ARTIST's Python interface

Reinhold Schaaf, Argelander-Institut für Astronomie, rschaaf@astro.uni-bonn.de

ARTIST's Python interface allows working with ARTIST's Model Library and the ARTIST version of LIME without ARTIST's GUI. In fact, the GUI uses the Python interface to communicate with the Model Library and LIME.

Both the Model Library and LIME have C++ or C interfaces that are exposed to Python via SWIG. While most users will use these Python interfaces (which are documented here), it is also possible to use the Model Library's C++ interface and LIME's C interface from C++. Since the functions of the Model Library's and LIME's Python interfaces have a direct connection to the functions of their C++ or C interfaces, this documentation can serve as a guide to these interfaces as well. (Note, however, that the SWIG layer between LIME's C and Python interface performs some parameter checks that a C++ user would have to implement himself.)

The general way for using ARTIST from Python is:

1) Select a model from the Model Library
2) Setup the selected model
3) Setup input parameters and images for LIME
4) Run LIME
5) Return to 1), 2), or 3)

A sample script that performs the steps 1) – 4) is given at the end of *2.2 Running LIME.*

In order to use the Model Library and LIME from Python, the related Python modules must be imported by executing the Python commands

```
> import modellib
> import lime
```

For these commands to succeed, the directory with the files `modellib.py` and `lime.py` must be in the `PYTHONPATH`. If a complete ARTIST installation was performed, sourcing the file `artistrc.sh` (or `artistrc.csh`) performs all necessary settings.

1 The Model Library

ARTIST's Model Library allows the user to retrieve several physical quantities like H2 density, gas and dust temperature, velocity ("Results") for certain predefined physical situations relevant in the context of start formation ("Models").

In *1.1 Concepts*, central concepts of the Model Library are explained. *1.2 The Model Library's Python Interface* gives a detailed account of the Model Library's Interface, including reference to all available functions and sample scripts.

1.1 Concepts

The Model Library has some central concepts that the user should understand:

1.1.1 Model

A Model is the implementation of a set of physical situations that are related to each other. This relation can be a common mathematical description; e.g. does the Model "BonnorEbert56" describe hydrostatic isothermal cloud cores according to the formulas derived by Bonnor 1955 and Ebert 1956. Another type of relations can be a common file format; e.g. does the Model "RATRAN" give the possibility to read in physical quantities from RATRAN Input Model files.

Models do have Parameters that select a specific physical solution. E.g. does the Model "BonnorEbert56" have the Parameters "central density" and "temperature". Other Models have different Parameters; the Model "RATRAN" has the parameter "filename" that specifies the Input Model file and hence the physical situation.

Within the Model Library, Models are uniquely identified by an ID, the modelID. The Model Library provides for every

implemented Model a name, a description, and a bibliographic reference.

1.1.2 Parameter

Parameters are necessary to select a specific physical situation from the set defined by a Model. This selection is done by assigning values to all parameters of a Model.

All Parameters of a Model are uniquely identified by an ID, the paramID; however, Parameters of different Models may share a paramID. Hence, any Parameter in the Model Library is uniquely identified by a modelID and a paramID. The Model Library provides for every Parameter a name, a description, and a data type.

Valid data types for Parameters are “double, “int”, “string”, and “enum”. Depending on the data type, the Model Library provides further information:

- “double”:
    - default value
    - unit
- “int”
    - default value
- “string”
    - default value
- “enum”
    - list of the enumeration's values
    - default index
    - unit

1.1.3 Result

Results are the physical quantities provided by Models once all Parameters are set. E.g. does the Model “Bonnor Ebert” have the Results “density” (the H2 density), “temperature” (the gas temperature), and “dust temperature”. Other Models may provide different Results.

Results can either be scalar (e.g. “density”, “temperature”) or vectorial (e.g. “velocity”, “magnetic field”).

All Results in the Model Library are uniquely identified by an ID, the resultID. The Model Library provides for each Result a name, description, unit, and a type (“scalar” or “vector”).

While new Models and Functions may be added to the Model Library with relative ease, the set of implemented Results is fixed, mainly due to performance reasons. The implemented Results are:

resultID	Name	Description	Unit	Type
abundance	Abundance	Molecular abundance		scalar
bmag	B magnetic	Magnetic field strength	T	vector
density	Density	H2 density	m^-3	scalar
doppler	Doppler	Doppler b-parameter	m/s	scalar
tdust	Tdust	Dust temperature	K	scalar
temperature	Temperature	Gas temperature	K	scalar
velocity	Velocity	Velocity	m/s	vector

1.1.4 Function

A Function is the implementation of a set of related mathematical expressions to be used as a Result. Functions can be used as Results either to modify a Model, or to add to a Model a Result that the Model does not provide.

The process of linking a Function to a Result is performed during the setup of a Model.

Functions do have Function Parameters that select a specific mathematical expression.

An example for a function is “scalar power law” that implements the mathematical function “offset + factor * R ** exponent”, where R is the distance to the center of the coordinate system, and “offset”, “factor”, and “exponent” are the Function's Function Parameters.

Special cases of Functions are Functions that read and interpolate tabulated physical data.

Functions are uniquely identified by an ID, the functionID. The Model Library provides for each Function a name, description, and a type (“scalar” or “vector”).

1.1.5 Function Parameter

Function Parameters are necessary to select a specific mathematical expression from the set defined by a Function. This selection is done by assigning values to all Function Parameters of a Function.

All Function Parameters of a Function are uniquely identified by an ID, the functionParamID; however, Function Parameters of different Functions may share a functionParamID. Hence, any Function Parameter is uniquely identified by a functionID and a functionParamID. The Model Library provides for every Function Parameter a name, a description, and a data type.

Valid data types for Function Parameters are “double, “int”, “string”, and “enum”. Depending on the data type, the Model Library provides further information:

- “double”:
    - default value
- “int”
    - default value
- “string”
    - default value
- “enum”
    - list of the enumeration's values
    - default index

1.1.6 Current Model

At any time, at most one Model implemented in the Model Library can be selected as the Current Model. Assigning values to Parameters, linking Functions to Results, or querying Results will always operate on the Current Model.

1.1.7 Enumerations

Both Parameters and Function Parameters may be of data type “enum”, meaning that the parameter is an enumeration. An enumeration is a list of double values where any value can be addressed via a (zero-based) integer index. Enumerations allow to restrict possible values of Parameter to a finite number of discreet values, without the need to work with the values themselves (which can lead to all sorts of problems, e.g. when comparing double values). Instead, the index is used (and thus avoiding the problems, e.g. by comparing integer indices).

The Model Library provides functions to query the values of an enumeration; the Library also provides functions to select one of these values by setting the index, and tfo query the selected value by querying the selected index.

1.2 The Model Library's Python Interface

The Model Library's Python Interface comprises a number of Python functions that can be grouped as follows:

- Functions to explore the content of the Model Library (see *1.2.1*)
- Functions to select a Model and set it up (see *1.2.2*)
- Functions to request Results from the selected Model (see *1.2.3*)

1.2.1 Exploring the Model Library's content

These functions allow retrieval of information about Models, Results, and Functions, including Parameters, and Function Parameters implemented on the Model Library. Note that these functions can not be used to get information about which Model is selected as Current Model or the current values of Parameters etc. See section "..." for functions that can used for this purpose.

A demonstration of the basic functions to explore the content of the Model Library is given in the following script:

```
# exploreModellib.py
#
# Simple script that demonstrates the basic functions to explore
# the content of the Model Library
#
# The script lists all implemented Models, their Paramaters and Results as
# well as all implemented Functions and their Function Parameters

import modellib

print
print "Implemented Models, their Parameters and Results:"
print

for modelID in modellib.getModelIDs():
    print "Model:"
    print "  modelID:        ", modelID
    print "  Name:           ", modellib.getModelName(modelID)
    print "  Description:    ", modellib.getModelDesc(modelID)
    print "  Bib. reference: ", modellib.getModelBibref(modelID)

    print "  Parameters:"
    for paramID in modellib.getParamIDs(modelID):
        print "    paramID:     ", paramID
        print "    Name:        ", modellib.getParamName(modelID, paramID)
        print "    Description: ", modellib.getParamDesc(modelID, paramID)
        print "    Type:        ", modellib.getParamType(modelID, paramID)
        if modellib.getParamType(modelID, paramID) == "double":
            print "    Def. Value:  ",
            print modellib.getParamDefValDouble(modelID, paramID)
            print "    Unit:        ",
            print modellib.getParamUnit(modelID, paramID)
        if modellib.getParamType(modelID, paramID) == "int":
            print "    Def. Value:  ",
            print modellib.getParamDefValInt(modelID, paramID)
        if modellib.getParamType(modelID, paramID) == "string":
            print "    Def. Value:  ",
            print modellib.getParamDefValString(modelID, paramID)
        if modellib.getParamType(modelID, paramID) == "enum":
            print "    Values:      ",
            print modellib.getParamEnumValues(modelID, paramID)
            print "    Def. Value:  ",
            print modellib.getParamEnumDefIndex(modelID, paramID)
            print "    Unit:        ",
            print modellib.getParamUnit(modelID, paramID)
    print "  Results:"
    for resultID in modellib.getModelResultIDs(modelID):
        print "    resultID:    ", resultID
        print "    Name:        ", modellib.getResultName(resultID)
        print "    Description: ", modellib.getResultDesc(resultID)
```

```
        print "    Unit:        ", modellib.getResultUnit(resultID)
        print "    Type:        ", modellib.getResultType(resultID)
        print "    ----"

    print "----"

print
print "Implemented Functions and their Function Parameters:"
print

for functionID in modellib.getFunctionIDs():
    print "Function:"
    print "  functionID:  ", functionID
    print "  Name:        ", modellib.getFunctionName(functionID)
    print "  Description: ", modellib.getFunctionDesc(functionID)

    print "  Parameters:"
    for functionParamID in modellib.getFunctionParamIDs(functionID):
        print "    functionParamID: ",
        print functionParamID
        print "    Name:            ",
        print modellib.getFunctionParamName(functionID, functionParamID)
        print "    Description:     ",
        print modellib.getFunctionParamDesc(functionID, functionParamID)
        print "    Type:            ",
        print modellib.getFunctionParamType(functionID, functionParamID)
        print "    ----"

    print "----"
```

Exploring implemented Models

The following table lists the functions that can be used to retrieve information about implemented Models and their Parameters. All functions throw exceptions when called with invalid arguments (e.g. when calling getParamUnit for a parameter of data type “int”).

getModelIDs()

Description:

The IDs of all Models registered with the Model Library

Arguments:

None

Return value:

List of strings: The modelIDs of all registered Models

isRegisteredModel(modelID)

Description:

Checks if a Model is registered with the Model library

Arguments:

string modelID: The ID of the Model

Return value:

bool: True if Model with modelID is registered, else False

getModelName(modelID)

Description:

Name of a Model

Arguments:

string modelID: The ID of the Model

Return value:

string: The name of the Model

getModelDesc(modelID)

Description:

Description of a Model

Arguments:

string modelID: The ID of the Model

Return value:

string: The description of the Model

getModelBibref(modelID)

Description:

Bibliographic reference of a Model

Arguments:

string modelID: The ID of the Model

Return value:

string: The bibliographic reference of the Model

getParamIDs(modelID)

Description:

The IDs of all Parameters registered for a Model

Arguments:

string modelID: The ID of the Model

Return value:

List of strings: The paramIDs of all registered for the Model

isRegisteredParam(modelID, paramID)

Description:

Checks if a Parameter is registered for a Model

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

bool: True if Parameter with paramID is registered for the Model with modelID, else False

getParamName(modelID, paramID)

Description:

Name of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

string: The name of the Parameter

getParamDesc(modelID, paramID)

Description:

Description of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

string: The description of the Parameter

getParamType(modelID, paramID)

Description:

Datatype of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

string: The datatype of the Parameter. Valid datatypes are “double”, “int”, “string”, and “enum”

getParamUnit(modelID, paramID)

Description:

Unit of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

string: The unit of the Parameter.

getParamDefValDouble(modelID, paramID)
getParamDefValInt(modelID, paramID)
getParamDefValString(modelID, paramID)

Description:

Default value of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

double, int, string: The default value of the Parameter.

getParamEnumValues(modelID, paramID)

Description:

Enumeration values of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

List of doubles: The enumeration values of the Parameter.

getParamEnumDefIndex(modelID, paramID)

Description:

Default enumeration index of a Parameter

Arguments:

string modelID: The ID of the Model

string paramID: The ID of the Parameter

Return value:

int: The default enumeration index of the Parameter.

Exploring implemented Results

The following table lists the functions that can be used to retrieve information about implemented Results:

getResultIDs()

Description:

The IDs of all Results registered with the Model Library

Arguments:

None

Return value:

List of strings: The resultIDs of all registered Results

isRegisteredResult(resultID)

Description:

Checks if a Result is registered with the Model Library

Arguments:

string resultID: The ID of the Result

Return value:

bool: True if Result with resultID is registered, else False

getResultName(resultID)

Description:

Name of a result

Arguments:

string resultID: The ID of the Result

Return value:

string: The name of the Result

getResultDesc(resultID)

Description:

Description of a Result

Arguments:

string resultID: The ID of the Result

Return value:

string: The description of the Result

getResultUnit(resultID)

Description:

Unit of a Result

Arguments:

string resultID: The ID of the Result

Return value:

string: The unit of the Result

getResultType(resultID)

Description:

Type of a Result

Arguments:

string resultID: The ID of the Result

Return value:

string: The type of the Result. Valid types are “scalar” and “vector”

getModelResultIDs(modelID)

Description:

The IDs of all Results registered for a Model

Arguments:

string modelID: The ID of the Model

Return value:

List of strings: The resultIDs of all registered Results for the Model

isResultModel(resultID, modelID)

Description:

Checks if a Result is registered for a Model

Arguments:

string resultID: The ID of the Result

string modelID: The ID of the Model

Return value:

bool: True if Result with resultID is registered for the Model with modelID, else False

Exploring implemented Functions

The following table lists the functions that can be used to retrieve information about implemented Functions and their Function Parameters:

getFunctionIDs()

Description:

The IDs of all Functions registered with the Model Library

Arguments:

None

Return value:

List of strings: The functionIDs of all registered Functions

isRegisteredFunction(functionID)

Description:

Checks if a function is registered with the Model Library

Arguments:

string functionID: The ID of the Function

Return value:

bool: True if a Function with functionID is registered, else False

getFunctionName(functionID)

Description:

Name of a Function

Arguments:

string functionID: The ID of the Function

Return value:

string: The name of the Function

getFunctionDesc(functionID)

Description:

Description of a Function

Arguments:

string functionID: The ID of the Function

Return value:

string: The description of the Function

getFunctionType(functionID)

Description:

Type of a Function

Arguments:

string functionID: The ID of the Function

Return value:

string: The type of the Function. Valid types are “scalar” and “vector”

getFunctionResultIDs(functionID)

Description:

The IDs of all Results to which a Function can be linked

Arguments:

string functionID: The ID of the Function

Return value:

List of strings: The resultIDs of all Results to which the Function can be linked

getResultFunctionIDs(resultID)

Description:

The IDs of all Functions that can be linked to a Result

Arguments:

string resultID: The ID of the Result

Return value:

List of strings: The functionIDs of all Functions that can be linked to the Result

isResultFunction(resultID, functionID)

Description:

Checks if a Function can be linked to a Result

Arguments:

string resultID: The ID of the Result

string functionID: The ID of the Function

Return value:

bool: True if the Function with functionID can be linked to the Result with resultID, else False

getFunctionParamIDs(functionID)

Description:

The IDs of all Function Parameters registered for a Function

Arguments:

string functionID: The ID of the Function

Return value:

List of strings: The functionParamIDs of all registered for the Function

isRegisteredFunctionParam(functionID, functionParamID)

Description:

Checks if a Function Parameter is registered for a Function

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

bool: True if Function Parameter with functionParamID is registered for the Function with functionID, else False

getFunctionParamName(functionID, functionParamID)

Description:

Name of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

string: The name of the Function Parameter

getFunctionParamDesc(functionID, functionParamID)

Description:

Description of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

string: The description of the Function Parameter

getFunctionParamType(functionID, functionParamID)

Description:

Datatype of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

string: The datatype of the Function Parameter. Valid datatypes are “double”, “int”, “string”, and “enum”.

getFunctionParamDefValDouble(functionID, paramID)
getFunctionParamDefValInt(functionID, paramID)
getFunctionParamDefValString(functionID, paramID)

Description:

Default value of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

double, int, string: The default value of the Function Parameter.

getFunctionParamEnumValues(functionID, paramID)

Description:

Enumeration values of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

List of doubles: The enumeration values of the Function Parameter.

getFunctionParamEnumDefIndex(functionID, paramID)

Description:

Default enumeration index of a Function Parameter

Arguments:

string functionID: The ID of the Function

string functionParamID: The ID of the Function Parameter

Return value:

int: The default enumeration index of the Parameter.

1.2.2 Selecting a Current Model and setting it up

The functions described in this section have the purpose to select one of the Model Library's implemented Models as the Current Model, to set the Parameters of it, to link Functions to Results, and to set these Function's Function Parameters. Once set up this way, subsequent requests of Results (see Section “Requesting Results”) will retrieve physical quantities defined by the thus configured combination of Current Model and Functions.

The general way for selecting a Current Model, setting it up, and linking Functions to Results is:

1. Select a Current Model (selectCurrentModel); all Parameters of the selected Model are set to their default values
2. Set its Parameters (setParamDouble, setParamInt, setParamString)
3. Link Functions to Results (optional; setFunction); all Function Parameters of the Function are set to their default values
4. Set FunctionParameters (setFunctionParamDouble, setFunctionParamInt, setFunctionParamString)
5. Finalize configuration (finalizeConfiguration)

Other functions described in this section allow to find out if a Current Model is selected, which Model it is, what the values of Parameters and Function Parameters is, and which Functions are linked to Results.

A basic demonstration of how to select and to set up a Current Model is given in the following script:

```
# setupModel.py
#
# Simple script to demonstrate selecting and setting up of
# the Current Model
#
# The script selects the Current Model, sets its Parameters,
# and links Functions to Results that are not provided by the Model.
# Some Results are requested after finilizing the configuration.

import modellib

AU = 1.49598e11    # AU to

# Select the Current Model:
modellib.setCurrentModel('Shu77')

# Set all parameters:
modellib.setParamDouble( 'time',  1.e5)
modellib.setParamDouble( 'T',     30.)

# Shu77 does not provide the results
# 'abundance', 'bmag', and 'doppler'. Hence these
# Results have to be provided by Functions

# Constant abundance:
modellib.setFunction('abundance', 'scalarConst')
modellib.setFunctionParamDouble('abundance', 'val', 1.e-9)

# Constant doppler (100 m/s):
modellib.setFunction('doppler', 'scalarConst')
modellib.setFunctionParamDouble('doppler', 'val', 100.)

# Zero bmag:
```

```
modellib.setFunction('bmag', 'vectorConstR')
modellib.setFunctionParamDouble('bmag', 'val', 0.)

# Done:
modellib.finalizeConfiguration()

# Request some Results:
x, y, z = 100.*AU, 0., 0
print
print "Some Results at r=100AU:"
print
print "density:     %.2f m^-3" % modellib.density(x, y, z)
print "temperature: %.2f K" % modellib.temperature(x, y, z)
print "abundance:   %.2e" % modellib.abundance(x, y, z)
print "bmag:        (%.2f, %.2f, %.2f) T" % modellib.bmag(x, y, z)
print "velocity:    (%.2f, %.2f, %.2f) m/s" % modellib.velocity(x, y, z)
print
```

Selecting the Current Model and setting its Parameters

setCurrentModel(modelID)

Description:

Select a Current Model; all Parameters of the selected Model are set to their default values

Arguments:

string modelID: The ID of the Model

Return value:

None

unsetCurrentModel(modelID)

Description:

Unset the Current Model

Arguments:

None

Return value:

None

isSetCurrentModel()

Description:

Checks if a Current Model is selected

Arguments:

None

Return value:

bool: Checks if a Current Model is selected

getCurrentModelID()

Description:

The ID of the Current Model

Arguments:

None

Return value:

string: The ID of the Current Model; if no Current Model is selcted, the empty string is returned.

setParamDouble(paramID, value)
setParamInt(paramID, value)
setParamString(paramID, value)

Description:

Set the value of a Parameter of the Current Model

Arguments:

string paramID: The ID of the Parameter

double, int, string value: The value of the Parameter

Return value:

None

setParamEnumIndex(paramID, index)

Description:

Set the enumeration index of a Parameter of data type “enum” of the Current Model

Arguments:

string paramID: The ID of the Parameter

int index: The enumeration index of the Parameter

Return value:

None

getParamDouble(paramID)
getParamInt(paramID)
getParamString(paramID)

Description:

Get the value of a Parameter of the Current Model

Arguments:

string paramID: The ID of the Parameter

Return value:

double, int, string: The value of the Parameter

getParamEnumIndex(paramID)

Description:

Get the enumeration index of a Parameter of data type “enum” of the Current Model

Arguments:

string paramID: The ID of the Parameter

Return value:

int: The enumeration index of the Parameter

Linking Functions to Results and setting Function Parameters

setFunction(resultID, functionID)

Description:

Link a Function to a Result; all Function Parameters of the Function are set to their default values

Arguments:

string resultID: The ID of the Result

string functionID: The ID of the Function

Return value:

None

unsetFunction(resultID)

Description:

Unlink any Function that is linked to a Result

Arguments:

string resultID: The ID of the Result

Return value:

None

getFunctionID(resultID)

Description:

Get the ID of the Function linked to a Result

Arguments:

string resultID: The ID of the Result

Return value:

string: The ID of the Function linked to the Result; if no Function is linked to the Result, the empty string is returned.

setFunctionParamDouble(resultID, functionParamID, value)
setFunctionParamInt(resultID, functionParamID, value)
setFunctionParamString(resultID, functionParamID, value)

Description:

Set the value of a Function Parameter

Arguments:

string resultID: The ID of the Result to which the Function is linked

string functionParamID: The ID of the Function Parameter

double, int, string value: The value of the Function Parameter

Return value:

None

setFunctionParamEnumIndex(resultID, functionParamID, index)

Description:

Set the enumeration index of a Function Parameter of data type “enum”

Arguments:

string resultID: The ID of the Result to which the Function is linked

string functionParamID: The ID of the Function Parameter

int index: The enumeration index of the Function Parameter

Return value:

None

getFunctionParamDouble(resultID, functionParamID)
getFunctionParamInt(resultID, functionParamID)
getFunctionParamString(resultID, functionParamID)

Description:

Get the value of a Function Parameter

Arguments:

string resultID: The ID of the Result to which the Function is linked

string functionParamID: The ID of the Function Parameter

Return value:

double, int, string: The value of the Function Parameter

getFunctionParamEnumIndex(resultID, functionParamID)

Description:

Get the enumeration index of a Function Parameter of data type “enum”

Arguments:

string resultID: The ID of the Result to which the Function is linked

string functionParamID: The ID of the Function Parameter

Return value:

int: The enumeration index of the Function Parameter

setTdustIdentTemp()

Description:

Identifies the Result “tdust” with the Result “temperature”.

Arguments:

None

Return value:

None

unsetTdustIdentTemp()

Description:

Unset the identification of the Result “tdust” with the Result “temperature”.

Arguments:

None

Return value:

None

isTdustIdentTemp()

Description:

Checks if the Result “tdust” is identified with the Result “temperature”.

Arguments:

None

Return value:

True if the Result “tdust” is identified with the Result “temperature”, else False

setTempIdentTdust()

Description:

Identifies the Result “temperature” with the Result “tdust”.

Arguments:

None

Return value:

None

unsetTempIdentTdust()

Description:

Unset the identification of the Result “temperature” with the Result “tdust”.

Arguments:

None

Return value:

None

isTempIdentTdust()

Description:

Checks if the Result “temperature” is identified with the Result “tdust”.

Arguments:

None

Return value:

True if the Result “temperature” is identified with the Result “tdust”, else False

isCurrentResultModel(resultID)

Description:

Checks if a Result is provided by the Current Model

Arguments:

string resultID: The ID of the Result

Return value:

bool: True if the Result is provided by the Current Model, else False

isCurrentResultFunction(resultID)

Description:

Checks if a Result is linked to a Function

Arguments:

string resultID: The ID of the Result

Return value:

bool: True if the Result is linked to a Function, else False

isConfigurationComplete()

Description:

Checks if all Results are either provided by the Current Model or linked to a Function

Arguments:

None

Return value:

True if the configuration is complete, else False.

Finalizing the Configuration

Finalizing the configuration is the last step in the configuration of the Model Library. Depending on the Current Model and Functions linked to Results, internal variables of the Current Model are set, data from files may be read in, etc.

finalizeConfiguration()

Description:

Finalize the configuration

Arguments:

None

Return value:

None

isFinalizedConfiguration()

Description:

Checks if the configuration has been finalized

Arguments:

None

Return value:

bool: True if configuration has been finalized, else False

1.2.3 Requesting Results

Once the configuration of the Model Library is finished (finalizeConfiguration was called), physical quantities can be queried with the functions described in this section.

While the functions of the Model Library's Python interface were designed to give maximum flexibility and ease when new Models and Functions are added, the functions described here were designed for maximum runtime performance. One consequence of this approach is that the relation between Results and the functions that request Results is fixed: for all Models, the Result “density” can only be accessed through the function “density(x, y, z)”; this is internally hard-coded.

Another consequence of the design for maximum performance is that the Model Library performs no runtime checks when Results are requested: Requesting a Result that is neither implemented by the Current Model nor linked to a function may result in segmentation faults; requesting Results before running finalizeConfiguration will give undefined values.

For all functions in this section, the arguments x, y, z are the cartesian coordinates of the point where the physical quantities are requested. Note that all quantities (coordinates and results) are in SI units!

abundance(x, y, z)

Description:

Molecular abundance

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

double: Molecular abundance (no unit)

bmag(x, y, z)

Description:

Cartesian components of the magnetic field strength (unit: T)

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

List of 3 doubles: Cartesian components of the magnetic field strength (unit: T)

density(x, y, z)

Description:

H2 density

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

double: H2 density (unit: 1/m^3)

doppler(x, y, z)

Description:

1/e half-width of line profile (Doppler b-parameter) (unit: m/s)

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

double: Doppler b-parameter (unit: m/s)

tdust(x, y, z)

Description:

Dust temperature (unit: K)

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

double: Dust temperature (unit: K)

temperature(x, y, z)

Description:

Gas temperature (unit: K)

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

double: Gas temperature (unit: K)

velocity(x, y, z)

Description:

Cartesian components of the velocity (unit: m/s)

Arguments:

double x, y, z: Cartesian coordinates (unit: m)

Return value:

List of 3 doubles: Cartesian components of the velocity (unit m/s)

2 LIME's Python Interfaces

ARTIST's Python interface to LIME allows the user to specify LIME's input parameters and images, to run LIME, and to query LIME's status while it is running. The classes and functions of this interface are described here.

2.1 Setting LIME's input parameters and images

LIME's Python interface provides two classes to specify input parameters for LIME: Class LimeInputPars contains all basic settings such as number of grid points, model radius, input and output filenames, etc. Class LimeImage contains information about a single image that is one result of a LIME run. Both classes do not provide any methods.

The Python interface provides also the functions createInputPars() and createImage() that create objects of the classes with appropriate default values set.

Class LimeInputPars

This class contains all basic settings such as number of grid points, model radius, input and output filenames, etc. Some of these parameters always need to be set by the user, while others are optional. The function createInputPars() will set

the optional parameters to the given default values.

Data members:

double radius (required)
This value sets the outer radius of the computational domain. It should be set large enough to cover the entire spatial extend of the model. In particular, if a cylindrical input model is used (e.g., the input file for the RATRAN code) one should not use the radius of the cylinder but rather the distance from the center to the corner of the (r,z)-plane.

double minScale (required)
minScale is the smallest scales sampled by the code. Structures smaller than minScale will not be sampled properly. If one uses spherical sampling (see below) this number can also be though upon as the inner edge of the grid. This number should not be set smaller than needed, because an undesirable large number of grid point will end up near the center of the model.

integer pIntensity (required)
This number is the number of model grid points. The more grid points that are used, the longer the code will take to run. Too few points however, will cause the model to be under-sampled with the risk of getting wrong results. Useful numbers are between a few thousands up to about one hundred thousand.

integer sinkPoints (required)
The sinkPoints are grid points that are distributed randomly at radius forming the surface of the model. As a photon from within the model reaches a sink point it is said to escape and is not traced any longer. The number of sink points is a user defined quantity since the exact number may affect the resulting image as well as the running time of the code. One should choose a number that gives a surface density large enough not to cause artifacts in the image and low enough not to slow down the gridding too much. Since this is model dependent, a global best value cannot be given, but a useful range is between a few thousands and about ten thousand.

integer sampling (optional)
The sampling parameter takes value 0 or 1. sampling=0 is used for uniform sampling in Log(radius) which is useful for models with a central condensation (i.e., envelopes, disks), whereas sampling=1 is uniform sampling in x, y, and z. The latter is useful for models with no central condensation (molecular clouds, galaxies, slab geometries). The default value is sampling=0.

double tcmb (optional)
This parameter is the temperature of the cosmic microwave background. This parameter defaults to 2.725K which is the value at zero redshift (i.e., the solar neighborhood). One should make sure to set this parameter properly when calculating models at a redshift larger than zero: TCMB = 2.725(1+z) K. It should be noted that even though LIME can now take the change in CMB temperature with increasing z into account, it does not (yet) take cosmological effects into account when ray-tracing (such as stretching of the frequencies when using Jansky as unit). This is currently under development.

string moldatfile (optional)
Path to the molecular data file. Molecular data files contains the energy states, Einstein coefficients, and collisional rates which are needed by LIME to solve the excitation. These files need to conform to the standard of the LAMDA database (http://www.strw.leidenuniv.nl/~moldata). LIME comes only with a single datafile, HCO+, but additional data files can be downloaded from the LAMDA database. There is no default value.

string dust (optional)
Path to a dust opacity table. This table should be a two column ascii file with wavelength in the first column and opacity in the second column. Currently LIME uses the same tables as RATRAN from Ossenkopf and Henning (1994), and so the wavelength should be given in microns (1e-6 meters) and the opacity in cm2/g. This is the only place in LIME where SI units are not used. The moldatfile and dust parameters are optional in the sense that at least one of them (or both) should be set. There is no default value.

string outputfile (optional)
This is the file name of the output file that contains the level populations. If this parameter is not set, LIME will not output the populations. There is no default value.

string gridfile (optional)
This is the file name of the output file that contains the grid. If this parameter is not set, LIME will not output the grid. The grid file is written out as a VTK file. This is a formatted ascii file that can be read with a number of 3D visualizing tools (Visualization Tool Kit, Paraview, and others). There is no default value.

string pregrid (optional)
A file containing an ascii table with predefined grid point positions. If this option is used, LIME will not generate its own grid, but rather use the grid defined in this file. The file needs to contain all physical properties of the grid points, i.e., density, temperature, abundance, velocity etc.

integer lte_only (optional)
If set, LIME performs an LTE calculation. Useful for quick checks. The default lte_only=0, i.e., full non-LTE calculation.

integer blend (optional)
If set, LIME takes line blending into account, however, only if there are any overlapping lines among the transitions found in the moldatfile. LIME will print a message on screen if it finds overlapping lines. Switching line blending on will slow the code down considerably, in particular if there is more than one molecular data file. The default is blend=0 (no line blending).

integer antialias (optional)
If set, LIME will anti-alias the output image. antialias can take the value of any positive integer, with the value 1 (default) being no anti-aliasing. Greater values correspond to stronger anti-aliasing. LIME uses stochastic super-sampling anti-aliasing. This is very effective in minimizing artifacts in the image, but it also slows down the ray-tracer by a factor equal to the value of antialias.

Class LimeImage

This class contains information about one image created by LIME as output. Some of the parameters have always to be set, some are optional (in which case the function createImage() will set the indicated default values), and some parameters are labeled “semi optional”. The parameters listed as semi optional determine what kind of image is produced (line or continuum). Only certain combinations are permitted, however some of them have to be set. LIME decides to make a continuum image if the parameter nchan is left unset. This will result in a single channel continuum image. In nchan is unset, the only other parameter which is allowed to be set is freq, which, however, has to be set. In addition, the parameter par->dust needs to be set as well, otherwise, LIME will produce an error. In order to produce a line image cube, either the parameter nchan, velres, and trans *or* nchan, freq, and bandwidth should be set. Any other combination will produce an error. For line images, at least one moldatfile should be provided and optionally a dust opacity table as well.

Data members:

integer pxls (required)
This is the number of pixels per spatial dimensions of the FITS file. The total amount of pixels in the image is thus the square of this number.

double imgres (required)
The image resolution or size of each pixel. This number is given in arc seconds. The image field of view is therefore pxls * imgres.

double theta (required)
Theta is the viewing angle (the angle between the model z axis and the ray-tracers line of sight). This number is given in radians, *not* degrees, so that a face-on view (of models where this term is applicable) is 0 and edge-on view is $\pi/2$.

double distance (required)
The source distance in meters. LIME knows the conversion between parsec, AU, and meters, so the distance can be given as 100*pc, for example. If the source is located at a cosmological distance, this parameter is the luminosity distance.

integer unit (required)

The unit of the image. This variable can take values between 0 and 4. 0 for Kelvin, 1 for Jansky per pixel, 2 for SI units, and 3 for Solar luminosity per pixel. The value 4 is a special option that will create an optical depth image cube (dimensionless).

string filename (required)
This variable is the name of the FITS file. It has no default value.

double phi (optional)
Phi is an optional geometric parameter. Like theta, it should be given in radians between 0 and 2π. Phi is the rotation angle of the model (x,y)-plane around the z-axis. If the model is view face-on (so that the line of sight coincides with the z-axis), phi corresponds to the position angle on the sky. The default value is 0.

double source_vel (optional)
The source velocity is an optional parameter that gives the spectra a velocity offset. This parameter is useful when comparing the model to an astronomical source with a known relative velocity.

integer nchan (semi optional)
nchan is the number of velocity channels in a spectral image cube. See the note below for additional information.

double velres (semi optional)
The velocity resolution of the spectral dimension of the FITS file (the width of a velocity channel). This number is given in m/s.

integer trans (semi optional)
The transition number when ray-tracing line images. This number refers to the transition number in the molecular data files. Contrary to the numbers in the data files, trans is zero-index, meaning that the first transition is labeled 0, the second transition 1, and so on. For linear rotor molecules without fine structure transition in their data files (CO, CS, HCN, etc.) the trans parameter is identified by the lower level of the transition. For example, for CO J=1-0 the trans label would be zero and for CO J=6-5 the trans label would be 5. For molecules with a complex level configuration (e.g., H2O), the user needs to refer to the datafile to find the correct label for a given transition. See the note below for additional information.

double freq (semi optional)
Center frequency of the spectral axis in Hz. This parameter can be used for both line and continuum images.

double bandwidth (semi optional)
Width of the spectral axis in Hz.

Function createInputPars()

Description:

Creates a LimeInputPars object with default values as given above

Arguments:

None

Return value:

LimeInputPars object with default values as given above

Function createImage()

Description:

Creates a LimeImage object with default values as given above

Arguments:

None

Return value:

LimeImage object with default values as given above

2.2 Running LIME

The function runLime() performs a LIME run. runLime takes as arguments a LimeInputPars object and a list of LimeImage objects that describe all necessary parameters for LIME. In the run, LIME will query the Model Library for physical quantities which are returned by the Model Library as the Results of the Current Model.

Before actually calling LIME, runLime performs a number of basic checks on the input parameters and images (e.g. required input files do nor exist, parameters have obviously invalid data, definition of images not consistent with either line or continuum, no images defined) and throws an exception if inconsistent values are met. RunLime does also check whether the the configuration of the Model Library's Current Model is complete, and throws an exception if this is not the case.

During the run, LIME will update an object of Class ProgStatus that can be accessed from Python via the function getStatus(). Note however that runLime() returns only after the LIME run is finished; hence, getStatus must be called from a different thread than the one in which runLime was called.

Function runLime(par, images, debug=False)

Description:

Performs basic tests on the passed input parameters and image definitions and runs LIME. During the run, LIME uses the Current Model of the Model Library to retrieve physical quantities.

If the optional argument debug is True, the values of the arguments actually passed to LIME will be printed to stdout before LIME is actually called.

Arguments:

LimeInputPars par: Basic settings for the LIME run

list of LimeImage images: Definitions for images for the LIME run

bool debug: If True, print values of arguments passed to LIME before actually calling LIME

Return value:

None

Function setSilent(int silent)

Description:

Enables or disables feedback about the status of the LIME run via a ProgStatus object. If silent is 0, the ProgStatus object is never updated, otherwise it will contain information about LIME's status.

Arguments:

int silent: If 0, do not update the ProgStatus object. Otherwise, update it.

Return value:

None

Function getSilent()

Description:

Returns whether the ProgStatus object is updated regularly or not.

Arguments:

None

Return value:

int: 0 if ProgStatus object is not updated; any other value means that it is updated.

Class ProgStatus

This class contains information about the status of the LIME run. It can be accessed from Python via the function getStatus(), which must be called from a different thread than the one in which runLime() was called.

Data members:

float progressGridBuilding
Progress of LIME's grid building (0 to 1)

float progressGridSmoothing
Progress of LIME's grid smoothing (0 to 1)

integer statusGrid
1 if LIME's grid is complete, 0 else

float progressConvergence
Progress of LIME's convergence (0 to 1)

integer numberIterations

double minsnr

double median

float progressPhotonPropagation
Progress of LIME's photon propagation (0 to 1)

float progressRayTracing
Progress of LIME's ray tracing (0 to 1)

integer statusRayTracing
1 if LIME's ray tracing is complete, 0 else

integer statusGlobal
1 if LIME's run is complete, 0 else

integer error
1 if an internal LIME error occured, 0 else. See messaeg for an error message

string message
Any message issued by LIME.

Function getStatus()

Description:

Returns a ProgStatus object with information about the current status of the LIME run. Note that getStatus must be called from a different thread than runLime was called, since runLime returns only after the LIME run was finished.

Arguments:

None

Return value:

ProgStatus: The current status of the LIME run.

The following script gives a demonstration of a complete course of activities to run LIME from Python. For a multi-threaded version of the script that displays the status of the LIME run on the screen, see the script runLimeThreads.py in the directory demo.

```
# runLime.py
```

```
#
# Script to demonstrate the complete course of selecting and
# setting up the Current Model, defining input parameters and
# images for LIME, and running LIME.
#
# The script selects the Current Model, sets its Parameters,
# and links Functions to Results that are not provided by the Model.
# After the Model Library is set up thus, LIME's input parameters
# and the parameters for two images are defined. Then LIME is run.
#
# Note that in this simple script no status of LIME's progress can be
# accessed.

import modellib
import lime

import time
import os
import math

artistdir = os.getenv("ARTIST_ROOT", "..")

AU = 1.49598e11    # AU to m
PC = 3.08568025e16 # PC to m

t0 = time.time()

# Select the Current Model:
modellib.setCurrentModel('CG97')

# Set all parameters:
modellib.setParamDouble('Mstar', 0.5)    # Msun
modellib.setParamDouble('Rstar', 2.0)    # Rsun
modellib.setParamDouble('Tstar', 4000.)  # K
modellib.setParamDouble('bgdens', 1.e-4) # 1/cm^3
modellib.setParamDouble('hph', 4.0)
modellib.setParamDouble('plsig1', -1.0)
modellib.setParamDouble('rin', 1.0)      # AU
modellib.setParamDouble('rout', 100.0)   # AU
modellib.setParamDouble('sig0', 0.01)    # g/cm^2

# Model CG97 does not provide the results 'abundance',
# 'doppler', 'temperature, and 'bmag'. Hence these
# Results have to be provided by Functions

# Constant abundance (1.e-9):
modellib.setFunction('abundance', 'scalarConst')
modellib.setFunctionParamDouble('abundance', 'val', 1.e-9)

# Constant doppler (100 m/s):
modellib.setFunction('doppler', 'scalarConst')
modellib.setFunctionParamDouble('doppler', 'val', 100.)

# Zero bmag:
modellib.setFunction('bmag', 'vectorConstR')
modellib.setFunctionParamDouble('bmag', 'val', 0.)

# Set temperature identical to t_dust:
modellib.setTempIdentTdust()

# Done setting up Model Library:
```

```
modellib.finalizeConfiguration()

# Set input parameters for lime:
par = lime.createInputPars()

par.radius      = 800 * AU
par.minScale    = 0.5 * AU
par.pIntensity  = 1000
par.sinkPoints  = 1000

par.dust        = os.path.join(artistdir, "data/jena_thin_e6.tab")
par.moldatfile  = os.path.join(artistdir, "data/hco+.dat")
par.outputfile  = "outfile.pop"
par.gridfile    = "grid.vtk"

# Define the images:
images = []

# Image #0: J=2-1
img = lime.createImage()

img.trans       = 2
img.filename    = "image2.fits"

img.pxls        = 101
img.imgres      = 0.02
img.distance    = 140 * PC
img.theta       = math.pi * 30. / 180.
img.phi         = 0.0
img.source_vel  = 0.0
img.unit        = 0

img.nchan       = 60
img.velres      = 100.

images.append(img)

# Image #1: J=3-2
img = lime.createImage()

img.trans       = 3
img.filename    = "image3.fits"

img.pxls        = 101
img.imgres      = 0.02
img.distance    = 140 * PC
img.theta       = math.pi * 30. / 180.
img.phi         = 0.0
img.source_vel  = 0.0
img.unit        = 0

img.nchan       = 60
img.velres      = 100.

images.append(img)


# Run LIME:
lime.setSilent(False)

print "Running LIME:"
```

```
try:
  lime.runLime(par, images)
except Exception, e:
  print "Exception:", e

t1 = time.time()
print "Runtime: %ds" % (t1 - t0)
```