Creating Vagrant Machines for Distribution of Software Environments

Hans Petter Langtangen[1,2]

Anders E. Johansen[1]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 21, 2013

Scientific software soon gets very complicated to install because packages build on numerous other packages, some of which may be hard to compile and link successfully on a system. Those who frequently need to make sure their target audience, consisting typically of students, collaborators, or customers, has a certain set of packages installed on their system, run into a serious problem due to the fact that few in the target audience have the competence, interest, and patience to install all the packages on their computer with its particular version of the operating system.

There are many working solutions to this problem:

- Long technical installation descriptions that in practice require considerable experience with compiling and linking software packages.
- Ready-made, easy-to-install files for particular platforms, e.g., Debian packages (`.deb` files) for Linux systems like Ubuntu, `.dmg` bundles for Mac, or `.exe` files for Windows. It can still be quite some work for a user to install the right combination of many packages, although each package gets installed by a double click.
- Virtual machines, such as VirtualBox and VMWare Fusion, can run a particular operating system and thereby take advantage of the most easy-to-use platform from an installation perspective. In particular, one can run Ubuntu or other Debian-based Linux systems and use the `apt-get install` and `pip install` commands to make installation of packages and their dependencies trivial.

A Vagrant machine[1] is essentially a wrapper around VirtualBox which makes it very easy to build, distribute, and use a virtual machine. The present document

[1]`http://www.vagrantup.com/`

describes how to create and operate a Vagrant machine. The target audience of the document is scientists who wants to spend a minimum of efforts on offering or using a complete computing environment with much sophisticated, hard-to-install mathematical software.

By *host* we mean the operating system used to build or run a Vagrant machine. Operating system commands issued on the host have a prompt `Terminal>` while commands issued in the Vagrant machine feature the prompt `Machine>`.

1 Problem setting

We shall work with a specific example: creating a computing environment for the participants in a course on computational X, where X is any science or engineering subject such as chemistry, physics, quantum mechanics, fluid dynamics, oceanography, and so forth. The challenge with courses featuring heavy computations is two-fold:

1. to minimize the amount of time the audience spends on installation issues and
2. to minimize the teacher's hassle with all types of operating systems that might be present on the laptops in the audience.

An attractive solution to this minimization problem is to create a Vagrant machine, which is simply a file with a virtual ready-made computer that anyone in the audience can easily download and use on any Windows or Mac computer.

Another advantage is that all users of a Vagrant machine have exactly the same computing environment (unless they modify the machine). The teacher can then easily debug a user problem inside the teacher's own Vagrant machine. And anything that the teacher demonstrates on her computer works out of the box on the participants' computers.

Different types of Vagrant machines can be made for different types of courses or purposes. For example, a research project can set up a software environment for its project members, as a Vagrant machine, to ensure that the environment is conserved for the future, which is a key principle for reproducible science. Users may have many Vagrant machines on their computers and switch between the computing environments.

1.1 Contents of the Vagrant machine

The Vagrant machine needs to have an operating system. Here we choose Ubuntu of two main reasons:

1. software on Ubuntu can be trivially installed as Debian packages
2. the Debian software repository is at the time of this writing the richest repository for pre-built mathematical software.

We remark that the user of the machine will mainly work with files and directories on the host system and only use the Ubuntu system to run computations.

To be specific, the sample computing environment to be illustrated here consists of a Python-based ecosystem for scientific computing. Examples on basic software includes

- Text editors: `emacs`, `vim`, `gedit`
- Compilers: `gcc`, `g++`, `gfortran`
- Numerical libraries: ATLAS
- Python packages: `numpy`, `scipy`, `sympy`, `matplotlib`, `ScientificPython`

Most of these packages are in Debian and trivially installed by a `sudo apt-get install packagename` command, but the Python packages are often more conveniently installed in their latest version by a `pip install packagename` command. A few Python packages must be installed directly from the source code, via downloading followed by the `sudo python setup.py install` command, if they do not exist in Debian, or if they are not supported by `pip install`, or if one needs to download the latest development version. The example will in detail illustrate the various cases.

Much more sophisticated packages than those listed above, for instance PETSc[2] and FEniCS[3], may be very challenging to build from scratch, but as long as Debian versions are offered (which is the case with PETSc and FEniCS), installation on a Debian-based system like Ubuntu is still just a trivial `apt-get install` command.

In the Vagrant machine, we create two directories:

- `~/bin` for executable programs and scripts
- `~/srclib` for Python packages installed from source code

We also include two useful files:

- A small, but illustrative `~/.bashrc`[4] file for setting up the Linux system.
- `~/.rsyncexclude`[5] for excluding certain files when running `rsync` for copying files between machines, or between machines and external disks or memory sticks.

1.2 Installing the necessary software for using Vagrant

Before going into details on how to utilize Vagrant, you need to have it on your host system.

[2] `http://www.mcs.anl.gov/petsc/`
[3] `http://fenicsproject.org`
[4] `https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/.bashrc`
[5] `https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/.rsyncexclude`

VirtualBox. Download and install VirtualBox[6]. Choose the version according to the operating system on the host. For example, if you want to build or run Vagrant machines under Mac OS X, choose *VirtualBox x.y.z for OS X hosts*, where `x.y.z` is the version number of VirtualBox. Double click the downloaded `.dmg` file to install Vagrant. Those who work on a Windows machines will select *VirtualBox x.y.z for Windows hosts*, which downloads an `.exe` file which can just be double clicked to perform the installation.

Installing VirtualBox on Ubuntu and other Linux systems can be challenging. Here is a recipe. Start with

```
Terminal> sudo apt-cache search virtualbox
```

to find a package `virtualbox-X`, where `X` denotes a particular version number (e.g., `4.2`). Then copy and paste the following commands into the terminal window:

```
Terminal> wget -q \
 http://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc \
 -O- | sudo apt-key add -
Terminal> sudo sh -c 'echo \
 "deb http://download.virtualbox.org/virtualbox/debian precise contrib" \
 >> /etc/apt/sources.list'
Terminal> sudo apt-get update
Terminal> sudo apt-get install virtualbox-X
```

(Recall to replace `X` by the appropriate version number.) You may need to run `sudo apt-get -f install` and upgrade packages. It is easier to work with VirtualBox on Mac or Windows if you run into trouble with Ubuntu.

We recommend to install VirtualBox as shown above on Ubuntu rather than downloading a particular `.deb` file (Debian package) from the VirtualBox site[7], because the `apt-get install` approach above makes it easier to explicitly get all the packages that VirtualBox depends on.

Vagrant. Download and install Vagrant[8]. Choose the latest version and the installation file corresponding to the host's operating system (where you installed VirtualBox). On a Mac, you select the `Vagrant-x.y.z.dmg` file (`x.y.z` denotes the version of the software), on Windows the `Vagrant_x.y.z.msi` file is the relevant choice. On Ubuntu, select `vagrant_x.y.z_*.deb` and install it by `sudo dpkg -i vagrant_x.y.z_*.deb`.

On Windows and Mac OS X, the `vagrant` command is automatically available after installation (because the directory where the `vagrant` executable resides is placed in your `PATH` environment variable). This is true for many Linux systems too, otherwise you must add the relevant directory where the `vagrant` program was installed (say `/opt/vagrant/bin`) to your `PATH` variable.

Cygwin (only on Windows). Windows computers do not feature an ssh client and an X server by default, which are needed in scientific applications.

[6] `https://www.virtualbox.org/wiki/Downloads`
[7] `https://www.virtualbox.org/wiki/Downloads`
[8] `http://downloads.vagrantup.com/`

Therefore, we recommend to install Cygwin[9], which gives easy access to an ssh-client and an X-server on Windows computers. Actually, Cygwin extends Windows with a complete Unix environment. Download the Cygwin's setup.exe[10] file and follow the instructions given by the installer. Only the minimal base packages from the Cygwin distribution are installed by default. This means that we need to manually select the 'X11' category during installation to install Cygwin's X11 server.

Once installed, we need to add Cygwin's ssh client to our `PATH`. Cygwin is by default installed to `C:\cygwin`, so the command is `set PATH=%PATH%;C:\cygwin\bin`.

2 Creating the Vagrant machine

In this section we explain how to select an operating system for the Vagrant machine, how to install pre-compiled binary packages, how to install (Python) packages from source code, and how to configure the machine.

2.1 Choice of operating system type

The first step of building a Vagrant machine is to choose a plain version of an operating system to base the machine on. This is called a *base box*. A lot of pre-made base boxes for various versions of operating systems are available at `http://www.vagrantbox.es`. (If, for some reason, you want to build a base box with another operating system, there are instructions[11] for that.) Let us decide on adopting *Ubuntu precise 64*, which we find down on the list. This is a version of Ubuntu 12.04 (`precise` refers to the official Ubuntu name Precise Pangolin for version 12.04). Click on *Copy* to copy the URL. You have now two choices:

1. you can build and distribute a complete virtual machine, or

2. the user can download a box and then automatically install a list of prescribed packages in the box.

The former approach, called a *complete Vagrant machine* in the following, results in one big file containing the machine. The latter approach, referred to as a *Vagrant machine specification* results in very small text files to be distributed to the users.

The advantage of a complete Vagrant machine is that users can download one big file and they immediately have an operative machine. You are also guaranteed that all users have identical environments. An empty Vagrant machine is easy to distribute, but the disadvantage is that a user's initialization of the machine takes (very) long time since a lot of packages must be downloaded and installed. Something can go wrong with the installation. It may also happen that different users get slightly different environments because they run the installation process of their machines at different times.

[9] `http://cygwin.com/install.html`
[10] `http://cygwin.com/setup.exe`
[11] `http://docs-v1.vagrantup.com/v1/docs/base_boxes.html`

2.2 Downloading a base box to create a complete Vagrant machine

Paste the copied URL of the chosen box in a new browser tab. This action should automatically download a file `precise64.box`. Say you store this file in a directory `~/vagrant`. Go to this directory and run

```
Terminal> vagrant box add mybox precise64.box
Terminal> vagrant init mybox
Terminal> vagrant up
```

The result is now an initialized Vagrant machine `mybox` which you can log into. The `vagrant` directory where these commands are run is known as the *project directory* in the Vagrant documentation[12].

2.3 Making an empty Vagrant machine

Make some directory (say) `~/vagrant`, move to this directory, and type

```
Terminal> vagrant init
```

This command creates a `Vagrantfile`. Invoke the file in a text editor and replace the line `config.vm.box = "base"` by the URL to the base box and add another line `config.ssh.forward_x11 = true` to enable X11 graphics. The `Vagrantfile` looks something like

```
Vagrant.configure("2") do |config|
  # All Vagrant configuration is done here. The most common configuration
  # options are documented and commented below. For a complete reference,
  # please see the online documentation at vagrantup.com.

  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = http://files.vagrantup.com/precise64.box
  config.ssh.forward_x11 = true
  ...
end
```

2.4 Installing packages in a complete Vagrant machine

This section assumes that you want to build and distribute a complete Vagrant machine as defined above. There is not much installed yet on the `mybox` machine, but this is an Ubuntu system where we can very easily install what we want via `sudo apt-get install` or `pip install` commands, or by downloading source code and performing manual installation. Section 2.6 describes a type of file for listing packages and Unix commands, with an associated tool `deb2sh.py` for automatic generation of installation scripts. Using these utilities, it is close to trivial to create a rich computing environment.

Creating files. Make sure you are logged out of the Vagrant machine (Ctrl-D) and located in the project directory on the host. Download default versions[13] of some key files: `deb2sh.py`, `debpkg_minimal.txt`, `.bashrc`, and `.rsyncexclude`.

[12] `http://docs.vagrantup.com/v2/`

[13] `https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant`

Just click on the files, choose the *Raw* version, and right-click to save each file to the project directory. Read about the former two files in Section 2.6 and the latter two in Section 2.7. Edit the files to your users' needs. Then run

```
Terminal> python deb2sh.py debpkg_minimal.txt
```

to produce a Bash script `install_minimal.sh` and an equivalent Python script `install_minimal.py`. Make sure you run all the commands in the project directory (`~/vagrant`).

You may alternatively download the more comprehensive `debpkg.txt`[14] package list and use that file as a starting point. Running `deb2sh.py debpkg.txt` will produce the scripts `install.sh` and `install.py`.

Installing files and packages. When you have edited the above files according to your users' needs, you are ready to log into the Vagrant machine, copy files to the machine and run the installation. The project directory is visible as `/vagrant` inside the Vagrant machine (see Section 3.4 for more details). The relevant login command is `vagrant ssh`, here followed by two copy commands:

```
Terminal> vagrant ssh
Machine> cp /vagrant/.bashrc .
Machine> cp /vagrant/.rsyncexclude .
```

Now you can run the (lengthy) installation process by

```
Machine> bash /vagrant/install_minimal.sh
```

or

```
Machine> python /vagrant/install_minimal.py
```

If something goes wrong with the installation, edit the script on the host system (invoke `/vagrant/install_minimal.sh` in an editor) and rerun the installation command inside the Vagrant machine.

You may want to include the installation scripts in the box so that users can see exactly what has been installed and rerun installation commands if necessary (e.g., at a later stage to update the software).

```
Machine> cp /vagrant/install_minimal.sh .
Machine> cp /vagrant/install_minimal.py .
```

Enabling X11 graphics. It is recommended that you test graphics programs and check that they display the graphics on the host appropriately. To this end, you need to enable X11 graphics on the host by editing the file `Vagrantfile` in the project directory so that it includes the line `config.ssh.forward_x11 = true`:

```
Vagrant::Config.run do |config|
    ...
    # Enable X11
    config.ssh.forward_x11 = true
    ...
end
```

[14] `https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/debpkg.txt`

To get X11 graphics to work, you must also start X11 on the host: run Applications - Utilities - X11 on a Mac, or invoke Start - All Programs - Cygwin-X - XWin Server on Windows.

A simple application just to test X11 is to run `xterm` from the Vagrant machine. A terminal window will pop up on the host.

Packaging a new box. When everything is copied to the box, installed, and tested, we need to package the installed virtual environment into a box in order to distribute it to other users. Log out of the machine and finalize the machine by running the `vagrant package` command in the project directory:

```
Terminal> vagrant package --output course.box \
          --vagrantfile Vagrantfile
```

The settings in `Vagrantfile` are now packed with the box. In particular, if X11 graphics has been enabled in `Vagrantfile` as described above, you have a fully functioning Ubuntu machine in `course.box` that will work seamlessly with X11 graphics on the host. Users can just do

```
Terminal> vagrant box add course course.box
Terminal> vagrant init course
Terminal> vagrant up
Terminal> vagrant ssh
```

2.5 Installing packages in an empty Vagrant machine

An empty Vagrant machine is distributed to users as a bundle of `Vagrantfile` and an installation script. Read Section 2.6 and make a Bash installation script.

You may want to distribute `.bashrc` and `.rsyncexlude` files too, as described in Section 2.7, but that is easiest done by letting the installation script download the files from site where they are available. Relevant lines may be

```
$ cd $HOME
$ wget http://tinyurl.com/m88bljf/.bashrc
$ wget http://tinyurl.com/m88bljf/.rsyncexclude
```

To ensure that the user's initialization process of the machine invokes an installation of the desired packages, you need to add a line to `Vagrantfile` that runs the Bash script. Say the name of the script is `install_minimal.sh`. The relevant line is shown below:

```
Vagrant.configure("2") do |config|
  ...
  # Run installation
  config.vm.provision :shell, :path => "install_minimal.sh"
  ...
end
```

Users must now have the files `Vagrantfile` and `install_minimal.sh` to create a complete Vagrant machine on their computers.

2.6 Scripts for installing ready-made packages

We have developed a little tool where one can list the desired Debian or Python packages in a computing environment in a file with default name `debkpg.txt`. This file may also contain plain Unix commands for doing other types of installation, like `pip install`, or cloning of source code repositories with subsequent execution of a `setup.py` file. Concrete examples are listed below.

A little Python script `deb2sh.py`[15] reads the installation specification in some file `debpkg_minimal.txt`[16] and creates a Bash script `install_minimal.sh`[17] and an equivalent Python script `install_minimal.py`[18] for running all the necessary operating system commands to install all the packages in the correct order. The script aborts if any package cannot be installed successfully. The problem must then be fixed, or the package must in worst case be removed (just comment out the install line(s) in the Bash or Python script). The script can thereafter be rerun again.

The following is an extract of packages as they are listed in the mentioned `debpkg_minimal.txt` file:

```
# Minimal installation for a Python ecosystem
# for scientific computing

# Editors
emacs python-mode gedit vim ispell

# Compilers
gcc g++ gawk f2c gfortran
autoconf automake autotools-dev

# Numerical libraries
libatlas-base-dev libsuitesparse-dev

# Python
idle
python-pip
python-dev
# Matplotlib requires libfreetype-dev libpng-dev
# (otherwise pip install matplotlib does not work)
libfreetype6-dev libpng-dev
pip install numpy
pip install sympy
#pip install matplotlib   # pip may fail for matplotlib
python-matplotlib
pip install scipy

# ScientificPython must be installed from source
$ if [ ! -d srclib ]; then mkdir srclib; fi
$ cd srclib
$ hg clone https://bitbucket.org/khinsen/scientificpython
$ cd scientificpython
$ sudo python setup.py install
$ cd ../..
```

The syntax has four elements:

[15] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/deb2sh.py
[16] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/debpkg_minimal.txt
[17] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/install_minimal.sh
[18] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/install_minimal.py

1. comment lines are just copied to the Bash and Python installation scripts,
2. lines starting with `$` are plain Unix commands and run by the installation scripts,
3. lines starting with `pip install` lists packages to be installed with `pip`, while
4. all other non-blank lines are supposed to list the name of Debian packages to be installed by `sudo apt-get install` commands.

The examples above show all four line types. Observe in particular how we can freely add Unix commands to download `ScientificPython` from its Bitbucket repo (done in the `srclib` subdirectory) and install the package manually by running `setup.py` the usual way.

Some examples on lines in the automatically generated `install_minimal.sh` script are

```
#!/bin/bash
# Automatically generated script. Based on debpkg.txt.

function apt_install {
  sudo apt-get -y install $1
  if [ $? -ne 0 ]; then
    echo "could not install $1 - abort"
    exit 1
  fi
}

function pip_install {
  for p in $@; do
    sudo pip install $p
    if [ $? -ne 0 ]; then
      echo "could not install $p - abort"
      exit 1
    fi
  done
}

function unix_command {
  $@
  if [ $? -ne 0 ]; then
    echo "could not run $@ - abort"
    exit 1
  fi
}

sudo apt-get update --fix-missing

# Minimal installation for a Python ecosystem
# for scientific computing

# Editors
apt_install python-mode gedit vim ispell
...
pip_install numpy
pip_install sympy
```

```
apt_install scipy
...
# ScientificPython must be installed from source
unix_command if [ ! -d srclib ]; then mkdir srclib; fi
unix_command cd srclib
unix_command hg clone https://bitbucket.org/khinsen/scientificpython
unix_command cd scientificpython
unix_command sudo python setup.py install
```

Notice.

- Installation commands may fail. Therefore we have made separate functions for doing the `apt-get` and `pip install` commands. We test the value of the environment variable `$?` after the installation of a package: a successful installation implies value of 0, while values different from 0 mean that something went wrong. We then abort the script with `exit 1`.
- The `apt-get install` command will prompt the user for questions for every package, but here we use the option `-y` to automatically rely on default answers, i.e., accepting `yes` to all questions.

The corresponding lines in the equivalent, automatically generated `install.py` file look as follows.

```
import commands, sys

def system(cmd):
    """Run system command cmd."""
    failure, output = commands.getstatusoutput(cmd)
    if failure:
       print 'Command\n  %s\nfailed.' % cmd
       print output
       sys.exit(1)

system('sudo apt-get update --fix-missing')

system('sudo apt-get -y install python-mode gedit vim ispell')
...
system('pip install numpy')
system('pip install sympy')
system('sudo apt-get -y install scipy')
...
system('if [ ! -d srclib ]; then mkdir srclib; fi')
system('cd srclib')
sytem('hg clone https://bitbucket.org/khinsen/scientificpython')
system(' cd scientificpython')
system('sudo python setup.py install')
```

The Python script does not test the Unix environment variable `$?`, but the first return value from the `getstatusoutput` function acts as the value of `$?`.

We can use the Bash or Python script to easily automate installation of packages in the Vagrant machine. More powerful, industry standard tools for setting up complete software environments are Chef[19] and Puppet[20].

2.7 Setting up a default environment with `.bashrc`

We should include a brief `.bashrc` file in the Vagrant machine as a starting point for the user's customization of her Unix environment. Here is an example[21]:

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files for examples

export PYTHONPATH=$PYTHONPATH:$HOME/pythonlib
export PATH=$PATH:$HOME/bin

# Create some aliases for rsync commands for copying files:
rsync_basic="-rtDvz -u -e ssh -b"
rsync_excl="--exclude-from=$HOME/.rsyncexclude"
rsync_del="--suffix=.rsync~ --delete --force"
scp_rsync="rsync $rsync_basic $rsync_excl"
scp_rsync_del="$scp_rsync $rsync_del"
alias scp_rsync="$scp_rsync"
alias scp_rsync_del="$scp_rsync_del"

# If running interactively, then:
if [ "$PS1" ]; then
    alias ls='ls -sF'
    alias grep='grep --color=auto'
    alias fgrep='fgrep --color=auto'
    alias egrep='egrep --color=auto'


    # enable programmable completion features (you don't need to enable
    # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
    # sources /etc/bash.bashrc).
    if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
        . /etc/bash_completion
    fi

    # set a new prompt and the directory as window title

    # PROMPT_DIRTRIM=1 makes the dir in window title have 1 trailing dir name
    # (instead of the whole path)
    export PROMPT_DIRTRIM=1

    # Let prompt in terminal window (PS1) display username, time and
    # current working directory
    PS1='\u:\D{%H.%M} \W> '
    # Add directory info to the title bar: (often done in terminal prefs too)
    PS1=$PS1"\[\e]0;\w\a\]"
fi
```

The handy `rsync` commands for copying files require a list of files to ignore, so a file `.rsyncexclude`[22] must be present in the home holder:

```
.#*
*.rsync~
```

[19] http://www.opscode.com/
[20] https://puppetlabs.com/
[21] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/.bashrc
[22] https://github.com/hplgit/vagrantbox/tree/master/doc/src/vagrant/src-vagrant/.rsyncexclude

```
*.a
*.o
*.so
*~
.*~
*.log
*.dvi
*.aux
*.old
tmp_*
*_tmp*
*.tmp
tmp.*
.tmp*
*.tar
*.tar.gz
*.tgz
*.pyc
```

3 Operating the Vagrant machine

For a user, the initialization of a new machine depends on whether it is a complete Vagrant machine or an empty Vagrant machine.

> **Important.**
> On a Windows computer, always operate the Vagrant machine from Cygwin's terminal, which has both an ssh-client and an X-server. The terminal can be started from Start - All Programs - Cygwin-X - XWin Server.

3.1 Operating a complete Vagrant machine

The Vagrant machine `course.box`, created as described in Section 2.4, can now be distributed to users. A user must do the following steps.

Step 1. Install VirtualBox and Vagrant as described in Section 1.2.

Step 2. Create a directory `vagrant` and move `course.box` to this directory. We also recommend to make a subdirectory `projects` where all files and directories to be used from the Vagrant machine reside. You edit files in the `vagrant/projects` directory tree on the host.

Step 3. Run the these commands from the `vagrant` directory:

```
Terminal> vagrant box add course course.box
Terminal> vagrant init course
```

Step 4. Start X11 on the host: run Applications - Utilities - X11 on a Mac, or Start - All Programs - Cygwin-X - XWin Server on Windows.

Step 5. Start (boot) the Vagrant machine:

```
Terminal> vagrant up
```

Step 5. Log in on the machine:

```
Terminal> vagrant ssh
```

Log out with Ctrl-D as usual in Unix terminal windows.

3.2 Operating an empty Vagrant machine

The user has the files `Vagrantfile` and some installation script, say `install_minimal.sh` as described in Section 2.5. The user should make some directory `vagrant`, copy `Vagrantfile` and `install_minimal.sh` to this directory, and from this directory run

```
Terminal> vagrant up
Terminal> vagrant ssh
```

The first command takes a long time to execute since it runs the installation script. Log out with Ctrl-D.

3.3 Working with an initialized Vagrant machine

The daily work with the Vagrant machine is very easy. Simply go to the `vagrant` directory where the machine resides and run

```
Terminal> vagrant up
Terminal> vagrant ssh
```

You are now inside the machine and can reach files on the host from `/vagrant/projects` (see the next section for more details). Log out with Ctrl-D and in again with `vagrant ssh`. Create and edit files on the host in `~/vagrant/projects` and its subdirectories.

Before closing a laptop or shutting it down, it is recommended to log out of the Vagrant machine and run `vagrant suspend`.

3.4 Shared directories

Inside the Vagrant machine, `/vagrant` is a directory shared with the user's file system. More precisely, `/vagrant` points to the *project directory* where the file `Vagrantfile` resides and where the `vagrant up` command was run (`~/vagrant` if you have followed the specific directory naming suggested in this document). If users of the Vagrant machine keeps all their files relevant for the machine in the project directory and its subdirectories, all these directories will be shared between the machine and the user's file system. Normally, this feature is enough for efficient communication of files between the Vagrant machine's file system and the user's file system. One can also set up other shared directories, see the Vagrant documentation for Synced Directories[23].

Since the Vagrant machine shares directories with the host system, users can safely edit files in the shared directories with their favorite editor on the host system. The Vagrant machine will have immediate access to the files.

Here is a typical example. Assume that `vagrant up` and `vagrant ssh` were run in a directory `myubuntu`. On the host, create a subdirectory `src` of `myubuntu`. Start an editor and type in the following Python program in a file `test1.py`:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 3, 11)
```

[23] `http://docs.vagrantup.com/v2/synced-directories/basic_usage.html`

```
y = np.exp(-x)
plt.plot(x, y)
plt.show()
```

This program will show X11 graphics on your host machine. If this machine runs the Linux operating system, everything is fine, but if this is a Mac or Windows machine, X11 must be started as described in Section **??**. If that is necessary, log out, start X11, log in again (`vagrant ssh`).

Run the `test1.py` program:

```
Terminal> cd /vagrant
Terminal> cd src
Terminal> python test1.py
```

A plot of the curve $y = e^{-x}$ should now be seen on the screen.

3.5 Troubleshooting

Troubleshooting: shared directory is invisible. It may happen that the `/vagrant` directory seems empty inside the Vagrant machine. Two steps will fix this problem. First, run

```
Machine> sudo /etc/init.d/vboxadd setup
```

inside the Vagrant machine. Second, log out and run

```
Terminal> sudo vagrant reload
```

outside the Vagrant machine. Then do `vagrant ssh` and take an `ls /vagrant` to see that the files in the project directory (e.g., `Vagrantfile` and the Vagrant box) are visible.

Troubleshooting: "couldn't connect to display ...". This error message points to the problem that X11 graphics on the Vagrant machine cannot be shown on the host's screen. Inserting the line `config.ssh.forward_x11 = true` in the file `Vagrantfile` in the project directory and starting X11 on the host are the two steps that will fix the problem. Unless you build a Vagrant box, the editing of `Vagrantfile` should not be required as a ready-made box was packaged with X11 forwarding (cf. the `vagrant package` command in Section 2.4). To start X11 on Mac, run Applications - Utilities - X11, while on Windows, go to Start - All Programs - Cygwin-X - XWin Server. Log out of the Vagrant machine (Ctrl-D) and in again (`vagrant ssh`).

Troubleshooting: Internet is not reachable. A test if Internet is reachable is to run a `ping` command inside the machine, e.g.,

```
Machine> ping us.ubuntu.archive.com
```

A hanging command indicates that Internet is not reachable. Log out of the box, run `vagrant reload`, and `vagrant ssh`. Try the `ping` command again.

3.6 Stopping the Vagrant machine

There are three ways to stop the virtual Vagrant machine from the host (i.e., you must be logged out by Ctrl-D from the machine):

- `vagrant suspend` sends the machine to sleep mode. Waking it up is done with `vagrant up`.
- `vagrant halt` shuts off the machine. To start it again, a full boot with `vagrant up` is needed.
- The machine can be removed forever by `vagrant destroy`.

3.7 Placing the Vagrant machine in the cloud

There are numerous free file hosting sites[24] where a Vagrant machine can be stored and shared with others. One service that offers enough space (50 Gb) for many big Vagrant machines is Mega: `https://mega.co.nz/`. You must create a free account before uploading your files. Right-click on any uploaded file, choose *Get link*, and a window pops up with the URL to the file. You can distribute this link to the target audience of your file.

3.8 Using VMWare Fusion

Not written yet.

3.9 Documentation of Vagrant

- The official Vagrant documentation[25] targets web developers, but contains more details than the tutorial above.
- An article in The Linux Journal[26] is technically slightly outdated, but gives much valuable additional information.

A Condensed instructions for students

Say you want distribute a complete Vagrant machine with the URL

`http://some.where.net/path/to/course.box`

Here is the need-know-information for users:

Step 1. Download and install VirtualBox[27]. Choose the version according to the operating system on the host. For example, if you want to build or run Vagrant machines under Mac OS X, choose *VirtualBox x.y.z for OS X hosts*, where `x.y.z` is the version number of VirtualBox. Double click the downloaded `.dmg` file to install Vagrant. Those who work on a Windows machines will select *VirtualBox x.y.z for Windows hosts*, which downloads an `.exe` file which can just be double clicked to perform the installation.

[24] `http://www.freewaregenius.com/the-best-free-send-large-files-services-ten-file-hosting-services-compare`
[25] `http://docs.vagrantup.com/v2/`
[26] `http://www.linuxjournal.com/content/introducing-vagrant`
[27] `https://www.virtualbox.org/wiki/Downloads`

Step 2. Download and install Vagrant[28]. Choose the latest version and the installation file corresponding to the host's operating system (where you installed VirtualBox). On a Mac, you select the `Vagrant-x.y.z.dmg` file (`x.y.z` denotes the version of the software), on Windows the `Vagrant_x.y.z.msi` file is the relevant choice. On Ubuntu, select `vagrant_x.y.z_*.deb` and install it by `sudo dpkg -i vagrant_x.y.z_*.deb`.

Step 3 for Windows users. If you have a Windows machine, you should install Cygwin[29]. Download the Cygwin's setup.exe[30] file and follow the instructions given by the installer. Make sure you manually select the 'X11' category during installation. Cygwin is not needed on Mac computers.

Step 4. Start X11: run Applications - Utilities - X11 on a Mac, or Start - All Programs - Cygwin-X - XWin Server on Windows.

Step 5. Move to your home directory and make a new directory `vagrant` and a subdirectory `projects`:

```
Terminal> cd
Terminal> mkdir vagrant
Terminal> mkdir vagrant/projects
Terminal> cd vagrant
```

All files that you run from the Vagrant machine are supposed to reside in `vagrant/projects` and its subdirectories.

Step 6. Download the file

```
http://some.where.net/path/to/course.box
```

Store `course.box` in the `vagrant` directory.

Step 7. Make sure you stand in the `vagrant` directory. Run

```
Terminal> vagrant box add course course.box
Terminal> vagrant init course
Terminal> vagrant up
Terminal> vagrant ssh
```

You are now inside a Ubuntu system!

Step 8. Open a file `vagrant/projects/test1.py` in an editor on the host system and write the following lines in the file:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 3, 11)
y = np.exp(-x)
plt.plot(x, y)
plt.show()
```

Save the file. Move to the terminal window with the Ubuntu (Vagrant) machine. Run

[28] `http://downloads.vagrantup.com/`
[29] `http://cygwin.com/install.html`
[30] `http://cygwin.com/setup.exe`

```
Machine> cd /vagrant/projects
Machine> python test1.py
```

You should see a plot of e^{-x} on the screen. If you encounter any problems, read the paragraphs below.

A.1 Troubleshooting: shared directory is invisible

It may happen that the `/vagrant` directory seems empty inside the Vagrant machine. Two steps will fix this problem. First, run

```
Machine> sudo /etc/init.d/vboxadd setup
```

inside the Vagrant machine. Second, log out and run

```
Terminal> sudo vagrant reload
```

outside the Vagrant machine. Then do `vagrant ssh` and take an `ls /vagrant` to see that the files in the project directory (e.g., `Vagrantfile` and the Vagrant box) are visible.

A.2 Troubleshooting: "couldn't connect to display ..."

This error message points to the problem that X11 graphics cannot be shown on the host. It should be sufficient to start X11 on the host, see Step 4 above.

A.3 Troubleshooting: Internet is not reachable

A test if Internet is reachable is to run a `ping` command inside the machine, e.g.,

```
Machine> ping us.ubuntu.archive.com
```

A hanging command indicates that Internet is not reachable. Log out of the box, run `vagrant reload`, and `vagrant ssh`. Try the `ping` command again.