RLSimion

Project: RLSimion

This file has been automatically generated. Please do not edit it

API Reference

- Actor
- ActorCritic
- BhatnagarSchedule
- CACLALearner
- ConfigNode
- DDPG
- DeterministicPolicyGaussianNoise
- DiscreteEpsilonGreedyDeepPolicy
- DiscreteFeatureMap
- DiscreteSoftmaxDeepPolicy
- DoubleQLearning
- DQN
- DynamicModel
- ETraces
- ExperienceReplay
- Experiment
- FeatureList
- FeatureMap
- FunctionSampler
- FunctionSampler2D
- FunctionSampler3D
- GaussianNoise
- GaussianRBFGridFeatureMap
- GreedyQPlusNoisePolicy
- IncrementalNaturalActorCritic
- InterpolatedValue
- LinearStateActionVFA
- LinearStateVFA
- LinearVFA
- Logger
- LQRController
- MemBlock
- OffPolicyActorCritic
- OffPolicyDeterministicActorCritic
- OrnsteinUhlenbeckNoise
- PIDController
- QEGreedyPolicy
- QLearningCritic
- QSoftMaxPolicy

Actor

Class Actor

Source: actor.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  Iterates over the actor's policy learners so that every one determines its output action

- **Parameters**
  - *s*: Input initial state
  - *a*: Output action
- **Return Value**

  Iterates over the actor's policy learners so that every one determines its output action

`void update(const State s, const Action a, const State* s_p, double r, double td)`

- **Summary**

  Iterates over all the actor's policy learners so that every one learns from an experience tuple {s,a,s_p,r,td}

- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *td*: Temporal-Difference error calculated by the critic

Actorcritic

Class ActorCritic

Source: actor-critic.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  Objects that implement both an actor and a critic call the actor's selectAction() method

- **Parameters**

  - *s*: Initial state
  - *a*: Output action

- **Return Value**

  Objects that implement both an actor and a critic call the actor's selectAction() method

`double update(const State s, const Action a, const State *s_p, double r, double behaviorProb)`

- **Summary**

  Encapsulates the basic Actor-Critic update: the critic calculates the TD error and the actor updates its policy accordingly

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *behaviorProb*: Probability by which the actor selected the action. Should be ignored

- **Return Value**

  Encapsulates the basic Actor-Critic update: the critic calculates the TD error and the actor updates its policy accordingly

Bhatnagarschedule

Class BhatnagarSchedule

Source: parameters-numeric.cpp

Methods

`double get()`

- **Summary**

  Implements the schedule function proposed by Bhatnagar

Caclalearner

Class CACLALearner

Source: actor-cacla.cpp

Methods

`void update(const State s, const Action a, const State *s_p, double r, double td)`

- **Summary**

  Updates the policy using the CACLA update rule

- **Parameters**

  - *s*: Initial state
  - *a*: Action

- *_sp*: Resultant state
- *r*: Reward
- *td*: Temporal-Difference error

Confignode

Class ConfigNode

Source: config.cpp

Methods

`int countChildren(const char* name)`

- **Summary**

  Returns the number of children this node has with the given name

`bool getConstBoolean(const char* paramName, bool defaultValue)`

- **Summary**

  Retrieves the value of a parameter as a boolean

- **Parameters**
  - *paramName*: The name of the parameter
  - *defaultValue*: Its default value (will be used if the parameter is not found)

- **Return Value**

  Retrieves the value of a parameter as a boolean

`int getConstInteger(const char* paramName, int defaultValue)`

- **Summary**

  Retrieves the value of a parameter as an integer

- **Parameters**
  - *paramName*: The name of the parameter
  - *defaultValue*: Its default value (will be used if the parameter is not found)

- **Return Value**

  Retrieves the value of a parameter as an integer

`double getConstDouble(const char* paramName, double defaultValue)`

- **Summary**

  Retrieves the value of a parameter as a double

- **Parameters**
  - *paramName*: The name of the parameter
  - *defaultValue*: Its default value (will be used if the parameter is not found)

- **Return Value**

  Retrieves the value of a parameter as a double

`void saveFile(const char* pFilename)`

- **Summary**

  Saves all the configuration nodes below the current to a file

- **Parameters**
  - *pFilename*: The path to the file

`void saveFile(FILE* pFile)`

- **Summary**

  Saves all the configuration nodes below the current to an already open file

- **Parameters**

- *pFile*: The handle to the already open file

`void clone(ConfigFile* parameterFile)`

- **Summary**

  Makes a shallow copy of the a configuration file

- **Parameters**
  - *parameterFile*:

Ddpg

Class DDPG

Source: DDPG.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  Implements action selection for the DDPG algorithm adding the output of the actor and exploration noise signal

- **Parameters**
  - *s*: State
  - *a*: Output action

`double update(const State s, const Action a, const State * s_p, double r, double behaviorProb)`

- **Summary**

  Updates the critic and actor using the DDPG algorithm

- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward

Deterministicpolicygaussiannoise

Class DeterministicPolicyGaussianNoise

Source: vfa-policy.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  A DeterministicPolicyGaussianNoise policy uses a single function representing the deterministic output. In training episodes, the noise signal used for exploration is sampled and added to the deterministic output

- **Parameters**
  - *s*: Initial state
  - *a*: Output state

`void getFeatures(const State state, FeatureList outFeatureList)`

- **Summary**

  Uses the policy's feature map to return the features representing the state

- **Parameters**
  - *state*: State
  - *outFeatureList*: Output feature list

Discreteepsilongreedydeeppolicy

Class DiscreteEpsilonGreedyDeepPolicy

Source: deep-vfa-policy.cpp

Methods

`int selectAction(const std::vector& values)`

- **Summary**

  Deep RL version of the epsilon-greedy action selection algorithm

- **Parameters**

  - *values*: Estimated Q(s,a) for each discrete action. Size should equal the number of discrete actions

- **Return Value**

  Deep RL version of the epsilon-greedy action selection algorithm

Discretefeaturemap

Class DiscreteFeatureMap

Source: featuremap-discrete.cpp

Methods

`void map(vector>& grids, const vector& values, FeatureList outFeatures)`

- **Summary**

  Implements a feature mapping function that maps state-actions to boxes. Only one feature will be active

- **Parameters**

  - *grids*: Input grids for every state-variable used
  - *values*: The values of every state-variable used
  - *outFeatures*: The output list of features

`void unmap(size_t feature, vector& grids, vector& outValues)`

- **Summary**

  Inverse of the feature mapping operation. Given a feature it returns the state-action to which it corresponds.

- **Parameters**

  - *feature*: The index of the feature
  - *grids*: The set of grids used to discretize each variable
  - *outValues*: The set of output values for every state-action variable

Discretesoftmaxdeeppolicy

Class DiscreteSoftmaxDeepPolicy

Source: deep-vfa-policy.cpp

Methods

`int selectAction(const std::vector& values)`

- **Summary**

  Deep-RL version of the Soft-Max action selection policy

- **Parameters**

  - *values*: Estimated Q(s,a) for each discrete action. Size should equal the number of discrete actions

- **Return Value**

  Deep-RL version of the Soft-Max action selection policy

Doubleqlearning

Class DoubleQLearning

Source: q-learners.cpp

Methods

```
double update(const State s, const Action a, const State *s_p, double r, double probability)
```

- **Summary**

  Updates the estimate of the Q-function using the Double Q-Learning update rule with tuple {s,a,s_p,r}. The main difference with respect to Q-function is that it uses two different sets of weights for the function, updating a random set of weights toward the other set of weights. Should offer better stability than regular Q-Learning

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward

- **Return Value**

  Updates the estimate of the Q-function using the Double Q-Learning update rule with tuple {s,a,s_p,r}. The main difference with respect to Q-function is that it uses two different sets of weights for the function, updating a random set of weights toward the other set of weights. Should offer better stability than regular Q-Learning

Dqn

Class DQN

Source: DQN.cpp

Methods

```
double selectAction(const State s, Action a)
```

- **Summary**

  Implements the action selection algorithm for a Q-based Deep RL algorithm

- **Parameters**

  - *s*: State
  - *a*: Output action

```
double update(const State s, const Action a, const State * s_p, double r, double behaviorProb)
```

- **Summary**

  Implements DQL algorithm update using only one Neural Network for both evaluation and update

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward

Dynamicmodel

Class DynamicModel

Source: world.cpp

Methods

```
size_t addStateVariable(const char name, const char units, double min, double max, bool bCircular)
```

- **Summary**

  This method must be called from the constructor of DynamicModel subclasses to register state variables. Calls are parsed by the source code parser and listed beside the parameters of the class in the class definition file (config.xml)

- **Parameters**

  - *name*: Name of the variable (i.e. "speed")
  - *units*: Metrical unit (i.e., "m/s")
  - *min*: Minimum value this variable may get. Below this, values are clamped
  - *max*: Maximum value this variable may get. Above this, values are clamped
  - *bCircular*: This flag indicates whether the variable is circular (as angles)

- **Return Value**

  This method must be called from the constructor of DynamicModel subclasses to register state variables. Calls are parsed by the source code parser and listed beside the parameters of the class in the class definition file (config.xml)

`size_t addActionVariable(const char name, const char units, double min, double max, bool bCircular)`

- **Summary**

  This method must be called from the constructor of DynamicModel subclasses to register action variables. Calls are parsed by the source code parser and listed beside the parameters of the class in the class definition file (config.xml)

- **Parameters**

  - *name*: Name of the variable (i.e. "speed")
  - *units*: Metrical unit (i.e., "m/s")
  - *min*: Minimum value this variable may get. Below this, values are clamped
  - *max*: Maximum value this variable may get. Above this, values are clamped
  - *bCircular*: This flag indicates whether the variable is circular (as angles)

- **Return Value**

  This method must be called from the constructor of DynamicModel subclasses to register action variables. Calls are parsed by the source code parser and listed beside the parameters of the class in the class definition file (config.xml)

`void addConstant(const char* name, double value)`

- **Summary**

  This method can be called from the constructor of DynamicModel subclasses to register constants. These are also parsed

- **Parameters**

  - *name*: Name of the constant
  - *value*: Literal value (i.e. 6.5). The parser will not recognise but literal values

`int getNumConstants()`

- **Summary**

  Returns the number of constants defined in the current DynamicModel subclass

`double getConstant(int i)`

- **Summary**

  Returns the value of the i-th constant

- **Parameters**

  - *i*: Index of the constant to be retrieved

- **Return Value**

  Returns the value of the i-th constant

`double getConstant(const char* constantName)`

- **Summary**

  An alternative version of getConstant() that uses the name as input

- **Parameters**

  - *constantName*: The name of the constant

- **Return Value**

  An alternative version of getConstant() that uses the name as input

`double getReward(const State s, const Action a, const State *s_p)`

- **Summary**

  This method calculates the reward associated with tuple {s,a,s_p}

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state

- **Return Value**

  This method calculates the reward associated with tuple {s,a,s_p}

Etraces

Class ETraces

Source: etraces.cpp

Methods

`void update(double factor)`

- **Summary**

  Etraces implement a technique that updates recently visited states with the current reward. This method updates the factor of each trace, so that they decay with time according to parameter Lambda. Not compatible with Experience-Replay, which is currently favored.

- **Parameters**

  - *factor*: Update factor (depends on the learning algorithm)

`void addFeatureList(FeatureList* inList, double factor)`

- **Summary**

  This method adds current state's features to the traces

- **Parameters**

  - *inList*: Features of the current state
  - *factor*: Factor given to these features

Experiencereplay

Class ExperienceReplay

Source: experience-replay.cpp

Methods

`bool bUsing()`

- **Summary**

  Returns whether Experience-Replay is enabled or not

`size_t getUpdateBatchSize()`

- **Summary**

  Returns the size of each update batch

`bool bHaveEnoughTuples()`

- **Summary**

  Returns whether there are enough tuples in the buffer to run a batch

`void addTuple(const State s, const Action a, const State* s_p, double r, double probability)`

- **Summary**

  Adds an experience tuple to the circular buffer used

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *probability*: Probability by which the action was taken

Experiment

Class Experiment

Source: experiment.cpp

Methods

`double getExperimentProgress()`

- **Summary**

  Returns the progress of the experiment (normalized in range [0,1])

`double getTrainingProgress()`

- **Summary**

  The normalized progress taking only into account the training episodes (normalized in range [0,1])

`double getEpisodeProgress()`

- **Summary**

  Normalized progress with respect to the current episode in range [0,1]

`bool isEvaluationEpisode()`

- **Summary**

  Returns whether the current is an evaluation episode

`void reset()`

- **Summary**

  Resets the experiment to the starting conditions

`void nextStep()`

- **Summary**

  Increments the current step

`bool isValidStep()`

- **Summary**

  Returns whether the current step is valid or we have already finished the episode

`bool isValidEpisode()`

- **Summary**

  Returns whether the current episode is valid or we have already finished the experiment

`void nextEpisode()`

- **Summary**

  Used to advance the simulation to the next episode

`bool isFirstEpisode()`

- **Summary**

  Is this the first episode?

`bool isLastEpisode()`

- **Summary**

  Is this the last episode?

`void timestep(State s, Action a, State s_p, Reward r)`

- **Summary**

  Called every time-step. Controls when and what information to log, and also the timers to decide if the progress must be udated

- **Parameters**

  - *s*: Initial state of the last tuple
  - *a*: Action in the last tuple
  - *_sp*: Resultant state of the last tuple
  - *r*: Reward of the last tuple

Featurelist

Class FeatureList

Source: features.cpp

Methods

`void resize(size_t newSize, bool bKeepFeatures)`

- **Summary**

  Resizes the feature list, allocating more memory

- **Parameters**

  - *newSize*: New size
  - *bKeepFeatures*: true if we want to preserve the features on the list

`void mult(double factor)`

- **Summary**

  Multiplies all the features by a factor

- **Parameters**

  - *factor*: The factor value

`double getFactor(size_t index)`

- **Summary**

  Returns the factor of a given feature on the list

- **Parameters**

  - *index*: The index of the feature

- **Return Value**

  Returns the factor of a given feature on the list

`double innerProduct(const FeatureList *inList)`

- **Summary**

  Implements an inner-product operation

- **Parameters**

  - *inList*: Second operand of the multiply operation

- **Return Value**

  Implements an inner-product operation

`void copyMult(double factor, const FeatureList *inList)`

- **Summary**

Copies a list multiplied by a factor on this feature list. Resizes the list if needed

- **Parameters**
  - *factor*: The factor to multiply by
  - *inList*: The feature list to copy

`void addFeatureList(const FeatureList *inList, double factor)`

- **Summary**

  Adds feature to this list, multiplied by a factor

- **Parameters**
  - *inList*: Feature list to be added
  - *factor*: Factor used to multiply

`void add(size_t index, double value)`

- **Summary**

  Adds a single feature

- **Parameters**
  - *index*: The index of the feature
  - *value*: The value of the feature

`void spawn(const FeatureList *inList, size_t indexOffset)`

- **Summary**

  All features (indices and values) are spawned by those in inList. This means that this list contains 2 features a 5-feature space, and inList contains 3 features a 6-feature space, after this operation, this list will contain 2**3= 6 features from a 5**6= 30 feature space

- **Parameters**
  - *inList*: Second list used as an operand
  - *indexOffset*: Feature-index offset used for the second list

`void applyThreshold(double threshold)`

- **Summary**

  Removes any feature with an activation factor under the threshold

- **Parameters**
  - *threshold*: Threshold value

`void normalize()`

- **Summary**

  Normalizes features so that the sum of all the activation factors are 1

`void copy(const FeatureList* inList)`

- **Summary**

  Copies in this list the given one

- **Parameters**
  - *inList*: Source feature list to copy

`void offsetIndices(size_t offset)`

- **Summary**

  Adds an offset to all the feature indices

- **Parameters**
  - *offset*: Offset value

`void split(FeatureList outList1, FeatureList outList2, size_t splitOffset)`

- **Summary**

  Splits this feature list in two lists: features with an index below splitOffset go to the first output list, and those above go to the second output list

- **Parameters**
  - *outList1*: Output list 1
  - *outList2*: Output list 2
  - *splitOffset*: Index used to split the feature list

`void multIndices(int mult)`

- **Summary**

  Multiplies all the feature indices by mult

- **Parameters**
  - *mult*: Value used to multiply

Featuremap

Class FeatureMap

Source: featuremap.cpp

Methods

`void getFeatures(const State s, const Action a, FeatureList* outFeatures)`

- **Summary**

  Calculates the features for any given state-action

- **Parameters**
  - *s*: State
  - *a*: Action
  - *outFeatures*: Output feature list

`void getFeatureStateAction(size_t feature, State s, Action a)`

- **Summary**

  Given a feature index, this method returns the state-action to which the feature corresponds. If the feature map uses only states, the output action is left unmodified

- **Parameters**
  - *feature*: Index of the feature
  - *s*: Output state
  - *a*: Output action

Functionsampler

Class FunctionSampler

Source: function-sampler.cpp

Methods

`size_t getNumOutputs()`

- **Summary**

  Returns the number of outputs of the sampler

Functionsampler2d

Class FunctionSampler2D

Source: function-sampler.cpp

Methods

`string getFunctionId()`

- **Summary**

  Returns the name of the function

`size_t getNumSamplesX()`

- **Summary**

  Returns the number of samples in X (the image's width)

`size_t getNumSamplesY()`

- **Summary**

  Returns the number of samples in Y (the image's height)

Functionsampler3d

Class FunctionSampler3D

Source: function-sampler.cpp

Methods

`string getFunctionId()`

- **Summary**

  Returns the name of the function

`size_t getNumSamplesX()`

- **Summary**

  Returns the number of samples in X (the image's width)

`size_t getNumSamplesY()`

- **Summary**

  Returns the number of samples in Y (the image's height)

Gaussiannoise

Class GaussianNoise

Source: noise.cpp

Methods

`double getNormalDistributionSample(double mean, double sigma)`

- **Summary**

  Returns a sample from a Gaussian distribution function. Used to generate noise

- **Parameters**

  - *mean*: Mean value of the distribution
  - *sigma*: Sigma of the distribution

- **Return Value**

  Returns a sample from a Gaussian distribution function. Used to generate noise

`double getSample()`

- **Summary**

  Returns a sample from a Gaussian distribution function. Used to generate noise

- **Return Value**

  Returns a sample from a Gaussian distribution function. Used to generate noise

Gaussianrbfgridfeaturemap

Class GaussianRBFGridFeatureMap

Source: featuremap-rbfgrid.cpp

Methods

```
void map(vector>& grids, const vector& values, FeatureList outFeatures)
```

- **Summary**

  Implements a Gaussian Radial-Basis feature mapping function that maps state-actions to feature.

- **Parameters**

  - *grids*: Input grids for every state-variable used
  - *values*: The values of every state-variable used
  - *outFeatures*: The output list of features

```
void unmap(size_t feature, vector& grids, vector& outValues)
```

- **Summary**

  Inverse of the feature mapping operation. Given a feature it returns the state-action to which it corresponds.

- **Parameters**

  - *feature*: The index of the feature
  - *grids*: The set of grids used to discretize each variable
  - *outValues*: The set of output values for every state-action variable

Greedyqplusnoisepolicy

Class GreedyQPlusNoisePolicy

Source: q-learners.cpp

Methods

```
double selectAction(LinearStateActionVFA pQFunction, const State s, Action* a)
```

- **Summary**

  Implements an action selection policy that adds noise to the greedily selected action

- **Parameters**

  - *pQFunction*: The Q-function
  - *s*: Current state
  - *a*: Output action

- **Return Value**

  Implements an action selection policy that adds noise to the greedily selected action

Incrementalnaturalactorcritic

Class IncrementalNaturalActorCritic

Source: actor-critic-inac.cpp

Methods

```
double update(const State s, const Action a, const State *s_p, double r, double behaviorProb)
```

- **Summary**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Model-free Reinforcement Learning with Continuous Action in Practice" (Thomas Degris, Patrick M. Pilarski , Richard S. Sutton), 2012 American Control Conference

- **Parameters**

  - *s*: Initial state
  - *a*: Action

- *_sp*: Resultant state
- *r*: Reward
- *behaviorProb*: Probability by which the actor selected the action

- **Return Value**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Model-free Reinforcement Learning with Continuous Action in Practice" (Thomas Degris, Patrick M. Pilarski , Richard S. Sutton), 2012 American Control Conference

`double selectAction(const State s, Action a)`

- **Summary**

  The actor selects an action following the policies it is learning

- **Parameters**
  - *s*: Initial state
  - *a*: Action

- **Return Value**

  The actor selects an action following the policies it is learning

Interpolatedvalue

Class InterpolatedValue

Source: parameters-numeric.cpp

Methods

`double get()`

- **Summary**

  Returns a sample from a linear function determined by linear interpolation between (x1,y1) and (x2,y2), where x1 are x2 are given as normalized experiment progress

- **Return Value**

  Returns a sample from a linear function determined by linear interpolation between (x1,y1) and (x2,y2), where x1 are x2 are given as normalized experiment progress

Linearstateactionvfa

Class LinearStateActionVFA

Source: vfa.cpp

Methods

`void getFeatures(const State s, const Action a, FeatureList* outFeatures)`

- **Summary**

  Given a state-action pair, it calculates the features for each feature map separately (state and action) and then combines using spawn() and offsetIndices() so that the resultant features belong to the full state-action feature space

- **Parameters**
  - *s*: State
  - *a*: Action
  - *outFeatures*: Output feature list

`void getFeatureStateAction(size_t feature, State s, Action a)`

- **Summary**

  Given a feature index, it returns in s and a the values of the variables to which the feature corresponds

- **Parameters**
  - *feature*: Index of the feature
  - *s*: Output state

- *a*: Output action

`double get(const State s, const Action a)`

- **Summary**

  Evaluates Q(s,a)

`void argMax(const State s, Action a, bool bSolveTiesRandomly)`

- **Summary**

  Calculates the action a that maximizes Q(s,a)

- **Parameters**

  - *s*: State
  - *a*: Output action that maximizes Q(s,a)
  - *bSolveTiesRandomly*: In case of tie, this flag sets whether return a random action or the first one

`double max(const State* s, bool bUseFrozenWeights)`

- **Summary**

  Calculates the maximum value of Q(s,a) for state s

- **Parameters**

  - *s*: State
  - *bUseFrozenWeights*: If set and it makes sense, will use the target function

- **Return Value**

  Calculates the maximum value of Q(s,a) for state s

`void getActionValues(const State s,double outActionValues)`

- **Summary**

  Returns an array with the values for each action feature

- **Parameters**

  - *s*: State
  - *outActionValues*: Output action values, one for every feature in the action feature map

Linearstatevfa

Class LinearStateVFA

Source: vfa.cpp

Methods

`void setInitValue(double initValue)`

- **Summary**

  Sets the initial value of the function

- **Parameters**

  - *initValue*:

`void getFeatures(const State s, FeatureList outFeatures)`

- **Summary**

  Uses the state feature map to calculate the features of a state

`void getFeatureState(size_t feature, State* s)`

- **Summary**

  Given a feature, it uses the state feature map to return the state variable's value in s

`double get(const State *s)`

- **Summary**

Evaluates V(s)

Linearvfa

Class LinearVFA

Source: vfa.cpp

Methods

`double get(const FeatureList *pFeatures,bool bUseFrozenWeights)`

- **Summary**

  Returns the value of the linear Value Function Approximator for the input state-action given as a list of features.

- **Parameters**

  - *pFeatures*: Input list of features
  - *bUseFrozenWeights*: Flag used to determine whether to use the online or target function

- **Return Value**

  Returns the value of the linear Value Function Approximator for the input state-action given as a list of features.

`void saturateOutput(double min, double max)`

- **Summary**

  Sets the function to saturate its output in range [min,max]

`void setIndexOffset(unsigned int offset)`

- **Summary**

  Sets the index offset used. Handy if we want to represent f(s,a) with two different feature maps: one for the state and another one for the action

- **Parameters**

  - *offset*: Offset added to feature indices

`void add(const FeatureList* pFeatures, double alpha)`

- **Summary**

  Adds a feature list (each feature has an index and a factor) to the weights in the function. Some of the indices might not belong to this function

- **Parameters**

  - *pFeatures*: Feature list to be added
  - *alpha*: Gain parameter used to move current weights toward those in the feature list

`void set(size_t feature, double value)`

- **Summary**

  Sets the value of a function weight

Logger

Class Logger

Source: logger-functions.cpp

Methods

`void openFunctionLogFile(const char* filename)`

- **Summary**

  Creates a file where functions will be logged

- **Parameters**

  - *filename*: Path to the output file

`void closeFunctionLogFile()`

- **Summary**

Closes the file used for logging functions

`void writeFunctionLogSample()`

- **Summary**

  Adds a sample from each function to the log file

`void setOutputFilenames()`

- **Summary**

  Registers the output files

`bool isEpisodeTypeLogged(bool evalEpisode)`

- **Summary**

  Returns whether the given type of episode is being logged

- **Parameters**

  - *evalEpisode*: true if we want to query about evaluation episodes, false otherwise

`void writeLogFileXMLDescriptor(const char* filename)`

- **Summary**

  Creates an XML file with the description of the log file: variables, scene file...

- **Parameters**

  - *filename*:

`void firstEpisode()`

- **Summary**

  Must be called before the first episode begins to initialize log files, timers,...

`void firstStep()`

- **Summary**

  Must be called before the first step to write episode headers and the initial state in the log file. It also takes a snapshot of the functions and logs them if we are logging functions

`void lastStep()`

- **Summary**

  Must be called after the last step in an episode has finished. The episode is marked as finished in the log file and, if the current is an evaluation episode, the progress is updated

`void timestep(State s, Action a, State s_p, Reward r)`

- **Summary**

  Logs if needed a new step {s,a,s_p,r}, and adds a new sample to statistics

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward

`void logMessage(MessageType type, const char* message)`

- **Summary**

  Logging function that formats and dispatches different types of messages: info/warnings/errors, progress, and evaluation. Depending on whether the app is connected via a named pipe, the message is either sent via the pipe or printed on the system console. Error log messages throw an exception to terminate the program

- **Parameters**

  - *type*:
  - *message*:

Lqrcontroller

Class LQRController

Source: controller.cpp

Methods

`double evaluate(const State s, const Action a, unsigned int index)`

- **Summary**

  Calculates one of the outputs of the LQR controller

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *index*: Index of the output

- **Return Value**

  Calculates one of the outputs of the LQR controller

Memblock

Class MemBlock

Source: mem-block.cpp

Methods

`void dumpToFile()`

- **Summary**

  Saves the contents of the memory block to a temporary file.

`void restoreFromFile()`

- **Summary**

  Restores the contents of a memory block from file

Offpolicyactorcritic

Class OffPolicyActorCritic

Source: actor-critic-offpac.cpp

Methods

`double update(const State s, const Action a, const State *s_p, double r, double behaviorProb)`

- **Summary**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Off-Policy Actor-Critic" (Thomas Degris, Martha White, Richard S. Sutton), Proceedings of the 29 th International Conference on Machine Learning, Edinburgh, Scotland, UK, 2012. arXiv:1205.4839v5 [cs.LG] 20 Jun 2013

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *behaviorProb*: Probability by which the actor selected the action

- **Return Value**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Off-Policy Actor-Critic" (Thomas Degris, Martha White, Richard S. Sutton), Proceedings of the 29 th International Conference on Machine Learning, Edinburgh, Scotland, UK, 2012. arXiv:1205.4839v5 [cs.LG] 20

Jun 2013

`double selectAction(const State s, Action a)`

- **Summary**

  The actor selects an action following the policies it is learning

- **Parameters**

  - *s*: Initial state
  - *a*: Action

- **Return Value**

  The actor selects an action following the policies it is learning

Offpolicydeterministicactorcritic

Class OffPolicyDeterministicActorCritic

Source: actor-critic-opdac.cpp

Methods

`double update(const State s, const Action a, const State *s_p, double r, double behaviorProb)`

- **Summary**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Off-policy deterministic actorcritic (OPDAC)" in "Deterministic Policy Gradient Algorithms" (David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, Martin Riedmiller). Proceedings of the 31 st International Conference on Machine Learning, Beijing, China, 2014. JMLR: WCP volume 32

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *behaviorProb*: Probability by which the actor selected the action

- **Return Value**

  Updates the policy and the value function using the Incremental Natural Actor Critic algorithm in "Off-policy deterministic actorcritic (OPDAC)" in "Deterministic Policy Gradient Algorithms" (David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, Martin Riedmiller). Proceedings of the 31 st International Conference on Machine Learning, Beijing, China, 2014. JMLR: WCP volume 32

`double selectAction(const State s, Action a)`

- **Summary**

  The actor selects an action following the policies it is learning

- **Parameters**

  - *s*: Initial state
  - *a*: Action

- **Return Value**

  The actor selects an action following the policies it is learning

Ornsteinuhlenbecknoise

Class OrnsteinUhlenbeckNoise

Source: noise.cpp

Methods

`double getSample()`

- **Summary**

  Returns a sample from an Ornstein Uhlenbeck process: https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process Used to generate temporally-

correlated noise

- **Return Value**

  Returns a sample from an Ornstein Uhlenbeck process: https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process Used to generate temporally-correlated noise

Pidcontroller

Class PIDController

Source: controller.cpp

Methods

```
double evaluate(const State s, const Action a, unsigned int output)
```

- **Summary**

  Calculates one of the outputs of the PID controller

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *index*: Index of the output

- **Return Value**

  Calculates one of the outputs of the PID controller

Qegreedypolicy

Class QEGreedyPolicy

Source: q-learners.cpp

Methods

```
double selectAction(LinearStateActionVFA pQFunction, const State s, Action* a)
```

- **Summary**

  Implements an epsilon-greedy action selection policy, selecting the action with the maximum Q(s,a) value

- **Parameters**

  - *pQFunction*: The Q-function
  - *s*: Current state
  - *a*: Output action

- **Return Value**

  Implements an epsilon-greedy action selection policy, selecting the action with the maximum Q(s,a) value

Qlearningcritic

Class QLearningCritic

Source: q-learners.cpp

Methods

```
double update(const State s, const Action a, const State *s_p, double r, double probability)
```

- **Summary**

  Updates the estimate of the Q-function using the Q-Learning update rule with tuple {s,a,s_p,r}

- **Parameters**

  - *s*: Initial state
  - *a*: Action

- *_sp*: Resultant state
- *r*: Reward

- **Return Value**

  Updates the estimate of the Q-function using the Q-Learning update rule with tuple {s,a,s_p,r}

Qsoftmaxpolicy

Class QSoftMaxPolicy

Source: q-learners.cpp

Methods

```
double selectAction(LinearStateActionVFA pQFunction, const State s, Action* a)
```

- **Summary**

  Implements a Soft-Max action selection policy controlled by temperature parameter Tau

- **Parameters**
  - *pQFunction*: The Q-function
  - *s*: Current state
  - *a*: Output action

- **Return Value**

  Implements a Soft-Max action selection policy controlled by temperature parameter Tau

Regularpolicygradientlearner

Class RegularPolicyGradientLearner

Source: actor-regular.cpp

Methods

```
void update(const State s, const Action a, const State *s_p, double r, double td)
```

- **Summary**

  Updates the policies using a regular gradient-descent update rule

- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *behaviorProb*: Probability by which the actor selected the action

Rewardfunction

Class RewardFunction

Source: reward.cpp

Methods

```
void addRewardComponent(IRewardComponent* rewardComponent)
```

- **Summary**

  RewardFunction can use more than one scalar reward and they are added using this method. Scalar rewards must derive from IRewardComponent

- **Parameters**
  - *rewardComponent*: The new scalar reward to be added

```
double getReward(const State s, const Action a, const State* s_p)
```

- **Summary**

  Calculates the total reward based on the different scalar rewards. If we only define one reward function, its value will be returned

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state

- **Return Value**

  Calculates the total reward based on the different scalar rewards. If we only define one reward function, its value will be returned

`void initialize()`

- **Summary**

  DynamicModel subclasses should call this initialization method after adding the reward functions

`void override(double reward)`

- **Summary**

  If we want to override the final reward in some special states (i.e. a negative reward if FAST simulator crashed) we can call this method from the DynamicModel

- **Parameters**

  - *reward*: The reward we want to give the agent

Sarsa

Class SARSA

Source: q-learners.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  implements SARSA On-policy action selection algorithm

- **Parameters**

  - *s*: Initial state
  - *a*: Output action

`double update(const State s, const Action a, const State* s_p, double r, double probability)`

- **Summary**

  Updates the estimate of the Q-function using the SARSA update rule with tuple {s,a,s_p,r}

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward

- **Return Value**

  Updates the estimate of the Q-function using the SARSA update rule with tuple {s,a,s_p,r}

Simgod

Class SimGod

Source: simgod.cpp

Methods

`double selectAction(State s, Action a)`

- **Summary**

  Iterates over all the Simions to let each of them select their actions

- **Parameters**

  - *s*: Initial state
  - *a*: Action variable where Simions write their selected actions

- **Return Value**

  Iterates over all the Simions to let each of them select their actions

`void update(State s, Action a, State* s_p, double r, double probability)`

- **Summary**

  Iterates over all the Simions to let them learn from the last real-time experience tuple

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *probability*: Probability by which the action was taken. Should be ignored

`void postUpdate()`

- **Summary**

  If Experience-Replay is enabled, several tuples are taken from the buffer and given to the Simions to learn from them

`void deferredLoad()`

- **Summary**

  Iterates over all the objects implementing DeferredLoad to do all the heavyweight-lifting stuff

`double getGamma()`

- **Summary**

  Returns the value of gamma

`int getTargetFunctionUpdateFreq()`

- **Summary**

  Returns the number of steps after which deferred V-Function updates are to be done. 0 if we don't use Freeze-V-Function

- **Return Value**

  Returns the number of steps after which deferred V-Function updates are to be done. 0 if we don't use Freeze-V-Function

`bool bUpdateFrozenWeightsNow()`

- **Summary**

  Returns whether we need to update the "frozen" copies of any function using Freeze-Target-Function

- **Return Value**

  Returns whether we need to update the "frozen" copies of any function using Freeze-Target-Function

`size_t getExperienceReplayUpdateSize()`

- **Summary**

  Returns the number of each update batch using Experience-Replay

- **Return Value**

  Returns the number of each update batch using Experience-Replay

Simionapp

Class SimionApp

Source: app.cpp

Methods

`void printRequirements()`

- **Summary**

  This method prints the run-time requirements of an experiment instead of running it

`void registerStateActionFunction(string name, StateActionFunction* pFunction)`

- **Summary**

  Called from function-learning objects to register an instance of a function to log (if configured to do so)

- **Parameters**

  - *name*: Name of the function
  - *pFunction*: Pointer to the function

`void wireRegister(string name)`

- **Summary**

  Wires allow us to connect inputs with outputs. This method registers a wire by name

- **Parameters**

  - *name*: The new wire's name

`void wireRegister(string name, double minimum, double maximum)`

- **Summary**

  This method registers a wire by name and also sets its value range

- **Parameters**

  - *name*:
  - *minimum*:
  - *maximum*:

`void registerTargetPlatformInputFile(const char targetPlatform, const char filepath, const char* rename)`

- **Summary**

  Used to register input files to a specific platform.

- **Parameters**

  - *targetPlatform*: Target platform: Win-32, Win-64, Linux-64,...
  - *filepath*: Path to the required file (exe, dll, data file, ...)
  - *rename*: Name given to the required file in the host machine

`void registerTargetPlatformOutputFile(const char targetPlatform, const char filepath)`

- **Summary**

  Used to register output files to a specific platform

- **Parameters**

  - *targetPlatform*: Target platform: Win-32, Win-64, Linux-64,...
  - *filepath*: Path to the required file (exe, dll, data file, ...)

`void registerInputFile(const char filepath, const char rename)`

- **Summary**

  Used to register input files common to all the target platforms

- **Parameters**

  - *filepath*: Path to the required file (exe, dll, data file, ...)
  - *rename*: Name given to the required file in the host machine

`void registerOutputFile(const char* filepath)`

- **Summary**

  Used to register output files common to all the target platforms

- **Parameters**

  - *filepath*: Path to the required file (exe, dll, data file, ...)
  - *rename*: Name given to the required file in the host machine

`void run()`

- **Summary**

  The app's main-loop that starts the simulation and runs until it finishes.

`void initRenderer(string sceneFile, State s, Action a)`

- **Summary**

  This method is called from run() and initializes the real-time rendering window (if configured to do so)

- **Parameters**

  - *sceneFile*: Name of the file with the definition of the scene
  - *s*: Current state
  - *a*: Current action

`void initFunctionSamplers(State s, Action a)`

- **Summary**

  After adding all the StateActionFunctions to be logged/drawn, this method is called to initialize them

- **Parameters**

  - *s*: State
  - *a*: Action

`void updateScene(State s, Action a)`

- **Summary**

  Updates the graphical objects bound in the scene file using the current value of their bounded state/action variables

- **Parameters**

  - *s*:
  - *a*:

Simionmembuffer

Class SimionMemBuffer

Source: mem-buffer.cpp

Methods

`BUFFER_SIZE getBlockSizeInBytes()`

- **Summary**

  Returns the size of a memory block in the parent memory pool

- **Return Value**

  Returns the size of a memory block in the parent memory pool

Simionmempool

Class SimionMemPool

Source: mem-pool.cpp

Methods

`void initialize(MemBlock* pBlock)`

- **Summary**

  This method resets each interleaved buffer within a MemBlock to its initial value

- **Parameters**

  - *pBlock*: Memory block

`void init(BUFFER_SIZE blockSize)`

- **Summary**

  After adding all the requested buffers, this method initializes all the necessary data so that several MemBlocks are allocated

- **Parameters**

  - *blockSize*: Desired MemBlock size (element count)

`void copy(IMemBuffer pSrc, IMemBuffer pDst)`

- **Summary**

  Copies data from one buffer to another. They must belong to the same MemPool

- **Parameters**

  - *pSrc*: Source buffer
  - *pDst*: Destination buffer

Simpleepisodelinearschedule

Class SimpleEpisodeLinearSchedule

Source: parameters-numeric.cpp

Methods

`double get()`

- **Summary**

  Returns a sample of a linear function determined by a starting value, an ending value and the current experiment progress

- **Return Value**

  Returns a sample of a linear function determined by a starting value, an ending value and the current experiment progress

Singledimensiongrid

Class SingleDimensionGrid

Source: single-dimension-grid.cpp

Methods

`size_t getClosestFeature(double value)`

- **Summary**

  Within the one-dimension grid, this method returns the index of the feature closest to the given value

- **Parameters**

  - *value*: Value of the variable to which this grid corresponds

- **Return Value**

  Within the one-dimension grid, this method returns the index of the feature closest to the given value

`double getFeatureValue(size_t feature)`

- **Summary**

  Given a feature index, it returns the value of the variable to which this single-dimension grid corresponds

- **Parameters**

  - *feature*:

- **Return Value**

Given a feature index, it returns the value of the variable to which this single-dimension grid corresponds

SinusoidalNoise

Class SinusoidalNoise

Source: noise.cpp

Methods

`double getSample()`

- **Summary**

  Returns a sample from a sinusoidal signal. Used to generate noise

- **Return Value**

  Returns a sample from a sinusoidal signal. Used to generate noise

Statsinfo

Class StatsInfo

Source: stats.cpp

Methods

`void reset()`

- **Summary**

  Resets the stats

`void addSample(double value)`

- **Summary**

  Adds a sample to the collection of values

- **Parameters**

  - *value*: New sample

`double getMin()`

- **Summary**

  Returns the maximum sampled value

`double getMax()`

- **Summary**

  Returns the minimum sampled value

`double getAvg()`

- **Summary**

  Returns the mean value of all samples

`double getStdDev()`

- **Summary**

  Returns the standard deviation of all the samples

Stochasticgaussianpolicy

Class StochasticGaussianPolicy

Source: vfa-policy.cpp

Methods

`double selectAction(const State s, Action a)`

- **Summary**

  A DeterministicPolicyGaussianNoise policy uses a function to represent the mean value of the function at each state and a second function to represent the variance of the output at each state. To calculate the policy's output, the two functions are evaluated for a state, and then the values used to sample a normal distribution, which is the actual output of the policy

- **Parameters**

  - *s*: Initial state
  - *a*: Output state

```
void getFeatures(const State state, FeatureList outFeatureList)
```

- **Summary**

  Uses the policy's feature map to return the features representing the state

- **Parameters**

  - *state*: State
  - *outFeatureList*: Output feature list

Tdclambdacritic

Class TDCLambdaCritic

Source: critic-tdc-lambda.cpp

Methods

```
double update(const State s, const Action a, const State *s_p, double r, double rho)
```

- **Summary**

  Updates the value function using the TDC update rule

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *rho*: Importance sampling

- **Return Value**

  Updates the value function using the TDC update rule

Tdlambdacritic

Class TDLambdaCritic

Source: critic-td-lambda.cpp

Methods

```
double update(const State s, const Action a, const State *s_p, double r, double rho)
```

- **Summary**

  Updates the value function using the popular TD(lambda) update rule

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *rho*: Importance sampling

- **Return Value**

Updates the value function using the popular TD(lambda) update rule

Tilecodingfeaturemap

Class TileCodingFeatureMap

Source: featuremap-tilecoding.cpp

Methods

```
void map(vector>& grids, const vector& values, FeatureList outFeatures)
```

- **Summary**

  Implements a Tile-Coding feature mapping function that maps state-actions to feature: https://www.cs.utexas.edu/~pstone/Papers/bib2html-links/SARA05.slides.pdf

- **Parameters**

  - *grids*: Input grids for every state-variable used
  - *values*: The values of every state-variable used
  - *outFeatures*: The output list of features

```
void unmap(size_t feature, vector& grids, vector& outValues)
```

- **Summary**

  Inverse of the feature mapping operation. Given a feature it returns the state-action to which it corresponds.

- **Parameters**

  - *feature*: The index of the feature
  - *grids*: The set of grids used to discretize each variable
  - *outValues*: The set of output values for every state-action variable

Toleranceregionreward

Class ToleranceRegionReward

Source: reward.cpp

Methods

```
double getReward(const State s, const Action a, const State *s_p)
```

- **Summary**

  This reward function returns a reward depending on the value of an error variable with respect to its tolerance region

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state

- **Return Value**

  This reward function returns a reward depending on the value of an error variable with respect to its tolerance region

Trueonlinetdlambdacritic

Class TrueOnlineTDLambdaCritic

Source: critic-true-online-td-lambda.cpp

Methods

```
double update(const State s,const Action a,const State *s_p, double r, double rho)
```

- **Summary**

  Updates the value function using the True-Online TD(Lambda) update rule in "True Online TD(lambda)" (Harm van Seijen, Richard Sutton), Proceedings of the 31st International Conference on Machine learning

- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *_sp*: Resultant state
  - *r*: Reward
  - *rho*: Importance sampling
- **Return Value**

  Updates the value function using the True-Online TD(Lambda) update rule in "True Online TD(lambda)" (Harm van Seijen, Richard Sutton), Proceedings of the 31st International Conference on Machine learning

Windturbineboukhezzarcontroller

Class WindTurbineBoukhezzarController

Source: controller.cpp

Methods

```
double evaluate(const State s,const Action a, unsigned int output)
```

- **Summary**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Boukhezzar
- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *index*: Index of the output
- **Return Value**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Boukhezzar

Windturbinejonkmancontroller

Class WindTurbineJonkmanController

Source: controller.cpp

Methods

```
double evaluate(const State s,const Action a, unsigned int output)
```

- **Summary**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Jonkman
- **Parameters**
  - *s*: Initial state
  - *a*: Action
  - *index*: Index of the output
- **Return Value**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Jonkman

Windturbinevidalcontroller

Class WindTurbineVidalController

Source: controller.cpp

Methods

```
double evaluate(const State s, const Action a, unsigned int output)
```

- **Summary**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Vidal

- **Parameters**

  - *s*: Initial state
  - *a*: Action
  - *index*: Index of the output

- **Return Value**

  Calculates one of the outputs of the Variable-Speed Wind Turbine by Vidal

Wireconnection

Class WireConnection

Source: parameters-numeric.cpp

Methods

`double get()`

- **Summary**

  Returns the current value of a wire connection

World

Class World

Source: world.cpp

Methods

`double getDT()`

- **Summary**

  This method returns the Delta_t used in the experiment

- **Return Value**

  This method returns the Delta_t used in the experiment

`void reset(State *s)`

- **Summary**

  Reset state variables to the initial state from which simulations begin (it may be random)

- **Parameters**

  - *s*: State variable that holds the initial state

`double executeAction(State s, Action a, State *s_p)`

- **Summary**

  Method called every control time-step. Internally it calculates calculates the length of the integration steps and calls several times DynamicModel::executeAction()

- **Parameters**

  - *s*: The variable with the current state values
  - *a*: The action to be executed
  - *_sp*: The variable that will hold the resultant state

- **Return Value**

  Method called every control time-step. Internally it calculates calculates the length of the integration steps and calls several times DynamicModel::executeAction()