Langmuir Manual

Adam Gagorik

November 11, 2013

Contents

1 Introduction

Langmuir is a program written in c++ for the simulation of charge transport in organic semiconductors. The simulation works by using Monte Carlo to predict the movement of charge carriers on a grid. Each site on the grid represents an organic semiconducting molecule, and charge carriers, such as electrons or holes, can occupy the sites. During each step of the simulation, all charge carriers propose hopping moves to adjacent sites, and use the Metropolis criterion to decide the acceptance rate. Langmuir monitors the flux of charge carriers in and out of the system, allowing the calculation of quantities such as current and mobility.

2 Usage

2.1 Langmuir

Langmuir is generally run inside a terminal.

```
adam@work: langmuir input.inp
```

Note that the langmuir command must be on your path. There is only one argument, the full path to the input file. However, note that Langmuir will write output files to the directory it is called from. The format of the input file is discussed in section 3. Often you will run Langmuir on a cluster. Details on how to run Langmuir on a cluster, as well as sample batch scripts are included in section 5.

2.2 LangmuirView

LangmuirView is used to watch simulations graphically in real time with a GUI. LangmuirView can be run from a terminal.

```
adam@work: langmuirView input.inp
```

Note that the langmuirView command must be on your path. Unlike the langmuir command, the input file argument is optional. If no input file is given, then LangmuirView will open a file dialogue for you to choose the location of an input file. An example of LangmuirView is shown in Figure 1.

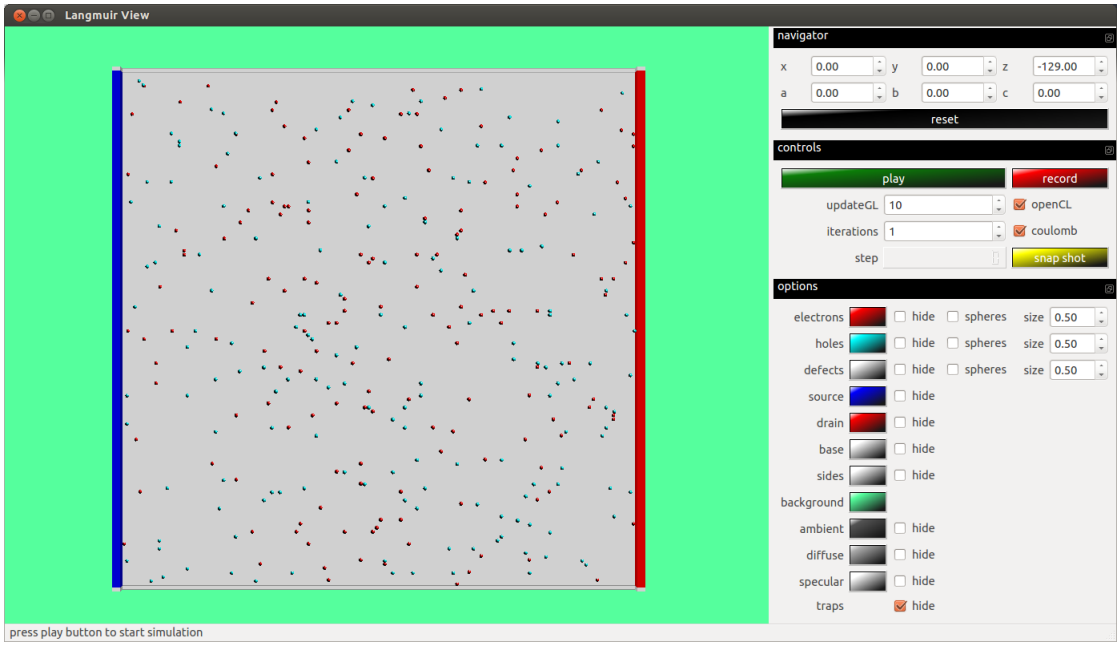

Figure 1: LangmuirView example.

LangmuirView is a tool to be used to aid in understanding and communication to others, for example, to make movies or screen shots. To perform actual simulations you should use Langmuir from the command line, as discussed in section 2.1. This is because LangmuirView is limited by the size of the simulation and the production of output data. Do not attempt to use LangmuirView with large systems that produce a large amount of output data.

3 Input Files

This section contains information on the format of Langmuir input files. Input files are just text files that you can edit with any text editor. However, be warned that input files can be very long if they contain information on traps. In this case, it is best to use a text editor capable of dealing with very large files. Input files can also be manipulated using LangmuirPython, as discussed in section 6. In Langmuir, input files and checkpoint files are the same thing. Periodically, a running simulation will save a checkpoint file. You can use this checkpoint file to extend the simulation or change its parameters. The `#` symbol serves as a comment inside the input file. Any text after the `#` symbol is ignored. The input file is divided into sections. Sections always start with a header. The valid section headers are shown below.

- `[Electrons]`
- `[Holes]`
- `[Defects]`
- `[Traps]`
- `[TrapPotentials]`
- `[FluxState]`
- `[RandomState]`
- `[Parameters]`

The only required section is the `[Parameters]` section, and it must be the last section in the input file. Other sections do not have to be in any particular order. Example input files are found in section 3.7.

3.1 Site IDs

The position of an agent in the grid can be thought of as a 3-tuple of integers (x_i, y_i, z_i). This 3-tuple can be hashed into a single number called the site-id, s_i. The dimensions of the grid are L_x, L_y, and L_z. Note that L_x, L_y, and L_z are the `grid.x`, `grid.y`, and `grid.z` parameters discussed in section 3.6. The following equations hold for site-ids, where all quantities are integers, and integer division applies.

$$0 \leq x_i < L_x \qquad x_i = s_i \% L_x \tag{1}$$
$$0 \leq y_i < L_y \qquad y_i = s_i / L_x - (s_i/(L_x L_y)) L_y \tag{2}$$
$$0 \leq z_i < L_z \qquad z_i = s_i/(L_x L_y) \tag{3}$$
$$0 \leq s_i < L_x L_y L_z \qquad s_i = L_x(y_i + z_i L_y) + x_i \tag{4}$$

3.2 Agents

Electrons, holes, defects, and traps all follow the same format. Note that these sections are for providing information on electrons, etc. already present in the system before the simulation starts. This is typically the case when extending a run, or placing traps at well defined locations. You may leave these sections out. Langmuir has the ability to place traps, defects, and carriers randomly if desired (see section 3.6).

It is very important that the parameters `electron.percentage`, `hole.percentage`, `defect.percentage`, and `trap.percentage` are consistent with these sections. For example, while the number of electrons in the `Electrons` section can be less than the maximum number of electrons allowed by `electron.percentage`, it can not exceed the max. If there is a problem, langmuir will raise an error.

The first line is the section header written in square brakets. The next line is always the number of elements to be read by langmuir. For example, for the electrons, the second line is the number of electrons. The remaining lines are the site-ids for electrons, holes, defects, and traps. Site-id's are discussed in section 3.1.

```
[Electrons]     # section header
2               # number of electrons
100             # site-id of electron 1
200             # site-id of electron 2
[Holes]         # section header
0               # number of holes
[Defects]       # section header
0               # number of defects
[Traps]         # section header
0               # number of traps
```

3.3 Trap Potentials

The `[TrapPotentials]` section is very similar in structure to the `[Trap]` section. The only difference is that instead of site-ids, one lists the trap potentials in units of eV. If present, the `[TrapPotentials]` section must be the same size as the `[Traps]` section. Note that if all traps have the same value, then this section can be omitted. In this case, the value used for trap potential is taken from the `trap.potential` parameter (see section 3.6).

```
[TrapPotentials]    # section header
2                   # number of traps
0.50                # trap potential of trap 1
0.50                # trap potential of trap 2
```

3.4 Flux State

The `[FluxState]` is a list of 20 integers detailing the number of attempts and successes made by a flux agent. Examples of flux agents are the sources and drains. There are 10 different flux agents in Langmuir. There are 2 source agents and 2 drain agents at $x_i = 0$ and $x_i = L_x - 1$, making a total of 8. The remaining 2 are an exciton source, and a recombination drain. You probably never have to edit this section.

```
# The codes should hopefully be easy to figure out...
# ESLA = Electron Source Left Attempt
# XDS  = Recombination Drain Success
# etc...
[FluxState]     # section header
20              # number of flux agents
0               # ESLA
0               # ESLS
0               # ESRA
0               # ESRS
0               # HSLA
0               # HSLS
0               # HSRA
0               # HSRS
0               # XSA
0               # XSS
0               # EDLA
0               # EDLS
0               # EDRA
0               # EDRS
0               # HDLA
0               # HDLS
0               # HDRA
0               # HDRS
0               # XDA
0               # XDS
```

3.5 Random State

The `[RandomState]` is a very long list of integers that describe the exact state of the random number generator. Due to limitations of the combination of boost, stdlib, and qt, it must be on one line. You should never have to edit this section, other than deleting it.

```
[RandomState] # section header
1371835351 1524755492 3319441753 617340572... # list of numbers
```

3.6 Parameters

The parameters section is a list of `key=value` pairs that alter the behavior of the simulation. This section will often be the only section in an input file. It must be present, and it must be the last section in the text file. Below is a list of parameters and their descriptions.

keyword	type	default	description
`simulation.type`	**string**	transistor	solarcell or transistor - changes the behavior of the sources and drains.
`random.seed`	**int**	0	if 0, then use the current time, else seed the random number generator.

keyword	type	default	description
`grid.z`	**int**	1	The height of the device, or number of sites in the z-direction (layers).
`grid.y`	**int**	128	The width of device, or number of sites in the y-direction.
`grid.x`	**int**	128	The length of device, or number of sites in the x-direction (source to drain).
`hopping.range`	**int**	1	The number of adjacent sites to consider as neighbors when hopping.

keyword	type	default	description
`iterations.real`	**int**	1000	The number of simulation steps, including equilibration. It is up to you to remove the equilibration steps from the output.
`iterations.print`	**int**	10	The number of steps between printing output
`current.step`	**int**	0	The starting step of the simulation. Needed for checkpoint files.

keyword	type	default	description
`output.is.on`	**bool**	True	Create output files. It is useful to turn off the output when using LangmuirView.
`output.precision`	**int**	15	The number of digits to print for numbers in various output files.
`output.width`	**int**	23	The width of columns in the output file.
`output.stub`	**string**	out	The naming scheme of output files. For example, if stub is "out", then the output files are "out.dat", "out.chk", etc.
`output.ids.on.delete`	**bool**	False	Save carrier lifetime and path length to a file when the carrier reaches a drain. This can make very large files.
`output.ids.on.encounter`	**bool**	False	Save carrier lifetime and path length to a file when the carrier forms an exciton. This can make very large files.
`output.coulomb`	**int**	0	Output the Coulomb energy of the entire grid every `iterations.print` × `output.coulomb` steps. If `output.coulomb` < 0, then save the Coulomb energy when then the simulation finishes. This requires OpenCL. If the grid is too large it may not work if the GPU is too small.
`output.step.chk`	**int**	1	Output checkpoint files every `iterations.print` × `output.step.chk`. When there is a large number of trap sites, writing checkpoint files will slow the simulation down. Use this parameter to make sure checkpoint files are written far less often than the `iterations.print` value.
`output.chk.trap.potential`	**bool**	False	Suppress the writing of trap potentials to the checkpoint file. It is redundant and slow to output trap potentials when they are all the same value.
`output.potential`	**bool**	False	Output the potential of the entire grid at the start of the simulation. This grid potential does not include the trap potential or the Coulomb interactions.

keyword	type	default	description
`output.xyz`	**int**	0	Output carrier locations to an xyz file every `iterations.print` × `output.xyz`. This file will be large. This file will not open easily in VMD without the use of a VMD extension because the number of particles can change. There is a `vmd.init` file in the Langmuir source directory to help with opening this file.
`output.xyz.e`	**bool**	True	Output the electrons to the xyz file.
`output.xyz.h`	**bool**	True	Output the holes to the xyz file.
`output.xyz.d`	**bool**	True	Output the defects to the xyz file.
`output.xyz.t`	**bool**	True	Output the traps to the xyz file. When there are tons of traps, the size of the xyz file can become too large to handle. You should suppress the output of traps to the xyz file in this case.
`output.xyz.mode`	**int**	0	When 0, the number of particles between frames in the xyz file can vary. If 1, the number of particles is kept constant using "phantom particles"

keyword	type	default	description
`image.traps`	**bool**	False	Save a png of the traps at the start. Assumes `grid.z` = 1.
`image.defects`	**bool**	False	Save a png of the defects at the start. Assumes `grid.z` = 1.
`image.carriers`	**int**	0	Save a png of the carriers every `iterations.print` × `image.carriers`. If `image.carriers` < 0, then save the png when then the simulation finishes. Assumes `grid.z` = 1.

keyword	type	default	description
`electron.percentage`	**float**	0.01	Sets the maximum number of allowed electrons to be the volume of the grid times this percentage. Between 0 and 1.
`hole.percentage`	**float**	0.0	Sets the maximum number of allowed holes to be the volume of the grid times this percentage. Between 0 and 1.
`seed.charges`	**float**	0.0	The fraction of the maximum electrons/holes to place randomly at the beginning of the simulation. Between 0 and 1. This helps with equilibration in transistors. Have not tested this in solar cells.

keyword	type	default	description
`defect.percentage`	**float**	0.0	Sets the maximum number of defects to be the volume of the grid times this percentage. Between 0 and 1. Defects are placed randomly at the start.
`defects.charge`	**int**	0	The charge of defects. If 0, then defects are not included in Coulomb calculations.

keyword	type	default	description
`trap.percentage`	**float**	0.0	Sets the maximum number of traps to be the volume of the grid times this percentage. Between 0 and 1. Traps are placed randomly.
`seed.percentage`	**float**	1.0	The fraction of the traps to place as seeds. Remaining traps are grown around these seeds. Between 0 and 1.
`trap.potential`	**float**	0.1	The trap energy to use for randomly placed traps.
`gaussian.stdev`	**float**	0.0	Standard deviations of random noise to be added to randomly placed traps.

keyword	type	default	description
`voltage.right`	**float**	0.0	The voltage of the drain electrode.
`voltage.left`	**float**	0.0	The voltage of the source electrode. Keep this zero and alter `voltage.right`.
`exciton.binding`	**float**	0.0	The energy of interaction when a hole and electron are on the same site.
`slope.z`	**float**	0.0	The voltage change along the z direction due to a gate electrode.
`coulomb.carriers`	**bool**	False	Turn on Coulomb interactions.
`coulomb.gaussian.sigma`	**float**	0.0	The standard deviation of smeared out Gaussian charges. If 0, then point charges are used. Assumes `grid.z` > 1.
`temperature.kelvin`	**float**	300.0	The temperature used in the Boltzmann factor.

keyword	type	default	description
`source.rate`	**float**	0.9	Default probability to inject charges. Between 0 and 1.
`e.source.l.rate`	**float**	-1.0	Injection rate of electrons from the left. Overrides `source.rate`. Ignored if < 0.
`e.source.r.rate`	**float**	-1.0	Injection rate of electrons from the right. Overrides `source.rate`. Ignored if < 0.
`h.source.l.rate`	**float**	-1.0	Injection rate of holes from the left. Overrides `source.rate`. Ignored if < 0.
`h.source.r.rate`	**float**	-1.0	Injection rate of holes from the right. Overrides `source.rate`. Ignored if < 0.
`generation.rate`	**float**	0.001	Injection rate of excitons. Overrides `source.rate`. Ignored if < 0.
`balance.charges`	**bool**	False	Try to keep the number of electrons and holes equal. Not physical.
`source.metropolis`	**bool**	False	Override source injection probability with a metropolis criterion involving site energy.
`source.coulomb`	**bool**	False	Include coulomb interactions with image charges in the metropolis criterion.
`source.scale.area`	**float**	65536.0	Scale the generation rate by dividing by this value and multiplying by the xy-area of the system.

keyword	type	default	description
`drain.rate`	**float**	0.9	Default probability to accept charges. Between 0 and 1.
`e.drain.l.rate`	**float**	-1.0	Acceptance rate of electrons on the left. Overrides `drain.rate`. Ignored if < 0.
`e.drain.r.rate`	**float**	-1.0	Acceptance rate of electrons on the right. Overrides `drain.rate`. Ignored if < 0.
`h.drain.l.rate`	**float**	-1.0	Acceptance rate of holes on the left. Overrides `drain.rate`. Ignored if < 0.
`h.drain.r.rate`	**float**	-1.0	Acceptance rate of holes on the right. Overrides `drain.rate`. Ignored if < 0.
`recombination.rate`	**float**	0.0	Probability to recombine excitons. Note - it is not really a rate like the others because the number of excitons in the system is hard to predict.
`recombination.range`	**int**	0	Number of adjacent sites to consider during recombination.

keyword	type	default	description
`use.opencl`	**bool**	False	Use OpenCL for Coulomb calculations.
`work.x`	**int**	4	The number of x-threads in a 3D work group. Only used for `output.coulomb`. The total size of a work group is $W =$ `work.x` $\times$ `work.y` $\times$ `work.z`. The total size of the grid is $G =$ `grid.x` $\times$ `grid.y` $\times$ `grid.z`. The total number of threads used by the 3D kernel is $T = G \times W$. The 3D kernel will fail if you exceed the limitations of the GPU. This could be fixed by dividing the grid into sections and using multiple GPU's or multiple calls to one GPU. The max W allowed on GTX460 is 1024. The max T allowed on GTX460 is $1024 \times 1024 \times 64$. Therefore, the maximum number of grid sites that could be handled is $65536 = 256^2$.
`work.y`	**int**	4	The number of y-threads in a 3D work group. See `work.x` for more info.
`work.z`	**int**	4	The number of z-threads in a 3D work group. See `work.x` for more info.
`work.size`	**int**	256	The number of threads W in a 1D work group. The total number of threads used during a coulomb calculation is $T = N \times W$, where N is the total number of charges. It is unlikely you will exceed the total number of allowed threads on the GPU. For magical reasons, this parameter seems to be optimal at 256. The maximum value of W on a GTX460 is 1024. The maximum number of threads on a GTX460 is $1024 \times 1024 \times 64$. Therefore, the maximum number of charges allowed when $W = 256$ is $N = 262144$.
`opencl.threshold`	**int**	256	The number of charges that must be present before turning on OpenCL. OpenCL will be slower than the CPU for small numbers of charges.
`opencl.device.id`	**int**	0	The id of the GPU to use when more than one is present. Currently this parameter is ignored. The GPU chosen defaults to 0, unless a PBS_GPUFILE is found, in which case the GPU used is chosen by PBS. The id of the GPU chosen is saved to this variable.
`max.threads`	**int**	-1	The max number of CPU threads allowed. If < 0, then use the number of threads recommended by QtConcurrent. However, if < 0 and a PBS_NODEFILE is found, then the number of threads is chosen by PBS. LangmuirView currently ignores this parameter and uses the number of threads recommended by QtConcurrent.

3.7 Examples

This section contains sample input files.

3.7.1 Transistor

```
[Parameters]
simulation.type         = transistor

grid.x                  = 1024
grid.y                  = 256
grid.z                  = 1

iterations.real         = 500000
iterations.print        = 1000

electron.percentage     = 0.10
seed.charges            = 1.00      # speed up equilibration

voltage.right           = 5.00
voltage.left            = 0.00

coulomb.carriers        = true
use.opencl              = true
```

3.7.2 Solar Cell

```
[Parameters]
simulation.type         = solarcell

grid.x                  = 256
grid.y                  = 256
grid.z                  = 1

iterations.real         = 20000000  # much longer than transistor
iterations.print        = 1000

electron.percentage     = 0.10
hole.percentage         = 0.10

trap.percentage         = 0.50
trap.potential          = 0.50
seed.percentage         = 0.10

voltage.right           = 9.00
voltage.left            = 0.00

source.rate             = 1e-3
recombination.rate      = 1e-4

coulomb.carriers        = true
use.opencl              = true
```

3.7.3 Scan

Set a variable equal to a list of values. This works for any variable. See section 6.

```
[Parameters]
...
voltage.right = [-2.0, -1.5, -1.0, -0.8, -0.6, -0.4, -0.2, 0.0]
...
```

To generate input files.

```
adam@work: python scan.py --real 500000 --print 1000 --mode gen
```

To run Langmuir in real time.

```
adam@work: python scan.py --real 500000 --print 1000 --mode scan
```

3.7.4 Traps

```
[Parameters]
...
trap.percentage          = 0.50 # 50-percent traps
seed.percentage          = 0.10 # 10-percent seeds
trap.potential           = 0.05 #
...
```

3.7.5 Defects

```
[Parameters]
...
defect.percentage        = 0.50 # 50-percent defects
defect.charge            = 0    # neutral defects
...
```

3.7.6 Coulomb

```
[Parameters]
...
coulomb.carriers         = true # turn on coulomb interations
use.opencl               = true # use GPU
output.coulomb           = 10   # output energy every 10 iterations.print
...
```

To calculate coulomb energy of a checkpoint file you can use the python script coulomb.py. If the system is too big for the GPU you have to use coulomb.py. Or you can run langmuir again.

```
[Parameters]
...
iterations.real          =  0 # do not simulate anything
output.coulomb           = -1 # output energy at the end
...
```

4 Output Files

4.1 Standard Output Files

Langmuir will always output the following files. The name of the output file is controlled with the `output.stub` parameter. The generation of output can be turned off with the `output.is.on` parameter.

- out.dat
- out.chk
- out.parm

If Langmuir finishes successfully, then additional files may appear.

- out.time

Langmuir also writes information to the screen of what it is doing. On a cluster this information is often captured by the stderr file.

4.1.1 out.chk

This is the checkpoint file. The format of this file is discussed in section 3. Checkpoint files allow one to extend a simulation or change its parameters. They can easily be manipulated in a text editor or using LangmuirPython (see section 6). Checkpoint files are output every `iterations.print` × `output.step.chk` steps.

It may be useful to structure your simulation directories to reflect the idea of "parts" of a simulation. The LangmuirPython script combine.py allows you to combine the output of the various parts. In the example below, the checkpoint file from `part.0` is used as the input file of `part.1`.

```
simulation/
    part.0/
        out.dat
        out.chk
    part.1/
        out.dat
        out.chk
```

4.1.2 out.parm

This is a condensed input-like file that contains only the `[Parameters]` section (see section 3). The file is produced only once, at the start of a simulation. It can be faster to parse this file in scripts than a full blown checkpoint file with many traps.

4.1.3 out.dat

This is the main output file of Langmuir. Every `iterations.print`, statistics from the simulation are written to this file. The output columns are named at the top of the file.

```
simulation:time  # step (ps)
eSourceL:success # success count
eSourceL:attempt # attempt count
...
electron:count   # number of electrons
hole:count       # number of holes
real:time        # clock time (ms)
```

There is a success and attempt column for each of the 10 FluxAgents.

Various quantities can be computed from these statistics. These quantities are not computed by Langmuir, it is left up to the simulator in post-analysis stage. To calculate the probability of a FluxAgent, divide the successes by the attempts.

$$P = \frac{S}{A} \times 100\% \tag{5}$$

To calculate the rate of a FluxAgent, divide the success by the step.

$$R = \frac{S}{T} \quad [\mathrm{ps}^{-1}] \tag{6}$$

To calculate the current of a FluxAgent, multiply the rate by e and convert to nA.

$$I = e \times R \quad [\mathrm{C\,ps}^{-1}] \tag{7}$$

Current is calculated from the success rate of drains. When simulations contain multiple drains for different carrier types, care must be taken when calculating the current. S_{eDL} is the success rate of the electron drain on the left. S_{eDR} is the success rate of the electron drain on the right. S_{hDL} is the success rate of the hole drain on the left. S_{hDR} is the success rate of the hole drain on the right. The current is defined to be positive for the movement of electrons to right.

$$I = \frac{(S_{eDR} - S_{eDL}) + (S_{hDL} - S_{hDR})}{T} \times e \tag{8}$$

To account for equilibration, one must subtract the line containing the start of "production steps" (chosen by the simulator) from the ending line of the simulation. This process was removed from Langmuir to simplify the generation of checkpoint files. For example, consider a system where the ElectronSourceAgent has a success rate of 100 at 100000 steps into the simulation and a success rate of 500 at 500000 steps into the simulation, the end of the simulation. To account for 100000 steps of equilibration, the final success rate will be $500 - 100 = 400$. This operation must be applied for every column of the output file. The LangmuirPython script gather.py performs this operation on the files produced by combine.py.

4.1.4 out.time

This file contains the total time taken to perform the simulation in various units. The file is only written at the end of a simulation. If Langmuir fails to finish, timing information is also present in `out.dat`.

4.2 Additional Output Files

If Langmuir is run on a cluster, various cluster related files may also appear.

- stdout files
- stderr files

Images of the grid can be produced using various input parameters (see section 3.6).

- out-traps.png
- out-defects.png
- out-%step-electrons.png
- out-%step-holes.png
- out-%step-carriers.png
- out-%step-all.png

Information on energy and potential may be produced (see section 3.6).

- out.grid
- out-%step.coulomb

Information on carrier lifetime and path length may be produced (see section 3.6).

- out-carriers.dat
- out-excitons.dat

Finally, the trajectory of carriers can be produced.

- out.xyz

4.2.1 *.png

These are just crappy png files produced using Qt. There are much better ways of making pictures. For example, you can use LangmuirPython to use information in a checkpoint file to draw a picture with matplotlib. Also, the LangmuirPython chk2vtk.py will produce vtk files that can be opened in various programs, such as paraview or mayavi.

4.2.2 out.coulomb

This file contains the following columns.

```
s # site-id
x # x-value
y # y-value
z # z-value
v # potential
```

It is useful to produce contour maps of this data in python with matplotlib. If you really want to see things in 3D you can make contour iso-surfaces in other programs, such as paraview.

```
from scipy.iterpolate import griddata
import matplotlib.pyplot as plt
import numpy as np

data = np.genfromtxt('out.coulomb', names=True)
x, y, z, v = data['x'], data['y'], data['z'], data['v']

# data must be put on a mesh to plot it
mesh_x, mesh_y, mesh_z = np.mgrid[0:64:1, 0:64:1, 0:1:1]
mesh_v = griddata((x, y, z), v, (mesh_x, mesh_y, mesh_z))

# we can only view things in 2D
mesh_x = mesh_x[:,:,0]
mesh_y = mesh_y[:,:,0]
mesh_v = mesh_v[:,:,0]

plt.contour(mesh_x, mesh_y, mesh_v, 32)
plt.show()
```

4.2.3 out.grid

This file is very similar to `out.coulomb`.

```
s # site-id
x # x-value
y # y-value
z # z-value
e # electron grid potential
h # hole grid potential
```

4.2.4 out-carriers.dat

```
s          # site-id
x          # x-value
y          # y-value
z          # z-value
agent      # electron or hole
address    # unique ID
lifetime   # ps
pathlength # nm
step       # ps
```

4.2.5 out-excitons.dat

```
s1          # site-id
x1          # x-value
y1          # y-value
z1          # z-value
agent1      # electron or hole
address1    # unique ID
lifetime1   # ps
pathlength1 # nm
s2          # site-id
x2          # x-value
y2          # y-value
z2          # z-value
agent2      # electron or hole
address2    # unique ID
lifetime2   # ps
pathlength2 # nm
step        # ps
recombined  # boolean
```

4.2.6 out.xyz

```
agent      # E, H, D, T
x          # x-value
y          # y-value
z          # z-value
```

5 Batch Files

Details on how to run Langmuir on a cluster are discussed in this section.

5.1 Cluster Commands

```
# show the jobs
qstat
```

```
# submit a batch script
qsub -N jobname run.batch
```

5.2 Hutchison

On `hutchison.chem.pitt.edu`, one can use the submitLangmuir command.

```
submitLangmuir jobname job.inp
```

Or you can use a batch script such as the one below.

```
#!/bin/sh
#£ -S /bin/bash
#£ -pe mpi 4
#£ -R y
#£ -cwd

INPUT=$1
WORK=$2
BIN=/usr/local/bin
EXE=langmuir
SCRATCH=/scratch/${USER}/${JOB_ID}

if [ -d /scratch/${USER} ]; then
touch /scratch/${USER}
else
mkdir /scratch/${USER}
fi

mkdir ${SCRATCH}

if [ -f ${INPUT} ]; then
cp ${INPUT} ${SCRATCH}
else
exit
fi

cd ${SCRATCH}
${BIN}/${EXE} ${INPUT}

gzip ${SCRATCH}/* -rv
cp -r ${SCRATCH}/* ${WORK}
```

5.3 Frank

On `frank.sam.pitt.edu`, use the following batch script. Make sure to set the path to langmuir in the `BIN` variable.

```
#!/bin/bash
#PBS -q gpu
#PBS -l nodes=1:ppn=3:gpus=1
#PBS -l mem=4gb
#PBS -l walltime=72:00:00
BIN=/home/ghutchison/agg7/bin
EXE=langmuir
NME=sim.inp
module load cuda/4.2
module load boost
module load qt
cd $PBS_O_WORKDIR
$BIN/$EXE $NME
```

5.4 Scan

Sometimes, instead of running a single langmuir simulation, one may want to scan over many values of a variable. In the LangmuirPython module (see section 6), there is a script called scan.py. This script (scan.py) lets python guide the scanning of a working variable. For example, one can scan `voltage.right` to create an IV curve. Instead of setting a variable in the input file to a single value, set it to a list of values.

```
voltage.right = [-1.0, -2.0, -3.0]
```

Actually, scan.py can parse any valid python statement, including numpy.

```
voltage.right = [float(i) for i in range(0, 100, 10)]
```

```
voltage.right = np.linspace(0, 100, 10)
```

Multiple working variables are not supported. You must use an altered batch script to run scan.py. Make sure the script correctly sets the `$PYTHONPATH` to point to LangmuirPython. Below is an example batch script for `frank.sam.pitt.edu`.

```
#!/bin/bash
#PBS -q gpu
#PBS -l nodes=1:ppn=4:gpus=1
#PBS -l mem=4gb
#PBS -l walltime=36:00:00

BIN=/home/ghutchison/agg7/bin
EXE=scan.py
NME=sim.inp

module load cuda/4.2
module load python
module load boost
module load qt

export PYTHONPATH=$PYTHONPATH:/home/ghutchison/agg7/LangmuirPython

cd $PBS_O_WORKDIR
python $BIN/$EXE --real 7500000 --print 1000 --fmt '%+.1f' --mode scan
```

6 Langmuir Python

There is a python project called LangmuirPython, that aids in the construction of input files, the running of simulations, and the analysis of output files. The use of LangmuirPython in not required. There is documentation in the code. A pdf and/or html webpage of the documentation can be generated with Sphinx. Simply navigate to the LangmuirPython doc directory, and use the Makefile. Sphinx must be installed.

6.1 Installation

LangmuirPython uses a number of python modules that may not be included in standard python distributions. You should make an effort to install these python modules, for example, using pip. If modules are missing, an effort has been made to disable certain features of the langmuir module.

- numpy
- scipy
- matplotlib
- pandas
- quantities
- vtk (optional)

To install LangmuirPython, you must do one of the following.

1. Put the path to LangmuirPython on your $PYTHONPATH.

```
# set £PYTHONPATH inside batch scripts or bashrc
export PYTHONPATH=$PYTHONPATH:/path/to/LangmuirPython
```

2. Use the setup.py script in the LangmuirPython directory.

```
adam@work: cd /path/to/LangmuirPython

# to install locally
adam@work: python setup.py install --user

# to install globally (need root password)
adam@work: python setup.py install
```

6.2 Usage

To use the langmuir module in python you must import it.

```
>>> import langmuir as lm
```

The checkpoint submodule is very useful for manipulating inputs file from python.

```
>>> import langmuir as lm
>>> chk = lm.checkpoint.load('out.chk')
>>> chk.electrons = [] # delete electrons
>>> chk['iterations.real'] += 100000
>>> chk.save('sim.inp')
```

The usage of most scripts can be determined by using the -h or –help command line flag.

```
adam@work: python /path/to/LangmuirPython/utils/scan.py --help
```

Some scripts should be pointed out.

- /utils/scan.py

```
# --mode scan actually runs langmuir
# the output of one simulation serves as the input of the next
# sim.inp has a working variable set to a list of values
# example: voltage.right = [1, 2, 3, 4, 5]
adam@work: python scan.py --real 500000 --mode scan sim.inp

# --mode gen generates simulations input
adam@work: python scan.py --real 500000 --mode gen sim.inp
```

- /analyze/combine.py

```
# combines the output of "parts"
adam@work: python combine.py -r
```

- /analyze/gather.py

```
# gathers output from simulations
# use this inside a "run" directory
adam@work: python gather.py -r --equil 10000
```

A lot of the scripts rely on you structuring your simulations.

```
system/
    run.0/
        voltage.right_+0.0/
            part.0/
                out.dat
                out.chk
            part.1/
                ...
            ...
        voltage.right_+0.1/
        ...
    run.1/
    run.2/
    ...
```