Lysis

Brittany Bannish <bbannish@uco.edu> & Brad Paynter <bpaynter

Sep 23, 2024

CONTENTS:

CHAPTER

ONE

LYSIS

1.1 lysis package

1.1.1 Subpackages

lysis.util package

Submodules

lysis.util.constants module

class lysis.util.constants.**BoundaryCondition**(*value*)

Bases: Enum

An enumeration.

CONTINUING = 2

PERIODIC = 1

REFLECTING = 0

class lysis.util.constants.**BoundaryDirection**(*value*)

Bases: IntEnum

An enumeration.

BACK = 5

BOTTOM = 1

FRONT = 4

LEFT = 2

RIGHT = 3

TOP = 0

class lysis.util.constants.**Const**

Bases: object

class lysis.util.constants.**ExpComponent**(*value*)

Bases: Flag

An enumeration.

ALL = 15

MACRO = 4

MACRO_POSTPROCESSING = 8

MICRO = 1

MICRO_POSTPROCESSING = 2

NONE = 0

class lysis.util.constants.**FiberDirection**(*value*)

Bases: Enum

An enumeration.

DOWN = -1

IN = -3

LEFT = 2

OUT = 3

RIGHT = -2

UP = 1

class lysis.util.constants.**Neighbors**

Bases: object

class lysis.util.constants.**RandomDraw**(*value*)

Bases: IntEnum

An enumeration.

BINDING_TIME_WHEN_MOVING = 1

BINDING_TIME_WHEN_UNBINDING = 0

CONFLICT_RESOLUTION = 6

LYSIS_TIME = 5

MICRO_UNBIND = 2

MOVE = 3

RESTRICTED_MOVE = 7

UNBINDING_TIME = 4

lysis.util.datastore module

class lysis.util.datastore.**DataStatus**(*value*)

Bases: Flag

An enumeration.

INITIALIZED = 1

LOADED = 2

NONE = 0

SAVED = 4

class lysis.util.datastore.**DataStore**(*path: str | bytes | PathLike*, *filenames: Mapping | None = None*)

Bases: object

append(*key: AnyStr*, *value: Any*, *axis: int | None = None*)

delete(*key: AnyStr*)

load_from_disk(*key: AnyStr*)

new(*key: AnyStr*, *filename: AnyStr*)

overwrite(*key: AnyStr*, *value: Any*)

save_to_disk(*key: AnyStr*)

status(*key: AnyStr*) → *DataStatus*

to_dict() → Mapping

lysis.util.datastore.**identity**(*x: Any*)

lysis.util.kiss module

Minimal skeleton for using the Marsaglia KISS generator

Wrapper for C module.

Example Usage:

- **Initialize**

```
>>> kiss = KissRandomGenerator()
>>> kiss.seed(123)
```

- **Generate random integer 0..(2^32)-1**

```
>>> kiss.kiss32()
```

- **Generate random U[0,1]**

```
>>> kiss.random()
```

class lysis.util.kiss.**KissRandomGenerator**(*seed: int | None = None*)

Bases: object

A Pseudo-random Number Generator.

Wrapper class for kiss.so C module

getstate() → Tuple[int, int, int, int]

Returns the current state of the Random Number Generator.

integers(*bottom: int*, *top: int | None = None*, *size: int | None = None*) → int | ndarray

kiss32() → int

Returns the next 32-bit unsigned integer from the pseudo-random number stream. This is distributed uniformly from 0 through (2^32)-1.

mscw() → int

Returns a pseudo-random 32-bit unsigned integer (i.e., [0..(2^32)-1]). This is based on the system clock and NOT the current seed or state.

random(*size: int | None = None*) → float | ndarray

Returns the next double-precision random number from the pseudo-random number stream. This is distributed uniformly from zero through one.

seed(*seed: int | None = None*)

Sets the seed for the Random Number Generator.

Parameters

seed – The seed.

Note: While Python will accept any 64-bit integer (-9,223,372,036,854,775,806 through 9,223,372,036,854,775,807), this will be converted to an unsigned 32-bit integer (0 through 4,294,967,295) when passed to the underlying C code with unpredictable results.

setstate(*state: Tuple[int, int, int, int]*)

Sets the state of the Random Number Generator.

Parameters

state – A tuple of four integers.

Note: While Python will accept any 64-bit integer (-9,223,372,036,854,775,806 through 9,223,372,036,854,775,807), these will be converted to unsigned 32-bit integers (0 through 4,294,967,295) when passed to the underlying C code with unpredictable results.

lysis.util.parameters module

Code for holding, storing, and reading information about an experiment

This module gives a uniform way to handle the data and parameters of a given experiment. It contains classes to house these and make them accessible to the rest of the code. It also handles the storing and reading of parameters and data to/from disk.

Typical usage example:

```
>>> # Create a new experiment
>>> exp = Experiment('path/to/data')
>>> param = {'override_parameter': 2.54, 'another_new_parameter': 32}
>>> exp.initialize_macro_param(param)
```

(continues on next page)

(continued from previous page)

```
>>> exp.to_file()
>>> # Load an existing experiment
>>> exp = Experiment('path/to/data', '2022_12_27_1100')
>>> exp.read_file()
>>> # Access a parameter
>>> exp.macro_params.pore_size
>>> # Access data
>>> exp.data.lysis_time[4][18]
```

class lysis.util.parameters.**Experiment**(*data_root: str | bytes | PathLike*, *experiment_code: str | None = None*)

Bases: `object`

Houses all information about a given experimental run.

This object contains:

- Data location
- Experiment parameters
- Experiment data

It includes methods for

- Initializing with default parameters
- Reading parameters from disk
- Saving parameters to disk
- Reading input data from disk
- Writing result data to disk

Parameters

- **data_root** – The path of the folder containing datasets
- **experiment_code** – The code number of the experiment. This will be the name of the folder containing the data specific to this experiment. This should be a date and time in 'YYYY-MM-DD-hhmm' format If no code is given, one will be generated from the current date and time.

experiment_code

The code number of the experiment.

Type
str

os_path

The path to the folder containing this experiment's data

Type
str

macro_params

A dictionary of

Type
DataClass

Raises
RuntimeError – An invalid data folder was given.

initialize_macro_param(*params: dict[str, Any] | None = None*) → None

Creates the parameters for the Macroscale model.

Parameters are set to the default values unless new values are passed in the params dictionary.

This method is essentially a wrapper for the MacroParameters constructor.

Parameters
params –

A dictionary of parameters that differ from the default
values.

For example,

```
>>> {'binding_rate': 10, 'pore_size': 3,}
```

initialize_micro_param(*params: Mapping[str, Any] | None = None*) → None

Creates the parameters for the Microscale model.

Parameters are set to the default values unless new values are passed in the params dictionary.

This method is essentially a wrapper for the MicroParameters constructor.

Parameters
params –

A dictionary of parameters that differ from the default
values.

For example,

```
>>> {'binding_rate': 10, 'pore_size': 3,}
```

read_file() → None

Load the experiment parameters from disk.

Raises
RuntimeError – No parameter file is available for this experiment.

to_dict() → dict

Returns the internally stored data as a dictionary.

Does not include system-specific information like paths.

to_file() → None

Stores the experiment parameters to disk.

Creates or overwrites the params.json file in the experiment's data folder. This file will contain the current experiment parameters (including any micro- and macroscale parameters) in JSON format.

class lysis.util.parameters.**MacroParameters**(*pore_size: ~pint.registry.Quantity = <Quantity(1.0135, 'micrometer')>, diffusion_coeff: ~pint.registry.Quantity = <Quantity(5e-07, 'centimeter ** 2 / second')>, forced_unbind: float = 0.0852, average_bound_time: ~pint.registry.Quantity = <Quantity(27.8, 'second')>, cols: int = 93, rows: int = 121, empty_rows: int = 28, total_molecules: int = 43074, moving_probability: float = 0.2, simulations: int = 10, total_time: ~pint.registry.Quantity = <Quantity(20, 'minute')>, seed: int = 0, save_interval: ~pint.registry.Quantity = <Quantity(10, 'second')>, macro_version: str = 'diffuse_into_and_along', log_lvl: int = 30, duplicate_fortran: bool = False, processing_library: str = 'numpy'*)

Bases: `object`

Contains parameters for the Macroscale model.

Parameters can be accessed as attributes. Independent parameters should only be set at initialization. Dependent parameters should never be set manually, but are automatically calculated by internal code.

Should only be used inside an Experiment object.

Example

```
>>> # Initialize using the default values
>>> macro_params_default = MacroParameters()
>>> # Get parameter value
>>> macro_params_default.pore_size
1.0135e-4
>>> # Initialize overriding some default values
>>> p = {'binding_rate': 10, 'pore_size': 3}
>>> macro_params_override = MacroParameters(**p)
```

average_bound_time: Quantity = <Quantity(27.8, 'second')>

This is the average time a tPA molecule stays bound to fibrin. For now I'm using 27.8 to be 1/0.036, the value in the absence of PLG.

Units
seconds

Fortran
avgwait = 1/koff

cols: int = 93

The number of lattice nodes in each (horizontal) row

Units
None

Fortran
N

diffusion_coeff: Quantity = <Quantity(5e-07, 'centimeter ** 2 / second')>

Diffusion coefficient

Units
cm^2/s

Fortran
Diff

`duplicate_fortran: bool = False`

Whether the Python code should follow the Fortran code step-by-step. Theoretically, with this set to "True", both sets of code will produce the exact same output. This will impact performance negatively. This currently does nothing.

Units
None

Fortran
None

`empty_edges: int`

The number of edges without fibrin. Also the 1-D index of the last edge without fibrin when 1-indexing

This is probably unnecessary when using a 2-D data structure, but is kept for historical reasons.

Units
None

Fortran
enoFB

`empty_rows: int = 28`

The number of fibrin-free rows at the top of the grid.

Equivalent to 'first_fiber_row', which is the 1st node in vertical direction containing fibers. So if first_fiber_row = 10, then rows 0-9 have no fibers, there's one more row of fiber-free planar vertical edges, and then the row with index 'first_fiber_row' (e.g. 11th) is a full row of fibers.

Units
None

Fortran
Ffree-1

`fiber_rows: int`

The number of rows containing fibrin

Units
None

Fortran
Fhat

`forced_unbind: float = 0.0852`

Fraction of times tPA was forced to unbind in microscale model.

Units
None

Fortran
frac_forced

static fortran_names()

Returns a dictionary whose keys are the names of all parameters (both micro- and macroscale) that have equivalents in Fortran. The values in the dictionary are the names of the equivalent Fortran variable names. These values are parsed from the docstrings in this file.

full_row: int

Edges in a full row of nodes

Units
None

Fortran
None

input_data: List[str]

The data (from the Microscale model) required to run the Macroscale model.

log_lvl: int = 30

How much debugging information to write out to the console

Units
None

Fortran
None

macro_version: str = 'diffuse_into_and_along'

A string identifying which version of the Macroscale model is being run. This string was included in data filenames stored by the Fortran code.

moving_probability: float = 0.2

The probability of moving.

Make sure it is small enough that we've converged.

Units
None

Fortran
q

number_of_saves: int

The number of times data will be saved from the model.

Units
None

Fortran
nplt

output_data: List[str]

The data output by the Macroscale model.

pore_size: Quantity = <Quantity(1.0135, 'micrometer')>

Pore size (distance between fibers/nodes)

Units
centimeters

Fortran
delx

static print_default_values() → str

Returns the default parameters for the Macroscale model.

processing_library: str = 'numpy'

Which library the macroscale model should use for processing. Options include

- 'numpy'
- 'cupy'

Units
None

Fortran
None

rows: int = 121

The number of lattice nodes in each (vertical) column

Units
None

Fortran
F

save_interval: Quantity = <Quantity(10, 'second')>

How often to record data from the model.

Units
sec

Fortran
save_interval

seed: int = 0

Seed for the random number generator

Units
None

Fortran
seed

simulations: int = 10

The number of independent simulations to be run

Units
None

Fortran
stats

state: Tuple[int, int, int, int]

State for the random number generator.

Units
None

Fortran
state

time_step: float

The length of one timestep.

Units
seconds

Fortran
tstep

total_edges: int

The total number of edges in the model

Units
None

Fortran
num

total_fibers: int

The total number of fibers in the model

Units
None

Fortran
None

total_molecules: int = 43074

The total number of tPA molecules:

- 43074 is Colin's [tPA]=0.6 nM
- 86148 is Colin's [tPA]=1.2 nM

Units
None

Fortran
M

total_time: Quantity = <Quantity(20, 'minute')>

Total running time for model.

Units
seconds

Fortran
tf

total_time_steps: int

The total number of timesteps.

Units
None

Fortran
num_t

static units()

Returns a dictionary whose keys are the names of all parameters (both micro- and macroscale) that have units. The values in the dictionary are those units. These values are parsed from the docstrings in this file.

xz_row: int

Number of all x- and z-edges in a row

Units
None

Fortran
None

class lysis.util.parameters.**MicroParameters**(*fibrinogen_length: ~pint.registry.Quantity = <Quantity(45, 'nanometer')>, fibrinogen_radius: ~pint.registry.Quantity = <Quantity(1.2, 'nanometer')>, fiber_radius: ~pint.registry.Quantity = <Quantity(36.35, 'nanometer')>, diss_const_tPA_wPLG: ~pint.registry.Quantity = <Quantity(0.02, 'micromolar')>, diss_const_tPA_woPLG: ~pint.registry.Quantity = <Quantity(0.36, 'micromolar')>, diss_const_PLG_intact: ~pint.registry.Quantity = <Quantity(38, 'micromolar')>, diss_const_PLG_nicked: ~pint.registry.Quantity = <Quantity(2.2, 'micromolar')>, bind_rate_tPA: ~pint.registry.Quantity = <Quantity(0.1, '1 / micromolar / second')>, bind_rate_PLG: ~pint.registry.Quantity = <Quantity(0.1, '1 / micromolar / second')>, conc_free_PLG: ~pint.registry.Quantity = <Quantity(2, 'micromolar')>, deg_rate_fibrin: ~pint.registry.Quantity = <Quantity(5.0, '1 / second')>, unbind_rate_PLi: ~pint.registry.Quantity = <Quantity(57.6, '1 / second')>, activation_rate_PLG: ~pint.registry.Quantity = <Quantity(0.1, '1 / second')>, exposure_rate_binding_site: ~pint.registry.Quantity = <Quantity(5.0, '1 / second')>, nodes_in_row: int = 7, snap_proportion: float = 0.6666666666666666, simulations: int = 50000, seed: int = 0, micro_version: str = 'micro_rates', log_lvl: int = 30*)

Bases: `object`

This will contain the parameters for the Microscale model.

Parameters can be accessed as attributes. Independent parameters should only be set at initialization. Dependent parameters should never be set manually, but are automatically calculated by internal code.

Should only be used inside an Experiment object.

Example

```
>>> # Initialize using the default values
>>> micro_params_default = MicroParameters()
>>> # Get parameter value
>>> micro_params_default.fibrinogen_length
45 nanometers
>>> # Initialize overriding some default values
>>> p = {'fibrinogen_radius': "10 nm", 'fiber_radius': "3 um"}
>>> micro_params_override = MicroParameters(**p)
```

activation_rate_PLG: Quantity = <Quantity(0.1, '1 / second')>

The catalytic rate constant, k_{cat}^{ap}, for activation of PLG into PLI.

Units
sec^-1

Fortran
kapcat

bind_rate_PLG: Quantity = <Quantity(0.1, '1 / micromolar / second')>

The binding rate of PLG, $k^{\text{on}}_{\text{PLG}}$, to fibrin.

Units
(micromolar*sec)^-1

Fortran
kPLGon

bind_rate_tPA: Quantity = <Quantity(0.1, '1 / micromolar / second')>

The binding rate of tPA, $k^{\text{on}}_{\text{tPA}}$, to fibrin.

Units
(micromolar*sec)^-1

Fortran
ktPAon

binding_sites: Quantity

Concentration of binding sites.

Units
micromolar

Fortran
bs

conc_free_PLG: Quantity = <Quantity(2, 'micromolar')>

The concentration of free plasminogen.

Units
micromolar

Fortran
freeplg

deg_rate_fibrin: Quantity = <Quantity(5.0, '1 / second')>

The plasmin-mediated rate of fibrin degradation.

Units
sec^-1

Fortran
kdeg

diss_const_PLG_intact: Quantity = <Quantity(38, 'micromolar')>

The dissociation constant of PLG, k^{D}_{PLG}, to intact fibrin.

Units
micromolar

Fortran
KdPLGintact

diss_const_PLG_nicked: Quantity = <Quantity(2.2, 'micromolar')>

The dissociation constant of PLG, k^D_{PLG}, to nicked fibrin.

Units
micromolar

Fortran
KdPLGnicked

diss_const_tPA_wPLG: Quantity = <Quantity(0.02, 'micromolar')>

The dissociation constant of tPA, k^D_{tPA}, to fibrin in the presence of PLG.

Units
micromolar

Fortran
KdtPAyesplg

diss_const_tPA_woPLG: Quantity = <Quantity(0.36, 'micromolar')>

The dissociation constant of tPA, k^D_{tPA}, to fibrin in the absence of PLG.

Units
micromolar

Fortran
KdtPAnoplg

exposure_rate_binding_site: Quantity = <Quantity(5.0, '1 / second')>

The catalytic rate constant, $k^{\text{n}}_{\text{cat}}$, for the PLi-mediated rate of exposure of new binding sites.

Units
sec^-1

Fortran
kncat

fiber_radius: Quantity = <Quantity(36.35, 'nanometer')>

The radius of each fiber in the model.

Units
microns

Fortran
radius

fibrin_conc_per_fiber: Quantity

The concentration of fibrin in each fiber

Units
micromolar

Fortran
None

fibrinogen_length: Quantity = <Quantity(45, 'nanometer')>

The length of a fibronogen molecule.

Units
microns

Fortran
None

fibrinogen_radius: Quantity = <Quantity(1.2, 'nanometer')>

The radius of a fibronogen molecule.

Units
microns

Fortran
None

static fortran_names()

Returns a dictionary whose keys are the names of all parameters (both micro- and macroscale) that have equivalents in Fortran. The values in the dictionary are the names of the equivalent Fortran variable names. These values are parsed from the docstrings in this file.

log_lvl: int = 30

How much debugging information to write out to the console

Units
None

Fortran
None

micro_version: str = 'micro_rates'

A string identifying which version of the Microscale model is being run.

nodes_in_row: int = 7

The number of protofibrils in one row of the lattice inside one fiber.

Units
None

Fortran
nodes

output_data: List[str]

The data output by the Microscale model.

static print_default_values() → str

Returns the default parameters for the Microscale model.

protein_per_fiber: Quantity

The fraction of protein in each fiber (by volume?)

Units
%

Fortran
None

protofibril_radius: Quantity

The radius of a protofibril.

Units
microns

Fortran
None

`seed: int = 0`

Seed for the random number generator

Units
None

Fortran
seed

`simulations: int = 50000`

The number of independent trials run in the microscale model.

Units
None

Fortran
runs

`snap_proportion: float = 0.6666666666666666`

The proportion of doublets that need to be degraded before the fiber snaps.

Units
None

Fortran
snap_proportion

`unbind_rate_PLG_intact: Quantity`

The unbinding rate of PLG, $k_{\text{PLG}}^{\text{off}}$, from intact fibrin.

Units
sec^-1

Fortran
kplgoff

`unbind_rate_PLG_nicked: Quantity`

The unbinding rate of PLG, $k_{\text{PLG}}^{\text{off}}$, from nicked fibrin.

Units
sec^-1

Fortran
kplgoffnick

`unbind_rate_PLi: Quantity = <Quantity(57.6, '1 / second')>`

The unbinding rate of PLi, $k_{\text{PLi}}^{\text{off}}$, from fibrin.

Units
sec^-1

Fortran
kplioff

`unbind_rate_tPA_wPLG: Quantity`

The unbinding rate of tPA, $k_{\text{tPA}}^{\text{off}}$, from fibrin in the presence of PLG.

Units
sec^-1

Fortran
kaoff12

unbind_rate_tPA_woPLG: Quantity

The unbinding rate of tPA, $k_{\text{tPA}}^{\text{off}}$, from fibrin in the absence of PLG.

Units

sec^-1

Fortran

kaoff10

static units()

Returns a dictionary whose keys are the names of all parameters (both micro- and macroscale) that have units. The values in the dictionary are those units. These values are parsed from the docstrings in this file.

lysis.util.util module

lysis.util.util.**dict_to_formatted_str**(*d: Mapping[AnyStr, Any]*) → str

Converts a dictionary into a formatted, JSON-like string.

Align keys and values, including the alignment of sub-dicts.

e.g.,

```
>>> measurements = {'Hat': 'Large', 'Pants': {'Waist': 40, 'Inseam': 38}}
>>> dict_to_formatted_str(measurements)
Hat   : Large
Pants : Waist  : 40
        Inseam : 38
```

Parameters

d (*dict*) – A dictionary with string-like keys.

Module contents

1.1.2 Submodules

1.1.3 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

G

I

K

L

M

X