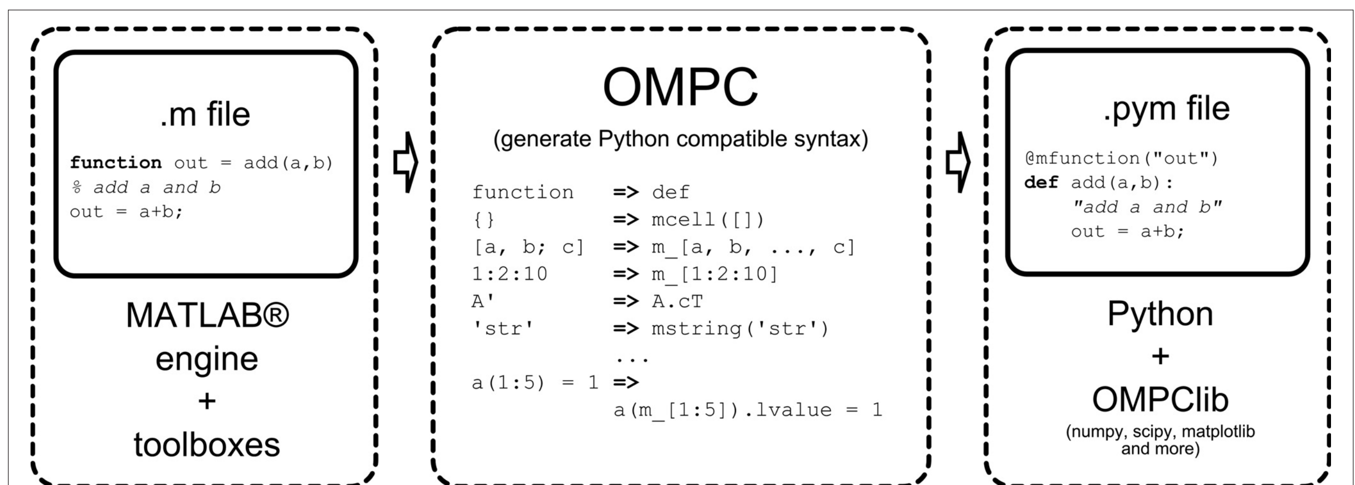


FIGURE 1 | OMPC structure. Each *.m* file has to be translated to Python compatible syntax. Statements for an *.m* file are replaced by their Python equivalents with minimal structural changes that allow emulation. This translated code relies on features implemented in a numerical object similar to *ndarray* of the *numpy* module.

documentation string of its processing function. The functions are designed to cover every syntactically correct MATLAB® language statement. The *PLY module* uses the grammar file to generate a parser, which searches a source text for language constructs and passes these to their corresponding processing functions. The parser produces the translated Python-compatible code. In the case of strings, the syntactical rule is the regular expression STRING = r'''((?:''|[^\n'])*)''' and the processing function looks as follows:

```
def p_expression_string(p):
    "expression : STRING"
    p[0] = "mstring(%s)"%p[1]
```

Every MATLAB® string that passes through this function will be enclosed in the expression mstring(.). Such a string can have all the features of a MATLAB® string. If this is not required, it is possible to replace the last line with *p[0] = p[1]*. As a result the strings from the original will stay intact.

The important advantage of using Python for the translation is that its code is easy to read and can be easily modified. Modifying a Python program does not require installation of a large complicated development system, common for low level languages like C++ or Java. The development advantages outweigh the negligible differences in processing speed.

NUMERICAL LIBRARY

Here we present the additional objects necessary for full compatibility with MATLAB®. The following MATLAB® example illustrates the impossibility of differentiating between variables and functions at translation.

```
>> add = @(a,b) a+b;
>> add(1,2)                    % Python -> add(1,2)
ans = 3
>> add = 1:10;
>> add(1,2)
ans = 2                        % Python -> add[0,1]
```

MATLAB® uses the same syntax for calling a function and retrieving elements from an array. This makes it impossible to determine if an identifier *add* in the above listing is a variable or a function. Therefore it is not possible to correctly translate the statement >> *add(1,2)* at compilation time. Our solution is based on the fact that object-oriented programming allows overloading of operators. We therefore have the option to overload the object's *__call__* function. Thus the OMPC code can be executed in Python, behaving equivalently to its MATLAB® original, independently of whether *add* is a function or a variable. Note that this added feature enhances the original numerical array (*numpy* in our examples) without altering its original function. The new object *marray* inherits all functionality from the original numerical array. This object enhanced by an overloaded *__call__* operator allows the following example to run in Python:

```
>>> add = lambda a,b: a+b;
>>> add(1,2)
3
>>> add = mslice[1:10];
>>> add(1,2)
ans = 2.0
```

The supplementary OMPC numerical object is currently based on *numpy*'s array object. This is however not the only option. It is possible to use base objects from another package like *Numarray*, *CVXOPT* (http://abel.ee.ucla.edu/cvxopt) or others. For non-numerical objects we can enhance Python built-in types. For example the OMPC string is based on the Python string implementation. The OMPC's cell array object is based on the Python built-in list object, which is equivalent in features to the cell array but, as is obvious from the following example, the performance boost achieved by using the Python list object is considerable.

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

Python does not allow on-demand growing of lists, but this feature can easily be emulated:

```
>>> 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 above example is not the optimal way of using the cell array. Such incorrect use of MATLAB®'s benevolent interpreter is, however, very common. As the last example shows, Python can help to greatly enhance the usability of such sub-optimal code.

ARRAY SLICING, INDEX BASE 1

The first element of a Python sequence type is 0, while MATLAB® uses 1 as the base for indexing, for instance a[0] in Python is equivalent to *a(1)* in MATLAB®. OMPC solves this incompatibility by overloading the numerical object's *__call__* method. The same technique of overloading the *__call__* function also makes it possible to use MATLAB® style array slicing. Consider again:

```
>>> b = a(1:10);
```

it is unclear until run-time if *a* is a function accepting a vector or a vector from which we are retrieving the first 10 elements. Python does not allow using a slice object outside of the index *[]* operator. By translating this statement into Python acceptable syntax

```
>>> b = a(mslice[1:10]);
```

and making *a* an object with overloaded *__call__* operation, this code can be executed in Python, behaving equivalently to its MATLAB® original independently of whether *a* is a function or a variable.

The *mslice* proxy object does two things. First it allows a slice object to be used as a parameter to a function call. Secondly it adapts MATLAB® index-base-1 slices from the syntax *start:step:stop* to Python's *start:stop:step*. Python's slice object returns slices up to the stop element, while MATLAB®'s slices range up to the stop element including it.

DYNAMIC UPDATE OF THE VARIABLE NAME SPACE, EMULATION OF *NARGIN/NARGOUT*

Python comes with a built-in module called *inspect*. Using this module it is possible to look into the execution stack to see in what context a function is being executed. This means that at any time a function is called we can look a couple of steps back in history and ask the interpreter about the code from which our function has been called. Consider the following statement:

```
>>> [a, b] = sort(rand(10,1))
```

Python accepts both *a, b*, and *[a, b]* (the correct syntax in MATLAB®) as left-value for an assignment. The *inspect* module makes it possible to ask the interpreter for the number of arguments on the left side of the assignment at the moment just before a function was called. The *OMPClib* module contains a function *_getnargout* that does exactly this. The following Python statement that leaves *nargout* undefined:

```
def f(x):
    if nargout == 2:
        return 1, 2
    else:
        return 1
```

can thus be rewritten to:

```
def f(x):
    nargout = _getnargout()
    if nargout == 2:
        return 1, 2
    else:
        return 1
```

The mfuction decorator, which will be discussed in detail in Section "The *mfunction* Decorator", makes sure that a call to the *_getnargout* function is inserted in the preamble of all functions translated by OMPC. This means that the original MATLAB® function body again can stay intact; we only need to apply the *mfunction* decorator that inserts *nargout* and, similarly, *nargin* into the variable namespace of the function during runtime.

ASSIGNMENTS TO NOVEL VARIABLES, ASSIGNMENTS TO SLICES

We explained that it is possible to use the *__call__* function to allow MATLAB®-style array slicing. There is one exception, however: Python does not allow function calls to be used for assignment. We circumvent this restriction by assigning to a property of the slice. The property mediates the assignment operation and makes the syntax acceptable to the Python parser. For instance,

```
>>> a(1) = 1          # Syntax error
```

is not allowed, but the following is:

```
>>> a(1).lvalue = 1
```

MATLAB® allows assignment to slices of variables that were not previously initialized. The module *inspect* allows us to detect assignment to non-existent variables. In the translated code, the variables are initialized during runtime by the *mfunction* decorator (see The *mfunction* Decorator).

ELEMENT-WISE OPERATIONS

MATLAB® offers a convenient way of differentiating between operations for matrices and their element-wise equivalents. Although such a differentiation was repeatedly proposed for Python (Zhu and Lielens, 2000) it never gained enough support from the broader Python community. In *numpy*, all numerical operations on arrays are element-wise by default. In principle, it would not have been a problem to use function calls to differentiate between these and matrix operations, for instance:

```
a .* b => multiply(a, b)    and    a * b => dot(a, b)
```

However in accordance with our principle to preserve as much as possible the original structure of the MATLAB® code, we suggest another solution. This solution is inspired by a recipe from the community-driven Python cookbook (http://code.activestate.com/recipes/384122/). Python allows overriding of operators on either side of an operand. This feature is commonly used to enable automatic coercion of types. For example, it allows the user to apply an arithmetic operation between a *numpy* array and anything else. So, for adding to array x a list *[1,2]*, instead of having to convert it to an array: $x + \text{array}([1,2])$, we can simply write: $x + [1,2]$. Therefore it is possible to change the above translation rule as follows:

```
a .* b => a *elmul* b       and       a * b => a * b
```

The *elmul* is an instance of an object that has overloaded the * operator (the *__mul__* and *__rmul__* function). Independently of the execution order of the operations in the statement, the *elmul* object remembers the operand from the first multiplication and instructs the second operand to perform element-wise multiplication (*a*elmul -> elmul.left = a, elmul*b -> elmul.left*b*).

IMPLIED RETURNS

MATLAB® uses implied returns; the "return" statement without parameters serves only for breaking the execution of a function. The return parameters of a function are specified in the function declaration. Python requires specification of these variables at each point of exit from the function. Python's return statement consists of a list of variables to be returned from a function call. Absence of the list means the empty object *None* is returned.

```
function [mi,ma] = minmax(a)
mi = min(a);
if nargout > 1, ma = max(a); end

@mfunction("mi, ma")
def minmax(a=None)
    mi = min(a)
    if nargout > 1: ma = max(a)
```

In the above example it is not possible to simply append a return statement *return mi, ma*. Because its value is being assigned to a single object (*mi*), the *minmax* function is expecting to return a single value. Python would therefore automatically assign a sequence, or tuple, containing both return values to the single variable at the output of the function call. This is illustrated in the following:

```
>>> mi = minmax(rand(1,10));
ans = (0.0574, None)
```

It would, in principle, be possible to add a statement *return (mi, ma)[:nargout]* in all locations where function exit could occur. This strategy would already rely on the introspection function to determine the value of *nargout*. However, adding such statements is cumbersome and destroys the structure of the original syntax. Introspection allows us to preserve the structure by automatically modifying the bytecode of translated functions, inserting the equivalent code wherever needed. This and other previously mentioned modifications to the bytecode are handled by the *mfunction* decorator.

THE *MFUNCTION* DECORATOR

Python offers the feature of decorators since version 2.4. Simply put, decorators are function factories. They allow us to turn a regular Python function into one that behaves like a MATLAB® function. A Python decorator receives a function just before it is loaded into the current workspace. The decorator can manipulate the function in arbitrary ways. The *mfunction* decorator modifies each function translated by OMPC. We use the decorator to emulate the existence of the variables *nargin/nargout*, to allow assignments to novel variables, and to implement implied returns (**Figure 2**).

This modification of byte-code happens at run-time. It happens only once when the interpreter loads a function, not every time the function is called. The performance of the decorated function does not differ from the performance of a function where modifications are stated explicitly in the source code.

OMPC EXTENSIONS

Here we deal with the issue of how OMPC handles C/C++ and FORTRAN (MEX) extensions for MATLAB®. Both MATLAB® and Python allow extensions and both have an official protocol for writing them. However, the interface between platform and extension differs considerably between the two respective languages. Extensions written for MATLAB®, therefore, do not work in Python. We can solve this problem by implementing a C support library that allows compilation of extensions independently of MATLAB. Compilation turns these routines into dynamic-link libraries that can be called by any language, including Python. The Supplementary Material has an example that shows how the *mxCreateDoubleMatrix* function can be implemented for example, using the Standard Template Library of C++.

In general it is very easy in Python to wrap external libraries by using the open-source application GCCXML (http://www.gccxml.org/). The Python community extensively uses this application for automatically generating Python extensions for libraries with complex structure and large numbers of exported symbols. The advantage of GCCXML over tools like Cython or Pyrex (http://cython.org/) and the multipurpose Swig (http://www.swig.org/) is that it is based on a production-stable GCC compiler. This means that any large project that relies on the latest features of C++, including the use of templates, can be automatically correctly parsed and analyzed to be further processed to

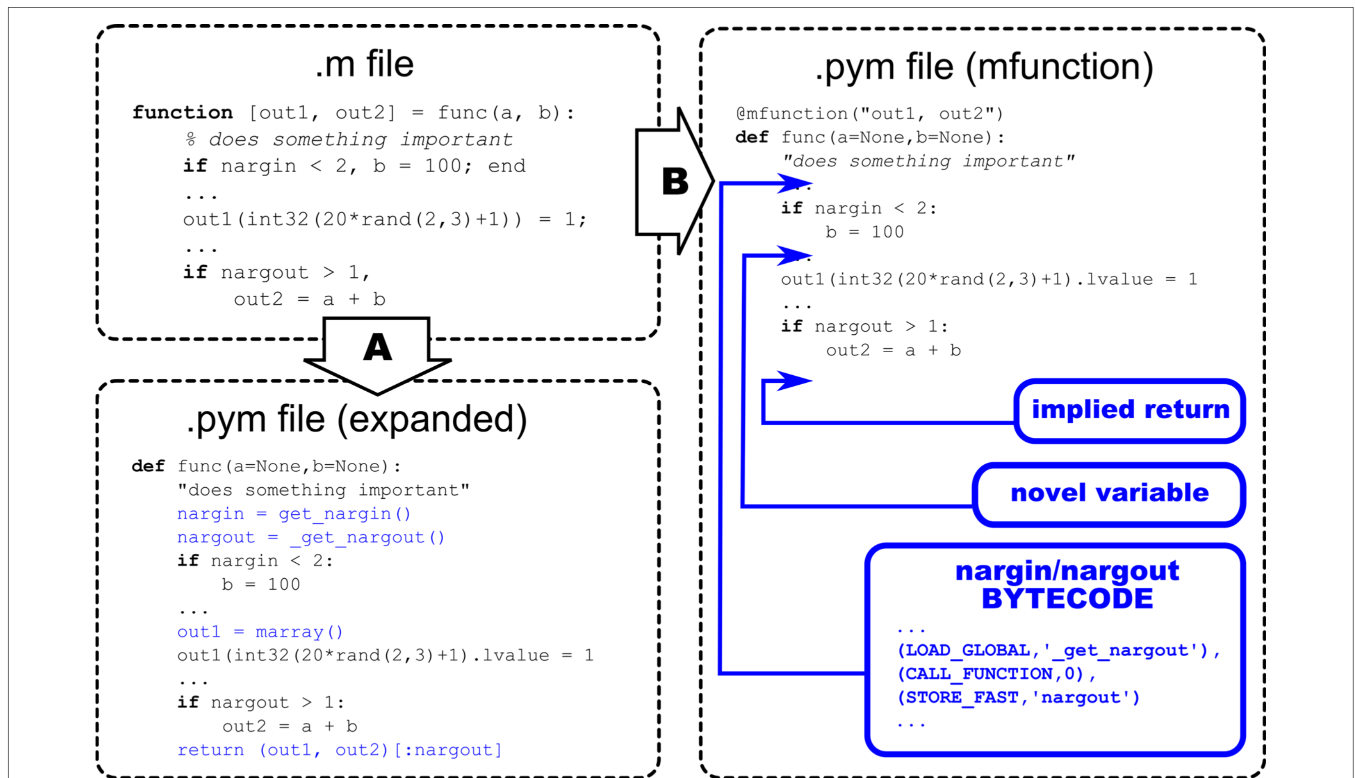


FIGURE 2 | Code injection by the *mfunction* decorator. Top-left panel: original MATLAB® code; Bottom-left panel **(A)**: translation with added code necessary for execution in Python without *mfunction*; Right panel **(B)**: illustration of how *mfunction* inserts byte-code into automatically translated functions at runtime. This is done only the first time each *mfunction* is loaded into the Python interpreter. Because these additions are invisible to the user, the structure of the original code remains intact.

generate extensions (www.boost.org/doc/libs/release/libs/python/doc/, http://pypi.python.org/pypi/ctypeslib/ and many others).

RESULTS

A website has been created for the project, http://ompc.juricap.com/. The compiler is also available on-line at http://ompclib.appspot.com/. This site will serve as a bug-tracking utility that will allow users to submit files that are not correctly processed by OMPC.

Because the formal specification of the MATLAB® syntax is not publicly available, it is difficult to properly test the OMPC compiler. However, we have successfully translated *m-files* that are part of the standard MATLAB® distribution. In addition, the compiler was tested successfully using source code collected from a number of users within the RIKEN Brain Science Institute and outside collaborators. The styling of MATLAB® source code varied significantly from person to person.

The following example consists of original source code, contained in online Supplementary Material to a neuroscience publication (Izhikevich, 2003). The example shows the original MATLAB® *m-file* and its fully automatic translation by OMPC.

```
% Created by Eugene M. Izhikevich, February 25, 2003
% Excitatory neurons   Inhibitory neurons
Ne=800;                Ni=200;
re=rand(Ne,1);         ri=rand(Ni,1);
a=[0.02*ones(Ne,1);    0.02+0.08*ri];
b=[0.2*ones(Ne,1);     0.25−0.05*ri];
c=[−65+15*re.^2;      −65*ones(Ni,1)];
d=[8−6*re.^2;          2*ones(Ni,1)];
S=[0.5*rand(Ne+Ni,Ne),−rand(Ne+Ni,Ni)];

v=−65*ones(Ne+Ni,1);  % Initial values of v
u=b.*v;               % Initial values of u
firings=[];           % spike timings

for t=1:1000          % simulation of 1000 ms
   I=[5*randn(Ne,1);2*randn(Ni,1)]; % thalamic input
   fired=find(v>=30); % indices of spikes
   if ~isempty(fired)
      firings=[firings; t+0*fired, fired];
      v(fired)=c(fired);
      u(fired)=u(fired)+d(fired);
      I=I+sum(S(:,fired),2);
   end;
   v=v+0.5*(0.04*v.^2+5*v+140−u+I);
   v=v+0.5*(0.04*v.^2+5*v+140−u+I);
   u=u+a.*(b.*v−u);
end;
plot(firings(:,1),firings(:,2),'.');
```

The OMPC equivalent is:

```
# Created by Eugene M. Izhikevich, February 25, 2003
# Excitatory neurons   Inhibitory neurons
Ne = 800
Ni = 200;

re = rand(Ne, 1)
ri = rand(Ni, 1);
a = mcat([0.02 * ones(Ne, 1),
         OMPCSEMI, 0.02 + 0.08 * ri])
b = mcat([0.2 * ones(Ne, 1),
         OMPCSEMI, 0.25 − 0.05 * ri])
c = mcat([−65 + 15 * re **elpow** 2,
         OMPCSEMI, −65 * ones(Ni, 1)])
d = mcat([8 − 6 * re **elpow** 2,
         OMPCSEMI, 2 * ones(Ni, 1)])
S = mcat([0.5 * rand(Ne + Ni, Ne), −rand(Ne + Ni, Ni)])

v = −65 * ones(Ne + Ni, 1)      # Initial values of v
u = b *elmul* v                 # Initial values of u
firings = mcat([])              # spike timings

for t in mslice[1:1000]:        # simulation of 1000 ms
    I = mcat([5 * randn(Ne, 1), OMPCSEMI,
             2 * randn(Ni, 1)]) # thalamic input
    fired = find(v >= 30)       # indices of spikes
    if not isempty(fired):
        firings = mcat([firings, OMPCSEMI,
                       t + 0 * fired, fired])
        v(fired).lvalue = c(fired)
        u(fired).lvalue = u(fired) + d(fired)
        I = I + sum(S(mslice[:], fired), 2)
    end
    v = v + 0.5 * (0.04 * v **elpow** 2 + 5 *
                  v + 140 − u + I)
    v = v + 0.5 * (0.04 * v **elpow** 2 + 5 *
                  v + 140 − u + I)
    u = u + a *elmul* (b *elmul* v − u)
end
plot(firings(mslice[:], 1), firings(mslice[:], 2),
     mstring('.'))
```

In this example we observe how well the translation preserves the structure of the original MATLAB® program. The above OMPC code is generated using rules that result in maximum compatibility. For example the last line contains the Python object *mstring('.')* that emulates the MATLAB® string object. As a consequence, the string is modifiable, as in the original. Since this is not necessary in the context of this program, a simple Python string could be used instead, as explained in Section "The Compiler". It is possible to further simplify the syntax by syntactical shortcuts, so called *index tricks* (*r_*, *c_*, *mgrid*), that are already part of the *numpy* library (Oliphant, 2006). The *plot* statement of the last program could therefore be simplified to, for example:

```
plot(firings(m_[:], 1), firings(m_[:], 2), '.')
```

The structural equivalence of both programs was made possible by using the introspection functionality of Python. Some of the dynamical features, however, can equally well be resolved by the OMPC compiler, provided that we are willing to compromise on structural equivalence. This would enhance the clarity of code for Python developers not familiar with implied variables of MATLAB®. Only adopting and testing OMPC will allow the users to make the correct decision. The final form of code generated by OMPC has still to be agreed upon. Future developments of the compiler will enable such options through switches.

In the Supplementary Material to this paper, we provide OMPC executables of the spiking neuron model described in (Izhikevich,

2003). At the moment of writing, two versions are available. One is based on the *ndarray* numerical array of the *numpy* library. However, optimized numerical packages such as *numpy* are not available yet for the newest Python interpreters. The other version, therefore, shows a pure Python implementation of an *n*-dimensional numerical array. This version is significantly slower for operations on large arrays but, because it runs on a clean Python installation, can be run on other realizations of Python as well; we have successfully tested this for Jython2.5a1, Python 2.6 and 3.0. The standard Python modules *array*, *random* and *math* are at the core of this second version; any Python interpreter sufficiently developed to contain these modules will execute the model. Maintaining a pure Python version of OMPC could enable acceleration of OMPC modules using PyPy (http://codespeak.net/pypy/, Rigo and Pedroni, 2006) or Shedskin (An Optimizing Python-to-C++ compiler, http://shedskin.blogspot.com/).

DISCUSSION

A number of different implementations of Python are currently available. We choose CPython because it is the primary Python engine; the most mature and stable implementation. All numerical extensions were originally developed for CPython. CPython, moreover, offers by default the *ctypes* module, which is of crucial importance as a support library for OMPC. CPython allows easy and efficient access to extension modules written in C/C++, FORTRAN and many other languages that allow us to create dynamic-link libraries.

Amongst forthcoming Python implementations that may influence the future development of OMPC, the most interesting one is PyPy. PyPy is an implementation of Python in Python itself and supports compilation of a restricted subset called RPython (Restricted Python, http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html#rpython, Section 1.4) into the C language and from there on into native binary executables. Although this possibility has not been tested, if PyPy will support specific CPython features it should be possible to compile OMPC generated files to native executables.

OMPC aims ultimately to offer full compatibility with the syntax *and* the engine of MATLAB®. A number of its features, however, have not yet been addressed in this article. The most sought after ones relate to its GUI components. Implementing these is practicable, based on the fact that the MATLAB® application GUI-designer stores its information in ".fig" files, which are actually .mat data files. This means that they can be loaded into Python using OMPC, enabled through the *scipy.io* module. These files hold enough information to identify and reconstruct the GUI components within a figure.

There is currently no plan to implement embedded Java, because we consider it not to be a crucial part of MATLAB®. While Java can be useful in MATLAB®, for example, for networking applications, the verbosity and complexity of Java are a great obstacle to use for anybody without a professional software engineering background. Moreover, all features that Java offers as an enhancement of MATLAB® are, most likely, present in Python as well. For networking purposes, therefore, Python is a much more suitable extension than Java for a high level language such as MATLAB®. Python includes support for networking by default. It contains modules with ready-to-use implementations of client-server applications. A good example is the OMPC on-line compiler currently hosted as a Python service at http://ompclib.appspot.com/.

In a broader scope, one of the great advantages of being able to parse source code is that it allows analysis and possible optimization of the code that will be executed. This is the approach taken by platforms based on virtual machines like.NET, Java and LLVM. Source code that can be parsed and translated into an intermediate format (CIL, formerly known as MSIL, Java Bytecode, or LLVM IR) can be run or translated to another low-level language including machine code. PyPy uses this technique to translate a sufficiently static subset of Python into C (Rigo and Pedroni, 2006). OMPC is an example of how to use Python byte-code as an intermediate representation.

Choosing Python as a platform for technical computation offers a number of additional benefits. As a popular general-purpose language, Python offers up-to-date facilities for online sharing, and enhancing the visibility of projects, in which computational methods are naturally embedded. The online OMPC compiler included in the Supplementary Material is one example of such an application. Python is currently one of the most popular tools in server-side Web 2.0 development.

The introduction mentions a number of attempts to provide MATLAB® functionality in Python. Currently there is only one actively developed project MlabWrap (http://mlabwrap.sourceforge.net/) that allows the use of MATLAB® functions along with the numerical extensions of Python. This project embeds the MATLAB® engine in a Python extension. This extension however requires a licensed copy of MATLAB®. A similar approach could be taken with the open-source library *liboctave* that is at the core of the GNU Octave (http://www.gnu.org/software/octave/). The design of OMPC allows any implementation of OMPClib to be used for execution of the OMPC generated Python code. An OMPClib could be built with liboctave's Array class as its base numerical object. The advantage of wrapping a library instead of embedding an interpreter is the great simplification of memory management. Embedding a interpreter in an extension is very similar to running a second process of which the data in memory are not directly accessible to Python and another extensions.

The interest of the scientific community in the Python language is growing (Langtangen, 2006, http://www.scipy.org/, http://www.neuralensemble.org/), making it ever more likely that it will become the main open-source language of scientific computation. One of the important obstacles in this transition is the large amount of legacy code written in MATLAB®. A fully automatic translation system could enable the reuse of large projects, the size of which makes human translation infeasible. By presenting OMPC, we demonstrated that Python could adopt MATLAB® code for reuse; without human intervention this code can be translated into Python. OMPC does this in a manner that, whenever possible, preserves the structure of the original. The syntax and design of MATLAB® language proved to be easy for beginners. In MATLAB® every object is also a multi-dimensional array, even a number is a 1×1 matrix. Python users however face the challenge of understanding concepts such as different types (numbers and arrays) and others common in programming, for example object reference. A

number of MATLAB® inspired features could help removing many obstacles for a user introduced to Python's numerical facilities. We discussed such features and their implementation in OMPC. By providing automatic translation of MATLAB® code to Python and the enhanced ease of use, OMPC will promote Python as the open-source alternative for scientific computation. To the Python community, OMPC offers this bridge as an incentive towards the further enhancement of numerical computation capabilities.

ACKNOWLEDGMENTS AND REMARKS

MATLAB® is a registered trademark of The MathWorks, Inc. "Python" and the Python logos are trademarks or registered trademarks of the Python Software Foundation.

SUPPLEMENTARY MATERIAL

The Supplemental Data for this article can be found online at http://ompc.juricap.com/.

REFERENCES

Ascher, D., Dubois, P. F., Hinsen, K., Hugunin, J., and Oliphant, T. (2001). Numerical Python, Technical Report UCRL-MA-128569, Lawrence Livermore National Laboratory. Available at: http://numpy.scipy.org.

Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering, Lecture Notes in Computer Science, Vol. 3039/2004. Berlin/Heidelberg, Springer, pp. 1210–1217.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572.

Langtangen, H. P. (2006). Python Scripting for Computational Science. Basel, Birkhäuser.

Moler, C., The Creator of MATLAB (2004). The Origins of MATLAB. Available at: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html.

Moler, C. (2006). The Growth of MATLAB and The MathWorks over Two Decades. Available at: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

Oliphant, T. E. (2006). Guide to NumPy. Trelgol Publishing, Spanish Fork, UT. Available at: http://numpy.scipy.org.

Oliphant, T. E., (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20.

Oliphant, T. E., and Banks, C. (2006). Index of Python Enhancement Proposals (PEPs), PEP 3118: Revising the Buffer Protocol. Available at: http://www.python.org/dev/peps/pep-3118/.

Rigo, A., and Pedroni, S. (2006). PyPy's Approach to Virtual Machine Construction, Dynamic Languages Symposium at OOPSLA. Available at: http://codespeak.net/svn/pypy/extradoc/talk/dls2006/pypy-vm-construction.pdf.

Zhu, H., and Lielens, G. (2000). Index of Python Enhancement Proposals (PEPs), PEP 225: Elementwise/Objectwise Operators. Available at: http://www.python.org/dev/peps/pep-0225/.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 14 September 2008; paper pending published: 13 October 2008; accepted: 30 January 2009; published online: 10 February 2009.

Citation: Jurica P and van Leeuwen C (2009) OMPC: an open-source MATLAB®-to-Python compiler. Front. Neuroinform. (2009) ***3****:5. doi: 10.3389/neuro.11.005.2009*

Copyright © 2009 Jurica and van Leeuwen. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.