Intro

Hello everyone,

Welcome to the Introduction to PySAM webinar.

Slide 2: This webinar is the first in the series of SAM webinars for 2019. We are recording this webinar, and will post a link to the recording and supporting materials on the SAM website.

Slide 3: We will leave about 20 minutes at the end of the presentation for questions and answers. If you have a question, please use the GoToWebinar control panel to type it as shown in these screenshots, and my colleagues will either type an answer, or read your question for me to answer during the Q&A session. We may unmute your line during the session so you can follow up by voice.

Slide 4: System Advisor Model is a free software for modeling the performance and economics of renewable energy projects. It's developed by NREL with funding from DOE. Supported platforms include Windows, Mac, and Linux. We release one or two new versions per year. We make the sam simulation core, or SSC, library available through the Software Development Kit (SDK). For user support, we offer a Help system, Documents on website, Online forum, and Website Contact form.

Slide: PySAM Intro

My goal today is to introduce PySAM, SAM's improved python wrapper, a thin layer of code that translates a library's existing interface into a Pythonic interface.

This webinar is targeted for experienced python programmers who have some familiarity with Anaconda, Python packages, and SAM.

You'll be able to tailor renewable energy system simulations from your python scripts with better visibility, error reporting, and easy conversion between Python and the underlying SAM library. I'm Darice, a renewable energy system modeler at NREL. During this webinar, we'll quickly demonstrate how to set up a Utility Scale PV Project in SAM, PySSC, and PySAM, then focus on demonstrating how to get Started with PySAM by going through Installation Executing Models Importing from SAM Accessing compute_modules

Slide: SAM

I'll be modeling a utility scale PV project in this webinar, starting with how to do this in the SAM Desktop Application. The three-part process is the same regardless: set up inputs, execute simulation, extract outputs.

I'll be demonstrating on a Mac but the process is similar for Linux and Windows.

To create a simple model of a utility scale PV project, we'll select the PVWatts model from the list of technologies and combine it with the PPA single owner (utility) financial model. SAM provides default configurations for each combination of technology and financial models. Many of these default values are informed by technoeconomic analysts here at NREL and are updated regularly.

Inputs are set in their corresponding pages. On this Location and Resource page, we can choose a weather file. On the System Design page, we can change the azimuth and the ground coverage ratio, or gcr.

I'll run the model using all the defaults. Clicking Simulate executes the pvwatts then the single owner model. The Summary table displays outputs from both models, such as the annual energy from pvwatts and the PPA price from single owner.

Slide: PySSC

Running a simulation through Python involves the same process of inputs, execute, outputs. Previously, python users accessed the SSC algorithms through a wrapper called PySSC, which I will demonstrate quickly here. However, we recently released a new wrapper, called PySAM, which adds more functionality in python and is easier to use. The old PySSC wrapper is still bundled with the new PySAM wrapper.

SAM's code generator can be used to get the python script that will set up and run this pvwatts single owner case. The Generate Code function works for many different languages. Currently, the Python code produced is for PySSC but will be updated for PySAM in a future release.

I've made a folder for this webinar, so I'll save the files here. SAM exports a csv containing the dispatch factors time series, the PySSC file, the ssc library and its header file, and a python script with the same name as the case.

Looking at that code, the function data_create in the PySSC class returns a ssc data table into which you'd insert key-value pairs using the set functions such as data_set_string, data_set_number, etc. To run a pvwatts simulation with that data, you'd create the appropriate module with module_create and provide the data table to the module_exec function. Once you've completed the simulation, you'd extract output values using the get functions: data_get_string, data_get_number, etc. Then we insert key-value pairs for the single owner simulation into the same ssc data table and execute that. Finally, we can extract outputs, and free the data.

Running this script, we see that the results, specifically the annual energy and the PPA price, are the same as when run from SAM.

Slide: PySAM

Now I'll show how to run the same simple case in pysam. Default configurations are now accessible from PySAM. Opening up the Python interpreter within a terminal, I'll: - create a default pvwatts and single owner model - execute pvwatts - extract the outputs I'll need for the single owner model - execute single owner - examine the results.

```
import PySAM.Pvwattsv5 as pv
import PySAM.Singleowner as so

d = pv.default('PVWattsSingleOwner')
d.LocationAndResource.solar_resource_file = '/Applications/SAM.app/Contents/solar_resource/p

f = so.default('PVWattsSingleOwner')

d.execute()

f.SystemOutput.gen = d.Outputs.ac
f.TimeOfDelivery.system_use_lifetime_output = 0
```

```
f.execute()


d.Outputs.ac_annual
f.Outputs.ppa
```

Again, we have the same results. Now that we've seen what PySAM looks like, we'll learn in the remainder of this webinar:

1. how to install PySAM
2. how to create, modify and use all the renewable energy system models in SAM
3. how to import a technology and financial simulation from SAM, and finally, 4. how to search for and access any of the functionality in SSC

Slide: Installation

The PySAM package and documentation is built from the PySAM repository here. The package is listed on PyPi and Anaconda cloud. The documentation is found on readthedocs. Everything we'll discuss here is documented in the Getting Started guide on the website.

The requirement is 64-bit Python version 3.5 and above. No additional software is necessary. We support Linux, Mac and Windows.

We're going to start with how to install PySAM. I'll demonstrate how to get it started with a new Anaconda environment.

First we'll create and activate a new conda environment with Python 3.7.

```
conda create --name demo python=3.7
conda activate demo
```

Then there're two ways to download, either through Anaconda Cloud or the python package index. We'll start with the second method since it's easier.

```
pip install NREL-PySAM
pip list
```

We see it on our list of installed packages. Note the name has NREL prefixed to distinguish it from other packages with similar names. You may notice that an additional package was downloaded as a dependency. This stubs package contains type annotations that help Integrated Development Environments, or IDEs, analyze your code and provide autosuggestions, but more on that later.

Downloading from Anaconda cloud will be useful if you're opening up a project repo from an IDE and NREL-PySAM is in the requirements.txt. Then the IDE will see that the python interpreter is using a conda environment and it'll try to download all the packages in the requirements.txt file from Anaconda cloud. As

these requirements change, such as if new versions of PySAM are used, you may want to have your IDE update them directly. Before we install, I'll remove the packages:

```
pip uninstall NREL-PySAM NREL-PySAM-stubs
pip list
```

From Anaconda cloud, PySAM's currently only available for Python 3.7 for Mac and Windows. If you need other versions, please use pip instead. To download PySAM using conda install, you'll add the nrel channel to your list of conda channels from which to download packages. Now, let's add the channel to the demo environment specifically.

```
conda config --show channels
conda config --env --append channels nrel
conda config --show channels
conda install NREL-PySAM NREL-PySAM-stubs
conda list
```

And that's it! One thing to note is that if you're on a Mac, the version you can use depends on your osx version. Please take a look at the github issue linked here and see if that helps you.

Slide: Inputs

Then, of course, we'd like to explore and run some models. The list of models is available on the documentation page.

Each model's page provides information for the default configurations and the groups of variables. For those of you familiar with the GUI, the model-matching name in the SAM Desktop is provided here. We'll use the pvwatts single owner example to go through the new features of PySAM. We will

- set up autocompletion in PyCharm, an IDE
- access variables
- and decipher error messages

To set up autocompletion in PyCharm, we'll start by creating a new project in the pysam_talk folder I've prepared and getting it set up with our newly created conda environment. Not everyone uses PyCharm or an IDE, so this step is optional.

Once I've gotten the interpreter set up, I'll configure it to run the demo.py script, which I've created for this webinar and which will be posted alongside it.

Now, we should get autosuggestions for this pysam package.

I've imported the pvwatts model. I'll create a new instance with the new function, which will return an empty pvwatts model.

```
import PySAM.Pvwattsv5 as pv
```

```
model = pv.new()
```

And now as we explore this class, you'll see the group names pop up. Within each group are the ssc variables, whose type, label, options, units and other information are available in the documentation. For those familiar with the GUI, the grouping of each model's variables usually corresponds with the tab on which the variables show up. We can set these directly in the code by assigning floats, strings, dictionaries, tuples and lists. I'll switch to the interactive Python console to better demonstrate.

```
model.SystemDesign.azimuth = 3
```

Another way to access model groups and variables is through the setattr and getattribute functions.

```
model.SystemDesign.__setattr__('azimuth', 10)
model.SystemDesign.__getattribute__('azimuth')
model.__getattribute__('SystemDesign').__getattribute__('azimuth')
```

The data can also be assigned or exported as a dictionary of variables. I'll create a dictionary called systemDesign of all the systemDesign variables I want to set. Then I can use the assign method of the systemDesign group. The export method of the group will produce a dictionary.

```
systemDesign = {'azimuth':100, 'gcr': .5} (ground  coverage ratio)
model.systemDesign.assign(systemDesign)
model.SystemDesign.export()
```

In order to assign variables from multiple groups, we can provide a nested dictionary to the model itself.

```
resource = {'solar_resource_file': 'file.csv'}
```

I'll make a nested dictionary with the key as the group name and the value as a dictionary of variable-value pairs.

```
modelDict = {'SystemDesign': systemDesign, 'LocationAndResource': resource}
model.assign(modelDict)
model.export()
```

Now I'll assign these group variables to our model. When I export all the variables from the model, we see the same nested dictionary structure.

Slide: PySAM Errors

Errors will be reported as PySAM.errors, and potential sources include:

- type mismatch during assignment
- assigning variables that don't exist
- missing fields during simulation and other execution errors

We'll go over these types and how to identify the cause. The first is straightforward, as it occurs when you try to assign values directly and you'll get a TypeError.

The second error tends to show up when using the other value-assignment methods. For instance, I'll create a dictionary with 1. a value that isn't the correct type, and 2. a variable that doesn't exist in that model, and try to assign this to the SystemDesign group.

```
errorDict = {'azimuth': 'df', 'doesnotexist':1}
model.SystemDesign.assign(errorDict)
```

First there's this PySAM.error stating that it cannot load a matching function and suggests the two possible reasons for error: Either the `azimuth` parameter does not exist or is not the correct type. To figure out which error you've made, referring to the documentation is most helpful. In the pvwatts page, azimuth is specified to be a float.

Changing it to a float will fix the error.

```
errorDict = {'azimuth': 0, 'DNE':1}
model.SystemDesign.assign(errorDict)
```

Then we come to the second error, and we see that this variable does not exist.

The final type of PySAM.error passes error and log messages from the SAM library. So if we try to execute a simulation before we provide all the required data, we'll see a message with the first missing input variable.

Slide: Running Simulations

Now that we know how to work with inputs, let's set up that default utility scale pv project we demonstrated at the beginning.

A list of available default configurations is provided in each model's page on the documentation site, where each entry is a pairing with a financial model. The set of defaults are copied from the SAM repo, and will be the same as that in the latest SAM release, but may or may not be the same as the latest set being developed.

We're looking for the 'PPA single owner (utility)' model, which corresponds to PVWattsSingleOwner.

```
```

```
d = pv.default('PVWattsSingleOwner')
d.export()
```

Creating this default-configured model, we can see many input parameters have been populated by calling export. Just a few inputs are missing such as weather resource. Once we provide a path to a file, we'll execute the simulation, with verbosity set to 1 to output any log messages

```
d.LocationAndResource.solar_resource_file = 'resource.csv'
d.execute(1)
```

We see that it completes without error. Taking a look at the model now, we can go into the Outputs group and extract values we're interested in using `help` to get a list of all the available outputs.

```
help(d.Outputs)
d.Outputs.ac_annual
```

Slide: Linking up Simulations

You may wonder how do we run the single owner model following the pvwatts model? As I showed earlier, in PySSC, you reuse the same data table, so that all the input and output values from pvwatts will be accessible to single owner.

You may notice that with PySAM, there isn't a direct way to access that data as it's wrapped and protected by the PySAM class with its getters and setters. Then, of course, we can't simply pass that data table from pvwatts to single owner. At the moment, you have to assign the required inputs of the single owner from the outputs of pvwatts. Luckily there are usually only a few, but not catching all of them would be an easy mistake to make, so be sure to check the documentation.

I'm going to create a single owner model with the defaults configured to the same technology-financial pair: PvWattsSingleOwner.

```
import PySAM.Singleowner as so
f = so.default("PvWattsSingleOwner")
```

In the single owner model documentation, you can look through the inputs and see which ones depend on results from the technology simulation. The financial model requires the energy production time series, the system_capacity and whether or not we're running lifetime simulations. To transfer the energy generation profile from pvwatts to singleowner, we simply assign that value. We'll check that the system_capacity values are equal. We have to specify that we're not running a lifetime simulation. If we were to run lifetime simulations, we'd also make sure the analysis_period values are the same.

```
f.SystemOutput.gen = d.Outputs.ac
f.SystemOutput.system_capacity == d.SystemDesign.system_capacity
f.TimeOfDelivery.system_use_lifetime_output = 0
```

A major difference between simulations in SAM and in the SDK, including PySSC and PySAM, regards dependencies between inputs. In SAM, the values for some inputs are actually calculated to assure consistency among all the inputs. These calculations help link up dependencies among inputs to a single model, as well as dependencies among inputs in two different models. If you modify an input that is upstream to one of these calculations, you may need to write additional code to ensure values for these inputs are correctly assigned.

```
f.execute()
f.Outputs.export()
```

Slide: Importing from SAM

So to tie together all the pieces we've discussed so far: setting up and using PySAM, modifying models and decoding error messages, and linking sequential models, we're going to import a SAM GUI case into python using the code generator function.

Going into the desktop application here, I've got a pvwattsv5 single owner case. What we really need is just all the inputs we've defined in the GUI as a json file.

To start, we'll need to import Pvwattsv5, Singleowner and PySSC, as well as json. Then we need to make a PySSC instance. Now we'll look at demo.py.

```
import json


import PySAM.Pvwattsv5 as Pvwatts
import PySAM.Singleowner as Singleowner
from PySAM.PySSC import *


ssc = PySSC()
```

We'll open the json file to load the data into a dictionary, and use a function to create that ssc data table I mentioned earlier, fill that with all the pvwattsv5 and single owner inputs, and return that data table. We'll need one per model.

```
with open("untitled.json") as f:
        dic = json.load(f)
        pv_dat = dict_to_ssc_table(dic, "pvwattsv5")
        so_dat = dict_to_ssc_table(dic, "singleowner")
```

Each PySAM class is a model-specific wrapper for a ssc data structure, so to feed our already-created data into our pysam models, we'll need the function wrap.

```
pv = Pvwatts.wrap(gs_dat)
        so = Singleowner.wrap(so_dat)
```

As the pysam class will automatically delete the data when it itself is deleted, be sure to not call data_free on the original data as you would have done using PySSC directly. Doing so will result in a error.

Now we can run the pvwatts model, which will complete without messages. Then we transfer over any required outputs from the pvwatts simulation, and execute.

```
pv.execute()
so.SystemOutput.gen = pv.Outputs.gen
so_model.TimeOfDelivery.system_use_lifetime_output = 0
so.execute()
```

So that completes the section of our talk on using PySAM technology and financial models. We've covered how to create a blank, a default, and an imported model.

Slide: Compute Modules

Our final section will be about modules in the ssc library which are not associated with a PySAM class. What kinds of functionality are included here? Some of the functionality underlying a simulation can be accessed as stand-alone modules, such as coefficient calculators for pv module and inverter models or weather file processors. Rather than go through all the options available, I'll go through how to find these modules and how to use them.

Imagine we'd like to fit pv module information found in the manufacturer data sheet to the CEC 6 parameter model. In the GUI, it's this page here, where you can enter fields such as the module type, the maximum power point voltage, the maximum power point current... the temperature coefficients, and fit these to the model from which the IV curve can be calculated and plotted. Can we access this function from Python?

If the module is a compute module, then yes. A compute module is a chunk of code that calculates results from a set of inputs. Compute modules can be combined to create a power system model, for example, combining pvwattsv5 and singleowner, or run individually, like running 6parsolve to calculate single-diode PV parameters.

There are a couple ways to check if this feature is indeed a compute module. The most direct way is to look at the associated UI page text file, but that's outside the scope of this webinar. Instead we'll look at the SSC api, and also use the SDKTool.

The list of all compute_modules available in the ssc library is listed in the sscapi.cpp file as module_entry_info's. There're models for solar water heating, geothermal, wind, many varieties of CSP, among other technologies.

To find the code associated with each of these compute_modules, simply copy the name after 'cm_entry_' and open the file 'cmod_' with the name appended, '.cpp'. So if we wanted to look at the 6parsolve code, we'll open `cmod_6parsolve.cpp` in the ssc folder in the ssc repo.

The var_info table contains input information such as the data type, a short description, the units and whether or not a variable is required. A short description of the compute module can be found next to DEFINE_MODULE_ENTRY.

Another way to explore available compute_modules is through the SDKtool, which is automatically installed when you install SAM.

Clicking through the available modules, we can see the inputs from the var_info table.

Once we've found the compute_module name and input information we're looking for, we can run a `6parsolve` module using a function from PySAM.PySSC.

So in my demo script, I have a dictionary containing containing a tech_model and a financial_model entry, where financial_model must be `None` if not required, as well as all the input parameters to the models.

```
from PySAM.PySSC import *
datadict = {
"tech_model": '6parsolve',
'financial_model':'none',
'celltype': 'multiSi',
'Vmp': 30,
'Imp': 6,
'Voc': 37,
'Isc': 7,
'alpha_isc':0.004,
'beta_voc':-0.11,
'gamma_pmp':-0.41,
'Nser': 60,
'Tref':25}
ssc_sim_from_dict(datadict)
print(datadict)
```

This function takes that dictionary and runs the appropriate models and exports the output into the same dictionary.

Running this script, I see that the dictionary has been populated with the output variables.

Note that all cmods, which includes all the technology and financial models wrapped by PySAM such as PvWatts and Singleowner, can be run using this method.

Slide: Outro

In this presentation, I hope you've learned how to install PySAM, how to create, modify and run models using a variety of methods, and how to find functionality you may be interested in from the open-source code.

Thank you for your time and attention. I look forward to collaborating with you.