
librat

Prof. P. Lewis

Apr 21, 2020

CONTENTS:

1	RATstart	3
1.1	RATstart (formerly start)	3
1.1.1	Getting started	3
1.1.2	Initialisation shells	5
1.2	Environment variables	7
1.3	Making some basic files	10
1.3.1	bash constructs	10
1.3.2	first.obj and associated files	12
1.3.3	height map visualisation	13
1.3.4	Ray tracing	13
1.3.5	Summary	16
2	RATstart options	17
2.1	Option 1: set sun vectors	18
2.2	Option 2: print sun vectors	18
2.3	Option 6: trace ray from f in direction d	19
2.4	Option 7: query materials	22
2.5	Option 9: print info on materials used	23
2.6	Option 10: get and set verbosity level (0-1)	23
2.7	Option 11: get and print object bbox information	24
2.8	Option 16: produce a height map	24
2.8.1	Summary	26
3	librat	27
3.1	libratlib	27
4	Basic librat / RATstart operation	29
4.1	Object example 1: planes and ellipsoids	29
4.2	Environment variables	31
4.3	Object example 2: clones	35
5	Appendix 1: bash help	37
5.1	Introduction to shell and environment variables	37
5.1.1	export	37
5.1.2	White space and single quotes '	38
5.1.3	echo	38
5.1.4	Double quotes " and backslash escape /	38
5.1.5	env, grep, pipe 	39
5.1.6	Shell variable	39
5.1.7	set, head, tail	40

5.2	Some important environment variables and related	41
5.2.1	PATH	41
5.2.2	which	41
5.2.3	ls	41
5.2.4	.bash_profile, .bashrc, wildcard *	42
5.2.5	ls -l	42
5.2.6	chmod, >, rm -f, mkdir -p	43
5.2.7	cat	44
5.2.8	pwd, cd	44
5.2.9	\$(pwd)	45
5.2.10	\${BPMS-\$(pwd)}	46
5.2.11	edit	46
5.2.12	Update PATH	47
5.2.13	LD_LIBRARY_PATH, DYLD_LIBRARY_PATH	48
5.2.14	Update LD_LIBRARY_PATH, DYLD_LIBRARY_PATH	49
5.2.15	Which operating system? uname, if	50
5.2.16	Contents of libraries: nm or ar	51
5.3	Important environment variables for librat	52
5.3.1	cat <<EOF > output ... EOF	52
5.3.2	MATLIB, RSRLIB etc.	53
5.4	Summary	54
6	Indices and tables	55

UCL Geography/NCEO librat software



RATSTART

This section deals with using the direct interface to the `librat` code at a high level, through the `RATstart` software. This is a short piece of C code, the main role of which is to present a set of simple functions controlled by text input on the command line. One of these ‘options’ (option 14) gives an extremely flexible interface for defining viewing and illumination conditions for radiative transfer simulations, and it is this part of the interface that is most widely used.

In this section, we introduce some of the functionality of `RATstart`. We use a `bash` environment here around `RATstart` to emphasise skills and code in running this from a standard `*nix` command lines. We use a `python3` environment for plotting and related codes.

We first set some local variables.

```
[1]: import sys
      sys.path.insert(0, '.')
      from prelim import *
      %set_env BPMS=$BPMS
      %set_env PATH=$BPMS/bin:$BPMS/src:$BPMS/bin/csh:$PATH_
      %set_env MATLIB=$BPMS/obj
      %set_env BPMSROOT=$BPMS/obj

env: BPMS=/Users/plewis/librat
env: PATH=/Users/plewis/librat/bin:/Users/plewis/librat/src:/Users/plewis/librat/bin/
↪csh:/Users/plewis/opt/anaconda3/bin:/Users/plewis/opt/anaconda3/condabin:/usr/local/
↪bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/VMware Fusion.app/Contents/Public:/
↪Library/TeX/texbin:/opt/X11/bin:/Library/Apple/usr/bin
env: MATLIB=/Users/plewis/librat/obj
env: BPMSROOT=/Users/plewis/librat/obj
```

1.1 RATstart (formerly start)

`RATstart` used to be called `start`, but that causes conflicts sometimes, so its now changed to `RATstart`.

1.1.1 Getting started

The minimum requirement for the `RATstart` command line is the declaration of some object file to use in the simulation. An object file describes the real world geometry and scattering properties of the objects we are using in the simulation, such as some representation of a vegetation canopy. We generate a very simple object file for demonstrating `RATstart`:

```
!{
usemtl WHITE
```

(continues on next page)

(continued from previous page)

```
v 0 0 0
sph -1 100
!}
```

Other than that, its functionality is mainly controlled through a series of option codes read on the input.

Valid options are (currently) between 0 and 16, inclusive, with some missing. You shouldn't use the ones not listed as they tend to be experimental and may not work as expected.

If we give a non-valid option code, we get the help message:

```
[2]: %%bash

# a simple object file
# and invalid code: 1000
# to see the options
# a very simple object file
cat <<EOF > $BPMSROOT/tmp.$$obj
!{
usemtl WHITE
v 0 0 0
sph -1 100
!}
EOF

export LD_LIBRARY_PATH="${BPMS}/src:${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="${BPMS}/src:${DYLD_LIBRARY_PATH}"
RATstart tmp.$$obj <<< 1000;

# tidy up, removing file
rm -f $BPMSROOT/tmp.$$obj

options:
 0                               : quit
 1 n slx sly slz ...             : set sun vectors
 2                               : print sun vectors
 6 fx fy fz dx dy dz            : trace ray from f in direction d
 7                               : get and print materials
 9                               : print info on materials used
10                               : get and set verbosity level (0-1)
11                               : get and print object bbox information
13                               : same as 14 assuming filenames camera.dat light.dat
14 camera.dat light.dat         : ray tracing using defined camera & illumination
16 cx cy cz sx sy nrows ncols rpp name
                               : produce a height map in name
```

As noted above, we most commonly use option 14. This allows the definition of a 'camera' and 'light source' defined in the files `camera.dat` and `light.dat` here, respectively and gives a very general interface for radiative transfer simulations. Using the code then amounts to setting up camera and illumination files for the scenarios you are interested in, which greatly simplifies the interface.

The command line option `-RAThelp` will provide the list of command line options:

```
[3]: %%bash
# a simple object file
# --help to see command line options
# a very simple object file
```

(continues on next page)

(continued from previous page)

```

cat <<EOF > $BPMSROOT/tmp.$$obj
!{
usemtl WHITE
v 0 0 0
sph -1 100
!}
EOF

export LD_LIBRARY_PATH="$BPMS/src:${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="$BPMS/src:${DYLD_LIBRARY_PATH}"
RATstart -RAThelp tmp.$$obj <<< 1000;

# tidy up, removing file
rm -f $BPMSROOT/tmp.$$obj

RATstart: [-RATm 1] [-RATl] [-RATsun_position x y z] [-RATr randomSeed] [-RATblack] [-
→RATblockSize blockSize] [-RATstoreVertex N] [-RATsensor_wavebands wavebands.dat] [-
→RATsun_fov 0 (degrees)] [-RATskymap skymap.hips] [-RATtolerance 0.00001] [-
→RATdirect directIllumination.dat] [-help | -RAThelp] [-RATv 0|1] [-RATn] [-Useless]_
→[-UsetSun x y z] data_filename.obj

```

In the rest of this section, we will learn how to make a call to `RATstart` and how to use the various command line options and input options.

1.1.2 Initialisation shells

Before proceeding, we make sure we set appropriate environment variables to run a set of examples with `RATstart`. We also want to test that the software runs ok. We generate a `bash` shell to achieve this here, and use this in subsequent notes.

You can modify any of this as a user, but be careful not to break it! . In an emergency, you can always just re-download this set of notes and software from [github](#) and start again.

If you want to know more about setting up this sort of file, and what it all means, see [Appendix 1](#). At the moment, you just need to be aware of the need to set environment variables, but not to understand all of the intricacies.

We simplify the interface for the environment variables by setting up the configuration file `examples_init.sh`. We then run it, to test the code.

```

[4]: %%bash
#
# create examples_init.sh
# for examples initialisation
#
# create the init shell
cat <<EOF > $BPMS/bin/examples_init.sh
#!/bin/bash
#
# defaults
#
export BPMS=\${BPMS-$BPMS}
export BPMSROOT=\${BPMSROOT-$BPMSROOT}
lib=\${lib-$BPMS/src}
bin=\${bin-$BPMS/src}
VERBOSE=\${VERBOSE-0}
export TEMP=\${TEMP-/tmp}

```

(continues on next page)

(continued from previous page)

```
# set up required environment variables for bash
export LD_LIBRARY_PATH="\${lib}:\${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="\${lib}:\${DYLD_LIBRARY_PATH}"
export PATH="\${bin}:\${PATH}"

export MATLIB=\$BPMSROOT
export RSRLIB=\$BPMSROOT
export ARARAT_OBJECT=\$BPMSROOT
export DIRECT_ILLUMINATION=\$BPMSROOT
export BPMS_FILES=\$BPMSROOT
export SKY_ILLUMINATION==\$BPMSROOT

if [ "\$(which RATstart)" == "\${bin}/RATstart" ]
then
    if [ "\$VERBOSE" == 1 ]; then
        echo "RATstart found ok"
    fi
else
    # we should create them
    make clean all
fi
EOF
chmod +x $BPMS/bin/examples_init.sh
```

Let us set the VERBOSE flag on, by setting:

```
export VERBOSE=1
```

before running \$INIT. This will cause the shell to print RATstart found ok is the executable RATstart is where it is expected.

```
[5]: %%bash
# run test
export VERBOSE=1
source examples_init.sh

RATstart found ok
```

The shell sets BPMSROOT to be \$BPMS/obj by default. This causes other variables such as MATLIB to be set to the same value, so RATstart will look for materials in \$BPMS/obj (see below):

```
[6]: %%bash

# run test
source examples_init.sh
echo "MATLIB is $MATLIB"

MATLIB is /Users/plewis/librat/obj
```

If we want to use this shell for BPMSROOT elsewhere, such as /home/me/mydata/somewhere, we just need to make sure BPMSROOT is set to this before running \$INIT:

```
[7]: %%bash

# run test
export BPMSROOT=/home/me/mydata/somewhere
```

(continues on next page)

(continued from previous page)

```
source examples_init.sh
echo "MATLIB is $MATLIB"

MATLIB is /home/me/mydata/somewhere
```

1.2 Environment variables

If we take a quick look at the `examples_init.sh` file we generated, we see:

```
[8]: cat $BPMS/bin/examples_init.sh

#!/bin/bash
#
# defaults
#
export BPMS=${BPMS-/Users/plewis/librat}
export BPMSROOT=${BPMSROOT-$BPMSROOT}
lib=${lib-$BPMS/src}
bin=${bin-$BPMS/src}
VERBOSE=${VERBOSE-0}
export TEMP=${TEMP-/tmp}

# set up required environment variables for bash
export LD_LIBRARY_PATH="${lib}:${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="${lib}:${DYLD_LIBRARY_PATH}"
export PATH="${bin}:${PATH}"

export MATLIB=$BPMSROOT
export RSRLIB=$BPMSROOT
export ARARAT_OBJECT=$BPMSROOT
export DIRECT_ILLUMINATION=$BPMSROOT
export BPMS_FILES=$BPMSROOT
export SKY_ILLUMINATION==$BPMSROOT

if [ "$(which RATstart)" == "${bin}/RATstart" ]
then
    if [ "$VERBOSE" == 1 ]; then
        echo "RATstart found ok"
    fi
else
    # we should create them
    make clean all
fi
```

We might notice that `examples_init.sh` sets a number of environment variables, namely `MATLIB`, `RSRLIB` etc.

If you are interested in the details, the meaning of these is given in the table below.

Alternatively, just notice that in the init shell, all of these variables are set to `BPMSROOT`, so if we want to point to the location of an object and material database for librat, we need only set the environment variable `BPMSROOT` appropriately. In this case it defaults to `$BPMS/obj`.

Table explaining librat object environment variables:

Name	File types
MATLIB	material library e.g. <code>plants.matlib</code> , all materials defined in a material library e.g. <code>white.dat</code>
ARARAT_OBJECT	(extended) wavefront object files e.g. <code>first.obj</code>
DIRECT_ILLUMINATION	spectral files for direct illumination: those defined in <code>-RATdirect</code> command line option
RSRLIB	sensor waveband files: those defined in <code>-RATsensor_wavebands</code> command line option
BPMS_FILES	Not used
SKY_ILLUMINATION	location of sky map image files: defined in <code>-RATskymap</code> command line option

You can set all of these to the same value as above (`$BPMSROOT`), in which case the database of files is all defined relative to that location in the file system. This is the most typical use of `librat`. We illustrate this setup below for the `librat` distribution, where a set of examples use files from the directory `test/test_example`.

Additionally in `librat`, the following environment variables can be set to extend the size of some aspects of the model. You would only need to use these in some extreme case.

Table explaining additional librat environment variables:

Name	Purpose
MAX_GROUPS	Maximum number of groups allowed (100000)
PRAT_MAX_MATERIALS	Maximum number of materials allowed (DEFAULT_PRAT_MAX_MATERIALS=1024 in <code>mtllib.h</code>)
MAX_SUNS	Maximum number of suns (180 in <code>rat.h</code>)

If you want to run a shell that uses the setup, you will need to use the command `source` to export the variables to your shell.

You can test the `init` file by running the cell (shell) below.

```
[9]: %%bash

# source if we want the info in this shell
source examples_init.sh

echo "MATLIB is set to $MATLIB"
echo "RSRLIB is set to $RSRLIB"

MATLIB is set to /Users/plewis/librat/obj
RSRLIB is set to /Users/plewis/librat/obj
```

EXERCISE 1

1. Try changing the environment variable `VERBOSE` to 1 (**True**) or 0 (**False**) to see the effect.
2. You can change the name of the directory where the **object** and material files are through the environment variable `BPMSROOT`. See **if** you can find what `BPMSROOT` is set to, **and** also see **if** you can modify it.

Answers below:

```
[10]: %%bash
# Answers to Exercise 1
```

(continues on next page)

(continued from previous page)

```

#-----
# this part same as above
# test the init file
# set INIT script
source $BPMS/bin/local_init.sh
source $INIT
#-----

# 1.1: Try changing the environment variable VERBOSE to 1
# (True) and 0 (False) to see the effect.

# ANSWER
# this sets verbose mode and prints a message
# 'RATstart found ok' if it finds the librat RATstart
# executable
echo "----set VERBOSE 1---"
export VERBOSE=1
source examples_init.sh
# it would be correct, in this instance to use
# export VERBOSE=1
# $INIT
# as there is no real need to source the file
# Above, the export in 'export VERBOSE=1' is not
# strictly needed if we source $INIT, but it is otherwise

# this turns off the verbose mode
# so no message is printed
echo "----set VERBOSE 0---"
export VERBOSE=0
source examples_init.sh
# it would be correct, in this instance to use
# export VERBOSE=0
# $INIT
# as there is no real need to source the file
# Above, the export in 'export VERBOSE=1' is not
# strictly needed if we source $INIT, but it is otherwise

# 1.2: You can change the name of the directory where the
# object and material files are through the
# environment variable BPMSROOT.
# See if you can find what BPMSROOT is set to,
# and also see if you can modify it.

# ANSWER
# we need to see the value of the
# environment variable BPMSROOT, but we need it
# in this shell. So we *source* the file
# rather than running it.
echo "----get BPMSROOT---"
source examples_init.sh
echo "BPMSROOT is $BPMSROOT"
echo "----set BPMSROOT---"
# To change it, just set it before sourcing
export BPMSROOT="/tmp"
source examples_init.sh
echo "BPMSROOT is $BPMSROOT"

```

```
----set VERBOSE 1---
RATstart found ok
----set VERBOSE 0---
----get BPMSROOT---
BPMSROOT is /Users/plewis/librat/obj
----set BPMSROOT---
BPMSROOT is /tmp
```

1.3 Making some basic files

1.3.1 bash constructs

We will create the files we need as we go along, using bash shell `cat <<EOF > filename` syntax for multi-line data input. You may have noticed that we did this above in creating the `init` file.

If we type:

```
cat <<EOF > filename
this is line 1
this is line 2
EOF
```

then a file called `filename` will be generated, containing the information up to the `EOF` marker:

```
this is line 1
this is line 2
```

In a similar fashion, if we type:

```
cat <<<"this is line 1" > filename
```

(note the quotes!) then we can create a single line file with the contents:

```
this is line 1
```

This is equivalent to:

```
echo "this is line 1" | cat > filename
```

```
[11]: %%bash
cat <<<"hello world"
hello world
```

```
[12]: %%bash
cat << EOF
hello world
EOF
hello world
```

EXERCISE 2

1. Use these shell constructs **as** appropriate to create a file called green.dat,
 ↳ **with** 2 columns of data, the first being wavelength **in** nm, the second being
 ↳ reflectance, representing a typical green spectrum, **for** samples at 450 nm, 550 nm
 ↳ **and** 650 nm.
2. Use these shell constructs **as** appropriate to create a file called white.dat,
 ↳ **with** 2 columns of data, the first being wavelength **in** nm, the second being
 ↳ reflectance, representing a white spectrum, **for** samples at 450 nm, 550 nm **and** 650
 ↳ nm.

Answers below:

```
[13]: %%bash

# Answers to Exercise 2

# 2.1 green: high at 550 nm, low at 450 and 650 nm
# Its a multi-line file that we need, so
cat << EOF > $BPMS/obj/green_.dat
450 0.1
550 0.9
650 0.1
EOF

# 2.2 white: high at all
# Its a multi-line file that we need, so
cat << EOF > $BPMS/obj/white_.dat
450 0.1
550 0.9
650 0.1
EOF

# show results:
echo "====="
echo "green"
echo "====="
cat $BPMS/obj/green_.dat

echo "====="
echo "white"
echo "====="
cat $BPMS/obj/white_.dat

# be polite and tidy up
rm -f $BPMS/obj/green_.dat $BPMS/obj/white_.dat

=====
green
=====
450 0.1
550 0.9
650 0.1
=====
white
=====
450 0.1
```

(continues on next page)

(continued from previous page)

```
550 0.9
650 0.1
```

1.3.2 first.obj and associated files

Now we will use these constructs to put some text some files in that directory. We will be creating files that we need to run a radiative transfer simulation. We will look into what these mean and their formats later in the notes.

```
[14]: %%bash

source examples_init.sh
# simple object file
# with green plane
# and sphere of radius 100 mm
# centred at (0,0,0)
cat <<EOF > $BPMSROOT/first.obj
# My first object file
mtllib plants.matlib.new
usemtl green
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 0
sph -1 1
!}
!}
EOF

# wavelengths (nm)
cat <<EOF > $BPMSROOT/wavebands.dat
1 650
2 550
3 450
EOF

# spectrum for white
cat <<EOF > $BPMSROOT/white.dat
450 1
550 1
650 1
EOF

# spectrum for green
cat <<EOF > $BPMSROOT/green.dat
450 0.1
550 0.5
650 0.1
EOF

# library listing materials
cat <<EOF > $BPMSROOT/plants.matlib.new
srm green green.dat
```

(continues on next page)

(continued from previous page)

```
srm white white.dat
EOF
```

The scene object described in `first.obj` is a sphere of radius 1, centred at location $(0, 0, 0)$, as well as a plane with normal $(0, 0, 1)$ passing through location $(0, 0, 0)$. The sphere is `white` and the plane is `green`.

1.3.3 height map visualisation

In the first example, we produce a visualisation of the object using option 16.

```
16 cx cy cz sx sy nrows ncols rpp name : produce a height map in name
```

Option 16 produces an image dataset (in a rather old ‘hips’ format) that is a height map of the scene, produced by ray casting. Imagine a rectangle on an x-y plane of physical dimensions (sx, sy) , centred at (cx, cy, cz) .

```
[15]: %%bash
source examples_init.sh
# option 8
RATstart first.obj << EOF
16 0 0 4 4 4 200 200 1 $BPMS/obj/out.tmp.hips
EOF

( 99.9975)
```

We can use the python library `libhipl` from `RATlibUtils` to read (and write) hips format images:

```
[16]: from RATlibUtils.libhipl import Hipl
import pylab as plt

f = BPMS+'/obj/out.tmp.hips'
plt.imshow(Hipl().read(f), cmap='gray')
plt.colorbar()

[16]: <matplotlib.colorbar.Colorbar at 0x118bd3790>
```

In the visualisation we can see the sphere and plane as a height map. The height varies from 0 (the plane) to 1 (at the top of the sphere). The extent of the scene is 4×4 units, so we can confirm that the radius of the sphere is 1 unit by inspection.

1.3.4 Ray tracing

And now another example.

We run this with the verbose flag set on running `RATstart`.

The input:

```
1 3 0 1 1 1 1 1 1 0 1
```

(option 1) sets the number of suns (3) followed by the sun direction vectors $(0, 1, 1)$ $(1, 1, 1)$ and $(1, 0, 1)$ here. In practice, we tend to run simulations with a single illumination (sun) vector, but we illustrate using 3 here to encourage the use of this option.

The input:

```
6 0 0 10 0 0 -1
```

(option 6) sets up a ray tracing case, for a ray *from* location (0 0 10), in direction (0 0 -1) (straight down).

We set the `-RATtolerance` command line option to a small number, relative to scene size objects: 0.0000001 is sufficient in this case.

We should be able to visualise this situation, and see that the ray will hit the sphere at (0 0 1), so be of length 9 units.

```
[17]: %%bash
source examples_init.sh

RATstart -RATtolerance 0.0000001 -RATsensor_wavebands wavebands.dat first.obj <<EOF
1 3 0 1 1 1 1 1 1 0 1
6 0 0 10 0 0 -1
EOF
```

```
RTD 0
order: 0      intersection point:      0.000000 0.000000 1.000000
              ray length:              9.000000
              intersection material:    3
              sun 0:                    1 reflectance
              direct:                    0.707107 0.707107 0.707107
              sun 1:                    1 reflectance
              direct:                    0.577350 0.577350 0.577350
              sun 2:                    1 reflectance
              direct:                    0.707107 0.707107 0.707107
              sky   :                    reflectance
              diffuse:                   1.000000 1.000000 1.000000
```

```
how many sun vectors? (>0): enter sun vector number 1 (3 floats): enter sun vector_
↪number 2 (3 floats): enter sun vector number 3 (3 floats):
```

In the example above, we hit the sphere at its apex. The normal vector of the sphere at that point is (0,0,1) and the material reflectance (white) is 1.0 at all wavelengths. So we might notice that the `direct` reflectance above is always the cosine of the angle between the normal vector (0,0,1) and the solar vector.

We explore this in the code below by calculating the `vector dot product` of the sun and normal unit vectors. Note that the unit vectors have unit (1) length, so if we define them as vectors of arbitrary length (e.g. (1, 1, 1)) we need to normalise them (divide by the length, to give the unit length):

```
[18]: import numpy as np
from numpy import linalg as LA

#### dot product for angle ####

sun = np.array([1.,1,1.])
# normalise vector
length = LA.norm(sun)
sunHat = sun/length

normal = np.array([0,0,1.])
# normalise vector
length = LA.norm(normal)
normalHat = normal/length
```

(continues on next page)

(continued from previous page)

```
cosine = np.dot(sunHat,normalHat)
print("cosine of",sun,"=",cosine)

cosine of [1. 1. 1.] = 0.5773502691896258
```

EXERCISE 3

Adapt the RATstart example above to have just a single sun vector.

Change the sun vector **in** the RATstart example, **and** use the python code above to **predict** what the direct reflectance will be. Do this **for** several examples.

Answers below:

```
[19]: %%bash
#
# set a single sun vector, here (0,0,1)
# so dot product = 1.0 and direct refl = 1
#
source examples_init.sh

RATstart -RATtolerance 0.0000001 -RATsensor_wavebands wavebands.dat first.obj <<EOF
1 1 0 0 1
6 0 0 10 0 0 -1
EOF

RTD 0
order: 0      intersection point:      0.000000 0.000000 1.000000
              ray length:              9.000000
              intersection material:    3
              sun 0:                    1 reflectance
              direct:                    1.000000 1.000000 1.000000
              sky   :                    reflectance
              diffuse:                    1.000000 1.000000 1.000000

how many sun vectors? (>0): enter sun vector number 1 (3 floats):
```

```
[20]: import numpy as np
from numpy import linalg as LA

#### dot product for angle ####

sun = np.array([0,0,1.])
# normalise vector
length = LA.norm(sun)
sunHat = sun/length

normal = np.array([0,0,1.])
# normalise vector
length = LA.norm(normal)
normalHat = normal/length

cosine = np.dot(sunHat,normalHat)
print("cosine of",sun,"=",cosine)

cosine of [0. 0. 1.] = 1.0
```

```
[21]: %%bash
#
# set a single sun vector, here (0,1,1)
# so dot product = 0.707 (sqrt(2))
# and so direct refl = 0.707
#
source examples_init.sh

RATstart -RATtolerance 0.0000001 -RATsensor_wavebands wavebands.dat first.obj <<EOF
1 1 0 1 1
6 0 0 10 0 0 -1
EOF
```

```
RTD 0
order: 0          intersection point:      0.000000 0.000000 1.000000
                  ray length:              9.000000
                  intersection material:    3
                  sun 0:                   1 reflectance
                  direct:                  0.707107 0.707107 0.707107
                  sky   :                  reflectance
                  diffuse:                  1.000000 1.000000 1.000000
```

how many sun vectors? (>0): enter sun vector number 1 (3 floats):

```
[22]: import numpy as np
      from numpy import linalg as LA

      ##### dot product for angle #####

      sun = np.array([0,1,1.])
      # normalise vector
      length = LA.norm(sun)
      sunHat = sun/length

      normal = np.array([0,0,1.])
      # normalise vector
      length = LA.norm(normal)
      normalHat = normal/length

      cosine = np.dot(sunHat,normalHat)
      print("cosine of",sun,"=",cosine)

      cosine of [0. 1. 1.] = 0.7071067811865475
```

1.3.5 Summary

In this section, we have introduced the RATstart code, some information about the requirements for running the code. We generated some initialisation scripts to simplify setting up environment variables, generated a simple world object (sphere on a plane) and visualised this.

RATSTART OPTIONS

This section explores the input options for `librat`.

Recall that we can access them in. e.g.:

```
[2]: import sys
      sys.path.insert(0, '.')
      from prelim import *
      %set_env BPMS=$BPMS
      %set_env PATH=$BPMS/bin:$BPMS/src:$BPMS/bin/csh:$PATH_
      %set_env MATLIB=$BPMS/obj
      %set_env BPMSROOT=$BPMS/obj

env: BPMS=/Users/plewis/librat
env: PATH=/Users/plewis/librat/bin:/Users/plewis/librat/src:/Users/plewis/librat/bin/
↪csh:/Users/plewis/opt/anaconda3/bin:/Users/plewis/opt/anaconda3/condabin:/usr/local/
↪bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/VMware Fusion.app/Contents/Public:/
↪Library/TeX/texbin:/opt/X11/bin:/Library/Apple/usr/bin
env: MATLIB=/Users/plewis/librat/obj
env: BPMSROOT=/Users/plewis/librat/obj
```

```
[3]: %%bash

source examples_init.sh
RATstart first.obj <<< 1000

options:
  0                               : quit
  1 n slx sly slz ... : set sun vectors
  2                               : print sun vectors
  6 fx fy fz dx dy dz : trace ray from f in direction d
  7                               : get and print materials
  9                               : print info on materials used
  10                            : get and set verbosity level (0-1)
  11                            : get and print object bbox information
  13                            : same as 14 assuming filenames camera.dat light.dat
  14 camera.dat light.dat
                               : ray tracing using defined camera & illumination
  16 cx cy cz sx sy nrows ncols rpp name
                               : produce a height map in name
```

2.1 Option 1: set sun vectors

```
1 n slx slz slz ... : set sun vectors
```

We will set a single sun vector, in direction $(0, 0, 0)$, so the command is:

```
1 1 0 0 1
```

```
[4]: %%bash
source examples_init.sh
# option 0
RATstart first.obj << EOF
1 1 0 0 1
EOF
```

```
how many sun vectors? (>0): enter sun vector number 1 (3 floats):
```

2.2 Option 2: print sun vectors

```
2 : print sun vectors
```

Now we will describe 2 suns (2 direct illumination sources), the first in direction $(0, 0, 1)$ and the second in direction $(0, 1, 1)$. The input for that is:

```
1 2 0 0 1 0 1 1
```

We then print the sun vectors, noticing that the outputs are normalised (unit direction vectors).

```
[5]: %%bash
source examples_init.sh
# option 0
RATstart first.obj << EOF
1 2 0 0 1 0 1 1
2
EOF
```

```
Sun vector 1: 0.000000 0.000000 1.000000
Sun vector 2: 0.000000 0.707107 0.707107
```

```
how many sun vectors? (>0): enter sun vector number 1 (3 floats): enter sun vector_
↩number 2 (3 floats):
```

We could use option 2 to check what the default sun vector is:

```
[6]: %%bash
source examples_init.sh
# option 0
RATstart first.obj << EOF
2
EOF
```

```
Sun vector 1: 0.000000 0.000000 1.000000
```

2.3 Option 6: trace ray from f in direction d

```
6 fx fy fz dx dy dz : trace ray from f in direction d
```

Option 6 is a useful generic interface that allows us to probe intersections with the world object by sending a sample ray from some given location (fx , fy , fz), in a particular direction (dx , dy , dz). We return information on what the ray interacts with, tracing sample direct and diffuse rays, to a maximum ray tree depth set by the command line option `-RATm N`. Normally, we should set N to a high number (e.g. 1000 or higher). The command line option `-RATtolerance 0.0000000001` sets a tolerance for ray intersections, and should be appropriate to the scale of the world scene. It should be a small number, relative to typical object sizes.

In this example, we fire a ray from $(0, 0, 50)$ in the direction $(0, 0, -1)$ (straight down) into the scene `HET01_DIS_ERE.obj` (one of the RAMI test scenes).

```
[7]: %%bash
source examples_init.sh

# option 6
# store the result in the file $BPMSROOT/tmp.dat
RATstart -RATr 1 \
    -RATsensor_wavebands wavebands.dat \
    -RATtolerance 0.0000000001 \
    -RATm 1000 \
    HET01_DIS_ERE.obj << EOF > $BPMSROOT/RATstart.out.dat
11
6 0 0 50 0 0 -1
EOF

# filter some information out of the object file
# for visualisation
# 1. $BPMSROOT/tmp.obj.dat
# locations of clone objects
awk '($1=="clone"){print $2,$3,$4}' \
    < $BPMSROOT/HET01_DIS_ERE.obj \
    > $BPMSROOT/RATstart.obj.dat

# 2. $BPMSROOT/tmp.int.dat
# filter ray intersections from file
# $BPMSROOT/tmp.dat
# Add the simulation interactions
# add Nan to indicate new primary ray
awk '($4 == "point:"){print $5,$6,$7} \
    ($1 == "RTD"){print "Nan Nan Nan"}' \
    < $BPMSROOT/RATstart.out.dat \
    > $BPMSROOT/RATstart.ray.dat

# output the file
cat $BPMSROOT/RATstart.out.dat

x: -51.199330 53.623519
y: -43.571414 51.254278
z: 1.493505 27.609156
bbox centre @ 1.212094 3.841432 14.551330
RTD 5
order: 0          intersection point:      -0.000000 -0.000000 15.236980
                ray length:                34.763020
                intersection material:      4
```

(continues on next page)

(continued from previous page)

order: 1	sun 0:	1 reflectance
	direct:	0.030690 0.044679 0.035031
	intersection point:	-1.512808 6.032760 0.000000
	ray length:	16.457470
	intersection material:	5
order: 2	sun 0:	1 reflectance
	direct:	0.003918 0.016226 0.000350
	intersection point:	3.805090 18.256928 16.902007
	ray length:	21.526453
	intersection material:	4
order: 3	sun 0:	1 transmittance
	direct:	0.000057 0.001247 0.000001
	intersection point:	2.156740 18.892552 14.457133
	ray length:	3.016370
	intersection material:	4
order: 4	sun 0:	1 reflectance
	direct:	0.000012 0.000106 0.000001
	intersection point:	-0.565249 22.253311 11.564100
	ray length:	5.203226
	intersection material:	4
order: 5	sun 0:	1 reflectance
	direct:	0.000000 0.000013 0.000000
	intersection point:	-26.101692 51.431591 0.000000
	ray length:	40.462457
	intersection material:	5
	sun 0:	1 reflectance
	direct:	0.000000 0.000007 0.000000
	sky :	reflectance
	diffuse:	0.000000 0.000007 0.000000

We see a number of orders of interaction orders in the ray tracing. We can visualise this:

```
[8]: %matplotlib inline
from RATlibUtils.plotters import plotter6
# these are the data files
# interactions
file=BPMS+"/obj/RATstart.ray.dat"
# object locations
file1=BPMS+"/obj/RATstart.obj.dat"
plotter6(file,file1)
```

```
[9]: %%bash
source examples_init.sh

# tidy up the files we created
rm -f $BPMSROOT/RATstart.obj.dat \
    $BPMSROOT/RATstart.out.dat \
    $BPMSROOT/RATstart.ray.dat
```

We can relate the path of the ray in the x-y plane (the red line, starting at the red circle) to the information on ray intersections above.

For example, looking at intersection material we see that most intersections are with material 4 – this is the Leaf material, within the spheres, and others are material 3, the soil material. Looking at the figure above, we can see that the first intersection (at -0.000000 -0.000000 15.236980) is on a sphere, so material 4. The next interaction is with the ground (so material 3). Then the ray path enters another sphere, bounces around within that for

several interactions, then escapes, passes through the space of another sphere without interaction, then hits the ground and finally escapes.

This sort of visualisation is useful in learning how the ray tracing works.

EXERCISE

1. Use the bash script above to generate paths for 5 (or more!) primary rays.
2. Visualise the paths, using the python script above (no change needed).
3. Vary the integer after the -RATr flag to use a different random number seed. What
 → is the effect?

Answers below

```
[10]: %%bash
source examples_init.sh

# option 6
# store the result in the file $BPMSROOT/tmp.dat
# just repeat the primary ray call to fire a new ray
RATstart -RATr 100 \
    -RATsensor_wavebands wavebands.dat \
    -RATtolerance 0.00000000001 \
    -RATm 1000 \
    HET01_DIS_ERE.obj << EOF > $BPMSROOT/RATstart.out.dat
6 0 0 50 0 0 -1
6 0 0 50 0 0 -1
6 0 0 50 0 0 -1
6 0 0 50 0 0 -1
6 0 0 50 0 0 -1
EOF

# filter some information out of the object file
# for visualisation
# 1. $BPMSROOT/tmp.obj.dat
# locations of clone objects
awk < $BPMSROOT/HET01_DIS_ERE.obj \
    '($1=="clone"){print $2,$3,$4}' > $BPMSROOT/RATstart.obj.dat
# 2. $BPMSROOT/tmp.int.dat
# filter ray intersections from file
# $BPMSROOT/tmp.dat
# Add the simulation interactions
# encode Nan to separate the primary rays
awk '($4 == "point:"){print $5,$6,$7} \
    ($1 == "RTD"){print "Nan Nan Nan"}' \
    < $BPMSROOT/RATstart.out.dat \
    > $BPMSROOT/RATstart.ray.dat
```

```
[11]: %matplotlib inline
from RATlibUtils.plotters import plotter6
# these are the data files
# interactions
file=BPMS+"/obj/RATstart.ray.dat"
# object locations
file1=BPMS+"/obj/RATstart.obj.dat"
plotter6(file,file1)
```

```
[12]: %%bash
# tidy up the files we created

source examples_init.sh
BPMSROOT=$BPMS/obj
rm -f $BPMSROOT/RATstart.obj.dat \
    $BPMSROOT/RATstart.out.dat \
    $BPMSROOT/RATstart.ray.dat
```

2.4 Option 7: query materials

```
7         : get and print materials
```

```
[13]: %%bash
source examples_init.sh

# Query materials
# option 7
RATstart $BPMSROOT/first.obj <<< 7

There are 4 materials:

***** Material 0 *****
      name: WHITE          is used? 0
      type: (0) = srm with reflectance
      Number in used material list = -1
      nBands: 2:
0.000000 1.000000
1000000000000000000000.000000 1.000000
***** Material 1 *****
      name: TRANSPARENT      is used? 0
      type: (10) = transparent
      Number in used material list = -1
***** Material 2 *****
      name: green            is used? 1
      type: (0) = srm with reflectance
      Number in used material list = 0
      nBands: 3:
450.000000 1.000000
550.000000 1.000000
650.000000 1.000000
***** Material 3 *****
      name: white            is used? 1
      type: (0) = srm with reflectance
      Number in used material list = 1
      nBands: 3:
450.000000 1.000000
550.000000 1.000000
650.000000 1.000000
```

2.5 Option 9: print info on materials used

```
9 : print info on materials used
```

```
[14]: %%bash
source examples_init.sh
# option 9
RATstart $BPMSROOT/first.obj <<< 9

There are 4 materials defined
There are 2 materials used:

***** Material 0 *****
      name: green      index: 2
      type: (0) = srm with reflectance
      nBands: 3:
450.000000 1.000000
550.000000 1.000000
650.000000 1.000000
***** Material 1 *****
      name: white      index: 3
      type: (0) = srm with reflectance
      nBands: 3:
450.000000 1.000000
550.000000 1.000000
650.000000 1.000000
```

2.6 Option 10: get and set verbosity level (0-1)

```
10 : get and set verbosity level (0-1)
```

```
[ ]: %%bash
source examples_init.sh

# interactions to set verbosity to 1
RATstart first.obj << EOF
10
Y
1
EOF
```

```
[ ]: %%bash
source examples_init.sh

# interactions to set verbosity to 0
RATstart first.obj << EOF
10
Y
0
EOF
```

But normally, we would just use the command line to set this:

```
[ ]: %%bash
source examples_init.sh

# interactions to set verbosity to 1
RATstart -RATv 1 first.obj << EOF
0
EOF
```

```
[ ]: %%bash
source examples_init.sh

# interactions to set verbosity to 0
RATstart -RATv 0 first.obj << EOF
0
EOF
```

2.7 Option 11: get and print object bbox information

11	: get and print object bbox information
----	--

```
[17]: %%bash
source examples_init.sh

# interactions to set verbosity to 1
RATstart first.obj <<< 11

x: -0.999800 1.000200
y: -0.999800 1.000200
z: -0.999800 1.000200
bbox centre @ 0.000200 0.000200 0.000200
```

2.8 Option 16: produce a height map

16	cx cy cz sx sy nrows ncols rpp name : produce a height map in name
----	---

Recall that `first.obj` is a plane at $z=0$ and a sphere of radius 1 centred at $(0, 0, 0)$. Option 16 produces an image dataset (in a rather old ‘hips’ format) that is a height map of the scene, produced by ray casting. Imagine a rectangle on an x - y plane of physical dimensions (sx, sy) , centred at (cx, cy, cz) . This option samples such a plane, with $(nrows, ncols)$ dimensions, with rpp samples per output pixel. The result is written to the image defined by name.

A simple use of this option, illustrated below, is to give a scene height map. In this case, we want (cx, cy, cz) to be above the z -dimension of the scene bounding box. Lets first re-familise ourselves with the object file, and look at its bounding box (option 11):

```
[18]: %%bash
source examples_init.sh

# lets look at it
cat $BPMS/obj/first.obj
echo "11" | RATstart first.obj
```

```
# My first object file
mtllib plants.matlib.new
usemtl green
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 0
sph -1 1
!}
!}
x: -0.999800 1.000200
y: -0.999800 1.000200
z: -0.999800 1.000200
bbox centre @ 0.000200 0.000200 0.000200
```

The maximum z is given as 1.000200 (some tolerance is added in calculating the bounding box extent, as really it is 1.000000). We therefore centre our simulation at $(0, 0, 2)$. We set the scene x and y dimensions to 4 units, and produce an image of 200 by 200 pixels with 1 ray per pixel.

```
[31]: %%bash
source examples_init.sh

# output image name
op=$BPMS/obj/out1.hips

echo "16 0 0 2 4 4 200 200 1 $op" | RATstart \
-RATsensor_wavebands wavebands.dat \
-RATv first.obj

RATstart:
  VERBOSE flag on (-v option)
read_spectral_file: 3 data entries read in file /Users/plewis/librat/obj/wavebands.
→dat
( 99.9975)
```

We use the python library `libhip1` (locally installed) to read the image, then display it using `matplotlib`:

```
[30]: from RATlibUtils.libhip1 import Hip1
import pylab as plt

f = BPMS+'/obj/out1.hips'
rabbit=Hip1().read(f)
plt.imshow(rabbit,cmap='gray')
plt.colorbar()

[30]: <matplotlib.colorbar.Colorbar at 0x10fe25790>
```

```
[26]: %%bash
source examples_init.sh

# output image name
op=$BPMS/obj/out2.hips

RATstart HET01_DIS_ERE.obj <<EOF
```

(continues on next page)

(continued from previous page)

```
11
16 0 0 30 200 200 200 200 1 $op
EOF

x: -51.199330 53.623519
y: -43.571414 51.254278
z: 1.493505 27.609156
bbox centre @ 1.212094 3.841432 14.551330

( 99.9975)
```

```
[29]: from RATlibUtils.libhipl import Hipl
import pylab as plt

f = BPMS+'/obj/out2.hips'
rabbit=Hipl().read(f)
plt.imshow(rabbit,cmap='gray')
plt.colorbar()

[29]: <matplotlib.colorbar.Colorbar at 0x1219bab90>
```

2.8.1 Summary

In this section, we have explored the input options for `librat`. These consist of a numeric code on on command line input, followed by some (possibly interactive) set of parameters. We have explored all of those listed, other than 13 and 14 which refer to camera and light files.

3.1 `libratlib`

Its a shared object library ... explain.

[]:

[]:

BASIC LIBRAT / RATSTART OPERATION

Librat is the library of function calls around which you can write your own code to do things such as read in and parse an object file, read in and parse camera, illumination files, waveband files and so on. However, RATstart (formerly start) is a wrapper code around these commands which gives you access to all the basic operations, and so is the de facto tool for doing simulations. The key things required to carry out a simulation are:

- A camera file
- An illumination file
- A waveband file
- An object file - this is always assumed to be the last file on the RATstart command line

Anything specific you want to do in any of these parts of the process is specified in these files. There are a limited number of additional command line options which either allow you to override a few key things in these files (the waveband file for example), or more usually are external to these things. Each of these can be passed through via the `-RAT` keyword. Examples are the ray tree depth (`-RATm`), verbose level (`-RATv`), waveband file (`-RATsensor_wavebands`) etc.

4.1 Object example 1: planes and ellipsoids

```
[1]: %%bash
mkdir -p test/test_examples
```

Now, a simple scene object `test/test_examples/first.obj` `<test/test_examples/first.obj>`__`

```
[2]: %%bash

cat <<EOF > test/test_examples/first.obj
# My first object file
mtllib plants.matlib
usemtl white
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 1000
ell -1 30000 30000 1000
!}
```

(continues on next page)

(continued from previous page)

```
!}  
EOF
```

This object uses a material library ``plants.matlib <test/test_examples/plants.matlib>`__` that specifies the reflectance and transmittance properties of the scene materials.

```
[3]: %%bash  
  
cat <<EOF > test/test_examples/plants.matlib  
srm white refl/white.dat  
EOF
```

In this example, the file contains the single line:

```
srm white refl/white.dat
```

so there is only a single material of type `srm` (standard reflectance material - Lambertian reflectance (and/or transmittance)). The material name is `white` and the (ASCII) file giving the spectral reflectance function is ``refl/white.dat <test/test_examples/refl/white.dat>`__`.

```
[4]: %%bash  
  
mkdir -p test/test_examples/refl  
  
cat <<EOF > test/test_examples/refl/white.dat  
0 1  
10000 1  
EOF
```

The file ``refl/white.dat <test/test_examples/refl/white.dat>`__` contains 2 columns: column 1 is ‘wavelength’ (really, a pseudo-wavelength in this case), column 2 is reflectance for that wavelength (wavelength units are arbitrary, but we usually use nm).

In this case, the file specifies:

```
0 1  
10000 1
```

which is a reflectance of 1.0 for any wavelength (less than or equal to an arbitrary upper limit 10000). If the file specifies transmittance as well, this is given as a third column.

Looking back to ``test/test_examples/first.obj <test/test_examples/first.obj>`__`, the line:

```
mtllib plants.matlib
```

tells the librat reader to load the ‘material library’ called ``plants.matlib <test/test_examples/plants.matlib>`__`. First, it will look in the current directory for the file. If it doesn’t find it there, it will see if the environment variable `MATLIB` is set. If so, it will look there next.

4.2 Environment variables

The following environmental variables can be used:

Name	File types
MATLIB	material library e.g. <code>plants.matlib</code> , all materials defined in a material library e.g. <code>white.dat</code>
ARARAT_OBJECT	(extended) wavefront object files e.g. <code>first.obj</code>
DIRECT_ILLUMINATION	Spectral files for direct illumination: those defined in <code>-RATdirect</code> command line option
RSRLIB	sensor waveband files: those defined in <code>-RATsensor_wavebands</code> command line option
BPMS_FILES	Not used
SKY_ILLUMINATION	location of sky map image files: defined in <code>-RATskymap</code> command line option

You can set all of these to the same value, in which case the database of files is all defined relative to that point. This is the most typical use of `librat`. We illustrate this setup below for the `librat` distribution, where a set of examples use files from the directory `test/test_example`.

Additionally, the following environment variables can be set to extend the size of some aspects of the model. You would only need to use these in some extreme case.

Name	Purpose
MAX_GROUPS	Maximum number of groups allowed (100000)
PRAT_MAX_MATERIALS	Maximum number of materials allowed (DEFAULT_PRAT_MAX_MATERIALS=1024 in <code>mtllib.h</code>)
MAX_SUNS	Maximum number of suns (180 in <code>rat.h</code>)
MAX_OBJECT_TYPES	Maximum number of types of object used (sph, f, etc): hardwired in <code>intersect_objects.h</code> at 16. Only used in <code>RATstart</code> option 8

In this case, we would want to set `MATLIB` to `test/test_examples` before invoking `librat`. In bash for example, this is done with:

```
[5]: %%bash
export MATLIB=test/test_examples
```

Let's put all of these into a shell called ``init.sh <test/test_examples/init.sh>`_``:

```
[6]: %%bash

# create the init.sh file we want
outfile=test/test_examples/init.sh

cat <<EOF > $outfile
#!/bin/bash
#
# preamble
#
BPMS=${BPMS-$(pwd)}
# set shell variables lib, bin, verbose
# with defaults in case not set
lib=${lib-"$BPMS/src"}
bin=${bin-"$BPMS/src"}
```

(continues on next page)

(continued from previous page)

```

VERBOSE=\${VERBOSE-1}

# set up required environment variables for bash
export LD_LIBRARY_PATH="\${lib}:\${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="\${lib}:\${DYLD_LIBRARY_PATH}"
export PATH="\${bin}:\${PATH}"

# set up required environment variables for librat
export TEST=\${BPMS}/test/test_example
export MATLIB=\$TEST
export RSRLIB=\$TEST
export ARARAT_OBJECT=\$TEST
export DIRECT_ILLUMINATION=\$TEST
export BPMS_FILES=\$TEST

if [ "\$(which RATstart)" == "\${bin}/RATstart" ]
then
    if [ "\$VERBOSE" ]; then
        echo "RATstart found ok"
    fi
else
    # we should create them
    make clean all
fi
EOF

# set executable mode
chmod +x \$outfile
# test run
\$outfile

```

```
make: *** No rule to make target `clean'. Stop.
```

```

-----
CalledProcessError                                Traceback (most recent call last)
<ipython-input-6-d988add8e183> in <module>
----> 1 get_ipython().run_cell_magic('bash', '', '\n# create the init.sh file we want\
↳noutfile=test/test_examples/init.sh\nncat <<EOF > $outfile\n#!/bin/bash\n#\n#_
↳preamble \n#\nBPMS=\${BPMS-\$(pwd)}\n# set shell variables lib, bin, verbose\n#_
↳with defaults in case not set \nlib=\${lib-"${BPMS}/src"}\nbin=\${bin-"${BPMS}/src_
↳")\nVERBOSE=\${VERBOSE-1}\n#\n# set up required environment variables for bash\
↳nexport LD_LIBRARY_PATH="\${lib}:\${LD_LIBRARY_PATH}"\nexport DYLD_LIBRARY_PATH="\
↳\${lib}:\${DYLD_LIBRARY_PATH}"\nexport PATH="\${bin}:\${PATH}"\n#\n# set up_
↳required environment variables for librat\nexport TEST=\${BPMS}/test/test_example\
↳nexport MATLIB=\$TEST\nexport RSRLIB=\$TEST\nexport ARARAT_OBJECT=\$TEST\nexport_
↳DIRECT_ILLUMINATION=\$TEST\nexport BPMS_FILES=\$TEST\n\nif [ "\$(which RATstart)
↳" == "\${bin}/RATstart" ]\nthen\n    if [ "\$VERBOSE" ]; then\n        echo "RATstart_
↳found ok"\n    fi\nelse\n    # we should create them\n    make clean all\nfi\nEOF\n#\n#_
↳set executable mode\nchmod +x $outfile\n# test run\n$outfile\n')

~/opt/anaconda3/envs/librat/lib/python3.8/site-packages/IPython/core/interactiveshell.
↳py in run_cell_magic(self, magic_name, line, cell)
    2360         with self.builtin_trap:
    2361             args = (magic_arg_s, cell)
-> 2362             result = fn(*args, **kwargs)
    2363         return result
    2364

```

(continues on next page)

(continued from previous page)

```

~/opt/anaconda3/envs/librat/lib/python3.8/site-packages/IPython/core/magics/script.py
-> in named_script_magic(line, cell)
    140         else:
    141             line = script
--> 142         return self.shebang(line, cell)
    143
    144         # write a basic docstring:

<decorator-gen-110> in shebang(self, line, cell)

~/opt/anaconda3/envs/librat/lib/python3.8/site-packages/IPython/core/magic.py in
-> <lambda>(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

~/opt/anaconda3/envs/librat/lib/python3.8/site-packages/IPython/core/magics/script.py
-> in shebang(self, line, cell)
    243         sys.stderr.flush()
    244         if args.raise_error and p.returncode!=0:
--> 245             raise CalledProcessError(p.returncode, cell, output=out,
-> stderr=err)
    246
    247     def _run_script(self, p, cell, to_close):

CalledProcessError: Command 'b'\n# create the init.sh file we want\noutfile=test/test_
-> examples/init.sh\nncat <<EOF > $outfile\n#!/bin/bash\n#\n# preamble \n#\nBPMS=\\$
-> {BPMS-\\${pwd}}\n# set shell variables lib, bin, verbose\n# with defaults in case
-> not set \nlib=\\${lib-\\$BPMS/src}\nbin=\\${bin-\\$BPMS/src}\nVERBOSE=\\$
-> {VERBOSE-1}\n\n# set up required environment variables for bash\nexport LD_LIBRARY_
-> PATH=\\${lib}:\\${LD_LIBRARY_PATH}"\nexport DYLD_LIBRARY_PATH=\\${lib}:\\${DYLD_
-> LIBRARY_PATH}"\nexport PATH=\\${bin}:\\${PATH}"\n\n# set up required environment
-> variables for librat\nexport TEST=\\${BPMS}/test/test_example\nexport MATLIB=\\
-> $TEST\nexport RSRLIB=\\$TEST\nexport ARARAT_OBJECT=\\$TEST\nexport DIRECT_
-> ILLUMINATION=\\$TEST\nexport BPMS_FILES=\\$TEST\n\nif [ "\\$(which RATstart)" == "\\
-> ${bin}/RATstart" ]\nthen\n if [ "\\$VERBOSE" ]; then\n     echo "RATstart found ok
-> "\n fi\nelse\n # we should create them\n make clean all \nfi\nEOF\n\n# set
-> executable mode\nchmod +x $outfile\n# test run\n$outfile\n'" returned non-zero exit
-> status 2.

```

[]:

The object code line:

```
usemtl white
```

tells librat to load the material named white. Since we defined that in ``plants.matlib`` as type srm with spectral file ``refl/white.dat`` `<test/test_examples/refl/white.dat>``, the material will have a Lambertian reflectance of 1.0 for all (up to 10000 units) wavelengths.

[]: %%bash

```
cat <<EOF > test/test_examples/white.dat
```

(continues on next page)

(continued from previous page)

```
1 1.0
1000 1.0
EOF
```

```
mtllib plants.matlib
usemtl white
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 1000
ell -1 30000 30000 1000
!}
!}
```

The fields starting `v` in ``test/test_examples/first.obj`` <test/test_examples/first.obj>`__ denote a vertex (vector) (as in the standard wavefront format). This requires 3 numbers to be given after the `v` giving the {x,y,z} coordinates of the vector. Note that `v` fields can specify a *location* or *direction* vector.

The fields `plane` and `ell` specify scene objects. We will look at a fuller range of such objects later, but these two allow for a simple scene specification. `plane` is an infinite planar object. It is defined by an intersection point (location vector) `I` and a direction vector `N`. These vectors need to be defined before a call is made to the object, so in this case, we define `I` as `0 0 0` and `N` as `0 0 1`, i.e. an x-y plane at `z=0`.

Thus `plane -1 -2` means ‘define a plane with `N` given by the previous `(-1)` specified vector that goes through `I` given by the second to last specified vector `(-2)`.’

`ell` is an ellipsoid object. Its description requires definition of:

- the base (N.B. not the centre) of the ellipsoid (`-1` here, meaning the previously-defined vector `- 0 0 1000` in this case);
- the semi-axis lengths in `x, y, z` directions (`30000 30000 1000` here).

so:

```
v 0 0 1000
ell -1 30000 30000 1000
```

is in fact a spheroid of x-y semi-axis length 30000 units (arbitrary linear units) and z-semi-axis length 1000 units: a *prolate* spheroid that extends from `-30000` to `30000` in the x- and y-directions and from `1000` to `3000` in the z-direction. Not that the physical unit for these dimensions is arbitrary, but must be consistent throughout.

The fields `!{` and `!}` in ``test/test_examples/first.obj`` <test/test_examples/first.obj>`__ specify that a bounding box should be placed around objects contained within the brackets. This allows for efficient intersection tests in the ray tracing.

We now want to use the code `RATstart` to run `librat` functionality.

If you have compiled the code, the executable and library should be in the directory ``src`` <src>`__ as

```
src/RATstart
src/libratlib.[dll,so]
```

The suffix for the library will be `dll` on windows, and `so` on other operating systems. Lets just check they are there:

```
[ ]: %%bash

lib='./src'
bin='./src'

ls -l ${lib}/RATstart ${bin}/libratlib.*
```

Don't worry too much if its not there as we can make it when we need it.

```
[ ]: %%bash

#
# shell preamble
#

# set shell variables lib, bin, verbose
# with defaults in case not set
lib=${lib- './src'}
bin=${bin- './src'}
verbose=${verbose-1}

# set up required environment variables for bash
export LD_LIBRARY_PATH="${lib}:${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="${lib}:${DYLD_LIBRARY_PATH}"
export PATH="${bin}:${PATH}"

# set up required environment variables for librat
export TEST=${BPMS}/test/test_example
export MATLIB=$TEST
export RSRLIB=$TEST
export ARARAT_OBJECT=$TEST
export DIRECT_ILLUMINATION=$TEST
export BPMS_FILES=$TEST

if [ $(which RATstart) == './src/RATstart' ]
then
    if [ $verbose ]; then
        echo "RATstart found ok"
    fi
else
    # we should create them
    make clean all
fi
```

4.3 Object example 2: clones

```
[ ]: %%bash

cat <<EOF > test/test_examples/second.obj
!{
mtllib plants.matlib
v 0.000000 0.000000 0.000000
```

(continues on next page)

(continued from previous page)

```
v 0.000000 0.000000 1.000000
usemtl full
plane -1 -2
!{
#define
g object 0
usemtl half
v 0 0 0
v 0 0 1
cyl -1 -2 0.1
sph -1 0.2
v -1 0 1
cyl -1 -2 0.1
!}
!{
clone 0 0 0 0 object 0
clone 0 1 0 90 object 0
clone -1 0 0 -90 object 0
!}
!}
EOF
```


APPENDIX 1: BASH HELP

To use `librat`, we need to have a passing awareness of some computer system settings called `environment variables`. We do this in this chapter, alongside a few other basic linux/unix commands that may be useful to know.

```
[1]: import sys
from prelim import *
%set_env BPMS=$BPMS
%set_env PATH=$BPMS/bin:$BPMS/src:$BPMS/bin/csh:$PATH_
%set_env MATLIB=$BPMS/obj
%set_env BPMSROOT=$BPMS/obj

env: BPMS=/Users/plewis/librat
env: PATH=/Users/plewis/librat/bin:/Users/plewis/librat/src:/Users/plewis/librat/bin/
↪csh:/Users/plewis/opt/anaconda3/bin:/Users/plewis/opt/anaconda3/condabin:/usr/local/
↪bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/VMware Fusion.app/Contents/Public:/
↪Library/TeX/texbin:/opt/X11/bin:/Library/Apple/usr/bin
env: MATLIB=/Users/plewis/librat/obj
env: BPMSROOT=/Users/plewis/librat/obj
```

In practical terms, the important thing here is that you can generate the file `test/test_examples/init.sh` and modify it to your needs. The rest you can skip for now, if you really want to. But you may well find yourself returning to this chapter when you want to ask more of your computer and of this tool.

Our focus will be on `bash` environment variables.

This chapter is not generally critical for understanding `librat` but may help if you go into any details on your setup, or have problems.

The chapter covers:

- Introduction to shell and environment variables
- Some important environment variables and related
- Important environment variables for `librat`

5.1 Introduction to shell and environment variables

5.1.1 export

An **environment variable** is one that is passed through from a shell to any child processes.

We can recognise these as they are usually defined in upper case (capital letters), and (in `bash`) defined with a `export` command: e.g.:

```
export MATLIB=test/test_examples
```

In this case, this would set the environment variable called `MATLIB` to `test/test_examples`. The syntax is:

```
export NAME=value
```

5.1.2 White space and single quotes '

If `value` has white space (gaps in the name), it will need quotes to contain the string, e.g.:

```
export SOMEHWERE='C:/Program Files'
```

Here, we contain the string `C:/Program Files`, which has white space, in single quotes (`'`). It's a good idea to avoid white space in filenames as they can cause problems. Use dash `-` or underscore `_` instead.

5.1.3 echo

We can see the value a variable is set to with the command `echo`, and refer to the *value* of a variable with a `$` symbol e.g.:

```
[2]: %%bash
source examples_init.sh

export MATLIB=$BPMS/obj
echo "MATLIB is set to $MATLIB"

MATLIB is set to /Users/plewis/librat/obj
```

Note that there must be no gaps in `NAME=value` part of the statement. That is a typical thing for new users to get wrong and which can cause problems.

5.1.4 Double quotes " and backslash escape \

If you want to replace the value of a variable in a string, then you should generally use double quotes (`"`) instead of single quotes `'` as above:

```
[3]: %%bash
source examples_init.sh

export MATLIB=$BPMS/obj

echo '1. MATLIB is set to $MATLIB in single quotes'
echo "2. MATLIB is set to $MATLIB in double quotes"
echo "2. MATLIB is set to \"$MATLIB in double quotes but with \ escaping the \"$"

1. MATLIB is set to $MATLIB in single quotes
2. MATLIB is set to /Users/plewis/librat/obj in double quotes
2. MATLIB is set to $MATLIB in double quotes but with \ escaping the $
```

However, we can also ‘escape’ the interpretation of the `$` symbol in the double quoted string, with the backslash escape symbol `\`, as in example 3.

5.1.5 env, grep, pipe |

To see the values of all environment variables, type `env` (or `printenv`). Because this list can be quite long, we might want to select only certain lines from the list. One way to do this is to use the command `grep`, which searches for patterns in the each line:

```
[4]: %%bash
source examples_init.sh
export MATLIB=$BPMS/obj

env | grep M

TERM_PROGRAM=Apple_Terminal
TERM=xterm-color
TMPDIR=/var/folders/mp/9cxd5s793bjd4q3zng6dv_cw0000gn/T/
TERM_PROGRAM_VERSION=433
CONDA_PROMPT_MODIFIER=(base)
SKY_ILLUMINATION==/Users/plewis/librat/obj
TERM_SESSION_ID=D1A31568-329D-434C-986B-E84F76B8FCB3
BPMS=/Users/plewis/librat
TEMP=/tmp
DIRECT_ILLUMINATION=/Users/plewis/librat/obj
KERNEL_LAUNCH_TIMEOUT=40
MATLIB=/Users/plewis/librat/obj
PATH=/Users/plewis/librat/src:/Users/plewis/librat/bin:/Users/plewis/librat/src:/
↪Users/plewis/librat/bin/csh:/Users/plewis/opt/anaconda3/bin:/Users/plewis/opt/
↪anaconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/
↪VMware Fusion.app/Contents/Public:/Library/TeX/texbin:/opt/X11/bin:/Library/Apple/
↪usr/bin
MPLBACKEND=module://ipykernel.pylab.backend_inline
BPMSROOT=/Users/plewis/librat/obj
XPC_SERVICE_NAME=0
_CE_M=
HOME=/Users/plewis
BPMS_FILES=/Users/plewis/librat/obj
LOGNAME=plewis
```

Here, we ‘pipe’ the output of the command `env` into the command `grep` with the pipe symbol `|`. `grep M` will filter only lines containing the character `M`. We see that this includes the variable `MATLIB` that we have set.

EXERCISE

1. Try removing the `'| grep M'` above to see the full `list` of environment variables.
2. Try some other `'grep'` filters, such `as` filtering lines containing the string `'PATH'`

5.1.6 Shell variable

A **shell variable** is one that is *not* passed through from a shell to any child processes. It is only relevant to the shell it is run in.

These are sometimes set as lower case variables (to distinguish from environment variables). The syntax is similar to that of the environment variable, but without the `export`. The syntax is:

```
name=value
```

for example:

```
[5]: %%bash

hello="hello world $USER"
echo $hello

hello world plewis
```

5.1.7 set, head, tail

We can see the values of shell variables with the `set` command.

Like `env`, this is likely to produce a long list. We could filter as above, with `grep`, or here, we use `tail` to take the *last* `N` lines produced or `head` for the first `N` lines. The syntax is:

```
head -N
tail -N
```

```
[6]: %%bash

echo '-----'
echo "1. The first 5 shell variables ..."
echo '-----'
set | head -5
echo

echo '-----'
echo "2. The last 5 shell variables ..."
echo '-----'
set | tail -5
```

```
-----
1. The first 5 shell variables ...
-----
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()

-----
2. The last 5 shell variables ...
-----
XPC_SERVICE_NAME=0
_-----
_CE_CONDA=
_CE_M=
__CF_USER_TEXT_ENCODING=0x1F5:0:2
```

5.2 Some important environment variables and related

When running *any* code, we should be aware of the following shell environment variables:

```
PATH
LD_LIBRARY_PATH
DYLD_LIBRARY_PATH
```

5.2.1 PATH

`$PATH` tells the `shell` where to look for executable files (codes that it can run). This is simply a list of locations (directories) in the computer file system that the shell will look. Elements of the list are separated by `:`. so, if for example we have the `PATH`:

```
PATH="/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin"
```

and tell the shell to run an executable called `ls`, then it will first look in `/usr/local/bin`, then `/usr/bin` and so on, until it finds `ls`.

We have used double quotes `"` around the variable, in case any of the elements had white space (they don't here).

5.2.2 which

We can see which one it finds with the command `which`:

```
[7]: %%bash
which ls
/bin/ls
```

5.2.3 ls

As we saw above, `ls` gives a listing of files and directories. If we use the `-C` option, it outputs multiple columns of information, which is handy if there are lots of entries.

```
[8]: %%bash
# get a listing of the current directory, just to see whats here
ls -C

Appendix1.ipynb      conf.py
Chapter1.ipynb       default.profrw
Chapter2.ipynb       index.rst
RATstart.ipynb       ipython_kernel_config.py
RATstartOptions.ipynb  prelim.py
__pycache__          references.bib
__static__            requirements.txt
__templates__         test
```

5.2.4 .bash_profile, .bashrc, wildcard *

These core environment variables are usually set with default values appropriate to your system. This may be done in system-wide files such as `/etc/profile`, or personal files such as `~/.bashrc` or `~/.bash_profile`, where `~` is the symbol for your home directory. This will almost certainly set `$PATH`.

```
[9]: %%bash
# -d -> no directories

ls -Cd ~/.bash*

/Users/plewis/.bash_history
/Users/plewis/.bash_profile
/Users/plewis/.bash_profile-anaconda3.bak
/Users/plewis/.bash_profile.backup
/Users/plewis/.bash_sessions
```

above, we use the wildcard symbol `*`, interpreted by the shell as any file matching the pattern `~/.bash*` with `*` being zero or more characters. The `~` is matched to the user's home directory name in this case.

For many purposes, the default options to `ls` will do. The `-C` option we would hardly use, but is useful above for better note formatting. The `-d` option is again rarely used, but useful in this case as we only want to see files in the home directory.

5.2.5 `ls -l`

One useful option to `ls` is `-l`, that gives 'long listing':

```
[10]: %%bash
# -d -> no directories

ls -lhd ~/.bash*

-rw-----  1 plewis  staff   1.5K 21 Apr 10:26 /Users/plewis/.bash_history
-rw-r--r--  1 plewis  staff   3.0K 17 Apr 17:00 /Users/plewis/.bash_profile
-rw-r--r--  1 plewis  staff   1.1K 15 Jul  2019 /Users/plewis/.bash_profile-
↪ anaconda3.bak
-rw-r--r--  1 plewis  staff   727B 15 Jul  2019 /Users/plewis/.bash_profile.backup
drwx----- 111 plewis  staff   3.5K 21 Apr 10:26 /Users/plewis/.bash_sessions
```

The `-l` option gives the file sizes and other useful information in this 'long' listing. The file sizes here are given in K or other human-readable (^3) units, as we have set the `-h` option. Many unix commands that involve file sizes will have a similar `-h` option.

The first set of information, such as `-rw-r--r--` gives us information on file permissions. It represents a 10-bit field, where bits are set 'on' (1) or off (0). After the first bit (the sticky bit), the fields are 3 sets of 3-bit fields (so, octal - base 8 = 2^3). These 3 bits represent `rx`, with

- `r`: read permission
- `w`: write permission
- `x`: execute permission

So:

- `rw-` means that permission is set for reading the file and writing to it
- `r--` means reading but not writing
- `rwX` means reading, writing and execute

The first set of 3 bits represents permissions for the file owner, the second for users in the same group, and the third for all users (others).

So:

- `-rw-r--r--` means read and write for the owner, but only read permission for group and all. This is the typical setting for a non-executable file: everyone can read it, but only the owner can write. In octal, this is 644.
- `-rwxr-xr-x` means read, write and execute for the owner, and read and execute permission for others. This is the typical setting for an executable file: everyone can execute it and read it, but only the owner can write. In octal, this is 755.

In fact, the final ‘bit’, known as the [sticky bit](#) can have more settings than just `-` or `x`, but we need not worry about that here.

5.2.6 `chmod`, `>`, `rm -f`, `mkdir -p`

We can change the file permissions, using the command `chmod`. Most typically, we use options such as `+x` to add an executable bit, or `go-r` to remove read permissions (for group and other, here).

We create a file in a directory `files.$$`, where `$$` is the [shell process ID](#) which we can use to give probably a unique directory name (i.e. one very unlikely to be created by any other process). First, we must create (make) the directory if it doesn’t already exist. This is done with `mkdir -p`. The `-p` option will not fail if the directory already exists, and also will create any depth of directories specified.

The file is called `files/hello.dat` and is created by [redirecting the standard output](#) (`stdout`) of a command to a file, i.e. sending the text coming from `echo "hello world"` into the file. The symbol for redirection of `stdout` is `>`. This redirection is the same process used above when we redirected output to a pipe.

Just in case the file already exists, and we have previously messed around with the file permissions, we first run the command `rm -f` to delete (remove) the file. The `-f` option tells us to ‘force’ this, regardless of the file’s permissions or whether the file already exists. At the end of the shell, we use `rm -rf` to delete the directory and anything in it (a recursive delete).

```
[11]: %%bash

# create a unique directory name
dir=/tmp/files.$$

# make directory
mkdir -p $dir

# force delete the file, in case it exists
rm -f $dir/hello.dat

# generate the file
# it should contain 11 characters (bytes) plus
# an End Of File (EOF) character (^D), so 12B
echo "hello world" > $dir/hello.dat

# listing
# The default permission should be rw-r--r--
ls -lh $dir/hello.dat

# We now remove the read permissions using chmod
# The permission should be rw-----
chmod go-r $dir/hello.dat
ls -lh $dir/hello.dat
```

(continues on next page)

(continued from previous page)

```
# now add user execute
chmod u+x $dir/hello.dat
ls -lh $dir/hello.dat

# clean up after ourselves
# remove everything in files.$$ , along with the directory
rm -rf $dir
```

-rw-r--r--	1	plewis	wheel	12B	21	Apr	18:47	/tmp/files.29478/hello.dat
-rw-----	1	plewis	wheel	12B	21	Apr	18:47	/tmp/files.29478/hello.dat
-rwx-----	1	plewis	wheel	12B	21	Apr	18:47	/tmp/files.29478/hello.dat

5.2.7 cat

We can use the command `cat` to create or to ‘view’ the contents of a file. For example, the command:

```
cat ~/.bash_profile
```

would ‘print’ (send to the terminal, rather) the contents of the file `~/.bash_profile`.

Since this may be quite long, we will use `head` just to see the first N lines:

```
[12]: %%bash
cat ~/.bash_profile | head -5

# added by Anaconda3 2019.03 installer
# >>> conda init >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$(CONDA_REPORT_ERRORS=false '/anaconda3/bin/conda' shell.bash hook 2> /
↪dev/null)"
if [ $? -eq 0 ]; then
```

5.2.8 pwd, cd

The command `pwd` returns the **current working directory**. This is extremely useful to know, especially as new users often get lost in a shell on the file system. To find out where you are, in a shell, type:

```
pwd
```

This will return the ‘location’ you are at in that shell.

The command `cd` is used to change directory. The syntax is:

```
cd location
```

where `location` is somewhere on the file system.

```
[13]: %%bash
source examples_init.sh

echo -n "where am I now?: "
pwd
```

(continues on next page)

(continued from previous page)

```
# go home using 'cd ~'
echo "go home (~): "
cd ~
echo -n "where am I now?: "
pwd

# go to directory librat 'cd librat'
echo "go to librat: "
cd $BPMS
echo -n "where am I now?: "
pwd

where am I now?: /Users/plewis/librat/docs/source
go home (~):
where am I now?: /Users/plewis
go to librat:
where am I now?: /Users/plewis/librat
```

5.2.9 \$(pwd)

Sometimes we want to set a variable to the result returned by running an executable. For example, the command `pwd` returns the [current working directory](#). We can set a variable to this, with the following example syntax:

```
PWD=$(pwd)
```

Note the round brackets `$ ()` enclosing the command (`pwd` here).

```
[14]: %%bash

# set PWD to the result of running `pwd`
echo -n "1. Run the command pwd: "
pwd

# Note the use of \$(pwd) in printing here. This will make sure $ is printed,
# rather than $(pwd) in this statement
echo "2. Set the variable PWD the result of running the command pwd with PWD=\$(pwd):"

PWD=$(pwd)

echo "3. Now print that out: PWD is set to $PWD"

1. Run the command pwd: /Users/plewis/librat/docs/source
2. Set the variable PWD the result of running the command pwd with PWD=$(pwd):
3. Now print that out: PWD is set to /Users/plewis/librat/docs/source
```

5.2.10 `${BPMS-${pwd}}`

In bash we often use syntax that only sets a variable if it is not already set. This is done in the example:

```
BPMS=${BPMS-${pwd}}
```

where some variable BPMS is set to the result of running `pwd`, unless it is already set.

Note the curly brackets in `${}`.

Note that the environment BPMS is generally used to define the top level directory of librat codes.

```
[15]: %%bash
source examples_init.sh

#
# example using ${BPMS-${pwd}}
#




# set BPMS variable to result of pwd, unless its already set
BPMS=${BPMS-${pwd}}
echo "1. BPMS set to $BPMS because BPMS is set"

# unset the variable, so its no longer set
unset BPMS
BPMS=${BPMS-${pwd}}
echo "2. BPMS set to $BPMS, from running pwd, because BPMS is not set"

1. BPMS set to /Users/plewis/librat because BPMS is set
2. BPMS set to /Users/plewis/librat/docs/source, from running pwd, because BPMS is
↳not set
```

5.2.11 edit

If you want to make changes to important environment variables, you would normally edit them in your `.bash_profile` file in your home directory. Here is an exercise to do that. It assumes that you know: (i) the location in the filesystem of your librat distribution; (ii) some text file editor (N.B. **Not** Microsoft word or similar: that is a word processor, not a text editor!). Examples would be:

		
textedit	Notepad	gedit
vi(m)	vi(m)	vi(m)

EXERCISE

1. Make a copy of your `~/.bash_profile`, just in case you mess things up. Do this,
↳only the once!

```
cp ~/.bash_profile ~/.bash_profile.bak
```

If the file doesn't already exist, don't worry about this part

2. Find out where your librat installation is located e.g. `/Users/plewis/librat`)

(continues on next page)

(continued from previous page)

3. Now, edit the file `~/.bash_profile` and add a line at the end of the file that `↪`says (the **equivalent** of):

```
export BPMS=/Users/plewis/librat
```

where you use the location of your librat distribution.

4. Save the file and quit the editor.

5. Open a new shell. At the command prompt, type:

```
source ~/.bash_profile
```

Then

```
echo $BPMS
```

It should show the value you set it to.

999. If you get stuck, or think you have messed up, copy the original `bash_↪`profile file back in place:

```
cp ~/.bash_profile.bak ~/.bash_profile
```

Then source that in a shell:

```
source ~/.bash_profile
```

to (mostly) set things back to how they were before.

5.2.12 Update PATH

Recall that `PATH` is a list (separated by `:`) fo directories to search for executables, e.g.:

```
PATH="/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin"
```

Then, if we want to put a `librat` directory at the front of this path (so we look there first), we follow the following example syntax:

```
[16]: %%bash
source examples_init.sh

# example initial setting of PATH
# NB Only an example, your shell will set something
# different!
PATH="/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin"

echo "1. PATH is $PATH"

# change directory from docs/source up to root
BPMS=${BPMS}-${(pwd)};

bin=$BPMS/src

# put $bin on the front of PATH
```

(continues on next page)

(continued from previous page)

```
export PATH="$bin:$PATH"
```

```
echo "2. PATH is $PATH"
```

1. PATH is /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
2. PATH is /Users/plewis/librat/src:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

EXERCISE

1. Edit your ~/.bash_profile to update your PATH variable

You should type the following lines into the end of ~/.bash_profile:

```
# replace this line below by BPMS= the location of your librat dist
BPMS=/Users/plewis/librat
bin=$BPMS/src
export PATH="$bin:$PATH"
```

2. Save the file and quit the editor.

3. Open a new shell. At the command prompt, type:

```
source ~/.bash_profile
```

Then

```
echo $PATH
```

It should show the updated PATH variable.

5.2.13 LD_LIBRARY_PATH, DYLD_LIBRARY_PATH

On some systems, LD_LIBRARY_PATH and/or DYLD_LIBRARY_PATH may be set in your bash shell. Just to make sure, we will set them in our examples.

These variables tell an executable where to look for shared object libraries (libraries of functions stored on the computer). Again, they are simply lists of locations (directories) in the computer file system that the shell will look. Elements of the list are separated by :. so, if for example we have the PATH:

```
LD_LIBRARY_PATH="/usr/local/lib:/usr/lib"
DYLD_LIBRARY_PATH="/usr/local/lib:/usr/lib"
```

then when an executable makes a call to a function in a shared object library, it will look first in /usr/local/lib, and then in /usr/lib for these libraries.

5.2.14 Update LD_LIBRARY_PATH, DYLD_LIBRARY_PATH

We can again add search directories to the front of the library paths:

```
[17]: %%bash
source examples_init.sh

# example initial setting of LD_LIBRARY_PATH
# NB Only an example, your shell will set something
# different!
LD_LIBRARY_PATH="/usr/local/lib:/usr/lib"

echo "1. LD_LIBRARY_PATH is $LD_LIBRARY_PATH"

# change directory from docs/source up to root
lib=$BPMS/src

# put $bin on the front of PATH
export LD_LIBRARY_PATH="$lib:$LD_LIBRARY_PATH"

echo "2. LD_LIBRARY_PATH is $LD_LIBRARY_PATH"

1. LD_LIBRARY_PATH is /usr/local/lib:/usr/lib
2. LD_LIBRARY_PATH is /Users/plewis/librat/src:/usr/local/lib:/usr/lib
```

```
[18]: %%bash
source examples_init.sh

# example initial setting of DYLD_LIBRARY_PATH
# NB Only an example, your shell will set something
# different!
DYLD_LIBRARY_PATH="/usr/local/lib:/usr/lib"

echo "1. DYLD_LIBRARY_PATH is $DYLD_LIBRARY_PATH"

lib=$BPMS/src

# put $bin on the front of PATH
export DYLD_LIBRARY_PATH="$lib:$DYLD_LIBRARY_PATH"

echo "2. DYLD_LIBRARY_PATH is $DYLD_LIBRARY_PATH"

1. DYLD_LIBRARY_PATH is /usr/local/lib:/usr/lib
2. DYLD_LIBRARY_PATH is /Users/plewis/librat/src:/usr/local/lib:/usr/lib
```

EXERCISE

1. Similar to the previous exercise, edit your ~/.bash_profile to now update your `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` variables

You should type the following lines into the end of ~/.bash_profile:

```
# replace this line below by BPMS= the location of your librat dist
BPMS=/Users/plewis/librat
lib=$BPMS/src
export LD_LIBRARY_PATH="$lib:$LD_LIBRARY_PATH"
export DYLD_LIBRARY_PATH="$lib:$DYLD_LIBRARY_PATH"
```

(continues on next page)

(continued from previous page)

```
2. Save the file and quit the editor.

3. Open a new shell. At the command prompt, type:

    source ~/.bash_profile

Then

    echo $LD_LIBRARY_PATH $DYLD_LIBRARY_PATH

It should show the updated variables.
```

5.2.15 Which operating system? `uname`, `if`

Before proceeding, it is useful to see how to determine which operating system we are using, and how to perform conditional statements in `bash`.

Mostly, you can get information on which operating system you are using by using either `uname -s`. You may sometime have problems if you are using virtual machines of any sort, as the top level operating system may not be apparent.

In the example below, we use `uname -s` to test for values of `MINGW64` (a common windows environment with compilers and some other useful features), `Darwin` (macOS of some sort), or other (assumed linux).

We set the variable `OS` to the result of running `uname -s`, then use `bash` conditional statement syntax:

```
if [ $VAR = value1 ]
then
    ... do something 1 ...
elif [ $VAR = value2 ]
then
    ... do something 2 ...
else
    ... do something else ...
fi
```

to test the options we consider. The syntax is a little fiddly.

Note that the spaces in `if [$VAR = value1]` are critical. Note that the `then` statements are also critical.

```
[19]: %bash
source examples_init.sh

# these to see what sort of computer we are running on
OS=$(uname -s)

# print the first 5 lines in the shared object
if [ $OS = MINGW64 ]
then
    echo "I am windows: $OS"
elif [ $OS = Darwin ]
then
    echo "I am macOS: $OS"
else
```

(continues on next page)

(continued from previous page)

```

    echo "I am neither macOS nor MINGW64: $OS"
fi
I am macOS: Darwin

```

5.2.16 Contents of libraries: nm or ar

The libraries will have the suffix `dll` on windows systems. On various unix systems, they may be `so` or for on OS X, `dylib`. Normally, you will only need `DYLD_LIBRARY_PATH` on OS X, but we might as well set it for all cases. If you want to see which functions are contained in a particular library then:

On OS X:

```
nm -gU src/libratlib.${ext}
```

Otherwise:

```
ar tv src/libratlib.${ext}
```

where `${ext}` is `so` or `dll` or `dylib` as appropriate. We use the construct above for determining the operating system and for using `ar` or `nm` as appropriate.

```

[20]: %%bash
source examples_init.sh

# these to see what sort of computer we are running on
OS=$(uname -s)
echo $OS

lib=$BPMS/src

# print the first 5 lines in the shared object
if [ $OS = MINGW64 ]
then
    # windows
    ar tv $lib/libratlib.dll | head -5
elif [ $OS = Darwin ]
then
    # OS X
    nm -gU $lib/libratlib.so | head -5
else
    # linux
    ar tv $lib/libratlib.dll | head -5
fi

Darwin
00000000000475f0 T _Add_2D
00000000000478b0 T _Affine_transform
0000000000049180 T _B_allocate
00000000000478e0 T _Backwards_affine_transform
00000000000471a0 T _Bbox

```

where we see that the shared object library for `librat` (in a file called `libratlib.${ext}`) contains some functions `_Add_2D()`, `_Affine_transform()` etc. which are part of the library we use.

Notice that `lib.${ext}` is added on the end of a library name to give its filename.

5.3 Important environment variables for librat

5.3.1 `cat <<EOF > output ... EOF`

We can conveniently create files in bash from text in the bash shell. This is done using `cat` and defining a marker (often EOF, meaning End Of File), such as:

```
[21]: %%bash
source examples_init.sh

# change directory from docs/source up to root

cat <<EOF > $BPMS/obj/second.obj
# My first object file
mtllib plants.matlib
usemtl white
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 1000
ell -1 30000 30000 1000
!}
!}
EOF
```


Let's look at the file we have just created:

```
[22]: %%bash
source examples_init.sh

cat $BPMS/obj/second.obj

# My first object file
mtllib plants.matlib
usemtl white
v 0 0 0
v 0 0 1
plane -1 -2
!{
usemtl white
!{
v 0 0 1000
ell -1 30000 30000 1000
!}
!}
!}
```

EXERCISE

Use the approach above (``cat <<EOF > output ... EOF``) to create your own text file,  then check the contents are as you expected.

5.3.2 MATLIB, RSRLIB etc.

In `librat`, there is a considerable set of data that we need to describe world data for any particular simulation. For example, we need to have one or more object files giving the geometry, material files describing the spectral scattering properties of materials, sensor spectral response functions etc.

To try to make models and simulation scenarios portable, we want to avoid ‘hardwiring’ these file locations. One way to do that is to simply use relative file names throughout the description, so that we can then determine the full filenames from some core base directory.

If we happen to run the simulation *from* this directory, then clearly the relative filenames we use would directly describe all file locations.

However, if we run the simulation from elsewhere on the system, we need a mechanism to describe the *base* of the scene description files. More generally, we might want to store spectral response files in one part of the file system, and spectral scattering properties elsewhere. In that case, we need a set of *base* descriptors for these different types of file.

That is the file system philosophy used in `librat`. These *base* locations are defined by environment variables which we will describe below. Whilst you do not *have* to use these, it makes sense to set them up, even if they are all set to the same value (i.e. the *base* of the model files is the same for all file types).

The following environmental variables can be used:

Name	File types
MATLIB	material library e.g. <code>plants.matlib</code> , all materials defined in a material library e.g. <code>white.dat</code>
ARARAT_OBJECT	(extended) wavefront object files e.g. <code>first.obj</code>
DIRECT_ILLUMINATION	Spectral files for direct illumination: those defined in <code>-RATdirect</code> command line option
RSRLIB	sensor waveband files: those defined in <code>-RATsensor_wavebands</code> command line option
BPMS_FILES	Not used
SKY_ILLUMINATION	location of sky map image files: defined in <code>-RATskymap</code> command line option

As noted, you can set all of these to the same value, in which case the database of files is all defined relative to that point. This is the most typical use of `librat`. We illustrate this setup below for the `librat` distribution, where a set of examples use files from the directory `test/test_examples`.

Additionally, the following environment variables can be set to extend the size of some aspects of the model. You would only need to use these in some extreme case.

Name	Purpose
MAX_GROUPS	Maximum number of groups allowed (100000)
PRAT_MAX_MATERIALS	Maximum number of materials allowed (DEFAULT_PRAT_MAX_MATERIALS=1024 in <code>mtllib.h</code>)
MAX_SUNS	Maximum number of suns (180 in <code>rat.h</code>)

```
[23]: %%bash
#
# create examples_init.sh
# for examples initialisation
#
# create the init shell
cat <<EOF > $BPMS/bin/examples_init_test.sh
```

(continues on next page)

(continued from previous page)

```
#!/bin/bash
#
# defaults
#
export BPMS=\${BPMS-$BPMS}
export BPMSROOT=\${BPMSROOT-$BPMSROOT}
lib=\${lib-$BPMS/src}
bin=\${bin-$BPMS/src}
VERBOSE=\${VERBOSE-0}
export TEMP=\${TEMP-/tmp}

# set up required environment variables for bash
export LD_LIBRARY_PATH="\${lib}:\${LD_LIBRARY_PATH}"
export DYLD_LIBRARY_PATH="\${lib}:\${DYLD_LIBRARY_PATH}"
export PATH="\${bin}:\${PATH}"

export MATLIB=\$BPMSROOT
export RSRLIB=\$BPMSROOT
export ARARAT_OBJECT=\$BPMSROOT
export DIRECT_ILLUMINATION=\$BPMSROOT
export BPMS_FILES=\$BPMSROOT
export SKY_ILLUMINATION==\$BPMSROOT

if [ "\$(which RATstart)" == "\${bin}/RATstart" ]
then
    if [ "\$VERBOSE" == 1 ]; then
        echo "RATstart found ok"
    fi
else
    # we should create them
    make clean all
fi
EOF
chmod +x $BPMS/bin/examples_init_test.sh
examples_init_test.sh
```

5.4 Summary

In this chapter, we have covered a range of basic unix/linux and `bash` commands, so you should be able to navigate your way around a unix file system, and find your way back safely. Being familiar with these tools takes some time of course, so you might now want to go on and take [some other unix/linux course](#) to see if you can deepen your understanding in that way. Alternatively, just spend some time exploring your system, looking to see what files are where, reading on the internet or help pages what they do, and so on.

Maybe that's wishful thinking on my part though. You may not feel you have time for basic unix at the moment ... and we did say at the top of this chapter that it was not compulsory ... I'd recommend you *do* spend some time on *unix* ... you'll develop skills that will last you a lifetime! ;-)

In practical terms, as we have said, the important thing here is that you can generate the file `examples_init_test.sh` and modify it to your needs.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`