Definitions:

- *Compound*:
  A chemical substance defined by particular set of physical properties, calculation methods and data. Examples: water, hydrogen, oxygen.
- *Phase*:
  Stable collection of compounds with a defined amount of substance and a homogeneous composition. It has an associated state of aggregation, e.g. liquid.
- *State of aggregation*:
  Is the physical state in which the compounds in that phase occur. Possible state of aggregation: vapor, liquid, solid.
- *Class*:
  A software component which consists of member variables (data fields) and associated functions (methods).
- *Objects*:
  Independent instances of specified class.

Interfaces of basic unit:

Functions to work with the basic info of the unit
GetUnitName *GetAuthorName* *GetUniqueID* *GetUnitVersion*
Functions to work with ports
AddPort *GetPortsNumber* *GetPortType* *GetPortStream*
Functions to work with material streams and holdups
AddHoldup *GetHoldup* *GetHoldups* *RemoveHoldup* *AddMaterialStream* *GetMaterialStream* *RemoveMaterialStream* *AddFeed* *GetFeed* *CopyStreamToStream* *CopyStreamToPort* *CopyPortToStream* *CalcTemperatureFromProperty* *CalcPressureFromProperty* *HeatExchange*
Virtual functions which can be overriden in inherited classes
Simulate *Initialize* *SaveState* *LoadState* *Finalize*
Functions to work with time points
GetAllInputTimePoints *GetAllDefinedTimePoints* *GetAllStreamsTimePoints*
Functions to work with unit parameters
AddConstParameter *AddTDParameter* *AddStringParameter* *AddSolverAggregation* *GetParametersNumber* *GetParameterName* *GetParameterMinVal* *GetParameterMaxVal* *GetConstParameterValue* *GetTDParameterValue* *GetStringParameterValue* *GetSolverAggregation*

SetParameterValue *GetParameterTimePoints*
Functions to work with state variables
AddStateVariable *GetStateVariablesNumber* *GetStateVariableName* *GetStateVariable* *SetStateVariable* *ClearStateVariables* *SaveStateVariables*
Functions to work with compounds
GetCompoundsList *GetCompoundsNames* *GetCompoundsNumber* *GetCompoundConstant* *GetCompoundTPDProp* *GetCompoundsInteractionProp* *IsCompoundNameDefined* *IsCompoundKeyDefined* *IsPropertyDefined*
Functions to work with phases
GetPhasesNumber *GetPhaseName* *GetPhaseSOA* *GetPhaseIndex* *IsPhaseDefined* *GetLiquidPhasesNumber*
Functions to work with solid distributed properties
GetDistributionsTypes *GetDistributionsClasses* *GetDistributionsNumber* *GetDistributionGridType* *GetNumericGrid* *GetSymbolicGrid* *GetClassesNumber* *GetClassesMeans* *GetPSDGridDiameters* *GetPSDGridVolumes* *GetPSDMeanDiameters* *GetPSDMeanSurfaces* *GetPSDMeanVolumes* *GetClassesSizes* *IsDistributionDefined* *CalculateTM*
Functions to work with tolerances
GetAbsTolerance *GetRelTolerance*
Functions to work with errors, warnings and logging info
RaiseError *RaiseWarning* *ShowInfo*
Functions to work with plots

AddPlot *AddCurveOnPlot* *AddPointOnCurve*
Tables of properties

Functions to work with the basic info of the unit

std::string GetUnitName *()*
Returns name of the unit.

std::string GetAuthorName *()*
Returns name of the unit's author.

std::string GetUniqueID *()*
Returns unique identifier of the unit.

std::string GetUnitVersion *()*
Returns version of the unit.

Functions to work with ports

unsigned AddPort *(std::string PortName, unsigned PortType)*
Adds port with *PortType* (*INPUT_PORT* or *OUTPUT_PORT*) to the unit. Returns index of the port. Should be used in unit's constructor only. *PortName* should be unique within the unit.

unsigned GetPortsNumber *()*
Returns number of ports in the unit.

unsigned GetPortType *(std::string PortName)*
Returns type of the port with name *PortName*. Returning values: *INPUT_PORT*, *OUTPUT_PORT*. If port with such name has not been defined *UNDEFINED_PORT* will be returned.

*CMaterialStream** GetPortStream *(std::string PortName)*
Returns pointer to the stream which is connected to the port with name *PortName*. Returns *NULL* if such port has not been defined.

Functions to work with material streams and holdups

*CHoldup** AddHoldup *(std::string HoldupName, std::string StreamKey = "")*
Adds new holdup with the specified name *HoldupName* to the unit. *HoldupName* should be unique within the unit. The structure of the holdup will be the same as the global stream's structure (phases, grids, compounds etc.). Should be used in unit's constructor; then the holdup will be automatically handled by the simulation system (saved and loaded during the simulation, cleared and removed after use). However, it is allowed to add holdups outside the constructor for temporal purposes, but saving and loading of this holdup during the simulation should be done then manually (in functions *SaveState()* and *LoadState()* of the unit), as well as its removing after use (with the help of function *RemoveHoldup()* (otherwise all such holdups will be removed at the end of the simulation)). Returns pointer to a created holdup. This pointer should not be used inside the constructor of the unit, since all changes of the holdup made through this pointer will be cancelled during the initialization of the unit.

*CHoldup** GetHoldup *(std::string HoldupName)*
Returns pointer to a holdup with the specified name *HoldupName*. This pointer should not be used inside the constructor of the unit, since all changes of the holdup made through this pointer will be cancelled during the initialization of the unit. *NULL* will be returned if such holdup has not been defined.

std::vector<CHoldup>* GetHoldups *()*

Returns pointers to all holdups of the unit. These pointers should not be used inside the constructor of the unit, since all changes of the holdup made through them will be cancelled during the initialization of the unit.

void RemoveHoldup *(std::string HoldupName)*

Removes holdup with the specified name *HoldupName* from the unit. Should be used only for those holdups, which have been added to the unit outside the constructor.

*CMaterialStream** AddMaterialStream *(std::string StreamName, std::string StreamKey = "")*

Adds new material stream with the specified name *StreamName* for internal temporary use to the unit. *StreamName* should be unique within the unit. Structure of the stream will be the same as the global stream's structure (phases, grids, compounds etc.). Should be used in unit's constructor; then the material stream will be automatically handled by the simulation system (saved and loaded during the simulation, cleared and removed after use). However, it is allowed to add material outside the constructor for temporal purposes, but saving and loading of this stream during the simulation should be done then manually (in functions *SaveState()* and *LoadState()* of the unit), as well as its removing after use (with the help of function *RemoveMaterialStream()* (otherwise all such material streams will be removed at the end of the simulation)). Returns pointer to a created material stream.

*CMaterialStream** GetMaterialStream *(std::string StreamName)*

Returns pointer to a material stream with specified name *StreamName*. *NULL* will be returned if such stream has not been defined.

void RemoveMaterialStream *(std::string StreamName)*

Removes material stream with the specified name *StreamName* from the unit. Should be used only for those material streams, which have been added to the unit outside the constructor.

*CMaterialStream** AddFeed *(std::string FeedName, std::string StreamKey = "")*

Adds new feed stream with the name *FeedName* to the unit. *FeedName* should be unique within the unit. The structure of the stream will be the same as the global stream structure (phases, grids, compounds etc.). Should be used only in constructor of the unit, which describes feed. Returns pointer to a created stream. This pointer should not be used inside the constructor of the unit, since all changes of the stream made through this pointer will be cancelled during the initialization of the unit.

*CMaterialStream** GetFeed *(std::string FeedName)*

Returns pointer to a feed with specified name *FeedName*. This pointer should not be used inside the constructor of the unit, since all changes of the stream made through this pointer will be cancelled during the initialization of the unit. *NULL* will be returned if such feed has not been defined.

void CopyStreamToStream *(CMaterialStream* SrcStream, CMaterialStream DstStream, double Time, bool DeleteDataAfter = true)*

Copies all data from *SrcStream* to *DestStream* for specified time point. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *Time* in the destination stream will be removed.

void CopyStreamToStream *(CMaterialStream* SrcStream, CMaterialStream DstStream, double StartTime, double EndTime, bool DeleteDataAfter = true)*

Copies all data from *SrcStream* to *DstStream* on a specified time interval. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *StartTime* in the destination stream will be removed.

void CopyStreamToPort *(CMaterialStream* Stream, std::string PortName, double Time, bool DeleteDataAfter = true)*

Copies all data from *Stream* to a stream connected to an output port *PortName* for specified time point. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *Time* in the destination stream will be removed.

void CopyStreamToPort *(CMaterialStream* Stream, std::string PortName, double StartTime, double EndTime, bool DeleteDataAfter = true)*

Copies all data from *Stream* to a stream connected to an output port *PortName* for specified time interval. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *StartTime* in the destination stream will be removed.

void CopyPortToStream *(std::string PortName, CMaterialStream* Stream, double Time, bool DeleteDataAfter = true)*

Copies stream's data of the input port *PortName* to another stream *Stream* for specified time point. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *Time* in the destination stream will be removed.

void CopyPortToStream *(std::string PortName, CMaterialStream* Stream, double StartTime, double EndTime, bool DeleteDataAfter = true)*

Copies stream's data of the input port *PortName* to another stream *Stream* for specified time interval. If flag *DeleteDataAfter* is set to *true*, then before copying all data after the time point *StartTime* in the destination stream will be removed.

double CalcTemperatureFromProperty *(ECompoundTPProperties Property, std::vector<double> CompoundFractions, double Value)*

Returns temperature of a generic system of composition *CompoundFractions* for a specific value *Value* of the property *Property*. Possible properties are those defined in material database. For more information, refer to "Thermodynamics.pdf".

double CalcPressureFromProperty *(ECompoundTPProperties Property, std::vector<double> CompoundFractions, double Value)*

Returns pressure of a generic system of composition *CompoundFractions* for a specific value *Value* of the property *Property*. Possible properties are those defined in material database. For more information, refer to "Thermodynamics.pdf".

void HeatExchange (CMaterialStream* Stream1, CMaterialStream* Stream2, double Time, double Efficiency);

Performs a heat exchange between material streams *Stream1* and *Stream2* at specified time point with a specified efficiency (0 ≤ *Efficiency* ≤ 1). For more information, refer to "Thermodynamics.pdf".

Virtual functions which can be overriden in inherited classes

void Simulate *(double Time)*

Calculates unit on a specified time point *Time* (for steady-state units). Is called by the suimulator iteratively for all time points for which this unit should be calculated. All logic of the unit's model must be implemented here.

void Simulate *(double StartTime, double EndTime)*

Calculates unit for specified time interval (for dynamic units). Is called by the suimulator iteratively for all time points for which this unit should be calculated. All logic of the unit's model must be implemented here.

void Initialize *(double Time)*
Initializes unit at the time point *Time*. Is called by the simulator only once at the start of the simulation. Here some additionaly objects can be initialized (for example holdups, material streams or state variables).

void SaveState *()*
Saves current state of the unit. All time dependent variables which weren't added to the unit with the help of *AddStateVariable()*, *AddMaterialStream*() or *AddHoldup*() functions should be manually saved here. This function will be called when the convergence on the current time interval is reached in order to have a possibility to return to the previous state of the unit if convergence failed during the calculation (in the case of recycled streams in the flowsheet).

void LoadState *()*
Loads last saved state of the unit. All time dependent variables which were previously saved in *SaveState()* function should be manually loaded here.

void Finalize *()*
Finalizes unit. Is called by the simulator only once at the end of the simulation. Here closing and cleaning operations can be performed.

Functions to work with time points

std::vector<double> GetAllInputTimePoints *(double StartTime, double EndTime, bool ForceStartBoundary = false, bool ForceEndBoundary = false)*
Returns all time points on which input streams of the unit are defined for specified time interval. Input streams are all streams, which are connected to the input ports. If *ForceStartBoundary* and/or *ForceEndBoundary* flag is enabled, corresponding boundary points will be forcibly added to the resulting vector.

std::vector<double> GetAllDefinedTimePoints *(double StartTime, double EndTime, bool ForceStartBoundary = false, bool ForceEndBoundary = false)*
Returns all time points for specified time interval on which input streams and unit parameters are defined. Input streams are all streams, which are connected to the input ports. If *ForceStartBoundary* and/or *ForceEndBoundary* flag is enabled, corresponding boundary points will be forcibly added to the resulting vector.

std::vector<double> GetAllStreamsTimePoints *(std::vector<CMaterialStream*> Srteams, double StartTime, double EndTime)*
Returns all time points for specified time interval on which *Srteams* are defined.

Functions to work with unit parameters

unsigned AddConstParameter *(std::string Name, double MinValue, double MaxValue, double InitValue, std::string Description = "")*
Adds new constant unit parameter to the unit with the name *Name*, boundary values *MinValue* and *MaxValue* and description *Description*, and initializes it with the value *InitValue*. The name of the parameter should be unique within the unit. Should be used in unit's constructor only. Returns index of the parameter.

unsigned AddTDParameter *(std::string Name, double MinValue, double MaxValue, double InitValue, std::string Description = "")*
Adds new time-dependent unit parameter to the unit with the name *Name*, boundary values *MinValue* and *MaxValue* and description *Description*, and initializes it in the time point 0s with the value *InitValue*. The name of the parameter should be unique within the unit. Should be used in unit's constructor only. Returns index of the parameter.

unsigned AddStringParameter *(std::string Name, std::string InitValue = "", std::string Description = "")*
Adds new string unit parameter to the unit with the name *Name*, description *Description*, and initializes it with the value *InitValue*. The name of the parameter should be unique within the unit. Should be used in unit's constructor only. Returns index of the parameter.

unsigned AddSolverAggregation *(std::string Name, std::string Description = "")*
Adds new solver parameter of type *SOLVER_AGGREGATION_1* with name *Name* and description *Description*. Allows choosing a specific solver with current type to use it in unit. The name of the parameter should be unique within the unit. Should be used in unit's constructor only. Returns index of the parameter.

unsigned GetParametersNumber *()*
Returns number of unit parameters which have been defined in the unit.

std::string GetParameterName *(unsigned Index)*
Returns name of the unit parameter with the specified index *Index*. Empty string is returned if such parameter has not been defined.

double GetParameterMinVal *(std::string ParameterName)*
Returns minimum allowable value of the time-dependent or constant unit parameter with the name *ParameterName*. If such parameter has not been defined or this is not a constant or time-dependent parameter, *0* will be returned.

double GetParameterMaxVal *(std::string ParameterName)*
Returns maximum allowable value of the time-dependent or constant unit parameter with the name *ParameterName*. If such parameter has not been defined or this is not a constant or time-dependent parameter, *0* will be returned.

double GetConstParameterValue *(std::string ParameterName)*
Returns value of a constant unit parameter with the name *ParameterName*. If such constant parameter has not been defined, *0* will be returned.

double GetTDParameterValue *(std::string ParameterName, double Time)*
Returns value of a time-dependent unit parameter with the name *ParameterName* at the specified time point. If the parameter is not defined at the *Time*, linear interpolation will be used to obtain the value. If such time-dependent parameter has not been defined, *0* will be returned.

std::string GetStringParameterValue *(std::string ParameterName)*
Returns value of a string unit parameter. If such string parameter has not been defined, empty string will be returned.

*CAggregationSolver** GetSolverAggregation *(std::string ParameterName)*
Returns pointer to a chosen solver of *SOLVER_AGGREGATION_1* type. If such unit parameter has not been defined, *nullptr* will be returned.

void SetParameterValue *(std::string ParameterName, double Value, double Time = 0)*
Sets *Value* of a constant or a time dependent unit parameter in the specified time point *Time*. If the time point does not exist, it will be created. If the time point already exists, its value will be overwritten.

void SetParameterValue *(std::string ParameterName, std::string Value)*
Sets new *Value* of a string unit parameter with the specified name *ParameterName*.

std::vector<double> GetParameterTimePoints *(std::string ParameterName, double TimeStart, double TimeEnd)*
Returns all time points for which time-dependent parameter is defined within the specified time interval [*TimeStart*; *TimeEnd*]. If such parameter has not been defined or this is not a time-dependent parameter, empty vector will be returned.

Functions to work with state variables

unsigned AddStateVariable *(std::string Name, double InitValue, bool SaveHistory = false)*
Adds new state variable and initializes it with *InitValue*. *Name* must be unique within the unit's state variables. Parameter *SaveHistory* specifies if the history of all changes of variable should be saved during calculation for further postprocessing. To save history function *SaveStateVariables()* should be called. All variables which are added with this function will be automatically saved and restored during the simulation. Should be used in *Initialize()* function of the unit. Returns index of added variable.

unsigned GetStateVariablesNumber *()*
Returns number of state variables which have been defined in this unit.

std::string GetStateVariableName *(unsigned Index)*
Returns the name of the state variable with specified index. Empty string is returned if such variable has not been defined.

double GetStateVariable *(std::string Name)*
Returns value of internal state variable with name *Name*. *0* will be returned if such variable has not been defined.

void SetStateVariable *(std::string Name, double Value)*
Sets new value *Value* of a state variable *Name*.

void ClearStateVariables *()*
Removes all state variables and history of their changes.

void SaveStateVariables *(double Time)*
Saves values of those internal variables, which were defined as having history. *Time* is a current time.

Functions to work with compounds

std::vector<std::string> GetCompoundsList *()*
Returns unique keys of all compounds defined in the current flowsheet.

std::vector<std::string> GetCompoundsNames *()*
Returns names of all compounds defined in the current flowsheet.

unsigned GetCompoundsNumber *()*
Returns number of compounds which are defined in the current flowsheet.

double GetCompoundConstant *(std::string CompoundKey, unsigned Constant)*
Returns value of constant physical property for specified compound. These properties are stored in the database of materials. Possible constants (Table 1):

- *CRITICAL_PRESSURE*
- *CRITICAL_TEMPERATURE*
- *HEAT_OF_FUSION_AT_NORMAL_FREEZING_POINT*
- *HEAT_OF_VAPORIZATION_AT_NORMAL_BOILING_POINT*
- *MOLAR_MASS*
- *NORMAL_BOILING_POINT*
- *NORMAL_FREEZING_POINT*
- *SOA_AT_NORMAL_CONDITIONS*
- *STANDARD_FORMATION_ENTHALPY*
- *CONST_PROP_USER_DEFINED_XX*

double GetCompoundTPDProp *(std::string CompoundKey, unsigned Property, double Temperature, double Pressure)*
Returns value of temperature/pressure-dependent physical properties (which are stored in the database of materials) for compound with specified *Temperature* [K] and *Pressure* [Pa]. Possible properties (Table 2):

- *HEAT_CAPACITY_CP*
- *ENTHALPY*
- *THERMAL_CONDUCTIVITY*
- *VAPOR_PRESSURE*
- *VISCOSITY*
- *DENSITY*
- *PERMITTIVITY*
- *TP_PROP_USER_DEFINED_XX*

double GetCompoundsInteractionProp *(std::string CompoundKey1, std::string CompoundKey2, unsigned Property, double Temperature, double Pressure)*
Returns the value of the interaction property for selected compounds under the specified *Temperature* [K] and *Pressure* [Pa]. These properties are stored in the database of materials. Possible properties (Table 3):

- *INTERFACE_TENSION*
- *INT_PROP_USER_DEFINED_XX*

bool IsCompoundNameDefined *(std::string CompoundName)*
Returns true if compound with specified name has been defined, otherwise returns false.

bool IsCompoundKeyDefined *(std::string CompoundKey)*
Returns true if compound with specified unique key has been defined, otherwise returns false.

bool IsPropertyDefined *(unsigned Property)*
Returns true if a physical property (constant, temperature/pressure-dependent or interaction) has been defined, otherwise returns false.

Functions to work with phases

unsigned GetPhasesNumber *()*
Returns number of phases which are currently defined in the flowsheet.

std::string GetPhaseName *(unsigned PhaseType)*
Returns name of the specified phase. Possible values of *PhaseType* are: *SOA_SOLID*, *SOA_LIQUID*, *SOA_LIQUID2, SOA_VAPOR*. Empty string will be returned if such phase has not been defined.

unsigned GetPhaseSOA *(unsigned PhaseIndex)*
Returns state of aggregation for the phase with index *PhaseIndex*. If the phase with specified index doesn't exist, *SOA_UNDEFINED* will be returned.

size_t GetPhaseIndex *(unsigned PhaseType)*
Returns index of the specified phase. Returns -1 if such phase has not been defined. Possible values of *PhaseType* are: *SOA_SOLID*, *SOA_LIQUID*, *SOA_LIQUID2, SOA_VAPOR*.

bool IsPhaseDefined *(unsigned PhaseType)*
Returns *true* if such phase has been defined in the flowsheet. Possible values of *PhaseType* are: *SOA_SOLID*, *SOA_LIQUID*, *SOA_LIQUID2, SOA_VAPOR*.

unsigned GetLiquidPhasesNumber *()*
Returns number of defined liquid phases.

Functions to work with solid distributed properties

std::vector<EDistrTypes> GetDistributionsTypes *()*
Returns list of types of solid distributions, which have been defined in the flowsheet. Possible types:

- *DISTR_COMPOUNDS*
- *DISTR_SIZE*
- *DISTR_PART_POROSITY*
- *DISTR_FORM_FACTOR*
- *DISTR_COLOR*

std::vector<unsigned> GetDistributionsClasses *()*
Returns list with number of classes for all defined distributions.

unsigned GetDistributionsNumber *()*
Retunrs number of solids distributions, which have been defined in the flowsheet.

EGridTypes GetDistributionGridType *(EDistrTypes distrType)*
Returns grid's type, which was defined for specified solid distribution *distrType*. Possible values:

- *GRID_NUMERIC*
- *GRID_SYMBOLIC*
- *GRID_UNDEFINED*

std::vector<double> GetNumericGrid *(EDistrTypes distrType)*
Returns grid of classes for specified solid distribution for Numeric grid. If this distribution has Symbolic grid, empty vector will be returned.

std::vector<std::string> GetSymbolicGrid *(EDistrTypes distrType)*
Returns grid of classes for specified solid distribution for Symbolic grid. If this distribution has Numeric grid, empty vector will be returned.

unsigned GetClassesNumber *(EDistrTypes distrType)*

Returns number of classes for specified solid distribution. If such distribution has not been defined, *0* will be returned.

std::vector<double> GetClassesMeans *(EDistrTypes distrType)*
Returns means of classes for specified solid distribution with Numeric grid. If such distribution has not been defined or has Symbolic grid, empty vector will be returned.

std::vector<double> GetPSDGridDiameters *()*
Returns size grid for particle diameters. If *DISTR_SIZE* distribution has not been defined, returns empty vector.

std::vector<double> GetPSDGridVolumes *()*
Returns size grid for particle volumes. If *DISTR_SIZE* distribution has not been defined, returns empty vector.

std::vector<double> GetPSDMeanDiameters *()*
Returns mean particle diameters. If *DISTR_SIZE* distribution has not been defined, returns empty vector.

std::vector<double> GetPSDMeanSurfaces *()*
Returns mean particle surfaces. If *DISTR_SIZE* distribution has not been defined, returns empty vector.

std::vector<double> GetPSDMeanVolumes *()*
Returns mean particle volumes. If *DISTR_SIZE* distribution has not been defined, returns empty vector.

std::vector<double> GetClassesSizes *(EDistrTypes distrType)*
Returns sizes of classes for specified solid distribution with Numeric grid. If such distribution has not been defined or has Symbolic grid, empty vector will be returned.

bool IsDistributionDefined *(EDistrTypes distrType)*
Returns *true* if such solids distribution has been defined in the flowsheet.

void CalculateTM *(EDistrTypes distrType, std::vector<double> InDistr, std::vector<double> OutDistr, CTransformMatrix &outTM)*
Calculates transformation matrix for one-dimensional distribution with type *distrType* according to input and output distributions. Obtained matrix can be applied to the stream instead of direct setting of distribution to retain secondary dimensions in multidimensional distribution. Following algorithm is used to setup transformation matrix:

1. Go through the classes of source and target distributions from left to right.
2. The mostleft notempty class of the initial distribution proceeds to the mostleft notempty class of the output distribution.
3. Transition to the next class of the initial distribution is performed if the current class is completely transferred to the output distribution.
4. Transition to the next class of the output distribution is performed if the current class is already full.

Functions to work with tolerances

double GetAbsTolerance *()*
Returns absolute tolerance, which has been defined for the flowsheet.

double GetRelTolerance *()*
Returns relative tolerance, which has been defined for the flowsheet.

Functions to work with errors and warnings

void RaiseError *(std::string Description = "")*
Can be called to indicate that an error occurred. *Description* will be displayed in the simulation's log and simulation will be stopped after setting.

void RaiseWarning *(std::string Description = "")*
Can be called to indicate warning. *Description* will be displayed in the simulation's log and simulation will **not** be stopped.

void ShowInfo *(std::string Description)*
Can be called to write out messages to the simulation's log screen during the simulation. *Description* will be displayed in the simulation's log.

Functions to work with plots

int AddPlot *(std::string PlotName, std::string XAxisName, std::string YAxisName)*
Adds new 2-dimensional plot with specified name and axis, returns index of the plot. *PlotName* must be unique within the unit's plots. Returns -1 on error.

int AddPlot *(std::string PlotName, std::string XAxisName, std::string YAxisName, std::string ZAxisName)*
Adds new 3-dimensional plot with specified name and axis, returns index of the plot. *PlotName* must be unique within the unit's plots. Returns -1 on error.

int AddCurveOnPlot *(std::string PlotName, std::string CurveName)*
Adds new curve with specified name on the 2-dimensional plot with name *PlotName*. Returns index of the curve within specified plot. Returns -1 on error.

int AddCurveOnPlot *(std::string PlotName, double ZValue)*
Adds new curve with specified z-value on the 2- or 3-dimensional plot with name *PlotName*. Returns index of the curve within specified plot. Returns -1 on error.

void AddPointOnCurve *(std::string PlotName, std::string CurveName, double X, double Y)*
Adds new point on specified curve for 2-dimensional plot.

void AddPointOnCurve *(std::string PlotName, double ZValue, double X, double Y)*
Adds new point on specified curve for 3-dimensional plot

void AddPointOnCurve *(std::string PlotName, std::string CurveName, std::vector<double> X, std::vector<double> Y)*
Adds new points on specified curve for 2-dimensional plot.

void AddPointOnCurve *(std::string PlotName, double ZValue, std::vector<double> X, std::vector<double> Y)*
Adds new points on specified curve for 3-dimensional plot.

Tables of properties

Table 1 – Constant properties for pure compounds

Name	Units	Define
State of aggregation at normal conditions	-	SOA_AT_NORMAL_CONDITIONS
Normal boiling point	K	NORMAL_BOILING_POINT
Normal freezing point	K	NORMAL_FREEZING_POINT
Critical temperature	K	CRITICAL_TEMPERATURE
Critical pressure	Pa	CRITICAL_PRESSURE
Molar mass	kg/mol	MOLAR_MASS
Standard formation enthalpy	J/mol	STANDARD_FORMATION_ENTHALPY
Heat of fusion at normal freezing point	J/mol	HEAT_OF_FUSION_AT_NORMAL_FREEZING_POINT
Heat of vaporization at normal boiling point	J/mol	HEAT_OF_VAPORIZATION_AT_NORMAL_BOILING_POINT
Reactivity type	-	REACTIVITY_TYPE
User defined property	-	CONST_PROP_USER_DEFINED_XX

Table 2 – Temperature-dependent pure compound properties

Name	Units	Define
Density	kg/m^3	DENSITY
Heat capacity Cp	J/(kg·K)	HEAT_CAPACITY
Enthalpy	J/kg	ENTHALPY
Vapor pressure	Pa	VAPOR_PRESSURE
Viscosity	Pa·s	VISCOSITY
Thermal conductivity	W/(m·K)	THERMAL_CONDUCTIVITY
Permittivity	F/m	PERMITTIVITY
User defined property	-	TP_PROP_USER_DEFINED_XX

Table 3 – Interaction properties between two pure compounds

Name	Units	Define
Interface tension	N/m	INTERFACE_TENSION
User defined property	-	INT_PROP_USER_DEFINED_XX