

cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation



Open-File Report 2011–1062

Cover photograph by Michael N. Fienen

cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

By Michael N. Fienen, Thomas C. Kunicki, and Daniel E. Kester

Open-File Report 2011–1062

**U.S. Department of the Interior
U.S. Geological Survey**

U.S. Department of the Interior
KEN SALAZAR, Secretary

U.S. Geological Survey
Marcia K. McNutt, Director

U.S. Geological Survey, Reston, Virginia 2011

For product and ordering information:
World Wide Web: <http://www.usgs.gov/pubprod>
Telephone: 1-888-ASK-USGS

For more information on the USGS—the Federal source for science about the Earth, its natural and living resources, natural hazards, and the environment:
World Wide Web: <http://www.usgs.gov>
Telephone: 1-888-ASK-USGS

Suggested citation:

Fienen, M. N., Kunicki, T. C., and Kester, D. E., 2011, cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation: U.S. Geological Survey Open-File Report 2011–1062, 22 p. [<http://pubs.usgs.gov/of/2011/1062>]

Any use of trade, product, or firm names in this publication is for descriptive purposes only and does not imply endorsement by the U.S. Government.

Although this report is in the public domain, permission must be secured from the individual copyright owners to reproduce any copyrighted materials contained within this report.

Contents

Abstract	2
Introduction	2
Purpose and Scope	2
Preliminaries	3
Python	3
EC2 Command-Line Tools	4
The AWS Console	4
Communicating with Nodes	4
cloudPEST Concepts	5
Security Groups	5
EC2 Key Pair	5
Amazon Machine Image (AMI)	5
Instance	6
Instance Types and Pricing	6
Stopping and Terminating	6
Session	6
Instructions	6
General Strategy	7
Summary of Steps for a Session	7
Details and Rationale for Session Steps	7
Create Master Image	7
Create Slave Image	8
Create Service on Slave	8
Windows Services	10
Create Both Images	10
Launch a Master Instance (optional)	10
Start SSH Tunnel (optional)	10
Configuring a Port-Forwarding Linux Machine on the Cloud	11
Establish the SSH Tunnel with Port Forwarding on Command Line	11
Establish the SSH Tunnel with Port Forwarding Using PuTTY	11
Launch beoPEST on the Master	12
Launch Slave Instances	12
Terminate Slaves and Master At Completion	13
cloudPEST Classes and Functions	13
Classes	13
images:	13
instances:	13
master:	14

cloudPEST Functions	14
query_images()	14
query_instances(*instance_id)	14
run_instances(ami_id, instance_count, keyname, group, *insttype, *cnodes, *availzone)	15
start_instances(instance_id)	15
stop_instances(instance_id)	15
terminate_instances(instance_id)	16
cloudPESTclient Functions	16
parseLine(line)	16
readFTPparfile(infile)	16
retrieveFTPparfile(FTPaddress, FTPdir, detailedParInfile)	17
readRUNparfile(infile)	17
StartNode(bPcasename, bPexec, bPhost, bPport, bPCurrentNode)	17
node_starter.py()	18
Example Application	18
Acknowledgments	19
References Cited	19
Glossary	21

Figures

1. Diagram illustrating the arrangement of computers for setting up initial images	8
2. Diagram illustrating the arrangement of computers for production runs. The multiple locations (1 and 2) in the diagram highlight the fact that each batch of virtual machines run may be hosted at a different physical location within the cloud network of servers	9
3. Dialogue box for the Non-Sucking Service Manager (NSSM) showing example input for starting the node_starter.py service	10
4. Dialogue of the SSH tunnel section of the PuTTY configuration	12
5. Parameter file with details of FTP downloading used by node_starter.py	16
6. Parameter file details of paths and executables needed to start beoPEST in slave mode, used by node_starter.py	18
7. Example implementation of the cloudPEST functions	19

cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

By Michael N. Fiener, Thomas C. Kunicki, and Daniel E. Kester

Abstract

This report documents cloudPEST—a Python module with functions to facilitate deployment of the model-independent parameter estimation code PEST on a cloud-computing environment. cloudPEST makes use of low-level, freely available command-line tools that interface with the Amazon Elastic Compute Cloud (EC2^{TM1}) that are unlikely to change dramatically. This report describes the preliminary setup for both Python and EC2 tools and subsequently describes the functions themselves. The code and guidelines have been tested primarily on the Windows^{®2} operating system but are extensible to Linux^{®3}.

Introduction

Cloud computing, in which Internet data-transfer protocols are used to connect many rented computers at data centers distributed around the world, facilitates large-scale numerical computing for parameter estimation applied to groundwater simulations (Hunt and others, 2010). In parameter estimation, for example using PEST (Doherty, 2010), the most computationally expensive aspect is to calculate the Jacobian matrix of sensitivities, which guides the gradient-based inversion. Calculation of the Jacobian matrix requires at least one run per parameter using the perturbation method but each model run is independent from the other; therefore, the problem is “embarrassingly parallel” and well suited for parallelization such as using Parallel PEST (Doherty, 2010) and beoPEST (Schreüder, 2009). Both PEST and beoPEST are available online at <http://www.pesthomepage.org>, and both use a network of “slave” computers to distribute the computational load. The ability to launch a virtually limitless number of slave computers on the cloud creates great opportunities for calculation of the Jacobian matrix required many times in the linearized least-squares parameter-estimation approach used in PEST.

beoPEST is a new parallel run management code that is well-suited for cloud computing. A key advantage to using beoPEST over the traditional run-management file approach of parallel PEST is that much of the computational load of input/output (I/O) and file construction is pushed out to the slave machines, which mitigates a bottleneck at the master processor that has been encountered in highly parameterized problems. Another advantage to beoPEST is that it does not require a static network with shared privileges and static names or Internet protocol (IP) addresses; only the master needs to have a static address and an open transmission control protocol/Internet protocol (TCP/IP) port. The slaves are started and send signals to the master indicating they are available to accept model runs using either TCP/IP or message passing interface (MPI) protocols; in this report, we focus on the TCP/IP protocol.

Another impediment to large-scale deployment of PEST problems on the cloud is the lack of tailored scripting functions to automate the launching, monitoring, and termination of the large number of slave machines required. Users are able to accomplish these tasks through websites or other manual means, but the automation aspect is important for deploying cloud-based PEST runs on a large scale. cloudPEST is a module of Python classes and functions designed to address this deficiency.

Purpose and Scope

The goal of this report and of cloudPEST—the software module this report describes—is to provide a utilitarian, relatively simple, and customizable module made up of functions to allow practitioners to rapidly deploy PEST runs in parallel using beoPEST on the cloud. The Elastic Compute Cloud from Amazon Web Services of Amazon.com, Inc. (AWS^{TM4}), hereafter termed “EC2,” was selected as the platform on which to

¹“Amazon EC2” is a trademark of Amazon, Inc. in the United States and/or other countries.

²“Windows” is a registered trademark of Microsoft Corporation in the United States and other countries.

³“Linux” is a registered trademark of Linus Torvalds in the United States and other countries

⁴“Amazon Web Services (AWS)” is a trademark of Amazon, Inc. in the United States and/or other countries.

deploy cloudPEST; other cloud-computing vendors are available and as they develop application programming interfaces (API) similar to that on EC2, cloudPEST could be adapted to work with them.

The scope of this work is to provide a baseline of information and functionality to allow users to begin harnessing the power of the cloud. Python was chosen for the module because it is freely available, open-source software-programming language that is platform independent. Python is a high-level language, meaning that the code is concise and compiles at run time, obviating the need for a compiler for each platform. The functions are really scripts wrapped around the EC2 command-line tools. The cloudPEST tools were developed specifically for EC2 owing to availability of an existing application programming interface (API) toolset on which to build scripts for large-scale deployment. Some of the protocols in this report are transferable to other vendors, but the main API-related functions are specific to EC2.

The EC2 command-line tools make it possible for a great amount of flexibility, control, and customization of using EC2. Low-level, command-line tools were chosen because they are likely to remain available even as other aspects of the EC2 services change rapidly. The intent of cloudPEST is not to provide access to all of the functionality of the EC2 tools; cloudPEST is intended to provide the minimum level of interaction required to deploy parallel PEST runs on the cloud using beoPEST. The interested user is encouraged to consult the EC2 command-line tools documentation and other online references regarding the AWS Console.

The decision was made to develop platform-independent tools, which would allow users the flexibility to work on Windows or Linux platforms. The choice between Windows and Linux on the cloud is based on similar considerations as for local use. Each platform has advantages and disadvantages: for example, networking on Linux is more advanced whereas many modeling-support tools, particularly for groundwater modeling, are supported principally in Windows. Users should also consider cost differences when making a decision regarding which platform to use.

This report is aimed at intermediate to advanced users with some aptitude toward adapting programs or original coding. No graphical user interface is provided for the cloudPEST module; hence, the use of cloudPEST is dependent on the user writing scripts as text files and running them from the command line. However, it is hoped that many applications will be able to use of the basic scripts provided, without significant modification.

Preliminaries

The setup of software required to run cloudPEST is described in the following subsections, including Python and EC2 tools. The versions used here are Python 2.6.5 or later and EC2-tools version 2009-11-30. Some basic information about preferred computer configuration for cloudPEST also is provided.

While the computational load in beoPEST is transferred to the cloud when using cloudPEST, a local computer also is used to coordinate the process. The cloudPEST module typically is run both locally (to start and stop slaves and to run the master) and on the cloud (to start the slaves and initiate their communication with the master). Depending on local firewall restrictions, port forwarding may be required to establish a secure connection through the firewall. Details of these operations are provided throughout this report.

Python

Several options exist for installing Python 2.6.5. Python can be downloaded from <http://www.Python.org/download/>. The install is accomplished using either an installer (Windows and OSX) or compiled from source (Linux, for example). After installation, the install folder (typically C:\Python26 on Windows) must be added to the system PATH so it can be called from the command line. Other options are bundles such as the Enthought Python Distribution (<http://www.enthought.com/products/getepd.php>) or Python(x,y)

4 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

(<http://www.Pythonxy.com>). With the bundled approach, many useful packages are included that must otherwise be added later. Python (x,y) is free for use on Windows while Enthought is not free (unless used in academia). Adding packages is most easily accomplished using `easy_install` (<http://pypi.Python.org/pypi/setuptools>). Python should be installed on all computers being used with cloudPEST, including all Amazon Machine Images (defined below).

EC2 Command-Line Tools

The EC2 command-line tools are used on a local machine to interact with EC2. As a result, these tools and associated files need only be downloaded locally in order to control the machines on the cloud.

The EC2 command-line tools can be downloaded from <http://tinyurl.com/ec2-api>. After downloading the files, a couple of modifications to the PATH and other environment variables need to be made, as discussed in detail later in this section. A security connection also must be made with AWS; the security connection allows for secure communication with computers on the cloud. Detailed instructions for OSX⁵ can be found at: <http://tinyurl.com/ec2-osx>, and similar instructions for Windows are found at: <http://tinyurl.com/ec2-win>. Users can access and download the X.509 Certificate, described in the installation instructions, from the user account area at <http://aws.amazon.com/account/>, by following the link to "Security Credentials."

The AWS Console

Much of the functionality that can be accessed through the EC2 command-line tools and, by extension, through cloudPEST also can be accessed through a graphical user interface (GUI) in a web browser, called the AWS Console. Initially setting up an Amazon Machine Image (AMI) is most efficient through the console. Furthermore, as a session is underway, the console allows for quick checking on the status. Note that when using the console, automatic refreshing of the page can be a bit slow, so frequent manually refreshing either of the entire page or using the "refresh" button is recommended. The console is initially accessed at <http://aws.amazon.com> from which there are links to access the console and also to set up and manage other aspects of the Amazon AWS account. The AWS Console shows conclusively how many instances are running. In other words, even if the cloudPEST tools lose communication, the AWS Console always is available with a definitive list of active instances and the ability to stop them and confirm they are stopped.

Every running instance is incurring an hourly cost, so at the completion of any work using AWS, a user should view the AWS Console to confirm that all instances are stopped. If, for any reason, the cloudPEST APIs have failed to stop instances, the user can stop them through the AWS Console. Failure to take this step could result in unintended costs for resources not being used.

Communicating with Nodes

The standard means of communicating remotely with a computer running the Windows operating system is through the Remote Desktop Protocol (RDP). This manual, graphical interface is useful for configuring a Windows image but is not a practical solution for running a distributed parallel-computing implementation. Unfortunately, there is not a built-in protocol on Windows to facilitate the necessary remote control. As a result, configuration of an initial image node on Windows is accomplished through RDP, and slaves are started using Windows services (described later) to launch beoPEST and initiate communication with master machines.

On Linux, the default method of communicating with nodes is through secure shell (SSH). This protocol is lower-level than RDP (it is primarily a command-line interface) so scripting of commands to the nodes is

⁵If using OSX, following the instructions for the environment variables `EC2_PRIVATE_KEY` and `EC2_CERT`, note that the entire path for the associated files must be provided if the user wants to use the EC2 command-line tools outside the installation directory.

much easier. It may be impractical to start slaves using SSH so daemons may be used similar to Windows services. Initial testing was conducted using daemons as a service with beoPEST on the cloud on Linux instances, but a complete test case has not been performed.

cloudPEST Concepts

The cloudPEST library is predicated on certain behavioral assumptions about how EC2 resources are used. There are many options available, and other approaches certainly are possible and are likely to have advantages over the approach presented here. A few concepts are explained here, the understanding of which is necessary to implement cloudPEST.

Security Groups

Each user must establish security groups (most easily done through the console) to configure which ports are open for communication. In case of attacks, it is inadvisable to open all ports for incoming traffic. For cloudPEST, it is only necessary to open a port for SSH (typically 22 (the standard) or 2222 (similar to the standard but marginally less likely to be attacked)), a port for RDP access (port 2289 is the standard), and a port for beoPEST. During testing of cloudPEST, the port 4040 was opened for beoPEST. This choice is arbitrary, but using a high number (greater than 4000) is less likely to interfere with other port usage on installed programs.

Opening ports on the security group does not mean all instances will have an open port through their firewalls. In fact, when using Windows, the user must open the beoPEST and SSH ports manually in the firewall as part of configuring a base image.

EC2 Key Pair

A key pair is required to associate a particular user account with an instance. This is a security measure to ensure that the connection between the cloud and the accessing computer is valid and associated with a specific account. New key pairs can be created at the console or using the command-line tools (a function in cloudPEST also allows this). When initiating (running) an instance from an AMI (both terms are described below), the user must specify a key pair by name to be used for that instance.

Amazon Machine Image (AMI)

An Amazon Machine Image (AMI) is a virtual machine configuration created either by Amazon (a base image), created by a user, or selected from a public library of machine images. An AMI may be cloned as many times as desirable. AMIs are identified by alphanumeric codes starting with "ami-#####". An AMI can be created from any running instance and, once created, it will be an image reflecting the exact configuration of the instance (for example, files, software, network configurations) at the time of its creation. While it is possible to create an AMI using command-line tools, it typically is easiest to create an AMI in the console by clicking on the active instance desired to be the base for the AMI and selecting Instance Actions > Create Image (EBS AMI) from the menu bar in the browser.

A user can **run** an instance based on an AMI. The term "run" has a very specific meaning in this context, which is that running an instance from an AMI means a brand-new virtual machine will be instanced that is a clone of the AMI configuration indicated. Regardless of the contents of other instances run from an AMI, each clone instance reflects the configuration of the AMI at the time of its creation.

An important consideration when building an image is that AMIs and instances must be of the same architecture. For example, various instances have both different architecture (64-bit Windows vs. 32-bit Windows

6 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

vs. 64-bit Ubuntu Linux, for example) and different specifications (for example, processing power and memory). Instances can change specifications from the AMI used as their clone base, but cannot change architecture. For example, an AMI based on a previously existing 32-bit Windows instance cannot be used to run a 64-bit Windows instance. An additional consideration pertains to image size. For example, the maximum size that can be run on 32-bit Windows is `c1.medium`, which is incompatible with 64-bit Windows under which the smallest image is `c1.large`. Therefore, careful consideration should be made when choosing which the architecture from which to build an AMI.

Instance

An instance is a clone of an AMI representing a specific virtual machine that can be accessed and used for computation. Many instances can be spawned or cloned from an AMI. Instances are identified with alphanumeric codes starting with `"i-#####"`.

A user can either **run** an instance, as described above, or can **start** an instance. The difference is, when **starting** an instance, rather than making a new clone of an AMI, an instance that was created using **run** earlier and later was stopped is re-started using **start**. When an instance is re-started, the configuration of files includes any modifications made to the image before it was stopped, so each instance may reflect changes made from the time it was launched from an AMI.

Instance Types and Pricing

The types of instances available at EC2 occasionally change. A complete list of available instances is maintained at <http://aws.amazon.com/ec2/pricing/>. The discrepancy between Linux and Windows instance pricing is based on the need to pay for Windows licensing when using Windows instances. New instance types are added periodically with different configurations of memory and processing capabilities.

Stopping and Terminating

Another important distinction must be made between “stopping” and “terminating” an instance. If a user stops an instance, charges will be suspended the instance will be parked as if a local computer was shut down and is waiting to be re-started. Some charges for the storage of the image files may be incurred. If an instance is terminated, it is destroyed and all of the data on it are lost forever. So, stopping an instance is a way to park it for future use, while termination is permanent and should be done at the end of a session.

Session

In the context of this report, “session” means a period of time in which virtual cloud machines are started, parallel runs are performed with them, and they are shut down. Activities during a session include starting the slave machines (instances), running beoPEST (and the model) on them, checking their status, and shutting them down.

Instructions

Broadly speaking, there are two approaches to managing nodes on the EC2. One is to configure, start, and stop machines using the AWS Console, exclusively and the other is to use the APIs in this module for machine management. The remainder of this section is general and discusses the approach regardless of whether the APIs in this module are used. In the discussion of functions and classes later in this section, more details are discussed specific to the APIs.

General Strategy

In this section, we describe the general strategy that can be implemented, using either Windows or Linux, to start on the cloud. First is a summary list of steps, followed by a detailed list including a description of the rationale behind the selection of these specific steps. We emphasize here that this approach is not meant to be the only approach, but rather is an example. Individual users may well create implementations that work better for their own applications. Nonetheless, we have found success with the following approach and believe it may work well for many general applications. Figure 1 illustrates the initial configuration of cloud resources, discussed in steps 1–4 of the following list. Figure 2 illustrates running a project after configuration, discussed in steps 5–9 of the following list.

Summary of Steps for a Session

1. *Create Master Image*: Custom software and model files must be installed, typically from a File Transfer Protocol (FTP) site. Each master image is configured as a base AMI illustrated in figure 1.
2. *Create Slave Image*: Custom software and all model files must be installed, typically from an FTP site. Each slave image is configured as a base AMI illustrated in figure 1.
3. *Create Service on Slave*: The service installed on the slave is created to automatically launch the slave version of beoPEST when instances are started.
4. *Create Both Images*: Private images are created as Elastic Block Storage (EBS)-backed images so that clones can be launched.
5. *Launch a Master Instance*: A single master instance is launched, and the IP address is noted and uploaded to the configuration file that will be used by the slave service.
6. *Start SSH Tunnel (optional)*: If running the master on a local machine (advised), establish an SSH tunnel to the cloud master instance and forward the beoPEST port from the cloud master to the local machine. This is necessary to provide secure communication among master and slave machines without the need to open a port for general access through the local (not on the cloud) firewall.
7. *Launch beoPEST on the Master*: If using port forwarding, this launch will be local.
8. *Launch Slave Instances*: The number of slaves launched depends on the requirements of the problem, account details, and cost considerations. In any case, launching in batches of 20 or less is advised. When the slaves start, they may retrieve configuration files from an FTP site.
9. *At Completion, Terminate Slaves and Master or Port Forwarder*: All instances can be safely terminated provided that all results are either stored on a local machine (through port forwarding of the master) or transferred off the cloud.

Details and Rationale for Session Steps

Create Master Image

First a master image should be created. This image typically is created from a general, public, base image provided by EC2. The first step is to select a base image and to start an instance of it for configuration and, ultimately, to save (create) as a new image. This is most easily accomplished through the AWS Console. All custom software must be installed (for example, Python, numpy, beoPEST, and all model files) on this instance

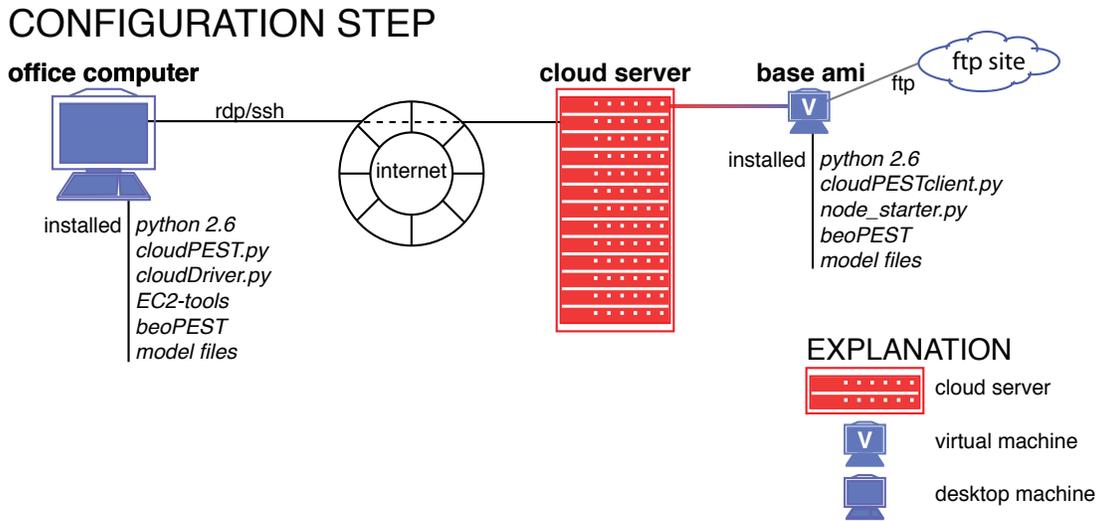


Figure 1. Diagram illustrating the arrangement of computers for setting up initial images.

using either RDP on Windows or SSH and curl on Linux. The time it takes to copy a large folder (even when compressed) of model files and executables can be long, so it is advised to download them once when creating the general image rather than transferring the files at run time. This way, any instance run based on this image will already have the files in place. On Windows, using RDP to access the machine, FTP (through a web browser or otherwise) is the recommended transfer method as using RDP directly to map drives will not handle the large files typically needed for models.

Create Slave Image

A similar suite of installations and file transfers as on the master image generally will be required on the slave image. Copying all model files to this image is the fastest way to deploy the files, as each instance started from the slave image will be configured with the entire suite of files. Another critical issue on Windows images is to disable renaming of the computer, which requires a reboot as part of the EC2 virtualization and instantiating protocol. Taking advantage of the TCP/IP communication protocols, the Windows computer name does not serve an important purpose, so while each slave instance will be started with a unique IP address, each will have the same name (based on an encoding of the IP address assigned to the image at the beginning). This naming protocol would cause problems if a Windows network was to be created, but such a step is unnecessary here. Disabling rebooting requires editing the file

`c:\Program Files\Amazon\Ec2ConfigService\Settings\config.xml` and changing the state for `Ec2SetComputerName` from `<State>Enabled</State>` to `<State>Disabled</State>`. The editing of this file only needs not be performed on the slave image—not on the master image. Preventing renaming obviates the reboot and also means the instances are available for use faster than if rebooting is allowed.

Create Service on Slave

A service is a program that runs in the background at the operating system root and can be configured to start automatically when a machine is powered on (or, in our case, an instance is launched). On Unix, services are called “daemons.” We use services with cloudPEST so that slave machines automatically start beoPEST in slave mode at launch, obviating the need for user intervention. Starting a service is platform specific and requires several steps. The rationale for using services to automatically launch beoPEST on slave instances is based on several factors. First, the alternative would require establishing an SSH connection to each slave

PRODUCTION RUNS

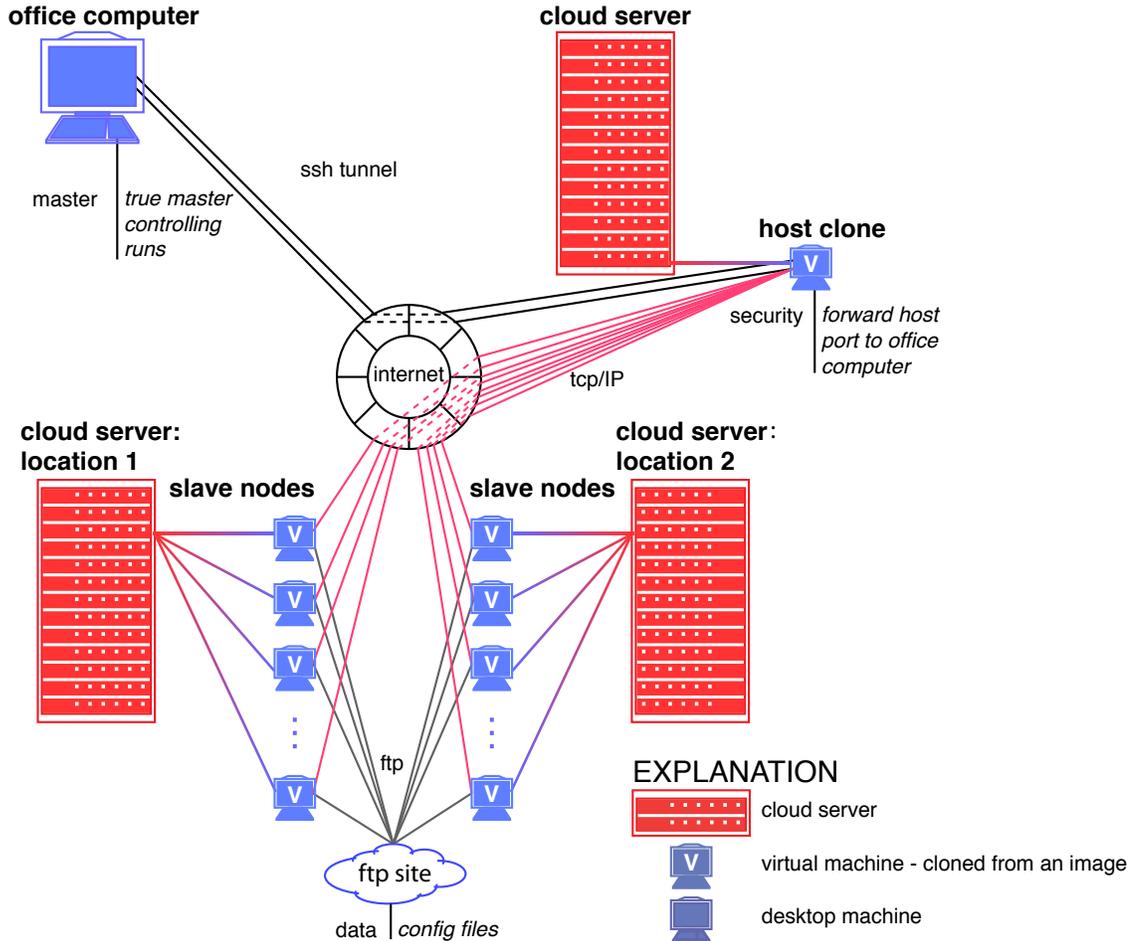


Figure 2. Diagram illustrating the arrangement of computers for production runs. The multiple locations (1 and 2) in the diagram highlight the fact that each batch of virtual machines run may be hosted at a different physical location within the cloud network of servers.

instance and the control of remotely launched processes through SSH. However, setting up an SSH host on Windows can be difficult. Furthermore, the most efficient way to transfer model files to each computational node is through creating instances from an image that contains the necessary files. Each instance is “throw-away” in the sense that its existence is only for the purpose of running models, and persistence is not important. Therefore, using a service, while specific to the problem at hand, is more efficient and poses fewer logistical challenges than a more general approach.

In this report, an example code is provided that can act as a service. The code `node_starter.py` depends on `cloudPESTclient.py`, which must be in the `PYTHONPATH` or in the same directory as `node_starter.py`. These Python code files must both be loaded onto the slave image. The `node_starter` code has two simple purposes. First, it (optionally) pulls a configuration file from an FTP site. This configuration file typically informs each node of the IP address and port for the master and the problem-specific information such as the exact version of beoPEST to run and the name of the case (.pst) file to run. Furthermore, paths to specific slave-node folders can be defined. These path designations are local on an individual slave image, and recall that the directory structure in a slave image will be recreated on every instance launched from it. Second, `node_starter.py` goes to each slave-node folder location and spawns a subprocess for beoPEST in each

10 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

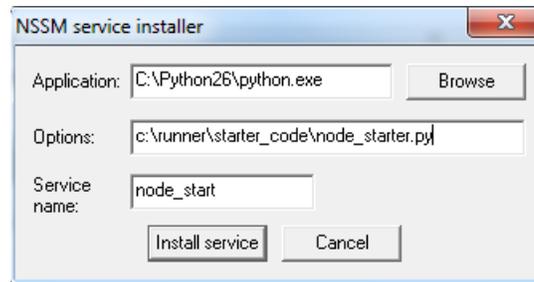


Figure 3. Dialogue box for the Non-Sucking Service Manager (NSSM) showing example input for starting the `node_starter.py` service.

location; this subprocess is kept active as long as the service is running. More details about the configuration files and slave functions are discussed later in the Functions section.

The specific process to establish a service is platform dependent, so the following paragraphs provide instructions for Windows. The details for Linux would require creating a daemon with associated permissions handling.

Windows Services

In Windows, there are many ways to establish services. Third-party applications provide the ability to convert any application to a service and in this report the Non-Sucking Service Manager (NSSM) (<http://iain.cx/src/nssm/>) is discussed. Note that, for the previous URL, “http” may be replaced by “https” if following the results of a Google search, so take care in following this link directly from this document, or confirm “http” in the URL if typing it in directly. When using NSSM, the application converted to a service actually is the Python engine. This should use a fully qualified path to define—in other words, the path should be defined in full from the root, which is the `c:` drive on Windows. The argument for the application should be the fully qualified path to `node_starter.py`. Starting NSSM uses the syntax `nssm install` on the command line. Figure 3 shows the arguments entered in the NSSM GUI.

Create Both Images

Once images are configured, they must be saved as EBS-backed images to enable instances to be started from them (as clones). In EC2 terminology, this process is referred to as “creating” the images. The easiest way to create the EBS-backed images is to use the menu on the AWS Console website (right-click on the Instance and select “Create Image (EBS AMI)”). The result is a new EBS-backed image, which can, in turn, be cloned as necessary.

Launch a Master Instance (optional)

A single master instance is launched and the IP address is noted and inserted into the configuration file that will be uploaded to an FTP site to later be downloaded and used by the slave service. This is optional, as the master can be run on the office computer (not on the cloud) and either directly host beoPEST or forward the port from an SSH tunnel machine, if necessary, to securely move through a firewall. On figure 2 no master instance is launched, rather the master is on the office computer and port forwarding is used through the host clone.

Start SSH Tunnel (optional)

SSH tunnels allow a secure and encrypted connection to be maintained across a firewall between a locally owned master computer and a cloud machine, which has the sole purpose of forwarding data from the cloud across the secure SSH tunnel to the master. Regardless of which operating system is being used to run beoPEST

and the models, the easiest and most reliable way to establish an SSH tunnel is to launch a Linux machine on the cloud to forward a port. In this example, Ubuntu 10.04 was selected as the Linux version, and steps to establish the tunnel are outlined. The host clone in figure 2 represents the port-forwarding virtual machine. The alternative to SSH tunneling is opening a port through the firewall, which may be considered a security risk in some environments.

Configuring a Port-Forwarding Linux Machine on the Cloud

First, select a suitable base Ubuntu 10.04 image as the starting point. Selecting an EBS-backed image is preferable in that it simplifies creating images for future use. One source of such images is available from the Canonical Ubuntu Team at <http://uec-images.ubuntu.com/releases/10.04/release/>. Next, connect to the instance using SSH and edit the file `/etc/ssh/sshd_config` or `/etc/ssh/sshd_config`. There should be only one file called `sshd_config` and it can be located at either path noted above. Using `vi` or another command-line text editor, go to the bottom of the file and add the line `GatewayPorts yes`. This will allow a port to be forwarded from the cloud through the SSH tunnel back to the local master. Note that this ability is disabled by default, and that when establishing the SSH tunnel, a user with non-administrative rights locally should always be used. Establishing the port forwarding over the SSH tunnel can pose a minor security risk, although establishing the SSH tunnel as a non-administrative user mitigates most of that risk because even if someone hacked the connection, with non-administrative rights, they would be limited in the damage they could do. To restart SSHD, type `sudo /etc/init.d/ssh restart` at the command line. This only is necessary if the SSH tunnel is to be established prior to restarting the system. On subsequent restarts, port forwarding will be enabled owing to the changes made above. Finally, to obviate the need for performing these steps on every session, the Ubuntu machine can be created as an EBS image so instances simply can be spawned whenever forwarding is to be established.

There are two options for establishing an SSH tunnel with Port Forwarding. The first option is a command-line sequence and the second employs a GUI.

Establish the SSH Tunnel with Port Forwarding on Command Line

If a command-line interface is available for SSH, such as on Linux or using CYGWIN on Windows, the following syntax is used on the local master to forward a remote port from the Ubuntu cloud machine:

```
ssh -i curr_key.pem -R ubuntu@remoteIP 4040:localhost:4040
```

In this example, `ssh` is the call to the SSH program; the `-i` argument indicates that RSA encryption using an identity file is used for authentication in lieu of a password. `curr_key.pem` is the identity file, which is also called the private key in previous discussions of establishing an Amazon AWS account. `ubuntu@remoteIP` is the account name (this should always be “ubuntu”) to log onto at the address represented by `remoteIP`. The address is obtained after the port-forwarding instance is started up through the AWS Console or through the APIs. Finally, the final three-argument string indicates the port to forward to (first) followed by the IP address to forward to, and finally the port to forward to. In this example, port 4040 on the cloud port-forwarding machine will be forwarded to the master machine `localhost` on port 4040. In effect, this establishes a listening port for TCP/IP on the cloud and that should be the target for the slaves (e.g. `###.###.###.###:4040`, where `###.###.###.###` is the IP address of the master). When the master is started, however, it is started with “:4040” as the port which will listen on localhost.

The terminal/command window in which the call was made, above, must remain open on the master to keep the SSH tunnel open.

Establish the SSH Tunnel with Port Forwarding Using PuTTY

A free GUI for establishing SSH connections—including tunnels—on Windows is called PuTTY

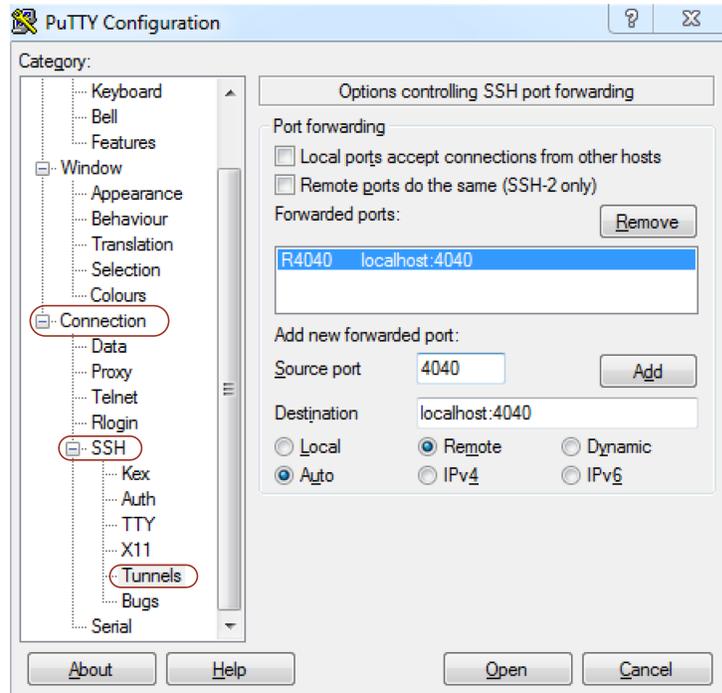


Figure 4. Dialogue of the SSH tunnel section of the PuTTY configuration.

(<http://www.chiark.greenend.org.uk/~sgtatham/putty/>). Figure 4 shows the dialogue box in the PuTTY GUI for establishing an SSH tunnel. In source port, the open port on the cloud (4040 in this example) is indicated, and in destination, the full address on the master is indicated (localhost:4040). The “Remote” radio button also should be selected as shown, and then clicking the “Add” button is the final step. This tunneling step should be added to the general PuTTY configuration step for using PuTTY to access cloud computers in Amazon AWS

(<http://it.toolbox.com/blogs/managing-infosec/connecting-to-amazon-aws-from-windows-to-a-linux-ami-30656>).

Launch beoPEST on the Master

If using port forwarding, this launch will be local and the port specified will be the port to which traffic is forwarded. An example is

```
beopest.exe casename.pst /H :4040. If not using port forwarding, the launch is still local.
```

Launch Slave Instances

Slave instances can be launched using either the AWS Console (ideal if a small number of slaves are to be launched) or the APIs in the cloudPEST module, discussed in detail later in this report. Note that if instances are launched using the APIs, they still can be monitored using the AWS Console. In either case, `cloudPESTclient.py` and `node_starter.py` must be present on the slave image in order to appear on each slave instance. The number of slaves launched depends on the requirements of the problem, account details, and cost considerations. In any case, launching in batches of 25 or less is advised. An important side issue is that the default allowance for concurrently run instances is 20 on EC2. To launch more than that, a special exception must be requested. However, even after receiving the exception, particularly when running Windows instances, EC2 may prevent large numbers of instances from starting owing to security and performance concerns. Currently (2011) the only solution to these limits is *ad hoc* communication with EC2 customer service.

In the process of increasing the limit, users should make a strong effort to communicate details of their computational needs and usage goals to enable a smooth transition to larger numbers of instances.

Terminate Slaves and Master At Completion

All instances can be safely terminated after all results are either stored on a local machine (through port forwarding to the master) or transferred off the cloud.

cloudPEST Classes and Functions

The remainder of this report outlines the classes and functions implemented in cloudPEST for interaction with EC2. The classes provided are objects largely used to organize the information used by the functions and, in turn, by the users. These objects contain attributes but not methods. The functions facilitate the actual interaction with the cloud. The classes are defined in `cloudPEST.py` as are the main functions. A separate file, `cloudPESTclient.py`, provides utility programs that are used on the slave machines to pull and interpret parameter files from an FTP site and to start beoPEST in slave mode. Finally, `node_starter.py` provides an example script that can be used as a service (as discussed above) to implement the necessary functions from `cloudPESTclient.py`.

Classes

images:

The `images` class contains the information for all AMIs owned by the user. Attributes are:

Attribute	Description
<code>ami_id</code>	AMI identification number for each available image.
<code>description</code>	Description of each image (this short description is the name given to the AMI at time of creation).

instances:

The `instances` class contains the information for all node machines for a particular session. Attributes are:

Attribute	Description
<code>ami_id</code>	AMI identification number from which the node is instanced.
<code>instance_id</code>	Instance-identification number for each node.
<code>password</code>	Administrator password-assigned to new instances and used only for RDP connection to a node.
<code>port</code>	Port opened for master communication using beoPEST.
<code>ip_address</code>	IP address for the node. This address persists throughout the session but is lost at stop/termination.
<code>state</code>	Current state (running, stopped, or pending).
<code>private_DNS</code>	Private Domain Name System (DNS) address accessible only within the EC2 cloud.
<code>public_DNS</code>	Public DNS address that should be accessible outside the EC2 cloud.
<code>group</code>	Security group with which the instance is associated.

14 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

master:

The `master` class contains the information for the master machine in a particular session. The master typically is a local machine rather than a cloud instance, although this is not required. Attributes are:

Attribute	Description
<code>ami_id</code>	Optional variable naming the AMI on which the master is instanced.
<code>instance_id</code>	Optional instance-identification number if master is on the cloud.
<code>address</code>	Address (IP address, DNS address, or computer name) at which the master is contacted by beoPEST.
<code>port</code>	Port opened for master communication using beoPEST.

cloudPEST Functions

Functions in `cloudPEST.py` are described in this section. All functions are described in terms of their input, output, description, an example call, and the underlying EC2-tools functions used. The functions for the client installation that starts slave machines is in the section titled `cloudPESTclient`.

To use the API functions, they need not be run on the master, but on any machine configured with the EC2 command-line tools and the `cloudPEST` Python module. The first step in running the functions, which are the subject of this report, is to import them into either an interactive Python session or a Python script. This can be done selectively, for example:

```
from cloudPEST import query_instances
```

Alternatively, all classes and functions can be imported from the entire package:

```
from cloudPEST import *
```

`query_images()`

The `query_images` function returns a `cloudPEST.images` object with a list of available AMIs owned by the user and their descriptive names.

Input None.

Output A `cloudPEST.images` object.

Underlying ec2-tools API Functions `ec2-describe-images`

`query_instances(*instance_id)`

The `query_instances` function returns a `cloudPEST.instances` object with a list of instances currently running, stopped, or pending. The optional input variable `instance_id` is a list of specific instances the user wishes to query. The default returns information about all instances owned by the user.

Input A specific instance ID or list of instance IDs—*optional*.

Output A `cloudPEST.instances` object.

Underlying ec2-tools API Functions `ec2-describe-instances`

run_instances(*ami_id*, *instance_count*, *keyname*, *group*, **insttype*, **cnodes*, **availzone*)

The `run_instances` function runs (creates) as many instances as requested by the call. Note that limits may be imposed by Amazon regarding the maximum number of allowed instances. Currently (2011), a default limit of 20 instances is imposed and users must contact Amazon to increase this limit. The instances run typically are either a single instance intended to be used as a master, or a group of instances intended to be used as slave nodes. When launching a large number of instances (greater than 10 or 20), it is advisable to launch in subgroups. This helps distribute the load among various physical locations. It is critical to remember that, once instances are run, the user is being billed for the time—even in partial hours. For example, if a user runs 100 instances in a single call, and those instances are allowed by EC2, even if it is a mistake and they are immediately terminated, the user will be charged for 1 hour of use on each of the 100 instances. Extreme caution should be exercised when launching multiple instances and frequent checking at either the AWS Console or using the `query_instances` should be conducted to keep tabs on exactly how many instances are running at a given time.

Input `ami_id` (*string*) identifier of the AMI from which clones should be run, `instance_count` (*integer*); number of instances to run, `keyname` (*string*); the name of the keypair used to communicate with the instances, `group` (*string*); the name of the security group in which the instances will be run; `insttype= []` (*string*); is the optional instance type, which must be chosen from the available list, that should be periodically updated within the code. The default is the smallest/least-expensive instance type supported by the AMI type. `cnodes= []` (*instances class*) is the instances class, which will contain information on the started instances. If blank, a new one will be started, or if provided, it will be appended to with the new instances. `availzone= []` (*string*) defines the availability zone in which instances are to be started. Typically, this should be left as default because AWS optimizes based on loads, etc.

Output Screen output indicates success or provides an appropriate error message on failure.

Underlying ec2-tools API Functions `ec2-run-instances`

start_instances(*instance_id*)

The `start_instances` function starts the instances identified in the calling `instance_id` list.

Input A specific instance ID or list of instance IDs. This applies to instances that were previously run and are currently in a stopped state.

Output Screen output indicates success or provides an appropriate error message on failure.

Underlying ec2-tools API Functions `ec2-start-instances`

stop_instances(*instance_id*)

The `stop_instances` function starts the instances identified in the calling `instance_id` list.

Input A specific instance ID or list of instance IDs. This applies to instances that were previously run and are currently in a stopped state.

Output Screen output indicates success or provides an appropriate error message on failure.

Underlying ec2-tools API Functions `ec2-stop-instances`

16 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

terminate_instances(*instance_id*)

The `terminate_instances` function starts the instances identified in the calling `instance_id` list.

Input A specific instance ID or list of instance IDs. This applies to instances that were previously run and are currently in a stopped state.

Output Screen output indicates success or provides an appropriate error message on failure.

Underlying ec2-tools API Functions `ec2-terminate-instances`

cloudPESTclient Functions

Functions in `cloudPESTclient.py` are described in this section. The functions are described similarly to those in the cloudPEST Functions section. An example calling function called `node_starter.py` is provided following the description of the functions. Several paths are hard coded in `cloudPESTclient.py`. Passing all these paths through configuration files quickly becomes confusing, so the user should create either the same directory structure on his or her slave instances and images as described here, or should change the paths in the code. A comment in the code stating “# changeable path” indicates that the path on the next line can be changed.

parseLine(*line*)

The `parseLine` function returns a string with the first value on a line prior to whitespace. This function is used to read in configuration files.

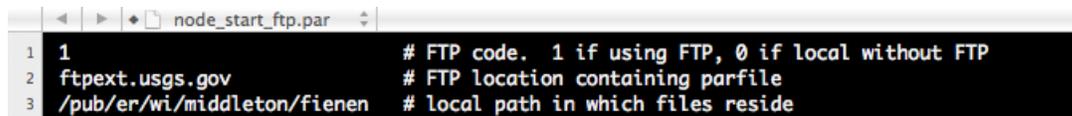
Input A string containing an entire line read from a file.

Output A string with the first value from the line.

readFTPparfile(*infile*)

The `readFTPparfile` function reads a configuration (parameter) file identified by the input variable `infile`. The function returns values indicating whether a configuration file containing run-specific information should be downloaded from an FTP site and, if so, the address of the FTP site and the path at the site where the file resides. The current (2011) version assumes anonymous login is enabled. A straightforward modification can expand functionality to allow passing login information; however, such login information must be represented unencrypted to properly communicate it to the process.

figure 5 shows an example configuration file. Note that the binary flag on the first line indicates whether the run-specific configuration file should be retrieved from an FTP file or if the local version should be used. The “local path” referred to is the subdirectory on the FTP site in which the files reside—not the path on the instance where the files will be saved.



```
node_start_ftp.par
1 1 # FTP code. 1 if using FTP, 0 if local without FTP
2 ftpext.usgs.gov # FTP location containing parfile
3 /pub/er/wi/middleton/fienen # local path in which files reside
```

Figure 5. Parameter file with details of FTP downloading used by `node_starter.py`.

Input A fully qualified path identifying the location of the configuration file, which contains a detailed configuration file for the `node_starter.py` program.

Output Three variables are returned: `FTPaddress` is the host address of the root FTP site from which the file will be downloaded such as `ftp.example.com`; `FTPflag` is a boolean flag indicating whether a remote detailed configuration file should be downloaded from an FTP site (the boolean value is converted from 1=True or 0=False in the configuration file); and `FTPdir` is the the working directory on the host containing the file to be downloaded.

`retrieveFTPparfile(FTPaddress, FTPdir, detailedParInfile)`

The `retrieveFTPparfile` function retrieves the detailed configuration file needed for `node_starter.py` from the host specified in `FTPaddress` and `FTPdir` to the fully qualified path and filename locally identified in `detailedParInfile`. `detailedParInfile` should be of the form:

```
r'c:\\runner\\Node_starter\\node_start_ftp.par'
```

Input `FTPaddress` is the host address of the root FTP site from which the file will be downloaded such as `ftp.example.com`, `FTPdir` is the the working directory on the host containing the file to be downloaded, and `detailedParInfile` is the fully qualified path.

Output Three variables are returned: `FTPflag` is a boolean flag indicating whether a remote detailed configuration file should be downloaded from an FTP site (the boolean value is converted from 1=True or 0=False in the configuration file); `FTPaddress` is the host address of the root FTP site from which the file will be downloaded such as `ftp.example.com`; `FTPdir` is the working directory on the host containing the file to be downloaded; and `detailedParInfile` is the fully qualified path and filename of the local destination (local meaning on the slave instance) for the configuration file being downloaded from the FTP site.

`readRUNparfile(infile)`

The `readRUNparfile` function reads the detailed configuration (parameter) file typically downloaded using `retrieveFTPparfile` and identified in `infile`, which typically is the fully qualified path identified in `detailedParInfile`. The returned values are case-specific values to be used when starting nodes. This function is highly customizable depending on the specific application. In this case, the specific variables should be used as a guide for customization by the user.

Input `infile` is the fully qualified path to the detailed configuration file for `node_starter.py`.

Output As currently (2011) constructed, the following variables are returned (this is, again, customizable to meet the user's needs):

`bPcasename` (*string*) is the name of the case to be run (e.g. `casename.pst`).

`bPexec` (*string*) is the name of the executable called to start the slaves (e.g. `beopest64.exe`).

`bPhost` (*string*) is the IP address of the master node.

`bPport` (*string*) is the port of the master node.

`bPlocalNodes` is a list of strings, each identifying the full path to a slave directory on the instance.

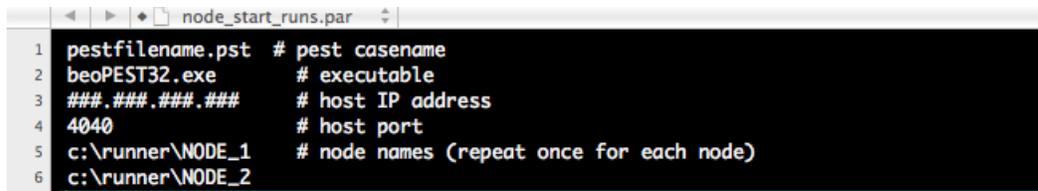
Because the instances are clones of an image (an AMI), the local node paths will be the same on each.

figure 6 shows an example parameter file for running `beoPEST` locally.

`StartNode(bPcasename, bPexec, bPhost, bPport, bPCurrentNode)`

The `StartNode` function performs the first step of starting a node, using all the calling information and path identification passed in the input variables. A critical note is, in the calling function, each handle to a process (the returned value from this function) must have a "wait" call performed so that the operating system keeps the subprocess active. An example section of code is as follows:

18 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation



```
node_start_runs.par
1 pestfilename.pst # pest casename
2 beoPEST32.exe   # executable
3 ###.###.###.### # host IP address
4 4040           # host port
5 c:\runner\NODE_1 # node names (repeat once for each node)
6 c:\runner\NODE_2
```

Figure 6. Parameter file details of paths and executables needed to start beoPEST in slave mode, used by `node_starter.py`.

```
for bPCurrentNode in bPlocalNodes:
    a = StartNode(bPcasename, bPexec, bPhost, bPport, bPCurrentNode)
    live_nodes.append(a)
for a in live_nodes:
    a.wait()
```

Input `bPcasename` (*string*) is the name of the PEST case, including the “.pst” extension.

`bPexec` (*string*) is the name of the executable that will be run.

`bPhost` (*string*) is the IP address of the master machine (or the port-forwarding machine if SSH tunneling is employed).

`bPport` (*string*) is the port on the master, which is listening for beoPEST.

`bPCurrentNode` (*string*) is the fully qualified path to the node.

Output `p` (*subprocess module Popen object*) references the specific beoPEST slave instance started by the function. This object can be used by other processes to which it is passed, specifically using the “wait” call as discussed above.

`node_starter.py()`

The file `node_starter.py` included in this distribution provides an example implementation of the cloud-PESTclient functions. Comments are provided within the code. The path on line 9 may be changed to any location where the logging files are desired to be accessible. All other manipulation of the behavior of this code is through the two parameter files discussed next. The main tasks performed in this code are as follows:

readFTPparfile The parameter file determining if the run parameter file should be downloaded from an FTP and, if so, details needed for the download is read. Figure 5 shows the details of an example FTP parameter file. Note that, as currently (2011) structured, only anonymous login to the FTP is supported. The function `readFTPparfile` can be altered to enable password authentication.

readRUNparfile The parameter file for runs, as illustrated in figure 6, is read next, providing all the information needed to start slave instances of beoPEST in the appropriate local directories. This file is read from the local directory, but can be in place either at the creation of the image or downloaded from an FTP site as indicated above.

Start Nodes The final step is to start each node (or slave instance of beoPEST) in a distinct directory. This code should not be altered, as it takes important steps to ensure that, once started, the nodes stay active.

Example Application

Figure 7 illustrates an example application of a Python script that can be used to query available images, start a master and several groups of slaves, and terminate all the slaves when a session is complete. These snippets of code can be combined into a menu-driven application, run with an integrated development environment using break points, or run as separate scripts.

```

1 from cloudPEST import run_instances, query_images, query_instances, terminate_instances
2
3 my_images = query_images()
4 print my_images.description[1]
5 # start a master machine
6 run_instances(my_images.ami_id[1], 1, 'mnf_office', 'mnf_runner')
7
8 # start slaves in groups. small groups first to make sure all is working, then groups of 25
9 print my_images.description[3]
10 print 'starting group 1'
11 run_instances(my_images.ami_id[3], 2, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
12 print 'starting group 2'
13 run_instances(my_images.ami_id[3], 23, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
14 print 'starting group 3'
15 run_instances(my_images.ami_id[3], 25, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
16 print 'starting group 4'
17 run_instances(my_images.ami_id[3], 25, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
18 print 'starting group 5'
19 run_instances(my_images.ami_id[3], 25, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
20 print 'starting group 6'
21 run_instances(my_images.ami_id[3], 25, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
22 print 'starting group 7'
23 run_instances(my_images.ami_id[3], 25, 'mnf_office', 'mnf_runner', insttype='m2.xlarge')
24
25
26 all_instances = query_instances()
27 i=1
28 #terminate all slaves without terminating the master
29 for cinst in all_instances.instance_id:
30     if i > 0:
31         print cinst
32         terminate_instances(cinst)
33
34     i+=1

```

Figure 7. Example implementation of the cloudPEST functions.

Acknowledgments

The authors acknowledge collaborative support from John Doherty (Watermark Numerical Computing), Randall Hunt (U.S. Geological Survey, USGS, Groundwater Systems Team, Middleton, Wis.), Matthew Ungaro and Nathaniel Booth (USGS, Center for Integrated Data Analytics, Middleton, Wis.). Reviews and testing by Jonathon Carter (Barr Engineering), Charles Spalding (McDonald Morrissey Associates), Erik Johnson (enStratus), Bonnie Stich Fink (USGS, Science Publishing Network, Louisville, Ky.), Kevin Breen (USGS, Office of Science Quality and Integrity, New Cumberland, Pa.), Rodney Sheets (USGS, Water Science Field Team, Columbus, Ohio), and Steven Peterson (USGS, Nebraska Water Science Center, Lincoln, Nebr.) also are greatly appreciated. Finally, John Shim of Amazon Web Services was a valuable resource for navigating the logistics of EC2.

References Cited

- Doherty, John, 2010, PEST—Model-independent parameter estimation—User manual (5th ed., with slight additions): Brisbane, Australia, Watermark Numerical Computing, 336 p.
- Hunt, R. J., Luchette, Joseph, Schreüder, W. A., Rumbaugh, J. O., Doherty, John, Tonkin, M. J., and Rumbaugh, D. B., 2010, Using a cloud to replenish parched groundwater modeling efforts. *Ground Water*, v. 48, no. 3, p. 360-365, doi: 10.1111/j.1745-6584.2010.00699.x.
- Schreüder, W. A., 2009, Running BeoPEST. *in* Proceedings of the 1st PEST Conference, Potomac, Md., 1–3 November.

Blank page

Glossary

Abbreviations

- AMI** — Amazon Machine Image
- API** — Application programming interface
- AWS** — Amazon Web Services
- DNS** — Dynamic Name System
- EBS** — Elastic Block Storage
- EC2** — Elastic Compute Cloud
- FTP** — File Transfer Protocol
- GUI** — Graphical User Interface
- IP** — Internet Protocol
- MPI** — Message Passing Interface
- NSSM** — Non-Sucking Service Manager
- RDP** — Remote Desktop Protocol
- SSH** — Secure Shell

Terms

- cloud computing** — Cloud computing, in general, refers to networked computers, potentially at distant locations, connected using Internet data-transfer protocols, being used to distribute data storage or computational loads. In the context of this report, cloud computing also implies the renting of CPUs to be used as a temporary super computer.
- image** — An image (typically an AMI in this report) is a configuration of a virtual computer that can be invoked as an instance. AMI images can be made from a running or stopped instance using the console or command-line tools.
- instance** — An instance is a specific invocation of an image, like a clone. Many instances may be invoked from a single image, and each will share the same base configuration.
- fully qualified path** — A fully qualified path, is a path in which every level of the directory structure is explicitly identified, starting at the drive level. In Windows, an example is; `c:\runner\subdir\file1.txt` and on Linux and OSX, an example is `/users/fienen/runner/file1.txt`.
- GUI** — A graphical user interface is the way many people interact with modern computer programs through menus, graphics, and mouse-related options. This is in contrast with scripts and command-line programs, which are run from a system terminal prompt and require typing commands.

22 cloudPEST – A Python Module for Cloud-Computing Deployment of PEST, a Program for Parameter Estimation

master — A computer processor on which the host process of beoPEST or parallel PEST is run. This host process coordinates the scheduling of runs distributed to slave processors.

node — The term “node” is used interchangeably with “slave.”

PATH — An environment variable for the operating system directing the search for applications when names are typed at the command line. For example, to run Python from the command line in an arbitrary location, the PATH variable must include `c:\python26`.

run — To run an instance is to create a fresh clone from an AMI.

PYTHONPATH — An environment variable, similar to PATH, in which Python searches for modules to import. When the comment `import` is used in Python, first the local directory is searched for the module, followed by the directories in PYTHONPATH. On Windows, this is set in Advanced settings in the control panel.

service — A process that starts automatically when a machine is booted. In cloudPEST, the slave processes are started on the cloud as services.

session — A session means a period in which instances are created and interacted with. A session starts when a group of one or more instances are started, and ends when those instances are stopped or terminated.

slave — A computer processor (typically one of many) in which individual model runs scheduled by a master processor are run. In cloudPEST, a large number of slave computers typically are started to facilitate parallel model runs using beoPEST.

start — To start an instance is to awaken an instance that was run already and has been stopped (parked).

stop — To stop an instance is to shut down a running instance. This is similar in behavior to shutting down a physical machine in that all data are retained if the instance is started again.

terminate — To terminate an instance is identical in behavior to physically destroying a tangible machine. No trace of its existence remains accessible again.