Python Bindings to PTV library

1) Changes in C source code to compile as dll.

1.1) Add `#include"exp_dll_so.h"` to globals.h.

exp_dll_so.h is needed when compiling for win32 platform and defines `__declspec(dllexport)` macro (EXPORT definition).

//====== exp_dll_so.h ==========================

```
#ifndef __EXP_DLL_SO_H
#define __EXP_DLL_SO_H

#ifdef BUILD_DLL
      #define EXPORT __declspec(dllexport)
      #include<windows.h>
#else
      #define EXPORT
#endif
#endif
//===============================================
```

1.2) Include dll entry point in only **<u>one of the source files</u>** (doesn't matter which – only needed when we compile dll form several sources – for our example in lsqadj.c). **Not needed when compiling for linux platform.**

```
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, LPVOID

lpReserved)
      {
            return TRUE;
         }
```

1.3) For each function that we will use externally (from python), we need to add EXPORT definition before function declaration (in globals.h):

For example: `EXPORT void pixel_to_metric();`

1.4) Each global variable that is defined in globals.h as "extern" need to be redefined as EXPORT. For example: `EXPORT int  n_img;`

1.5) Several pointers to arrays that are used in globals.h need to be redefined in the following way:

`Exterior Ex[]` → `Exterior  *Ex`

Note: steps 1-4 are needed for WIN32 platform only

1.6) All references to tcl/tk are wiped from ptv.h

2) Compiling

2.1 Win32 Platform, using Visual Studio 2008 Express (freeware)

2.1.1 As example, we combine lsqadj.c and pointpos.c into single dll. In order to compile them, besides the changes described in 1.1-1.5 we need to compile together with multimed.c, imgcoord.c, intersect.c, ray_tracing.c, trafo.c since functions in these files are referenced from lsadj.c and pointpos.c

Compiling:

a) Enter Visual Studio 2008 command prompt
b) Enter following command:

cl -LD -DBUILD_DLL lsqadj.c pointpos.c multimed.c imgcoord.c intersect.c ray_tracing.c trafo.c -Felsq_point.dll

File that was produced is lsq_point.dll that we will use from python.

Flags:

-LD – our target is .dll not exe

-DBUILD_DLL - we use this only on win32 platform (see exp_dll_so.h)

-Fe – output dll name.

2.2 **Linux platform, using gcc**

For the example described in 2.1.1:

2.2.1 Enter directory with source code.

2.2.2 Compile:

gcc -c -fPIC lsqadj.c pointpos.c multimed.c imgcoord.c intersect.c ray_tracing.c trafo.c

this will compile the c files and will make .o object files.

2.2.3 Link:

gcc -shared lsqadj.o pointpos.o multimed.o imgcoord.o intersect.o ray_tracing.o trafo.o -o lsq_point.so

The output will be file named lsq_point.so which is our library.

Note: for linux/unix compling make sure that code mentioned in 1.2 is commented in source file.

On Mac OS X 10.6

gcc -c -fPIC -arch i386 -arch x86_64 lsqadj.c pointpos.c multimed.c imgcoord.c intersect.c ray_tracing.c trafo.c

gcc -shared lsqadj.o pointpos.o multimed.o imgcoord.o intersect.o ray_tracing.o trafo.o -arch i386 -arch x86_64 -o lsq_point.so

% > python test.py

[1. 2. 3. 4. 5. 6. 7. 8.]

[0. 0. 0. 0. 0. 0. 0. 0.]

[1. 5. 2. 6. 3. 7. 4. 8.]

Great !!!!

3) Making binding to python.

There are few options for binding C code to python. We use ctypes as method known for it's simplicity, minor changes in C code, and good integration with numpy. For other options, see:

http://www.suttoncourtenay.org.uk/duncan/accu/integratingpython.html

Now when we have lsq_point.dll for windows and lsq_point.so for linux we can make binding for python named lsq_point.py that converts input/output parameters from .dll or .so to python numpy arrays. In this example we make binding to mat_transpose function described in lsqadj.c:

============ lsq_point1.py==

```
import numpy as nm
import ctypes as ct

test_lib = nm.ctypeslib.load_library('lsq_point', '.')

# Set up interfaces
```

```
test_lib.mat_transpose.argtypes= [ct.POINTER(ct.c_double),ct.POINTER(ct.c_double), ct.c_int, ct.c_int]

# Define python function.
def mat_transpose(mat1, mat2, m, n):
test_lib.mat_transpose(mat1.ctypes.data_as(ct.POINTER(ct.c_double)), mat2.ctypes.data_as(ct.POINTER(ct.c_double)), m, n)
  return
```

==

Few explanations:

test_lib = nm.ctypeslib.load_library('lsq_point', '.') – loads lsq_point.dll or lsq_point.so to test_lib.

test_lib.mat_transpose.argtypes= [ct.POINTER(ct.c_double),ct.POINTER(ct.c_double), ct.c_int, ct.c_int] - describes input data types in terms of ctypes datatypes.

Last 3 lines in lsq_point1.py defines mat_transpose python function that will use mat_transpose C function by using numpy arrays mat1, mat2. Note for build in datatypes conversions numpy->ctypes:

mat1.ctypes.data_as(ct.POINTER(ct.c_double))

Test code that uses our new created bindings:

```
======= test.py ===================================
import lsq_point1 as lsq
import numpy as nm

data1 = nm.array([1.,2.,3.,4.,5.,6.,7.,8.])
data2 = nm.array([0.,0.,0.,0.,.0,0.,0.,0.])
print data1
print data2
lsq.mat_transpose(data1, data2, 2, 4)
print data2
==============================================
```

4. Creating bindings with help of Cython

As an example, we create bindings for lsqadj.c, for 2 functinos :

a) void ata(double *a, double *ata, int m, int)

b) void mat_transpose (double *mat1, double *mat2, int m, int n)

4.1 We create ptv1.pyx which is a Cython file with necessary declarations:

```
======= ptv1.pyx ==========================================
cimport numpy as np
cdef extern void ata(double *a, double *ata, int m, int)
cdef extern void mat_transpose (double *mat1, double *mat2, int m, int n)

def p_ata(np.ndarray s, np.ndarray sata, m, n):
        ata(<double *>s.data, <double *>sata.data,m,m)

def p_mat_transpose(np.ndarray s, np.ndarray sata, m, n):
        mat_transpose(<double *>s.data, <double *>sata.data,m,n)
```

Here we define p_mat_transpose which is python function that converts numpy arrays to pointers to arrays and calls C function mat_transpose

4.2 We create setup.py which is some kind of makefile to compile the extensions:

```
=========== Setup.py ================================================
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

import numpy as np
setup(
    name="ptv1",
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("ptv1", ["ptv1.pyx", "lsqadj.c"],
```

```
                    include_dirs = [np.get_include(),'.'],
                    extra_compile_args=['-O3'])],
   py_modules = ['ptv1',],
)
```

In this example we make use of single lsadj.c file only . If extension module is compiled from several sources, replace ext_modules line with:

ext_modules = [Extension("ptv1", ["ptv1.pyx", "lsqadj.c", “source2.c”,”source3.c”],

4.3 Compiling

Enter source directory and enter:

python setup.py build_ext –inplace

As a result, ptv1.so is created

4.4 Testing

Self explaining test.py to test the module:

```
====== test.py ================================================
import ptv1 as ptv
import numpy as nm

data1 = nm.array([1.,2.,3.,4.,5.,6.,7.,8.], dtype=nm.double)
data2 = nm.array([0.,0.,0.,0.,.0,0.,0.,0.], dtype=nm.double)
print data1
print data2
ptv.p_mat_transpose(data1, data2, 2, 4)
print data2
```