libdw API

Submodules

libdw.basicGridMap module

Simple grid map with values equal to `True` and `False`. Initialized by reading in a soar world file.

class `libdw.basicGridMap.`**`BasicGridMap`**(*worldPath*, *gridSquareSize*, *windowWidth*=400)

Bases: **`libdw.gridMap.GridMap`**

Implements the `GridMap` interface.

`indicesToBoxSegs`(*indices*)

Parameters: **indices** – pair of `(ix, iy)` indices of a grid cell

Returns: list of four line segments that constitute the boundary of the cell, grown by the radius of the robot, which is found in `gridMap.robotRadius`.

`makeStartingGrid`()

Called by `gridMap.GridMap.__init__`. Returns the initial value for the grid, which will be stored in `self.Grid`.

`robotCanOccupy`((*xIndex*, *yIndex*))

Returns `True` if the robot's center can be at any location within this cell and not cause a collision.

libdw.bayesMap module

class `libdw.bayesMap.`**`BayesGridMap`**(*xMin*, *xMax*, *yMin*, *yMax*, *gridSquareSize*)

Bases: **`libdw.dynamicGridMap.DynamicGridMap`**

`clearCell`((*xIndex*, *yIndex*))

`cost`((*xIndex*, *yIndex*))

`cost1`((*xIndex*, *yIndex*))

`explored`((*xIndex*, *yIndex*))

`makeStartingGrid`()

`occProb`((*xIndex*, *yIndex*))

`occupied`((*xIndex*, *yIndex*))

`setCell`((*xIndex*, *yIndex*))

`squareColor`((*xIndex*, *yIndex*))

`libdw.bayesMap.`**`oGivenS`**(*s*)

`libdw.bayesMap.`**`testCellDynamics`**(*cellSSM*, *input*)

`libdw.bayesMap.`**`uGivenAS`**(*a*)

libdw.boundarySM module

libdw.cc module

class `libdw.cc.`**`Circuit`**(*components*)

`addComponentEquations`(*equationSet*)

`addKCLEquations`(*equationSet*, *groundVoltage*)

`displaySolution`(*groundNode*)

`makeEquationSet`(*groundVoltage*)

class `libdw.cc.`**`CircuitNode`**

`addConnection`(*sign*, *currentName*)

`kclEquation`()

class `libdw.cc.`**`Component2Leads`**(*value*, *v1*, *v2*)

`addKCLToNodes`(*nodeDict*)

class `libdw.cc.`**`ISrc`**(*value*, *v1*, *v2*)

Bases: **`libdw.cc.Component2Leads`**

`componentEquation`()

class `libdw.cc.`**`OpAmp`**(*v1*, *v2*, *v3*, *K=10000*)

`addKCLToNodes`(*nodeDict*)

`componentEquation`()

class `libdw.cc.`**`Resistor`**(*value*, *v1*, *v2*)

Bases: **`libdw.cc.Component2Leads`**

`componentEquation`()

class `libdw.cc.`**`VSrc`**(*value*, *v1*, *v2*)

Bases: **`libdw.cc.Component2Leads`**

`componentEquation`()

class `libdw.cc.`**`Wire`**(*v1*, *v2*)

Bases: **`libdw.cc.Component2Leads`**

`componentEquation`()

`libdw.cc.`**`isrc`**(*Is*, *i*)

`libdw.cc.`**`kcl`**(*pos*, *neg*)

`libdw.cc.`**`opamp`**(*K*, *vPlus*, *vMinus*, *voutPlus*)

`libdw.cc.`**`resistor`**(*R*, *vn1*, *vn2*, *i12*)

`libdw.cc.`**`setGround`**(*vn1*)

`libdw.cc.`**`thevenin`**(*VR*, *vn1*, *vn2*, *i12*)

`libdw.cc.`**`vsrc`**(*Vs*, *vn1*, *vn2*)

`libdw.cc.`**`wire`**(*vn1*, *vn2*)

libdw.circ module

Describe a circuit in terms of its components; generates equations and solves them.

class `libdw.circ.`**`Circuit`**(*components*)

`addComponent`(*component*)

`isc`(*nPlus*, *nMinus*)

Find the short-circuit current: Add a wire across the positive and negative terminals and measure the current there

`solve`(*gnd*)

Parameters: **gnd** – Name of the node to set to ground (string)
Returns: instance of `le.Solution`, mapping node names to values

`theveninEquivalent`(*nPlus*, *nMinus*)

`voc`(*nPlus*, *nMinus*)

Find the open-circuit voltage by setting nMinus to ground and finding the voltage at nPlus

class `libdw.circ.`**`Component`**

Generic superclass. Every component type has to provide

- `getCurrents(self)`: Returns a list of tuples `(i, node, sign)`, where `i` is the name of a current variable, `node` is the name of a node, and `sign` is the sign of that current at that node.
- `getEquation(self)`: Returns an instance of `le.Equation`, representing the constituent equation for this component.

`getCurrents`()

Default method that works for components with two leads, assuming they define attributes `current`, `n1`, and `n2`.

class `libdw.circ.`**`ISrc`**(*i*, *n1*, *n2*)

Bases: **`libdw.circ.Component`**

`current` = *None*

Name of the current variable for this component

`getEquation`()

class `libdw.circ.`**`NodeToCurrents`**

Keep track of which currents are flowing in and out of which nodes in a circuit.

`addCurrent`(*current*, *node*, *sign*)

Parameters:
- **current** – name of a current variable (string)
- **node** – name of a node (string)
- **sign** – +1 or -1, indicating whether the current is flowing into or out of the node

Adds the new current, with approrpiate sign to `node`. Adds an entry for `node`, if doesn't already exist in the dictionary.

`addCurrents`(*currents*)

Parameters: **currents** – list of tuples `(currentName, nodeName, sign)`, with the same meaning as for `addCurrent`. Add several currents at once.

`d` = *None*

Dictionary, mapping a node name to a list of current descriptions. Each current description is a list of a current name and a sign (+1 or -1), indicating whether the current is flowing into or out of that node.

`getKCLEquations`(*gnd*)

Parameters: **gnd** – name of a node that will have its voltage assigned to 0 (string)
Returns: a list of equations, one for each node. For the ground node, it just asserts that its voltage is 0. For the other nodes, the equation asserts that the sum of the currents going into the node minus the sum of currents going out of the node is equal to zero.

class `libdw.circ.`**`OpAmp`**(*nPlus*, *nMinus*, *nOut*, *K=10000*)

Bases: **libdw.circ.Component**

Asserts that nOut = K(nPlus - nMinus).

current = *None*

Name of the current variable for this component

getCurrents()

getEquation()

class `libdw.circ.`**Resistor**(*r*, *n1*, *n2*)

Bases: **libdw.circ.Component**

current = *None*

Name of the current variable for this component

getEquation()

class `libdw.circ.`**Thevenin**(*v*, *r*, *n1*, *n2*)

Bases: **libdw.circ.Component**

An abstract component consisting of a resistor and a voltage source in series.

current = *None*

Name of the current variable for this component

getEquation()

class `libdw.circ.`**VSrc**(*v*, *n1*, *n2*)

Bases: **libdw.circ.Component**

current = *None*

Name of the current variable for this component

getEquation()

class `libdw.circ.`**Wire**(*n1*, *n2*)

Bases: **libdw.circ.Component**

Just describes a wire between nodes `n1` and `n2`; nodes are specified by their names (strings)

current = *None*

Name of the current variable for this component

getEquation()

libdw.circuitConnect module

libdw.coloredHall module

State estimation example: localization in a colored hallway

class `libdw.coloredHall.`**TextInputSM**(*legalInputs*)

Bases: **libdw.sm.SM**

Machine that prompts a user for an input on each step. That input is the output of this machine. If the user types 'quit', then the machine terminates.

done(*state*)

getNextValues(*state*, *inp*)

startState = *False*

libdw.coloredHall.**drawBelief**(*belief*, *window*, *numStates*, *drawNums=True*)

libdw.coloredHall.**hallSE**(*hallwayColors*, *legalInputs*, *obsNoise*, *dynamics*, *transNoise*, *initialDist=None*, *verbose=True*)

libdw.coloredHall.**leftSlipTransNoiseModel**(*nominalLoc*, *hallwayLength*)

Parameters:
- **nominalLoc** – location that the robot would have ended up given perfect dynamics
- **hallwayLength** – length of the hallway

Returns: distribution over resulting locations, modeling noisy execution of commands; in this case, the robot goes to the nominal location with probability 0.9, and goes one step too far left with probability 0.1. If any of these locations are out of bounds, then the associated probability mass stays at `nominalLoc`.

libdw.coloredHall.**makeObservationModel**(*hallwayColors*, *obsNoise*)

Parameters:
- **hallwayColors** – list of colors, one for each room in the hallway, from left to right
- **obsNoise** – conditional distribution specifying the probability of observing a color given the actual color of the room

Returns: conditional distribution specifying probability of observing a color given the robot's location

Remember that a conditional distribution P(A | B) is represented as a function from values of b to distributions over A.

libdw.coloredHall.**makeSESwithGUI**(*worldSM*, *realColors*, *legalInputs*, *initBelief=None*, *verbose=False*, *title='hallway'*)

Makes a colored hallway simulator and state estimator. Text input for actions and graphical display of world and belief state.

Parameters:
- **worldSM** – instance of `ssm.StochasticSM` representing the world
- **realColors** – A list of the colors of the rooms in the hallway, from left to right.
- **legalInputs** – A list of the possible action commands
- **verbose** – if `True` then print out belief state after each update
- **title** – title of window being created

libdw.coloredHall.**makeSim**(*hallwayColors*, *legalInputs*, *obsNoise*, *dynamics*, *transNoise*, *title='sim'*, *initialDist=None*)

Make an instance of the simulator with noisy motion and sensing models.

Parameters:
- **hallwayColors** – A list of the colors of the rooms in the hallway, from left to right.
- **legalInputs** – A list of the possible action commands
- **obsNoise** – conditional distribution specifying the probability of observing a color given the actual color of the room
- **dynamics** – function that takes the robot's current location, action, and hallwaylength, and returns its nominal new location
- **transNoise** – P(actualResultingLocation | nominalResultingLoc) represented as a function from ideal location to the actual location the robot will end up in
- **title** – String specifying title for simulator window

libdw.coloredHall.**makeTransitionModel**(*dynamics*, *noiseDist*, *hallwayLength*)

Parameters:
- **dynamics** – function that takes the robot's current location, action, and hallwaylength, and returns its nominal new location
- **noiseDist** – P(actualResultingLocation | nominalResultingLoc) represented as a function from ideal location to the actual location the robot will end up in
- **hallwayLength** – number of rooms in the hallway

Returns: P(actualResultingLoc | previousLoc, action) represented as a function that takes an action and returns a function that takes a previous location and returns a distribution over actual resulting locations.

`libdw.coloredHall.`**`noisyObsNoiseModel`**(*actualColor*)

Parameters: **actualColor** – actual color in a location

Returns: `DDist` over observed colors when in a room that has `actualColor`. In this case, we observe the actual color with probability 0.8, and the remaining 0.2 probability is divided uniformly over the other possible colors in this world.

`libdw.coloredHall.`**`noisyTransNoiseModel`**(*nominalLoc*, *hallwayLength*)

Parameters:
- **nominalLoc** – location that the robot would have ended up given perfect dynamics
- **hallwayLength** – length of the hallway

Returns: distribution over resulting locations, modeling noisy execution of commands; in this case, the robot goes to the nominal location with probability 0.8, goes one step too far left with probability 0.1, and goes one step too far right with probability 0.1. If any of these locations are out of bounds, then the associated probability mass goes is assigned to the boundary location (either 0 or `hallwayLength-1`).

`libdw.coloredHall.`**`perfectObsNoiseModel`**(*actualColor*)

Parameters: **actualColor** – actual color in a location

Returns: `DDist` over observed colors when in a room that has `actualColor`. In this case, we observe the actual color with probability 1.

`libdw.coloredHall.`**`perfectTransNoiseModel`**(*nominalLoc*, *hallwayLength*)

Parameters:
- **nominalLoc** – location that the robot would have ended up given perfect dynamics
- **hallwayLength** – length of the hallway

Returns: distribution over resulting locations, modeling noisy execution of commands; in this case, the robot goes to the nominal location with probability 1.0

`libdw.coloredHall.`**`possibleColors`** *= ('black', 'white', 'red', 'green', 'blue', 'purple', 'orange', 'darkGreen', 'gold', 'chocolate', 'PapayaWhip', 'MidnightBlue', 'HotPink', 'chartreuse')*

Possible colors for rooms in our hallway

`libdw.coloredHall.`**`ringDynamics`**(*loc*, *act*, *hallwayLength*)

Parameters:
- **loc** – current loc (integer index) of the robot
- **act** – positive or negative integer offset
- **hallwayLength** – number of cells in the hallway

Returns: new loc of the robot, assuming perfect execution where the hallway is actually a ring (so that location 0 is next to location `hallwayLength -1`).

`libdw.coloredHall.`**`standardDynamics`**(*loc*, *act*, *hallwayLength*)

Parameters:
- **loc** – current loc (integer index) of the robot
- **act** – a positive or negative integer (or 0) indicating the nominal number of squares moved
- **hallwayLength** – number of cells in the hallway

Returns: new loc of the robot assuming perfect execution. If the action would take it out of bounds, the robot stays where it is.

`libdw.coloredHall.`**`standardHallway`** *= ['white', 'white', 'green', 'white', 'white']*

Our favorite configuration of hallway colors

`libdw.coloredHall.`**`textOutput`**(*result*)

`libdw.coloredHall.`**`wrapTextUI`**(*m*)

Parameters: **m** – An instance of `sm.SM`

Returns: A composite machine that prompts the user for input to, and prints the output of `m` on each step.

`libdw.coloredHall.`**`wrapWindowUI`**(*m*, *worldColors*, *legalInputs*, *windowName='Belief'*, *initBelief=None*)

Parameters:
- **m** – A machine created by applying `se.makeStateEstimationSimulation` to a hallway world, which take movement commands as input and generates as output structures of the form `(b, (o, a))`, where `b` is a belief state, `a` is the action command, and `o` is the observable output generated by the world.
- **worldColors** – A list of the colors of the rooms in the hallway, from left to right.

Returns: A composite machine that prompts the user for input to, and graphically displays the output of `m` on each step.

libdw.colors module

Utility procedures for manipulating colors

`libdw.colors.`**`HSVtoRGB`**(*h*, *s*, *v*)

Convert a color represented in hue, saturation, value space into RGB space. :param h: hue, in range (0, 360) :param s: saturation, in range (0, 1) :param v: value, in range (0, 1) :returns: (r, g, b) with each value in the range (0, 1)

`libdw.colors.`**`RGBToPyColor`**(*colorVals*)

Parameters: **colorVals** – tuple (r, g, b) of values in (0, 1) representing a color in rgb space

Returns: a python color string

`libdw.colors.`**`probToMapColor`**(*p*, *hue=60.0*)

Parameters: **p** – probability value

Returns: a Python color that's good for mapmaking. It's yellow when p = 0.5, black when p = 1, white when p = 1.

`libdw.colors.`**`probToPyColor`**(*p*, *uniformP=0.5*, *upperVal=None*)

Converts a probability to a Python color. Probability equal to uniform converts to black. Closer to 1 is brighter blue; closer to 0 is brighter red. :param p: probability value in range (0, 1) :param uniformP: probability value that will be colored black :param upperVal: in situations when there are lots of choices and so the highest reasonable value to occur is nowhere near 1, it can be useful to set this to the highest probability value you expect, in order to get some useful visual dynamic range. :returns: A Python color

`libdw.colors.`**`rootToPyColor`**(*p*, *minV*, *maxV*)

Color map for making root-locus plots

`libdw.colors.`**`safeLog`**(*v*)

Log, but it returns -1000 for arguments less than or equal to 0.

libdw.corruptInput module

State machine to add random noise to sonar and odometry

class `libdw.corruptInput.`**`CorruptedSensorInput`**(*sonars*, *odometry*)

This class has the same interface as `io.SensorInput`, so instances can be used anywhere we use instances of `io.SensorInput`

`analogInputs` = *None*

Analog inputs are 0

`odometry` = *None*

Instance of `util.Pose`

`sonars` = *None*

List of 6 sonar readings

class `libdw.corruptInput.`**`SensorCorrupter`**(*sonarStDev*, *odoStDev*)

Bases: **`libdw.sm.SM`**

State machine that takes instances of `io.SensorInput` and adds noise to them. Sonars have additive noise, drawn from a Gaussian with 0 mean and `sonarStDev` standard deviation. Odometry is changed only in the x dimension, with additive noise with 0 mean and `odoStDev` standard deviation. Output of the state machine are instances of `CorruptedSensorInput`.

`getNextValues`(*state*, *inp*)

libdw.dist module

Discrete probability distributions

class `libdw.dist.`**`DDist`**(*dictionary*)

Discrete distribution represented as a dictionary. Can be sparse, in the sense that elements that are not explicitly contained in the dictionary are assumed to have zero probability.

`conditionOnVar`(*index*, *value*)

Parameters:
- **index** – index of a variable in the joint distribution
- **value** – value of that variable

Returns: new distribution, conditioned on variable `i` having value `value`, and with variable `i` removed from all of the elements (it's redundant at this point).

`d` = *None*

Dictionary whose keys are elements of the domain and values are their probabilities.

`dictCopy`()

Returns: A copy of the dictionary for this distribution.

`draw`()

Returns: a randomly drawn element from the distribution

`marginalizeOut`(*index*)

Parameters: **index** – index of a random variable to sum out of the distribution

Returns: DDist on all the rest of the variables

`maxProbElt`()

Returns: The element in this domain with maximum probability

`prob`(*elt*)

Parameters: **elt** – an element of the domain of this distribution (does not need to be explicitly represented in the dictionary; in fact, for any element not in the dictionary, we return probability 0 without error.)

Returns: the probability associated with `elt`

`support`()

Returns: A list (in arbitrary order) of the elements of this distribution with non-zero probabability.

`libdw.dist.`**`DeltaDist`**(*v*)

Distribution with all of its probability mass on value `v`

`libdw.dist.`**`JDist`**(*PA*, *PBgA*)

Create a joint distribution on P(A, B) (in that order), represented as a `DDist`

Parameters:
- **PA** – a `DDist` on some random var A
- **PBgA** – a conditional probability distribution specifying P(B | A) (that is, a function from elements of A to `DDist` on B)

class `libdw.dist.`**`MixtureDist`**(*d1*, *d2*, *p*)

A mixture of two probabability distributions, d1 and d2, with mixture parameter p. Probability of an element x under this distribution is p * d1(x) + (1 - p) * d2(x). It is as if we first flip a probability-p coin to decide which distribution to draw from, and then choose from the approriate distribution.

This implementation is lazy; it stores the component distributions. Alternatively, we could assume that d1 and d2 are DDists and compute a new DDist.

`draw`()

`prob`(*elt*)

`support`()

`libdw.dist.`**`UniformDist`**(*elts*)

Uniform distribution over a given finite set of `elts` :param elts: list of any kind of item

`libdw.dist.`**`bayesEvidence`**(*PA*, *PBgA*, *b*)

Parameters:
- **PBgA** – conditional distribution over B given A (function from values of a to `DDist` over B)
- **PA** – prior on A
- **b** – evidence value for B = b

Returns: P(A | b)

`libdw.dist.`**`incrDictEntry`**(*d*, *k*, *v*)

If dictionary `d` has key `k`, then increment `d[k]` by `v`. Else set `d[k] = v`.

Parameters:
- **d** – dictionary
- **k** – legal dictionary key (doesn’t have to be in `d`)
- **v** – numeric value

`libdw.dist.`**`removeElt`**(*items*, *i*)

non-destructively remove the element at index i from a list; returns a copy; if the result is a list of length 1, just return the element

`libdw.dist.`**`squareDist`**(*lo*, *hi*, *loLimit=None*, *hiLimit=None*)

Construct and return a DDist over integers. The distribution will have a uniform distribution on integers from lo to hi-1 (inclusive). Any probability mass that would be below `lo` or above `hi` is assigned to `lo` or `hi`.

`libdw.dist.`**`totalProbability`**(*PA*, *PBgA*)

Parameters:
- **PBgA** – conditional distribution over B given A (function from values of a to `DDist` over B)
- **PA** – distribution over A (object of type `DiscreteDist`)

Returns: P(B) using the law of total probability. `self` represents P(B | A); P(A) is the argument to the method; we compute and return P(B) as sum_a P(B | a) P(a)

`libdw.dist.`**`triangleDist`**(*peak*, *halfWidth*, *lo=None*, *hi=None*)

Construct and return a DDist over integers. The distribution will have its peak at index `peak` and fall off linearly from there, reaching 0 at an index `halfWidth` on either side of `peak`. Any probability mass that would be below `lo` or above `hi` is assigned to `lo` or `hi`

libdw.distPlot module

class `libdw.distPlot.`**`IntDistSignal`**(*d*)

Bases: **`libdw.sig.Signal`**

plotDist()

sample(*n*)

libdw.distPlot.**plot**(*d*)

libdw.dw module

class libdw.dw.**DrawingWindow**(*windowWidth*, *windowHeight*, *xMin*, *xMax*, *yMin*, *yMax*, *title*)

clear()

delete(*thing*)

destroy()

drawLine(*(a, b, c)*, *color='black'*)

drawLineSeg(*x1*, *y1*, *x2*, *y2*, *color='black'*, *width=2*)

drawPoint(*x*, *y*, *color='blue'*)

drawRect(*(x1, y1)*, *(x2, y2)*, *color='black'*)

drawRobot(*x*, *y*, *noseX*, *noseY*, *color='blue'*, *size=8*)

drawRobotWithNose(*x*, *y*, *theta*, *color='blue'*, *size=6*)

drawSquare(*x*, *y*, *size*, *color='blue'*)

drawText(*x*, *y*, *label*, *color='blue'*)

drawUnscaledLineSeg(*x1*, *y1*, *xproj*, *yproj*, *color='black'*, *width=1*)

drawUnscaledRect(*x1*, *y1*, *xproj*, *yproj*, *color='black'*)

save()

scaleX(*x*)

scaleY(*y*)

scaleYMag(*y*)

libdw.dynamicCountingGridMap module

class libdw.dynamicCountingGridMap.**DynamicCountingGridMap**(*xMin*, *xMax*, *yMin*, *yMax*, *gridSquareSize*)

Bases: **libdw.gridMap.GridMap**

Implements the `GridMap` interface.

clearCell(*(xIndex, yIndex)*)

grid = *None*

values stored in the grid cells

occupied(*(xIndex, yIndex)*)

robotCanOccupy(*(xIndex, yIndex)*)

setCell(*(xIndex, yIndex)*)

squareColor(*indices*)

:param documentme

xMax = *None*
X coordinate of right edge

xMin = *None*
X coordinate of left edge

xN = *None*
number of cells in x dimension

xStep = *None*
size of a side of a cell in the x dimension

yMax = *None*
Y coordinate of top edge

yMin = *None*
Y coordinate of bottom edge

yN = *None*
number of cells in y dimension

yStep = *None*
size of a side of a cell in the y dimension

libdw.dynamicGridMap module

Grid map class that allows values to be set and cleared dynamically.

class `libdw.dynamicGridMap.`**`DynamicGridMap`**(*xMin*, *xMax*, *yMin*, *yMax*, *gridSquareSize*)

Bases: **`libdw.gridMap.GridMap`**

Implements the `GridMap` interface.

`clearCell`(*(xIndex*, *yIndex)*)
Takes indices for a grid cell, and updates it, given information that it does not contain an obstacle. In this case, it sets the cell to `True`, and redraws it if its color has changed.

`makeStartingGrid`()
Returns the initial value for `self.grid`. Can depend on `self.xN` and `self.yN` being set.

In this case, the grid is an array filled with the value `False`, meaning that the cells are not occupied.

`occupied`(*(xIndex*, *yIndex)*)
Returns `True` if there is an obstacle in any part of this cell. Note that it can be the case that a cell is not occupied, but the robot cannot occupy it (because if the robot's center were in that cell, some part of the robot would be in collision.

`robotCanOccupy`(*(xIndex*, *yIndex)*)
Returns `True` if the robot's center can be at any location within the cell specified by `(xIndex, yIndex)` and not cause a collision. This implementation is very slow: it considers a range of boxes around the spcified box, and ensures that none of them is `self.occupied`.

`setCell`(*(xIndex*, *yIndex)*)
Takes indices for a grid cell, and updates it, given information that it contains an obstacle. In this case, it sets the cell to `True`, and redraws it if its color has changed.

`squareColor`(*indices*)

Parameters: **indices** – `(ix, iy)` indices of a grid cell
Returns: a color string indicating what color that cell should be drawn in.

libdw.dynamicMoveToPoint module

class `libdw.dynamicMoveToPoint.`**`DynamicMoveToPoint`**(*maxRVel=0.5*, *maxFVel=0.5*)

Bases: **`libdw.sm.SM`**

Drive to a goal point in the frame defined by the odometry. Goal points are part of the input, in contrast to `moveToPoint.MoveToPoint`, which takes a single goal pose at initialization time.

Assume inputs are `(util.Point, io.SensorInput)` pairs

`angleEps` = *0.05*

`distEps` = *0.05*

`done`(*state*)

`forwardGain` = *2.0*

`getNextValues`(*state*, *inp*)

`rotationGain` = *1.5*

`startState` = *False*

State is `True` if we have reached the goal and `False` otherwise

libdw.eBotsonarDist module

Useful constants and utilities for dealing with sonar readings in soar.

`libdw.eBotsonarDist.`**`distAndAngle`**(*h0*, *h1*)

`libdw.eBotsonarDist.`**`getDistanceRight`**(*sonarValues*)

Parameters: **sonarValues** – list of 6 sonar readings
Returns: the perpendicular distance to a surface on the right of the robot, assuming there is a linear surface.

`libdw.eBotsonarDist.`**`getDistanceRightAndAngle`**(*sonarValues*)

Parameters: **sonarValues** – list of 6 sonar readings
Returns: (d, a) where, d is the perpendicular distance to a surface on the right of the robot, assuming there is a linear surface; and a is the angle to that surface.

Change to use `sonarHit`, or at least point and pose transforms.

`libdw.eBotsonarDist.`**`line`**(*h0*, *h1*)

`libdw.eBotsonarDist.`**`sonarHit`**(*distance*, *sonarPose*, *robotPose*)

Parameters:
- **distance** – distance along ray that the sonar hit something
- **sonarPose** – `util.Pose` of the sonar on the robot
- **robotPose** – `util.Pose` of the robot in the global frame

Returns: `util.Point` representing position of the sonar hit in the global frame.

`libdw.eBotsonarDist.`**`sonarMax`** = *1.5*

Maximum good sonar reading.

`libdw.eBotsonarDist.sonarPoses` = *[pose:(-0.012000, -0.040000, 1.570796), pose:(0.012000, -0.028000, 0.785398), pose:(0.029000, 0, 0.000000), pose:(0.012000, 0.028000, -0.785398), pose:(-0.012000, 0.040000, -1.570796), pose:(-0.029000, 0, 3.141593)]*

Positions and orientations of sonar sensors with respect to the center of the robot.

libdw.eyeServo module

`libdw.eyeServo.runTest`(*lines*, *parent=None*, *nsteps=150*)

`libdw.eyeServo.simpleSignal`(*dist=1.0*, *simTime=3.0*)

`libdw.eyeServo.simpleSignal2`(*dist=1.0*, *simTime=3.0*)

`libdw.eyeServo.testSignal`(*dist=1.0*, *simTime=3.0*)

libdw.fr module

class `libdw.fr.ForwardTSM`(*delta*, *maxVel=0.5*)

Bases: **`libdw.sm.SM`**

State machine that will cause the robot to drive forward a distance d from its pose at the time it takes its first step.

Uses a proportional controller, but may clip velocities.

`distTargetEpsilon` = *0.01*

`done`(*state*)

`forwardGain` = *1.0*

`getNextValues`(*state*, *inp*)

`maxVel` = *0.5*

`startState` = *'start'*

class `libdw.fr.RotateTSM`(*headingDelta*, *maxVel=0.5*)

Bases: **`libdw.sm.SM`**

State machine that will cause the robot to rotate to an angle specified as an offset from its angle at the time the machine takes its first step.

If you command a rotation of 2*Pi, it will stay still. If you want it to go all the way around, you have to give it several subgoals. Asking for math.pi/2 four times would work fine.

Uses a proportional controller.

`angleEpsilon` = *0.01*

`done`(*state*)

`getNextValues`(*state*, *inp*)

`rotationalGain` = *3.0*

`startState` = *'start'*

class `libdw.fr.StopSM`

Bases: **`libdw.sm.SM`**

Robot controller that always generates the stop action

getNextValues(*state*, *inp*)

libdw.gauss module

`libdw.gauss.`**gaussSolve**(*Ain*, *bin*)

`libdw.gauss.`**swap1**(*b*, *i1*, *i2*)

`libdw.gauss.`**swap2**(*a*, *i1*, *j1*, *i2*, *j2*)

libdw.gfx module

class `libdw.gfx.`**PlotJob**(*xname*, *yname*, *connectPoints*, *xfunc=None*, *yfunc=None*, *xbounds='auto'*, *ybounds='auto'*)

callFunc(*func*, *inp*)

class `libdw.gfx.`**RobotGraphics**(*drawSlimeTrail=False*, *sonarMonitor=False*)

addDynamicPlotFunction(*y=('step', None)*)

Parameters: **y** – function to call for y-axis of dynamic plot

addDynamicPlotSMProbe(*y=('step', None, None, None)*, *connectPoints=False*)

Parameters: **y** – probe for y-axis of dynamic plot

addProbe(*probe*)

addStaticPlotFunction(*x=('step', None)*, *y=('step', None)*, *connectPoints=False*)

Parameters:
- **x** – function to call for x-axis of static plot
- **y** – function to call for y-axis of static plot
- **connectPoints** – Boolean, whether or not to draw lines between the points. Default is False.

addStaticPlotSMProbe(*x=('step', None, None, None)*, *y=('step', None, None, None)*, *connectPoints=False*)

Parameters:
- **x** – probe for x-axis of static plot
- **y** – probe for y-axis of static plot
- **connectPoints** – Boolean; whether or not to draw lines between the points. Default is False.

clearPlotData()

closePlotWindows()

connectPointsDef = *False*

doDataPlotJobs()

enableSonarMonitor()

getBounds(*data*, *bounds*)

getEquallyScaledBounds(*xData*, *yData*)

makeTraceFun(*machineName*, *mode*, *valueFun*, *stream*)

plot()

plotDataVersusData(*xData*, *xBounds*, *yData*, *yBounds*, *name*, *connectPoints*, *windowSize=None*)

plotSlime()

recentPt(*name*)

reset()

setUpPlotting(*dataProbes*, *plotTasks*)

stepPlotting()

tasks()

libdw.gridDynamics module

class `libdw.gridDynamics.`**GridCostDynamicsSM**(*theMap*)

Bases: **`libdw.sm.SM`**

Fix me

getNextValues(*state*, *inp*)

Parameters:
- **state** – tuple of indices `(ix, iy)` representing robot's location in grid map
- **inp** – an action, which is one of the legal inputs

Returns: `(nextState, cost)`

legal(*ix*, *iy*)

legalInputs = *None*

In any state, you can move to any of the eight neighboring squares or stay in place. Actions are (dx, dy), where dx and dy changes in x and y indices, in the set (-1, 1, 0).

probCost(*(ix, iy)*, *(newX, newY)*)

theMap = *None*

instance of `gridMap.GridMap` representing locations of obstacles, with discretized poses

class `libdw.gridDynamics.`**GridDynamics**(*theMap*)

Bases: **`libdw.sm.SM`**

An SM representing an abstract grid-based view of a world. Use the XY resolution of the underlying grid map. Action space is to move to a neighboring square States are grid coordinates Output is just the state

To use this for planning, we need to supply both start and goal.

getNextValues(*state*, *inp*)

Parameters:
- **state** – tuple of indices `(ix, iy)` representing robot's location in grid map
- **inp** – an action, which is one of the legal inputs

Returns: `(nextState, cost)`

legal(*ix*, *iy*, *newX*, *newY*)

legalInputs = *None*

In any state, you can move to any of the eight neighboring squares or stay in place. Actions are (dx, dy), where dx and dy changes in x and y indices, in the set (-1, 1, 0).

theMap = *None*

instance of `gridMap.GridMap` representing locations of obstacles, with discretized poses

libdw.gridDynamicsWithAngle module

class `libdw.gridDynamicsWithAngle.`**GridDynamics**(*theMap*, *rotationCost=None*)

Bases: **`libdw.sm.SM`**

An SM representing an abstract grid-based view of a world. Use the XY resolution of the underlying grid map. Action space is to move to a neighboring square States are grid coordinates Output is just the state

To use this for planning, we need to supply both start and goal.

`getNextValues`(*state*, *inp*)

Parameters:
- **state** – tuple of indices `(ix, iy)` representing robot's location in grid map
- **inp** – an action, which is one of the legal inputs

Returns: `(nextState, cost)`

`legal`(*ix*, *iy*, *newX*, *newY*)

`legalInputs` = *None*

In any state, you can move to any of the eight neighboring squares or stay in place. Actions are (dx, dy), where dx and dy changes in x and y indices, in the set (-1, 1, 0).

`theMap` = *None*

instance of `gridMap.GridMap` representing locations of obstacles, with discretized poses

libdw.gridMap module

Abstract superclass for various grid maps.

class **`libdw.gridMap.GridMap`**(*xMin*, *xMax*, *yMin*, *yMax*, *gridSquareSize*, *windowWidth=400*)

`boxDim`()

Returns: size of a grid cell in the drawing window in pixels

`drawNewSquare`(*indices*, *color=None*)

Parameters:
- **indices** – `(ix, iy)` indices of grid cell
- **color** – Python color to draw the square; if `None` uses the `self.squareColor` method to determine a color.

Draws a box at the specified point, on top of whatever is there

`drawPath`(*path*)

Draws list of cells; first one is purple, last is yellow, rest are blue :param path: list of pairs of `(ix, iy)` grid indices

`drawSquare`(*indices*, *color=None*)

Recolors the existing square :param indices: `(ix, iy)` indices of grid cell :param color: Python color to draw the square; if `None` uses the `self.squareColor` method to determine a color.

`drawWorld`()

Clears the whole window and redraws the grid

`graphicsGrid` = *None*

graphics objects

`grid` = *None*

values stored in the grid cells

`indexToX`(*ix*)

Parameters: **ix** – grid index in the x dimension

Returns: the real x coordinate of the center of that grid cell

`indexToY`(*iy*)

Parameters: **iy** – grid index in the y dimension
Returns: the real y coordinate of the center of that grid cell

`indicesToPoint`(*(ix, iy)*)

Parameters:
- **ix** – x index of grid cell
- **iy** – y index of grid cell

Returns: c{Point} in real world coordinates of center of cell

`makeWindow`(*windowWidth=400*, *title='Grid Map'*)

Create a window of the right dimensions representing the grid map. Store in `self.window`.

`pointToIndices`(*point*)

Parameters: **point** – real world point coordinates (instance of `Point`)
Returns: pair of (x, y) grid indices it maps into

`squareColor`(*indices*)

Default color scheme: squares that the robot can occupy are white and others are black.

`undrawPath`(*path*)

Draws list of cells using the underlying grid color scheme, effectively 'undrawing' a path. :param path: list of pairs of `(ix, iy)` grid indices

`xMax` = *None*

X coordinate of right edge

`xMin` = *None*

X coordinate of left edge

`xN` = *None*

number of cells in x dimension

`xStep` = *None*

size of a side of a cell in the x dimension

`xToIndex`(*x*)

Parameters: **x** – real world x coordinate
Returns: x grid index it maps into

`yMax` = *None*

Y coordinate of top edge

`yMin` = *None*

Y coordinate of bottom edge

`yN` = *None*

number of cells in y dimension

`yStep` = *None*

size of a side of a cell in the y dimension

`yToIndex`(*y*)

Parameters: **y** – real world y coordinate
Returns: y grid index it maps into

libdw.gw module

class `libdw.gw.`**`Color`**(*widget*, *nodes*, *accuracy*)

`color`(*scalar='foo'*)

`colormap`()

`function`(*f*)

`rgbcolor`(*tkcolor*)

`tkcolor`(*r*, *g*, *b*)

class `libdw.gw.`**`Continuous`**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

`eval`(*args*)

`type`()

class `libdw.gw.`**`Continuousset`**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

`eval`(*args*)

`type`()

class `libdw.gw.`**`Discrete`**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

`eval`(*args*)

`type`()

class `libdw.gw.`**`Function`**(*canvas*, *func*, *color*)

`setInput`(*input*)

`tryrange`(*tryarg*)

class `libdw.gw.`**`GraphCanvas`**(*parent*, *w=300*, *h=300*, *xmin=0*, *xmax=100*, *ymin=0*, *ymax=100*, *xminlabel=0*, *xmaxlabel=0*, *axeson=True*, *labelson=True*)

Bases: **`Tkinter.Canvas`**

`canvas_left_clicked_down`(*event*)

`canvas_left_clicked_up`(*event*)

`canvas_left_moved`(*event*)

`canvas_right_clicked_down`(*event*)

`clear`()

`create_dottedline`(*id*, *xa*, *ya*, *xb*, *yb*, *w*, *col*, *spacing=6*)

`draw`()

`drawAxes`()

`drawFunctions`()

`drawLabels`(*xshift=0*, *yshift=0*)

`getCx`(*px*)

`getCy`(*py*)

`getPx`(*cx*)

getPy(*cy*)

graphFunc(*f*, *mode*, *color*)

initCBounds()

initObjects()

initPointSets()

path(*id*, *pts*, *col*, *w*)

shift(*xshift*, *yshift*)

updateInput()

visible(*cx='foo'*, *cy='foo'*)

class `libdw.gw.`**GraphingWindow**(*width*, *height*, *xmin*, *xmax*, *ymin*, *ymax*, *title=''*, *parent=None*, *xminlabel=0*, *xmaxlabel=0*, *timeStamp=True*)

clear()

close()

destroy()

getDomain()

graphContinuous(*f*, *color='black'*)

graphContinuousSet(*xset*, *yset*, *color='black'*)

graphDiscrete(*f*, *color='black'*)

graphPointSet(*xset*, *yset*, *color='black'*)

graphScalarfield(*f*)

graphSlopefield(*f*, *color='black'*)

initToolbarTop()

openWindow(*width*, *height*, *xmin*, *xmax*, *ymin*, *ymax*, *title=''*, *parent=None*, *xminlabel=0*, *xmaxlabel=0*, *timeStamp=True*)

postscript(*filename*)

reopenWindow(*width*, *height*, *xmin*, *xmax*, *ymin*, *ymax*, *title=''*, *parent=None*, *xminlabel=0*, *xmaxlabel=0*, *timeStamp=True*)

resizeit(*event=None*)

save()

setDomain(*(xmin*, *xmax)*, *(ymin*, *ymax)*)

updateBoxes(*event=None*)

class `libdw.gw.`**Pointset**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

eval(*args*)

type()

class `libdw.gw.`**Scalarfield**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

eval(*args*)

type()

class `libdw.gw.`**`Slopefield`**(*canvas*, *func*, *color*)

Bases: **`libdw.gw.Function`**

`eval`(*args*)

`type`()

`libdw.gw.`**`argFor`**(*vec*, *apply*, *current=[]*)

`libdw.gw.`**`clip`**(*a*, *lo*, *hi*)

`libdw.gw.`**`scinot`**(*num*)

libdw.idealReadings module

Utility for computing ideal sonar readings

`libdw.idealReadings.`**`computeIdealReadings`**(*worldPath*, *xMin*, *xMax*, *y*, *numStates*, *numObs*)

Parameters:
- **worldPath** – string naming file to read the world description from
- **xMin** – minimum x coordinate for center of robot
- **xMax** – maximum x coordinate for center of robot
- **y** – constant y coordinate for center of robot
- **numStates** – number of discrete states into which to divide the range of x coordinates
- **numObs** – number of discrete observations into which to divide the range of good sonar observations, between 0 and `goodSonarRange`

Returns: list of `numStates` values, each of which is between 0 and `numObs-1`, which lists the ideal discretized sonar reading that the robot would receive if it were at the midpoint of each of the x bins.

`libdw.idealReadings.`**`discreteSonar`**(*d*, *numBins*)

Parameters:
- **d** – value of a sonar reading
- **numBins** – number of bins into which to divide the interval between 0 and `sonardist.sonarMax`

Returns: number of the bin into which this sonar reading should fall; any reading greater than or equal to c{sonarDist.sonarMax} is put into bin `numBins - 1`.

`libdw.idealReadings.`**`discreteSonarValue`**(*d*, *numBins*)

Parameters:
- **d** – value of a sonar reading
- **numBins** – number of bins into which to divide the interval between 0 and `sonardist.sonarMax`

Returns: number of the bin into which this sonar reading should fall; any reading greater than or equal to c{sonarDist.sonarMax} is put into bin `numBins - 1`.

`libdw.idealReadings.`**`idealSonarReading`**(*robotPose*, *sensorPose*, *world*)

Parameters:
- **robotPose** – `util.Pose` representing pose of robot in world
- **sensorPose** – c{util.Pose} representing pose of sonar sensor with respect to the robot
- **world** – `soarWorld.SoarWorld` representing obstacles in the world

Returns: length of ideal sonar reading; if the distance is longer than `sonarDist.sonarMax` or there is no hit at all, then `sonarDist.sonarMax` is returned.

libdw.io module

class `libdw.io.`**`Action`**(*fvel=0.0*, *rvel=0.0*, *voltage=5.0*)

One set of commands to send to the robot

`execute`()

class `libdw.io.`**`FakeSensorInput`**(*sonars, odometry, analogInputs=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]*)

Fake version that takes values at init time Represents one set of sensor readings from the robot, incluing sonars, odometry, and readings from the analogInputs

`sonars` = *None*

List of 6 sonar readings, in meters.

libdw.layout module

`libdw.layout.`**`CoordinateRangeX`**(*clist*)

`libdw.layout.`**`MoveX`**(*clist, xleft, xright, offset*)

`libdw.layout.`**`ShiftLeft`**()

`libdw.layout.`**`ShiftRight`**()

`libdw.layout.`**`Simulate`**()

`libdw.layout.`**`addComponent`**(*c, canvas*)

`libdw.layout.`**`boundtocanvas`**(*x, y*)

`libdw.layout.`**`bus`**(*i, j*)

`libdw.layout.`**`busLine`**(*y, a, color*)

`libdw.layout.`**`bussed`**(*i*)

`libdw.layout.`**`clear`**()

`libdw.layout.`**`clearnew`**(*resetflag*)

class `libdw.layout.`**`component`**

`erase`()

`move`(*dx, dy*)

`libdw.layout.`**`cresButton`**(*event*)

`libdw.layout.`**`cresEnter`**(*event*)

`libdw.layout.`**`cresLeave`**(*event*)

`libdw.layout.`**`directoryonly`**(*filename*)

`libdw.layout.`**`drawConnector`**(*z*)

`libdw.layout.`**`drawNewResistor`**()

`libdw.layout.`**`drawProtoboard`**()

`libdw.layout.`**`fHeadButton`**(*event*)

`libdw.layout.`**`fHeadEnter`**(*event*)

`libdw.layout.`**`fHeadLeave`**(*event*)

`libdw.layout.`**`fMotorButton`**(*event*)

`libdw.layout.`**`fMotorEnter`**(*event*)

`libdw.layout.`**`fMotorLeave`**(*event*)

`libdw.layout.`**`fampButton`**(*event*)

`libdw.layout.`**`fampEnter`**(*event*)

`libdw.layout.`**`fampLeave`**(*event*)

class `libdw.layout.`**`fhead`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

`libdw.layout.`**`filenameonly`**(*filename*)

class `libdw.layout.`**`fmeter`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

class `libdw.layout.`**`fmotor`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

class `libdw.layout.`**`fopamp`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

class `libdw.layout.`**`fpot`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

`libdw.layout.`**`fpotButton`**(*event*)

`libdw.layout.`**`fpotEnter`**(*event*)

`libdw.layout.`**`fpotLeave`**(*event*)

class `libdw.layout.`**`fpower`**(*z*)

Bases: **`libdw.layout.component`**

`add`(*canvas*)

`highlight`()

`inside`(*x*, *y*)

class `libdw.layout.`**`frobot`**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

libdw.layout.**getChanged**()

libdw.layout.**getLabel**(*prefix*)

libdw.layout.**grid**(*i*, *j*)

libdw.layout.**gridx**(*i*)

libdw.layout.**gridy**(*j*)

libdw.layout.**hresButton**(*event*)

libdw.layout.**hresEnter**(*event*)

libdw.layout.**hresLeave**(*event*)

class libdw.layout.**hresistor**(*z*, *c1*, *c2*, *c3*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

in1(*x*, *y*)

in2(*x*, *y*)

in3(*x*, *y*)

inside(*x*, *y*)

libdw.layout.**iRobotButton**(*event*)

libdw.layout.**iRobotEnter**(*event*)

libdw.layout.**iRobotLeave**(*event*)

libdw.layout.**iampButton**(*event*)

libdw.layout.**iampEnter**(*event*)

libdw.layout.**iampLeave**(*event*)

libdw.layout.**igrid**(*x*)

class libdw.layout.**ihead**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

libdw.layout.**ijgrid**(*x*, *y*)

class libdw.layout.**imeter**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

class libdw.layout.**imotor**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

class libdw.layout.**iopamp**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

class libdw.layout.**ipot**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

libdw.layout.**ipotButton**(*event*)

libdw.layout.**ipotEnter**(*event*)

libdw.layout.**ipotLeave**(*event*)

class libdw.layout.**ipower**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

class libdw.layout.**irobot**(*z*)

Bases: **libdw.layout.component**

add(*canvas*)

highlight()

inside(*x*, *y*)

libdw.layout.**isDuplicate**(*c*, *clist*)

libdw.layout.**jgrid**(*y*)

libdw.layout.**keyPress**(*event*)

libdw.layout.**keyRelease**(*event*)

libdw.layout.**label**(*z*, *a*)

libdw.layout.**meterButton**(*event*)

`libdw.layout.`**`meterEnter`**(*event*)

`libdw.layout.`**`meterLeave`**(*event*)

class `libdw.layout.`**`movingState`**

- **`move`**(*event*)
- **`push`**(*event*)
- **`release`**(*event*)

`libdw.layout.`**`openFile`**()

`libdw.layout.`**`pin`**(*i*, *j*)

`libdw.layout.`**`powerButton`**(*event*)

`libdw.layout.`**`powerEnter`**(*event*)

`libdw.layout.`**`powerLeave`**(*event*)

`libdw.layout.`**`quit`**()

`libdw.layout.`**`readFile`**(*filename*)

`libdw.layout.`**`removeComponent`**(*c*)

`libdw.layout.`**`revert`**()

`libdw.layout.`**`save`**()

`libdw.layout.`**`saveAs`**()

`libdw.layout.`**`setChanged`**(*change*)

`libdw.layout.`**`unDo`**()

`libdw.layout.`**`vresButton`**(*event*)

`libdw.layout.`**`vresEnter`**(*event*)

`libdw.layout.`**`vresLeave`**(*event*)

class `libdw.layout.`**`vresistor`**(*z*, *c1*, *c2*, *c3*)

Bases: **`libdw.layout.component`**

- **`add`**(*canvas*)
- **`highlight`**()
- **`in1`**(*x*, *y*)
- **`in2`**(*x*, *y*)
- **`in3`**(*x*, *y*)
- **`inside`**(*x*, *y*)

class `libdw.layout.`**`wire`**(*z0*, *z1*)

Bases: **`libdw.layout.component`**

- **`add`**(*canvas*)
- **`highlight`**()
- **`inside`**(*x*, *y*)
- **`move`**(*dx*, *dy*)

nearend(*x*, *y*)

render()

`libdw.layout.`**yremap**(*y*)

libdw.le module

Specify and solve systems of linear equations.

class `libdw.le.`**Equation**(*coeffs*, *variableNames*, *constant*)

Represent a single linear equation as a list of variable names, a list of coefficients, and a constant. Assume the coeff * var terms are on the left of the equality and the constant is on the right.

coeffs = *None*

List of coefficients in the same order as the variable names

constant = *None*

Constant (right hand side)

variableNames = *None*

List of variable names

class `libdw.le.`**EquationSet**

Represent a set of linear equations

addEquation(*eqn*)

Parameters: **eqn** – instance of `Equation` Adds it to the set

addEquations(*eqns*)

Parameters: **eqns** – list of instances of `Equation` Adds them to the set

equations = *None*

List of instances of `Equation`.

solve()

Returns: an instance of `Solution`

class `libdw.le.`**NameToIndex**

Construct a unique mapping of names to indices. Every time a new name is inserted, it is assigned a new index. Indices start at 0 and increment by 1. For example:

```
>>> n2n = nameToIndex()
>>> n2n.insert('n1')
>>> n2n.insert('n2')
>>> n2n.insert('n1')   # has no effect since it is a duplicate
>>> n2n.lookup('n1')
0
>>> n2n.names()
['n1', 'n2']
```

insert(*name*)

If `name` has been inserted before, do nothing. Otherwise, assign it the next index.

lookup(*name*)

Returns the index associated with `name`. Generates an error if it `name` has not previously been inserted.

names()

Returns list of names that have been inserted so far, in the order they were inserted.

namesList = *None*

List of names in order of insertion

namesToNums = *None*

Dictionary mapping names to their assigned indices

nextIndex = *None*

The next index to be allocated.

class `libdw.le.`**Solution**(*n2i*, *values*)

Solution to a set of linear equations

n2i = *None*

Mapping from variable names to indices, an instance of the NameToIndex class

translate(*name*)

Returns: the value of variable `name` in the solution

values = *None*

List of values of the variables, in order of their indices

libdw.leNumpy module

class `libdw.leNumpy.`**Equation**(*coeffs*, *variableNames*, *constant*)

Represent a single linear equation as a list of variable names, a list of coefficients, and a constant. Assume the coeff * var terms are on the left of the equality and the constant is on the right.

coeffs = *None*

List of coefficients in the same order as the variable names

constant = *None*

Constant (right hand side)

variableNames = *None*

List of variable names

class `libdw.leNumpy.`**EquationSet**

Represent a set of linear equations

addEquation(*eqn*)

Parameters: **eqn** – instance of `Equation`

Adds it to the set

addEquations(*eqns*)

Parameters: **eqns** – list of instances of `Equation`

Adds them to the set

equations = *None*

List of instances of `Equation`.

solve()

Returns: an instance of `Solution`

class `libdw.leNumpy.`**`NameToIndex`**

Construct a unique mapping of names to indices. Every time a new name is inserted, it is assigned a new index. Indices start at 0 and increment by 1. For example:

```
>>> n2n = nameToIndex()
>>> n2n.insert('n1')
>>> n2n.insert('n2')
>>> n2n.insert('n1')   # has no effect since it is a duplicate
>>> n2n.lookup('n1')
0
>>> n2n.names()
['n1', 'n2']
```

`insert`(*name*)

If `name` has been inserted before, do nothing. Otherwise, assign it the next index.

`lookup`(*name*)

Returns the index associated with `name`. Generates an error if it `name` has not previously been inserted.

`names`()

Returns list of names that have been inserted so far.

`namesToNums` = *None*

Dictionary mapping names to their assigned indices

`nextIndex` = *None*

The next index to be allocated.

class `libdw.leNumpy.`**`Solution`**(*n2i*, *values*)

Solution to a set of linear equations

`n2i` = *None*

Mapping from variable names to indices

`translate`(*name*)

Returns: the value of variable `name` in the solution

`values` = *None*

List of values of the variables, in order of their indices

libdw.lineLocalize module

class `libdw.lineLocalize.`**`PreProcess`**(*numObservations*, *stateWidth*)

Bases: **`libdw.sm.SM`**

State machine that takes, as input, instances of { t io.SensorInput} and generates as output pairs of (observation, input). The observation is a discretized sonar reading from time t-1; the input is an action, which is a discretized distance, computed from the difference between the x coordinate of the robot at time t and at time t-1.

`getNextValues`(*state*, *inp*)

class `libdw.lineLocalize.`**`SensorInput`**(*sonars*, *odometry*)

`libdw.lineLocalize.`**`discreteAction`**(*oldPose*, *newPose*, *stateWidth*)

`libdw.lineLocalize.`**`makeLineLocalizer`**(*numObservations*, *numStates*, *ideal*, *xMin*, *xMax*, *robotY*)

Create behavior controlling robot to move in a line and to
localize itself in one dimension

Parameters:
- **numObservations** – number of discrete observations into which to divide the range of good sonar observations, between 0 and `goodSonarRange`
- **numStates** – number of discrete states into which to divide the range of x coordinates
- **ideal** – list of ideal sonar readings
- **xMin** – minimum x coordinate for center of robot
- **xMax** – maximum x coordinate for center of robot
- **robotY** – constant y coordinate for center of robot

Returns: an instance of { t sm.SM} that implements the behavior

`libdw.lineLocalize.`**`makeMetricMachine`**(*xMin*, *xMax*, *y*, *numStates*)

Parameters:
- **xMin** – minimum x coordinate for center of robot
- **xMax** – maximum x coordinate for center of robot
- **y** – constant y coordinate for center of robot
- **numStates** – number of discrete states into which to divide the range of x coordinates

Returns: a state machine that takes two inputs: `(sensorInput, belief)`, where `sensorInput` is the true sensor input (we can trust the pose in simulation to be the actual truth) and `belief` is a distribution over possible discrete robot locations, delivered by the state estimator. The state machine can deliver a metric (averaged over time) as output; it should also print the metric value on each step.

`libdw.lineLocalize.`**`makeRobotNavModel`**(*ideal*, *xMin*, *xMax*, *numStates*, *numObservations*)

Create a model of a robot navigating in a 1 dimensional world with a single sonar.

Parameters:
- **ideal** – list of ideal sonar readings
- **xMin** – minimum x coordinate for center of robot
- **xMax** – maximum x coordinate for center of robot
- **numStates** – number of discrete states into which to divide the range of x coordinates
- **numObservations** – number of discrete observations into which to divide the range of good sonar observations, between 0 and `goodSonarRange`

Returns: an instance of { t ssm.StochasticSM} that describes the dynamics of the world

libdw.ltism module

State machines that are representable as LTI systems.

class `libdw.ltism.`**`LTISM`**(*dCoeffs*, *cCoeffs*, *previousInputs=None*, *previousOutputs=None*)

Bases: **`libdw.sm.SM`**

Class of state machines describable as LTI systems

`cCoeffs` = *None*

Output coefficients

`dCoeffs` = *None*

Input coefficients

`getNextValues`(*state*, *input*)

`startState` = *None*

State is last j input values and last k output values

libdw.mapMaker module

class `libdw.mapMaker.MapMaker`(*xMin*, *xMax*, *yMin*, *yMax*, *gridSquareSize*, *useClearInfo=False*, *useCountingMap=False*, *useBayesMap=True*)

Bases: **`libdw.sm.SM`**

It violates the state machine protocol because it changes the grid map by side effect, rather than making a fresh copy each time.

`clearUnderRobot`(*grid*, *robotPose*)

`getNextValues`(*state*, *inp*)

Parameters:
- **inp** – instance of `SensorInput`
- **state** – is `grid`

Modifies grid

`processSonarReadings`(*grid*, *robotPose*, *sonars*)

For each reading that is less than the reliable length, set the point at the end to be occupied and the points along the ray up to that point to be free.

class `libdw.mapMaker.SensorInput`(*sonars*, *odometry*)

`libdw.mapMaker.testMapMaker`(*data*)

`libdw.mapMaker.testMapMakerClear`(*data*)

`libdw.mapMaker.testMapMakerN`(*n*, *data*)

libdw.move module

Drive robot to goal specified as odometry pose.

class `libdw.move.MoveToDynamicPoint`

Bases: **`libdw.sm.SM`**

Drive to a goal point in the frame defined by the odometry. Goal points are part of the input, in contrast to `MoveToFixedPoint`, which takes a single goal point at initialization time.

Assume inputs are `(util.Point, io.SensorInput)` pairs

This is really a pure function machine; defining its own class, though, so we can easily modify the parameters.

`angleEps` = *0.1*

Tolerance for angles

`forwardGain` = *1.0*

Gain for driving forward

`getNextValues`(*state*, *inp*)

`maxVel` = *0.5*

Maximum velocity

`rotationGain` = *0.5*

Gain for rotating

class `libdw.move.MoveToFixedPoint`(*goalPoint*, *maxVel=0.5*)

Bases: **libdw.sm.SM**

State machine representing robot behavior that drives to a specified point. Inputs are instances of `io.SensorInput`; outputs are instances of `io.Action`. Robot first rotates toward goal, then moves straight. It will correct its rotation if necessary.

angleEps = *0.05*
Tolerance for angles

distEps = *0.05*
Tolerance for distances

done(*state*)

forwardGain = *1.0*
Gain for driving forward

getNextValues(*state*, *inp*)

maxVel = *0.5*
Maximum velocity

rotationGain = *1.0*
Gain for rotating

startState = *False*

class libdw.move.**MoveToFixedPose**(*goalPose*, *maxVel=0.5*)
Bases: **libdw.sm.SM**

State machine representing robot behavior that drives to a specified pose. Inputs are instances of `io.SensorInput`; outputs are instances of `io.Action`. Robot first rotates toward goal, then moves straight, then rotates to desired final angle.

angleEps = *0.05*
Tolerance for angles

distEps = *0.05*
Tolerance for distances

done(*state*)

forwardGain = *1.0*
Gain for driving forward

getNextValues(*state*, *inp*)

maxVel = *0.5*
Maximum velocity

rotationGain = *1.0*
Gain for rotating

startState = *False*

libdw.move.**actionToPoint**(*goalPoint*, *robotPose*, *forwardGain*, *rotationGain*, *maxVel*, *angleEps*)
Internal procedure that returns an action to take to drive toward a specified goal point.

libdw.move.**actionToPose**(*goalPose*, *robotPose*, *forwardGain*, *rotationGain*, *maxVel*, *angleEps*, *distEps*)
Internal procedure that returns an action to take to drive toward a specified goal pose.

libdw.nlcc module

class `libdw.nlcc.`**`Circuit`**(*components*)

`addConstituentConstraints`(*constraintSet*)

`addKCLConstraints`(*constraintSet*, *groundNode*)

`displaySolution`(*groundNode='gnd'*)

`makeConstraintSet`(*groundNode*)

`makeEquationSet`(*groundNode*)

class `libdw.nlcc.`**`CircuitNode`**

`addConnection`(*sign*, *currentName*)

class `libdw.nlcc.`**`Component2Leads`**(*n1*, *n2*)

`addKCLToNodes`(*nodeDict*)

class `libdw.nlcc.`**`OpAmp`**(*n1*, *n2*, *n3*, *K=1000*, *Vcc=10*, *Vss=0*)

`addKCLToNodes`(*nodeDict*)

`constraintFn`()

class `libdw.nlcc.`**`Resistor`**(*resistance*, *n1*, *n2*)

Bases: **`libdw.nlcc.Component2Leads`**

`constraintFn`()

class `libdw.nlcc.`**`SymbolGenerator`**

`gensym`(*prefix='i'*)

class `libdw.nlcc.`**`VSrc`**(*voltage*, *n1*, *n2*)

Bases: **`libdw.nlcc.Component2Leads`**

`constraintFn`()

class `libdw.nlcc.`**`Wire`**(*n1*, *n2*)

Bases: **`libdw.nlcc.Component2Leads`**

`constraintFn`()

`libdw.nlcc.`**`kcl`**(*signs*)

`libdw.nlcc.`**`setGround`**(*x*)

libdw.nleNumpy module

class `libdw.nleNumpy.`**`ConstraintSet`**

`FdF`(*x*, *F*, *JF*)

`addConstraint`(*f*, *variables*)

`display`(*solution*)

`getConstraintEvaluationFunction`()

`listVariables`()

`solve`()

`translate`(*variable*, *solution*)

class `libdw.nleNumpy.`**`Solution`**(*n2i*, *values*)

`n2i` = *None*

Mapping from variable names to indices

`translate`(*name*)

Returns: the value of variable `name` in the solution

`values` = *None*

List of values of the variables, in order of their indices

`libdw.nleNumpy.`**`compute_fdf`**(*f*, *vals*)

class `libdw.nleNumpy.`**`name2num`**(*variable_list=[]*)

`max_num`()

`names`()

`libdw.nleNumpy.`**`resolveConstraints`**(*fdf*, *maxiters=100*)

libdw.noInput module

`libdw.noInput.`**`runTest`**(*lines*, *parent=None*, *nsteps=70*)

`libdw.noInput.`**`testSignal`**(*simTime=3.0*)

libdw.oneStep module

`libdw.oneStep.`**`runTest`**(*lines*, *parent=None*, *nsteps=50*)

`libdw.oneStep.`**`testSignal`**(*simTime=1.0*)

libdw.optimize module

Procedures for finding values of a function to optimize its output.

`libdw.optimize.`**`argopt`**(*f*, *stuff*, *comp*)

Parameters:
- **f** – a function that takes a single argument of some type `x` and returns a value of some type `y`
- **stuff** – a list of elements of type `x`
- **comp** – a function that takes two arguments of type `y` and returns a Boolean; it is intended to return `True` if the first argument is 'better' than the second.

Returns: a pair `(bestVal, bestArg)`, where `bestArg` is the element of `stuff` such that `f(bestArg)` is better, according to `comp` than `f` applied to any other element of `stuff`, and `bestVal` is `f(bestArg)`.

The types `x` and `y` are not actual types; they're just intended to show that the types of the functions have to match up in the right way.

For example, get the team with the highest score, you might do something like

```
argopt(seasonScore, ['ravens', 'crows', 'buzzards'], operator.gt)
```

where `seasonScore` is a function that takes the name of a team and returns a numerical score.

`libdw.optimize.`**`floatRange`**(*lo*, *hi*, *stepsize*)

Returns: a list of numbers, starting with `lo`, and increasing by `stepsize` each time, until `hi` is equaled or exceeded.

`lo` must be less than `hi`; `stepsize` must be greater than 0.

`libdw.optimize.`**`optOverGrid`**(*objective*, *xmin*, *xmax*, *numXsteps*, *ymin*, *ymax*, *numYsteps*, *compare=<built-in function lt>*)

Like `optOverLine`, but `objective` is now a function from two numerical values, one chosen from the `x` range and one chosen from the `y` range. It returns `(objective(x, y), (x, y))` for the optimizing pair `(x,y)`.

`libdw.optimize.`**`optOverLine`**(*objective*, *xmin*, *xmax*, *numXsteps*, *compare=<built-in function lt>*)

Parameters:
- **objective** – a function that takes a single number as an argument and returns a value
- **compare** – a function from two values (of the type returned by `objective`) to a Boolean; should return `True` if we like the first argument better.

Returns: a pair, `(objective(x), x)`. `x` one of the numeric values achieved by starting at `xmin` and taking `numXsteps` equal-sized steps up to `xmax`; the particular value of `x` returned is the one for which `objective(x)` is best, according to the `compare` operator.

libdw.poly module

Polynomials, with addition, multiplication, and roots.

class `libdw.poly.`**`Polynomial`**(*coeffs*)

Represent polynomials, and supports addition, subtraction, and root finding.

`add`(*p1*, *p2*)

Parameters: **p2** (*p1,*) – polynomials

Returns: a new polynomial, which is their sum. Does not affect either input.

`coeff`(*i*)

`coeffs` = *None*

List of coefficients of the polynomial, highest order first

`mul`(*p1*, *p2*)

Parameters: **p2** (*p1,*) – polynomials

Returns: a new polynomial, which is their product.

Does not affect either input.

`order` = *None*

Order of the polynomial; one less than the number of coeffs

`roots`()

Returns: list of the roots, found by numpy

`scalarMult`(*s*)

Parameters: **s** – a scalar

Returns: a new polynomial with all coefficients of self, multiplied by s

`shift`(*p*, *a*)

Parameters: **a** – integer

Returns: a new polynomial, multiplied by x**a.

Just adds zeros for new low-order coefficients.

`val`(*x*)

Parameters: **x** – number

Returns: the value of the polynomial with the variable assigned to x.

`libdw.poly.`**`assertSameLength`**(*a*, *b*)

Generate an error if the arguments do not have the same length `libdw.poly.`**`fixType`**(*n*)

If n is an integer, convert to a float, but leave complex as complex.

`libdw.poly.`**`prettyNum`**(*value*)

`libdw.poly.`**`prettyTerm`**(*coefficient*, *power*, *var='z'*)

`libdw.poly.`**`vectorAdd`**(*a*, *b*)

Parameters: b (*a*,) – lists of numbers of the same length
Returns: (a[1]+b[1], ..., a[n]+b[n])

libdw.replanner module

State machine classes for planning paths in a grid map.

class `libdw.replanner.`**`Replanner`**(*goalPoint*, *worldPath*, *gridSquareSize*, *mapClass*)

Bases: **`libdw.sm.SM`**

This replanner state machine has a fixed map, which it constructs at initialization time. Input to the machine is an instance of `io.SensorInput`; output is an instance of `util.Point`, representing the desired next subgoal. The planner should guarantee that a straight-line path from the current pose to the output pose is collision-free.

`getNextValues`(*state*, *inp*)

class `libdw.replanner.`**`ReplannerWithDynamicMap`**(*goalPoint*, *useCostDynamics=False*)

Bases: **`libdw.sm.SM`**

This replanner state machine has a dynamic map, which is an input to the state machine. Input to the machine is a pair `(map, sensors)`, where `map` is an instance of a subclass of `gridMap.GridMap` and `sensors` is an instance of `io.SensorInput`; output is an instance of `util.Point`, representing the desired next subgoal. The planner should guarantee that a straight-line path from the current pose to the output pose is collision-free in the current map.

`getNextValues`(*state*, *inp*)

class `libdw.replanner.`**`ReplannerWithDynamicMapAndGoal`**(*useCostDynamics=False*)

Bases: **`libdw.sm.SM`**

This replanner state machine has a dynamic map and a dynamic goal, both of which are inputs to the state machine. Input to the machine is a structure `(goal, (map, sensors))`, where `map` is an instance of a subclass of `gridMap.GridMap`, `goal` is an instance of `util.Point`, and `sensors` is an instance of `io.SensorInput`; output is an instance of `util.Point`, representing the desired next subgoal. The planner should guarantee that a straight-line path from the current pose to the output pose is collision-free in the current map.

`getNextValues`(*state*, *inp*)

`libdw.replanner.`**`adjacent`**(*(x1, y1)*, *(x2, y2)*)

`libdw.replanner.`**`newPathAndSubgoal`**(*worldMap*, *sensorInput*, *goalPoint*, *dynamicsModel*, *path*, *timeToReplan*, *scale=1*)

This procedure does the primary work of both replanner classes. It tests to see if the current plan is empty or invalid. If so, it calls the planner to make a new plan. Then, given a plan, if the robot has reached the first grid cell in the plan, it removes that grid cell from the front of the plan. Finally, it gets the the center of the current first grid-cell in the plan, in odometry coordinates, and generates that as the subgoal.

It uses a heuristic in the planning, which is the Cartesian distance between the current location of the robot in odometry coordinates (determined by finding the center of the grid square) and the goal location.

Whenever a new plan is made, it is drawn into the map. Whenever a subgoal is achieved, it is removed from the path drawn in the map.

Parameters:
- **worldMap** – instance of a subclass of `gridMap.GridMap`
- **sensorInput** – instance of `io.SensorInput`, containing current robot pose
- **goalPoint** – instance of `util.Point`, specifying goal
- **dynamicsModel** – a state machine that specifies the transition dynamics for the robot in the grid map
- **path** – the path (represented as a list of pairs of indices in the map) that the robot is currently following. Can be `None` or `[]`.
- **timeToReplan** – a procedure that takes `path`, the robot's current indices in the grid, the map, and the indices of the goal, and returns `True` or `False` indicating whether a new plan needs to be constructed.

Returns: a tuple `(path, subgoal)`, where `path` is a list of pairs of indices indicating a path through the grid, and `subgoal` is an instance of `util.Point` indicating the point in odometry coordinates that the robot should drive to.

`libdw.replanner.`**`planInvalidInMap`**(*map*, *state*)

Just checks to be sure the first two cells are occupiable. In low noise conditions, it's good to check the whole plan, so failures are discovered earlier; but in high noise, we often have to get close to a location before we decide that it is really not safe to traverse.

`libdw.replanner.`**`timeToReplanDynamicMap`**(*plan*, *currentIndices*, *map*, *goalIndices*)

Replan if the current plan is `None`, if the plan is invalid in the map (because it is blocked), or if the plan is empty and we are not at the goal (which implies that the last time we tried to plan, we failed).

`libdw.replanner.`**`timeToReplanDynamicMapAndGoal`**(*plan*, *currentIndices*, *map*, *goalIndices*)

Replan if the current plan is `None`, if the plan is invalid in the map (because it is blocked), if the plan is empty and we are not at the goal (which implies that the last time we tried to plan, we failed), or if the end of the plan is not the same as the goal indices (which means the goal changed).

`libdw.replanner.`**`timeToReplanDynamicMapWithKidnap`**(*state*, *currentIndices*, *map*, *goalIndices*)

Replan if the current plan is `None`, if the plan is invalid in the map (because it is blocked), or if the robot is not in a grid cell that is adjacent to the first one in the plan.

`libdw.replanner.`**`timeToReplanStaticMap`**(*plan*, *currentIndices*, *worldMap*, *goalIndices*)

When the map is static, we just test for kidnapping. Replan if the current plan is `None` or if the robot is not in a grid cell that is adjacent to the first one in the plan.

libdw.replannerRace module

State machine classes for planning paths in a grid map.

class `libdw.replannerRace.`**`ReplannerWithDynamicMap`**(*goalPoint*)

Bases: **`libdw.sm.SM`**

This replanner state machine has a dynamic map, which is an input to the state machine. Input to the machine is a pair `(map, sensors)`, where `map` is an instance of a subclass of `gridMap.GridMap` and `sensors` is an instance of `io.SensorInput`; output is an instance of `util.Point`, representing the desired next subgoal. The planner should guarantee that a straight-line path from the current pose to the output pose is collision-free in the current map.

`getNextValues`(*state*, *inp*)

`startState` = *None*

State is the plan currently being executed. No plan to start with.

`libdw.replannerRace.planInvalidInMap`(*map*, *plan*, *currentIndices*)

Checks to be sure all the cells between the robot's current location and the first subgoal in the plan are occupiable. In low-noise conditions, it's useful to check the whole plan, so failures are discovered earlier; but in high noise, we often have to get close to a location before we decide that it is really not safe to traverse.

We actually ignore the case when the robot's current indices are occupied; during mapMaking, we can sometimes decide the robot's current square is not occupiable, but we should just keep trying to get out of there.

`libdw.replannerRace.timeToReplan`(*plan*, *currentIndices*, *map*, *goalIndices*)

Replan if the current plan is `None`, if the plan is invalid in the map (because it is blocked), or if the plan is empty and we are not at the goal (which implies that the last time we tried to plan, we failed).

libdw.se module

State machine that acts as a state estimator, given a world model expressed as a c{ssm.StochasticSM}.

class `libdw.se.StateEstimator`(*model*, *verbose=False*)

Bases: **`libdw.sm.SM`**

A state machine that performs state estimation, based on an input stream of (observation, input) pairs)and a stochastic state-machine model. The output at time t is a `dist.DDist` object, representing the 'belief' distribution P(s | i_0, ... i_t, o_0, ..., o_t)

`getNextValues`(*state*, *inp*)

Parameters:
- **state** – Distribution over states of the subject machine, represented as a `dist.Dist` object
- **inp** – A pair `(o, i)` of the observation (output) and input of the subject machine on this time step.

`startState` = *None*

The state of this machine is the same as its output: the distribution over states of the subject machine given the input sequence so far; the start state of this machine is the starting distribution of the subject machine.

class `libdw.se.StateEstimatorTriggered`(*model*, *verbose=False*)

Bases: **`libdw.se.StateEstimator`**

Like `StateEstimator`, but the inputs are `(observation, action, trigger)`. If `trigger` is `True` then do the state update, otherwise, just pass the state through. Output is belief state, and a boolean indicating whether an update was just done.

`getNextValues`(*state*, *inp*)

`libdw.se.makeStateEstimationSimulation`(*worldSM*, *verbose=False*)

Make a machine that simulates the state estimation process. It takes a state machine representing the world, at construction time. Let i be an input to the world machine. The input is fed into the world machine, generating (stochastically) an output, o. The (o, i) pair is fed into a state-estimator using worldSM as its model. The output of the state estimator is a belief state, b. The output of this entire composite machine is (b, (o, i)).

Parameters: **worldSM** – an instance of `ssm.StochasticSM`
Returns: a state machine that simulates the world and executes the state estimation process.

libdw.seFast module

Just like se, but a more efficient implementation

class `libdw.seFast.StateEstimator`(*model*)

Bases: **`libdw.sm.SM`**

A state machine that performs state estimation, based on an input stream of (input, output pairs) and a stochastic state-machine model. The output at time t is a `dist.DDist` object, representing the 'belief' distribution P(s | i_0, … i_t, o_0, …, o_t)

`getNextValues`(*state*, *inp*)

Parameters:
- **state** – Distribution over states of the subject machine, represented as a `dist.Dist` object
- **inp** – A pair `(o, a)` of the input and output of the subject machine on this time step. If this parameter is `None`, then no update occurs and the state is returned, unchanged.

`startState` = *None*

The state of this machine is the same as its output: the distribution over states of the subject machine given the input sequence so far; the start state of this machine is the starting distribution of the subject machine.

libdw.seGraphics module

State estimator that calls procedures for visualization or debugging

class `libdw.seGraphics.`**`StateEstimator`**(*model*)

Bases: **`libdw.seFast.StateEstimator`**

By default, this is the same as `seFast.StateEstimator`. If the attributes `observationHook` or `beliefHook` are defined, then as well as doing `getNextValues` from `seFast.StateEstimator`, it calls the hooks.

`getNextValues`(*state*, *inp*)

`libdw.seGraphics.`**`beliefHook`** = *None*

Procedure that takes one argument, a belief distribution, and does some useful display. If `None`, then no display is done.

`libdw.seGraphics.`**`observationHook`** = *None*

Procedure that takes two arguments, an observation and an observation model, and does some useful display. If `None`, then no display is done.

libdw.search module

Procedures and classes for doing basic breadth-first and depth-first search, with and without dynamic programming.

class `libdw.search.`**`Queue`**

Simple implementation of queue using a Python list.

`isEmpty`()

Returns `True` if the queue is empty and `False` otherwise.

`pop`()

Return the oldest item that has not yet been popped, and removes it from the queue.

`push`(*item*)

Push `item` onto the queue.

class `libdw.search.`**`SearchNode`**(*action*, *state*, *parent*)

A node in a search tree

`action` = *None*

Action that moves from `parent` to `state`

inPath(*s*)

Returns: `True` if state `s` is in the path from here to the root

path()

Returns: list of `(action, state)` pairs from root to this node

class `libdw.search.`**`Stack`**

Simple implementation of stack using a Python list.

isEmpty()

Returns `True` if the stack is empty and `False` otherwise.

pop()

Return the most recently pushed item that has not yet been popped, and removes it from the stack.

push(*item*)

Push `item` onto the stack.

`libdw.search.`**`breadthFirst`**(*initialState*, *goalTest*, *actions*, *successor*)

See `search` documentation

`libdw.search.`**`breadthFirstDP`**(*initialState*, *goalTest*, *actions*, *successor*)

See `search` documentation

`libdw.search.`**`depthFirst`**(*initialState*, *goalTest*, *actions*, *successor*)

See `search` documentation

`libdw.search.`**`depthFirstDP`**(*initialState*, *goalTest*, *actions*, *successor*)

See `search` documentation

`libdw.search.`**`pathValid`**(*smToSearch*, *path*)

Parameters:
- **smToSearch** – instance of `sm.SM` defining a search domain
- **path** – list of the form `[(a0, s0), (a1, s1), (a2, s2), ...]` where the `a`'s are legal actions of c{smToSearch} and `s`'s are states of that machine.

Returns: `True` if taking action a1 in state s0 results in state s1, taking action a2 in state s1 results in state s2, etc. That is, if this path through the state space is executable in this state machine.

`libdw.search.`**`search`**(*initialState*, *goalTest*, *actions*, *successor*, *depthFirst=False*, *DP=True*, *maxNodes=10000*)

Parameters:
- **initialState** – root of the search
- **goalTest** – function from state to Boolean
- **actions** – a list of possible actions
- **successor** – function from state and action to next state
- **depthFirst** – do depth-first search if `True`, otherwise do breadth-first
- **DP** – do dynamic programming if `True`, otherwise not
- **maxNodes** – kill the search after it expands this many nodes

Returns: path from initial state to a goal state as a list of (action, state) tuples

`libdw.search.`**`smSearch`**(*smToSearch*, *initialState=None*, *goalTest=None*, *maxNodes=10000*, *depthFirst=False*, *DP=True*)

Parameters:
- **smToSearch** – instance of `sm.SM` defining a search domain; `getNextValues` is used to determine the successor of a state given an action
- **initialState** – initial state for the search; if not provided, will use `smToSearch.startState`
- **goalTest** – function that takes a state as an argument and returns `True` if it is a goal state, and `False` otherwise

- **maxNodes** – maximum number of nodes to be searched; prevents runaway searches
- **depthFirst** – if `True`, use depth first search; usually not a good idea
- **DP** – if `True`, use dynamic programming; usually a good idea

Returns: a list of the form `[(a0, s0), (a1, s1), (a2, s2), ...]` where the a's are legal actions of c{smToSearch} and s's are states of that machine. `s0` is the start state; the last state is a state that satisfies the goal test. If the goal is unreachable (within the search limit), it returns `None`.

`libdw.search.`**`somewhatVerbose`** = *False*

If `True`, prints a trace of the search

`libdw.search.`**`verbose`** = *False*

If `True`, prints a verbose trace of the search

libdw.searchTest module

class `libdw.searchTest.`**`EightPuzzleSM`**(*goal*)

Bases: **`libdw.sm.SM`**

`done`(*state*)

`getNextValues`(*state, action*)

`legalInputs` = *[(1, 0), (-1, 0), (0, 1), (0, -1)]*

`nextState`(*state, action*)

`startState` = *(((2, 8, 3), (1, 6, 4), (7, None, 5)), (2, 1))*

`libdw.searchTest.`**`HN1`**(*s, g*)

`libdw.searchTest.`**`HN2`**(*s, g*)

`libdw.searchTest.`**`NH`**(*s, g*)

class `libdw.searchTest.`**`NumberTestCostSM`**(*goal*)

Bases: **`libdw.sm.SM`**

`done`(*state*)

`getNextValues`(*state, action*)

`legalInputs` = *['x*2', 'x+1', 'x-1', 'x**2', '-x']*

`nextState`(*state, action*)

`startState` = *1*

class `libdw.searchTest.`**`NumberTestFiniteSM`**(*goal, maxVal*)

Bases: **`libdw.searchTest.NumberTestSM`**

`getNextValues`(*state, action*)

class `libdw.searchTest.`**`NumberTestSM`**(*goal*)

Bases: **`libdw.sm.SM`**

`done`(*state*)

`getNextValues`(*state, action*)

`legalInputs` = *['x*2', 'x+1', 'x-1', 'x**2', '-x']*

`nextState`(*state, action*)

startState = *1*

libdw.searchTest.**bar**(*s*, *g*)

libdw.searchTest.**big8**()

libdw.searchTest.**bigTest**(*s*, *g*)

libdw.searchTest.**eightTest**(*s*, *g*, *h*)

libdw.searchTest.**foo**(*s*, *g*)

libdw.searchTest.**h0**(*s*, *g*)

libdw.searchTest.**h1**(*s*, *g*)

libdw.searchTest.**h2**(*s*, *g*)

libdw.searchTest.**mapD**(*s*, *g*)

libdw.searchTest.**mapDistTest**(*map*, *start*, *goal*, *searchFn=<function search at 0x000000000BED16D8>*, *h=<function <lambda> at 0x00000000084A1CF8>*)

libdw.searchTest.**mapTest**(*map*, *start*, *goal*, *searchFn=<function breadthFirstDP at 0x000000000B29F6D8>*)

libdw.searchTest.**mapTestAll**(*map*, *start*, *goal*)

libdw.searchTest.**numberCostCompare**(*s*, *g*, *h*)

libdw.searchTest.**numberCostTest**(*s*, *g*, *h*)

libdw.searchTest.**searchTestSM**(*goal*)

libdw.searchTest.**sign**(*x*)

libdw.searchTest.**swap**(*board*, *(ox*, *oy)*, *(nx*, *ny)*)

libdw.sf module

Class and some supporting functions for representing and manipulating system functions.

libdw.sf.**Cascade**(*sf1*, *sf2*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the cascade of `sf1` and `sf2`

class libdw.sf.**DifferenceEquation**(*dCoeffs*, *cCoeffs*)

Represent a difference equation in a form that makes it easy to simulate.

cCoeffs = *None*

Output coefficients

dCoeffs = *None*

Input coefficients

stateMachine(*previousInputs=None*, *previousOutputs=None*)

Parameters:
- **previousInputs** – list of historical inputs running from M{x[-1]} (at the beginning of the list) to M{x[-j]} at the end of the list, where M{j} is `len(self.dCoeffs)-1`. Defaults to the appropriate number of zeros.
- **previousOutputs** – list of historical outputs running from M{y[-1]} (at the beginning of the list) to M{y[-k]} (at the end of the list), where M{k} is `len(self.cCoeffs)`. Defaults to the appropriate number of zeros.

Returns: A state machine that uses this difference equation to transduce the sequence of inputs X to the sequences of outputs Y, starting from a state determined by `previousInputs` and `previousOutputs`

systemFunction()

Returns: A `SystemFunction` equivalent to this difference equation

`libdw.sf.`**FeedbackAdd**(*sf1*, *sf2=None*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the result of feeding the output of `sf1` back, with (optionally) `sf2` on the feedback path, adding it to the input, and feeding the resulting signal into `sf1`.

`libdw.sf.`**FeedbackSubtract**(*sf1*, *sf2=None*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the result of feeding the output of `sf1` back, with (optionally) `sf2` on the feedback path, subtracting it from the input, and feeding the resulting signal into `sf1`. This situation can be characterized with Black's formula.

`libdw.sf.`**FeedforwardAdd**(*sf1*, *sf2*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the sum of `sf1` and `sf2`; this models the situation when the two component systems have the same input and the output of the whole system is the sum of the outputs of the components.

`libdw.sf.`**FeedforwardSubtract**(*sf1*, *sf2*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the difference of `sf1` and `sf2`; this models the situation when the two component systems have the same input and the output of the whole system is the output of the first component minus the output of the second component.

`libdw.sf.`**Gain**(*k*)

Parameters: **k** – gain parameter

Returns: `SystemFunction` representing a system that multiplies the input signal by `k`.

`libdw.sf.`**R**()

Returns: `SystemFunction` representing a system that delays the input signal by one step.

`libdw.sf.`**Sum**(*sf1*, *sf2*)

Parameters:
- **sf1** – `SystemFunction`
- **sf2** – `SystemFunction`

Returns: `SystemFunction` representing the system that sums the outputs of the two systems

class `libdw.sf.`**SystemFunction**(*numeratorPoly*, *denominatorPoly*)

Represent a system function as a ratio of polynomials in R

denominator = *None*

Polynomial in R representing the denominator

differenceEquation()

Returns: a `DifferenceEquation` representation of this same system

dominantPole()

Returns: the pole with the largest magnitude

numerator = *None*

Polynomial in R representing the numerator

poleMagnitudes()

Returns: a list of the magnitudes of the poles of the system

poles()

Returns: a list of the poles of the system

`libdw.sf.`**complexPolar**(*p*)

Parameters: **p** – int, float, or complex number
Returns: polar representation as a pair of r, theta

`libdw.sf.`**periodOfPole**(*p*)

Parameters: **p** – int, float, or complex number
Returns: period = 2 pi / phase of pole or None (if phase is 0)

libdw.sig module

Signals, represented implicitly, with plotting and combinations.

class `libdw.sig.`**ConstantSignal**(*c*)

Bases: **libdw.sig.Signal**

Primitive constant sample signal.

sample(*n*)

class `libdw.sig.`**CosineSignal**(*omega=1*, *phase=0*)

Bases: **libdw.sig.Signal**

Primitive family of sinusoidal signals.

sample(*n*)

class `libdw.sig.`**FilteredSignal**(*s*, *f*, *w*)

Bases: **libdw.sig.Signal**

Signal filtered by a function, applied to a fixed-sized window of previous values

sample(*n*)

class `libdw.sig.`**ListSignal**(*samples*)

Bases: **libdw.sig.Signal**

Signal defined with a specific list of sample values, from 0 to some fixed length; It has value 0 elsewhere.

length = *None*

The length of the explicitly-represented part of this signal

sample(*n*)

samples = *None*

The non-zero sample values of this signal (starting at index 0)

class `libdw.sig.`**ListSignalSampled**(*samples*, *subsample*)

Bases: **`libdw.sig.Signal`**

Signal defined with a specific list of sample values, from 0 to some fixed length; It has the last value past the end and the first value before the start.

`length` = *None*

The length of the explicitly-represented part of this signal

`sample`(*n*)

`samples` = *None*

The non-zero sample values of this signal (starting at index 0)

class `libdw.sig.`**`R`**(*s*)

Bases: **`libdw.sig.Signal`**

Signal delayed by one time step, so that `R(S).sample(n+1) = S.sample(n)`

`sample`(*n*)

class `libdw.sig.`**`Rn`**(*s*, *n*)

Bases: **`libdw.sig.Signal`**

Signal delayed by several time steps

`sample`(*n*)

class `libdw.sig.`**`ScaledSignal`**(*s*, *c*)

Bases: **`libdw.sig.Signal`**

Signal multiplied everywhere by a constant

`sample`(*n*)

class `libdw.sig.`**`Signal`**

Represent infinite signals. This is a generic superclass that provides some basic operations. Every subclass must provide a `sample` method.

Be sure to start idle with the `-n` flag, if you want to make plots of signals from inside idle.

`crossings`(*n=None*, *z=None*)

Parameters:
- **n** – number of samples to use; if not provided, it will look for a `length` attribute of `self`
- **z** – zero value to use when looking for zero-crossings of the signal; will use the mean by default.

Returns: a list of indices into the data where the signal crosses the z value, up through time n

`mean`(*n=None*)

Parameters: **n** – number of samples to use to estimate the mean; if not provided, it will look for a `length` attribute of `self`

Returns: sample mean of the values of the signal from 0 to n

`period`(*n=None*, *z=None*)

Parameters:
- **n** – number of samples to use to estimate the period; if not provided, it will look for a `length` attribute of `self`
- **z** – zero value to use when looking for zero-crossings of the signal; will use the mean by default.

Returns: an estimate of the period of the signal, or 'aperiodic' if it can't get a good estimate

plot(*start=0*, *end=100*, *newWindow='Signal value versus time'*, *color='blue'*, *parent=None*, *ps=None*, *xminlabel=0*, *xmaxlabel=0*, *yOrigin=None*)

Make a plot of this signal.

Parameters:
- **start** – first value to plot; defaults to 0
- **end** – last value to plot; defaults to 100; must be > start
- **newWindow** – makes a new window with this value as title, unless the value is False, in which case it plots the signal in the currently active plotting window
- **color** – string specifying color of plot; all simple color names work
- **parent** – An instance of `tk.tk`. You probably should just leave this at the default unless you're making plots from another application.
- **ps** – If not `None`, then it should be a pathname; we'll write a postscript version of this graph to that path.

samplesInRange(*lo*, *hi*)

Returns: list of samples of this signal, from `lo` to `hi-1`

class `libdw.sig.`**StepSignal**

Bases: **`libdw.sig.Signal`**

Signal that has value 1 for all n >= 0, and value 0 otherwise.

sample(*n*)

class `libdw.sig.`**SummedSignal**(*s1*, *s2*)

Bases: **`libdw.sig.Signal`**

Sum of two signals

sample(*n*)

class `libdw.sig.`**UnitSampleSignal**

Bases: **`libdw.sig.Signal`**

Primitive unit sample signal has value 1 at time 0 and value 0 elsewhere.

sample(*n*)

`libdw.sig.`**gaps**(*data*)

Return a list of the gap sizes, given a list of numbers. (If input is length n, result is length n-1)

`libdw.sig.`**listMean**(*vals*)

Parameters: **vals** – list of numbers

Returns: mean of `vals`

`libdw.sig.`**makeSignalFromPickle**(*pathName*)

Parameters: **pathName** – string specifying directory and file name

Returns: `ListSignal` with data read in from `pathname`. That path must contain a pickled list of numbers.

`libdw.sig.`**meanFiltered**(*s*, *k*)

Parameters:
- **s** – `Signal`
- **k** – positive integer filter size

Returns: `s` filtered with a mean filter of size `k`

`libdw.sig.`**polyR**(*s*, *p*)

Parameters:
- **s** – `Signal`
- **p** – `poly.Polynomial`

Returns: New signal that is `s` transformed by `p` interpreted as a polynomial in *R*.

libdw.sig.us = <*libdw.sig.UnitSampleSignal instance at 0x00000000056E68C8*>

Unit sample signal instance

libdw.simulate module

class `libdw.simulate.`**`Circ`**

class `libdw.simulate.`**`CircuitSM`**(*components*, *inComponents*, *groundNode*)

Bases: **`libdw.sm.SM`**

`getNextValues`(*state*, *inp*)

class `libdw.simulate.`**`Connector`**(*ctype*, *nodeNames*)

class `libdw.simulate.`**`Feedback`**(*m1*, *m2*, *name=None*)

Bases: **`libdw.sm.SM`**

`done`(*state*)

`getNextValues`(*state*, *inp*)

`startState`()

class `libdw.simulate.`**`Feedback2`**(*m1*, *m2*, *name=None*)

Bases: **`libdw.simulate.Feedback`**

Like previous `Feedback`, but takes a machine with two inps and one output at initialization time. Feeds the output back to the second inp. Result is a machine with a single inp and single output.

`getNextValues`(*state*, *inp*)

class `libdw.simulate.`**`Ground`**(*v1*)

class `libdw.simulate.`**`GroundInput`**(*ctype*, *v1*)

class `libdw.simulate.`**`MotorAccel`**(*init*, *motorNodes*)

Bases: **`libdw.sm.SM`**

`getNextValues`(*state*, *inp*)

class `libdw.simulate.`**`MotorFeedback`**

Bases: **`libdw.sm.SM`**

`getNextValues`(*state*, *inp*)

class `libdw.simulate.`**`Pot`**(*v0*, *v1*, *v2*)

class `libdw.simulate.`**`Power`**(*v1*)

class `libdw.simulate.`**`PowerInput`**(*ctype*, *v1*)

class `libdw.simulate.`**`Probe`**(*ctype*, *v1*)

`libdw.simulate.`**`addNew`**(*entries*, *l*)

`libdw.simulate.`**`allDefined`**(*struct*)

`libdw.simulate.`**`checkConnected`**(*circuit*, *nodes*)

`libdw.simulate.`**`circuitSolve`**(*components*, *groundNode*)

`libdw.simulate.`**`componentDict`**(*componentList*) libdw.simulate.**connectedTo**(*connections*, *targets*)

libdw.simulate.**diagnoseCircuit**(*circuit*, *sigIn*)

libdw.simulate.**getAnalogViNodes**(*circuit*, *warning=False*)

libdw.simulate.**getAnalogVoNodes**(*circuit*, *warning=False*)

libdw.simulate.**getHeadPhotoNodes**(*circuit*, *warning=False*)

libdw.simulate.**getHeadPotNodes**(*circuit*, *warning=False*)

libdw.simulate.**getMotorNodes**(*circuit*, *warning=False*)

libdw.simulate.**getPotNodes**(*circuit*, *warning=False*)

libdw.simulate.**groundFromPins**(*pin0*)

libdw.simulate.**hasDuplicate**(*l*)

libdw.simulate.**headFromPins**(*pin0*, *pin1*)

libdw.simulate.**intensityFromAngles**(*headAngle*, *lightAngle*, *lightDist*)

libdw.simulate.**isDefined**(*v*)

libdw.simulate.**makeMotor**(*emf*, *nodes*)

libdw.simulate.**makePhoto**(*headAngle*, *lightAngle*, *lightDist*, *nodes*)

libdw.simulate.**makePhotoFromIntensity**(*intensityL*, *intensityR*, *nodes*)

libdw.simulate.**makePot**(*alpha*, *nodes*, *value=5000*)

libdw.simulate.**makeVoltage**(*value*, *nodes*)

libdw.simulate.**motorAngleAlpha**(*angle*)

libdw.simulate.**motorFromPins**(*pin0*, *pin1*)

libdw.simulate.**newDynamics**(*vm0*, *vm1*, *state*)

libdw.simulate.**nodeNameForPin**(*x*, *y*)

libdw.simulate.**nodeVals**(*sol*, *nodes*)

libdw.simulate.**oldDynamics**(*vm0*, *vm1*, *state*)

libdw.simulate.**opAmpsFromPins**(*pin0*, *pin1*)

libdw.simulate.**plotAnalogInputs**(*inputs*, *sols*, *parent*)

libdw.simulate.**plotAnalogOutputs**(*inValues*, *parent*)

libdw.simulate.**plotInputs**(*inValues*, *parent*)

libdw.simulate.**plotMotorPotAlpha**(*outValues*, *parent*)

libdw.simulate.**plotMotorVelocity**(*outValues*, *parent*)

libdw.simulate.**plotProbes**(*probes*, *sols*, *parent*)

libdw.simulate.**plotparams**(*Tsim*, *Tplot*, *nlen*)

libdw.simulate.**potFromPins**(*pin0*, *pin1*, *pin2*)

libdw.simulate.**powerFromPins**(*pin0*)

libdw.simulate.**probeFromPins**(*sign*, *pin0*) libdw.simulate.**readCircuit**(*lines*)

libdw.simulate.**readLines**(*filename*)

libdw.simulate.**reduceToDict**(*inp*)

libdw.simulate.**resistorFromPins**(*colors*, *pin0*, *pin1*)

libdw.simulate.**resistorValue**(*c*)

libdw.simulate.**robotFromPins**(*pin0*, *pin1*)

libdw.simulate.**runCircuit**(*lines*, *sigIn*, *parent=None*, *nsteps=50*)

libdw.simulate.**runRealCircuit**(*icircuit*, *sigIn*, *parent=None*, *nsteps=50*)

libdw.simulate.**safe**(*f*)

libdw.simulate.**safeAdd**(*a1*, *a2*)

libdw.simulate.**safeMul**(*a1*, *a2*)

libdw.simulate.**safeSub**(*a1*, *a2*)

libdw.simulate.**signum**(*x*)

libdw.simulate.**systemSM**(*circuit*, *inComponents*, *motorNodes*, *initState*)

libdw.simulate.**traceElement**(*etype*, *components*, *dict*)

libdw.simulate.**traceWires**(*node*, *dict*, *done*)

libdw.simulate.**verifyCircuit**(*componentList*)

libdw.simulate.**warn**(*message*)

libdw.simulate.**wireFromPins**(*pin0*, *pin1*)

libdw.sm module

Classes for representing and combining state machines.

class libdw.sm.**Cascade**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Cascade composition of two state machines. The output of `sm1` is the input to `sm2`

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class libdw.sm.**Constant**(*c*)

Bases: **libdw.sm.SM**

Machine whose output is a constant, independent of the input

getNextState(*state*, *inp*)

class libdw.sm.**DebugParams**(*traceTasks*, *verbose*, *compact*, *printInput*)

Housekeeping stuff

libdw.sm.Delay

Delay is another name for the class R, for backward compatibility

alias of **R**

class **libdw.sm.Feedback**(*m*, *name=None*)

Bases: **libdw.sm.SM**

Take the output of `m` and feed it back to its input. Resulting machine has no input. The output of `m` **must not** depend on its input without a delay.

done(*state*)

getNextValues(*state*, *inp*)

Ignores input.

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class **libdw.sm.Feedback2**(*m*, *name=None*)

Bases: **libdw.sm.Feedback**

Like previous `Feedback`, but takes a machine with two inps and one output at initialization time. Feeds the output back to the second inp. Result is a machine with a single inp and single output.

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

class **libdw.sm.FeedbackAdd**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Takes two machines, m1 and m2. Output of the composite machine is the output to m1. Output of m1 is fed back through m2; that result is added to the input and used as the 'error' signal, which is the input to m1.

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class **libdw.sm.FeedbackSubtract**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Takes two machines, m1 and m2. Output of the composite machine is the output to m1. Output of m1 is fed back through m2; that result is subtracted from the input and used as the 'error' signal, which is the input to m1. Transformation is the one described by Black's formula.

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class **libdw.sm.Gain**(*k*)

Bases: **libdw.sm.SM**

Machine whose output is the input, but multiplied by k. Specify k in initializer.

getNextValues(*state*, *inp*)

class libdw.sm.**If**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **libdw.sm.SM**

Given a condition (function from inps to boolean) and two state machines, make a new machine. The condition is evaluated at start time, and one machine is selected, permanently, for execution.

Rarely useful.

done(*state*)

getFirstRealState(*inp*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState *= ('start', None)*

class libdw.sm.**Mux**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **libdw.sm.Switch**

Like `Switch`, but updates both machines no matter whether the condition is true or false. Condition is only used to decide which output to generate. If the condition is true, it generates the output from the first machine, otherwise, from the second.

getNextValues(*state*, *inp*)

class libdw.sm.**Parallel**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.SM**

Takes a single inp and feeds it to two machines in parallel. Output of the composite machine is the pair of outputs of the two individual machines.

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class libdw.sm.**Parallel2**(*m1*, *m2*)

Bases: **libdw.sm.Parallel**

Like `Parallel`, but takes two inps. Output of the composite machine is the pair of outputs of the two individual machines.

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

class libdw.sm.**ParallelAdd**(*m1*, *m2*, *name=None*)

Bases: **libdw.sm.Parallel**

Like `Parallel`, but output is the sum of the outputs of the two machines.

getNextValues(*state*, *inp*)

class libdw.sm.**PureFunction**(*f*)

Bases: **libdw.sm.SM**

Machine whose output is produced by applying a specified Python function to its input.

getNextValues(*state*, *inp*)

class libdw.sm.**R**(*v0=0*)

Bases: **libdw.sm.SM**

Machine whose output is the input, but delayed by one time step. Specify initial output in initializer.

getNextValues(*state*, *inp*)

startState = *None*

State is the previous input

class libdw.sm.**Repeat**(*sm*, *n=None*, *name=None*)

Bases: **libdw.sm.SM**

Given a terminating state machine, generate a new one that will execute it n times. If n is unspecified, it will repeat forever.

advanceIfDone(*counter*, *smState*)

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class libdw.sm.**RepeatUntil**(*condition*, *sm*, *name=None*)

Bases: **libdw.sm.SM**

Given a terminating state machine and a condition on the input, generate a new one that will run the machine until the condition becomes true. However, the condition is **only** evaluated when the sub-machine terminates.

done(*state*)

getNextValues(*state*, *inp*)

printDebugInfo(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

startState()

class libdw.sm.**SM**

Generic superclass representing state machines. Don't instantiate this: make a subclass with definitions for the following methods:

- `getNextValues: (state_t, inp_t) -> (state_t+1, output_t)` or `getNextState: (state_t, inpt_t) -> state_t+1`
- `startState:  state` or `startState() -> state`

optional:

- `done: (state) -> boolean` (defaults to always false)
- `legalInputs: list(inp)`

See State Machines chapter in 6.01 Readings for detailed explanation.

check(*thesm*, *inps=None*)

Run a rudimentary check on a state machine, using the list of inputs provided. Makes sure that getNextValues is defined, and that it takes the proper number of input arguments (three: self, start, inp). Also print out the start state, and check that getNextValues provides a legal return value (list of 2 elements: (state,output)). And tries to check if getNextValues is changing either self.state or some other attribute of the state machine instance (it shouldn't: getNextValues should be a pure function).

Raises exception 'InvalidSM' if a problem is found.

Parameters:
- **thesm** – the state machine instance to check
- **inps** – list of inputs to test the state machine on (default None)

Returns: none

`doTraceTasks`(*inp*, *state*, *out*, *debugParams*)

Actually execute the trace tasks. A trace task is a list consisting of three components:

- `name`: is the name of the machine to be traced
- `mode`: is one of `'input'`, `'output'`, or `'state'`
- `fun`: is a function

To **do** a trace task, we call the function `fun` on the specified attribute of the specified mahine. In particular, we execute it right now if its machine name equals the name of this machine.

`done`(*state*)

By default, machines don't terminate

`getNextValues`(*state*, *inp*)

Default version of this method. If a subclass only defines `getNextState`, then we assume that the output of the machine is the same as its next state.

`getStartState`()

Handles the case that self.startState is a function. Necessary for stochastic state machines. Ignore otherwise.

`guaranteeName`()

Makes sure that this instance has a unique name that can be used for tracing.

`isDone`()

Should only be used by transduce. Don't call this.

`legalInputs` = *[]*

By default, the space of legal inputs is not defined.

`name` = *None*

Name used for tracing

`printDebugInfo`(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

Default method for printing out all of the debugging information for a primitive machine.

`run`(*n=10*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*, *check=False*)

For a machine that doesn't consume input (e.g., one made with `feedback`, for `n` steps or until it terminates.

See documentation for the `start` method for description of the rest of the parameters.

Parameters: **n** – number of steps to run

Returns: list of outputs

`start`(*traceTasks=[]*, *verbose=False*, *compact=True*, *printInput=True*)

Call before providing inp to a machine, or to reset it. Sets self.state and arranges things for tracing and debugging.

Parameters:
- **traceTasks** – list of trace tasks. See documentation for `doTraceTasks` for details
- **verbose** – If `True`, print a description of each step of the machine
- **compact** – If `True`, then if `verbose = True`, print a one-line description of the step; if `False`, print out the recursive substructure of the state update at each step
- **printInput** – If `True`, then if `verbose = True`, print the whole input in each step, otherwise don't. Useful to set to `False` when the input is large and you don't want to see it all.

`startState` = *None*

By default, startState is none

`step`(*inp*)

Execute one 'step' of the machine, by propagating `inp` through to get a result, then updating `self.state`. Error to call `step` if `done` is true. :param inp: next input to the machine

`transduce`(*inps*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*, *check=False*)

Start the machine fresh, and feed a sequence of values into the machine, collecting the sequence of outputs

For debugging, set the optional parameter check = True to (partially) check the representation invariance of the state machine before running it. See the documentation for the `check` method for more information about what is tested.

See documentation for the `start` method for description of the rest of the parameters.

Parameters: **inps** – list of inputs appropriate for this state machine
Returns: list of outputs

`transduceF`(*inpFn*, *n=10*, *verbose=False*, *traceTasks=[]*, *compact=True*, *printInput=True*)

Like `transduce`, but rather than getting inputs from a list of values, get them by calling a function with the input index as the argument.

class `libdw.sm.`**`Select`**(*k*)

Bases: **`libdw.sm.SM`**

Machine whose input is a structure list and whose output is the `k` th element of that list.

`getNextState`(*state*, *inp*)

class `libdw.sm.`**`Sequence`**(*smList*, *name=None*)

Bases: **`libdw.sm.SM`**

Given a list of state machines, make a new machine that will execute the first until it is done, then execute the second, etc. Assume they all have the same input space.

`advanceIfDone`(*counter*, *smState*)

Internal use only. If that machine is done, start new machines until we get to one that isn't done

`done`(*state*)

`getNextValues`(*state*, *inp*)

`printDebugInfo`(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

`startState`()

class `libdw.sm.`**`Switch`**(*condition*, *sm1*, *sm2*, *name=None*)

Bases: **`libdw.sm.SM`**

Given a condition (function from inps to boolean) and two state machines, make a new machine. The condition is evaluated on every step, and the selected machine is used to generate output and has its state updated. If the condition is true, `sm1` is used, and if it is false, `sm2` is used.

`done`(*state*)

`getNextValues`(*state*, *inp*)

`printDebugInfo`(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

`startState`()

class `libdw.sm.`**`Until`**(*condition*, *sm*, *name=None*)

Bases: **`libdw.sm.SM`**

Execute SM until it terminates or the condition becomes true. Condition is evaluated on the inp

`done`(*state*)

`getNextValues`(*state*, *inp*)

`printDebugInfo`(*depth*, *state*, *nextState*, *inp*, *out*, *debugParams*)

`startState`()

class `libdw.sm.`**`Wire`**

Bases: **`libdw.sm.SM`**

Machine whose output is the input

`getNextValues`(*state*, *inp*)

`libdw.sm.`**`allDefined`**(*struct*)

`libdw.sm.`**`coupledMachine`**(*m1*, *m2*)

Couple two machines together. :param m1: `SM` :param m2: `SM` :returns: New machine with no input, in which the output of `m1` is the input to `m2` and vice versa.

`libdw.sm.`**`isDefined`**(*v*)

`libdw.sm.`**`safe`**(*f*)

`libdw.sm.`**`safeAdd`**(*a1*, *a2*)

`libdw.sm.`**`safeMul`**(*a1*, *a2*)

`libdw.sm.`**`safeSub`**(*a1*, *a2*)

`libdw.sm.`**`splitValue`**(*v*, *n=2*)

If `v` is a list of `n` elements, return it; if it is 'undefined', return a list of `n` 'undefined' values; else generate an error

libdw.soarWorld module

Read in a soar simulated world file and represent its walls as lists of line segments.

class `libdw.soarWorld.`**`SoarWorld`**(*path*)

Represents a world in the same way as the soar simulator

`addWall`(*(xlo, ylo)*, *(xhi, yhi)*)

`dims`(*dx*, *dy*)

`initialLoc`(*x*, *y*)

`wallSegs` = *None*

Walls represented as list of `util.lineSeg`

walls = *None*

Walls represented as list of pairs of endpoints

`libdw.soarWorld.`**dimensions**(*x*, *y*)

`libdw.soarWorld.`**initialRobotLoc**(*x*, *y*)

`libdw.soarWorld.`**wall**(*p1*, *p2*)

libdw.sonarDist module

Useful constants and utilities for dealing with sonar readings in soar.

`libdw.sonarDist.`**distAndAngle**(*h0*, *h1*)

`libdw.sonarDist.`**getDistanceRight**(*sonarValues*)

Parameters: **sonarValues** – list of 6 sonar readings

Returns: the perpendicular distance to a surface on the right of the robot, assuming there is a linear surface.

`libdw.sonarDist.`**getDistanceRightAndAngle**(*sonarValues*)

Parameters: **sonarValues** – list of 6 sonar readings

Returns: (d, a) where, d is the perpendicular distance to a surface on the right of the robot, assuming there is a linear surface; and a is the angle to that surface.

Change to use `sonarHit`, or at least point and pose transforms.

`libdw.sonarDist.`**line**(*h0*, *h1*)

`libdw.sonarDist.`**sonarHit**(*distance*, *sonarPose*, *robotPose*)

Parameters:
- **distance** – distance along ray that the sonar hit something
- **sonarPose** – `util.Pose` of the sonar on the robot
- **robotPose** – `util.Pose` of the robot in the global frame

Returns: `util.Point` representing position of the sonar hit in the global frame.

`libdw.sonarDist.`**sonarMax** = *1.5*

Maximum good sonar reading.

`libdw.sonarDist.`**sonarPoses** = *[pose:(0.073000, 0.105000, 1.570796), pose:(0.130000, 0.078000, 0.715585), pose:(0.154000, 0.030000, 0.261799), pose:(0.154000, -0.030000, -0.261799), pose:(0.130000, -0.078000, -0.715585), pose:(0.073000, -0.105000, -1.570796)]*

Positions and orientations of sonar sensors with respect to the center of the robot.

libdw.ssm module

Class for representing stochastic state machines.

class `libdw.ssm.`**StochasticSM**(*startDistribution*, *transitionDistribution*, *observationDistribution*, *beliefDisplayFun=None*, *sensorDisplayFun=None*)

Bases: **`libdw.sm.SM`**

Stochastic state machine.

beliefDisplayFun = *None*

(Optional) function that is not used here, but that state estimator, for example, might call to display a belief state. Takes a belief state {dist.DDist} as input.

getNextValues(*state*, *inp*)

observationDistribution = *None*

A procedure that takes a state and returns a distribution over observations.

sensorDisplayFun = *None*

(Optional) function that is not used here, but that state estimator, for example, might call to display a sensor likelihoods. Takes an observation as input.

startDistribution = *None*

`dist.DDist` over states.

startState()

transitionDistribution = *None*

A procedure that takes an action and returns a procedure, which takes an old state and returns a distribution over new states.

class libdw.ssm.**StochasticSMWithStateObservation**(*startDistribution*, *transitionDistribution*, *observationDistribution*, *beliefDisplayFun=None*, *sensorDisplayFun=None*)

Bases: **libdw.ssm.StochasticSM**

Special kind of stochastic state machine whose observation includes its state

getNextValues(*state*, *inp*)

libdw.threeSteps module

libdw.threeSteps.**runTest**(*lines*, *parent=None*, *nsteps=125*)

libdw.threeSteps.**testSignal**(*simTime=2.5*)

libdw.tk module

libdw.tk.**init**()

libdw.tk.**setInited**()

libdw.ts module

A class of signals that is created by putting another signal through a transducer (state machine).

class libdw.ts.**TransducedSignal**(*s*, *m*)

Bases: **libdw.sig.Signal**

Given a signal s, and a state machine m, generate a new signal that has value 0 for any k < 0, and otherwise has the output of m, with s as its input, as its value

sample(*k*)

class libdw.ts.**TransducedSignalSlow**(*s*, *m*)

Bases: **libdw.sig.Signal**

Given a state a signal s and a state machine m, generate a new signal that has value 0 for any k < 0, and otherwise has the output of m, with s as its input, as its value

sample(*k*)

Generate sample k of this signal. Wildly inefficient.

libdw.ucSearch module

Procedures and classes for doing uniform cost search, always with dynamic programming. Becomes A* if a heuristic is specified.

class `libdw.ucSearch.PQ`

Slow implementation of a priority queue that just finds the minimum element for each extraction.

isEmpty()

Returns `True` if the PQ is empty and `False` otherwise.

pop()

Returns and removes the least cost item. Assumes items are instances with an attribute `cost`.

push(*item*, *cost*)

Push an item onto the priority queue. Assumes items are instances with an attribute `cost`.

class `libdw.ucSearch.SearchNode`(*action*, *state*, *parent*, *actionCost*)

A node in a search tree

action = *None*

Action that moves from `parent` to `state`

cost = *None*

The cost of the path from the root to `self.state`

inPath(*s*)

Returns: `True` if state `s` is in the path from here to the root

path()

Returns: list of `(action, state)` pairs from root to this node

`libdw.ucSearch.search`(*initialState*, *goalTest*, *actions*, *successor*, *heuristic=<function <lambda> at 0x000000000BED1128>*, *maxNodes=10000*)

Parameters:
- **initialState** – root of the search
- **goalTest** – function from state to Boolean
- **actions** – a list of possible actions
- **successor** – function from state and action to next state and cost
- **heuristic** – function from state to estimated cost to reach a goal; defaults to a heuristic of 0, making this uniform cost search
- **maxNodes** – kill the search after it expands this many nodes

Returns: path from initial state to a goal state as a list of (action, state) tuples

`libdw.ucSearch.smSearch`(*smToSearch*, *initialState=None*, *goalTest=None*, *heuristic=<function <lambda> at 0x000000000535C668>*, *maxNodes=10000*)

Parameters:
- **smToSearch** – instance of `sm.SM` defining a search domain; `getNextValues` is used to determine the successor of a state given an action; the output field of getNextValues is interpreted as a cost.
- **initialState** – initial state for the search; if not provided, will use `smToSearch.startState`

- **goalTest** – function that takes a state as an argument and returns `True` if it is a goal state, and `False` otherwise
- **heuristic** – function from state to estimated cost to reach a goal; defaults to a heuristic of 0, making this uniform cost search
- **maxNodes** – maximum number of nodes to be searched; prevents runaway searches

Returns: a list of the form `[(a0, s0), (a1, s1), (a2, s2), ...]` where the a's are legal actions of c{smToSearch} and s's are states of that machine. `s0` is the start state; the last state is a state that satisfies the goal test. If the goal is unreachable (within the search limit), it returns `None`.

`libdw.ucSearch.`**`somewhatVerbose`** = *False*

If `True`, prints a trace of the search

`libdw.ucSearch.`**`verbose`** = *False*

If `True`, prints a verbose trace of the search

libdw.util module

A wide variety of utility procedures and classes.

class `libdw.util.`**`Line`**(*p1*, *p2*)

Line in 2D space

`nx` = *None*

x component of normal vector

`ny` = *None*

y component of normal vector

`off` = *None*

offset along normal

`pointOnLine`(*p*, *eps*)

Return true if p is within eps of the line

`theta` = *None*

normal angle

class `libdw.util.`**`LineSeg`**(*p1*, *p2*)

Line segment in 2D space

`M` = *None*

Vector from the stored point to the other point

`closestPoint`(*p*)

Return the point on the line that is closest to point p

`distToPoint`(*p*)

Shortest distance between point p and this line

`intersection`(*other*)

Return a `Point` where `self` intersects `other`. Returns `False` if there is no intersection. :param other: a `LineSeg`

`p1` = *None*

One point

p2 = *None*

Other point

class **libdw.util.Point**(*x*, *y*)

Represent a point with its x, y values

add(*point*)

Vector addition

angleTo(*p*)

Parameters: **p** – instance of `util.Point` or `util.Pose`

Returns: angle in radians of vector from self to p

distance(*point*)

Parameters: **point** – instance of `util.Point`

Returns: Euclidean distance between `self` and `util.Point`

dot(*p*)

Dot product

isNear(*point*, *distEps*)

Parameters:
- **point** – instance of `util.Point`
- **distEps** – positive real number

Returns: true if the distance between `self` and `util.Point` is less than distEps

magnitude()

Returns: Magnitude of this point, interpreted as a vector in 2-space

near(*point*, *distEps*)

Parameters:
- **point** – instance of `util.Point`
- **distEps** – positive real number

Returns: true if the distance between `self` and `util.Point` is less than distEps

scale(*s*)

Vector scaling

sub(*point*)

Vector subtraction

x = *0.0*

x coordinate

xyTuple()

Returns: pair of x, y values

y = *0.0*

y coordinate

class **libdw.util.Pose**(*x*, *y*, *theta*)

Represent the x, y, theta pose of an object in 2D space

diff(*pose*)

Parameters: **pose** – an instance of `util.Pose`

Returns: a pose that is the difference between self and pose (in x, y, and theta)

distance(*pose*)

Parameters: **pose** – an instance of `util.Pose`

Returns: the distance between the x,y part of self and the x,y part of pose.

`inverse`()

Return a pose corresponding to the transformation matrix that is the inverse of the transform associated with this pose. If this pose's transformation maps points from frame X to frame Y, the inverse maps points from frame Y to frame X.

`isNear`(*pose*, *distEps*, *angleEps*)

Returns: True if pose is within distEps and angleEps of self

`point`()

Return just the x, y parts represented as a `util.Point`

`theta` = *0.0*

rotation in radians

`transform`()

Return a transformation matrix that corresponds to rotating by theta and then translating by x,y (in the original coordinate frame).

`transformDelta`(*point*)

Does the rotation by theta of the pose but does not add the x,y offset. This is useful in transforming the difference(delta) between two points. :param point: an instance of `util.Point` :returns: a `util.Point`.

`transformPoint`(*point*)

Applies the pose.transform to point and returns new point. :param point: an instance of `util.Point`

`transformPose`(*pose*)

Make self into a transformation matrix and apply it to pose. :returns: Af new `util.pose`.

`x` = *0.0*

x coordinate

`xytTuple`()

Returns: a representation of this pose as a tuple of x, y, theta values

`y` = *0.0*

y coordinate

class `libdw.util.`**`SymbolGenerator`**

Generate new symbols guaranteed to be different from one another Optionally, supply a prefix for mnemonic purposes Call gensym("foo") to get a symbol like 'foo37'

`gensym`(*prefix='i'*)

class `libdw.util.`**`Transform`**(*matrix=None*)

Rotation and translation represented as 3 x 3 matrix

`applyToPoint`(*point*)

Transform a point into a new point.

`applyToPose`(*pose*)

Transform a pose into a new pose.

`compose`(*trans*)

Returns composition of self and trans

inverse()

Returns transformation matrix that is the inverse of this one

matrix = *None*

matrix representation of transform

pose()

Convert to Pose

`libdw.util.`**argmax**(*l*, *f*)

Parameters:
- **l** – `List` of items
- **f** – `Procedure` that maps an item into a numeric score

Returns: the element of `l` that has the highest score

`libdw.util.`**argmaxIndex**(*l*, *f=<function <lambda> at 0x000000000C711908>*)

Parameters:
- **l** – `List` of items
- **f** – `Procedure` that maps an item into a numeric score

Returns: the index of `l` that has the highest score

`libdw.util.`**argmaxIndices3D**(*l*, *f=<function <lambda> at 0x000000000C711978>*)

`libdw.util.`**argmaxWithVal**(*l*, *f*)

Parameters:
- **l** – `List` of items
- **f** – `Procedure` that maps an item into a numeric score

Returns: the element of `l` that has the highest score and the score

`libdw.util.`**clip**(*v*, *vMin*, *vMax*)

Parameters:
- **v** – number
- **vMin** – number (may be None, if no limit)
- **vMax** – number greater than `vMin` (may be None, if no limit)

Returns: If `vMin <= v <= vMax`, then return `v`; if `v < vMin` return `vMin`; else return `vMax`

`libdw.util.`**dotProd**(*a*, *b*)

Return the dot product of two lists of numbers

`libdw.util.`**findFile**(*filename*)

Takes a filename and returns a complete path to the first instance of the file found within the subdirectories of the brain directory.

`libdw.util.`**fixAngle02Pi**(*a*)

Parameters: **a** – angle in radians

Returns: return an equivalent angle between 0 and 2 pi

`libdw.util.`**fixAnglePlusMinusPi**(*a*)

A is an angle in radians; return an equivalent angle between plus and minus pi

`libdw.util.`**gaussian**(*x*, *mu*, *sigma*)

Value of the gaussian distribution with mean mu and stdev sigma at value x

`libdw.util.`**gensym** = *<bound method SymbolGenerator.gensym of <libdw.util.SymbolGenerator instance at 0x000000000B145F08>>*

Call this function to get a new symbol

`libdw.util.`**`globalDeltaToLocal`**(*pose*, *deltaPoint*)

Applies inverse of pose to delta using transformDelta. :param pose: instance of `util.Pose` :param deltaPoint: instance of `util.Point`

`libdw.util.`**`globalPoseToLocalPose`**(*pose1*, *pose2*)

Applies inverse of pose1 to pose2. :param pose1: instance of `util.Pose` :param pose2: instance of `util.Pose`

`libdw.util.`**`globalToLocal`**(*pose*, *point*)

Applies inverse of pose to point. :param pose: instance of `util.Pose` :param point: instance of `util.Point`

`libdw.util.`**`inversePose`**(*pose*)

Same as pose.inverse() :param pose: instance of `util.Pose`

`libdw.util.`**`lineIndices`**(*(i0, j0)*, *(i1, j1)*)

Takes two cells in the grid (each described by a pair of integer indices), and returns a list of the cells in the grid that are on the line segment between the cells.

`libdw.util.`**`lineIndicesConservative`**(*(i0, j0)*, *(i1, j1)*)

Takes two cells in the grid (each described by a pair of integer indices), and returns a list of the cells in the grid that are on the line segment between the cells. This is a conservative version.

`libdw.util.`**`localPoseToGlobalPose`**(*pose1*, *pose2*)

Applies the transform from pose1 to pose2 :param pose1: instance of `util.Pose` :param pose2: instance of `util.Pose`

`libdw.util.`**`localToGlobal`**(*pose*, *point*)

Same as pose.transformPoint(point) :param point: instance of `util.Point`

`libdw.util.`**`logGaussian`**(*x*, *mu*, *sigma*)

Log of the value of the gaussian distribution with mean mu and stdev sigma at value x

`libdw.util.`**`make2DArray`**(*dim1*, *dim2*, *initValue*)

Return a list of lists representing a 2D array with dimensions dim1 and dim2, filled with initialValue

`libdw.util.`**`make2DArrayFill`**(*dim1*, *dim2*, *initFun*)

Return a list of lists representing a 2D array with dimensions `dim1` and `dim2`, filled by calling `initFun(ix, iy)` with `ix` ranging from 0 to `dim1 - 1` and `iy` ranging from 0 to `dim2-1`.

`libdw.util.`**`make3DArray`**(*dim1*, *dim2*, *dim3*, *initValue*)

Return a list of lists of lists representing a 3D array with dimensions dim1, dim2, and dim3 filled with initialValue

`libdw.util.`**`makeVector`**(*dim*, *initValue*)

Return a list of dim copies of initValue

`libdw.util.`**`makeVectorFill`**(*dim*, *initFun*)

Return a list resulting from applying initFun to values from 0 to dim-1

`libdw.util.`**`mapArray3D`**(*array*, *f*)

Map a function over the whole array. Side effects the array. No return value.

`libdw.util.`**`mm`**(*t1*, *t2*)

Multiplies 3 x 3 matrices represented as lists of lists `libdw.util.nearAngle`(*a1*, *a2*, *eps*)

Parameters:
- **a1** – number representing angle; no restriction on range
- **a2** – number representing angle; no restriction on range
- **eps** – positive number

Returns: `True` if `a1` is within `eps` of `a2`. Don't use within for this, because angles wrap around!

`libdw.util.nearlyEqual`(*x*, *y*)

Like within, but with the tolerance built in

`libdw.util.prettyPrint`(*struct*)

`libdw.util.prettyString`(*struct*)

Make nicer looking strings for printing, mostly by truncating floats

`libdw.util.randomMultinomial`(*dist*)

Parameters: **dist** – List of positive numbers summing to 1 representing a multinomial distribution over integers from 0 to `len(dist)-1`.

Returns: random draw from that distribution

`libdw.util.reverseCopy`(*items*)

Return a list that is a reversed copy of items

`libdw.util.sign`(*x*)

Return 1, 0, or -1 depending on the sign of x

`libdw.util.sum`(*items*)

Defined to work on items other than numbers, which is not true for the built-in sum.

`libdw.util.valueListToPose`(*values*)

Parameters: **values** – a list or tuple of three values: x, y, theta

Returns: a corresponding `util.Pose`

`libdw.util.within`(*v1*, *v2*, *eps*)

Parameters:
- **v1** – number
- **v2** – number
- **eps** – positive number

Returns: `True` if `v1` is with `eps` of `v2`

libdw.windows module

Module contents