API Documentation

API Documentation

May 29, 2007

Contents

1 Package Epigrass

1.1 Modules

- **CustomModel** *(Section 2, p. 3)*
- **about** *(Section 3, p. 4)*
- **cpanel** *(Section 4, p. 5)*
- **dataObject** *(Section 5, p. 6)*
- **data_io**: This module contains functions to read and write from ascii files, as well as to/from MySQL databases.
  (Section 6, p. 7)
- **dgraph**: This module is a graph visualizing tool.
  (Section 7, p. 8)
- **epigrass** *(Section 8, p. 11)*
- **epiplay** *(Section 9, p. 14)*
- **manager**: Model Management and simulation objects.
  (Section 10, p. 16)
- **report**: This module generates a report of the network simulation model using LaTeX.
  (Section 11, p. 19)
- **rinterface**: This module is a collection of fucntions to interface with "R":http://cran.r-project.org
  (Section 12, p. 21)
- **simobj**: This Module contains the definitions of objects for spatial simulation on geo reference spaces.
  (Section 13, p. 22)
- **spread** *(Section 14, p. 38)*
- **testinst**: Unit testing script
  (Section 15, p. 39)

2 Module Epigrass.CustomModel

2.1 Functions

Model(*self*, *vars*, *par*, *theta*=0, *npass*=0)

```
Calculates the model SIR, and return its values.
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

3 Module Epigrass.about

3.1 Variables

Name	Description

image0_data	**Value:**
	`"\x89\x50\x4e\x47\x0d\x0a\x1a\x0a\x00\x00\x00\x0d"`
	`"\x49\...`

3.2 Class aboutDialog

??-2 ─┐
Epigrass.about.aboutDialog

3.2.1 Methods

__init__(*self*, *parent*=`None`, *name*=`None`, *modal*=`0`, *fl*=`0`)

languageChange(*self*)

4 Module Epigrass.cpanel

4.1 Variables

Name	Description

image0_data	**Value:**
	`"\x89\x50\x4e\x47\x0d\x0a\x1a\x0a\x00\x00\x00\x0d"`
	`"\x49\...`

4.2 Class MainPanel

```
??-5 ─┐
      │
     Epigrass.cpanel.MainPanel
```

4.2.1 Methods

__init__(*self*, *parent*=`None`, *name*=`None`, *fl*=`0`)

languageChange(*self*)

Main_layout_destroyed(*self*, *a0*)

5 Module Epigrass.dataObject

5.1 Functions

Connect(*backend*, *user*, *pw*, *host*, *port*, *db*)

Initializes the connection.

5.2 Class Site

sqlobject.SQLObject

Epigrass.dataObject.Site

5.2.1 Class Variables

Name	Description
geocode	**Value:** `IntCol()`
time	**Value:** `IntCol()`
totpop	**Value:** `IntCol()`
name	**Value:** `UnicodeCol()`
lat	**Value:** `FloatCol()`
longit	**Value:** `FloatCol()`

5.3 Class Edge

sqlobject.SQLObject

Epigrass.dataObject.Edge

5.3.1 Class Variables

Name	Description
source_code	**Value:** `IntCol()`
dest_code	**Value:** `IntCol()`
time	**Value:** `IntCol()`
ftheta	**Value:** `FloatCol()`
btheta	**Value:** `FloatCol()`

6 Module Epigrass.data_io

This module contains functions to read and write from ascii files, as well as to/from MySQL databases.

6.1 Functions

load(*fname*, *sep*=`None`)

Load ASCII data from fname into an array and return the array.
The data must be regular, same number of values in every row
fname must be a filename.
optional argument sep must be set if the separators are not some kind of white space.
well suited for GRASS ascii site data.
Example usage:

x,y = load('test.dat') # data in two columns
X = load('test.dat') # a matrix of data
x = load('test.dat') # a single column of data
X = load('test.dat, sep = ',') # file is csv

loadData(*fname*, *sep*=`None`, *encoding*=`'latin-1'`)

Loads from ascii files with separated values and returns a list of lists.

queryDb(*usr*, *passw*, *db*, *table*, *host*=`'localhost'`, *port*=`3306`, *sql*=`''`)

Fetch the contents of a table on a MySQL database sql,usr,passw,db,table and host must be strings.
returns a tuple of tuples and a list of dictionaries.

loadEdgeData(*fname*)

getSiteFromGeo(*fin*=`None`, *fout*=`None`)

Search the locality and census databases tables for codes in a file and creates a sites file.

sitesToDb(*fname*, *table*, *db*=`'epigrass'`, *usr*=`'root'`, *passw*=`'mysql'`, *host*=`'localhost'`, *port*=`3306`)

Creates a site table on a mysql db from a sites file (fname).

7 Module Epigrass.dgraph

This module is a graph visualizing tool. It tries to resolves the graph layout by making an analogy of the nodes and edges to masses and springs. the nodes repel each other with a force inversely proportional to their distance, and the edges do the opposite.

7.1 Variables

Name	Description
rho	**Value:** `23.8732414637845`

7.2 Class Node

Physical model and visual representation of a node as a mass.

7.2.1 Methods

__init__(*self*, *m*, *pos*, *r*=`.1`, *fixed*=`0`, *pickable*=`1`, *v*=`visual.vector(0.,0.,0.)`, *color*=`(0.,1.,0.)`, *name*=`''`, ***keywords*)

Construct a mass.

showName(*self*, *t*)

Show the node name for t seconds

calcGravityForce(*self*, *g*)

Calculate the gravity force.

calcViscosityForce(*self*, *viscosity*)

Calculate the viscosity force.

calcNewLocation(*self*, *dt*)

Calculate the new location of the mass.

clearForce(*self*)

Clear the Force.

7.2.2 Class Variables

Name	Description
factor	**Value:** `3./(4* math.pi* rho)`

7.3 Class RubberEdge

Epigrass.dgraph._Edge

Epigrass.dgraph.RubberEdge

Visual representation of a spring using a single cylinder with variable radius.

7.3.1 Methods

__init__(*self*, *n0*, *n1*, *k*, *l0*=`None`, *damping*=`None`, *radius*=`None`, *color*=`(0.5,0.5,0.5)`, ***keywords*)
Construct a spring edge.

Overrides: Epigrass.dgraph._Edge.__init__ extit(inherited documentation)

update(*self*)

Update the visual representation of the spring.

calcDampingForce(*self*)

Calculate the damping force.

calcSpringForce(*self*)

Calculate the spring force.

7.4 Class Graph

The Graph.self.data(start)[5]

7.4.1 Methods

__init__(*self*, *timestep*, *oversample*=`1`, *gravity*=`1`, *viscosity*=`None`, *name*=`'EpiGrass Viewer'`, ***keywords*)

Construct a Graph.

insertNode(*self*, *node*)

Insert node into the system.

insertMap(*self*, *map*)

Insert map into the system.

insertNodeList(*self*, *nodelist*)

Insert all Nodes in nodelist into the system.

insertEdge(*self*, *edge*)

Insert edge into the system.

insertEdgeList(*self*, *edgelist*)

Insert all Edges in edgelist into the system.

getEdgeFromMatrix(*self*, *matrix*)

Extract edges from the adjacency matrix.

centerView(*self*)

advance(*self*)

Perform one Iteration of the system by advancing one timestep.

dispatchDnD(*self*)

Process the drag and drop interaction from the mouse.

step(*self*)

Perform one step. This is a convenience method. It actually calls dispatchDnD() and advance().

mainloop(*self*)

Start the mainloop, which means that step() is called in an infinite loop.

7.5 Class Map

7.5.1 Methods

__init__(*self*, *fname*)

Reader(*self*, *fname*)

Reads Grass 5.x ascii vector files.

dbound(*self*, *pol*)

Draws a polygon

8 Module Epigrass.epigrass

8.1 Functions

loadLang(*app*, *tr*, *lang*)

loads the language based on the user's choice.

main()

8.2 Class MainPanel_Impl

??-5
Epigrass.cpanel.MainPanel
Epigrass.epigrass.MainPanel_Impl

8.2.1 Methods

__init__(*self*, *parent*=None, *name*=None, *fl*=0)
Overrides: Epigrass.cpanel.MainPanel.__init__

initRc(*self*)

Initializes the epigrassrc file.

updateRc(*self*)

Updates the epigrassrc file with the contents of the configuration dictionary.

loadRcFile(*self*, *fname*, *config*={})

Loads epigrassrc. Returns a dictionary with keys of the form <section>.<option> and the corresponding values.

fillGui(*self*, *conf*)

Fill the gui fields from configuration dictionary conf.

checkConf(*self*)

Check if the configuration dictionary is complete

editScript(*self*)

Opens the model script with the user's preferred editor.

chooseScript(*self*)

starts the file selction dialog for the script.

readGui(*self*)

Gets the info from the GUI into the configuration dictionary.

onRunFuture(*self*)

Run the simulation on a separate thread

onRun(*self*)

starts the simulation when the run button is pressed

repRuns(*self*, *S*)

Do repeated runs

onDbBackup(*self*)

Dumps the epigrass databse to an sql file.

onDbInfo(*self*)

onHelp(*self*)

Opens the userguide

onRepOpen(*self*)

Opens the report PDF

onVisual(*self*)

Scan the epigrass database an shows available simulations.

getVariables(*self*)

Fill the variables list whenever a table is selected in the table list

onPlayButton(*self*)

Calls the epiplay module to replay the epidemic from a database table.

onConsensus(*self*)

Build the consensus tree

setBackend(*self*)

activate/deactivate db spec groupbox

onExit(*self*)

Close the gui

Main_layout_destroyed(*self, a0*)

languageChange(*self*)

8.3 Class Future

By David Perry - http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/84317 To run a function in a separate thread, simply put it in a Future:

```
>>> A=Future(longRunningFunction, arg1, arg2 ...)
```

It will continue on its merry way until you need the result of your function. You can read the result by calling the Future like a function, for example:

```
>>> print A()
```

If the Future has completed executing, the call returns immediately. If it is still running, then the call blocks until the function completes. The result of the function is stored in the Future, so subsequent calls to it return immediately.

A few caveats: Since one wouldn't expect to be able to change the result of a function, Futures are not meant to be mutable. This is enforced by requiring the Future to be "called", rather than directly reading __result. If desired, stronger enforcement of this rule can be achieved by playing with __getattr__ and __setattr__.

The Future only runs the function once, no matter how many times you read it. You will have to re-create the Future if you want to re-run your function; for example, if the function is sensitive to the time of day.

For more information on Futures, and other useful parallel programming constructs, read Gregory V. Wilson's _Practical Parallel Programming_.

8.3.1 Methods

__init__(*self, func, *param*)

__repr__(*self*)

__call__(*self*)

Wrapper(*self, func, param*)

9 Module Epigrass.epiplay

9.1 Class viewer

9.1.1 Methods

__init__(*self*, *host*='localhost', *port*=3306, *user*='epigrass', *pw*='epigrass', *db*='epigrass', *backend*='mysql', *encoding*='latin-1')

getTables(*self*)

Returns list of table names from current database connection

getFields(*self*, *table*)

Returns a list of fields (column names) for a given table. table is a string with table name

readNodes(*self*, *name*, *table*)

Reads geocode and coords from database table for each node and adjacency matrix.

readData(*self*, *table*)

read node time series data

readEdges(*self*, *table*)

Read edge time series

viewGraph(*self*, *nodes*, *am*, *var*, *mapa*='')

Starts the Vpython display of the graph.

anim(*self*, *data*, *edata*, *numbsteps*, *pos*, *rate*=20)

```
Starts the animation
- data: time series from database
- edata: infectious traveling for edge painting
- pos: column number of variable to animate
```

paintNode(*self*, *t*, *node*, *numbsteps*, *col*='r')

Paints red the box corresponding to the node in the visual display

plotTs(*self*, *ts*, *name*)

Uses gcurve to plot the time-series of a given city object

keyin(*self*, *data*, *edata*, *numbsteps*, *pos*, *rate*)

Implements keyboard and mouse interactions

10 Module Epigrass.manager

Model Management and simulation objects.

10.1 Functions

storeSimulation(*s*, *db*=`'epigrass'`, *host*=`'localhost'`, *port*=`3306`)

store the Simulate object *s* in the epigrass database to allow distributed runs. Currently not working.

10.2 Class simulate

This class takes care of setting up the model, simulating it, and storing the results

10.2.1 Methods

__init__(*self*, *fname*=`None`, *host*=`'localhost'`, *port*=`3306`, *db*=`'epigrass'`, *user*=`'epigrass'`, *password*=`'epigrass'`, *backend*=`'mysql'`)

loadModelScript(*self*, *fname*)

Loads the model specification from the text file. Returns a dictionary with keys of the form <section>.<option> and the corresponding values.

evalConfig(*self*, *config*)

Takes in the config dictionary and generates the global variables needed.

chkScript(*self*)

Checks the type of the variables on the script

deg2dec(*self*, *coord*)

converts lat/long to decimal

instSites(*self*, *sitelist*)

Instantiates and returns a list of siteobj instances, from a list of site specification as returned by data_io.loadData. Here the site specific events are passed to each site, and the site models are created.

instEdges(*self*, *sitelist*, *edgelist*)

Instantiates and returns a list of edge objects,
sitelist – list of siteobj objects.
edgelist – list of edge specifications as returned by data_io.loadData.

instGraph(*self*, *name*, *digraph*, *siteobjlist*, *edgeobjlist*)

Instantiates and returns a graph object from a list of edge objects.

randomizeSeed(*self*)

Generate and return a list of randomized seed sites to be used in a repeated runs scenario. Seed sites are sampled with a probability proportional to the log of their population size.

setSeed(*self*, *seed*, *n*=1)

Resets the number of infected to zero in all nodes but seed, which get n infected cases. seed must be a siteobj object.

start(*self*)

Start the simulation

createDataFrame(*self*, *site*)

Saves the time series *site.ts* to outfile as specified on the model script.

grassVect2ascii(*self*, *layer*)

Convert Grass vector layer to an ascii file epigrass can read

outToCsv(*self*, *table*)

Save simulation results in csv file.

outToODb(*self*, *table*, *mode*='b')

Insert simulation results in a database using sqlobject. if mode = b, do batch inserts if mode = p, do parallel inserts

batchInsert(*self*)

Do the inserts all at once. After the simulation is done

parInsert(*self*)

do the insertions in a separate process.

outToDb(*self*, *table*)

Insert simulation results on a mysql table

criaAdjMatrix(*self*)

dumpData(*self*)

Dumps data as csv (comma-separated-values)

saveModel(*self*, *fname*)

Save the fully specified graph.

loadModel(*self*, *fname*)

Loads a pre-saved graph.

fins(*self*)

call parallel inserts

runGraph(*self*, *graphobj*, *iterations*=1, *transp*=0)

Starts the simulation on a graph.

Say(*self*, *string*)

Exits outputs messages to the console or the gui accordingly

10.3 Class Bunch

class to store a bunch os variables

10.3.1 Methods

__init__(*self*, ***kw*)

10.4 Class Tree

Tree object representing the spread of an epidemic

10.4.1 Methods

__init__(*self*, *Simulation*)

writeNexus(*self*)

Writes a Nexus file containing(.nex) the phylogeographical tree of the epidemic.

getVenn(*self*)

” Generates Venn Diagram from epipath.

11 Module Epigrass.report

This module generates a report of the network simulation model using LaTeX.

11.1 Variables

Name	Description
header	**Value:** `r...`

11.2 Class report

Generates reports in pdf format. Takes as input arg. a simulation object.

11.2.1 Methods

__init__(*self*, *simulation*)

genNetTitle(*self*)

Generates title section of the preamble from data extracted from the simulation.

genEpiTitle(*self*)

Generates title section of the preamble from data extracted from the simulation.

genFullTitle(*self*)

Generates title section of the preamble from data extracted from the simulation.

graphDesc(*self*)

Generates the Graph description section.

siteReport(*self*, *geoc*)

Puts together a report for a given site.

genSiteEpi(*self*, *geoc*)

Generate epidemiological reports at the site level.

genEpi(*self*)

Generate epidemiological report.

Assemble(*self*, *type*)

Assemble the type of report desired types: 1: network only 2: epidemiological only 3: both

Say(*self*, *string*)

Exits outputs messages to the console or the gui accordingly

savenBuild(*self*, *name*, *src*)

Saves the LaTeX in a newly created directory and builds it.

12 Module Epigrass.rinterface

This module is a collection of fucntions to interface with ”R”:http://cran.r-project.org

You need to have R and the following libraries installed for it to work:

*RMySQL

*DBI

*lattice

12.1 Functions

qnplot(*table*)

This function query simulation results stored in a MySQL table and generates some plots.

13 Module Epigrass.simobj

This Module contains the definitions of objects for spatial simulation on geo reference spaces.

13.1 Class siteobj

Basic site object containing attributes and methods common to all site objects.

13.1.1 Methods

__init__(*self*, *name*, *initpop*, *coords*, *geocode*, *values*=`()`)

Set initial values for site attributes.
-name: name of the locality
-coords: site coordinates.
-initpop: total population size.
-geocode: integer id code for site
-values: Tuple containing adicional values from the sites file

createModel(*self*, *init*, *par*, *modtype*=`'SEIR'`, *name*=`'model1'`, *v*=`[]`, *bi*=`None`, *bp*=`None`)

Creates a model of type modtype and defines its initial parameters. init – initial conditions for the state variables tuple with fractions of the total population in each category (state variable). par – initial values for the parameters. v – List of extra variables passed in the sites files bi, bp – dictionaries containing all the inits and parms defined in the .epg model

runModel(*self*)

Iterate the model

vaccinate(*self*, *cov*)

At time t the population will be vaccinated with coverage cov.

intervention(*self*, *par*, *cov*, *efic*)

From time t on, parameter par is changed to par * (1-cov*efic)

getTheta(*self*, *npass*, *delay*)

Returns the number of infected individuals in this site commuting through the edge that called this function.
npass – number of individuals leaving the node.

getThetaindex(*self*)

Returns the Theta index. Measures the function of a node, that is the average amount of traffic per intersection. The higher theta is, the greater the load of the network.

receiveTheta(*self*, *thetai*, *npass*, *sname*)

Number of infectious individuals arriving from site i

plotItself(*self*)

plot site timeseries

isNode(*self*)

find is given site is a node of a graph

getNeighbors(*self*)

Returns a dictionary of neighbooring sites as keys, and distances as values.

getDistanceFromNeighbor(*self*, *neighbor*)

Returns the distance in Km from a given neighbor. neighbor can be a siteobj object, or a geocode number

getDegree(*self*)

Returns the degrees of this site if it is part of a graph. The order (degree) of a node is the number of its attached links and is a simple, but effective measure of nodal importance.
The higher its value, the more a node is important in a graph as many links converge to it. Hub nodes have a high order, while terminal points have an order that can be as low as 1.
A perfect hub would have its order equal to the summation of all the orders of the other nodes in the graph and a perfect spoke would have an order of 1.

doStats(*self*)

Calculate indices describing the node and return them in a list.

getCentrality(*self*)

Also known as closeness. A measure of global centrality, is the inverse of the sum of the shortest paths to all other nodes in the graph.

getBetweeness(*self*)

Is the number of times any node figures in the the shortest path between any other pair of nodes.

13.2 Class popmodels

Defines a library of discrete time population models

13.2.1 Methods

__init__(*self*, *parentsite*, *par*, *type*=`'SIR'`, *v*=`[]`, *bi*=`None`, *bp*=`None`)

defines which models a given site will use and set variable names accordingly.

selectModel(*self*, *type*)

sets the model engine

multipleStep(*self*, *inits*, *par*, *theta*=`0`, *npass*=`0`, *modelos*=`[]`)

```
Run multiple models on a single site
- Inits and par are a list of lists.
- modelos is a list of of the modeltypes.
```

stepFlu(*self*, *vars*, *par*, *theta*=`0`, *npass*=`0`)

Flu model with classes S,E,I subclinical, I mild, I medium, I serious, deaths

stepSIS(*self*, *inits*, *par*, *theta*=`0`, *npass*=`0`)

```
calculates the model SIS, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIS_s(*self*, *inits*=`(0,0,0)`, *par*=`(0.001,1,1,0.5,1,0,0,0)`, *theta*=`0`, *npass*=`0`, *dist*=`'poisson'`)

```
Defines an stochastic model SIS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIR(*self*, *inits*, *par*, *theta*=`0`, *npass*=`0`)

```
calculates the model SIR, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIR_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*=’poisson’)

```
Defines an stochastic model SIR:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIS(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0)

```
Defines the model SEIS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIS_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*=’poisson’)

```
Defines an stochastic model SEIS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIR(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0)

```
Defines the model SEIR:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIR_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*=’poisson’)

```
Defines an stochastic model SEIR:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIpRpS(*self*, *inits*, *par*, *theta*=0, *npass*=0)

```
calculates the model SIpRpS, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIpRpS_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*='poisson')

```
Defines an stochastic model SIpRpS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIpRpS(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0)

```
Defines the model SEIpRpS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIpRpS_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*='poisson')

```
Defines an stochastic model SEIpRpS:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIpR(*self*, *inits*, *par*, *theta*=0, *npass*=0)

```
calculates the model SIpR, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIpR_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*='poisson')

```
Defines an stochastic model SIpRs:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIpR(*self*, *inits*, *par*, *theta*=0, *npass*=0)

```
calculates the model SEIpR, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSEIpR_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*='poisson')

```
Defines an stochastic model SEIpRs:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIRS(*self*, *inits*, *par*, *theta*=0, *npass*=0)

```
calculates the model SIRS, and return its values (no demographics)
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B, w, p) see docs.
- theta = infectious individuals from neighbor sites
```

stepSIRS_s(*self*, *inits*=(0,0,0), *par*=(0.001,1,1,0.5,1,0,0,0), *theta*=0, *npass*=0, *dist*='poisson')

```
Defines an stochastic model SIR:
- inits = (E,I,S)
- par = (Beta, alpha, E,r,delta,B,w,p) see docs.
- theta = infectious individuals from neighbor sites
```

13.3 Class edge

Defines an edge connecting two nodes (node source to node dest). with attributes given by value.

13.3.1 Methods

__init__(*self*, *source*, *dest*, *fmig*=0, *bmig*=0, *Leng*=0)

Main attributes of *Edge*.
source – Source site object.
dest – Destination site object.
fmig – forward migration rate in number of indiv./day.
bmig – backward migration rate in number of indiv./day.
Leng – Length in kilometers of this route

calcDelay(*self*)

calculate the Transportation delay given the speed and length.

transportStoD(*self*)

Get infectious individuals commuting from source node and inform them to destination

transportDtoS(*self*)

Get infectious individuals commuting from destination node and inform them to source

13.4 Class graph

Defines a graph with sites and edges

13.4.1 Methods

__init__(*self*, *graph_name*, *digraph*=`0`)

addSite(*self*, *sitio*)

Adds a site object to the graph. It takes a siteobj object as its only argument and returns None.

dijkstra(*self*, *G*, *start*, *end*=`None`)

Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
The input graph G is assumed to have the following representation: A vertex can be any object that can be used as an index into a dictionary. G is a dictionary, indexed by vertices. For any vertex v, G[v] is itself a dictionary, indexed by the neighbors of v. For any edge v->w, G[v][w] is the length of the edge. This is related to the representation in <http://www.python.org/doc/essays/graphs.html> where Guido van Rossum suggests representing graphs as dictionaries mapping vertices to lists of neighbors, however dictionaries of edges have many advantages over lists: they can store extra information (here, the lengths), they support fast existence tests, and they allow easy modification of the graph by edge insertion and removal. Such modifications are not needed here but are important in other graph algorithms. Since dictionaries obey iterator protocol, a graph represented as described here could be handed without modification to an algorithm using Guido's representation.
Of course, G and G[v] need not be Python dict objects; they can be any other object that obeys dict protocol, for instance a wrapper in which vertices are URLs and a call to G[v] loads the web page and finds its links.
The output is a pair (D,P) where D[v] is the distance from start to v and P[v] is the predecessor of v along the shortest path from s to v.
Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges seen before the end vertex is reached), but will correctly compute shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

getSite(*self*, *name*)

Retrieved a site from the graph.
Given a site's name the corresponding Siteobj instance will be returned.
If multiple sites exist with that name, a list of Siteobj instances is returned.
If only one site exists, the instance is returned. None is returned otherwise.

addEdge(*self*, *graph_edge*)

Adds an edge object to the graph.
It takes a edge object as its only argument and returns None.

getGraphdict(*self*)

Generates a dictionary of the graph for use in the shortest path function.

getEdge(*self*, *src*, *dst*)

Retrieved an edge from the graph.
Given an edge's source and destination the corresponding Edge instance will be returned.
If multiple edges exist with that source and destination, a list of Edge instances is returned.
If only one edge exists, the instance is returned. None is returned otherwise.

getSiteNames(*self*)

returns list of site names for a given graph.

getCycles(*self*)

The maximum number of independent cycles in a graph.
This number (u) is estimated by knowing the number of nodes (v), links (e) and of sub-graphs (p); u = e-v+p.
Trees and simple networks will have a value of 0 since they have no cycles.
The more complex a network is, the higher the value of u, so it can be used as an indicator of the level of development of a transport system.

shortestPath(*self*, *G*, *start*, *end*)

Find a single shortest path from the given start node to the given end node. The input has the same conventions as self.dijkstra(). 'G' is the graph's dictionary self.graphdict. 'start' and 'end' are site objects. The output is a list of the vertices in order along the shortest path.

clearVisual(*self*)

Clear the visual graph display

viewGraph(*self*, *mapa*=`'limites.txt'`)

Starts the Vpython display of the graph.

lightGRNode(*self*, *node*, *color*=`'r'`)

Paints red the sphere corresponding to the node in the visual display

drawGraph(*self*)

Draws the network using pylab

drawGraphR(*self*)

Draws the network using R

getAllPairs(*self*)

Returns a distance matrix for the graph nodes where the distance is the shortest path. Creates another distance matrix where the distances are the lengths of the paths.

getShortestPathLength(*self*, *origin*, *sp*)

Returns sp Length

getConnMatrix(*self*)

The most basic measure of accessibility involves network connectivity where a network is represented as a connectivity matrix (C1), which expresses the connectivity of each node with its adjacent nodes.
The number of columns and rows in this matrix is equal to the number of nodes in the network and a value of 1 is given for each cell where this is a connected pair and a value of 0 for each cell where there is an unconnected pair. The summation of this matrix provides a very basic measure of accessibility, also known as the degree of a node.

getWienerD(*self*)

Returns the Wiener distance for a graph.

getMeanD(*self*)

Returns the mean distance for a graph.

getDiameter(*self*)

Returns the diameter of the graph: longest shortest path.

getIotaindex(*self*)

Returns the Iota index of the graph
Measures the ratio between the network and its weighed vertices. It considers the structure, the length and the function of a graph and it is mainly used when data about traffic is not available.
It divides the length of a graph (L(G)) by its weight (W(G)). The lower its value, the more efficient the network is. This measure is based on the fact that an intersection (represented as a node) of a high order is able to handle large amounts of traffic.
The weight of all nodes in the graph (W(G)) is the summation of each node's order (o) multiplied by 2 for all orders above 1.

getWeight(*self*)

The weight of all nodes in the graph (W(G)) is the summation of each node's order (o) multiplied by 2 for all orders above 1.

getLength(*self*)

Sum of the length in kilometers of all edges in the graph.

getPiIndex(*self*)

Returns the Pi index of the graph.
The relationship between the total length of the graph L(G) and the distance along the diameter D(d). It is labeled as Pi because of its similarity with the real Pi (3.14), which is expressing the ratio between the circumference and the diameter of a circle.
A high index shows a developed network. It is a measure of distance per units of diameter and an indicator of the shape of a network.

getBetaIndex(*self*)

The Beta index measures the level of connectivity in a graph and is expressed by the relationship between the number of links (e) over the number of nodes (v).
Trees and simple networks have Beta value of less than one. A connected network with one cycle has a value of 1. More complex networks have a value greater than 1. In a network with a fixed number of nodes, the higher the number of links, the higher the number of paths possible in the network. Complex networks have a high value of Beta.

getAlphaIndex(*self*)

The Alpha index is a measure of connectivity which evaluates the number of cycles in a graph in comparison with the maximum number of cycles. The higher the alpha index, the more a network is connected. Trees and simple networks will have a value of 0. A value of 1 indicates a completely connected network.
Measures the level of connectivity independently of the number of nodes. It is very rare that a network will have an alpha value of 1, because this would imply very serious redundancies.

getGammaIndex(*self*)

The Gamma index is a A measure of connectivity that considers the relationship between the number of observed links and the number of possible links.
The value of gamma is between 0 and 1 where a value of 1 indicates a completely connected network and would be extremely unlikely in reality. Gamma is an efficient value to measure the progression of a network in time.

doStats(*self*)

Generate the descriptive stats about the graph.

plotDegreeDist(*self*, *cum*=`False`)

Plots the Degree distribution of the graph maybe cumulative or not.

getMedianSurvival(*self*)

Returns the time taken by the epidemic to reach 50% of the nodes.

getTotVaccinated(*self*)

Returns the total number of vaccinated.

getTotQuarantined(*self*)

Returns the total number of quarantined individuals.

getEpistats(*self*)

Returns a list of all epidemiologically related stats.

getInfectedCities(*self*)

Returns the number of infected cities.

getEpisize(*self*)

Returns the size of the epidemic

getEpispeed(*self*)

Returns the epidemic spreading speed.

getSpreadTime(*self*)

Returns the duration of the epidemic in units of time.

resetStats(*self*)

Resets all graph related stats

13.5 Class line

Basic line object containing attributes and methods common to all line objects.

13.5.1 Methods

__init__(*self*, *nodes*)

define a line based on sequence of nodes.

intersect(*self*)

finds out if this line intersects another

xarea(*self*)

finds out if this line extends to more than one area.

13.6 Class area

Basic area object containing attributes and methods common to all area objects.

13.6.1 Methods

__init__(*self*, *nodes*)

centroid(*self*)

calculates the centroid of the area.

area(*self*)

calculates the area

neighbors(*self*)

return neighbors

isIsland(*self*)

return true if the area is an island, i.e. has only one nejghboor and is not on a border.

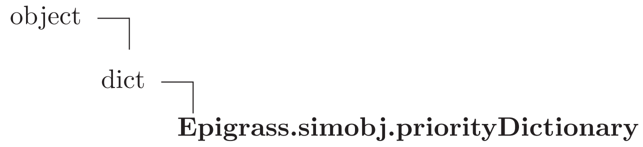
borderSizeWith(*self*, *area*)

returns the size of the border with a given area.

perimeter(*self*)

returns the perimeter of the area

13.7 Class priorityDictionary



13.7.1 Methods

__init__(*self*)

by David Eppstein. <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/117228> Initialize priorityDictionary by creating binary heap of pairs (value,key). Note that changing or removing a dict entry will not remove the old pair from the heap until it is found by smallest() or until the heap is rebuilt.

Return Value

```
new empty dictionary
```

Overrides: dict.__init__

smallest(*self*)

Find smallest item after removing deleted items from heap.

__iter__(*self*)

Create destructive sorted iterator of priorityDictionary.

Overrides: dict.__iter__

__setitem__(*self*, *key*, *val*)

Change value stored in dictionary and add corresponding pair to heap. Rebuilds the heap if the number of deleted items grows too large, to avoid memory leakage.

Overrides: dict.__setitem__

update(*self*, *other*)

Update D from E and F: for k in E: D[k] = E[k] (if E has keys else: for (k, v) in E: D[k] = v) then: for k in F: D[k] = F[k]

Return Value

```
None
```

Overrides: dict.update extit(inherited documentation)

setdefault(*self*, *key*, *val*)

Reimplement setdefault to call our customized __setitem__.

Return Value

```
D.get(k,d), also set D[k]=d if k not in D
```

Overrides: dict.setdefault

__cmp__(*x*, *y*)

cmp(x,y)

__contains__(*D*, *k*)

Return Value

```
True if D has a key k, else False
```

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__delitem__(*x*, *y*)

del x[y]

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getattribute__(...)

x.__getattribute__('name') <==> x.name

Overrides: object.__getattribute__

__getitem__(*x*, *y*)

x[y]

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

Overrides: object.__hash__

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__lt__(*x*, *y*)

x<y

__ne__(*x*, *y*)

x!=y

__new__(*T*, *S*, ...)

Return Value

```
    a new object with type S, a subtype of T
```

Overrides: object.__new__

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

clear(*D*)

Remove all items from D.

Return Value

```
None
```

copy(*D*)

Return Value

```
a shallow copy of D
```

fromkeys(*dict*, *S*, *v*=...)

v defaults to None.

Return Value

```
New dict with keys from S and values equal to v
```

get(*D*, *k*, *d*=...)

d defaults to None.

Return Value

```
D[k] if k in D, else d
```

has_key(*D*, *k*)

Return Value

```
True if D has a key k, else False
```

items(*D*)

Return Value

```
list of D's (key, value) pairs, as 2-tuples
```

iteritems(*D*)

Return Value

```
an iterator over the (key, value) items of D
```

iterkeys(*D*)

Return Value

```
an iterator over the keys of D
```

itervalues(*D*)

Return Value

```
an iterator over the values of D
```

keys(*D*)

Return Value

```
list of D's keys
```

pop(*D*, *k*, *d*=...)

If key is not found, d is returned if given, otherwise KeyError is raised

Return Value

```
v, remove specified key and return the corresponding value
```

popitem(*D*)

2-tuple; but raise KeyError if D is empty

Return Value

```
(k, v), remove and return some (key, value) pair as a
```

values(*D*)

Return Value

```
list of D's values
```

13.7.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

14 Module Epigrass.spread

14.1 Class Spread

14.1.1 Methods

__init__(*self*, *graphobj*, *outdir*=`'.'`, *encoding*=`'latin-1'`)

cleanTree(*self*)

Generates a unambiguous spread tree by selecting the most likely infector for each site

dotDraw(*self*, *tr*)

generate a jpeg image of the spread tree using pydot

display(*self*)

display the epidemic tree

writeGML(*self*, *tree*, *outdir*, *encoding*, *fname*=`"spreadtree.gml"`)

Save the tree in the GML format

writeENGML(*self*, *fobj*, *tree*)

Write the edges and Nodes section of a GML file

14.2 Class Consensus

14.2.1 Methods

__init__(*self*, *path*, *cutoff*=`0.0`)

readTress(*self*, *path*)

Read all files named epipath* from the current dir and return a collection of trees.

parseEpipath(*self*, *lines*)

Receives a list of strings and returns a list of tuples

consensus(*self*, *treelist*, *cutoff*)

Generate a consensus tree from the various trees generated by multiple runs. Saves the tree in gml format file

15 Module Epigrass.testinst

Unit testing script

15.1 Class testObjInstantiation

unittest.TestCase

Epigrass.testinst.testObjInstantiation

15.1.1 Methods

setUp(*self*)

Hook method for setting up the test fixture before exercising it.

Overrides: unittest.TestCase.setUp extit(inherited documentation)

testSites(*self*)

testEdges(*self*)

testGraph(*self*)

__call__(*self*, **args*, ***kwds*)

__init__(*self*, *methodName*=`'runTest'`)

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

__repr__(*self*)

__str__(*self*)

assertAlmostEqual(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

assertAlmostEquals(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

assertEqual(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are unequal as determined by the '==' operator.

assertEquals(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are unequal as determined by the '==' operator.

assertFalse(*self*, *expr*, *msg*=`None`)

Fail the test if the expression is true.

assertNotAlmostEqual(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

assertNotAlmostEquals(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

assertNotEqual(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are equal as determined by the '==' operator.

assertNotEquals(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are equal as determined by the '==' operator.

assertRaises(*self*, *excClass*, *callableObj*, **args*, ***kwargs*)

Fail unless an exception of class excClass is thrown by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

assertTrue(*self*, *expr*, *msg*=`None`)

Fail the test unless the expression is true.

assert_(*self*, *expr*, *msg*=`None`)

Fail the test unless the expression is true.

countTestCases(*self*)

debug(*self*)

Run the test without collecting errors in a TestResult

defaultTestResult(*self*)

fail(*self*, *msg*=`None`)

Fail immediately, with the given message.

failIf(*self*, *expr*, *msg*=`None`)

Fail the test if the expression is true.

failIfAlmostEqual(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

failIfEqual(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are equal as determined by the '==' operator.

failUnless(*self*, *expr*, *msg*=`None`)

Fail the test unless the expression is true.

failUnlessAlmostEqual(*self*, *first*, *second*, *places*=`7`, *msg*=`None`)

Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

failUnlessEqual(*self*, *first*, *second*, *msg*=`None`)

Fail if the two objects are unequal as determined by the '==' operator.

failUnlessRaises(*self*, *excClass*, *callableObj*, **args*, ***kwargs*)

Fail unless an exception of class excClass is thrown by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

id(*self*)

run(*self*, *result*=`None`)

shortDescription(*self*)

Returns a one-line description of the test, or None if no description has been provided.
The default implementation of this method returns the first line of the specified test method's docstring.

tearDown(*self*)

Hook method for deconstructing the test fixture after testing it.

16 Module epigrass

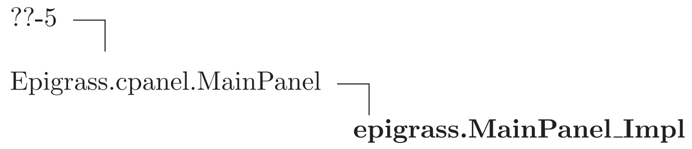
16.1 Functions

loadLang(*app*, *tr*, *lang*)

loads the language based on the user's choice.

main()

16.2 Class MainPanel_Impl



16.2.1 Methods

__init__(*self*, *parent*=`None`, *name*=`None`, *fl*=`0`)
Overrides: Epigrass.cpanel.MainPanel.__init__

initRc(*self*)

Initializes the epigrassrc file.

updateRc(*self*)

Updates the epigrassrc file with the contents of the configuration dictionary.

loadRcFile(*self*, *fname*, *config*={})

Loads epigrassrc. Returns a dictionary with keys of the form <section>.<option> and the corresponding values.

fillGui(*self*, *conf*)

Fill the gui fields from configuration dictionary conf.

checkConf(*self*)

Check if the configuration dictionary is complete

editScript(*self*)

Opens the model script with the user's preferred editor.

chooseScript(*self*)

starts the file selction dialog for the script.

readGui(*self*)

Gets the info from the GUI into the configuration dictionary.

onRunFuture(*self*)

Run the simulation on a separate thread

onRun(*self*)

starts the simulation when the run button is pressed

repRuns(*self*, *S*)

Do repeated runs

onDbBackup(*self*)

Dumps the epigrass databse to an sql file.

onDbInfo(*self*)

onHelp(*self*)

Opens the userguide

onRepOpen(*self*)

Opens the report PDF

onVisual(*self*)

Scan the epigrass database an shows available simulations.

getVariables(*self*)

Fill the variables list whenever a table is selected in the table list

onPlayButton(*self*)

Calls the epiplay module to replay the epidemic from a database table.

onConsensus(*self*)

Build the consensus tree

setBackend(self)

activate/deactivate db spec groupbox

onExit(self)

Close the gui

Main_layout_destroyed(self, a0)

languageChange(self)

16.3 Class Future

By David Perry - http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/84317 To run a function in a separate thread, simply put it in a Future:

```
>>> A=Future(longRunningFunction, arg1, arg2 ...)
```

It will continue on its merry way until you need the result of your function. You can read the result by calling the Future like a function, for example:

```
>>> print A()
```

If the Future has completed executing, the call returns immediately. If it is still running, then the call blocks until the function completes. The result of the function is stored in the Future, so subsequent calls to it return immediately.

A few caveats: Since one wouldn't expect to be able to change the result of a function, Futures are not meant to be mutable. This is enforced by requiring the Future to be "called", rather than directly reading __result. If desired, stronger enforcement of this rule can be achieved by playing with __getattr__ and __setattr__.

The Future only runs the function once, no matter how many times you read it. You will have to re-create the Future if you want to re-run your function; for example, if the function is sensitive to the time of day.

For more information on Futures, and other useful parallel programming constructs, read Gregory V. Wilson's _Practical Parallel Programming_.

16.3.1 Methods

__init__(self, func, *param)

__repr__(self)

__call__(self)

Wrapper(self, func, param)

Index