Embedding Python Models in Xspec

Arthea Valderrama

August 2023

Although recent AtomDB models are written in Python to ease development, spectral analysis is still largely done using C++ based XSPEC. Utilizing a C++ based code that can import Python models in XSPEC enables an astronomer access to newer models, without having to entirely write them in a different language. This document works with PyApec, as XSPEC already has a built-in Apec model, to utilize an embedded Python model. This is different from PyXSPEC, which uses XSPEC from a Python terminal, while embedding a Python model entails C++ code to call a built-in Python interpreter to run Python code.

1 Installation

For now, this method works using the default Python interpreter in a Linux system, not on something similar to the Anaconda version of Python. It is important to have the XSPEC software, as well as the HEASOFT package, which are available at `https://heasarc.gsfc.nasa.gov/docs/software/lheasoft/`. The following procedure should work as long as HEASOFT is installed and ready, meaning that it is initliazed in the environment and typing 'xspec' at the command terminal will start the software.

1.1 Modifying Python Code

Some edits to Python code must be made first. Any calls to `import xspec` must be removed due to circular dependency. Additionally, Python code that creates parameters for Python models should also be removed.

1.2 Modifying the Makefile

Firstly, wherever HEASOFT is installed, change into that directory. Then, change into the directory that XSPEC uses to create Makefiles for custom models.

```
> cd Xspec/src/tools/initpackage
```

It is recommended you know the location of your Python include directory and your Python library path. To find the Python include directory, run and open a Python terminal on your system as follows:

```
>import sys
>from distutils.sysconfig import get_python_inc
>python_include_dir = get_python_inc()
>print("Python Include Directory:", python_include_dir)
```

The result printed is the Python include directory, which may look something like `-I/usr/include/python3.9`. The -I flag before the Python include directory is imperative to linking the Makefile. To find the Python library path on your system, run and open a python terminal as follows:

```
>import sysconfig
>python_paths = sysconfig.get_paths()
>python_lib_dir = python_paths['platstdlib']
>print("Python Library Directory:", python_lib_dir)
```

The result may look something like `-L/usr/lib64/python3.9`. The -L flag before that path is important in linked to the Makefile. It will also need a link to the version of Python being ran, i.e, `"-lpython3.9"`, but it may vary depending on Python versions, so if it was Python3.6 it would then change to `"-lpython3.6m"` or so.

Using a text editor, open the `xspackage.tmpl` document. Under the `HD_CXXFLAGS` section, type in your Python include director preceded by "-I", followed by a space and a "\." Under the `HD_SHLIB_LIBS` section, type in your Python version preceded by "-l", and your Python library path preceded by a "-L" then followed by a space and a "\" to signify the end of line.

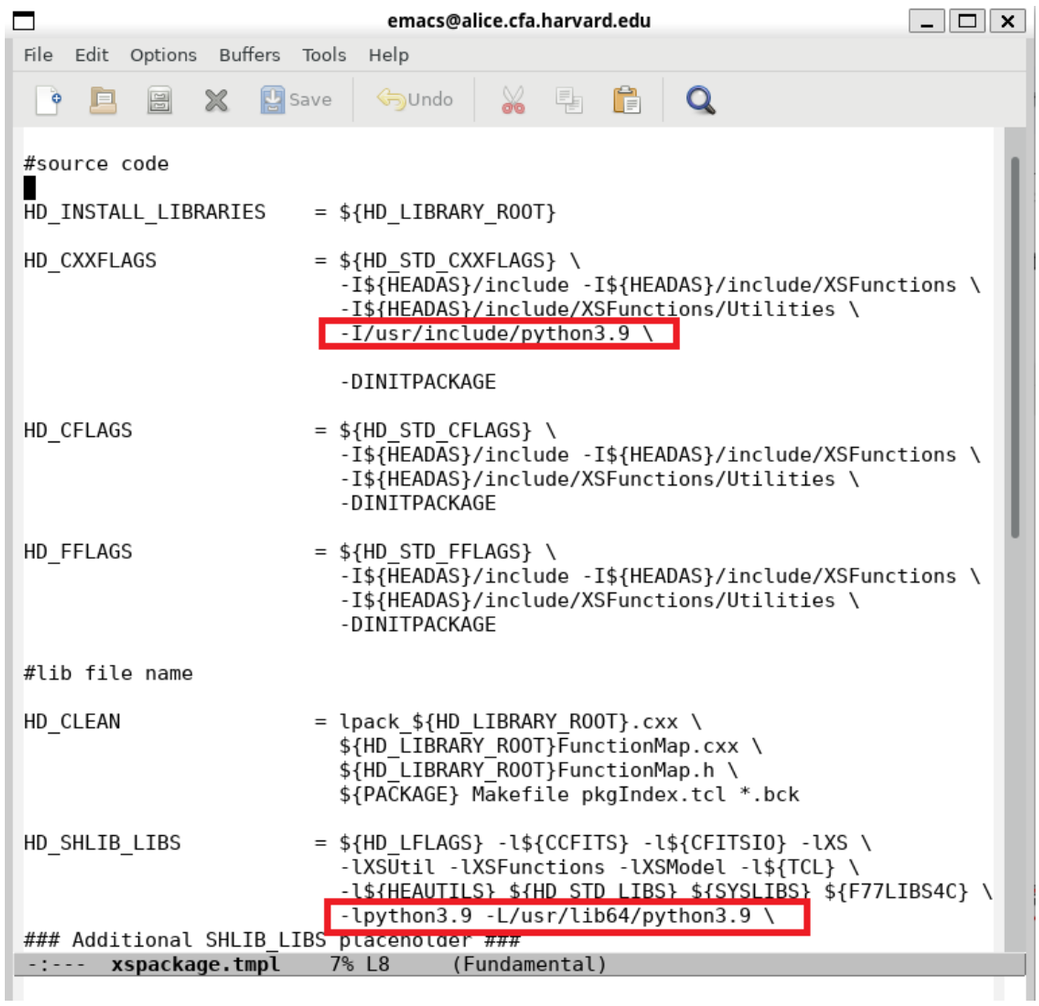


Figure 1: sample Makefile that links to proper Python libraries and directives, highlighted in red

Save the changes, and change up one directory (`"cd .."`, making sure you are in `Xspec/src/tools`), then run `"hmake"` and `"hmake install"`. This changes the Makefile so that now all local models will have these changes and the proper linkage to the Python interpreter in XSPEC.

1.3 Making a local model

If modifying the Makefile worked out correctly, you should now be able to run and write Python code for XSPEC. More information on making a custom model can be found at `https://heasarc.gsfc.nasa.gov/xanadu/xspec/manual/XSappendixLocal.html`.
In summary, one must have a `.dat` file detailing the model and its parameters (the file should contain the name of the model and the name of the model function) and at least one `.cpp` file with the main function being an extern "C" void function with the same name as the function outlined in the `.dat` file. Attached below is a sample C++ wrapper.

```
1
2 #include <python3.9/Python.h>
3 #include <numpy/arrayobject.h>
4 #include <vector>
5 #include <cmath>
6 #include <iomanip>
7 #include <iostream>
8 #include <string>
```

```
#include "xsTypes.h"

using namespace std;

class WrapperCalc {
public:
    WrapperCalc() {
        Py_Initialize();
        initPython();
        PyRun_SimpleString("import scipy.stats");
        PyRun_SimpleString("import sys");
        PyRun_SimpleString("sys.path.append(\"/scratch1/avalderr/pyspectrum
    \")"); //replace with path to your directory
        pyModule = PyImport_ImportModule("pyapec_script"); //replace with
    name of python script, ignoring extension
        if (!pyModule) {
            PyErr_Print();
            std::cerr << "Failed to import python script module\n";
            return;
        }
    }

    // RealArray is defined as an std::valarray<Real>, while Real is defined
     as a double in xsTypes.h
    void caller(const RealArray& energy, const RealArray& params, RealArray&
     flux) {
      if (energy.size() == 0 || params.size() == 0) {
        std::cerr << "Error: Input arrays must be not be empty." << std::
    endl;
        return;
    }


      PyObject* pyFunc = PyObject_GetAttrString(pyModule, "pyapec"); //
    follows name of the function listed in python script, may change
        if (!pyFunc || !PyCallable_Check(pyFunc)) {
            Py_XDECREF(pyFunc);
            Py_DECREF(pyModule);
            std::cerr << "python function not found or not callable\n";
            return;
        }

        // Create new Python lists with the same size as the RealArray
    vectors
        PyObject* pyEngs = PyList_New(energy.size());
        PyObject* pyParams = PyList_New(params.size());
        PyObject* pyFlux = PyList_New(flux.size());

        // Loop through each list and set each element with the
    corresponding value from its vector
        for (size_t i = 0; i < energy.size(); ++i) {
            PyList_SetItem(pyEngs, i, PyFloat_FromDouble(energy[i]));
        }

        for (size_t i = 0; i < params.size(); ++i) {
            PyList_SetItem(pyParams, i, PyFloat_FromDouble(params[i]));
        }

        for (size_t i = 0; i < flux.size(); ++i) {
```

```
60              PyList_SetItem(pyFlux, i, PyFloat_FromDouble(flux[i]));
61          }
62
63          // Create a new tuple object to pass the arguments to pyFunc
64          PyObject* pyArgs = PyTuple_New(3);
65          PyTuple_SetItem(pyArgs, 0, pyEngs);
66          PyTuple_SetItem(pyArgs, 1, pyParams);
67          PyTuple_SetItem(pyArgs, 2, pyFlux);
68
69          // Call pyFunc in pyapec_script with the arguments contained in the
    python tuple pyArgs, return the result in pyResult
70          PyObject* pyResult = PyObject_CallObject(pyFunc, pyArgs);
71          if (!pyResult) {
72              PyErr_Print();  // Print Python error information
73              Py_XDECREF(pyFunc);
74              Py_DECREF(pyModule);
75              Py_DECREF(pyArgs);
76              std::cerr << "Error occurred during python function call\n";
77              return;
78          }
79
80          // Extract the results from the pyFlux list (output of pyapec
    function) and store it in the C++ "flux" array
81          for (size_t i = 0; i < flux.size(); ++i) {
82              flux[i] = PyFloat_AsDouble(PyList_GetItem(pyFlux, i));
83          }
84
85          Py_DECREF(pyResult);
86          Py_DECREF(pyFunc);
87          Py_DECREF(pyModule);
88          Py_DECREF(pyArgs);
89      // Py_Finalize();
90      }
91
92      ~WrapperCalc() { }
93
94  private:
95      bool initPython() {
96          import_array();
97          if (PyErr_Occurred()) {
98              PyErr_Print();
99              PyErr_SetString(PyExc_ImportError, "Failed to import numpy.core.
    multiarray");
100             return false;
101         }
102         return true;
103     }
104
105     PyObject* pyModule = nullptr;
106 };
107
108 extern "C" void pyapecInfo(const RealArray& energyArray, const RealArray&
    params, int spectrumNumber, RealArray& fluxArray, RealArray&
    fluxErrArray, const string& initString) {
109     size_t N(energyArray.size());
110     fluxArray.resize(N-1);
111     fluxErrArray.resize(0);
112     WrapperCalc calc;
113     calc.caller(energyArray, params, fluxArray);
```

```
114 }
```

The first line indicates the Python include directive, which may vary by your version of Python. A C++ class is used to contain the Python interpreter object, which is then called by the model routine function. The constructor initializes the C++ Python interpreter, imports numpy, and attempts locates the name of your python script module depending on the directory in which the file is located. The bulk of the conversion of data is located in the `caller()` function, which converts data values from C++ to Python, performs a function call to an embedded Python interface, and returns the data values from python back to C++ in a `RealArray& Flux`. It then cleans up all of the objects created, and the destructor frees up the class object once ran.
For the purposes of this tutorial, the directory of the location in which the custom model is held also contains a `.py` file for the python script to be imported. Creating a model for example would be as follows:

```
>initpackage <modelname> model.dat .
>hmake
```

When you type in `ls` into the terminal, a list of files created by initpackage should be shown. The most important one to note is the `lpack_<modelname>.cxx`. Open that file with a text editor, and add `#include <dlfcn.h>` to the libraries. Inside the first function, add the following line:

```
dlopen("lib<yourversionofpython>.so", RTLD_LAZY | RTLD_GLOBAL);
```

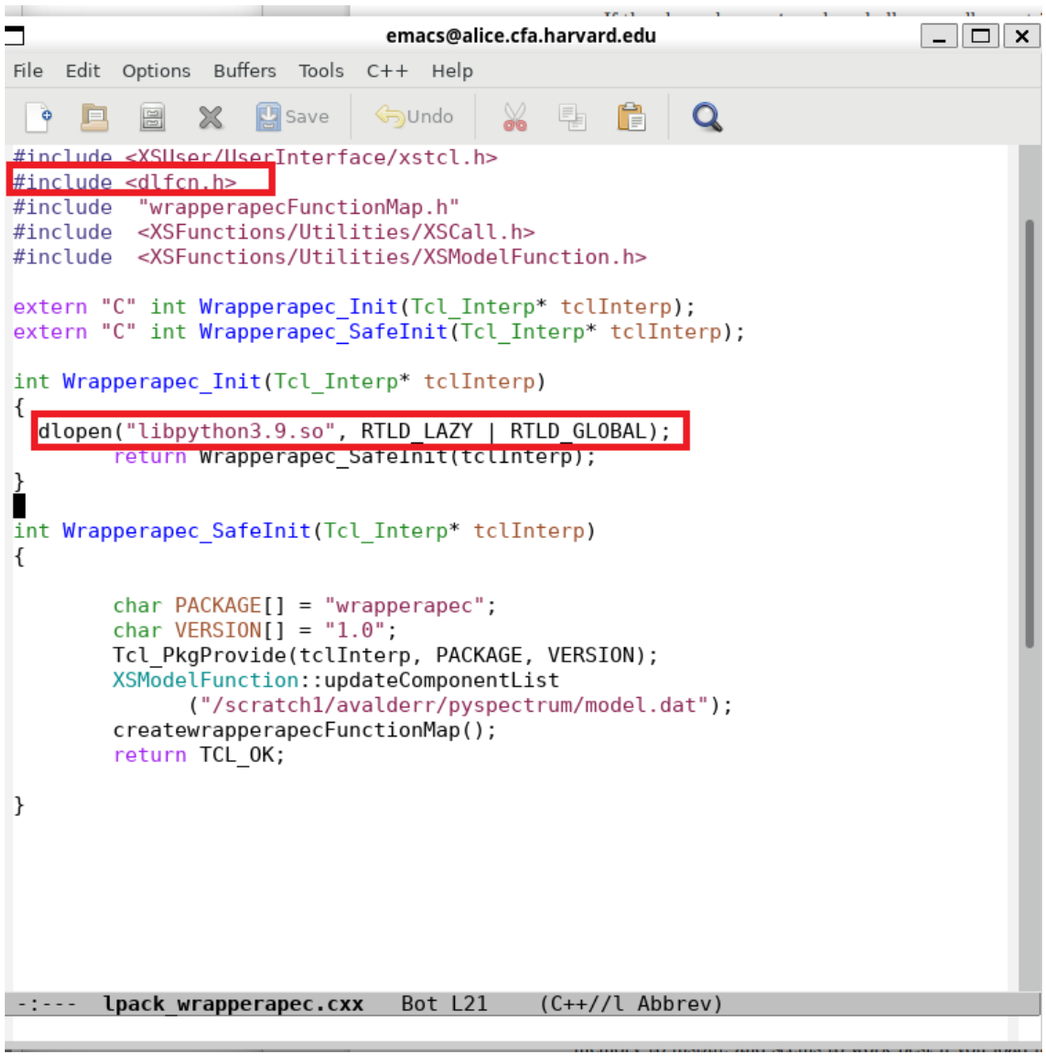


Figure 2: Sample lpack_<modelname>.cxx file edited, highlighted in red.

It should be noted that if you have an earlier version of Python, i.e Python3.7 to also include the "m" that supercedes that version. Some earlier versions of Python would be written as "libpython3.6m.so" to work. This is important in importing Python modules like numpy into XSPEC.
Ideally all the changes should allow one to import and run their custom embedded Python model into XSPEC. Open XSPEC, and load the model as follows:

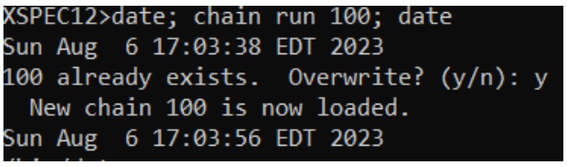
```
xspec>lmod <modelname> /path/to/local/model
```

The model should load successfully! Now you can use it and treat it as any other XSPEC model.

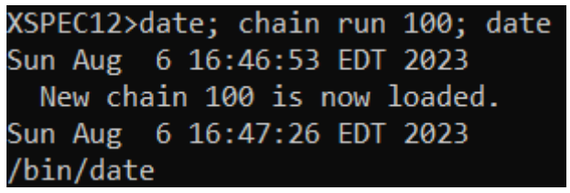
2 Usage

Ideally, embedded Python models properly load into XSPEC without unexpected core dumps or improper linkage issues. Inside XSPEC, and typing in the "`model`" command allows you to view the list of models that XSPEC has loaded in that section. Your custom routine function name should be listed, followed by an `*` indicating that it is an imported model. Running the model command again followed by the name of your custom routine will initialize the model to be used in spectral analysis, as you will later be provided with the prompt to input parameters as listed in the custom `.dat` file.

In the context of speed, fitting times vary between routines such as the built-in model in XSPEC, Python models, and embedded Python models.



(a) Benchmark of an apec model running 100 different fits



(b) Benchmark of an embedded model running 100 different fits

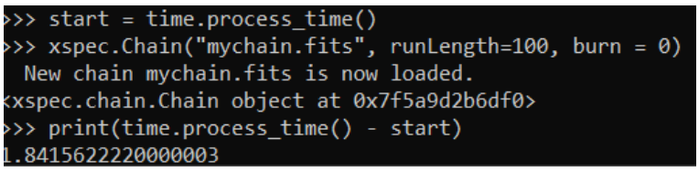


Figure 4: Benchmark of apec model in Python running 100 different fits

The embedded Python model in XSPEC is slower than the C++ built-in model, but by an approximate factor of 2 due to the data conversions between C++ and Python. Purely running Python is the fastest route. While these benchmarks show differences in speed between the routines, there should not be a significant noticeable difference running models one at a time.

3 Acknowledgements

In the process of discovering this method to load Python models into regular XSPEC, I have run into many unexpected core dumps and memory issues. Potentially the largest issues I have come across is importing numpy into XSPEC, and trying the method with Anaconda Python. I would like to thank Adam Foster and Nancy Brickhouse for advising me throughout this process and guiding me through XSPEC. I would also like to thank Craig Gordon for assisting me with debugging and rewriting XSPEC to run a Python interpreter. Lastly, internet sources such as StackOverflow and the official Python/C documentation (which can be found here:`https://docs.python.org/3/extending/embedding.html`) were essential in helping me figure out this process.