SMOP

Summary

SMOP stands for Small Matlab/Octave to Python compiler. It converts Matlab and Octave programs to Python. The resulting .py program imports `runtime.py`, which contains the necessary runtime support -- definition of the `matlabarray` class and less than 100 small everyday functions.

SMOP is not a polished product, nor a replacement to octave and matlab. Taking into account its size (less than 3000 lines), this is not surprizing. Most annoying -- there are no matlab toolboxes. SMOP supports most -- but not all -- octave extensions to matlab. Support to octave toolboxes varies -- see below for their implementation status.

SMOP is written in Python, using PLY -- Python Lex/Yacc for lexical analysis and parsing, and numpy for runtime environment. SMOP is platform-independent, but is tested only on Linux. It is a command-line utility.

Command-line options

```
lei@dilbert ~/smop-github/smop $ python main.py -h
SMOP compiler version 0.25.1
Usage: smop [options] file-list
    Options:
    -V --version
    -X --exclude=FILES      Ignore files listed in comma-separated list FILES
    -d --dot=REGEX          For functions whose names match REGEX, save debugging
                            information in "dot" format (see www.graphviz.org).
                            You need an installation of graphviz to use --dot
                            option.  Use "dot" utility to create a pdf file.
                            For example:
                                $ python main.py fastsolver.m -d "solver|cbest"
                                $ dot -Tpdf -o resolve_solver.pdf resolve_solver.dot
    -h --help
    -o --output=FILENAME    By default create file named a.py
    -o- --output=-          Use standard output
    -s --strict             Stop on the first error
    -v --verbose
```

Example

It is possible to run an example without installing smop. Just unzip it somewhere, and `cd` to `smop` subdirectory. You will find a bunch of `.py` files and two `.m` files named `solver.m` and `fastsolver.m`. They are taken from the winning submission to Matlab programming competition in 2004 (Moving Furniture `http://www.mathworks.cn/matlabcentral/contest/contests/12/submissions/29989`).

```
lei@dilbert ~/smop-github/smop $ python main.py solver.m
```

Here are, side-by-side, the original file `solver.m`, and the SMOP output `.py`.

```
01   function mv = solver(ai,af,w)
02   nBlocks = max(ai(:));
03   [m,n] = size(ai);
04   I = [0  1  0 -1];
05   J = [1  0 -1  0];
```

```
01 def solver_(ai,af,w,nargout=1):
02     nBlocks=max_(ai[:])
03     m,n=size_(ai,nargout=2)
04     I=matlabarray([0,1,0,- 1])
05     J=matlabarray([1,0,- 1,0])
```

```
06   a = ai;
07   mv = [];
08   while ~isequal(af,a)
09     bid = ceil(rand()*nBlocks);
10     [i,j] = find(a==bid);
11     r = ceil(rand()*4);
12     ni = i + I(r);
13     nj = j + J(r);
14     if (ni<1) || (ni>m) || (nj<1) || (nj>n)
15         continue
16     end
17     if a(ni,nj)>0
18         continue
19     end
20     [ti,tj] = find(af==bid);
21     d = (ti-i)^2 + (tj-j)^2;
22     dn = (ti-ni)^2 + (tj-nj)^2;
23     if (d<dn) && (rand()>0.05)
24         continue
25     end
26     a(ni,nj) = bid;
27     a(i,j) = 0;
28     mv(end+1,[1 2]) = [bid r];
29 end
30
```

```
06     a=copy_(ai)
07     mv=matlabarray([])
08     while not isequal_(af,a):
09         bid=ceil_(rand_() * nBlocks)
10         i,j=find_(a == bid,nargout=2)
11         r=ceil_(rand_() * 4)
12         ni=i + I[r]
13         nj=j + J[r]
14         if (ni < 1) or (ni > m) or (nj < 1) or (nj > n):
15             continue
16
17         if a[ni,nj] > 0:
18           continue
19
20         ti,tj=find_(af == bid,nargout=2)
21         d=(ti - i) ** 2 + (tj - j) ** 2
22         dn=(ti - ni) ** 2 + (tj - nj) ** 2
23         if (d < dn) and (rand_() > 0.05):
24             continue
25
26         a[ni,nj]=bid
27         a[i,j]=0
28         mv[mv.shape[0] + 1,[1,2]]=[bid,r]
29
30     return mv
```

Though only 30 lines long, this example shows many of the complexities of converting matlab code to python.

line 3

Matlab function `size` returns variable number of return values. In python, functions are usually unaware of the expected number of return values. To solve this, nargout -- the expected number of return values is explicitly passed.

lines 4-5

Matlab array indexing starts with 1, python starts with zero. New class matlabarray derives from ndarray, but exposes matlab array behaviour.

line 6

Matlab array assignment implies copying; python assignment implies data sharing. We use explicit copy here.

line 7

Empty matlabarray object is created, and then extended at line 28.

line 9

Matlab functions of zero arguments can be used without the parentheses, such as *rand*.

line 10

Matlab function `find` returns variable number of values, which is explicitly passed in *nargout*.

line 12

Variables I and J contain matlab-style arrays.

Run-time support

- abs
- arange
- ceil
- cell
- copy
- disp
- false

- find
- floor
- fullfile
- intersect
- iscellstring
- isempty
- isequal
- length
- load
- max
- min
- ndims
- numel
- ones
- rand
- ravel
- round
- rows
- size
- strread
- strrep
- sum
- true
- zeros