**Python Bindings to PTV library**

1. Changes in C source code to compile as dll.
   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. Include dll entry point in only **one of the source files** (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. 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. Each global variable that is defined in globals.h as “extern” need to be redefined as EXPORT. For example: EXPORT int n\_img;
  2. 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. All references to tcl/tk are wiped from ptv.h

1. **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:

1. Enter Visual Studio 2008 command prompt
2. 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 !!!!

1. **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 :

1. void ata(double \*a, double \*ata, int m, int)
2. 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