Computer Science Master of Engineering Project

PyFEAP – A Python Interface to Finite Element Analysis Program (FEAP)

Name: Chen-Yi (Charlie) Chen
NetID: cc2275
I. Introduction:

This M. Eng. Project is about PyFEAP, which is a Python interface to the Finite Element Analysis Program (FEAP) server. By utilizing this PyFEAP module, users can access both Python's and FEAP's features simultaneously, and can share data between two environments. In brief, PyFEAP allows users to perform the following tasks:

- Write a script in Python to perform relevant finite element analysis through FEAP.
- Use Python scripting ability to run parameters studies in FEAP.
- Extract mass, damping, and stiffness matrices for experiments with new linear solvers in a finite element system.
- Evaluate reaction forces for a given displacement to experiment with new solvers.
- Plot FEAP structure using Standard FEAP's X11 graphics
- Debug FEAP features by directly examining and manipulating FEAP's internal data structures.

The PyFEAP module helps FEAP users to script more easily since Python is marvelous when it comes to scripting. In addition, Python's stability and version backward compatibility makes it a good platform to create an interface on.

II. Background:

This project idea stemmed from Professor David Bindel, who has worked with finite element analysis and FEAP for years. Before this project started, Professor Bindel developed some extensions to FEAP to help him and others perform more complicated tasks that are difficult in FEAP. Two such extensions are MatFEAP and FEAPMEX; both of them provide an interface from MATLAB to FEAP and extend some basic functions that FEAP can perform in FEAP environment. FEAPMEX, which is the predecessor of MatFEAP, uses the MATLAB MEX files feature to extend MATLAB with external functions written in C and Fortran. Therefore, FEAPMEX is written in a mix of C, Fortran, and MATLAB, and the MATLAB functions in FEAPMEX provide a high-level scripting interface to FEAP.

FEAPMEX was a tightly integrated code that relied on low-level interfaces in both FEAP and MATLAB. As FEAP and MATLAB evolved, this organization caused maintenance problems, and it became difficult to maintain compatibility across multiple versions of FEAP and MATLAB. In order to address the maintenance problem, MatFEAP decouples the system into a FEAP-based server and a MATLAB client that communicate via a socket. This approach, MatFEAP eliminates the dependence of Mex files and minimizes the dependence between FEAP and MATLAB.

MatFEAP works pretty well as long as users have MATLAB installed. However, MATLAB is a licensed software so users need to buy it. Thus, Professor Bindel had the idea to create a Python interface that is similar to MatFEAP. Python is a great platform for such tasks. It is open-sourced and free to install; it has great backward compatibility between versions (except the revolutionary Python 3). Python is also a very popular scripting language that supports scientific computing through different third party libraries. All these great features make Python our choice of platform to create this project—PyFEAP.

III. Architecture:

PyFEAP is built around two processes: one for Python, and one for FEAP. The processes are effectively cooperatively multi-tasked. Both processes have to be active in order to make the whole system runs. The FEAP processes starts a FEAP server and acts as a daemon to listen to the client request. Thus, when we initialize PyFEAP process, namely calls to pyFeap.start, we create a TCP
socket connection between FEAP process and Python process. In addition, when issuing different PyFEAP commands such as `pyFeap.cmd`, commands use this socket connection to transfer to FEAP server for calculation, then PyFEAP retrieves the computation results from FEAP to display to the user in Python console. Similarly, the variables are sent and received between processes using standard TCP socket connection as well.

As stated above, the communication method between PyFEAP and FEAP server uses the standard TCP socket built-in Python native library. PyFEAP uses the `AF_INET` and `SOCK_STREAM` parameters as the socket connection and socket type. `AF_INET` in a Python socket represent a tuple of connecting to `(host, port)`, where the `host` is a string domain name or a IPv4 IP string, and the `port` is the port number that socket will connect to. `SOCK_STREAM` specifies the socket type PyFEAP is using here as a binary stream socket. As default, PyFEAP uses blocking socket which means Python blocks the routines when socket is still waiting on sent information or has nothing more to return. Therefore, it would be dangerous to "block" a the PyFEAP internal socket connection to FEAP server by directly calling `PyFEAP.socket.send` or `PyFEAP.socket.recv`, as it might block the socket connection and hangs the PyFEAP object. It is always best to use the PyFEAP defined `_send()` and `_recv()` functions, as they have the buffer size determination that would prevent PyFEAP internal socket from blocking.

Often, when PyFEAP encounters errors, it will raise custom exceptions and kill the connection to FEAP immediately. So the FEAP server would still be in a consistent state. However, there might be times when PyFEAP's lower level functions encounter an error that is not handled. In those cases, Python would still throw standard Python exception generated by PyFEAP, but the state of FEAP server is indeterminable. Even though most of the time when this case happens, the restarted computation script using PyFEAP seems to be okay, it would still be safe to restart the FEAP server instance with the corrected computation script. In that way, we can guarantee the FEAP server would be in a consistent state thus guarantees correct computations.

In general, FEAP server has its own signal handler Ctrl-C, which allows the user to stop the daemon to close everything. Similarly, Python has its own signal handler for Ctrl-C. The user can use Ctrl-C to break out any running script or computations to return to Python console or terminal prompt.

---

**IV. Examples:**

In this report, I provide two examples to show how to generate python scripts that utilize PyFEAP interface to communicate to FEAP:

1. **Modal analysis of a block**

   Our first, and simplest, example involves computing the first few modes of a 2D block of material which is anchored on two sides. In this example, we use the PyFEAP interface to set mesh parameters and to script graphics display.

   **The FEAP input deck**

   The *FEAP input deck* for this example describes a simple Cartesian mesh, \( n \) elements on a side:

   ```
   feap
   0 0 0 2 4 4
   block
   cart n n
   ```
The Python script

The first Python script uses the PyFEAP interface to load the deck with $n = 10$ elements on a side; run a modal analysis; and plot the first four modes.

```python
#!/usr/bin/env python
# Example of FEAP / Python interface using PyFEAP:
# Run a modal analysis in FEAP and plot the original and displaced node coordinates for the first mode using FEAP graphics.
from pyFeap import pyFeap
import numpy

# create an interface object with the path to all the example files
obj = pyFeap.pyFeap("/home/charlie/matfeap-0.8/python/example")
# Build a 10-by-10 block of material
params = {'n': 10}
obj.start('Iblock', params)

# Run a modal analysis
obj.cmd('mass', 'tang', 'subs,,10')

# Plot the first four eigenvalues
obj.cmd('plot,deform,,,0.1')
for i in range(1,5):
    obj.cmd('plot,wipe')
    obj.cmd('plot,eigv,%d' % i)
    print 'plotting eigenvalue of %d: ' % i
    print 'Press any key to continue.'
    raw_input()

# Wrap up
obj.quit()
```

Detailed description

The import in the beginning of Python script declares what Python modules to import into this script. We import our PyFEAP module as well as numpy here.
from pyFeap import pyFeap
import numpy

We need to first create a PyFeap object in order to start everything. The constructor of PyFEAP takes at least one argument, which is where the example FEAP blocks and Python scripts located.

```
obj = pyFeap.pyFeap("/home/charlie/matfeap-0.8/python/example")
```

The `start` command runs FEAP on the specified input deck. Notice that the parameter $n$ is not defined in the FEAP input deck; we pass it in as a parameter set. The parameters we passed in is a Python dictionary structure.

```
params = {'n': 10}
obj.start('Iblock', params)
```

After this mesh is constructed, the input deck tells FEAP to enter interactive mode with the `inte` command. When using the PyFEAP interface, interactive mode returns control to Python console.

Now we want to run an actual modal analysis. In the first script, We use FEAP's solvers for the analysis by running `mass`, `tang`, and `subs` commands as we would from the command line interface. The command to feed commands to the FEAP command line interface is `cmd`:

```
obj.cmd('mass', 'tang', 'subs,,10')
```

After computing the mode, we want to display the first four modes. We can accomplish this through a for loop and pause between each plot's display.

```
obj.cmd('plot,deform,,,0.1')
for i in range(1,5):
    obj.cmd('plot,wipe')
    obj.cmd('plot,eigv,%d' % i)
    print 'plotting eigenvalue of %d: ' % i
    print 'Press any key to continue.'
    raw_input()
```

When we are done with this input deck, we need to leave interactive mode and shut down FEAP. In the usual FEAP interface, we would type `quit` and then answer `n` to the question Continue with interactive input options for control? From the PyFEAP interface, we use the command:

```
obj.quit()
```
Plots from the script

2. Linear solves in Python

Our second example involves deformation of a block of material under a load. The effect is the same as running `tang, 1` in FEAP, but we do all the computations with Python, numPy and sciPy solvers.

The FEAP input deck for this example describes a simple Cartesian mesh, very similar to the mesh used in the previous example. The block of material is anchored on the left edge, and loaded along the bottom edge. This Python script uses the PyFEAP interface to load the deck with $n = 10$ elements on a side; compute the tangent, residual forces, and deformation; and plot the deformed structure.

```python
#!/usr/bin/env python
# Example of FEAP / Python interface:
# Run a simple small-deformation static displacement problem.
from pyFeap import pyFeap
import numpy

# Build a 10-by-10 block of material
obj = pyFeap.pyFeap("/home/charlie/matfeap-0.8/example/")
params = {'n': 10}
obj.start('Iblock2', params)

# Form the tangent matrix
```
K = obj.tang()
# Form the residual force vector
R = obj.resid()
# Compute a Newton update
du = numpy.linalg.solve(K.todense(), R)
# Set the displacement vector
obj.setu(du)

# Plot the results
obj.cmd('plot', 'defo', 'mesh', 'boun', 'load', 'end')

# Quit
print 'Press any key to exit.'
raw_input()
obj.quit()

**Detailed description**

As before, we begin by loading the input deck and changing the output and plotting parameters. We then extract the tangent stiffness $K$ and the residual $R$ from FEAP. These commands form the tangent and residual; there is no need to execute separate `tang` and `form` commands. Once the tangent and residual are constructed, we solve the linear system, and then write the resulting displacement vector back into FEAP's memory.

Once we write the displacement vector into FEAP's memory, it is available for other routines from within FEAP. In particular, when we run the `plot` line, the deformed mesh reflects the updated displacements.

As before, the last line shuts down the FEAP system.

**Plot from the script**

![Plot from the script](image)

**V. Future Work:**

Possible future work for the PyFEAP module is to further develop functionality from FEAPMEX that was never implemented in MatFEAP. It would be ideal to implement the utility C functions in FEAPMEX to PyFEAP, just to complete the whole functionalities of this Python interface. Ideally, the translation from C functions to Python module would be compatible since we used Cpython for
everything; it would make perfect sense that those compiled C functions can be imported to Python and thus extend the functionality of current PyFEAP.