nanoHUB_remote

Using the nanoHUB web API with Python

Suppose you want to run the Composite Laminate Analysis tool (complam) using the nanoHUB web API. Perhaps you wish to write a Python script that runs the tool with several different inputs to analyze the results, and you don't want to click through the GUI for each run.

You need to run the tool in a GUI session at least once in order to identify the inputs and outputs that you want to specify and capture. For the complam tool you examine the GUI inputs shown here.

If you want to vary the Longitudinal Young's Modulus and the Transverse Young's Modulus (in plane), make note of the labels that these input values have in the GUI (the text labels next to the input boxes): **E1** and **E2**, respectively, in this case.

Note the labels of the output values that you want to record, as shown in this figure:
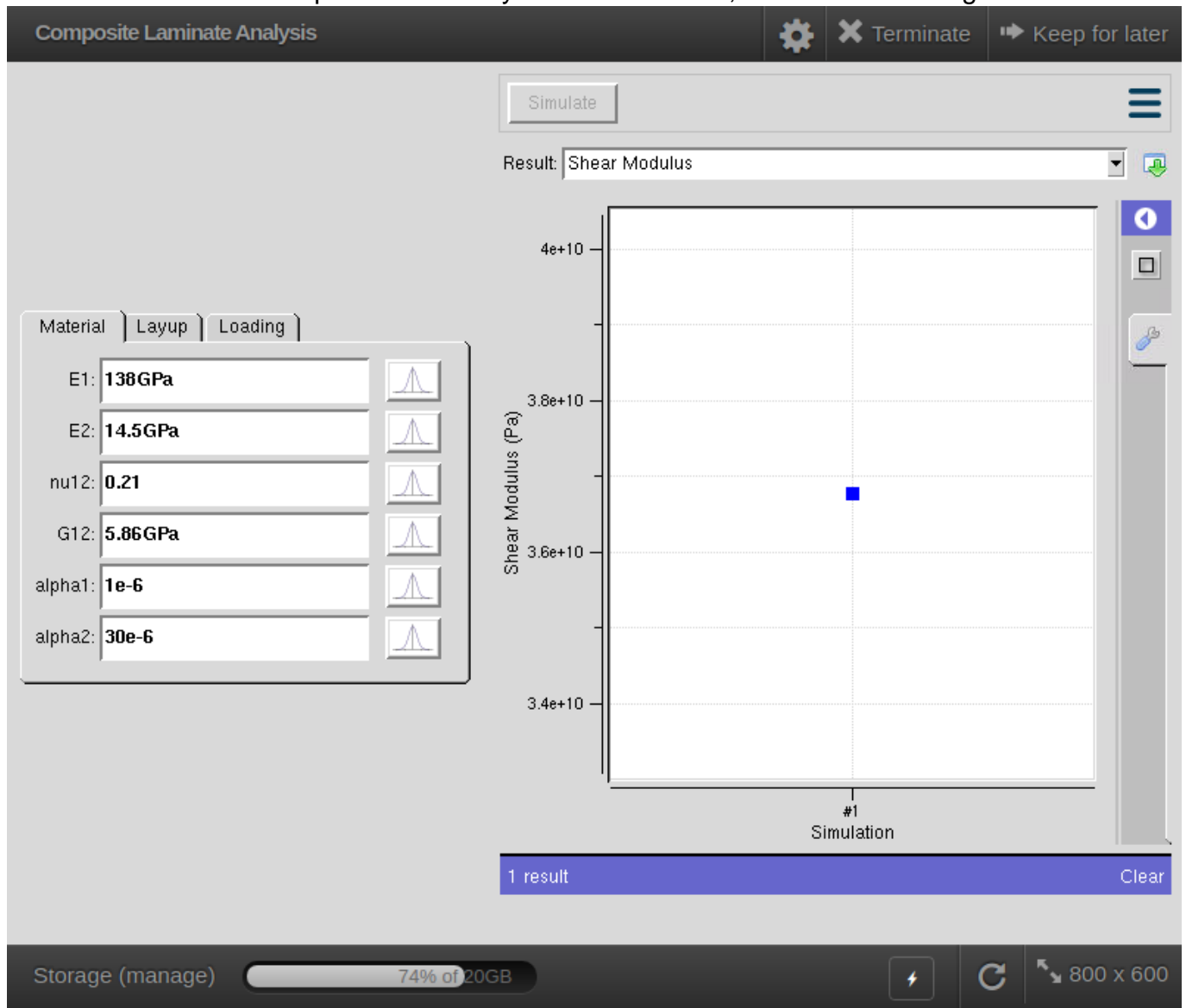
For example, the shear modulus calculated by the complam tool has the label **Shear Modulus** in the result selector.

Let's look at two methods for running the complam tool via the nanoHUB web API. The first method is difficult; it requires the user to know the details of XML files produced by Rapture tools. The second method hides all the XML and API endpoint details behind a simple Python API.

Prerequisites

In order to use the nanoHUB web API you need a nanoHUB account. You also need to create a web app.

The hard way

The nanoHUB web API uses standard HTTP GET and POST methods. The following Python code demonstrates most of the steps required to use the web API to run the complam tool.

```python
from urllib import urlencode
from urllib2 import urlopen, Request, HTTPError
import sys, json, time

url = r'https://nanohub.org/api'
app = 'complam'
sleep_time = 1.5

def do_get(url, path, data, hdrs):
    """Send a GET to url/path; return JSON output"""
    d = urlencode(data)
    r = Request('{0}/{1}?{2}'.format(url, path, d) , data=None, header
s=hdrs)
    try:
        u = urlopen(r)
    except HTTPError as e:
        msg = 'GET {0} failed ({1}): {2}\n'.format(r.get_full_url(), \
                                                    e.code, \
                                                    e.reason)
        sys.stderr.write(msg)
        sys.exit(1)
    return json.loads(u.read())

def do_post(url, path, data, hdrs):
    """Send a POST to url/path; return JSON output"""
    d = urlencode(data)
    r = Request('{0}/{1}'.format(url, path) , data=d, headers=hdrs)
    try:
        u = urlopen(r)
    except HTTPError as e:
        msg = 'POST {0} failed ({1}): {2}\n'.format(r.get_full_url(),
\
                                                    e.code, \
                                                    e.reason)
        sys.stderr.write(msg)
        sys.exit(1)
    return json.loads(u.read())

#
# 1. Get authentication token
#
auth_data = {
    'client_id':      # XXX Get this info when you create a web app
```

```python
    'client_secret': # XXX Get this info when you create a web app
    'grant_type': 'password',
    'username': # XXX Your nanoHUB username
    'password': # XXX Your nanoHUB password
}
auth_json = do_post(url, 'developer/oauth/token', auth_data, hdrs={})
sys.stdout.write('Authenticated\n')

hdrs = {
    'Authorization': 'Bearer {}'.format(auth_json['access_token'])
}


#
# 2. Run the job
#
run_data = {
    'app': app,
    'xml': driver_xml
}

run_json = do_post(url, 'tools/run', run_data, hdrs)
session_id = run_json['session']
sys.stdout.write('Started job (session {})\n'.format(session_id))

#
# 3. Get job status and run file path
#
status_data = {
    'session_num': session_id
}
while True:
    time.sleep(sleep_time)
    status_json = do_get(url, 'tools/status', status_data, hdrs)
    if status_json['finished'] is True:
        break
sys.stdout.write('Job is finished\n')
time.sleep(sleep_time)

#
# 4. Retrieve the run file
#
runfile = status_json['run_file']
result_data = {
    'session_num': session_id,
    'run_file': runfile
}
```

```
result_json = do_get(url, 'tools/output', result_data, hdrs)
with open(runfile, 'w') as f:
    f.write(result_json['output'])
sys.stdout.write('Retrieved {}\n'.format(runfile))
```

This code is incomplete; it does not show the driver_xml string passed to the tools/run endpoint in step 2. The driver_xml string is an XML string that specifies the inputs to the complam tool.

Most nanoHUB tools use the [Rappture](#) toolkit to specify inputs and outputs. The complam GUI shown above is an example of an interface created by Rappture. When the user clicks Simulate in a tool session, the input values from the GUI are inserted into an XML file that describes the interface. This XML file, called the driver file, is read by a program that controls the tool. To see an example of the driver XML for the complam tool click here [driver.xml](#) (9 KB, uploaded by Benjamin P Haley 6 years 6 months ago).

For each set of inputs, E1 and E2, you need to modify the driver XML string, driver_xml, and pass it to the web API. The result produced by this Python code is another XML file, the run file, that contains the outputs calculated by the tool. You will need to extract those results from the XML (e.g. the shear modulus). The difficulties with this process include the following:

- How do I get the driver XML file if I am not the developer of the tool?
- How do I get the driver XML for another tool?
- Why do I have to parse XML at all?
- Why do I have to know the web API endpoint details?

For these reasons, we have developed an easier way to interact with the nanoHUB web API.

An easier way

The [nanoHUB remote](#) Python library allows users to run nanoHUB tools without knowing the web API endpoint details or the specifics of the XML inputs and outputs of a Rappture tool. To get the library use this command:

```
git clone https://github.com/bhaley/nanoHUB_remote
```

The following example shows how to use nanoHUB_remote to run the same complam example shown above. The auth_data dict is the same as in the first example.

```
from nanoHUB_remote import authenticate, get_driver, launch_tool, get_
results, extract_results
```

```
auth_data = {
    'client_id':     # XXX Get this info when you create a web app
    'client_secret': # XXX Get this info when you create a web app
    'grant_type': 'password',
    'username': # XXX Your nanoHUB username
    'password': # XXX Your nanoHUB password
}

# Authenticate; use headers in all subsequent steps
headers = authenticate(auth_data)

# The short name of the nanoHUB tool to run; this is the final stanza
of
# the tool URL (e.g. https://nanohub.org/tools/complam)
tool_name = 'complam'

# Input values; keys are the labels of the inputs in the GUI
tool_inputs = {
    'E1': '132GPa',        # Longitudinal Young's Modulus
    'E2': '12.9GPa',       # Transverse Young's Modulus (in-plane)
}

# Generate the XML driver to run the tool with our inputs
driver_json = get_driver(tool_name, tool_inputs, headers)

# Start the simulation
session_id = launch_tool(driver_json, headers)

# This is useful for debugging
print session_id

# Get the results when available
run_results = get_results(session_id, headers)

# The outputs we want; these are the labels in the result selector in
the
# tool GUI
outputs = ['Shear Modulus']

# Get the desired outputs
results = extract_results(run_results, outputs)
print results
```

The output of this program looks like this:

```
{'Shear Modulus': 35021400000.0}
```

Note that we didn't need to know the XML details (inputs or outputs) or the web API endpoints. This allows the user to focus on the science of the simulation.