librat

Prof. P. Lewis

Apr 19, 2020

CONTENTS:

UCL Geography/NCEO librat software

CHAPTER

ONE

INITIALISATION

Before proceeding, we make sure that an appropriate initialisation ('init') file has been generated. We also want to test that the software runs ok. We generate a bash shell to achieve this here, and use to his 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 it, not understand all of the intricacies.

[1]:
```
%%bash

# change directory from docs/source up to root
cd ../..
BPMS=${BPMS-$(pwd)}
# create the init.sh file we want
INIT=$BPMS/test/test_examples/init.sh

cat <<EOF > $INIT
#!/bin/bash
#
# preamble
#
export BPMS=$BPMS
# set shell variables lib, bin, verbose
# with defaults in case not set
lib=\${lib-"\$BPMS/src"}
bin=\${bin-"\$BPMS/src"}
VERBOSE=\${VERBOSE-0}

# 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 BPMSROOT=\${BPMSROOT-\$BPMS/test/test_examples}

export MATLIB=\$BPMSROOT
export RSRLIB=\$BPMSROOT
export ARARAT_OBJECT=\$BPMSROOT
export DIRECT_ILLUMINATION=\$BPMSROOT
export BPMS_FILES=\$BPMSROOT
if [ "\$(which start)" == "\${bin}/start" ]
then
```
(continues on next page)

(continued from previous page)

```
  if [ "\$VERBOSE" == 1 ]; then
      echo "start found ok"
  fi
else
  # we should create them
  make clean all
fi
EOF
# set executable mode
chmod +x $INIT
```

Let's look at the file we generated:

[2]:
```
%%bash

cat ../../test/test_examples/init.sh
```

1.1 Environment variables

We might notice that the init shell sets a number of environment variables, namely `MATLIB`, `RSRLIB` etc.

If you are interested, 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/test/test_example`.

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

[3]:
```
%%bash

source ../../test/test_examples/init.sh

echo "MATLIB is set to $MATLIB"
```

Table explaining librat object environment variables:

Name	File types
MATLIB	material library e.g. `plants.matlib <test/test_examples/plants.matlib>`__, all materials defined in a material library e.g. `refl/white.dat <test/test_examples/refl/white.dat>`__
ARARAT_OBJECT	(extended) wavefront object files e.g. `first.obj <test/test_examples/first.obj>`__
DIRECT_ILLUMINATION	spectral files for direct illumination: those defined in -RATdirect command line option
RSRLIB	sensor waveband files: those defined in -RATsensor_wavebands command line option
BPMS_FILES	Not used

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.

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 mtllib.h)

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

[4]:
```
%%bash
# test the init file

# change directory from docs/source up to root
INIT=../../test/test_examples/init.sh

export VERBOSE=1
$INIT
```

```
start found ok
```

```
EXERCISE 1.1

    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 yoiu can modify it.
```

Answers below:

[5]:
```
%%bash
#----------------------
# this part same as above
#
# test the init file
INIT=../../test/test_examples/init.sh
#----------------------

# 1.1.1: 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
# 'start found ok' if it finds the librat start
# executable
echo  "----set VERBOSE 1---"
export VERBOSE=1
$INIT

# this turns off the verbose mode
# so no message is printed
echo "----set VERBOSE 0---"
export VERBOSE=0
$INIT

# 1.1.2: You can change the name of the directory where the
# object and material files are through the
# environment variable BPMSROOT.
```

(continues on next page)

(continued from previous page)

```
# See if you can find what BPMSROOT is set to,
# and also see if yoiu 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 $INIT
echo "BPMSROOT is $BPMSROOT"
echo "----set BPMSROOT---"
# To change it, just set it before sourcing
export BPMSROOT="/tmp"
source $INIT
echo "BPMSROOT is $BPMSROOT"
```

```
----set VERBOSE 1---
start found ok
----set VERBOSE 0---
----get BPMSROOT---
BPMSROOT is /Users/plewis/librat/test/test_examples
----set BPMSROOT---
BPMSROOT is /tmp
```

1.2 Making files

We will create the files we need as we go along, using `bash` shell `cat <<EOF > filename` syntax. 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
```

Wew can try that now:

We first create a directory to do our work (using the `linux` command `mkdir` in a `bash` shell):

[18]:
```
%%bash
export BPMS=../..
mkdir -p $BPMS/test/test_examples
```

Now we will 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.

[67]:
```
%%bash
export BPMS=../..
```

(continues on next page)

(continued from previous page)

```
cd $BPMS
export BPMS=$(pwd)

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

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

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

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

# library listing materials
cat <<EOF > $BPMS/test/test_examples/plants.matlib
srm green green.dat
srm white white.dat
EOF
```

And now run a simple example:

[68]:
```
%%bash
export BPMS=../..
cd $BPMS
export BPMS=$(pwd)
source $BPMS/test/test_examples/init.sh
```

(continues on next page)

(continued from previous page)

```
echo 11 | start -RATsensor_wavebands $BPMS/test/test_examples/wavebands.dat \
                -RATsun_position 0 1 1 $BPMS/test/test_examples/first.obj
```

```
x: -99.980001 100.020001
y: -99.980001 100.020001
z: -99.980001 100.020001
bbox centre @ 0.020000 0.020000 0.020000
```

[69]:

```
%%bash
export BPMS=../..
cd $BPMS
export BPMS=$(pwd)
source $BPMS/test/test_examples/init.sh


echo "16 0 0 400 400 400  200 200 1 $BPMS/test/test_examples/out.hips" | start \
    -RATsensor_wavebands $BPMS/test/test_examples/wavebands.dat  \
    -RATv  -RATsun_position 0 1 1 $BPMS/test/test_examples/first.obj
```

```
start:
        VERBOSE flag on (-v option)
read_spectral_file:   3 data entries read in file /Users/plewis/librat/test/test_
↪examples/wavebands.dat
( 99.9975)
```

[70]:

```
from libhipl import Hipl
import pylab as plt

f = '../../test/test_examples/out.hips'
rabbit=Hipl().read(f)
plt.imshow(rabbit,cmap='gray')
plt.colorbar()
```

[70]:

```
<matplotlib.colorbar.Colorbar at 0x116d4a6a0>
```

[65]:

```
%%bash
export BPMS=../..
cd $BPMS
export BPMS=$(pwd)
source $BPMS/test/test_examples/init.sh

echo 6 0 0 110 0 0 -1 | start -RATsensor_wavebands $BPMS/test/test_examples/wavebands.
↪dat -RATsun_position 0 1 1 $BPMS/test/test_examples/first.obj
```

```
RTD 0
order: 0        intersection point:     0.000000 0.000000 100.000010
                ray length:             9.999990
                intersection material:  3
                sun 0:                  no hit
                sky  :                  reflectance
                diffuse:                1.000000 1.000000
```

[17]:

```
%%bash

### wavebands
```

(continues on next page)

(continued from previous page)

```
export BPMS=../..
source $BPMS/test/test_examples/init.sh


for i in $(seq 1 2 40)
do
cat <<EOF | start -RATr $i $BPMS/test/test_examples/first.obj
3 3 200 20 300 20 400 20
4
EOF
done


# wavelength is min and width
```

```
wavebands: 202.630756 315.112106 409.173003
wavebands: 207.892267 305.336319 407.519008
wavebands: 213.153779 315.560532 405.865013
wavebands: 218.415290 305.784745 404.211018
wavebands: 203.676802 316.008958 402.557024
wavebands: 208.938313 306.233171 400.903029
wavebands: 214.199825 316.457384 419.249034
wavebands: 219.461336 306.681597 417.595040
wavebands: 204.722848 316.905810 415.941045
wavebands: 209.984359 307.130022 414.287050
wavebands: 215.245871 317.354235 412.633055
wavebands: 200.507383 307.578448 410.979061
wavebands: 205.768894 317.802661 409.325066
wavebands: 211.030406 308.026874 407.671071
wavebands: 216.291917 318.251087 406.017077
wavebands: 201.553429 308.475300 404.363082
wavebands: 206.814940 318.699513 402.709087
wavebands: 212.076452 308.923726 401.055092
wavebands: 217.337963 319.147938 419.401098
wavebands: 202.599475 309.372151 417.747103
```

[28]:
```
%%bash
export BPMS=../..
source $BPMS/test/test_examples/init.sh
echo 10000 | start $BPMS/test/test_examples/first.obj
```

```
options:
        -2                   : print PID
        -1                   : print memory use
         0                   : quit
         1 n s1x s1y s1z ... : set sun vectors
         2                   : print sun vectors
         3 n b1 w1 ...i bn wn: set wavebands
         4                   : print wavebands
         5 file.obj          : read object file
         6 fx fy fz dx dy dz : trace ray from f in direction d
         7                   : get and print materials
         8                   : print object information
         9                   : print info on materials used
         10                  : get and set verbosity level (0-1)
```

(continues on next page)

(continued from previous page)

```
        11                   : get and print object bbox information
        12                     :
        13                   : same as 14 assuming filenames camera.dat light.dat
        14 camera.dat light.dat                 : ray tracing using defined camera &␣
↪illumination
        15                     : dont go there
        16 cx cy cz sx sy nrows ncols rpp name : produce a height map in name
```

[ ]:

```

```

[ ]:

```

```

CHAPTER

TWO

BASIC LIBRAT / START 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, 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 start 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.

2.1 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
```

(continues on next page)

(continued from previous page)

```
ell -1 30000 30000 1000
!}
!}
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.

2.2 Environment variables

The following environmental variables can be used:

Name	File types
MATLIB	material library e.g. `plants.matlib <test/test_examples/plants.matlib>`__, all materials defined in a material library e.g. `refl/white.dat <test/test_examples/refl/white.dat>`__
ARARAT_OBJECTS	(extended) wavefront object files e.g. `first.obj <test/test_examples/first.obj>`__
DIRECT_ILLUMINATION	spectral files for direct illumination: those defined in -RATdirect command line option
RSRLIB	sensor waveband files: those defined in -RATsensor_wavebands command line option
BPMS_FILES	Not used

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 `mtllib.h`)

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"}
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}"
```

(continues on next page)

(continued from previous page)

```
# 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 start)" == "\${bin}/start" ]
then
  if [ "\$VERBOSE" ]; then
      echo "start 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\n\ncat <<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 start)"␣
↪== "\\${bin}/start" ]\nthen\n  if [ "\\$VERBOSE" ]; then\n      echo "start 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

~/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
```

(continues on next page)

(continued from previous page)

```
    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\n\ncat <<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 start)" == "\\$
↪{bin}/start" ]\nthen\n  if [ "\\$VERBOSE" ]; then\n      echo "start 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 <test/test_examples/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
1 1.0
1000 1.0
EOF
```

```
mtllib plants.matlib
usemtl white
v 0 0 0
```

(continues on next page)

(continued from previous page)

```
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 `start` to run `librat` functionality.

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

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

2.3 Object example 2: clones

[ ]:
```
%%bash

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

(continues on next page)

(continued from previous page)

```
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
```

CHAPTER

THREE

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.

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

3.1 Introduction to shell and environment variables

3.1.1 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
```

3.1.2 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 (`'`). Its a good idea to avoid white space in filenames as they can cause problems. Use dash `-` or underscore `_` instead.

3.1.3 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.:

[1]:
```
%%bash

export MATLIB=test/test_examples
echo "MATLIB is set to $MATLIB"
```

```
MATLIB is set to test/test_examples
```

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.

3.1.4 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:

[2]:
```
%%bash

export MATLIB=test/test_examples

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 test/test_examples 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.

3.1.5 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:

[3]:
```
%%bash

export MATLIB=test/test_examples

env | grep M
```

```
TERM_PROGRAM=Apple_Terminal
TERM=xterm-color
TMPDIR=/var/folders/mp/9cxd5s793bjd4q3zng6dv_cw0000gn/T/
CONDA_PROMPT_MODIFIER=(librat)
TERM_PROGRAM_VERSION=433
GSETTINGS_SCHEMA_DIR_CONDA_BACKUP=
TERM_SESSION_ID=7FC61198-BCF5-4CED-8B68-076E150723FD
KERNEL_LAUNCH_TIMEOUT=40
MATLIB=test/test_examples
PATH=/Users/plewis/opt/anaconda3/envs/librat/bin:/Users/plewis/opt/anaconda3/condabin:
↪/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/VMware Fusion.app/
↪Contents/Public:/opt/X11/bin:/Library/Apple/usr/bin
GSETTINGS_SCHEMA_DIR=/Users/plewis/opt/anaconda3/envs/librat/share/glib-2.0/schemas
MPLBACKEND=module://ipykernel.pylab.backend_inline
_CE_M=
XPC_SERVICE_NAME=0
HOME=/Users/plewis
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'
```

3.1.6 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:

[4]:
```
%%bash

hello="hello world $USER"
echo $hello
```

```
hello world plewis
```

3.1.7 `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
```

[5]:
```
%%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
```

3.2 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
```

3.2.1 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).

3.2.2 which

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

[6]:
```
%%bash

which ls
```

```
/bin/ls
```

3.2.3 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.

[7]:
```
%%bash

# get a listing of the current directory, just to see whats here
ls -C
```

```
Appendix1.ipynb                 default.profraw
Chapter1.ipynb                  index.rst
Chapter2.ipynb                  ipython_kernel_config.py
_static                                references.bib
_templates                     requirements.txt
conf.py                         test
```

3.2.4 .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`.

[8]:
```
%%bash
# -d -> no directories

ls -Cd  ~/.bash*
```

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

(continues on next page)

(continued from previous page)

```
/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.

3.2.5 `ls -l`

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

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

ls -lhd  ~/.bash*
```

```
-rw-------    1 plewis  staff    27K 17 Apr 18:32 /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------  108 plewis  staff   3.4K  2 Feb 14:42 /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 `rwx`, 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.

3.2.6 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 alreay 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 anythinbg in it (a recursice delete).

[10]:
```
%%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

# 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 18 Apr 20:41 /tmp/files.88512/hello.dat
-rw-------  1 plewis  wheel    12B 18 Apr 20:41 /tmp/files.88512/hello.dat
-rwx------  1 plewis  wheel    12B 18 Apr 20:41 /tmp/files.88512/hello.dat
```

3.2.7 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:

[11]:
```
%%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
```

3.2.8 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.

[12]:
```
%%bash


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

# 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 librat
echo -n "where am I now?: "
pwd

# go to directory librat 'cd ~/librat'
echo "go to ~/librat: "
cd ~/librat
```

(continues on next page)

(continued from previous page)

```
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
go to ~/librat:
where am I now?: /Users/plewis/librat
```

3.2.9 `$(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).

[13]:
```
%%bash

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

# Note the use of \$ 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
```

3.2.10 `${BPMS-$(pwd)}`

In `bash` we often use syntax that only sets a valiable 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 curley brackets in `${}`.

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

[14]:
```
%%bash

#
# example using ${BPMS-$(pwd)}
#
```

(continues on next page)

(continued from previous page)

```
# set BPMS variable to result of pwd, unless its already set
BPMS="Previous Value"
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 Previous Value because BPMS is set
2. BPMS set to /Users/plewis/librat/docs/source, from running pwd, because BPMS is not set
```

3.2.11 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)

    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
```

(continues on next page)

(continued from previous page)

```
    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.
```

3.2.12 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:

[15]:
```
%%bash

# 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
HERE=$(pwd);cd ../..
BPMS=${BPMS-$(pwd)};cd $HERE

bin=$BPMS/src

# put $bin on the front of PATH
export PATH="$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
```

(continues on next page)

(continued from previous page)

```
    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.
```

3.2.13 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.

3.2.14 Update LD_LIBRARY_PATH, DYLD_LIBRARY_PATH

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

[16]:
```
%%bash

# 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
HERE=$(pwd);cd ../..
BPMS=${BPMS-$(pwd)};cd $HERE

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
```

[17]:
```
%%bash

# 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"


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

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"


    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.
```

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

Before proceeding, it is useful to see how to determine which operatinbg 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 apparant.

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 variabler `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.

[18]:
```
%%bash

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

```
I am macOS: Darwin
```

3.2.16 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.

[19]:
```
%%bash

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

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

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
0000000000047500 T _Add_2D
00000000000477c0 T _Affine_transform
0000000000049090 T _B_allocate
00000000000477f0 T _Backwards_affine_transform
00000000000470b0 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.

3.3 Important environment variables for `librat`

3.3.1 `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:

[20]:
```
%%bash

# change directory from docs/source up to root
cd ../..
```

(continues on next page)

(continued from previous page)

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

cat <<EOF > $BPMS/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
!}
!}
EOF
```

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

[21]:
```
%%bash

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

cat $BPMS/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
!}
!}
```

```
EXERCISE

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

3.3.2 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. plants.matlib, all materials defined in a material library e.g. refl/white.dat
`ARARAT_OBJECT`	(extended) wavefront object files e.g. first.obj
`DIRECT_ILLUMINATION`	spectral files for direct illumination: those defined in -RATdirect command line option
`RSRLIB`	sensor waveband files: those defined in -RATsensor_wavebands command line option
`BPMS_FILES`	Not used

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 `mtllib.h`)

[22]:
```
%%bash

# change directory from docs/source up to root
cd ../..
BPMS=${BPMS-$(pwd)}
# create the init.sh file we want
INIT=$BPMS/test/test_examples/init.sh

cat <<EOF > $INIT
#!/bin/bash
#
# preamble
```
(continues on next page)

(continued from previous page)

```
#
export BPMS=$BPMS

# set shell variables lib, bin, verbose
# with defaults in case not set
lib=\${lib-"\$BPMS/src"}
bin=\${bin-"\$BPMS/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 BPMSROOT=\${BPMSROOT-"\$BPMS/test/test_examples"}
export MATLIB=\$BPMSROOT
export RSRLIB=\$BPMSROOT
export ARARAT_OBJECT=\$BPMSROOT
export DIRECT_ILLUMINATION=\$BPMSROOT
export BPMS_FILES=\$BPMSROOT
echo $BPMS
if [ "\$(which start)" == "\${bin}/start" ]
then
  if [ "\$VERBOSE" == 1 ]; then
      echo "start found ok"
  fi
else
  # we should create them
  make clean all
fi
EOF
# set executable mode
chmod +x $INIT
# test run
export VERBOSE=1
$INIT
```

```
/Users/plewis/librat
start found ok
```

3.4 Summary

In this chapter, we have covered a range of basic unix/linux and `bash` commands, so you should be able to navigate you way around a unix file system, and find your way back safely. Being familiar with these tools takes somne 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 thats 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 reccomend you *do* spend some time on *unix* … you'll develop skills that willlast you a lifetime! ;-)

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

```
[ ]:
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search