Creating Vagrant Machines for Distribution of Software Environments

Hans Petter Langtangen[1,2]

Anders Johansen[1]

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

Aug 10, 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, ranging from long, technical installation descriptions to ready-made, easy-to-install, tailored packages for particular platforms, e.g., Debian packages for Ubuntu, `.dmg` bundles for Mac, or `.exe` bundles on Windows. Virtual machines (VirtualBox, VMWare Fusion) can be used if a tailored package is available for one operating system. Nevertheless, the clearly best solution so far, in the authors' opinion, is a Vagrant machine: you give people in your target audience a file, which provides a complete new "computer" with all necessary software involved. This new computer lives side by side with a normal computer.

The present note describes how to create and operate a Vagrant machine. Some nomenclature is needed: by *host* we mean the operating system used to build or run a Vagrant machine.

1 Problem setting

We shall work with a specific example: creating a computing environment for a short course on scientific computing. The challenge is to minimize the amount of time the audience spends on installation and minimize the teacher's hassle with all types of operating systems and their versions in the audience. To reach this goal, we shall create a Vagrant machine, which ensures that everybody is working in exactly the same computing environment.

1.1 Contents of the Vagrant machine

The Vagrant machine needs to have an operating system. Here we choose Ubuntu of one main reason: software on Ubuntu can be installed as Debian packages, and the Debian software repository is now the richest repository for pre-built mathematical software.

We have developed a little tool where one can list the desired Debian 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` and cloning of source code repositories with subsequent execution of a `setup.py` file. Specific examples on content are listed below.

A little Python script `deb2sh.py` reads the installation specification in `debpkg.txt` and creates a Bash script `install.sh` and an equivalent Python script `install.py` for running all the necessary operating system commands. If any package cannot be installed successfully, the script stops. The problem must then be fixed, or the package must in worst case be removed (just comment out the install line(s) in `install.sh` or `install.py`). The script can thereafter be rerun again.

The following is an extract of packages as they are listed in a `debpkg.txt` file:

```
# 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 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: comment lines are just copied to the `install.sh` and `install.py` scripts, lines starting with `$` are plain Unix commands and also copied to the output scripts, lines starting with `pip install` lists packages to be installed with `pip`, while 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` subfolder/subdirectory) and install the package manually by running `setup.py` the usual way.

Some examples on lines in the `install.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
}

sudo apt-get update

apt_install ispell
pip_install numpy
pip_install sympy
```

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` will prompt the user for questions for every package, but here we use the option `-y` to automatically answer `yes` to all questions.

The corresponding lines in `install.py` are

```
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')

system('sudo apt-get -y install ispell')
system('pip install numpy')
system('pip install sympy')
```

The Python script does not test the environment variable `$?`, but the first return value from the `getstatusoutput` is basically `$?`.

We can use `install.sh` or `install.py` to automate installation of packages in the Vagrant machine. More powerful tools for setting up complete software environments are Chef and Puppet.

In the Vagrant machine, we create two folders:

- `~/bin` for executable programs and scripts
- `~/srclib` for Python packages installed locally

We also include two useful files:

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

1.2 Installing Vagrant

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

Download and install VirtualBox. 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 that can just be double clicked.

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, because the `apt-get install` makes it easier to get all the packages that VirtualBox depends.

hpl 1: *I didn't manage to install VB on Ubuntu. Gave up after 6 h. Tried all sorts of things, .deb first, then apt-get, the commands above. Always some mismatch of VB and the Ubuntu version.*

Download and install Vagrant. 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`. See above for how to install `.dmg`, `.exe`, and `.deb` files.

On Windows and Mac OS X, the `vagrant` command is automatically be available after installation (because the folder/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 folder where the `vagrant` program was installed (say `/opt/vagrant/bin`) to your `PATH` variable.

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 for that.) Let us decide on adopting *Ubuntu precise 64*, which we find down on the list, for the Vagrant machine. 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 and paste the URL in a new browser tab. This action should download a file `precise64.box`. Say you store this file in a folder `~/vagrant`. We can now

log in to the Ubuntu precise 64 virtual machine: open some terminal window, make some folder (say) `vagrant_project`, to this folder, and type

```
Terminal> vagrant box add precise64 ~/vagrant/precise64.box
Terminal> vagrant init precise64
Terminal> vagrant up
Terminal> vagrant ssh
```

The first line defines a Vagrant machine with the name `precise64`. The second creates a central file, `Vagrantfile`, with settings for the Vagrant machine. The third command starts (boots) the Vagrant machine, and the fourth makes us log in to the Vagrant machine (just as the normal `ssh` command does). The `vagrant_project` folder where these commands are run is known as the *project folder* in the Vagrant documentation.

hpl 2: *Might be problems with ssh on Windows, see http://docs-v1.vagrantup.com/v1/docs/getting-started/ssh.html. This must be tested on Windows 7.*

2.2 Installing packages

There is not much installed yet on the `precise64` 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 1.1 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. We simply run `deb2sh.py` in the folder where our package specification `debpkg.txt` resides. The resulting `install.sh` must be located in the project folder `vagrant_project`, as created above.

There are two ways to install the software in the machine: either we do this beforehand, or we put a call to the installation script in the `Vagrantfile`. The difference between the methods is that the former strategy may result in a big machine that takes a lot of time for users to download, while the latter approach provides a smaller machine, but the `vagrant up` command that users must run, takes much time since it will install all the packages.

To install the packages in the machine before users downloads the machine, run this command inside the machine:

```
Terminal> sh -x /vagrant/install.sh
```

The other approach inserts a line for executing `install.sh` in the `Vagrantfile`:

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

The script will be run as part of the `vagrant up` command.

2.3 Setting up a default environment with `.bashrc`

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

```
# ~/.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` must be present in the home holder:

```
.#*
*.rsync~
*.a
*.o
*.so
*~
.*~
*.log
*.dvi
*.aux
*.old
tmp_*
```

```
*_tmp*
*.tmp
tmp.*
.tmp*
*.tar
*.tar.gz
*.tgz
*.pyc
```

If you have these and other files on your file system, they can easily be copied into the Vagrant machine by placing the files in the project folder (where the `vagrant up` command was run and where the `Vagrantfile` resides). This folder is visible from the Vagrant machine as `/vagrant` (see also Section 3.1).

The work inside the Vagrant machine is now over for this time. Logging out is done by Ctrl-D as in any Unix shell.

2.4 Adding X11 support

We would like to adjust some preferences of the Vagrant machine. For most scientific applications it is useful to enable X11 forwarding such that the user can see graphics launched from applications running in the machine. Invoke the file `Vagrantfile` in an editor and type the line `config.ssh.forward_x11 = true` in the file as shown below:

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

2.5 Finalizing the machine

When everything is installed, we need to package the virtual environment into a box in order to distribute it to other users. This can be done by logging out of the virtual machine and running the vagrant package command (in the project folder):

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

A real machine (containing what is listed earlier, plus the FEniCS software) is available from GitHub through the address `http://goo.gl/h7mv4` (note the file size: 4.7Gb!).

3 Operating the Vagrant machine

The Vagrant machine `ourpackage.box`, created above, can now be distributed together with `Vagrantfile` to everyone working on a project to ensure that they all have the same software versions, no matter if they are using Windows, Mac, or Linux. Here is the user's recipe.

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

Step 2. Run the commands

```
Terminal> vagrant box add ourpackage ourpackage.box
```

hpl 3: *Is this command necessary for users?*

Anders 4: *It depends. We actually should not distribute both a Vagrantfile and a box. We could i) distribute just a Vagrantfile (plus install scripts) with a lot of configurations. This Vagrantfile must contain a URL to a box. The user may then start with Step 3 below. Or ii) distribute just a box. In this case, a Vagrantfile with configurations can be added to the box when packing it. With this approach the step above is highly necessary, and should be succeeded by the command 'vagrant init ourpackage' before continuing to Step 3.*

Step 3. Start (boot) the Vagrant machine:

```
Terminal> vagrant up
```

Step 4. Log in on the machine:

```
Terminal> vagrant ssh
```

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

Step 5 (if necessary). If the `Vagrantfile` does not contain the string "install.sh", the `install.sh` script must be run inside the box:

```
Terminal> bash -x /vagrant/install.sh
```

3.1 Shared folders

Inside the Vagrant machine, `/vagrant` is a folder shared with the user's file system. More precisely, `/vagrant` points to the *project folder* where the file `Vagrantfile` resides and where we ran the `vagrant up` command. If users of the Vagrant machine keeps all their files relevant for the machine in the project folder and its subfolders, all these folders 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 folders, see the Vagrant documentation for Synced Folders.

Since the Vagrant machine shares folders with the host system, users can safely edit files in the shared folders 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 folder `myubuntu`. On the host machine, create a subfolder `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, 4*np.pi, 501)
y = np.exp(-x/2.)*np.sin(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. On a Mac, open Finder, go to Applications, and then the Utilities subfolder, and double-click `X11.app` to start X11.

Log out of the Vagrant machine (`Ctrl-D`) and log in again (`vagrant ssh`). Move to the `src` folder and run the `test1.py` program:

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

A plot of the curve $y = e^{-x/2} \sin x$ should now be seen on the screen.

hpl 5: *Anders, what to do with X11 on Windows?*

Anders 6: *hpl, could you please read the rest of this section, question everything that seems unclear, rewrite the some of my bad formulations and put it in a suitable place in the document?*

In Windows neither an ssh-client nor an X-server is installed by default. However, both these issues can be solved by installing Cygwin. Download the Cygwin's setup.exe 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 manually to select the 'X11' category during installation to install Cygwin/X.

Anders 7: *Don't remember if the following was necessary. Need to check it next time I'm in Windows*

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`.

Anders 8: *hpl, here's the recipe for distributing only the Vagrantfile. Maybe copy this to the same place as the box?*

Cygwin's terminal, which now has both an ssh-client and an X-server, can be started from Start - All Programs - Cygwin-X - XWin Server. In this terminal we can download a Vagrantfile with all necessary configurations and start the Vagrant machine:

```
Terminal> wget http://dl.dropboxusercontent.com/u/13793917/Vagrantfile
Terminal> vagrant up
Terminal> vagrant ssh
```

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

```
Terminal> 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 folder 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. Make sure the line with `config.ssh.forward_x11 = true` is present in the file `Vagrantfile` in the project folder (see above). Also make sure that X11 is running on the host computer (see above for how to do this).

Each time the user wants to access the virtual environment, the following two lines are enough:

```
vagrant up
vagrant ssh
```

3.2 Stopping the Vagrant machine

There are three ways to stop the virtual Vagrant machine:

- `vagrant suspend` sends the machine to sleep mode. Waking it up is done with `vagrant resume`.
- `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`.

3.3 Using VMWare Fusion

Not written yet.

3.4 Documentation of Vagrant

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