# LangmuirPython Documentation

***Release 0.0.1***

**Adam Gagorik**

June 23, 2013

# CONTENTS

**1 Getting Started** **3**

**2 Module List** **5**

2.1 common . . . 5
2.2 regex . . . 8
2.3 find . . . 8
2.4 checkpoint . . . 9
2.5 parameters . . . 11
2.6 database . . . 13
2.7 datfile . . . 14
2.8 surface . . . 14
2.9 grid . . . 20
2.10 analyze . . . 24
2.11 ivline . . . 25
2.12 plot . . . 27
2.13 fit . . . 28
2.14 Indices and tables . . . 32

**Python Module Index** **33**

**Index** **35**

LangmuirPython is a set of python modules and python scripts that aid in the use of the Langmuir charge transport simulation code.

CHAPTER

ONE

# GETTING STARTED

LangmuirPython depends on the following python modules, which you should install:

- numpy
- scipy
- pandas
- quantities
- matplotlib

CHAPTER

TWO

# MODULE LIST

## 2.1 common

**class** `common.DictDiffer`

Set of static functions used to compare dictionary keys and values.

```
>>> d1 = dict(A=1, B=2, C=3)
>>> d2 = dict(B=2, C=4, D=5)
>>> print langmuir.common.DictDiff.addedKeys(d1, d2)
... set(['A'])
```

**classmethod** `addedKeys`(*d1*, *d2*)

Calculates keys in d1 not in d2.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** set

**classmethod** `addedValues`(*d1*, *d2*)

Calculates keys in d1 not in d2 with values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** dict

**classmethod** `changedKeys`(*d1*, *d2*)

Calculates keys in both that have different values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** set

**classmethod** `changedValues`(*d1*, *d2*, *split=False*)

Calculates key : value pairs in both that have different values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2
- **split** (*bool*) – split result up into seperate dictionaries

**Returns** (dict, [dict])

**static `keySets`** (*d1*, *d2*)
Turns keys into sets and calculates their intersection / union.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** (set, set, set, set)

**classmethod `removedKeys`** (*d1*, *d2*)
Calculates keys in d2 not in d1.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** set

**classmethod `removedValues`** (*d1*, *d2*)
Calculates keys in d2 not in d1 with values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** dict

**classmethod `unchangedKeys`** (*d1*, *d2*)
Calculates keys in both that have the same values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2

**Returns** set

**classmethod `unchangedValues`** (*d1*, *d2*, *split=False*)
Calculates key : value pairs in both that the same values.

**Parameters**

- **d1** (*dict*) – dictionary 1
- **d2** (*dict*) – dictionary 2
- **split** (*bool*) – split result up into seperate dictionaries

**Returns** (dict, [dict])

`common.`**`command_script`** (*paths*, *name=None*, *stub='run'*, *command=None*)
Create a handy bash script that loops over the paths.

**Parameters**

- **paths** (*list*) – list of paths
- **name** (*str*) – name of script
- **stub** (*str*) – jobname stub
- **command** (*str*) – command to insert at each directory

`common.`**`evaluate`**(*obj*)
Wrapper around eval.

**Parameters** **obj** (*str, object*) – arbitrary python object or string to evaluate

`common.`**`grep`**(*fname*, *regex*)
Run grep on a file.

**Parameters**

- **fname** (*str*) – name of file
- **regex** (*str*) – pattern to match

`common.`**`parameter`**(*fname*, *key*)
Get parameter from keypoint file using grep.

**Parameters**

- **fname** (*str*) – name of file
- **key** (*str*) – key to match

`common.`**`tail`**(*fname*, *n=1*)
Run tail on a file.

**Parameters**

- **fname** (*str*) – name of file
- **n** (*int*) – number of lines

`common.`**`timestamp`**(*fname*)
Get the last time a file was modified.

**Parameters** **fname** (*str*) – name of file

`common.`**`zgrep`**(*fname*, *regex*)
Run grep on a gzipped file.

**Parameters**

- **fname** (*str*) – name of file
- **regex** (*str*) – pattern to match

`common.`**`zhandle`**(*name*, *mode*)
Get a file handle, checking to see if the file is gzipped first.

**Parameters**

- **name** (*str*) – filename
- **mode** (*str*) – open mode

`common.`**`ztail`**(*fname*, *n=1*)
Run tail on a gzipped file.

**Parameters**

- **fname** (*str*) – name of file

- **n** (*int*) – number of lines

## 2.2 regex

`regex.`**`fix_boolean`**(*string*)
return string with true/false in correct python format

`regex.`**`fix_name`**(*string*)
Turn non alpha numeric characters into underscores.

returns: the fixed string

`regex.`**`number`**(*string*, *index=1*, *pytype=<type 'float'>*)
get numbers in a string at index and convert them to **type_**

`regex.`**`numbers`**(*string*, *type_=<type 'float'>*)
find all numbers in a string and convert them to **type_**

`regex.`**`part`**(*string*)
extract the part as int from a string

`regex.`**`run`**(*string*)
extract the run as int from a string

`regex.`**`sim`**(*string*)
extract the sim as int from a string

`regex.`**`strip_comments`**(*string*)
return string will all comments stripped

`regex.`**`voltage`**(*string*)
extract the votlage as float from a string

## 2.3 find

**class** `find.`**`Part`**(*work*, *stub='*'*)
Looks for simulation output files in a part directory

example path: run/sim/part.0

**Files searched for include:** stub.dat stub.chk stub.parm stub.time

The default stub from Langmuir is stub=out, so files are name out.dat, etc.

work: the path of the part directory stub: the output file stub

**class** `find.`**`Run`**(*work*, *sim_stub='voltage*'*, *stub='*'*)
Looks for simulations in a run directory

example path: run

**The directories searched for:** run/sim.0 run/sim.1 run/sim.2 ...

**A more common example might be:** run/voltage.right_+0.2 run/voltage.right_+0.4 run/voltage.right_+0.6 ...

Use sim_stub='voltage.right*' to search for these directories

work : the path of the part directory sim_stub: the simulation directory stub (example=voltage.right*) stub : the output file stub (passed to the Part constructor)

**class** `find.`**`Sim`**(*work*, *stub='*'*)

Looks for parts in a simulation directory

example path: run/sim

**The directories searched for:** run/sim/part.0 run/sim/part.1 run/sim/part.2 ...

work: the path of the part directory stub: the output file stub (passed to the Part constructor)

`find.`**`find`**(*work*, *single=True*, *recursive=True*, *absolute=True*, *stub='*'*, *ext=None*, *exclude_dirs=False*, *exclude_files=False*, *sort_by=<function numbers at 0x442a410>*, *at_least_one=False*, *follow_links=False*)

A method for searching for files and directories using patterns.

single : do not return a list recursive : perform the search recursivly absolute : return absolute paths stub : the wildcard-able search pattern (* is the wildcard) exclude_dirs : do not include directories in the search exclude_files : do not include files in the search sort_by : a function (applied to paths) used to sort the results at_least_one : make sure at least one thing was found follow_links : do not follow symbolic links

`find.`**`slice_part`**(*path*, *regex='part'*)

Return dirname of path where run directory is

`find.`**`slice_path`**(*path*, *regex*)

Return dirname of path where the regex matches

`find.`**`slice_run`**(*path*, *regex='run'*)

Return dirname of path where run directory is

`find.`**`slice_sim`**(*path*, *regex='sim'*)

Return dirname of path where run directory is

## 2.4 checkpoint

**class** `checkpoint.`**`CheckPoint`**(*handle=None*)

A class to open langmuir checkpoint files.

| Attribute | Description |
|---|---|
| `electrons` | `list` of `int` |
| `holes` | `list` of `int` |
| `traps` | `list` of `int` |
| `defects` | `list` of `int` |
| `trapPotentials` | `list` of `float` |
| `fluxState` | `list` of `int` |
| `randomState` | `list` of `int` |
| `parameters` | `Parameters` |

**Parameters** **handle** (*str*) – filename or file object; Can be `None`.

**`clear`**()

Forget all stored information.

```
>>> chk.clear()
>>> print chk
[Electrons]     : 0
[Holes]         : 0
[Traps]         : 0
[Defects]       : 0
[TrapPotenials] : 0
[FluxState]     : 0
[RandomState]   : 0
[Parameters]    : 0
```

**fix_traps**()

Check trap parameter validity and remove extra or inconsistant information.

```
>>> chk.fix_traps()
```

**load**(*handle*)

Load checkpoint from a file.

**Parameters** **handle** (*str*) – filename or file object

```
>>> chk = langmuir.checkpoint.CheckPoint()
>>> chk.load('out.chk')
```

**reset**(*keep_elecs=False*, *keep_holes=False*)

Reset current step to zero, clear random state, clear flux state, and delete charge carriers.

**Parameters**

- **keep_elecs** (*bool*) – do not delete electrons
- **keep_holes** (*bool*) – do not delete holes

```
>>> chk.reset()
>>> print chk
[Electrons]     : 0
[Holes]         : 0
[Traps]         : 1000
[Defects]       : 1000
[TrapPotenials] : 1000
[FluxState]     : 0
[RandomState]   : 0
[Parameters]    : 10
...
```

**save**(*handle*)

Save checkpoint to a file.

**Parameters** **handle** (*str*) – filename or file object

```
>>> chk.save('sim.inp')
```

checkpoint.**compare**(*chk_i*, *chk_j*)

Rigorously compare to checkpoint files.

**Parameters**

- **chk_i** (*langmuir.checkpoint.CheckPoint*) – checkpoint object 1
- **chk_j** (*langmuir.checkpoint.CheckPoint*) – checkpoint object 1

**Returns** dict, bool

`checkpoint.`**`load`**(*handle*)

Load checkpoint file.

**Parameters** **handle** (*str*) – filename or file object

```
>>> chk = langmuir.checkpoint.load('out.chk')
```

`checkpoint.`**`load_last`**(*work*, ***kwargs*)

Load the last checkpoint file found in the working directory.

**Parameters** **work** (*str*) – directory to look in

chk = langmuir.checkpoint.load_last('/home/adam/Desktop/simulations')

## 2.5 parameters

**class** `parameters.`**`Parameters`**(*handle=None*)

A class to hold parameters in a dictionary. Inherits `'dict'. Will load parameters from a file if handle is not :py:obj:`None.`

**Parameters** **handle** (*str*) – filename or file object; Can be `None`.

```
>>> parm = langmuir.parameters.Parameters('out.parm')
```

**`load`**(*handle*)

Load parameters from a file.

**Parameters** **handle** (*str*) – filename or file object

```
>>> parm = langmuir.parameters.Parameters()
>>> parm.load('out.parm')
```

**`reset_output_parameters`**()

Reset output parameters to typical values; Minimizes output without turning it off completely.

```
>>> parm.reset_output_parameters()
>>> print parm
output.precision            = 15
output.width                = 23
output.stub                 = out
output.ids.on.delete        = False
output.ids.on.encounter     = False
output.coulomb              = 0
output.step.chk             = 0
output.potential            = False
output.xyz                  = 0
output.xyz.e                = True
output.xyz.h                = True
output.xyz.d                = False
output.xyz.t                = False
output.xyz.mode             = 0
image.traps                 = False
image.defects               = False
image.carriers              = 0
```

**`save`**(*handle*)

Save parameters to a file.

**Parameters** **handle** (*str*) – filename or file object

```
>>> parm.save('sim.inp')
```

**set_defaults**()

Set parameters to defaults found in database.

```
>>> parm.set_defaults()
>>> print parm
simulation.type            = solarcell
current.step               = 0
iterations.real            = 10000
random.seed                = 0
grid.z                     = 1
grid.y                     = 256
grid.x                     = 256
hopping.range              = 2
output.is.on               = True
iterations.print           = 1000
output.precision           = 15
output.width               = 23
output.stub                = out
output.ids.on.delete       = False
output.ids.on.encounter    = False
output.coulomb             = 0
output.step.chk            = 10
output.chk.trap.potential  = False
output.potential           = False
output.xyz                 = 0
output.xyz.e               = True
output.xyz.h               = True
output.xyz.d               = False
output.xyz.t               = False
output.xyz.mode            = 0
image.traps                = False
image.defects              = False
image.carriers             = 0
electron.percentage        = 0.01
hole.percentage            = 0.01
seed.charges               = 0.0
defect.percentage          = 0.0
trap.percentage            = 0.0
trap.potential             = 0.0
gaussian.stdev             = 0.0
seed.percentage            = 0.0
voltage.right              = 0.0
voltage.left               = 0.0
exciton.binding            = 0.0
slope.z                    = 0.0
coulomb.carriers           = True
coulomb.gaussian.sigma     = 1.0
defects.charge             = 0
temperature.kelvin         = 300.0
source.rate                = 0.001
e.source.l.rate            = -1.0
e.source.r.rate            = -1.0
h.source.l.rate            = -1.0
h.source.r.rate            = -1.0
generation.rate            = -1.0
balance.charges            = False
```

```
source.metropolis          = False
source.coulomb             = False
source.scale.area          = 65536.0
drain.rate                 = 0.9
e.drain.l.rate             = -1.0
e.drain.r.rate             = -1.0
h.drain.l.rate             = -1.0
h.drain.r.rate             = -1.0
recombination.rate         = 0.0001
recombination.range        = 0
use.opencl                 = True
work.x                     = 4
work.y                     = 4
work.z                     = 4
work.size                  = 256
opencl.threshold           = 256
opencl.device.id           = 0
max.threads                = -1
```

**to_ndarray**(*result=None*, *rows=1*, *i=0*)
Copy parameters into a `np.ndarray()`.

**Parameters**

- **result** (`numpy.array()`) – in place array to be modified; if `None`, a new array is created.
- **rows** (*int*) – number of rows to allocate in new array
- **i** (*int*) – row-id to write paramters to

**Returns** `numpy.array()`

```
>>> a = parm.to_ndarray()
```

parameters.**create_ndarray**(*rows=0*)
Create empty `numpy.array()` with correct column headers.

**Parameters** **rows** (*int*) – number of rows to allocate in new array

**Returns** `numpy.array()`

```
>>> a = create_ndarray(10)
```

parameters.**load**(*handle*)
Load parameters from file.

**Parameters** **handle** (*str*) – filename or file object

**Returns** `Parameters`

```
>>> parm = langmuir.paramters.load('out.parm')
```

## 2.6 database

**class** database.**ColumnList**
List of `ColumnMetaData`

**append**(*column*)
Append a column.

**Parameters** **column** (`ColumnMetaData`) – column object

**class** `database.`**`ColumnMetaData`**(*name*, *key*, *pytype*, *dflt*, *units*, *fmt*, *calculated*)

Class to store meta data about output files and parameters.

**Parameters**

- **name** (*str*) – parameter name
- **key** (*str*) – alias
- **dflt** (*object*) – default value
- **units** (*str*) – unit string
- **fmt** (*str*) – format string
- **calculated** (*bool*) – if calculated by Langmuir

## 2.7 datfile

`datfile.`**`add_field`**(*a*, *name*, *fmt*, *before=False*)

Add a field to ndarray

a: ndarray name: field name fmt: field type before: add it before the other fields, otherwise after

`datfile.`**`calculate`**(*data*)

calculate current, etc for ndarray

`datfile.`**`combine`**(*objs*, *load_func=<function load_dat at 0x6715ed8>*)

combine a list of dat files, assumed to be indexed by simulation:time

objs: list of file names, or ndarrays load_func: function used to open files

`datfile.`**`combine_first`**(*A*, *B*, *index=None*)

combine two tables with same dtype

`datfile.`**`create`**(*rows=0*)

create dat array with values set to zero

rows: number of rows

`datfile.`**`equilibrate`**(*data*, *i1=None*, *i0=None*)

extract result, taking out equilibration steps

data: ndarray i1: int, index of "last step" i0: int, index of "equilibration step"

`datfile.`**`load_dat`**(*handle*)

load dat file into ndarray

`datfile.`**`load_pkl`**(*handle*)

load pkl file into ndarray

`datfile.`**`save_pkl`**(*obj*, *handle*)

save ndarray into pkl file

## 2.8 surface

**class** `surface.`**`WaveDimensions`**(*L=6.283185307179586*, *n=1*)

Compute wavelength, wavenumber, etc from an interval length (L) and number of waves (n).

**Parameters**

- **L** (*float*) – interval length
- **n** (*int*) – number of waves in interval

```
>>> wx = WaveDimensions(10, 2)
>>> print wx
[Wave Dimensions]
    L      = 10
    n      = 2
    lambda = 5.00000e+00
    nubar  = 2.00000e-01
    k      = 1.25664e+00
```

**calc**(*L=None*, *n=None*)
Perform calculations to compute wavelength, wavenumber, etc. Called automatically in constructor.

**Parameters**

- **L** (*float*) – interval length
- **n** (*int*) – number of waves in interval

surface.**band3D**(*x*, *y*, *z*, *kx*, *ky*, *kz*)
Surface function f(x,y,z) for bands

**Parameters**

- **x** (*float*) – x-value(s)
- **y** (*float*) – y-value(s)
- **z** (*float*) – z-value(s)
- **kx** (*float*) – 2 pi nx / Lx
- **ky** (*float*) – 2 pi ny / Ly
- **kz** (*float*) – 2 pi nz / Lz

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = band3D(x, y, v, w.k, w.k, w.k)
```

surface.**gyroid**(*x*, *y*, *z*, *kx*, *ky*, *kz*)
Surface function f(x,y,z) for gyroid

**Parameters**

- **x** (*float*) – x-value(s)
- **y** (*float*) – y-value(s)
- **z** (*float*) – z-value(s)
- **kx** (*float*) – 2 pi nx / Lx
- **ky** (*float*) – 2 pi ny / Ly
- **kz** (*float*) – 2 pi nz / Lz

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = gyroid(x, y, v, w.k, w.k, w.k)
```

`surface.`**`paint_checkerboard_xy`**(*data*, *dx*, *dy*, *value=1.0*)

Extend a checkerboard pattern along the z-direction.

Draw a checkerboard pattern in xy plane, for all z values

**Parameters**

- **dx** (*int*) – x-size of checkerboard square
- **dy** (*int*) – y-size of checkerboard square
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_checkerboard_xy(data, 8, 8, 1.0)
```

`surface.`**`paint_checkerboard_yz`**(*data*, *dy*, *dz*, *value=1.0*)

Extend a checkerboard pattern along the x-direction.

Draw a checkerboard pattern in yz plane, for all x values

**Parameters**

- **dy** (*int*) – y-size of checkerboard square
- **dz** (*int*) – z-size of checkerboard square
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_checkerboard_yz(data, 8, 8, 1.0)
```

`surface.`**`paint_cube`**(*data*, *x*, *dx*, *y*, *dy*, *z*, *dz*, *value=0.1*)

Draw a cube of traps

The cube has a (length, width, height) of (dx, dy, dz) and lower back left cornor of (x, y, z)

**Parameters**

- **x** (*int*) – the lower front left cornor x-value
- **y** (*int*) – the lower front left cornor y-value
- **z** (*int*) – the lower front left cornor z-value
- **dx** (*int*) – the width of the plane
- **dy** (*int*) – the height of the plane
- **dz** (*int*) – the height of the plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_cube(data, 0, 16, 0, 16, 0, 16, 1.0)
```

`surface.`**`paint_plane_xy`**(*data*, *z*, *dz*, *value=1.0*)

Draw an xy plane of traps

The plane is parallel to the xy-plane and has a thickness dz and is located at z.

**Parameters**

- **z** (*int*) – distance of plane from the zero xy-plane
- **dz** (*int*) – thickness of plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_plane_xy(data, 0, 8, 1.0)
```

surface.**paint_plane_xz**(*data*, *y*, *dy*, *value=1.0*)

Draw an xz plane of traps

The plane is parallel to the xz-plane and has a thickness dy and is located at y.

**Parameters**

- **y** (*int*) – distance of plane from the zero xz-plane
- **dy** (*int*) – thickness of plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_plane_xz(data, 0, 8, 1.0)
```

surface.**paint_plane_yz**(*data*, *x*, *dx*, *value=1.0*)

Draw an yz plane of traps

The plane is parallel to the yz-plane and has a thickness dx and is located at x.

**Parameters**

- **x** (*int*) – distance of plane from the zero yz-plane
- **dx** (*int*) – thickness of plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_plane_yz(data, 0, 8, 1.0)
```

surface.**paint_square_xy**(*data*, *x*, *dx*, *y*, *dy*, *z=0*, *value=0.1*)

Draw a square of traps in the xy plane

The plane is parallel to the xy-plane, has a thickness 1, cornor at (x, y), and (width,height) of (dx, dy).

**Parameters**

- **x** (*int*) – the lower left cornor x-value
- **dx** (*int*) – the width of the plane
- **y** (*int*) – the lower left cornor y-value
- **dy** (*int*) – the height of the plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_square_xy(data, 0, 16, 0, 16, 0, 1.0)
```

surface.**paint_square_xz**(*data*, *x*, *dx*, *z*, *dz*, *y=0*, *value=0.1*)

Draw a square of traps in the xy plane

The plane is parallel to the xz-plane, has a thickness 1, cornor at (x, z), and (width,height) of (dx, dz).

**Parameters**

- **x** (*int*) – the lower left cornor x-value
- **dx** (*int*) – the width of the plane
- **z** (*int*) – the lower left cornor z-value

- **dz** (*int*) – the height of the plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_square_xz(data, 0, 16, 0, 16, 0, 1.0)
```

`surface.`**`paint_square_yz`**(*data*, *y*, *dy*, *z*, *dz*, *x=0*, *value=0.1*)

Draw a square of traps in the xy plane

The plane is parallel to the yz-plane, has a thickness 1, cornor at (y, z), and (width,height) of (dy, dz).

**Parameters**

- **y** (*int*) – the lower left cornor y-value
- **dy** (*int*) – the height of the plane
- **z** (*int*) – the lower left cornor z-value
- **dz** (*int*) – the width of the plane
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_square_yz(data, 0, 16, 0, 16, 0, 1.0)
```

`surface.`**`paint_stripe_dx`**(*data*, *dx*, *value=1.0*)

Stack slabs along the x-direction of thickness dx.

Draw alternating rectangular prisms with of thickness dx forall x, y

**Parameters**

- **dx** (*int*) – thickness of slab
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_stripe_dx(data, 8, 1.0)
```

`surface.`**`paint_stripe_dy`**(*data*, *dy*, *value=1.0*)

Stack slabs along the y-direction of thickness dy.

Draw alternating rectangular prisms with of thickness dy forall x, z

**Parameters**

- **dy** (*int*) – thickness of slab
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_stripe_dy(data, 8, 1.0)
```

`surface.`**`paint_stripe_dz`**(*data*, *dz*, *value=1.0*)

Stack slabs along the z-direction of thickness dz.

Draw alternating rectangular prisms with of thickness dz forall x, y

**Parameters**

- **dz** (*int*) – thickness of slab
- **value** (*float*) – trap energy value in eV

```
>>> data = np.zeros((64, 64, 64))
>>> data = paint_stripe_dz(data, 8, 1.0)
```

surface.**scherk_first_surface**(*x, y, z, kx, ky, kz*)

Surface function f(x,y,z) for scherk

**Parameters**

- **x** (*float*) – x-value(s)
- **y** (*float*) – y-value(s)
- **z** (*float*) – z-value(s)
- **kx** (*float*) – 2 pi nx / Lx
- **ky** (*float*) – 2 pi ny / Ly
- **kz** (*float*) – 2 pi nz / Lz

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = scherk_first_surface(x, y, v, w.k, w.k, w.k)
```

surface.**schwarz_d_surface**(*x, y, z, kx, ky, kz*)

Surface function f(x,y,z) for dsurface

**Parameters**

- **x** (*float*) – x-value(s)
- **y** (*float*) – y-value(s)
- **z** (*float*) – z-value(s)
- **kx** (*float*) – 2 pi nx / Lx
- **ky** (*float*) – 2 pi ny / Ly
- **kz** (*float*) – 2 pi nz / Lz

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = schwarz_d_surface(x, y, v, w.k, w.k, w.k)
```

surface.**schwarz_p_surface**(*x, y, z, kx, ky, kz*)

Surface function f(x,y,z) for psurface

**Parameters**

- **x** (*float*) – x-value(s)
- **y** (*float*) – y-value(s)
- **z** (*float*) – z-value(s)
- **kx** (*float*) – 2 pi nx / Lx
- **ky** (*float*) – 2 pi ny / Ly
- **kz** (*float*) – 2 pi nz / Lz

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = schwarz_p_surface(x, y, v, w.k, w.k, w.k)
```

`surface.`**`show3D`** (*x*, *y*, *z*, *v*, *show=False*, ***kwargs*)

Wrapper around mayavi contour3D (slow) to visualize surface.

**Parameters**

- **x** (*float*) – set of x-points
- **y** (*float*) – set of y-points
- **z** (*float*) – set of z-points
- **v** (*float*) – set of v-points (surface values)
- **show** (*bool*) – open mayavi window

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = gyroid(x, y, v, w.k, w.k, w.k)
>>> show3D(x, y, z, v)
>>> mlab.show()
```

`surface.`**`showXY`** (*x*, *y*, *z*, *v*, *zlevel=0*, **args*, ***kwargs*)

Wrapper around pyplot.contourf to visualize surface slice.

**Parameters**

- **x** (*float*) – set of x-points
- **y** (*float*) – set of y-points
- **z** (*float*) – set of z-points
- **v** (*float*) – set of v-points (surface values)
- **zlevel** (*int*) – z-value of slice

```
>>> w = WaveDimensions(10, 2)
>>> x, y, z = np.mgrid[0:10:100j,0:10:100j,0:10:100j]
>>> v = gyroid(x, y, v, w.k, w.k, w.k)
>>> showXY(x, y, z, v)
>>> plt.show()
```

## 2.9 grid

**class** `grid.`**`Grid`**

A class to represent a rectangular grid of points. There are many alternative constructors.

**See Also:**

`linspace()`, `arange()`, `vtk()` `checkpoint()`

**Warning:** Do not use the main constructor.

```
>>> grid = Grid.linspace(0.5, 0.5, 0.5, 9.5, 9.5, 9.5, 10, 10, 10)
>>> grid = Grid.arange(0.5, 0.5, 0.5, 9.5, 9.5, 9.5, 1, 1, 1)
>>> grid = Grid.vtk(0.5, 0.5, 0.5, 1, 1, 1, 10, 10, 10)
```

**classmethod** **`arange`** (*x0*, *y0*, *z0*, *x1*, *y1*, *z1*, *dx*, *dy*, *dz*)

Alternative Grid constructor similar to `np.arange()`

**Parameters**

- **x0** (*float*) – x origin
- **y0** (*float*) – x origin
- **z0** (*float*) – x origin
- **x1** (*float*) – x endpoint
- **y1** (*float*) – y endpoint
- **z1** (*float*) – z endpoint
- **dx** (*float*) – x delta
- **dy** (*float*) – y delta
- **dz** (*float*) – z delta

```
>>> grid = Grid.arange(0.5, 0.5, 0.5, 9.5, 9.5, 9.5, 1, 1, 1)
```

**classmethod checkpoint**(*chk*)

Alternative Grid constructor to create grid from checkpoint file.

**Parameters chk** (`langmuir.checkpoint.Checkpoint`) – checkpoint file object, file name, or file handle

```
>>> chk = langmuir.checkpoint.load('out.chk')
>>> grid = grid.checkpoint(chk)
```

**create_mgrid**(*force=False*)

Compute and return a `np.mgrid()`. The mgrid is also stored as `grid.mx`, `grid.my`, `grid.mz`.

**Parameters force** (*bool*) – force the recomputation of the mgrid

```
>>> grid = Grid.vtk(0, 0, 0, 1, 1, 1, 3, 3, 3)
>>> x, y, z = grid.create_mgrid()
>>> print x
... [[[ 0.  0.  0.]
...   [ 0.  0.  0.]
...   [ 0.  0.  0.]]
...
...  [[ 1.  1.  1.]
...   [ 1.  1.  1.]
...   [ 1.  1.  1.]]
...
...  [[ 2.  2.  2.]
...   [ 2.  2.  2.]
...   [ 2.  2.  2.]]]
```

**create_ogrid**(*force=False*)

Compute and return a `np.ogrid()`. The ogrid is also stored as `grid.ox`, `grid.oy`, `grid.oz`.

**Parameters force** (*bool*) – force the recomputation of the ogrid

```
>>> grid = Grid.vtk(0, 0, 0, 1, 1, 1, 3, 3, 3)
>>> x, y, z = grid.create_ogrid()
>>> print x
... [ 0.  1.  2.]
```

**create_zeros**()

Compute a matrix of zeros with the same shape as the grid.

```
>>> v = grid.create_zeros()
```

**classmethod `linspace`** (*x0, y0, z0, x1, y1, z1, px, py, pz*)
Alternative Grid constructor similar to `np.linspace()`

**Parameters**

- **x0** (*float*) – x origin
- **y0** (*float*) – x origin
- **z0** (*float*) – x origin
- **x1** (*float*) – x endpoint
- **y1** (*float*) – y endpoint
- **z1** (*float*) – z endpoint
- **px** (*int*) – x points
- **py** (*int*) – y points
- **pz** (*int*) – z points

```
>>> grid = Grid.linspace(0.5, 0.5, 0.5, 9.5, 9.5, 9.5, 10, 10, 10)
```

**`setup`** (*x0, y0, z0, x1, y1, z1, px, py, pz*)
Same as `linspace()`. Called by all constructors. You probably will not use this explicitly.

**See Also:**

`linspace()`

**classmethod `vtk`** (*x0, y0, z0, dx, dy, dz, px, py, pz*)
Alternative Grid constructor

**Parameters**

- **x0** (*float*) – x origin
- **y0** (*float*) – x origin
- **z0** (*float*) – x origin
- **dx** (*float*) – x delta
- **dy** (*float*) – y delta
- **dz** (*float*) – z delta
- **px** (*int*) – x points
- **py** (*int*) – y points
- **pz** (*int*) – z points

```
>>> grid = Grid.vtk(0.5, 0.5, 0.5, 1, 1, 1, 10, 10, 10)
```

**class** `grid.`**`IndexMapper`** (*xsize, ysize, zsize*)
A class that maps between site indicies the Langmuir way.

**Parameters**

- **xsize** (*int*) – x-dimension of grid, like grid.x
- **ysize** (*int*) – y-dimension of grid, like grid.y

- **zsize** (*int*) – z-dimension of grid, like grid.z

```
>>> imap = langmuir.grid.IndexMapper(10, 10, 10)
```

**indexS** (*x_index*, *y_index*, *z_index*)
Compute (langmuir's) site index from x, y, and z index

**Parameters**

- **x_index** (*int*) – x-site
- **y_index** (*int*) – y-site
- **z_index** (*int*) – z-site

```
>>> s = imap.indexS(0, 0, 0)
```

**indexX** (*s_index*)
Compute (langmuir's) x index from site index

**Parameters** **s_index** (*int*) – site index

```
>>> x = imap.indexX(10)
```

**indexY** (*s_index*)
Compute (langmuir's) y index from site index

**Parameters** **s_index** (*int*) – site index

```
>>> y = imap.indexX(10)
```

**indexZ** (*s_index*)
Compute (langmuir's) z index from site index

**Parameters** **s_index** (*int*) – site index

```
>>> z = imap.indexX(10)
```

**class** grid.**Mesh** (*grid*)
Perform fast computation of Coulomb interactions and distances with a precomputed mesh.

**Parameters** **grid** (Grid) – grid object

```
>>> grid = langmuir.grid.Grid.vtk(0, 0, 0, 1, 1, 1, 10, 10, 10)
>>> mesh = langmuir.grid.Mesh(grid)
```

**coulomb** (*xi_ids*, *yi_ids*, *zi_ids*, *xj_ids=None*, *yj_ids=None*, *zj_ids=None*, *q=1*)
Compute coulomb interaction at j's due to charges at i's. If no j's are passed, then the answer will be computed at every mesh point (expensive!)'

**Parameters**

- **xi_ids** (*list, int*) – charge x-position(s)
- **yi_ids** (*list, int*) – charge y-position(s)
- **zi_ids** (*list, int*) – charge z-position(s)
- **xj_jds** – energy x-position(s)
- **yj_jds** – energy y-position(s)
- **zj_jds** – energy z-position(s)
- **q** (*int, float*) – charge

```
>>> grid = langmuir.grid.Grid.vtk(0, 0, 0, 1, 1, 1, 10, 10, 10)
>>> mesh = langmuir.grid.Mesh(grid)
>>> coul = mesh.coulomb(0, 0, 0, 1, 1, 1, -1)
```

**distances** (*xi_ids*, *yi_ids*, *zi_ids*, *xj_ids=[ ]*, *yj_ids=[ ]*, *zj_ids=[ ]*)
Compute all distances between i's, or compute all distances between i's and j's

**Parameters**

- **xi_ids** (*list, int*) – initial x-position(s)
- **yi_ids** (*list, int*) – initial y-position(s)
- **zi_ids** (*list, int*) – initial z-position(s)
- **xj_jds** – final x-position(s)
- **yj_jds** – final y-position(s)
- **zj_jds** – final z-position(s)

```
>>> grid = langmuir.grid.Grid.vtk(0, 0, 0, 1, 1, 1, 10, 10, 10)
>>> mesh = langmuir.grid.Mesh(grid)
>>> dist = mesh.distances(0, 0, 0, 1, 1, 1)
```

**class** grid.**XYZV** (*grid*, *site_ids*, *site_values=None*)
Put site values on a mesh using site ids.

**Parameters**

- **grid** (Grid) – grid object
- **site_ids** (*list*) – site indcies
- **site_values** (*list or scalar*) – site values

```
>>> chk  = langmuir.checkpoint.load('out.chk')
>>> grid = langmuir.grid.Grid.checkpoint(chk)
>>> xyzv = langmuir.grid.XYZV(grid, chk.electrons, -1)
```

## 2.10 analyze

analyze.**calculate** (*obj*)
Compute all flux

analyze.**calculate_flux** (*obj*, *flux*)
Compute current for flux

analyze.**calculate_in** (*obj*)
Compute carrier in flux

analyze.**calculate_left** (*obj*)
Compute carrier left flux

analyze.**calculate_out** (*obj*)
Compute carrier out flux

analyze.**calculate_right** (*obj*)
Compute carrier right flux

analyze.**combine** (*objs*, *load_func=<function load_dat at 0x5a31b90>*)
Combine a set of panda's DataFrames into a single DataFrame

analyze.**equilibrate**(*obj*, *last*, *equil=None*)
Get the difference between two steps

analyze.**fix**(*obj*)
Fix a panda's DataFrame to have correct types and column names

analyze.**load_dat**(*handle*, *compression=None*, *sep='\\s*'*, ***kwargs*)
Load a Langmuir output dat file into a panda's DataFrame

handle : file name or obj kwargs : keyword arguments passed on to pandas.read_table

analyze.**load_pkl**(*handle*, *max_objs=10*)
Load max objs from a pkl file

analyze.**load_pkls**(*pkls*)
Load a set of pkls into a Panda's Panel

analyze.**save_pkl**(*obj*, *handle*)
Save obj to a pkl file

## 2.11 ivline

**class** ivline.**IVLine**(**args*, ***kwargs*)
Collection of data for an IV curve and methods to analyze it

```
>>> iv = IVLine()
>>> iv.load_pkls(['run_0.pkl', 'run_1.pkl'])
>>> iv.calculate()
```

**calculate**(*panel=None*)
Calculate power, density, etc from loaded data.

```
>>> iv.calculate()
```

**csv**(*handle*)
Save data to a CSV file.

**data**()
Convert data into a python dict

**dataframe**()
Convert data into a pandas DataFrame

**classmethod from_pkls**(*pkls*)
Alternative constructor. Creates instance and loads the pkl files.

**Parameters** **pkls** (*str*) – list of paths

**Returns** `IVLine`

**classmethod from_search**(*path*, ***kwargs*)
Alternative constructor. Search the path for a set of pkl files, and load them into an IVLine instance.

**Parameters** **path** (*str*) – path to search

**Returns** `IVLine`

**load_pkls**(*paths*, ***kwargs*)
Load a set of pkl files created using langmuir.analyze into IVLine.

**Parameters** **paths** (*list*) – paths to pkl files

```
>>> iv.load_pkls(['run_0.pkl', 'run_1.pkl'])
>>> iv.calculate()
```

**pkl**(*handle*)
Save data to a CSV file.

**process_panel**(*panel=None*)
Extract current from panda's panel.

**results**()
Convert calculate results to a python dict

**class** ivline.**IVLineS**(**args*, ***kwargs*)
An IVLine specialized for solar cells.

**calculate**(*s=1.5*, *mode='tanh'*, *recycle=False*, *order=8*, *k=2*, *kind='linear'*)
Calculate fill factor.

**Parameters**

- **s** (*float*) – voltage shift
- **mode** (*str*) – fitting mode (tanh, power, interp1d)
- **order** (*int*) – order for power fit
- **kind** (*str*) – kind for interp1d
- **recycle** (*bool*) – reuse popt from current fit for power fit
- **k** (*int*) – order of univariate spline

```
>>> iv.calculate(langmuir.fit.FitTanh, langmuir.fit.FitXTanh)
```

**results**()
Get a summary of results.

**test**(*points=20*, *error=0.05*)
Create some test data.

**Parameters**

- **points** (*int*) – number of points
- **error** (*float*) – std of simulated error

```
>>> iv.test()
>>> iv.calculate()
```

**class** ivline.**Stats**(*array*, *name*)
Compute various statistics of an array like object.

| Attr | Description |
|---|---|
| **name** | stub |
| **max** | max of data |
| **min** | min of data |
| **rng** | range of data |
| **avg** | average of data |
| **std** | standard deviation of data |

```
>>> s = Stats([1, 2, 3, 4, 5])
```

**dataDict**()
Get summary of stats.

**class** `ivline.`**`units`**

Struct to manage units for IV curves.

| Data | Description |
|---|---|
| **v** | voltage |
| **i** | current |
| **p** | power |
| **d** | current density |
| **r** | power density |
| **a** | area |

| Attribute | Description |
|---|---|
| ? | v, i, p, d, r |
| ?units_input | units assumed for input data |
| ?units | units used for quantity |
| ?str1 | units as latex str |
| ?str2 | units as latex str (no frac) |
| q | quantities module |

**static** **`cfactor`**(*u1*, *u2*)

Compute conversion factor.

**Parameters**

- **u1** – object with units, can be a :py:obj'str'.
- **u2** – object with units, can be a :py:obj'str'.

**Returns** float factor

**`q`** **= <module 'quantities' from '/home/adam/.local/lib/python2.7/site-packages/quantities/__init__.pyc'>**

**static** **`rescale`**(*obj*, *u1*, *u2=None*)

Rescale units of object.

**Parameters**

- **obj** – object with or without units
- **u1** – object with units, can be a :py:obj'str'.
- **u2** – object with units, can be a :py:obj'str' or `None`.

```
>>> obj = units.rescale([1, 2, 3], 'nA', 'A')         # nA   -> A
>>> obj = units.rescale([1, 2, 3], 'A')               # None -> A
>>> obj = units.rescale([1] * units.q.nA, 'A')        # nA   -> A
>>> obj = units.rescale([1] * units.q.nA, 'A', 'nA') # nA   -> A -> nA
```

returns: `quantities.Quantity`

**static** **`to_units`**(*u*)

Turn object into `quantities.Quantity` and extract units object

**Parameters** **u** – object with units, can be a :py:obj'str'.

## 2.12 plot

`plot.`**`errorbar`**(*x*, *y*, *color='r'*, ***kwargs*)

wrapper around errorbar

`plot.`**`mtext`**(*s*)
Puts string into a latex math environment.

**Parameters** **s** (*str*) – string to be placed in math environment

`plot.`**`transformA`**(*x*, *y*, *a=None*)
transform angle from data to screen coordinates

`plot.`**`transformX`**(*x*, *transform_from=None*, *transform_to=None*)
transform x from axes to data coords

`plot.`**`transformY`**(*y*, *transform_from=None*, *transform_to=None*)
transform y from axes to data coords

## 2.13 fit

A set of classes used to fit 2D data.

**class** `fit.`**`Fit`**(*x*, *y*, *func*, *popt=None*, *yerr=None*)
Base class for fitting.

**Parameters**

- **x** (*list*) – array of x-values
- **y** (*list*) – array of y-values
- **func** (*function*) – function object
- **popt** (*list*) – initial guess for parameters

**Warning:** Do not explicitly create Fit instance, its a base class.

```
>>> fit = Fit(xdata, ydata, func)
>>> x = np.linspace(0, 10, 100)
>>> y = fit(x)
>>> plt.plot(x, y, 'r-')
```

**`brute`**(*a=0*, *b=1*, *find_max=False*, *return_y=False*, ***kwargs*)
Wrapper around `scipy.optimize.brute()`. Finds minimum in range.

**Parameters**

- **a** (*float*) – lower x-bound
- **b** (*float*) – upper x-bound
- **find_max** (*bool*) – find max instead of min
- **return_y** (*bool*) – return x and fit(x)

**Returns** xval, yval

```
>>> xval, yval = fit.brute(a=-1, b=1)
```

**`derivative`**(*x*, ***kwargs*)
Wrapper around `scipy.misc.derivative()`, unless an inheriting class implements the derivative analytically.

**Parameters** **x** (*float*) – x-value to evaluate derivative at

**Returns** float

**maxbrute** (*a=0*, *b=1*, *return_y=False*, ***kwargs*)
Calls `Fit.brute()` with find_max=True

**maximize** (*x0*, *return_y=True*, ***kwargs*)
Calls `Fit.minimize()` with find_max=True

**minimize** (*x0=0*, *find_max=False*, *return_y=True*, ***kwargs*)
Wrapper around `scipy.optimize.minimize()`. Finds minimum using a initial guess.

**Parameters**

- **x0** (*float*) – initial guess for x
- **find_max** (*bool*) – find max instead of min
- **return_y** (*bool*) – return x and fit(x)

**Returns** xval, yval

```
>>> xval, yval = fit.minimize(x0=0.1)
```

**plot** (*xmin=None*, *xmax=None*, *xpoints=100*, ***kwargs*)
Wrapper around `matplotlib.pyplot.plot()`. Plots the fit line.

**Parameters**

- **xmin** (*float*) – min x-value
- **xmax** (*float*) – max x-value
- **xpoints** (*int*) – number of xpoints

```
>>> fit.plot(xmin=-1.0, xmax=1.0, xpoints=10, lw=2, color='blue')
```

**plot_tangent** (*x*, *xmin=None*, *xmax=None*, *xpoints=100*, ***kwargs*)
Plot a tangent line at x.

**Parameters**

- **x** (*float*) – x-value to evaluate derivative at
- **xmin** (*float*) – min x-value
- **xmax** (*float*) – max x-value
- **xpoints** (*int*) – number of xpoints

```
>>> fit.plot_tangent(x=1.5, lw=2, color='blue')
```

**solve** (*y=0*, *x0=0*, *return_y=False*, ***kwargs*)
Wrapper around `scipy.optimize.fsolve()`. Solves y=fit(x) for x.

**Parameters**

- **y** (*float*) – y-value
- **x0** (*float*) – initial guess
- **return_y** (*bool*) – return x and fit(x)

**Returns** xval, yval

```
>>> xval, yval = fit.solve(y=0, x0=1.0)
```

**sort** (***kwargs*)
Sort x and y values.

**summary**()
Create a summary `dict` of fit results.

**Returns** `dict`

**tangent**(*x*, ***kwargs*)
Compute a tangent line at x.

**Parameters**

- **x** (*float*) – x-value to evaluate derivative at
- **xmin** (*float*) – min x-value
- **xmax** (*float*) – max x-value
- **xpoints** (*int*) – number of xpoints

```
>>> func = fit.tangent(x=1.5)
>>> print func(1.5)
... 0.0
```

**text**(*s*, *x*, *y=None*, *transform=None*, *rotate=False*, *draw_behind=True*, ***kwargs*)
Wrapper around `matplotlib.pyplot.text()`. Useful for putting text along the fit line drawn by `Fit.plot()`, and rotating the text using the derivative.

**Parameters**

- **s** (*str*) – text
- **x** (*float*) – x-value of text
- **y** (*float*) – y-value of text; if None, use the fit line
- **transform** (`matplotlib.transforms.Transform`) – matplotlib transform; if None, plt.gca().transAxes
- **rotate** (*bool*) – rotate text using angle computed from derivative
- **draw_behind** (*bool*) – hide the plot objects behind the text

```
>>> fit.text('Hello!', 0.1, rotate=True, fontsize='large')
```

class `fit.`**FitBarycentric**(*x*, *y*, *popt=None*, *yerr=None*, ***kwargs*)
Fit to lagrange interpolating spline. It sucks.

class `fit.`**FitCos**(*x*, *y*, *popt=None*, *yerr=None*)
$a\cos(bx + c) + d$

**derivative**(*x*, ***kwargs*)
analytic derivative of function

**summary**()
create a summary `dict`

class `fit.`**FitErf**(*x*, *y*, *popt=None*, *yerr=None*)
$a\,\mathrm{erf}(bx + c) + d$

**summary**()
create a summary `dict`

class `fit.`**FitGaussian**(*x*, *y*, *popt=None*, *yerr=None*)
$ae^{\frac{-(x-m)^2}{2\sigma^2}}$

**summary**()
create a summary `dict`

**class** `fit.`**`FitInterp1D`**(*x*, *y*, *popt=None*, *yerr=None*, ***kwargs*)
Fit to interpolating function.

**Parameters** **kind** (*str*) – linear, nearest, zero, slinear, quadratic, cubic.

**class** `fit.`**`FitLagrange`**(*x*, *y*, *popt=None*, *yerr=None*, ***kwargs*)
Fit to lagrange interpolating spline. It sucks.

**class** `fit.`**`FitLinear`**(*x*, *y*, *popt=None*, *yerr=None*)
$mx + b$

**`derivative`**(*x*, ***kwargs*)
analytic derivative of function

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitPower`**(*x*, *y*, *order=1*, *popt=None*, *yerr=None*)
$\sum_{i=0}^{N} c_i x^i$

**Parameters** **order** (*int*) – order of polynomial

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitQuadratic`**(*x*, *y*, *popt=None*, *yerr=None*)
$ax^2 + bx + c$

**`derivative`**(*x*, ***kwargs*)
analytic derivative of function

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitSin`**(*x*, *y*, *popt=None*, *yerr=None*)
$a \sin(bx + c) + d$

**`derivative`**(*x*, ***kwargs*)
analytic derivative of function

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitTanh`**(*x*, *y*, *popt=None*, *yerr=None*)
$a \tanh(bx + c) + d$

**`derivative`**(*x*, ***kwargs*)
analytic derivative of function

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitUnivariateSpline`**(*x*, *y*, *popt=None*, *yerr=None*, ***kwargs*)
Fit to spline.

**Parameters** **k** – degree of spline (<=5)

**class** `fit.`**`FitXErf`**(*x*, *y*, *popt=None*, *yerr=None*)
$ax \operatorname{erf}(bx + c) + d$

**`summary`**()
create a summary `dict`

**class** `fit.`**`FitXTanh`**(*x*, *y*, *popt=None*, *yerr=None*)
$ax \tanh(bx + c) + d$

**`derivative`**(*x*, ***kwargs*)
analytic derivative of function

**`summary`**()
create a summary `dict`

## 2.14 Indices and tables

- *genindex*
- *modindex*
- *search*

# PYTHON MODULE INDEX

a

analyze, 24

c

checkpoint, 9
common, 5

d

database, 13
datfile, 14

f

find, 8
fit, 28

g

grid, 20

i

ivline, 25

p

parameters, 11
plot, 27

r

regex, 8

s

surface, 14

# INDEX

## A

add_field() (in module datfile), 14
addedKeys() (common.DictDiffer class method), 5
addedValues() (common.DictDiffer class method), 5
analyze (module), 24
append() (database.ColumnList method), 13
arange() (grid.Grid class method), 20

## B

band3D() (in module surface), 15
brute() (fit.Fit method), 28

## C

calc() (surface.WaveDimensions method), 15
calculate() (in module analyze), 24
calculate() (in module datfile), 14
calculate() (ivline.IVLine method), 25
calculate() (ivline.IVLineS method), 26
calculate_flux() (in module analyze), 24
calculate_in() (in module analyze), 24
calculate_left() (in module analyze), 24
calculate_out() (in module analyze), 24
calculate_right() (in module analyze), 24
cfactor() (ivline.units static method), 27
changedKeys() (common.DictDiffer class method), 5
changedValues() (common.DictDiffer class method), 5
CheckPoint (class in checkpoint), 9
checkpoint (module), 9
checkpoint() (grid.Grid class method), 21
clear() (checkpoint.CheckPoint method), 9
ColumnList (class in database), 13
ColumnMetaData (class in database), 14
combine() (in module analyze), 24
combine() (in module datfile), 14
combine_first() (in module datfile), 14
command_script() (in module common), 6
common (module), 5
compare() (in module checkpoint), 10
coulomb() (grid.Mesh method), 23
create() (in module datfile), 14
create_mgrid() (grid.Grid method), 21
create_ndarray() (in module parameters), 13
create_ogrid() (grid.Grid method), 21
create_zeros() (grid.Grid method), 21
csv() (ivline.IVLine method), 25

## D

data() (ivline.IVLine method), 25
database (module), 13
dataDict() (ivline.Stats method), 26
dataframe() (ivline.IVLine method), 25
datfile (module), 14
derivative() (fit.Fit method), 28
derivative() (fit.FitCos method), 30
derivative() (fit.FitLinear method), 31
derivative() (fit.FitQuadratic method), 31
derivative() (fit.FitSin method), 31
derivative() (fit.FitTanh method), 31
derivative() (fit.FitXTanh method), 31
DictDiffer (class in common), 5
distances() (grid.Mesh method), 24

## E

equilibrate() (in module analyze), 24
equilibrate() (in module datfile), 14
errorbar() (in module plot), 27
evaluate() (in module common), 7

## F

find (module), 8
find() (in module find), 9
Fit (class in fit), 28
fit (module), 28
FitBarycentric (class in fit), 30
FitCos (class in fit), 30
FitErf (class in fit), 30
FitGaussian (class in fit), 30
FitInterp1D (class in fit), 31
FitLagrange (class in fit), 31
FitLinear (class in fit), 31
FitPower (class in fit), 31
FitQuadratic (class in fit), 31

FitSin (class in fit), 31
FitTanh (class in fit), 31
FitUnivariateSpline (class in fit), 31
FitXErf (class in fit), 31
FitXTanh (class in fit), 31
fix() (in module analyze), 25
fix_boolean() (in module regex), 8
fix_name() (in module regex), 8
fix_traps() (checkpoint.CheckPoint method), 10
from_pkls() (ivline.IVLine class method), 25
from_search() (ivline.IVLine class method), 25

## G

grep() (in module common), 7
Grid (class in grid), 20
grid (module), 20
gyroid() (in module surface), 15

## I

IndexMapper (class in grid), 22
indexS() (grid.IndexMapper method), 23
indexX() (grid.IndexMapper method), 23
indexY() (grid.IndexMapper method), 23
indexZ() (grid.IndexMapper method), 23
IVLine (class in ivline), 25
ivline (module), 25
IVLineS (class in ivline), 26

## K

keySets() (common.DictDiffer static method), 6

## L

linspace() (grid.Grid class method), 22
load() (checkpoint.CheckPoint method), 10
load() (in module checkpoint), 10
load() (in module parameters), 13
load() (parameters.Parameters method), 11
load_dat() (in module analyze), 25
load_dat() (in module datfile), 14
load_last() (in module checkpoint), 11
load_pkl() (in module analyze), 25
load_pkl() (in module datfile), 14
load_pkls() (in module analyze), 25
load_pkls() (ivline.IVLine method), 25

## M

maxbrute() (fit.Fit method), 28
maximize() (fit.Fit method), 29
Mesh (class in grid), 23
minimize() (fit.Fit method), 29
mtext() (in module plot), 27

## N

number() (in module regex), 8
numbers() (in module regex), 8

## P

paint_checkerboard_xy() (in module surface), 15
paint_checkerboard_yz() (in module surface), 16
paint_cube() (in module surface), 16
paint_plane_xy() (in module surface), 16
paint_plane_xz() (in module surface), 17
paint_plane_yz() (in module surface), 17
paint_square_xy() (in module surface), 17
paint_square_xz() (in module surface), 17
paint_square_yz() (in module surface), 18
paint_stripe_dx() (in module surface), 18
paint_stripe_dy() (in module surface), 18
paint_stripe_dz() (in module surface), 18
parameter() (in module common), 7
Parameters (class in parameters), 11
parameters (module), 11
Part (class in find), 8
part() (in module regex), 8
pkl() (ivline.IVLine method), 26
plot (module), 27
plot() (fit.Fit method), 29
plot_tangent() (fit.Fit method), 29
process_panel() (ivline.IVLine method), 26

## Q

q (ivline.units attribute), 27

## R

regex (module), 8
removedKeys() (common.DictDiffer class method), 6
removedValues() (common.DictDiffer class method), 6
rescale() (ivline.units static method), 27
reset() (checkpoint.CheckPoint method), 10
reset_output_parameters() (parameters.Parameters method), 11
results() (ivline.IVLine method), 26
results() (ivline.IVLineS method), 26
Run (class in find), 8
run() (in module regex), 8

## S

save() (checkpoint.CheckPoint method), 10
save() (parameters.Parameters method), 11
save_pkl() (in module analyze), 25
save_pkl() (in module datfile), 14
scherk_first_surface() (in module surface), 19
schwarz_d_surface() (in module surface), 19
schwarz_p_surface() (in module surface), 19
set_defaults() (parameters.Parameters method), 12
setup() (grid.Grid method), 22
show3D() (in module surface), 19

showXY() (in module surface), 20
Sim (class in find), 9
sim() (in module regex), 8
slice_part() (in module find), 9
slice_path() (in module find), 9
slice_run() (in module find), 9
slice_sim() (in module find), 9
solve() (fit.Fit method), 29
sort() (fit.Fit method), 29
Stats (class in ivline), 26
strip_comments() (in module regex), 8
summary() (fit.Fit method), 29
summary() (fit.FitCos method), 30
summary() (fit.FitErf method), 30
summary() (fit.FitGaussian method), 30
summary() (fit.FitLinear method), 31
summary() (fit.FitPower method), 31
summary() (fit.FitQuadratic method), 31
summary() (fit.FitSin method), 31
summary() (fit.FitTanh method), 31
summary() (fit.FitXErf method), 31
summary() (fit.FitXTanh method), 32
surface (module), 14

## T

tail() (in module common), 7
tangent() (fit.Fit method), 30
test() (ivline.IVLineS method), 26
text() (fit.Fit method), 30
timestamp() (in module common), 7
to_ndarray() (parameters.Parameters method), 13
to_units() (ivline.units static method), 27
transformA() (in module plot), 28
transformX() (in module plot), 28
transformY() (in module plot), 28

## U

unchangedKeys() (common.DictDiffer class method), 6
unchangedValues() (common.DictDiffer class method), 6
units (class in ivline), 27

## V

voltage() (in module regex), 8
vtk() (grid.Grid class method), 22

## W

WaveDimensions (class in surface), 14

## X

XYZV (class in grid), 24

## Z

zgrep() (in module common), 7
zhandle() (in module common), 7
ztail() (in module common), 7