Py6S Documentation

Release 1.7.0

Robin Wilson

February 25, 2017

Py6S is a interface to the Second Simulation of the Satellite Signal in the Solar Spectrum (6S) atmospheric Radiative Transfer Model through the Python programming language. It allows you to run many 6S simulations using a simple Python syntax, rather than dealing with the rather cryptic 6S input and output files. As well as generally making it easier to use 6S, Py6S adds a number of new features including:

- The ability to run many simulations easily and quickly, with no manual editing of input files
- The ability to run for many wavelengths and/or angles and easily plot the results
- The ability to import real-world data to parameterise 6S, such as radiosonde measurements, AERONET sun photometer measurements and ground reflectance spectra from spectral libraries

Py6S was originally created as part of my PhD, to allow me to easily run a number of 6S simulations - to perform sensitivity analyses, for example - but has now been extended to cover the entire range of 6S functionality. **Anything that can be done using the standard 6S model can be done through Py6S.**

If you've just arrived here for the first time then you might like to read the Introduction and Intended Audience pages to see whether Py6S is suitable for your needs, and then follow the Installation instructions and then the Quick Start guide. The Py6S posts on my blog may also be of interest to you, as these explain various features of Py6S using case studies and example code. Full documentation of every function in Py6S can be accessed from the Table of Contents below.

If you need further help, or just want to send me some comments about Py6S, then visit the Support page, where you will find details of the Py6S mailing list. If you use Py6S as part of some research you publish then you **must** cite the first paper listed on the Publications page.

Py6S is fully-working, but also under active development. The Release Notes give information on what has changed in recent releases, and the Roadmap describes plans for future features.

CHAPTER 1

Table of Contents

Introduction

Py6S is a Python interface to the 6S Radiative Transfer Model. It allows you to run many 6S simulations using a simple Python syntax, rather than dealing with the rather cryptic 6S input and output files. As well as generally making it easier to use 6S, Py6S adds some new features:

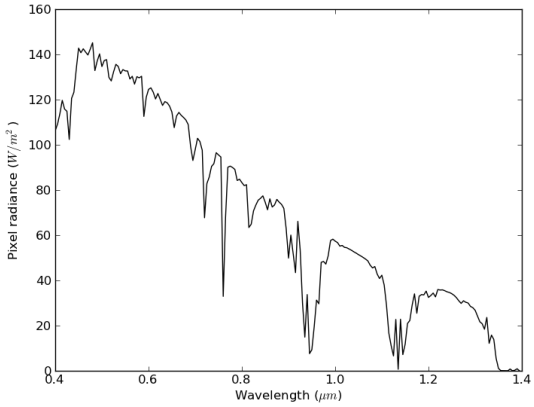
- The ability to run many simulations easily and quickly, with no manual editing of input files
- The ability to run for many wavelengths and/or angles and easily plot the results
- The ability to import real-world data to parameterise 6S, from radiosonde measurements and AERONET sun photometer measurements

Py6S has been designed to be easy to use, and to work on the 'principle of least surprise'. Far more details are available in the rest of this documentation, but a quick code example should give you an idea of what Py6S can do:

```
# Import the Py6S module
from Py6S import *
# Create a SixS object
s = SixS()
# Set the wavelength to 0.675um
s.wavelength = Wavelength(0.675)
# Set the aerosol profile to Maritime
s.aero_profile = AeroProfile.PredefinedType(AeroProfile.Maritime)
# Run the model
s.run()
# Print some outputs
print s.outputs.pixel_reflectance, s.outputs.pixel_radiance, s.outputs.direct_solar_irradiance
# Run the model across the VNIR wavelengths, and plot the result
wavelengths, results = SixSHelpers.Wavelengths.run_vnir(s, output_name='pixel_radiance')
SixSHelpers.Wavelengths.plot_wavelengths(wavelengths, results, "Pixel radiance ($W/m^2$)")
```

This will produce the results shown below:

```
0.283 112.095 667.589
```



Py6S was described in a journal article in Computers and Geosciences which **must** be cited if Py6S is used for producing outputs for a scientific report/publication - see the Publications page for more information.

This project was written as part of my PhD at the University of Southampton. The code is open-source, released under the Lesser GNU Public License, and is available at GitHub.

I'm very interested in receiving feedback, bug reports and feature suggestions - please see the Support page to find out how to get in touch with me.

Intended Audience

Py6S is an interface to the Second Simulation of the Satellite Signal in the Solar Spectrum (6S) radiative transfer model, and therefore the use of Py6S assumes some familiarity with the 6S model and the concepts behind atmospheric Radiative Transfer Modelling for remote-sensing applications.

The best resources to gain some familiarity with these topics are:

- A basic remote sensing textbook for information on the use of Radiative Transfer Models (look in your local academic library)
- The original paper on 6S by Vermote et al
- The 6S manual, particularly the first section which introduces the conceptual basis of the model (the later parts document each function in the original Fortran code, which may be less useful), here.

Py6S is a Python module, so you will need to have a basic understanding of Python programming to use the model. Don't worry too much though - you can use the model without being a Python expert! Some good tutorials for basic Python programming include LearnPython and the Google Python class.

Installation

Recommended: Using conda

Py6S and all of its dependencies (including 6S itself) are available through the `conda` package manager on the `conda-forge` channel. This is by far the easiest way to install Py6S, and is strongly recommended.

If you already have `conda` installed, then you can create a new environment containing Py6S and all of its dependencies by running:

```
$ conda create -n py6s-env -c conda-forge py6s
```

This will create a new environment called `py6s-env`, and will then tell you how to 'activate' the environment, which you should now do. You can now skip to *Testing Py6S*.

If you don't already have `conda` installed then:

1. **Install either Miniconda or Anaconda.** These are two different distributions of Python, both of which include the `conda` package manager. Miniconda comes with the 'bare minimum' of Python, conda and a few essential libraries, whereas Anaconda comes with many libraries required for data science work in Python.

   Anaconda and Miniconda are available in Python 2 and Python 3 versions, although both of these can create 'environments' using either version of Python. Py6S also works with both Python 2 and Python 3, although we recommend using Python 3.

   If you don't know which one to choose, then choose Miniconda3.

2. **Open up a terminal window and create a new environment containing Py6S.** To open up the command-line window, run a program called *Terminal* on OS X or Linux, and *Command Prompt* on Windows).

   In the terminal window, run:

```
$ conda create -n py6s-env -c conda-forge py6s
```

   This will create a separate 'conda environment' for Py6S. In conda, environments are kept entirely separate, meaning that anything you install for Py6S won't affect any other Python installation you use on your system.

   You'll need to agree to the package installation plan that conda provides, and then it will download and install Py6S and its dependencies (including the underlying 6S model).

3. **Activate the environment by running the command shown at the end of the installation** On Linux/OS X this is `source activate py6s-env`.

   On Windows this is `activate py6s-env`.

   Remember, you will need to do this **every** time before using Py6S.

You can now skip to *Testing Py6S*.

Alternative: Manual installation

Prerequisites

Executables

- `Python 2.7` or greater
- `6S v1.1` (installation instructions below)

NB: Py6S is an interface to 6S, not a replacement, so to use Py6S the 6S executable MUST exist on your system.

Python modules

- `nose`
- `numpy`
- `scipy`
- `matplotlib`
- `python-dateutil`
- `pysolar` v0.6 (optional: only required for setting the geometry from a location and time)
- `pandas` (optional: only required for importing AERONET data)
- `ipython` (recommended)

An easy way to sort all of this out is to use the Enthought Python Distribution or Anaconda, either of which will install Python plus many modules which are often used for scientific computing.

Installing 6S

6S is provided as a number of Fortran 77 source-code files from the 6S website, and must be compiled for your specific computer system. Detailed instructions are provided in the sections below.

1. Download UNIX tools: (**Windows only**)
   (a) We need to download the `make` and `tar` tools to allow us to install 6S. The easiest way to get these is through a project called GNUWin32. Go to the GnuWin32 project and choose the setup link next to `tar` and `make` and download the files.
   (b) Run the two executable files you just downloaded and work through the setup wizard for each, accepting the default options.
2. Install the Fortran compiler:

   Windows To compile the 6S code we will need a Fortran 77 compiler. These are a little difficult to find, as most compilers are now based on the (more modern) Fortran 95 standard. However, for some reason 6S does not compile using the newer compilers, so we need to find a Fortran 77 compiler. The best place I've found to get one for Windows is: http://www.cse.yorku.ca/~roumani/fortran/ftn.htm. #. Download the `FORT99.zip` file, and extract it somewhere.

   (a) Copy the `G77` folder to the root of the C drive (so that the folder is `C:\G77`).
   (b) Right-click on the **My Computer** icon on your desktop, or the Computer item on your Start Menu and select **Properties**.
   (c) Choose the **Advanced System Settings** option on the left-hand side of the resulting window and then click the **Environment Variables** button in the next dialog.
   (d) Scroll down in the bottom list box until you find a variable called `PATH`. Click **Edit** and add the following string to the end of its contents:

```
C:\Program Files\GNUWin32\bin;C:\G77\bin
```

   OS X Install `gfortran` with Homebrew.

```
$ brew install gcc
```

   Linux This may already be installed in your system. To find out,run:

```
$ gfortran -v
```

   If you don't get an error, it is installed. If not, install it using the standard installation method for your distribution. You can often do this via a GUI tool, such as Synaptic Package Manager, or via the command-line, for example:

```
$ sudo apt-get install gfortran  # Debian/Ubuntu-based distributions or...
$ sudo emerge gfortran           # Gentoo or...
$ sudo pacman -S gfortran        # Arch or... etc.
```

3. **Download the source code for 6SV1.1.** *Do not use the current available versions (v2.1 or v1.0Beta) from* http://6s.ltdri.org/ *(1.1b or 2.1) as they are not yet supported by Py6S*
4. Extract the download:

   Windows Open the command window by opening the **Start Menu** and typing *'cmd'*. In the terminal:

```
$ MD C:\Users\robin\source
$ MD C:\Users\robin\build\6SV\1.1
$ MOVE C:\Users\robin\Downloads\6SV-1.1.tar C:\Users\robin\source
$ CD C:\Users\robin\build\6SV\1.1
$ tar -xvf C:\Users\robin\source\6SV-1.1.tar .
$ CD 6SV1.1
```

Linux/OS X

```
$ mkdir source
$ mv ~/Downloads/6SV-1.1.tar source/
$ mkdir -p build/6SV/1.1
$ cd build
$ tar -xvf ../source/6SV-1.1.tar .
$ cd 6SV1.1
```

5. Edit Makefile:

   Windows Browse to the 6SV1.1 folder in **Windows Explorer** (it should in your **Downloads** folder). Inside the folder you should find a file called `Makefile`. Open the file by double-clicking on it, and selecting Notepad (*not Word*) when asked which program to open the file with. When the file has opened, find the text saying `-lm` (it will be near the end of the file) and delete it. Save the file.

   Linux/OS X The `Makefile` that comes with 6S expects to use the `g77` compiler, so we need to instruct it to use `gfortran` instead. Open the file called `Makefile` in an editor of your choice, for example:

```
$ nano Makefile
```

   Change the line:

```
FC      = g77 $(FFLAGS)
```

   to:

```
FC      = gfortran -std=legacy -ffixed-line-length-none -ffpe-summary=none $(FFLAGS)

(*Note:* The ``-ffpe-summary=none`` flag isn't available when using
GCC 4.8.4. Some people have had success leaving it out, but others
have found problems. Ideally use GCC > 4.8.4, but if that is impossible
then try without this flag.)
```

6. Compile 6S:

   (a) Compile the source code: `$ make`

   (b) If no errors have been produced, then test the 6S executable by typing:

   Windows `$ sixsV1.1.exe < ..\Examples\Example_In_1.txt`

   Linux/OS X `$ sixsV1.1 < ../Examples/Example_In_1.txt`

   Note: on Windows, make sure you run this in the standard Command Prompt (cmd.exe), not Powershell (PowerCmd.exe).

   (c) If this is working correctly you should see a number of screen's worth of output, finishing with something that looks like:

```
******************************************************************************
*                        atmospheric correction result                       *
*                        -----------------------------                       *
*       input apparent reflectance            :    0.100                     *
*       measured radiance [w/m2/sr/mic]       :   38.529                     *
```

```
*       atmospherically corrected reflectance                                  *
*       Lambertian case :      0.22187                                         *
*       BRDF       case :      0.22187                                         *
*       coefficients xa xb xc                 :  0.00685  0.03870  0.06820   *
*       y=xa*(measured radiance)-xb;  acr=y/(1.+xc*y)                         *
*************************************************************************
```

Using 6S

Once you have compiled 6S, you must place the executable (which is, by default, called `sixsV1.1` or `sixsV1.1.exe`) somewhere where Py6S can find it. The best thing to do is place it somewhere within your system path, as defined by the `PATH` environment variable. There are three ways to do this:

- **Modify your system PATH to include the location of 6S:** To do this, leave 6S where it is (or place it anywhere else that you want) and then edit the `PATH` environment variable (see above) to include that folder. The method to do this varies by platform, but a quick Google search should show you how to accomplish this.
- **Move 6S to a location which is already in the PATH:** This is fairly simple as it just involves copying a file. Sensible places to copy to include `/usr/local/bin` (Linux/OS X) or `C:\Windows\System32` (Windows).
- **Link 6S to a location on your** `PATH`:

  Windows `$ MKLINK sixsV1.1.exe C:\Windows\System`

  Linux/OS X `$ ln sixsV1.1 /usr/local/bin/sixs`

If it is impossible (for some reason) to point to the 6S executable with `PATH`, it is possible to specify the location manually when running Py6S (see below).

Installing Py6S

Installation from PyPI

The easiest way to install Py6S is from the Python Package Index (PyPI; http://pypi.python.org/pypi). Simply open a command prompt and type:

```
$ pip install Py6S
```

If you get an error saying that `pip` cannot be found or is not installed, simply run:

```
$ easy_install pip
$ pip install Py6S
```

Installation from a .egg file

Py6S is also distributed as a Python Egg file, with a name like `Py6S-0.51-py2.7.egg`. You will need to choose the correct egg file for your version of python. To find out your Python version run:

```
$ python -V
Python 2.7.2 -- EPD 7.1-2 (64-bit)
```

Then simply run the following code, which will install PySolar (required for some Py6S functions), and then Py6S itself:

```
$ pip install PySolar
$ easy_install <eggfile>
```

Where `<eggfile>` is the correct egg file for your Python version.

Testing Py6S

To check that Py6S can find the 6S executable:

```
$ python
>>> from Py6S import *
>>> SixS.test()
6S wrapper script by Robin Wilson
Using 6S located at <PATH_TO_SIXS_EXE>
Running 6S using a set of test parameters
The results are:
Expected result: 619.158000
Actual result: 619.158000
#### Results agree, Py6S is working correctly
```

This shows where the 6S executable that Py6S is using has been found at <PATH_TO_SIXS_EXE>. If the executable cannot be found then it is possible to specify the location manually (this is unlikely to be necessary if you are using the `conda`-based installation method):

```
$ python
>>> from Py6S import *
>>> SixS.test("C:\Test\sixsV1.1")
```

If you choose this method then remember to include the same path whenever you instantiate the `SixS` class, as follows:

```
>>> from Py6S import *
>>> s = SixS("C:\Test\sixsV1.1")
```

To run the full test suite to verify that both 6S and Py6S have been installed correctly (recommended):

```
$ python
>>> import os.path
>>> import Py6S; print os.path.dirpath(Py6S.__file__)
<PATH_TO_PY6S_MODULE>
>>> exit()
cd <PATH_TO_PY6S_MODULE>
$ py.test
```

Quick Start

Now that you've installed Py6S, this section will give you a brief guide on how to use it.

A note on IPython

IPython is an interactive Python shell that has many more features than the standard Python shell. The most useful of these is tab-completion, which will provide significant help in learning how to use Py6S. If you type part of an identifier - for example, `s.a` and press *TAB* you will see all of the possible completions. This works for all parts of Py6S, including the output values. For example:

```
In [3]: s.a<TAB>
s.aero_profile   s.altitudes      s.aot550         s.atmos_corr s.atmos_profile

In [4]: s.outputs.tra<TAB>
s.outputs.transmittance_aerosol_scattering   s.outputs.transmittance_oxygen
s.outputs.transmittance_ch4                  s.outputs.transmittance_ozone
s.outputs.transmittance_co                   s.outputs.transmittance_rayleigh_scattering
s.outputs.transmittance_co2                  s.outputs.transmittance_total_scattering
s.outputs.transmittance_global_gas           s.outputs.transmittance_water
s.outputs.transmittance_no2
```

It is also very easy to get help on individual parts of Py6S from within IPython. Simply type any name, and append a `?` to it - for example `AeroProfile.MultimodalLogNormal?`. The documentation which is shown will describe the parameters required and give an example of usage.

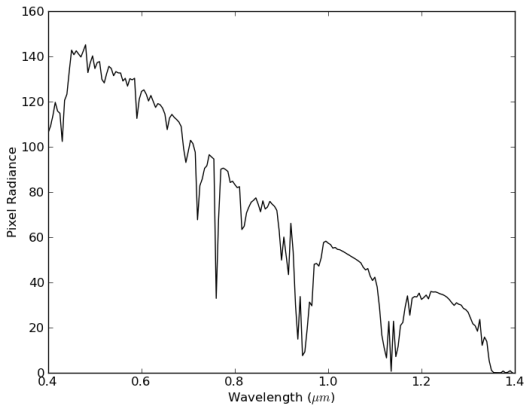
A first run

The *SixS* class is at the heart of Py6S. It has methods and attributes that allow you to set 6S parameters, run 6S and then view the outputs.

Py6S sets every 6S parameter to a sensible default, so the simplest possible code just uses the default values. As a nice introduction, we're going to plot one of the 6S outputs across the whole Visible-NIR wavelength range:

```
# Import all of the Py6S code
from Py6S import *
# Create a SixS object called s (used as the standard name by convention)
s = SixS()
# Run the 6S simulation defined by this SixS object across the
# whole VNIR range
wavelengths, results = SixSHelpers.Wavelengths.run_vnir(s, output_name="pixel_radiance")
# Plot these results, with the y axis label set to "Pixel Radiance"
SixSHelpers.Wavelengths.plot_wavelengths(wavelengths, results, "Pixel Radiance")
```

This will produce a graph like the following:



You will see a number of buttons in the window that is showing the graph. These allow you to zoom in to specific areas of the plot, move the plot around, adjust the margins, and save the plot to a file.

This shows the utility of Py6S very nicely - imagine how long it would have taken to produce this plot by editing and running 6S input files manually! However, the plot probably isn't particularly helpful as the defaults I've chosen probably aren't the parameters that you want to use for your simulation, and you may not be interested in the calculated pixel radiance. The sections below will explain how to alter this simple program to produce more useful results.

Setting parameters

We'll start with an example, and then explain the details:

```
from Py6S import *
s = SixS()
s.atmos_profile = AtmosProfile.PredefinedType(AtmosProfile.Tropical)
s.wavelength = Wavelength(0.357)
s.run()
print s.outputs.pixel_radiance
```

You can see here that we have changed the atmospheric profile to a pre-defined profile called 'Tropical', and changed the wavelength that we are using for the simulation to 0.357 micrometres. You can also see that here we're accessing the outputs directly, rather than running it over a specific wavelength range and plotting it. Try finding out what other outputs you can access, by typing `s.outputs.` and pressing *TAB*.

We'll look in detail at extracting outputs later, but first, lets have a look at the parameters that we can change, summarised in the table below:

SixS Parameter	Description	Possible values
`atmos_profile`	Atmospheric profile (pressure, water vapour, ozone etc)	Any outputs from *`AtmosProfile`*
`aero_profile`	Aerosol profile (types, distributions etc)	Any outputs from *`AeroProfile`*
`ground_reflectance`	Ground reflectance (Homogeneity, BRDF etc.)	Any outputs from *`GroundReflectance`*
`geometry`	Viewing/Illumination geometry (manual or satellite-specific)	A `Geometry*` class, for example *`Geometry.User`*
`aot550`	Aerosol Optical Thickness at 550nm	Floating point number
`visibility`	Visibility in km	Floating point number
`altitude`	Altitudes of the sensor and target	An instance of the `Altitudes` class
`atmos_corr`	Atmospheric correction settings (yes/no, reflectances)	Any outputs from *`AtmosCorr`*

As you can see, the parameter and class names are designed to be fairly self-explanatory. Using the details from above, a more advanced parameterisation is shown below:

```
from Py6S import *
s = SixS()
s.atmos_profile = AtmosProfile.UserWaterAndOzone(3.6, 0.9) # Set the atmosphere profile to be bas
s.wavelength = Wavelength(PredefinedWavelengths.LANDSAT_TM_B3) # Set the wavelength to be that of
s.ground_reflectance = GroundReflectance.HomogeneousWalthall(1.08, 0.48, 4.96, 0.5) # Set the sur
s.geometry = Geometry.Landsat_TM()
s.geometry.month = 7
s.geometry.day = 14
s.geometry.gmt_decimal_hour = 7.75
s.geometry.latitude = 51.148
s.geometry.longitude = 0.307
s.run()
print s.outputs.pixel_radiance
```

This is far more detailed, but should be self-explanatory given the comments and the table above. Far more details about the individual parameterisations are available in their documentation pages.

The real power of Py6S comes when you combine the paramterisation abilities of Py6S with the standard Python programming constructs. This is basically what we did above for the `run_vnir` example, although there we used a `SixSHelpers` method to make it easier for us. We can also do this manually, for example, you can easily loop over a number of parameter values and produce the outputs for each of them:

```
from Py6S import *
s = SixS()

for param in [AtmosProfile.Tropical, AtmosProfile.MidlatitudeSummer, AtmosProfile.MidlatitudeWinte
  s.atmos_profile = AtmosProfile.PredefinedType(param)
  s.run()
  print s.outputs.pixel_radiance
```

You can see that in this instance the change in pixel radiance over different atmospheric profiles is fairly low (< 0.8). Again, this saves a lot of time and complex input file editing.

That's it for the quick guide to setting parameters - for more details see the rest of the documentation.

Accessing outputs

The outputs from the 6S model are available under the `s.outputs` attribute. The outputs are actually stored as dictionaries, and the main set of outputs can be printed (and saved) from the `s.outputs.values` attribute. For example:

```
from Py6S import *
s = SixS()
s.run()
print s.outputs.values
```

However, it's normally more useful to access individual outputs. This can be done using the standard Python dictionary access methods - for example, `print s.outputs.values['pixel_radiance']`, but it is generally easy to do this by appending the output name to `s.outputs.`. For example:

```
from Py6S import *
s = SixS()
s.run()
print s.outputs.pixel_radiance
print s.outputs.environmental_irradiance
print s.outputs.total_gaseous_transmittance
```

The outputs stored under `s.outputs.values` are the main outputs of 6S provided on the first two 'screenfulls' of raw 6S output. The names of the outputs in Py6S have been kept as similar to the labels in the raw 6S output as possible, although sometimes names have been changed to improve clarity. Remember that a list of all possible outputs can be gained by typing `s.outputs.` and pressing *TAB* in IPython.

The tables showing the integrated values of various transmittances (rayleigh, water, ozone etc) are stored under the `s.outputs.trans` dictionary as instances of the `Transmittance` class. This allows the easy storage of the three different transmittances: downward, upward and total. Again, rather than dealing with the dictionary directly, courtesy methods are provided, for example:

```
from Py6S import *
s = SixS()
s.run()
print s.outputs.transmittance_rayleigh_scattering
print s.outputs.transmittance_rayleigh_scattering.downward
print s.outputs.transmittance_rayleigh_scattering.upward
print s.outputs.transmittance_rayleigh_scattering.total
print s.outputs.transmittance_water
```

Outputs from the other large grid shown in the raw 6S output, which includes outputs like spherical albedo, total optical depth and polarized reflectance, are also available:

```
from Py6S import *
s = SixS()
s.run()
print s.outputs.spherical_albedo
print s.outputs.optical_depth_total
print s.outputs.polarized_reflectance
```

SixSHelpers

A number of 'helper' methods have been written to make it easier to perform common operations using Py6S. These can be split into two categories:

Running for a set of parameters

It is often necessary to run a simulation across a number of wavelengths - as it is very rare that we are only interested in a single wavelength. We saw an example of this above, when we used the `run_vnir` method to run

a simulation across the Visible-NIR wavelengths. We can do similar things for other wavelengths really easily. For example:

```
from Py6S import *
s = SixS()
# Run for the whole range of wavelengths that 6S supports
wv, res = SixSHelpers.Wavelengths.run_whole_range(s, output_name='pixel_radiance')
# Do the same, but at a coarser resolution, so that it's quicker
wv, res = SixSHelpers.Wavelengths.run_whole_range(s, spacing=0.030, output_name='pixel_radiance')
# Run for the Landsat TM bands
wv, res = SixSHelpers.Wavelengths.run_landsat_tm(s, output_name='pixel_radiance')
```

Py6S supports running across all of the bands for all of the sensors that 6S supports - see the documentation for `SixSHelpers.Wavelengths` for more details.

You can plot the results really easily too, just by passing the resulting wavelengths and results to the *`SixSHelpers.Wavelengths.plot_wavelengths()`* function:

```
wv, res = SixSHelpers.Wavelengths.run_landsat_tm(s, output_name='pixel_radiance')
# Plot the results, setting the y-axis label appropriately
SixSHelpers.Wavelengths.plot_wavelengths(wv, res, 'Pixel radiance ($W/m^2$)')
```

You'll note that all of the `run_xxx` methods require a *`SixS`* instance as the first argument, and then an optional `output_name` argument. This specifies the output that you want to return from the function, and should whatever you would put after `s.outputs.` to print the output. For example, the output name could be any of the following:

```
pixel_reflectance
background_reflectance
transmittance_co2.downward
```

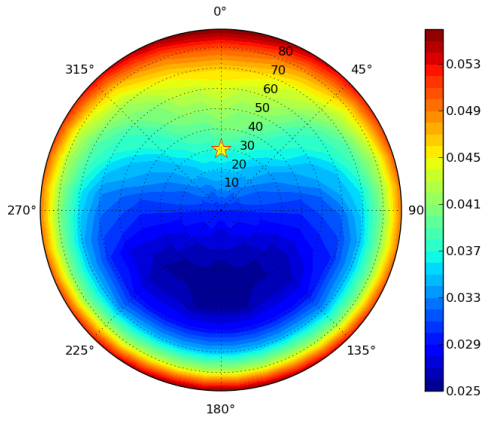
If you don't set the `output_name` argument then the function will return lots of *`Outputs`* instances rather than actual values. This can be handy if you want to work with lots of the outputs from a simulation, as it saves you having to run the whole simulation many times. For example:

```
s = SixS()
# Run for the whole range (takes a long time!)
wv, res = SixSHelpers.Wavelengths.run_landsat_tm(s)
# Look at what is in the results list - it should be an outputs instance
print res[0]
# We can't do anything with the outputs instances directly, but lets
# extract some outputs - we can do all of this without having to run
# the whole simulation again, as the res variable is storing all of the
# outputs
refl = SixSHelpers.Wavelengths.extract_output(res, "pixel_reflectance")
rad = SixSHelpers.Wavelengths.extract_output(res, "pixel_radiance")
SixSHelpers.Wavelengths.plot_wavelengths(wv, refl, "Pixel reflectance")
SixSHelpers.Wavelengths.plot_wavelengths(wv, rad, "Pixel radiance")
```

Another common use is to simulate a number of different view or solar angles, to examine the changes in the reflectance of a target due to its Bi-Directional Reflectance Distribution Factor. Doing this manually can be very tricky, as many simulations must be run, and then the results must be put into the right format to be plotted. Py6S makes this nice and easy by reducing it to one function call:

```
from Py6S import *
s = SixS()
# Set the ground reflectance to have some sort of BRDF, or the plot will
# be really boring! In this case, we're using the Roujean model
s.ground_reflectance = GroundReflectance.HomogeneousRoujean(0.037, 0.0, 0.133)
# Run the model and plot the results, varying the view angle (the other
#  option is to vary the solar angle) and plotting the pixel reflectance.
SixSHelpers.Angles.run_and_plot_360(s, 'view', 'pixel_reflectance')
```

This will produce a plot like the following:



Using real-world measurements to parameterise Py6S

Another common task is to parameterise 6S with some values collected from real-world measurements, so that the results of 6S simulations can be directly related to measurements in the field.

Py6S provides two ways to parameterise 6S from real-world measurements:

- By importing radiosonde data to set the atmospheric profile, using the *import_uow_radiosonde_data()* function
- By importing AERONET data to set the aerosol profile, using the *import_aeronet_data()* function

Detailed descriptions of these functions are given on their respective pages.

The SixS class

The *SixS* class is the heart of Py6S. It has attributes and methods that allow you to set parameters, run 6S and access the outputs. These are described in detail below, but the basic usage pattern is:

```
from Py6S import *
s = SixS() # Instantiate the class
s.aero_profile = AeroProfile.PredefinedType(AeroProfile.Maritime) # Set various parameters
s.run() # Run the model
print s.outputs.pixel_irradiance # Access the outputs
```

class Py6S.**SixS**(*path=None*)

Wrapper for the 6S Radiative Transfer Model.

This is the main class which can be used to instantiate an object which has the key methods for running 6S.

The most import method in this class is the *run()* method which writes the 6S input file, runs the model and processes the output.

The parameters of the model are set as the attributes of this class, and the outputs are available as attributes under the output attribute.

For a simple test to ensure that Py6S has found the correct executable for 6S simply run the *test()* method of this class:

```
SixS.Test()
```

Attributes:

- atmos_profile – The atmospheric profile to use. Should be set to the output of an AtmosProfile method. For example:

```
s.atmos_profile = AtmosProfile.PredefinedType(AtmosProfile.MidlatitudeSummer)
```

- `aero_profile` – The aerosol profile to use. Should be set to the output of an AeroProfile method. For example:

```
s.aero_profile = AeroProfile.PredefinedType(AeroProfile.Urban)
```

- `ground_reflectance` – The ground reflectance to use. Should be set to the output of a GroundReflectance method. For example:

```
s.ground_reflectance = GroundReflectance.HomogeneousLambertian(0.3)
```

- `geometry` – The geometrical settings, including solar and viewing angles. Should be set to an instance of a Geometry class, which can then have various attributes set. For example:

```
s.geometry = GeometryUser()
s.geometry.solar_z = 35
s.geometry.solar_a = 190
```

- `altitudes` – The settings for the sensor and target altitudes. This should be set to an instance of the `Altitudes()` class, which can then have various attributes set. For example:

```
s.altitudes = Altitudes()
s.altitudes.set_target_custom_altitude(2.3)
s.altitudes.set_sensor_sea_level()
```

- `wavelength` – The wavelength settings. Should be set to the output of the `Wavelength()` method. For example:

```
s.wavelength = Wavelength(0.550)
```

- `atmos_corr` – The settings for whether to perform atmospheric correction or not, and the parameters for this correction. Should be set to the output of a AtmosCorr method. For example:

```
s.atmos_corr = AtmosCorr.AtmosCorrLambertianFromReflectance(0.23)
```

produce_debug_report()
Prints out information about the configuration of Py6S generally, and the current SixS object specifically, which will be useful when debugging problems.

run()
Runs the 6S model and stores the outputs in the output variable.

May raise an `ExecutionError` if the 6S executable cannot be found.

classmethod test(*path=None*)
Runs a simple test to ensure that 6S and Py6S are installed correctly.

write_input_file(*filename=None*)
Generates a 6S input file from the parameters stored in the object and writes it to the given filename.

The input file is guaranteed to be a valid 6S input file which can be run manually if required

Attributes

sixs_path
The sixs path

Accessing outputs

The *Outputs* class stores the outputs produced by a 6S run and allows easy access to them.

class Py6S.**Outputs**(*stdout*, *stderr*)
Stores the output from a 6S run.

Attributes:

- `fulltext` – The full output of the 6S executable. This can be written to a file with the write_output_file method.
- `values` – The main outputs from the 6S run, stored in a dictionary. Accessible either via standard dictionary notation (`s.outputs.values['pixel_radiance']`) or as attributes (`s.outputs.pixel_radiance`)

Methods:

- `__init__()` – Constructor which takes the stdout and stderr from the model and processes it into the numerical outputs.
- *`extract_results()`* – Function called by the constructor to parse the output into individual variables
- *`to_int()`* – Convert a string to an int, so that it works even if passed a float.
- *`write_output_file()`* – Write the full textual output of the 6S model to a file.

extract_aot(*data*)
Extracts the AOT from the visibility and AOT line in the output.

extract_results()
Extract the results from the text output of the model and place them in the `values` dictionary.

extract_vis(*data*)
Extracts the visibility from the visibility and AOT line in the output

to_int(*str*)
Converts a string to an integer.

Does this by converting to float and then converting that to int, meaning that converting "5.00" to an integer will actually work.

Arguments:

- `str` – The string containing the number to convert to an integer

write_output_file(*filename*)
Writes the full textual output of the 6S model run to the specified filename.

Arguments:

- `filename` – The filename to write the output to

Parameter Setting

Parameters for 6S can be set using the classes and methods described below.

Atmospheric Profiles

class Py6S.**AtmosProfile**
Stores a enumeration for the pre-specified atmospheric model types

classmethod FromLatitudeAndDate(*latitude*, *date*)
Automatically pick the atmospheric profile based on the latitude and date.

Based on the table provided at http://www.exelisvis.com/docs/FLAASH.html

classmethod **PredefinedType**(*type*)
Set 6S to use a predefined atmosphere type.

Arguments:

•`type` – the predefined atmosphere type, one of the constants defined in this class

Example usage:

```
s.atmos_profile = AtmosProfile.PredefinedType(AtmosProfile.MidlatitudeSummer)
```

classmethod **RadiosondeProfile**(*data*)
Set 6S to use an atmosphere defined by a profile from a radiosonde measurements.

Arguments:

•**`data` – A dictionary containing five iterables (eg. lists) with the radiosonde measurements in them. The dict**

– `altitude` – in km

– `pressure` – in mb

– `temperature` – in k

– `water` – in g/m^3

– `ozone` – in g/m^3

There must be 34 items in each iterable, or a `ParameterExeception` will be thrown.

classmethod **UserWaterAndOzone**(*water*, *ozone*)
Set 6S to use an atmosphere defined by an amount of water vapour and ozone.

Arguments:

•`water` – The total amount of water in a vertical path through the atmosphere (in g/cm^2)

•`ozone` – The total amount of ozone in a vertical path through the atmosphere (in cm-atm)

Example usage:

```
s.atmos_profile = AtmosProfile.UserWaterAndOzone(3.6, 0.9)
```

Aerosol Profiles

class `Py6S.`**AeroProfile**
Class representing options for Aerosol Profiles

class **AerosolDistribution**(*rmin*, *rmax*, *numtype*)
Stores data regarding a specific Aerosol Distribution.

Used by the following methods:

•*MultimodalLogNormalDistribution()*

•*ModifiedGammaDistribution()*

•*JungePowerLawDistribution()*

add_component(*rmean*, *sigma*, *percentage_density*, *refr_real*, *refr_imag*)
Adds a component to the aerosol distribution.

Wavelength dependent values must be input at the following wavelengths (given in micrometers): 0.350, 0.400, 0.412, 0.443, 0.470, 0.488, 0.515, 0.550, 0.590, 0.633, 0.670, 0.694, 0.760, 0.860, 1.240, 1.536, 1.650, 1.950, 2.250, 3.750

Arguments:
•`rmean` – The mean radius of the aerosols
•`sigma` – Sigma, as defined by the distribution (Log Normal etc)

- `percentage_density` – The percentage density of the aerosol
- `refr_real` – A 20-element iterable giving the real part of the refractive indices at the specified wavelengths (see above)
- `refr_imag` – A 20-element iterable giving the imaginary part of the refractive indices at the specified wavelengths (see above)

classmethod `AeroProfile.`**`JungePowerLawDistribution`**(*rmin*, *rmax*)

Set 6S to use a Junge Power Law distribution.

Arguments:

- `rmin` – The minimum aerosol radius
- `rmax` – The maximum aerosol radius

This returns an *`AerosolDistribution`* object. Components can then be added to this distribution using the *`add_component()`* method of the returned class.

Example usage:

```
s.aeroprofile = AeroProfile.JungePowerLawDistribution(0.1, 0.3)
s.aeroprofile.add_component(...)
```

classmethod `AeroProfile.`**`ModifiedGammaDistribution`**(*rmin*, *rmax*)

Set 6S to use a Modified Gamma distribution.

Arguments:

- `rmin` – The minimum aerosol radius
- `rmax` – The maximum aerosol radius

This returns an *`AerosolDistribution`* object. Components can then be added to this distribution using the *`add_component()`* method of the returned class.

Example usage:

```
s.aeroprofile = AeroProfile.ModifiedGammaDistribution(0.3, 0.1)
s.aeroprofile.add_component(...)
```

classmethod `AeroProfile.`**`MultimodalLogNormalDistribution`**(*rmin*, *rmax*)

Set 6S to use a Multimodal Log-Normal distribution.

Arguments:

- `rmin` – The minimum aerosol radius
- `rmax` – The maximum aerosol radius

This returns an *`AerosolDistribution`* object. Components can then be added to this distribution using the *`add_component()`* method of the returned class.

Example usage:

```
s.aeroprofile = AeroProfile.MultimodalLogNormalDistribution(0.3, 0.1)
s.aeroprofile.add_component(...)
```

classmethod `AeroProfile.`**`PredefinedType`**(*type*)

Set 6S to use a predefined aerosol type, one of the constants defined in this class.

Arguments:

- `type` – the predefined aerosol type, one of the constants defined in this class

Example usage:

```
s.aeroprofile = AeroProfile.PredefinedType(AeroProfile.Urban)
```

classmethod AeroProfile.**SunPhotometerDistribution**(*r*, *dvdlogr*, *refr_real*, *refr_imag*)

Set 6S to use an aerosol parameterisation from Sun Photometer measurements.

The real and imaginary parts of the refractive indices must be input at the following wavelengths (given in micrometers): 0.350, 0.400, 0.412, 0.443, 0.470, 0.488, 0.515, 0.550, 0.590, 0.633, 0.670, 0.694, 0.760, 0.860, 1.240, 1.536, 1.650, 1.950, 2.250, 3.750

Arguments:

- `r` – A list of radius measurements from a sun photometer (microns)
- `dvdlogr` – A list of dV/d(logr) measurements from a sun photometer, for the radiuses as above (cm^3/cm^2/micron)
- `refr_real` – A list containing the real part of the refractive indices for each of the 20 wavelengths (above). If a single float value is given then the value is treated as constant for all wavelengths.
- `refr_imag` – A list containing the imaginary part of the refractive indices for each of the 20 wavelengths (above). If a single float value is given then the value is treated as constant for all wavelengths.

classmethod AeroProfile.**User**(***kwargs*)

Set 6S to use a user-defined aerosol profile based on proportions of standard aerosol components.

The profile is set as a mixture of pre-defined components, each given as an optional keyword. Not all keywords need to be given, but the values for the keywords given must sum to 1, or a ParameterError will be raised.

Optional keywords:

- `dust` – The proportion of dust-like aerosols
- `water` – The proportion of water-like aerosols
- `oceanic` – The proportion of oceanic aerosols
- `soot` – The proportion of soot-like aerosols

Example usage:

```
s.aeroprofile = AeroProfile.User(dust=0.3, oceanic=0.7)
s.aeroprofile = AeroProfile.User(soot = 0.1, water = 0.3, oceanic = 0.05, dust = 0.55)
```

class AeroProfile.**UserProfile**(*atype*)

Set 6S to use a user-defined aerosol profile, with differing AOTs over the height of the profile.

Arguments:

- `atype` – Aerosol type to be used for all layers. Must be one of the pre-defined types defined in this class.

Methods:

- *`add_layer()`* – Adds a layer to the user-defined aerosol profile, with the specified height and aerosol optical thickness.

Example usage:

```
s.aeroprofile = AeroProfile.UserProfile(AeroProfile.Maritime)
s.aeroprofile.add_layer(5, 0.34) # Add a 5km-thick layer with an AOT of 0.34
s.aeroprofile.add_layer(10, 0.7) # Add a 10km-thick layer with an AOT of 0.7
s.aeroprofile.add_layer(100, 0.01) # Add a 100km-thick layer with an AOT of 0.01
```

add_layer(*height*, *optical_thickness*)

Adds a layer to the user-defined profile.

Arguments:

- `height` – Height of the layer (in km)

- optical_thickness – Optical thickness of the layer

Example usage:

```
s.aeroprofile.add_layer(5, 0.34) # Add a 5km-thick layer with an AOT of 0.34
```

Ground Reflectances

class Py6S.**GroundReflectance**

Produces strings for the input file for a number of different ground reflectance scenarios.

Options are:

- Homogeneous
  - Lambertian
  - BRDF
    - Walthall et al. model
    - Rahman et al. model
    - etc
- Heterogeneous
  - Lambertian

These are combined to give function names like:

HomogeneousLambertian() or *HomogeneousWalthall()*

The standard functions (*HomogeneousLambertian()* and *HeterogeneousLambertian()*) will decide what to do based on the types of inputs they are given:

```
model.ground_reflectance = GroundReflectance.HomogeneousLambertian(0.7) # A spectrally-consta
model.ground_reflectance = GroundReflectance.HomogeneousLambertian(GroundReflectance.GreenVeg
model.ground_reflectance = GroundReflectance.HomogeneousLambertian([0.6, 0.8, 0.34, 0.453]) #
# A 2D ndarray, such as that returned by any of the Spectra.import_* functions
model.ground_reflectance = GroundReflectance.HomogeneousLambertian(Spectra.import_from_usgs("
```

classmethod **HeterogeneousLambertian**(*radius*, *ro_target*, *ro_env*)

Provides parameterisation for heterogeneous Lambertian (ie. uniform BRDF) surfaces.

These surfaces are modelled in 6S as a circular target surrounded by an environment of a different reflectance.

Arguments:

- radius – The radius of the target (in km)
- ro_target – The reflectance of the target
- ro_env – The reflectance of the environment

Both of the reflectances can be set to any of the following:

- A single float value (for example, 0.634), in which case it is interpreted as a spectrally-constant reflectance value.
- A constant defined by this class (one of GroundReflectance.GreenVegetation, GroundReflectance.ClearWater, GroundReflectance.Sand or GroundReflectance.LakeWater) in which case a built-in spectrum of the specified material is used.
- An array of values (for example, [0.67, 0.85, 0.34, 0.65]) in which case the values are taken to be reflectances across the whole wavelength range at a spacing of 2.5nm. In this case, if the start

wavelength is s and the end wavelength is e, the values must be given for the wavelengths: `s, s+2.5, s+5.0, s+7.5, ..., e-2.5, e`

•A multidimensional ndarray giving wavelength (column 0) and reflectance (column 1) values

classmethod HomogeneousHapke(*albedo*, *assymetry*, *amplitude*, *width*)

Parameterisation for a surface BRDF based on the Hapke model.

The parameters are:

•albedo

•assymetry parameter for the phase function

•amplitude of hot spot

•width of the hot spot

classmethod HomogeneousIaquintaPinty(*leaf_dist*, *hot_spot*, *lai*, *hot_spot_param*, *leaf_reflec*, *leaf_trans*, *soil_albedo*)

Parameterisation for a surface BRDF based on the Iaquinta and Pinty model.

The parameters are:

•Leaf distribution (one of the `GroundReflectance.LeafDistXXX` constants)

•Hot spot setting (`GroundReflectance.HotSpot` or `GroundReflectance.NoHotSpot`)

•Leaf Area Index (1-15)

•Hot spot parameter 2*r*lambda (0-2)

•Leaf reflectance (0-0.99)

•Leaf transmittance (0-0.99)

•Soil albedo (0-0.99)

Leaf reflectance + Leaf transmittance must be less than 0.99. If this is not the case, a `ParameterException` is raised.

classmethod HomogeneousKuuskMultispectralCR(*lai*, *lad_eps*, *lad_thm*, *relative_leaf_size*, *chlorophyll_content*, *leaf_water_equiv_thickness*, *effective_num_layers*, *ratio_refractive_indices*, *weight_first_price_function*)

Parameterisation for a surface BRDF based on Kuusk's multispectral CR model.

The Parameters are:

•Leaf Area Index (0.1-10)

•LAD eps (0.0-0.9)

•LAD thm (0.0-90.0)

•Relative leaf size (0.01-1.0)

•Chlorophyll content (ug/cm^2, 0-30)

•Leaf water equivalent thickness (0.01-0.03)

•Effective number of elementary layers inside a leaf (1-225)

•Ratio of refractive indices of the leaf surface wax and internal material (0-1.0)

•Weight of the 1st Price function for the soil reflectance (0.1-0.8)

classmethod HomogeneousLambertian(*ro*)

Provides parameterisation for homogeneous Lambertian (ie. uniform BRDF) surfaces.

The single argument can be either:

- A single float value (for example, 0.634), in which case it is interpreted as a spectrally-constant reflectance value.
- A constant defined by this class (one of `GroundReflectance.GreenVegetation`, `GroundReflectance.ClearWater`, `GroundReflectance.Sand` or `GroundReflectance.LakeWater`) in which case a built-in spectrum of the specified material is used.
- An array of values (for example, [0.67, 0.85, 0.34, 0.65]) in which case the values are taken to be reflectances across the whole wavelength range at a spacing of 2.5nm. In this case, if the start wavelength is s and the end wavelength is e, the values must be given for the wavelengths: `s, s+2.5, s+5.0, s+7.5, ..., e-2.5, e`
- A multidimensional ndarray giving wavelength (column 0) and reflectance (column 1) values

classmethod HomogeneousMODISBRDF(*par1*, *par2*, *par3*)
Parameterisation for a surface BRDF based on the MODIS Operational BRDF model.

The parameters are:

- Weight for lambertian kernel
- Weight for Ross Thick kernel
- Weight for Li Spare kernel

classmethod HomogeneousMinnaert(*k*, *alb*)
Parameterisation for a surface BRDF based on the Minnaert BRDF model.

The parameters are: - K surface parameter - Surface albedo

classmethod HomogeneousOcean(*wind_speed*, *wind_azimuth*, *salinity*, *pigment_concentration*)
Parameterisation for a surface BRDF based on the Ocean BRDF model.

The parameters are:

- wind speed (in m/s)
- azimuth of the wind (in degrees)
- salinity (in ppt) (set to 34.3ppt if < 0)
- pigment concentration (in mg/m3)

classmethod HomogeneousRahman(*intensity*, *asymmetry_factor*, *structural_parameter*)
Parameterisation for a surface BRDF based on the Rahman BRDF model.

The parameters are:

- Intensity of the reflectance of the surface (N/D value >= 0)
- Asymmetry factor, N/D value between -1.0 and 1.0
- Structural parameter of the medium

classmethod HomogeneousRoujean(*albedo*, *k1*, *k2*)
Parameterisation for a surface BRDF based on the Roujean et al. model.

The parameters are:

- albedo
- geometric parameter for hot spot effect
- geometric parameter for hot spot effect

classmethod HomogeneousUserDefined(*observed_reflectance*, *albedo*, *ro_sun_at_thetas*, *ro_sun_at_thetav*)
Parameterisation for a user-defined surface BRDF.

The parameters are:

- *observed_reflectance* – Observed reflectance in the geometry specified in the Geometry parameterisation
- *albedo* – Surface spherical albedo
- *ro_sun_at_thetas* – A reflectance table (described below) for the scenario when the sun is at theta_s (the solar zenith angle specified in the Geometry parameterisation)
- *ro_sun_at_thetav* – A reflectance table (described below) for the scenario when the sun is at theta_v (the view zenith angle specified in the Geometry parameterisation)

The reflectance tables mentioned above must be NumPy arrays (that is, instances of `ndarray`) with a shape of (10, 13) where the table headers are as below, and each cell contains the reflectance of the surface in the specified geometry:

```
          zenith
          0   10    20    30    40    50    60    70    80    85
      a 0
      z 30
      i 60
      m 90
      u 120
      t 150
      h .
        .
        .
```

classmethod **HomogeneousVerstaeteEtAl**(*kappa_param*, *phase_funct*, *scattering_type*, *leaf_area_density*, *sun_flecks_radius*, *ssa*, *legendre_first*, *legendre_second*, *k1*, *k2*, *asym_factor*, *chil*)

Parameterisation for a surface BRDF based on the Verstraete, Pinty and Dickinson model.

The parameters are:

- The type of Kappa parameterisation (one of the `GroundReflectance.KappaXXX` constants)
- The phase function to use (one of the `GroundReflectance.PhaseXXX` constants)
- The scattering type to use (either `GroundReflectance.SingleScatteringOnly` or `GroundReflectance.DickinsonMultipleScattering`)
- Leaf area density (m^2/m^-3)
- Radius of the sun flecks on the scatterer (m)
- Single Scattering Albedo (0-1)
- First coefficient of Legendre polynomial (Only used if phase function is not `GroundReflectance.PhaseIsotropic`, set to `None` otherwise)
- Second coefficient of Legendre polynomial (Only used if phase function is not `GroundReflectance.PhaseIsotropic`, set to `None` otherwise)
- Kappa value k1 (Only used if Kappa parameterisation was `GroundReflectance.KappaGivenValues`, set to `None` otherwise)
- Kappa value k2 (Only used if Kappa parameterisation was `GroundReflectance.KappaGivenValues`, set to `None` otherwise)
- Asymmetry factor for Heyney-Greenstein parameterisation (Only used if Phase function is set to `GroundReflectance.PhaseHeyneyGreenstein`, set to `None` otherwise)
- Goudriaan's chil parameter (Only used if Kappa parameterisation was NOT `GroundReflectance.KappaGivenValues`, set to `None` otherwise)

classmethod **HomogeneousWalthall**(*param1*, *param2*, *param3*, *albedo*)

Parameterisation for a surface BRDF based on the Walthall et al. model.

The parameters are:

- term in square ts*tv
- term in square ts*ts+tv*tv
- term in ts*tv*cos(phi) (limacon de pascal)
- albedo

Geometries

class Py6S.**Geometry**

class **AVHRR_AM**
Stores parameters for a AVHRR morning pass geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- column – The AVHRR column of the image
- ascendant_node_longitude – The longitude of the ascendant node of the image
- ascendant_node_hour – The hour of the ascendant node of the image

class Geometry.**AVHRR_PM**
Stores parameters for a AVHRR afternoon pass geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- column – The AVHRR column of the image
- ascendant_node_longitude – The longitude of the ascendant node of the image
- ascendant_node_hour – The hour of the ascendant node of the image

class Geometry.**GoesEast**
Stores parameters for a GOES East geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- gmt_decimal_hour – The time in GMT, as a decimal, in hours (eg. 7.5 for 7:30am)
- column – The GOES East column of the image
- line – The GOES East line of the image

class Geometry.**GoesWest**
Stores parameters for a GOES West geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- gmt_decimal_hour – The time in GMT, as a decimal, in hours (eg. 7.5 for 7:30am)
- column – The GOES West column of the image

- line – The GOES West line of the image

class Geometry.**Landsat_TM**
Stores parameters for a Landsat TM geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- gmt_decimal_hour – The time in GMT, as a decimal, in hours (eg. 7.5 for 7:30am)
- latitude – The latitude of the centre of the image
- longitude – The longitude of the centre of the image

class Geometry.**Meteosat**
Stores parameters for a Meteosat geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- gmt_decimal_hour – The time in GMT, as a decimal, in hours (eg. 7.5 for 7:30am)
- column – The Meteosat column of the image
- line – The Meteosat line of the image

class Geometry.**SPOT_HRV**
Stores parameters for a SPOT HRV geometry for 6S.

Attributes:

- month – The month the image was acquired in (0-12)
- day – The day the image was acquired in (1-31)
- gmt_decimal_hour – The time in GMT, as a decimal, in hours (eg. 7.5 for 7:30am)
- latitude – The latitude of the centre of the image
- longitude – The longitude of the centre of the image

class Geometry.**User**
Stores parameters for a user-defined geometry for 6S.

Attributes:

- solar_z – Solar zenith angle
- solar_a – Solar azimuth angle
- view_z – View zenith angle
- view_a – View azimuth angle
- day – The day the image was acquired in (1-31)
- month – The month the image was acquired in (0-12)

from_time_and_location(*lat*, *lon*, *datetimestring*, *view_z*, *view_a*)
Sets the user-defined geometry to a given view zenith and azimuth, and a solar zenith and azimuth calculated from the lat, lon and date given.

Uses the PySolar module for the calculations.

Arguments:
- lat – The latitude of the location (0-90 degrees)
- lon – The longitude of the location

- `datetimestring` – Any string that can be parsed to produce a date/time object. All that is really needed is a time - eg. "14:53"
- `view_z` – The view zenith angle
- `view_a` – The view azimuth angle

Altitudes

class `Py6S.`**`Altitudes`**

Allows the specification of target and sensor altitudes.

`set_sensor_custom_altitude`(*altitude*, *aot=-1*, *water=-1*, *ozone=-1*)

Set the altitude of the sensor, along with other variables required for the parameterisation of the sensor.

Takes optional arguments of *aot*, *water* and *ozone* to specify atmospheric contents underneath the sensor. If these aren't specified then the water and ozone contents will be interpolated from the US-1962 standard atmosphere, and the AOT will be interpolated from a 2km exponential aerosol profile.

Arguments:

- *altitude* – The altitude of the sensor, in km.
- *aot* – (Optional, keyword argument) The AOT at 550nm at the sensor
- *water* – (Optional, keyword argument) The water vapour content (in g/cm^2) at the sensor
- *ozone* – (Optional, keyword argument) The ozone content (in cm-atm) at the sensor

Example usage:

```
s.altitudes.set_sensor_custom_altitude(8, 0.35, 1.6, 0.4) # Altitude of 8km, AOT of 0.35,
```

`set_sensor_satellite_level`()

Set the sensor altitude to be satellite level.

`set_sensor_sea_level`()

Set the sensor altitude to be sea level.

`set_target_custom_altitude`(*altitude*)

Set the altitude of the target.

Arguments:

- *altitude* – The altitude of the target, in km

`set_target_pressure`(*pressure*)

Set the pressure of the target (a proxy for the height of the target).

Arguments:

- *pressure* – The pressure at the target, in mb

`set_target_sea_level`()

Set the altitude of the target to be at sea level (0km)

Wavelengths

class `Py6S.`**`Wavelength`**

Select one or more wavelengths for the 6S simulation.

There are a number of ways to do this:

1. Pass a single value of a wavelength in micrometres. The simulation will be performed for just this wavelength:

```
Wavelength(0.43)
```

2. Pass a start and end wavelength in micrometres. The simulation will be performed across this wavelength range with a constant filter function (spectral response function) of 1.0:

```
Wavelength(0.43, 0.50)
```

3. Pass a start and end wavelength, and a filter given at 2.5nm intervals. The simulation will be performed across this wavelength range using the given filter function:

```
Wavelength(0.400, 0.410, [0.7, 0.9, 1.0, 0.3, 0.15])
```

The filter function must include values for the start and end wavelengths, plus values every 2.5nm (0.0025um) in between. So, in the example above, there are five values given: one each for 0.400, 0.4025, 0.405, 0.4075, 0.410.

4. Pass a constant (as defined in this class) for a pre-defined wavelength range:

```
Wavelength(PredefinedWavelengths.LANDSAT_TM_B1)
```

Atmospheric Corrections

class Py6S.**AtmosCorr**

Class representing options for selecting atmospheric correction settings for 6S.

classmethod AtmosCorrBRDFFromRadiance(*radiance*)

Set 6S to perform atmospheric correction using a fully BRDF-represented surface, using a given radiance value.

Arguments: * `radiance` – Radiance of the surface

classmethod AtmosCorrBRDFFromReflectance(*reflectance*)

Set 6S to perform atmospheric correction using a fully BRDF-represented surface, using a given reflectance value.

Arguments: * `reflectance` – Reflectance of the surface.

classmethod AtmosCorrLambertianFromRadiance(*radiance*)

Set 6S to perform atmospheric correction assuming a Lambertian surface, using a given radiance value.

Arguments: * `radiance` – Radiance of the surface.

classmethod AtmosCorrLambertianFromReflectance(*reflectance*)

Set 6S to perform atmospheric correction assuming a Lambertian surface, using a given reflectance value.

Arguments: * `reflectance` – Reflectance of the surface.

classmethod NoAtmosCorr()

Set 6S not to perform any atmospheric correction

Helper methods

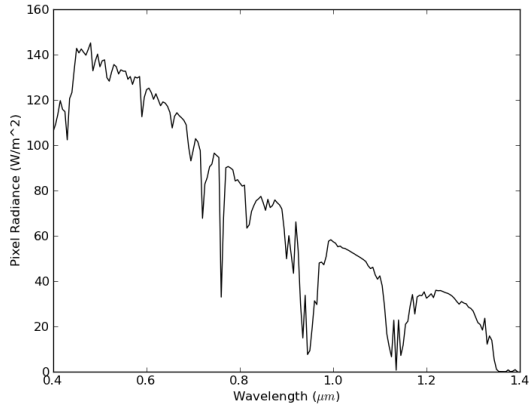
The SixSHelpers module contains a number of helper functions that improve the ease-of-use of Py6S. These include functions to set 6S parameters from various external data sources, as well as functions to make it easy to produce wavelength and BRDF plots from 6S runs.

Running for many wavelengths

The Wavelengths class contains functions to run 6S over a number of wavelength ranges.

For example, the following code runs 6S simulations across the Visible-Near Infrared wavelength range and plots the results, producing the output shown below:

```
from Py6S import *
s = SixS()
s.aero_profile = AeroProfile.PredefinedType(AeroProfile.Maritime)
wavelengths, values = SixSHelpers.Wavelengths.run_vnir(s, output_name='pixel_radiance')
SixSHelpers.Wavelengths.plot_wavelengths(wavelengths, values, 'Pixel Radiance (W/m^2)')
```



A similar function exist to run across the whole 6S wavelength range (*run_whole_range()*), and arbritary lists of wavelengths can be run using the *run_wavelengths()* function. For example, you can manually specify a number of wavelengths to run for:

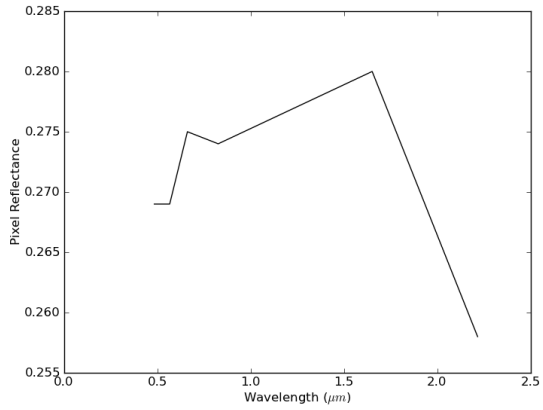
```
wv, res = SixSHelpers.Wavelengths.run_wavelengths(s, [0.46, 0.67, 0.98], output_name='apparent_ra
```

If you want to run user-specific ranges of wavelengths then you can use the handy numpy functions arange and linspace to generate the lists of wavelengths for you. For example:

```
# Run the model between 0.5 and 0.7um with a step of 0.001um (1nm)
wv, res = SixSHelpers.Wavelengths.run_wavelengths(s, np.arange(0.5, 0.7, 0.001), output_name='app

# Run the model at 50 equally-spaced wavelengths in the range 0.9-1.5um
wv, res = SixSHelpers.Wavelengths.run_wavelengths(s, np.linspace(0.9, 1.5, 50), output_name='appa
```

Functions also exist to run for all bands of the various sensors supported in 6S (for example, *run_landsat_tm()*, *run_modis()* and *run_aatsr()*). It should be noted that for these functions, bands which are outside of the 6S wavelength range (0.2-4.0um), such as the Landsat thermal band, will not be simulated. The example below shows the creation of a plot for the Landsat ETM bands:

```
from Py6S import *
s = SixS()
s.aero_profile = AeroProfile.PredefinedType(AeroProfile.Maritime)
wavelengths, values = SixSHelpers.Wavelengths.run_landsat_etm(s, output_name='pixel_reflectance')
SixSHelpers.Wavelengths.plot_wavelengths(wavelengths, values, 'Pixel Reflectance')
```



The supported sensors are:

- Landsat MSS, TM and ETM
- SPOT HRV1, HRV2 and Vegetation
- MERIS
- MODIS
- POLDER
- SeaWiFS
- AATSR
- ASTER
- VIIRS
- ER2 MODIS Airborne Simulator (MAS)
- ALI
- GLI

class `Py6S.SixSHelpers.`**`Wavelengths`**

Helper functions for running the 6S model for a range of wavelengths, and plotting the result

classmethod **`extract_output`**(*results*, *output_name*)

Extracts data for one particular SixS output from a list of SixS.Outputs instances.

Basically just a wrapper around a list comprehension.

Arguments:

- `results` – A list of `SixS.Outputs` instances
- `output_name` – The name of the output to extract. This should be a string containing whatever is put after the *s.outputs* when printing the output, for example *'pixel_reflectance'*.

classmethod **`plot_wavelengths`**(*wavelengths*, *values*, *y_axis_label*)

Plot the given wavelengths and values, such as those produced by the other functions in this class.

Arguments:

- `wavelengths` – A list of wavelengths (in um)
- `values` – A corresponding list of values at the wavelengths above
- `y_axis_label` – A string containing tha axis label to use for the Y axis

Example usage:

```
SixSHelpers.PredefinedWavelengths.plot_wavelengths(wavelengths, values, 'Pixel Radiance
```

classmethod **`run_aatsr`**(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the AATSR bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_ali(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the ALI bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_aqua(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the MODIS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_aster(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the ASTER bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_er2_mas(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the ER2 MODIS Airborne Simulator (MAS) bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_gli(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the GLI bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_landsat_etm(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Landsat ETM bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_landsat_mss(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Landsat MSS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_landsat_oli(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Landsat TM bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_landsat_tm(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Landsat TM bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_meris(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the MERIS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_modis(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the MODIS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_polder(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the POLDER bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod **run_s2a_msi**(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Sentinel2/MSI bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod **run_s3a_olci**(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Sentinel3/OLCI bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod **run_s3a_slstr**(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the Sentinel3/SLSTR bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod **run_seawifs**(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the SeaWiFS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_spot_hrv(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the SPOT HRV (both 1 and 2, as the only bands specified are the same for both) bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_spot_vgt(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the SPOT Vegetation bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_terra(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the MODIS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_viirs(*s*, ***kwargs*)

Runs the given SixS parameterisation for all of the VIIRS bands within the 6S band range, optionally extracting a specific output.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the centre wavlengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

classmethod run_vnir (*s*, *spacing=0.005*, ***kwargs*)

Runs the given SixS parameterisation for wavelengths over the Visible-Near Infrared range, optionally extracting a specific output.

By default, the given model is run for wavelengths from 0.4-1.4um, with a spacing of 5nm.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `spacing` – (Optional) The spacing to use between each wavelength, in um. Eg. a spacing of 0.001 is a spacing of 1nm.
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the wavelengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

Example usage:

```
# Run for the VNIR wavelengths, with the default spacing (5nm), returning SixS.Outputs ir
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_vnir(s)
# Run for the VNIR wavelengths, with a spacing of 10nm, returning pixel reflectance value
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_vnir(s, spacing=0.010, outpu
```

classmethod run_wavelengths (*s*, *wavelengths*, *output_name=None*, *n=None*, *verbose=True*)

Runs the given SixS parameterisation for each of the wavelengths given, optionally extracting a specific output.

This function is used by all of the other wavelengths running functions, such as **:method:'run_vnir'**, and thus any arguments that are passed to this function can also be passed to these other functions.

The calls to 6S for each wavelength will be run in parallel, making this function far faster than simply running a for loop over each wavelength.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `wavelengths` – An iterable containing the wavelengths to iterate over
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`
- `n` – (Optional) The number of threads to run in parallel. This defaults to the number of CPU cores in your system, and is unlikely to need changing.
- `verbose` – (Optional) Print wavelengths as Py6S is running (default=True)

Return value:

A tuple containing the wavelengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

Example usage:

```
# Run for all wavelengths from 0.4 to 0.5 micrometers, with a spacing of 1nm, returns Six
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_wavelengths(s, np.arange(0.4
# Run for all wavelengths from 0.4 to 0.5 micrometers, with a spacing of 1nm, returns a l
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_wavelengths(s, np.arange(0.4
# Run for the first three Landsat TM bands
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_wavelengths(s, [PredefinedWa
```

classmethod run_whole_range (*s*, *spacing=0.01*, ***kwargs*)

Runs the given SixS parameterisation for the entire wavelength range of the 6S model, optionally extracting a specific output.

By default, the given model is run for wavelengths from 0.2-4.0um, with a spacing of 10nm.

Arguments:

- `s` – A *SixS* instance with the parameters set as required
- `spacing` – (Optional) The spacing to use between each wavelength, in um. Eg. a spacing of 0.001 is a spacing of 1nm.
- `output_name` – (Optional) The output to extract from `s.outputs`, as a string that could be placed after `s.outputs.`, for example `pixel_reflectance`

Return value:

A tuple containing the wavelengths used for the run and the results of the simulations. The results will be a list of `SixS.Outputs` instances if `output_name` is not set, or a list of values of the selected output if `output_name` is set.

Example usage:

```
# Run for the VNIR wavelengths, with the default spacing (5nm), returning SixS.Outputs ir
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_vnir(s)
# Run for the VNIR wavelengths, with a spacing of 10nm, returning pixel reflectance value
wavelengths, results = SixSHelpers.PredefinedWavelengths.run_vnir(s, spacing=0.010, outpu
```

classmethod to_centre_wavelengths (*item*)

Get centre wavelengths for a sensor from a list of the wavelength tuples.

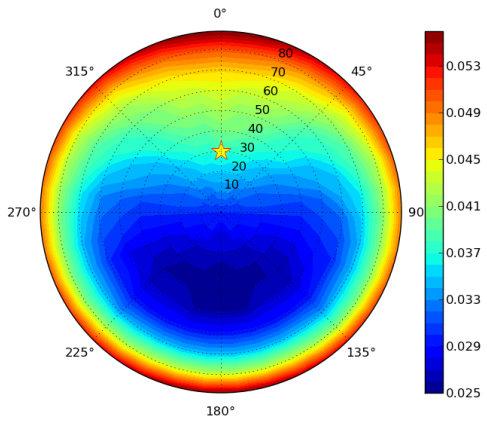
This is calculated simple as minwv+((maxwv-minwv)/2) and is the CENTER wavelength (that is the wavelength in the middle of the range) not necessarily the peak wavelength.

Running for many angles

The Angles class contains functions to run 6S over a number of different angles, which are particularly useful when dealing with surfaces with a modelled-BRDF.

For example, the following code runs 6S simulations for many view zenith and azimuth angles and plots a polar contour plot of the resulting reflectance distribution. The output is shown below:

```
s = SixS()
s.ground_reflectance = GroundReflectance.HomogeneousRoujean(0.037, 0.0, 0.133)
s.geometry.solar_z = 30
s.geometry.solar_a = 0
SixSHelpers.Angles.run_and_plot_360(s, 'view', 'pixel_reflectance')
```



class Py6S.SixSHelpers.**Angles**

classmethod extract_output(*results*, *output_name*)
Extracts data for one particular SixS output from a list of SixS.Outputs instances.

Basically just a wrapper around a list comprehension.

Arguments:

- `results` – A list of `SixS.Outputs` instances
- `output_name` – The name of the output to extract. This should be a string containing whatever is put after the *s.outputs* when printing the output, for example *'pixel_reflectance'*.

classmethod plot360(*data*, *output_name=None*, *show_sun=True*, *colorbarlabel=None*)
Plot the data returned from `run360()` as a polar contour plot, selecting an output if required.

Arguments:

- `data` – The return value from `run360()`
- `output_name` – (Optional) The output name to extract (eg. "pixel_reflectance") if the given data is provided as instances of the Outputs class
- `show_sun` – (Optional) Whether to show the location of the sun on the resulting polar plot.
- `colorbarlabel` – (Optional) The label to use on the color bar shown with the plot

classmethod plot_polar_contour(*values*, *azimuths*, *zeniths*, *filled=True*, *colorbarlabel=''*)
Plot a polar contour plot, with 0 degrees at the North.

Arguments:

- `values` – A list (or other iterable - eg. a NumPy array) of the values to plot on the contour plot (the *z* values)
- `azimuths` – A list of azimuths (in degrees)
- `zeniths` – A list of zeniths (that is, radii)
- `filled` – (Optional) Whether to plot a filled contour plot, or just the contours (defaults to filled)
- `yaxislabel` – (Optional) The label to use for the colorbar
- `colorbarlabel` – (Optional) The label to use on the color bar shown with the plot

The shapes of these lists are important, and are designed for a particular use case (but should be more generally useful). The values list should be *len(azimuths) * len(zeniths)* long with data for the first azimuth for all the zeniths, then the second azimuth for all the zeniths etc.

This is designed to work nicely with data that is produced using a loop as follows:

```
values = []
for azimuth in azimuths:
  for zenith in zeniths:
    # Do something and get a result
    values.append(result)
```

After that code the azimuths, zeniths and values lists will be ready to be passed into this function.

plot_principal_plane(*zeniths*, *values*, *y_axis_label*)
Plot the results from a principal plane simulation (eg. a run of *run_principal_plane()*).

Arguments:

- `zeniths` – A list of view zenith angles in degrees
- `values` – A list of simulated values for each of these angles
- `y_axis_label` – A string to use as the label for the y axis

classmethod run360(*s*, *solar_or_view*, *na=36*, *nz=10*, *output_name=None*, *n=None*)

Runs Py6S for lots of angles to produce a polar contour plot.

The calls to 6S for each angle will be run in parallel, making this function far faster than simply running a for loop over all of the angles.

Arguments:

- `s` – A *SixS* instance configured with all of the parameters you want to run the simulation with
- `solar_or_view` – Set to `'solar'` if you want to iterate over the solar zenith/azimuth angles or `'view'` if you want to iterate over the view zenith/azimuth angles
- `output_name` – (Optional) The name of the output from the 6S simulation to plot. This should be a string containing exactly what you would put after `s.outputs` to print the output. For example *pixel_reflectance*.
- `na` – (Optional) The number of azimuth angles to iterate over to generate the data for the plot (defaults to 36, giving data every 10 degrees)
- `nz` – (Optional) The number of zenith angles to iterate over to generate the data for the plot (defaults to 10, giving data every 10 degrees)
- `n` – (Optional) The number of threads to run in parallel. This defaults to the number of CPU cores in your system, and is unlikely to need changing.

For example:

```
s = SixS()
s.ground_reflectance = GroundReflectance.HomogeneousWalthall(0.48, 0.50, 2.95, 0.6)
s.geometry.solar_z = 30
s.geometry.solar_a = 0
data = SixSHelpers.Angles.run360(s, 'view', output_name='pixel_reflectance')
```

classmethod run_and_plot_360(*s*, *solar_or_view*, *output_name*, *show_sun=True*, *na=36*, *nz=10*, *colorbarlabel=None*)

Runs Py6S for lots of angles to produce a polar contour plot.

Arguments:

- `s` – A *SixS* instance configured with all of the parameters you want to run the simulation with
- `solar_or_view` – Set to `'solar'` if you want to iterate over the solar zenith/azimuth angles or `'view'` if you want to iterate over the view zenith/azimuth angles
- `output_name` – The name of the output from SixS to plot. This should be a string containing exactly what you would put after `s.outputs` to print the output. For example *pixel_reflectance*.
- `show_sun` – (Optional) Whether to place a marker showing the location of the sun on the contour plot (defaults to True, has no effect when `solar_or_view` set to `'solar'`.)
- `na` – (Optional) The number of azimuth angles to iterate over to generate the data for the plot (defaults to 36, giving data every 10 degrees)
- `nz` – (Optional) The number of zenith angles to iterate over to generate the data for the plot (defaults to 10, giving data every 10 degrees)
- `colorbarlabel` – (Optional) The label to use on the color bar shown with the plot

For example:

```
s = SixS()
s.ground_reflectance = GroundReflectance.HomogeneousWalthall(0.48, 0.50, 2.95, 0.6)
s.geometry.solar_z = 30
s.geometry.solar_a = 0
SixSHelpers.Angles.run_and_plot_360(s, 'view', 'pixel_reflectance')
```

classmethod run_principal_plane(*s*, *output_name=None*, *n=None*)

Runs the given 6S simulation to get the outputs for the solar principal plane.

This function runs the simulation for all zenith angles in the azimuthal line of the sun. For example, if the solar azimuth is 90 degrees, this function will run simulations for:

```
Azimuth   Zenith
90        85
90        80
90        75
90        70
90        65
90        60
90        55
...       ..
90        0
270       5
270       10
270       15
...       ..
270       80
270       85
```

The calls to 6S for each angle will be run in parallel, making this function far faster than simply running a for loop over each angle.

Arguments:

- `s` – A *SixS* instance configured with all of the parameters you want to run the simulation with
- `output_name` – (Optional) The output name to extract (eg. "pixel_reflectance") if the given data is provided as instances of the Outputs class
- `n` – (Optional) The number of threads to run in parallel. This defaults to the number of CPU cores in your system, and is unlikely to need changing.

Return values:

A tuple containing zenith angles and the corresponding values or Outputs instances (depending on the arguments given). The zenith angles returned have been modified so that the zenith angles on the 'sun-side' are positive, and those on the other side (ie. past the vertical) are negative, for ease of plotting.

Importing atmospheric profiles from radiosonde data

6S is provided with a number of pre-defined atmospheric profiles, such as Midlatitude Summer, Tropical and Subarctic Winter. However, it also possible to parameterise 6S using data acquired from radiosonde (weather balloon) measurements.

The main function in this class (*import_uow_radiosonde_data()*) imports radiosonde data from the University of Wyoming's radiosonde data website to 6S, allowing accurate parameterisation based on real-world measurements.

class `Py6S.SixSHelpers.`**Radiosonde**

classmethod **import_bas_radiosonde_data**(*filename*, *base_profile*)
Imports a radiosonde profile from the British Antarctic Survey radiosonde format.

TODO: More details here after checking with Martin

classmethod **import_uow_radiosonde_data**(*url*, *base_profile*)
Imports radiosonde data from the University of Wyoming website (http://weather.uwyo.edu/upperair/sounding.html) for use in Py6S.

Arguments:

- `url` – The URL of the sounding results page on the UoW website

- `base_profile` – One of the predefined Atmospheric Profiles to use for any parts of the profile which the radiosonde data does not cover (>40km normally)

Return value:

A value suitable for assigning to `s.atmos_profile`, where `s` is a *SixS* instance.

How to use:

1. Go to http://weather.uwyo.edu/upperair/sounding.html and use the interface to select the sounding that you want. Ensure that the From and To date/times are the same, so that only one sounding is retrieved.
2. Copy the URL of the page displaying the sounding. It will look something like http://weather.uwyo.edu/cgi-bin/sounding?region=europe&TYPE=TEXT%3ALIST&YEAR=2012&MONTH=02&
3. Call this function with the URL as the first argument, and one of the predefined atmospheric profiles (eg. `AtmosProfile.MidlatitudeSummer` or `AtmosProfile.Tropical`) as the second argument, and store the result in the atmos_profile attribute of a SixS instance. For example:

```
s.atmos_profile = SixSHelpers.Radiosonde.import_uow_radiosonde_data("http://weather.u
```

The water density, pressure and temperature values from the radiosonde sounding will be interpolated to the 6S atmospheric grid and used for the 6S parameterisation. As radiosonde data tends to end at an altitude of around 30-40km, the data from the selected base profile is used above that height. Ozone data is not imported from the radiosonde data, as most radiosondes do not collect ozone density measurements, so the entire profile is taken from the base profile selected.

Importing aerosol data from AERONET data

6S also has a number of pre-defined aerosol profiles, such as Maritime and Urban, as well as a number of methods for setting aerosol particle distributions based on theoretical distributions. However, the AERONET network (http://aeronet.gsfc.nasa.gov/) stores and processes sun photometer data from many stations around the world which allow the more accurate parameterisation of aerosols in 6S.

The main function in this class (*`import_aeronet_data()`*) imports data from an AERONET CSV file and sets the `aero_profile` and `aot550` parameters of the 6S model accordingly.

class `Py6S.SixSHelpers.`**`Aeronet`**

Contains functions for importing AERONET measurements to set the 6S aerosol profile.

classmethod `import_aeronet_data`(*s*, *filename*, *time*)

Imports data from an AERONET data file to a given SixS object.

This requires a valid AERONET data file and the *pandas* package (see http://pandas.pydata.org/ for installation instructions).

The type of AERONET file required is a *Combined file* for All Points (Level 1.5 or 2.0)

To download a file like this:

1. Go to http://aeronet.gsfc.nasa.gov/cgi-bin/webtool_opera_v2_inv
2. Choose the site you want to get data from
3. Tick the box near the bottom labelled as "Combined file (all products without phase functions)"
4. Choose either Level 1.5 or Level 2.0 data. Level 1.5 data is unscreened, so contains far more data meaning it is more likely for you to find data near your specified time.
5. Choose All Points under Data Format
6. Download the file
7. Unzip

8.Pass the filename to this function

Arguments:

- `s` – A *SixS* instance whose parameters you would like to set with AERONET data
- `filename` – The filename of the AERONET file described above
- `time` – The date and time of the simulation you want to run, used to choose the AERONET data which is closest in time. Provide this as a string in almost any format, and Python will interpret it. For example, `"12/03/2010 15:39"`. When dates are ambiguous, the parsing routine will favour DD/MM/YY rather than MM/DD/YY.

Return value:

The function will return `s` with the `aero_profile` and `aot550` fields filled in from the AERONET data.

Notes:

Beware, this function makes a number of assumptions and performs a number of possibly-inaccurate steps.

1. The refractive indices for aerosols are only provided in AERONET data at a few wavelengths, but 6S requires them at 20 wavelengths. Thus, the refractive indices are extrapolated outside of their original range, to provide the necessary data. This is generally not a wonderful idea, but it is the only way to be able to use the data within 6S. In many cases the refractive indices seem to change very little - but please do check this yourself!

2. The AERONET AOT measurement at the wavelength closest to 550nm (the wavelength required for the AOT specification in 6S) is used. This varies depending on the AERONET site, but may be 50-100nm (or more) away from 550nm. In future versions this code will interpolate the AOT at 550nm using the Angstrom coefficent.

Importing ground reflectance spectra from spectral libraries

These functions allow you to import spectra from two widely-used spectral libraries: the USGS Spectral Library and the ASTER Spectral Library and use the spectra to define the ground reflectance of a 6S model run.

class `Py6S.SixSHelpers.`**Spectra**

Class allowing the import of spectral libraries from various sources

classmethod **import_from_aster**(*loc*)

Imports a spectral library from the ASTER Spectral Library (http://speclib.jpl.nasa.gov/)

Arguments:

- `loc` – Location of the data to import. Can either be a URL (eg. http://speclib.jpl.nasa.gov/speclibdata/jhu.becknic.vegetation.trees.conifers.solid.conifer.spectrum.txt) or a file path.

Returns:

An `ndarray` with two columns: wavelength (um) and reflectance (fraction)

Example usage:

```
from Py6S import *
s = SixS()
s.ground_reflectance = GroundReflectance.HomogeneousLambertian(Spectra.import_from_aster(
s.run()
# Bear in mind this will produce a result for a single Wavelength
# To see what the whole spectrum will look like after atmospheric
# radiative transfer has taken place you must run for multiple wavelengths
# For example
wavelengths, reflectances = SixSHelpers.Wavelengths.run_vnir(s, output_name="apparent_rad
```

classmethod **import_from_usgs**(*loc*)

Imports a spectral library from the USGS Spectral Library (available at http://speclab.cr.usgs.gov/spectral.lib06/).

Arguments:

- `loc` – Location of the data to import. Can either be a URL (eg. http://speclab.cr.usgs.gov/spectral.lib06/ds231/ASCII/V/russianolive.dw92-4.30728.asc) or a file path.

Returns:

An `ndarray` with two columns: wavelength (um) and reflectance (fraction)

Example usage:

```
from Py6S import *
s = SixS()
s.wavelength = Wavelength(0.500)
s.ground_reflectance = GroundReflectance.HomogeneousLambertian(Spectra.import_from_usgs("
s.run()
# Bear in mind this will produce a result for a single Wavelength
# To see what the whole spectrum will look like after atmospheric
# radiative transfer has taken place you must run for multiple wavelengths
# For example
wavelengths, reflectances = SixSHelpers.Wavelengths.run_vnir(s, output_name="apparent_rad
```

Case Study: Assessing the effect of atmospheric changes during the NCAVEO Field Campaign

This page describes an example use of some of the more advanced features of Py6S which should allow you to get a sense of how Py6S can be used as part of real-world scientific research.

Background

The NCAVEO Field Campaign took place in June 2006 and involved the collection of a large number of ground, airborne and spaceborne measurements of the area around Chilbolton, Hampshire, UK with the aim to produce a large dataset of many observations which could be used for model testing and validation (REF HERE). On the main day of data collection (17th June) the atmospheric conditions worsened as the day progressed, thus leading to issues in comparing the data taken at different periods during the day. This issue was a particular problem when attempting to mosaic a number of flightlines obtained from airborne data over a period of around 2 hours. The aim of this example is to show how simulations can easily be performed with Py6S to assess the significance of these atmospheric changes, and the possible influence on uses of this data.

The AERONET site at Chilbolton provides automatic regular sun photometry measurements, with AOT data available approximately every 15 minutes. However, only cloud-screened (level 1.5) data is processed through an inversion algorithm to retrieve more detailed data on the aerosol properties. During the morning of the 17th June, full inverted data is only available at 08:02 and 11:08. However, as these two times are likely to be close to the two extremes of atmospheric conditions, analysis at only these times will still provide a useful test of the significance of the changes.

Code

Py6S code has been written to import this AERONET data, set suitable other parameters, and run simulations for both times with the ground reflectance set to a standard green vegetation spectral reflectance profile. The resulting at sensor radiances and the percentage difference between them are shown below the code.

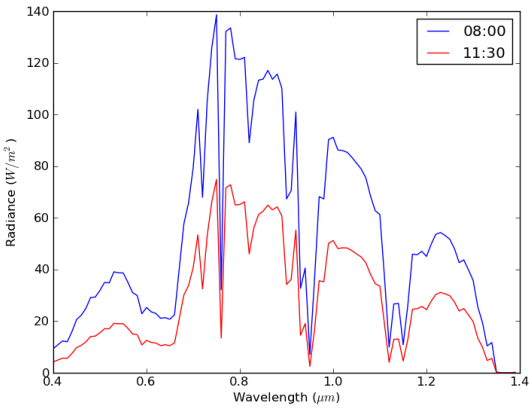
The code is included below and in the `casestudy` folder in the Github repository. The code requires the `CHL` file to exist, this is also available in this folder:

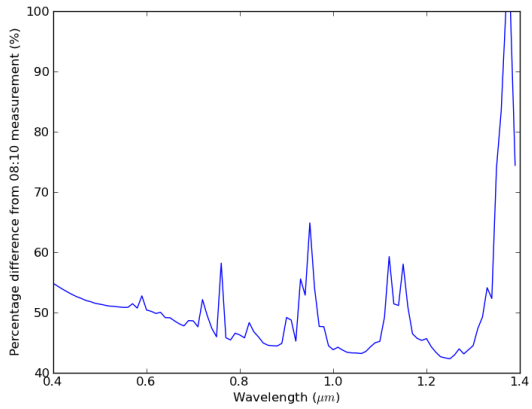
```
# Import Py6S1#
from Py6S import *
# Import the Matplotlib plotting environment
from matplotlib.pyplot import *
# Import the functions for copying objects
import copy

# Define a function to easily calculate NDVI
def ndvi(red, nir):
  return ((nir - red) / (nir + red))

# Create a SixS object for the 'early' time (~08:00)
early = SixS()
# Set the altitudes
early.altitudes.set_target_sea_level()
early.altitudes.set_sensor_satellite_level()
# Set the ground reflectance to be a typical green veg spectrum
early.ground_reflectance = GroundReflectance.HomogeneousLambertian(GroundReflectance.GreenVegetat
early.geometry = Geometry.User()

# Make a copy of the SixS object to use for the 'late' time (~11:30)
late = copy.deepcopy(early)

# Import the AERONET data into each SixS object
SixSHelpers.Aeronet.import_aeronet_data(early, "CHL", "17/06/2006 08:00:00")
SixSHelpers.Aeronet.import_aeronet_data(late, "CHL", "17/06/2006 11:30:00")

# Set the geometry for each SixS object
# With solar angles from location and time and
# with view from nadir
early.geometry.from_time_and_location(51.14510, -1.43861, "17/06/2006 08:00:00", 0, 0)
late.geometry.from_time_and_location(51.14510, -1.43861, "17/06/2006 11:30:00", 0, 0)

# Run each simulation for the VNIR wavelengths - using a wider spacing than default
# to make the simulation faster
wv, early_res = SixSHelpers.Wavelengths.run_vnir(early, spacing=0.01, output_name='pixel_radiance
wv, late_res = SixSHelpers.Wavelengths.run_vnir(late, spacing=0.01, output_name='pixel_radiance')

# Plot the two radiance curves
clf()
plot(wv, early_res, 'b-', label="08:00")
plot(wv, late_res, 'r-', label="11:30")
xlabel("Wavelength ($\mu m$)")
ylabel("Radiance ($W/m^2$)")
legend()
savefig("ncaveo_radiances.png")

# Calculate the percentage difference and plot it
clf()
perc_diff = ((early_res - late_res) / early_res) * 100
plot(wv, perc_diff)
xlabel("Wavelength ($\mu m$)")
ylabel("Percentage difference from 08:10 measurement (%)")
savefig("ncaveo_perc_diff.png")

# Run simulations again for the SPOT HRV sensor
# to then calculate the NDVI difference
wv, early_spot = SixSHelpers.Wavelengths.run_spot_hrv(early, output_name='pixel_radiance')
wv, late_spot = SixSHelpers.Wavelengths.run_spot_hrv(late, output_name='pixel_radiance')

print early_spot
print late_spot
```

```
# Calculate NDVIs
early_ndvi = ndvi(early_spot[1], early_spot[2])
late_ndvi = ndvi(late_spot[1], late_spot[2])

print "Early NDVI:\t%f" % early_ndvi
print "Late NDVI:\t%f" % late_ndvi
print "Percentage Difference:\t%f" % (((early_ndvi - late_ndvi) / early_ndvi) * 100)
```

Text output:

```
Early NDVI:      0.660894
Late NDVI:       0.678239
Percentage Difference:  -2.624595
```





It seems that the change in radiance is high, with an average change of 50%, which suggests that there are likely to be issues with using raw radiance data collected over the course of the morning. However, when assessing the significance of this change it is often helpful to look at standard remote sensing outputs like NDVI, rather than radiance. NDVI calculations show a reduction of 2.62% between 08:00 and 11:30, suggesting that although the deteriorating atmospheric conditions did have a spectrally-varying effect, and thus changed the NDVI values, this change is unlikely to be significant.

Support

Py6S was developed by the author as part of his PhD (which is still in progress).

Support is primarily available through the Py6S mailing list - please join the list and send a message if you want help with Py6S, or have bugs or feature requests to report. The mailing list is also where release announcements will be made and the future direction of Py6S discussed.

Release Notes

Details on the changes in recent versions of Py6S can be found below. More detailed information is available by examining the commit history via Github.

1.7 (31st Jan 2017)

- **Added new spectral response functions**
  - Sentinel 2 Multispectral Instrument (MSI) as *PredefinedWavelengths.S2A_MSI_xx*, and with a *run_s2a_msi* function
  - Sentinel 3 Ocean and Land Colour Instrument (OLCI) as *PredefinedWavelengths.S3A_OLCI_xx*, and with a *run_s3a_olci* function
  - Accurate MODIS bands, based on actual measurements of the MODIS sensor on the Aqua and Terra satellites and taking into account differences between the sensors. Includes the ocean bands. Provided as *PredefinedWavelengths.ACCURATE_MODIS_TERRA_x* and the same for Aqua, with run functions *run_aqua* and *run_terra*.
- Changed radiosonde import to manage missing values properly
- Fixed tab-completion on Python 3
- Added range checking for the altitude parameter to give a sensible error if ground altitudes > 100km or < 0km are given.
- Improved various parts of the documentation

1.6.2 (13th Jan 2016)

- Fixed bug with all SixSHelpers.Wavelengths.run_xxx functions. They now work with extracting outputs like transmittance_total_scattering.total

1.6.1 (9th July 2015)

- Added Pleiades spectral response functions, so simulations can now be run using Pleiades wavelengths

1.6 (14th June 2015)

- Py6S now works properly with Python 3. This was meant to be the case as of v1.5, but various things seemed to go wrong. Thanks to Pete Bunting and Dan Clewley for help fixing this.
- Made setup.py script executable (thanks Matthew Hanson)
- Fixed unusable Homogeneous User Defined BRDF specification (thanks J Gomez-Dans)
- Fixed requirement for pysolar, to deal with upstream changes
- Added many more tests

1.5.4 (16th July 2014)

Fixed minor error on install (didn't affect any functionality)

1.5.3 (16th July 2014)

- Added RapidEye bands to PredefinedWavelengths

1.5.2 (8th July 2014)

- Added extraction of two outputs that had been missed out before: the integrated filter function, and the integrated solar spectrum.

1.5.1 (3rd July 2014)

- Added an option to write_input_file to allow a filename to be given - allowing users to easily export standard 6S input files from Py6S.

1.5.0 (22nd April 2014)

- First release compatible with Python 3. All Py6S functionality should work fine on Python 3 - please contact me if there are any problems.
- Added Landsat 8 spectral response functions, and a run_landsat_oli function.

1.4.2 (20th Feb 2014)

- Fixed bug in the AERONET import routine which meant that ambiguous dates would be imported as MM/DD/YYYY rather than DD/MM/YYYY as specified in the documentation (thanks Marcin)

1.4.1 (22nd Jan 2014)

- Fixed a minor bug which means that running for multiple wavelengths/angles after having already run the SixS object manually would crash

1.4.0 (21st Jan 2014)

- Added parallel processing support for the methods in SixSHelpers that run for multiple wavelengths and multiple angles. This will significantly speed up these runs: on a dual-core machine they should take approximately half the time, and the speedup will be even better on quad-core or eight-core computers. The parallelisation abilities (including the speedup) may be improved in the future, but this should be a significant improvement for now.
- Added produce_debug_report() function to the SixS object. This gives all of the debugging information that I would need when helping to fix a problem - so please run this and send me the output whenever problems occur.

1.3.1 (15th Jan 2014)

- Added proper error handling for dealing with erroneous 6S output, now things shouldn't crash if 6S produces strange results
- Bugfix for error when setting custom altitudes in certain situations
- Added more detailed error messages for failure to import AERONET data
- Bugfix for the specification of geometry parameters within the 6S input file - now more accurate
- Improvements to documentation (typos, clearer explanations etc)
- Added CITATION file to explain how to cite Py6S

1.3 (6th April 2013)

- Fixed a number of bugs relating to geometry specification (thanks Matthew Hanson).
- Significantly improved the code for importing AERONET data - this is now far less likely to go wrong, and more intelligent about what measurements it takes.

1.2.4 (28th Feb 2013)

Bugfix release to fix issue with importing AERONET data from instruments which don't take measurements at 500nm. Importing should now work for any AERONET data, with a warning raised if the instrument doesn't have a band within 70nm of 550nm.

1.2.3 (10th Feb 2013)

Bugfix release to fix issue with importing geometry details from time and location, due to issues with importing PySolar.

1.2.2 (4th Jan 2013)

Bugfix release to fix issue with installation not finding README.rst on some systems.

1.2.1 (3rd Jan 2013)

Bugfix release to fix an issue with the BRDF options in *Py6S.GroundReflectance*, as none of them worked any more due to an issue with the features that were added in v1.2.

1.2 (2nd Jan 2013)

Added ability to import a spectrum from a spectral library (USGS or ASTER spectral libraries are currently supported) and then specify it as the ground reflectance. See *Py6S.SixSHelpers.Spectra* and *Py6S.GroundReflectance*.

This also means that anything that can produce a 2D array with wavelengths (column 0, in micrometres) and reflectances (column 1) can be used to set the ground reflectance. For example, the Python interface to the ProSAIL model (PyProSAIL) can do this, and thus outputs from PyProSAIL can easily be used with 6S (see here for more detailed instructions).

1.1.1 (18th Oct 2012)

Fixed bug which caused Py6S to crash when performing atmospheric correction on Linux (Thanks Vincent!)

1.1 (11th August 2012)

- Updated code for running for multiple wavelengths to make it far easier to maintain
- Fixed bug with user-defined aerosol profile

1.0

This is the first public release of Py6S, which includes all of the functionality detailed in the documentation.

Roadmap

The current release of Py6S has achieved the original aim of producing a Python interface to the 6S model, and it appears to be relatively stable. However, there are many plans for the future, as listed below. If you have any comments on these plans - or want to suggest features that you'd really love to be included in Py6S - then contact me using the details on the Support page.

Soon

- Ability to import spectra from ENVI Spectral Library files
- Ability to import radiosonde data from a wider range of formats
- Add spectral response functions for other modern sensors (eg. Sentinel-2, IKONOS and many others)
- Developments in the API for atmospheric correction, making it easier to atmospherically-correct multiple wavelengths using the same settings

Medium-term

- (Possibly) A restructuring of the Py6S API to make it 'more pythonic' and better suited for future development.
- Ability to use Py6S (probably with a lookup-table) to perform atmospheric correction of satellite images.

Eventually

- A GUI interface allowing easier (but obviously far less flexible) use by those with no Python knowledge (particularly useful for teaching)

Publications

Py6S v1.0 was described in Wilson (2013). You **must** cite this paper if you use Py6S as part of any research which you then publish.

Wilson, R. T., 2013, Py6S: A Python interface to the 6S radiative transfer model, *Computers and Geosciences*, 51, p166-171, PDF

A list of papers citing Py6S is available here.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

U

W