PyGelsA Graphical User Interface User's Manual

(User's Manual)

Diffusion : see last page

Qualité	For the writers	For the checkers	Approver
Function	Head of interface	Head of quality	Project head
Name	M. Lazareff	A.-M. Vuillot	L. Cambier
Visa			

Informatic management : GCL ELSA

Applicability date : Immédiate

HISTORIC

VERSION EDITION	DATE	CAUSE and/or NATURE of EVOLUTION
1.0	21st May 2002	Extracted from MU-98057 (User's Manual)

CONTENTS

1. INTRODUCTION

The *elsA* (*e*nsemble *l*ogiciel de *s*imulation en *A*érodynamique) Computational Fluid Dynamics software package, in its present version, is dedicated to numerical simulation of single-species viscous compressible flows, on three-dimensional (or two-dimensional, or axisymmetric) block-structured grids.

This User's Manual covers the PyG*elsA* graphical front-end of the `elsa` software described in the `/ELSA/MU-98057` document.

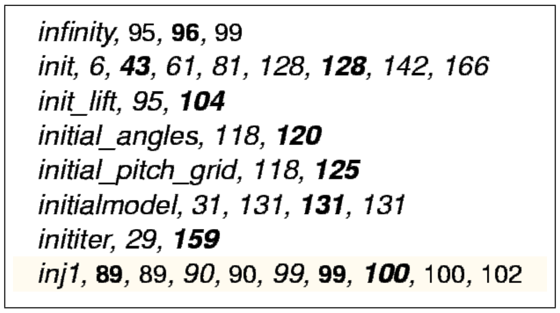
Installation of, and required environment variables for execution of, PyG*elsA* are described in appendix I.

It is highly recommended that this Manual be read while "playing" with the PyG*elsA* software, to get a better feel of its dynamic behaviour.

Index, specific typography

Usage of the index, *p.* 35–36, is highly recommended to navigate this manual. Please refer to the above-mentioned `/ELSA/MU-98057` document for the explanation of the various terms(*attribute* . . .). Please take note that the same name may refer both to an *attribute* and an *attribute value*.

In this index, page numbers are displayed with a specific font :



infinity, 95, **96**, 99
init, *6*, ***43***, *61*, *81*, *128*, ***128***, *142*, *166*
init_lift, *95*, ***104***
initial_angles, *118*, ***120***
initial_pitch_grid, *118*, ***125***
initialmodel, *31*, *131*, ***131***, *131*
inititer, *29*, ***159***
inj1, **89**, 89, *90*, 90, *99*, **99**, ***100***, 100, 102

– small upright font for attribute values ;
– slanted font for other elements (functions, classes, methods, attributes . . .) ;
– bold face for definitions.

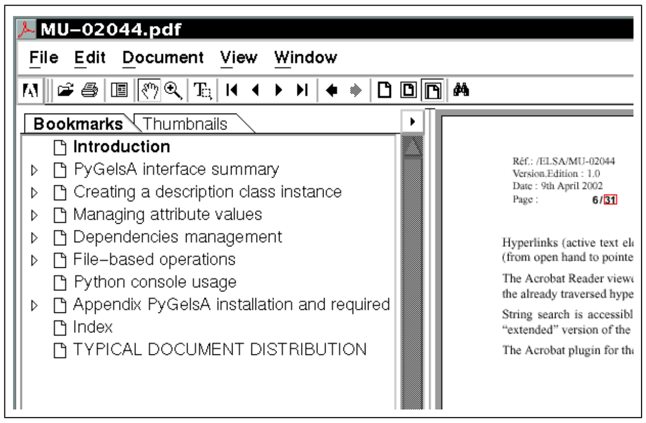
which makes up a total of four combinations, ex. :

89, 89 : definition of, and reference to, `type='inj1'` ;
100, *99* : definition of, and reference to, the `inj1` attribute.

PDF version of this document

The PDF [1] version of this document (`MU-02044.pdf`), which can be visualised on most platforms using the Acrobat Reader software [2], may be interactively navigated using four methods :

[1]"Portable Document Format" of Adobe Inc..
[2]Freely downlowdable, `http://www.adobe.com`



Introduction
PyGelsA interface summary
Creating a description class instance
Managing attribute values
Dependencies management
File-based operations
Python console usage
Appendix PyGelsA installation and required
Index
TYPICAL DOCUMENT DISTRIBUTION

Réf.: /ELSA/MU-02044
Version.Edition : 1.0
Date : 9th April 2002
Page : 6 / 31

Hyperlinks (active text el
(from open hand to pointe
The Acrobat Reader view
the already traversed hype
String search is accessibl
"extended" version of the
The Acrobat plugin for th

1. the "bookmarks" pane of the viewer (always visible) ;
2. the (hyperlinked) contents table, at the beginning of the document ;
3. in-text hyperlinks ;
4. the (hyperlinked) index, at the end of the document.

Hyperlinks (active text elements) are highlighted by a red border and the change of appearance, from open hand (or looking glass) to pointed index finger, of the mouse pointer.

The Acrobat Reader viewer, as any Web browser, allows to move backwards and forward through the already traversed hyperlinks, using the (right-mouse) contextual menu.

String search is accessible through the "field glasses" icon on the tool panel (this requires an "extended" version of the Acrobat Reader viewer, which may not be available on all platforms).

Browsing this manual online (i.e. screen viewing) is especially recommended as many colour illustrations are easier to read this way.

Remark : The Acrobat plug-in for the Netscape browser may also be used, but is not recommended.

2. PYG*ELSA* INTERFACE SUMMARY

2.1. PyG*elsA* aim and focus

The PyG*elsA* GUI aims to provide better comfort and security for the man-machine *elsA* interface, while preserving the efficiency of the plain-text mode.

It has been built as a graphical wrapper for the Python–*elsA* plain-text mode, most functionality of which is encapsulated in graphical metaphors (buttons, pop-down menus, click-able lists . . .).

It is focused on compressing the learning timescale for new users, or for seasoned users who need to master new features of the *elsA* software.

The main conceptual difference, relative to the pure text-based `elsa` interface, is in this light not in the new possibilities in user data entry (through pop-down lists . . .), although they are most useful, but in the greater information content of the graphical software feedback. The larger volume of information which can be simultaneously presented to the user allows for *a priori* checking, see 5.2, as opposed to user-requested *a posteriori* checking, see 5.3.

2.2. PyG*elsA* structure

PyG*elsA*'s main outline is :

– an upper part with the usual elements of a graphical interface ;
– a lower part with the Python console for Python–*elsA* commands.

All interface elements are interconnected, meaning that graphical commands show up in text mode in the console, and text mode commands are reflected in GUI changes.

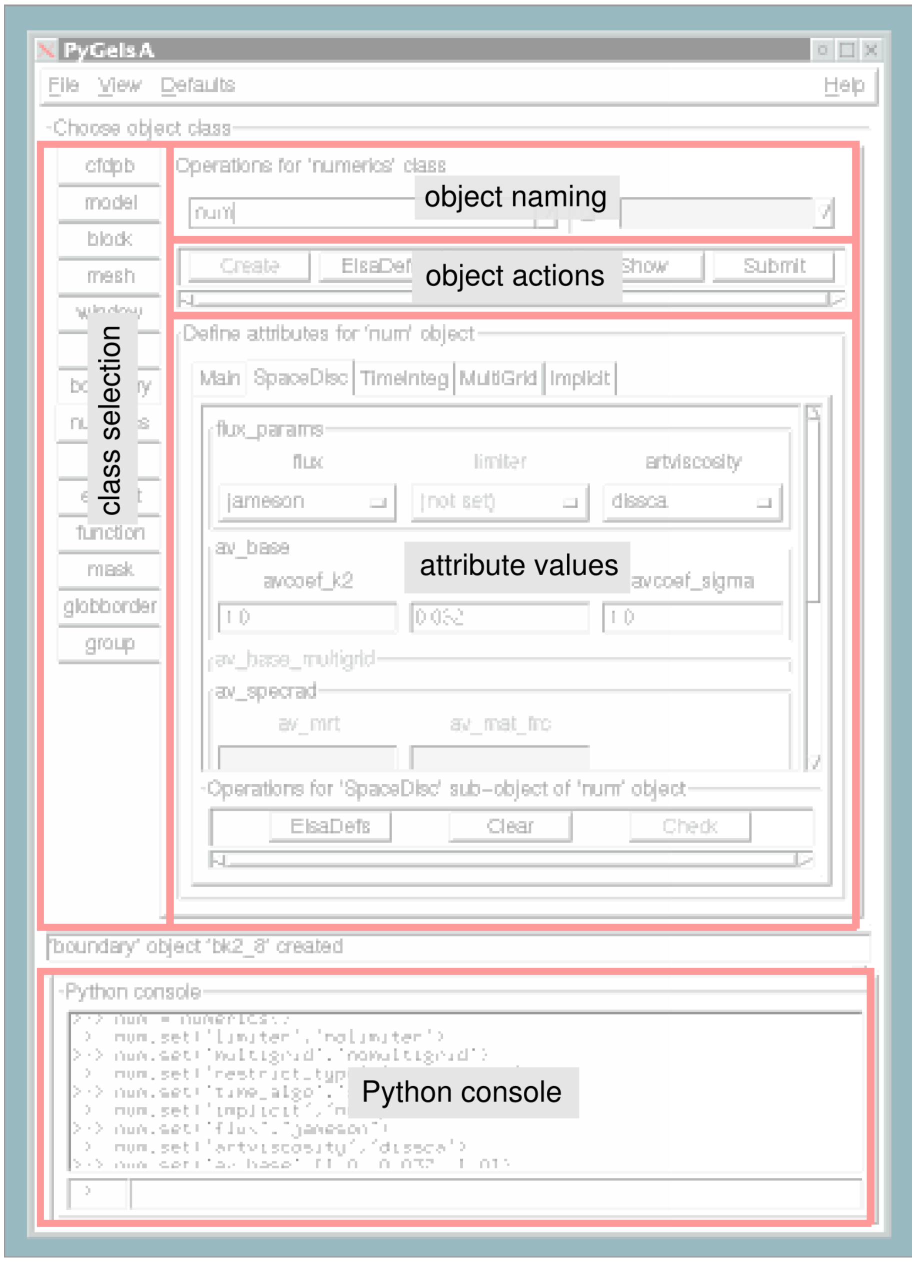
The "widget" word will be used hereafter for all the graphical elements participating in human-computer interaction (HCI) (buttons, selectors, labels, text fields . . .). The main widget types are defined in appendix II.

As a general rule, the scope of action of a button for a function or method call (ex. `Clear` for the `clear()` call) is that of the enclosing frame, so that `Clear` may mean :

– clear the object, in the scope of an object, that is, un-define all attribute values ;
– clear the script, in the scope of the script (try `View/Toggle global actions` to make the global `clear` button appear), that is, delete all description objects.

2.2.1. Main interface regions

This screen-shot outlines the main regions of the interface, which will be individually described below.

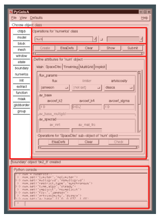
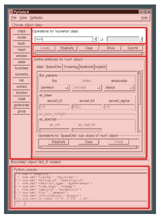
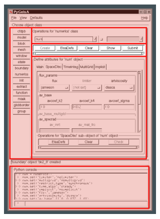
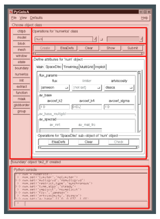
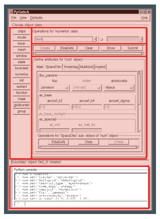


'boundary' object 'bk2_8' created

```
>-> num = numerics()
 >  num.set('limiter','nolimiter')
>-> num.set('multigrid','nomultigrid')
 >  num.set('restrict_typ
>-> num.set('time_algo','
 >  num.set('implicit','n
>-> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>-> num.set('av_base',[1.0, 0.032, 1.0])
```

```
 >
```

The sizes of the corresponding images (seen below as icons) for individual regions are purposely small, the information to be conveyed being only their relative positions, and not their content (see *p. 26* for a plain full view of the interface).

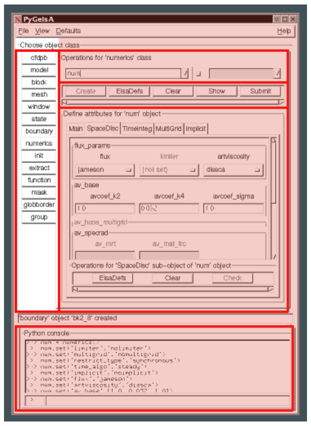






In each case, the region of interest is displayed with a light background, while the remainder of the interface is overlayed by a (red) semi-transparent veil.

The tool bar, holding the `File` ... menus, and the status bar, just above the Python console, are not detailed.

2.2.2. *Classes tabnotebook widget*

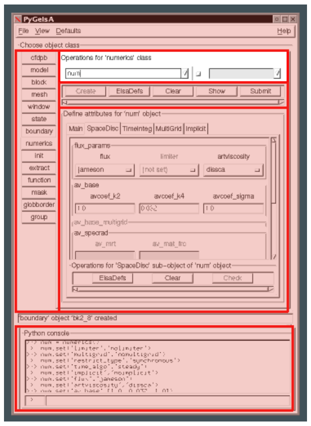
The upper part's structure is a mirror of the description classes (and sub-classes) of the Python–*elsA* interface, displayed as tabnotebook widgets (overlayed notebook-like "pages" with protruding selection tabs, one of which only is active at any time).


PyGelsA
File View Defaults Help
Choose object class
cfdpb
model
block
mesh
window
state
boundary
numerics
init
extract
function
mask
globborder
group
Operations for 'numerics' class
num
Create ElsaDefs Clear Show Submit
Define attributes for 'num' object
Main SpaceDisc TimeInteg MultiGrid Implicit
flux_params
flux limiter artviscosity
jameson [not set] dissca
av_base
avcoef_k2 avcoef_k4 avcoef_sigma
1.0 0.032 1.0
av_base_multigrid
av_specrad
av_mrt av_mat_frc
Operations for 'SpaceDisc' sub-object of 'num' object
ElsaDefs Clear Check
'boundary' object 'bk2_8' created
Python console

```
>-> num = numerics()
 >  num.set('limiter','nolimiter')
>-> num.set('multigrid','nomultigrid')
 >  num.set('restrict_type','synchronous')
>-> num.set('time_algo','steady')
 >  num.set('implicit','noimplicit')
>-> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>-> num.set('av_base',[1.0, 0.032, 1.0])
```

The "main" tabnotebook widget, in vertical position on the left side, displays the various classes (`cfdpb` ...) and is always visible.

2.2.3. *Class instance management*


PyGelsA
File View Defaults Help
Choose object class
cfdpb
model
block
mesh
window
state
boundary
numerics
init
extract
function
mask
globborder
group
Operations for 'numerics' class
num
Create ElsaDefs Clear Show Submit
Define attributes for 'num' object
Main SpaceDisc TimeInteg MultiGrid Implicit
flux_params
flux limiter artviscosity
jameson [not set] dissca
av_base
avcoef_k2 avcoef_k4 avcoef_sigma
1.0 0.032 1.0
av_base_multigrid
av_specrad
av_mrt av_mat_frc
Operations for 'SpaceDisc' sub-object of 'num' object
ElsaDefs Clear Check
'boundary' object 'bk2_8' created
Python console

```
>-> num = numerics()
 >  num.set('limiter','nolimiter')
>-> num.set('multigrid','nomultigrid')
 >  num.set('restrict_type','synchronous')
>-> num.set('time_algo','steady')
 >  num.set('implicit','noimplicit')
>-> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>-> num.set('av_base',[1.0, 0.032, 1.0])
```

Object name choice

The `Operations for xxx class` frame contains an entry field for the name of an object of the current `xxx` class (active tab of the classes tabnotebook widget), and buttons for creation and other operations on this instance.

The object name may be a new one (for object creation) or the name of an existing object. A new name must be typed in the name entry field, while an existing name may be typed or selected from the pop-down list, displayed when the arrow at the right of the entry field is clicked.

In all cases, the chosen name must be validated by a press on the `Return` key ; the validated name then appears overstriked in colour, see 3.2.

Name-based objects filtering

On the right of the instance name field, and separated from it by a toggle button, is the name filter field. When the toggle button is in the pressed state, the name filter field may define a regular expression (in the sense of the `re` Python module), defining a display filter for the names in the pop-down list under the instance name field.

This may be useful for faster lookup of objects ; ex. the `bnd.*a` regular expression will select description objects whose name begins with `bnd` and ends with `a`.

This is a very powerful mechanism if the objects naming pattern is chosen adequately, although it cannot fully replace a continuously kept definition of related configuration fragments, ex. taken from the CAD description.

Group creation

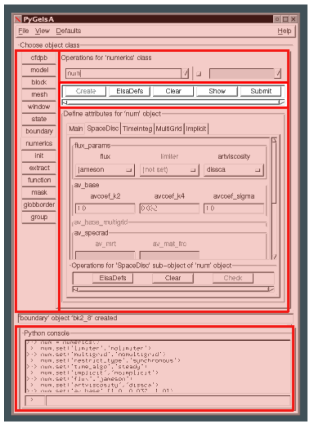
Direct creation of a group instance (ex. from the "filtered" names list, see above) is not yet implemented. In the current version, this must be done in text mode, using the Python console :

```
elsA >>> wnd_g = group(window, 'wnd_[2-4]+')
```

will create a new instance of the group class, with window members, selecting only those with names defined by the `wnd_[2-4]+` regular expression. That is, names beginning with `wnd_`, followed by at least one repetition of one of `2|3|4` (as specified by the + element at the end of the regular expression). Possible names (if they exist) are `bnd_4`, `bnd_32` and `bnd_222333444`.

2.2.4. *Object action buttons*

Under the object name selection entry field, a row of buttons (with associated scrollbar if needed for many buttons) defines the main global actions available for the object.


PyGelsA
File View Defaults Help
Choose object class
cfdpb
model
block
mesh
window
state
boundary
numerics
init
extract
function
mask
globborder
group
Operations for 'numerics' class
Create ElsaDefs Clear Show Submit
Define attributes for 'num' object
Main SpaceDisc TimeInteg MultiGrid Implicit
flux_params
flux limiter artviscosity
av_base
avcoef_k2 avcoef_k4 avcoef_sigma
av_base_multigrid
av_specrad
av_mrt av_mat_frc
Operations for 'SpaceDisc' sub-object of 'num' object
ElsaDefs Clear Check
Python console

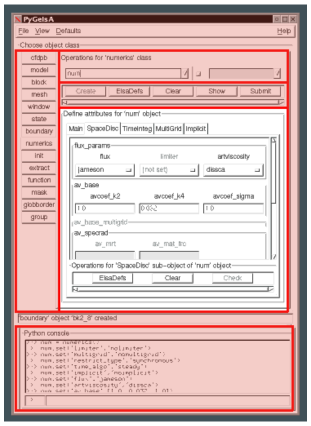
num: num
flux: jameson; limiter: (not set); artviscosity: dissca
avcoef_k2: 1.0; avcoef_k4: 0.032; avcoef_sigma: 1.0

'boundary' object 'bk2_8' created

```
>>> num = numerics()
 >  num.set('limiter','nolimiter')
>>> num.set('multigrid','nomultigrid')
 >  num.set('restrict_type','synchronous')
>>> num.set('time_algo','steady')
 >  num.set('implicit','noimplicit')
>>> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>>> num.set('av_base',[1.0, 0.032, 1.0])
```

2.2.5. Sub-classes tabnotebook widget and widget attributes

When displaying an existing object, the interface's central part is used for the attribute widgets (managing the data in the description object).



'boundary' object 'bk2_8' created

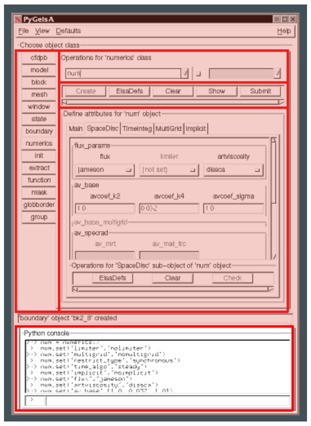
```
>-> num = numerics()
 >  num.set('limiter','nolimiter')
>-> num.set('multigrid','nomultigrid')
 >  num.set('restrict_type','synchronous')
>-> num.set('time_algo','steady')
 >  num.set('implicit','noimplicit')
>-> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>-> num.set('av_base',[1.0, 0.032, 1.0])
```

For classes with only a limited number of attributes, or when only a small number of them is meaningful at once, these widgets are all contained in the single `Main` tab of the secondary, horizontal, tabnotebook widget.

For the numerics class (only case in the current version), several tabs are displayed for sub-classes (which were not documented for the text mode interface) : `Main`, `SpaceDisc`, `TimeInteg`, `Implicit`, `MultiGrid`. These sub-classes, and the associated tabs, are meant to provide faster searches for the location of specific attribute widgets.

2.2.6. Python *console*

The lower part of the interface is dedicated to a text console, which accepts Python–*elsA* commands and is fully linked to the "pure GUI" upper part.


PyGelsA
File View Defaults Help
Choose object class
Operations for 'numerics' class
Create ElsaDefs Clear Show Submit
Define attributes for 'num' object
Main SpaceDisc TimeInteg MultiGrid Implicit
Operations for 'SpaceDisc' sub-object of 'num' object
ElsaDefs Clear Check
Python console

'boundary' object 'bk2_8' created

```
>-> num = numerics()
 >  num.set('limiter','nolimiter')
>-> num.set('multigrid','nomultigrid')
 >  num.set('restrict_type','synchronous')
>-> num.set('time_algo','steady')
 >  num.set('implicit','noimplicit')
>-> num.set('flux','jameson')
 >  num.set('artviscosity','dissca')
>-> num.set('av_base',[1.0, 0.032, 1.0])
```

2.3. Functionalities of the present version

The present PyG*elsA* version is v0.7 (available in the `Help/About` menu).

This version integrates a beginning of solution for the problems of :

– default values ;
– dependency relations between attributes.

Its main limitations are :

1. the group class is not directly available in graphical mode (creation in text mode only) ; group instances appear as instances of their members' class ;
2. execution is always interactive (no "detach" mode).

The second limitation may be circumvented by re-using in `batch` mode the `user_log.epy` trace file generated by the interactive PyG*elsA* session. In this case, PyG*elsA* is used as an (offline) script editor and not as a direct control interface for the `elsa` application.

Remark :

Scripts *imported* into PyG*elsA* (`File/Import script file`) are never automatically executed ; in this case, description objects are created and filled, but calls to "executive" methods (`submit`, `compute`, `extract`) are routed to a "pending operations" stack, which may be validated on demand to start the actual computation (`File/Activate pending ops`), or only visualised (`File/Show pending ops`). The management of this stack is presently incomplete, which may lead to difficulties (improper operations order) when creating new objects relative to the original script.

2.4. GUI representation of newly added interface elements

The PyG*elsA* GUI is self-configurating, using the `EpAttrDefs.py` (for attribute definitions) and `EpDepends.py` (for dependencies) resource files at startup.

Thus, any new elements (classes, methods, attributes) added by a developer will naturally appear in PyG*elsA* if they are correctly declared in `EpAttrDefs.py`, and dependency/influence rules defined in `EpDepends.py` will be applied.

The simple rule is : if it works in text-mode elsa, then it works in PyG*elsA*.

Remark : A current limitation is that `EpAttrDefs.py` (or rather, the associated engine in `EpAttr-Dict.py`) does not define a mechanism for the declaration of method arguments, so that, currently, using the button for a method which requires arguments will fail (by lack of an adequate popup to fill them in).

2.5. Fonts, colours and widget appearance

The default appearance of the interface is defined by the toolkit default values for the various parameters (fonts, colours, textures . . .) ; these may be overridden by a `~/.gtkrc` resource file for the `gtk+` toolkit, or by a full theme (which may use specific libraries for widget customisation), see `http://gtk.themes.org`.

A colour theme example is provided by the `PyGelsA/_.PyGelsArc` theme file, which should be copied to the user's home directory to be found by PyG*elsA* on startup.

3. MANAGING DESCRIPTION CLASS INSTANCES

Class instance management includes creating new objects (class instances), selecting them among existing objects, and creating groups of objects.

3.1. Class selection

The proper class for the instance about to be created must first of all be selected in the tabnotebook widget on the left side of the interface.

A cfdpb instance must be created before instances of some classes may be created.

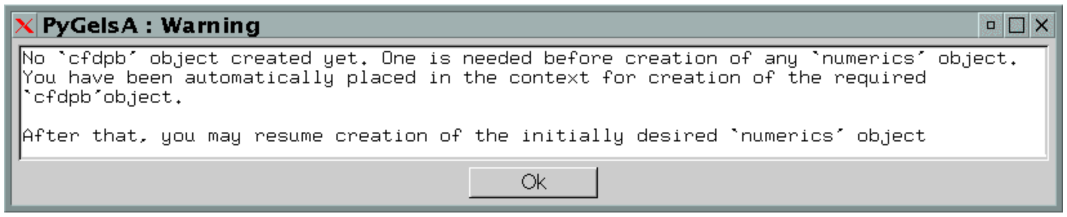
In text mode, failure to verify this condition (ex. trying to create a numerics instance when no cfdpb exists) is an error, with the following message :

```
Welcome to the elsA Python interface ; type '^D' or 'close()' to exit
elsA >>> n1=numerics()

elsA interface : ERROR   : Coherency error : missing required 'cfdpb' instance
 ... no 'cfdpb' instance is defined (or missing 'name=' argument) ;
 ... one is required before you can create a 'numerics' instance

>>> Press 'Return' to continue <<<
```

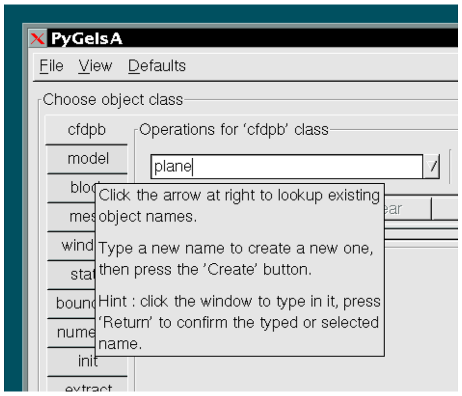
In GUI mode, a message explaining the situation is displayed in a pop-up window, and the interface is locked until its OK button is pressed :



```
No `cfdpb' object created yet. One is needed before creation of any `numerics' object.
You have been automatically placed in the context for creation of the required
`cfdpb'object.

After that, you may resume creation of the initially desired `numerics' object
```

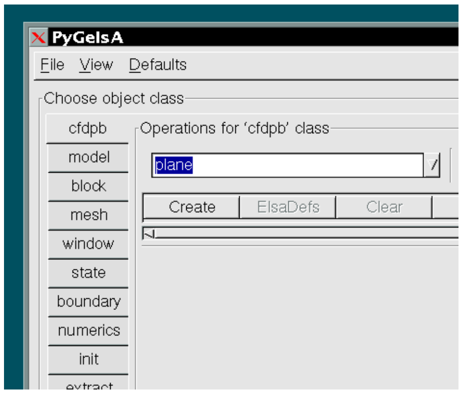
3.2. Instance name specification



Click the arrow at right to lookup existing object names.

Type a new name to create a new one, then press the 'Create' button.

Hint : click the window to type in it, press 'Return' to confirm the typed or selected name.



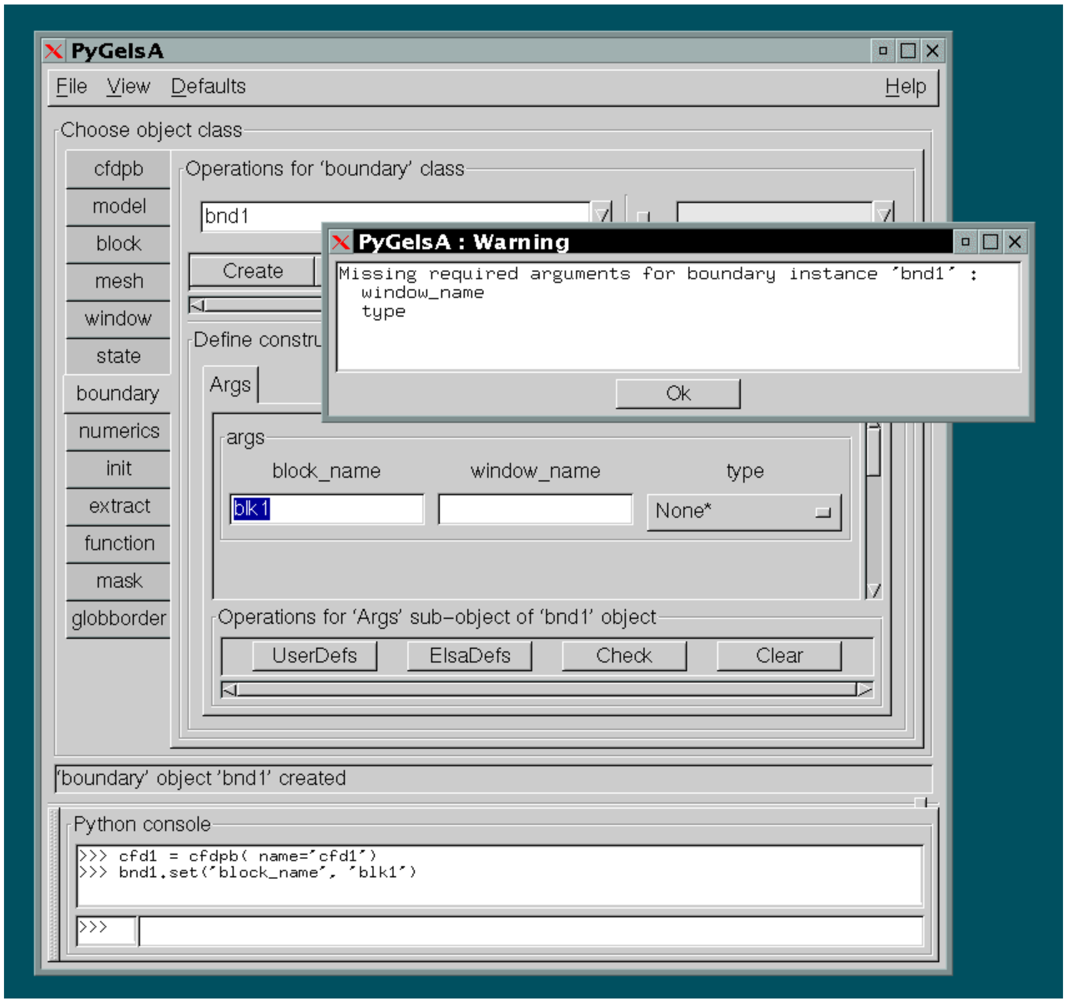
Selection of an instance name (manually typed in the entry widget's window or chosen among the existing names in the pop-down list) is complete only after validation through a `Return` key press ; the chosen name is then overstriked in colour and the `Create` button activated if needed (new object name).

The pop-up help string for the name entry field is a reminder of the above procedure.

3.3. 'Args' tab for constructor arguments

For those classes requiring argument(s) for instance construction, a specific `Args` tab is created simultaneously with the activation of the `Create` button (this is the only tab which exists before the object itself . . .).

Until all required arguments in this tab are filled in, use of the `Create` button leads to a warning message in a pop-up window, listing still required arguments :



```
Missing required arguments for boundary instance 'bnd1' :
  window_name
  type
```

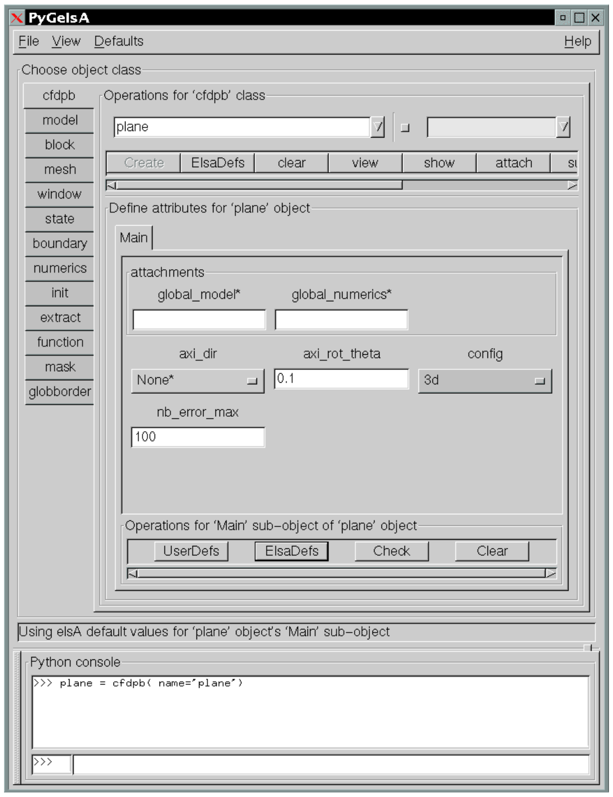
'boundary' object 'bnd1' created

```
>>> cfd1 = cfdpb( name='cfd1')
>>> bnd1.set('block_name', 'blk1')
```

3.4. Instance creation

When a new object name has been typed in the adequate entry field and validated (see above), and required arguments filled in, a mouse click on the `Create` button completes the creation of a new object, which is the value of a Python variable with this object's name.

At this time, the `Create` button is deactivated, the `ElsaDefs`, `clear` ... buttons for various methods to be called on the object are activated, and the by-sub-class tabnotebook widget for attributes is created and filled with current values (using any defined class defaults, see `MU-98057`).



plane

global_model*
global_numerics*

axi_dir: None*
axi_rot_theta: 0.1
config: 3d
nb_error_max: 100

Using elsA default values for 'plane' object's 'Main' sub–object

```
>>> plane = cfdpb( name='plane')
```

Empty page

4. MANAGING DESCRIPTION ATTRIBUTE VALUES

4.1. Plain ("atomic") attributes

Modification and display of attribute values is done through the associated widgets, whose type (entry field, pop-down list . . . , see appendix II) is automatically adapted to the type and value range of the attribute.

Each attribute widget has its tool-tip (help pop-up), which appears when the mouse pointer stays a bit long over it (the pop-up string is the description string from the `EpAttrDefs.py` resource file).

Remark : In the case of the entry widget, the sensitive region for tool-tip action is the entry widget itself, and not the label (displaying the attribute name) over it.

4.2. Value checking

Value checking is performed at two levels :

– by the attribute domain of definition (from `EpAttrDefs.py`) ;
– in the context of other attributes, when possible (from `EpDepends.py`).

4.3. Macro-attributes

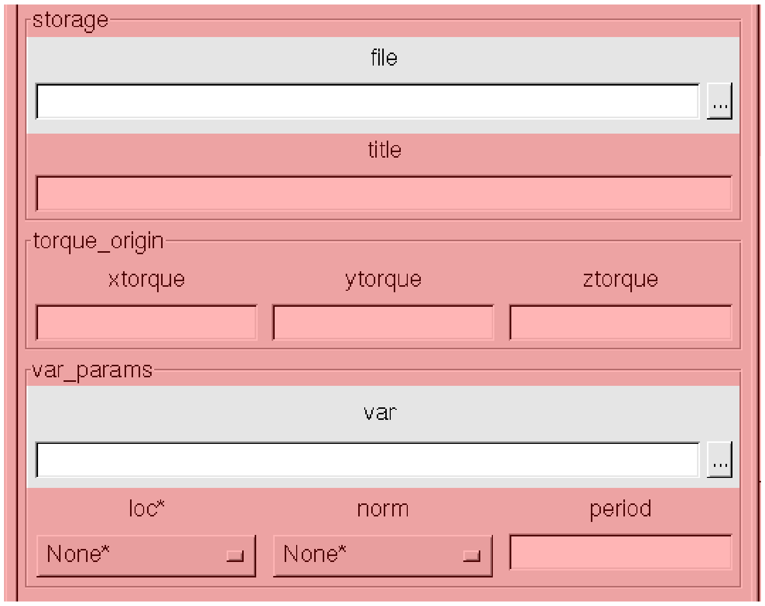
Individual widgets for atoms of a macro-attribute (or variants of a variable-length macro-attribute) are gathered in a frame, labelled with the macro-attribute name.

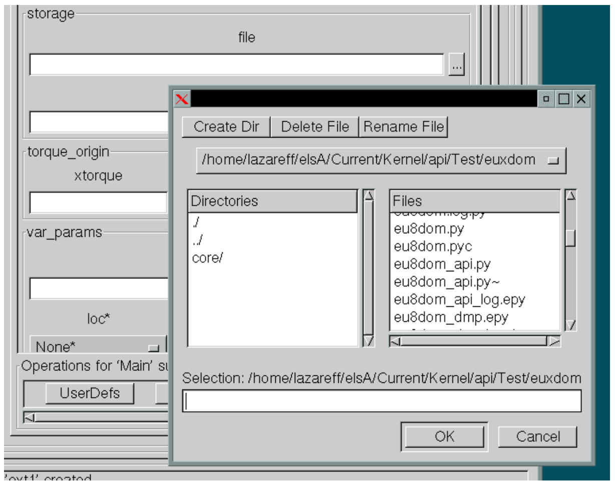
Dependency rules may lead to global (macro-attribute frame) or partial (some atoms) inactivation (see chapter 5).

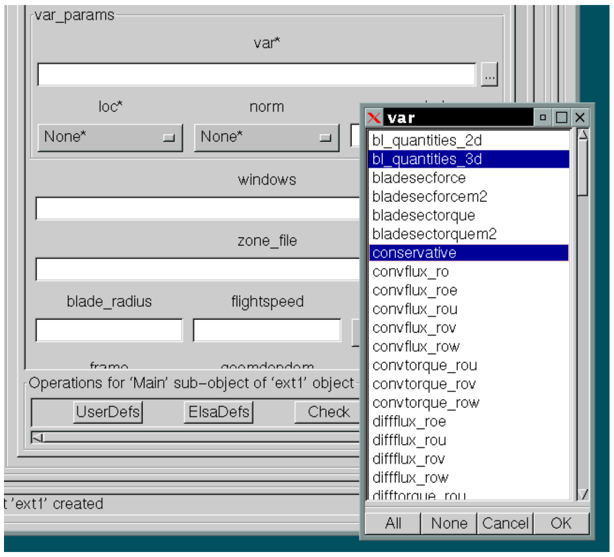
4.4. Special widgets

The `file` and `var` attributes use special widgets, which combine an entry field and a `'...'` button, which opens :

– for `file`, a file selector dialog ;
– for `var`, a dedicated list selector for `var` values.


storage
file
...
title
torque_origin
xtorque
ytorque
ztorque
var_params
var
...
loc*
norm
period
None*
None*


storage
file
torque_origin
xtorque
var_params
loc*
None*
Operations for 'Main' su
UserDefs
Create Dir
Delete File
Rename File
/home/lazareff/elsA/Current/Kernel/api/Test/euxdom
Directories
./
../
core/
Files
eu8dom.py
eu8dom.pyc
eu8dom_api.py
eu8dom_api.py~
eu8dom_api_log.epy
eu8dom_dmp.epy
Selection: /home/lazareff/elsA/Current/Kernel/api/Test/euxdom
OK
Cancel


var_params
var*
loc*
norm
None*
None*
windows
zone_file
blade_radius
flightspeed
Operations for 'Main' sub–object of 'ext1' object
UserDefs
ElsaDefs
Check
'ext1' created
var
bl_quantities_2d
bl_quantities_3d
bladesecforce
bladesecforcem2
bladesectorque
bladesectorquem2
conservative
convflux_ro
convflux_roe
convflux_rou
convflux_rov
convflux_row
convtorque_rou
convtorque_rov
convtorque_row
diffflux_roe
diffflux_rou
diffflux_rov
diffflux_row
All
None
Cancel
OK

5. DEPENDENCIES MANAGEMENT

The current PyG*elsA* version (as the text-mode Python–*elsA* interface through the `check` method) provides a partial implementation of dependencies management.

At this time, the scope of this management is mostly limited to the context of one description object (no general inter-object dependencies). An exception is the dependency of the extract.`var` attribute to the model.`phymod` one.

5.1. Dependency and influence rules

Relations between attributes are described not by a complete tree graph of all possible scenarii, which would be un-manageable, but by a set of transition rules from one state to another (i.e. when defining a new attribute value), describing the relations between neighbours along the tree graph. These rules may be inspected in the `EpDepends.py` file.

The rules may be divided in two sub-sets :

– dependency rules, which specify when an attribute's widget should be active according to the context of other widgets' values ;

– influence rules, which specify which attributes should be defined when a specific attribute has been given a specific value.

The first sub-set is responsible for grayed-out (or folded-up for macro-attributes) widgets, while the second one governs the appearance of `*` "attention" symbols alongside the names of attributes whose value is requested.

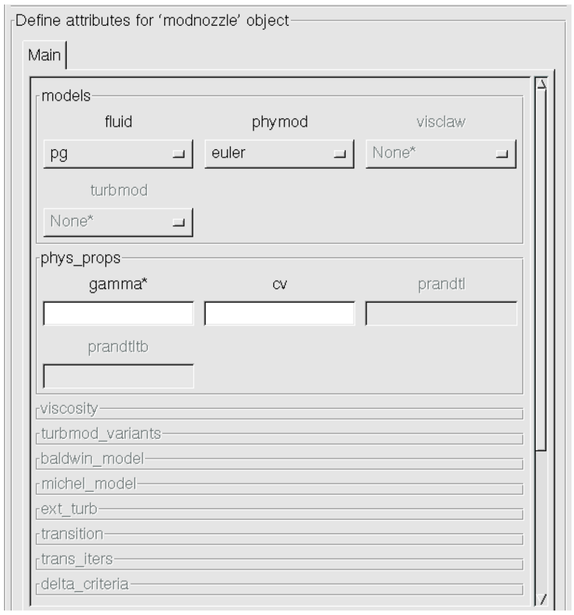
Differently put, when sitting on the tree graph at the location of an attribute, with his/her back to already-defined values and at their edge (last defined attribute on this tree segment), a new-age Maxwell daemon could reflect during this break on the data-filling voyage that he/she was able to get there through dependency rules, and will (probably, user permitting) go where influence rules call him/her.

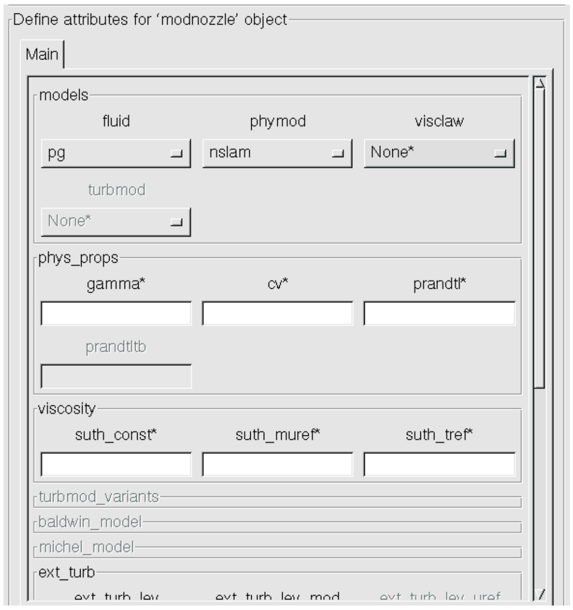
This is hyperbolic behaviour (provided that, hopefully, the rules contain no circular references, or loops) !

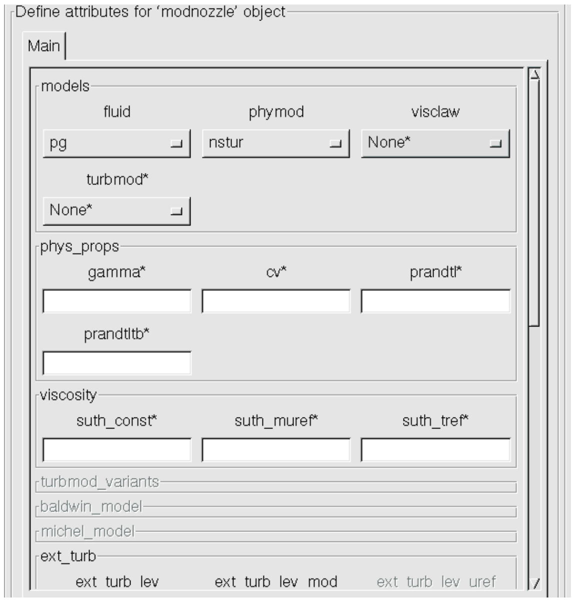
5.2. Automatic behaviour

5.2.1. A priori checking

The following screen-shot fragments are an illustration for this functionality, in the case of dependency upon the physical model parameter (the model.`phymod` attribute) :


Define attributes for 'modnozzle' object
Main
models
fluid: pg
phymod: euler
visclaw: None*
turbmod: None*
phys_props
gamma*
cv
prandtl
prandtltb
viscosity
turbmod_variants
baldwin_model
michel_model
ext_turb
transition
trans_iters
delta_criteria


Define attributes for 'modnozzle' object
Main
models
fluid: pg
phymod: nslam
visclaw: None*
turbmod: None*
phys_props
gamma*
cv*
prandtl*
prandtltb
viscosity
suth_const*
suth_muref*
suth_tref*
turbmod_variants
baldwin_model
michel_model
ext_turb
ext_turb_lev
ext_turb_lev_mod
ext_turb_lev_uref


Define attributes for 'modnozzle' object
Main
models
fluid: pg
phymod: nstur
visclaw: None*
turbmod*: None*
phys_props
gamma*
cv*
prandtl*
prandtltb*
viscosity
suth_const*
suth_muref*
suth_tref*
turbmod_variants
baldwin_model
michel_model
ext_turb
ext_turb_lev
ext_turb_lev_mod
ext_turb_lev_uref

Those widgets for attributes with no meaning in the current context are put in in-active state (they appear grayed-out) ; frames for in-activated macro-attributes are grayed-out and "folded up" (ex. `baldwin_model`, above `michel_model`), so that only the macro-attribute name and the (collapsed) frame are visible. This is *a priori* checking, as opposed to user-requested *a posteriori* checking, see below 5.3.

5.2.2. Automated data definition (not implemented)

The current PyG*elsA* version does not sport any automatic mechanism for data definition, other than "from here on, use all default values" (using the `ElsaDefs` button).

The rules-based approach employed here for (hard logic) dependency/influence management is the bread-and-butter of expert systems, but is not of great help for filling out description objects.

A more complete (and a dangerous one if not carefully crafted) mechanism would involve fuzzy logic (probabilistic rules) to determine the "correct" value of at least a fraction of attribute values.

The time is probably a little too early now for the development of such an fuzzy expert system (`http://www-2.cs.cmu.edu/Groups/AI/html/faqs/ai/fuzzy/part1/faq-doc-4.html`) for the *elsA* interface

So, for the time being, the Maxwell daemon see 5.1, will not be allowed to proceed on its own,.

5.3. User-requested a posteriori checks

The `check` button invokes a `check()` call ; this is useful when attribute values have been modified through the Python console, see chapter 6.

User-required checks are normally not needed when the interface is used in "pure GUI" mode, because in this case all checks are made *a priori*, and all attribute values should be consistent.

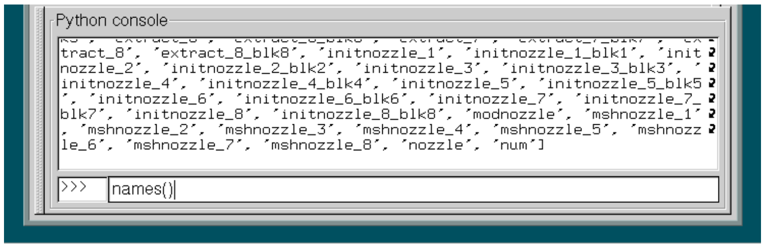
One exception is when using the `ElsaDefs` button (request for object completion from default values), because default values may together with user-specified values (or on their onw, even) form an inconsistent whole.

Empty page

6. PYTHON CONSOLE USAGE (TEXT-MODE INSIDE GUI)

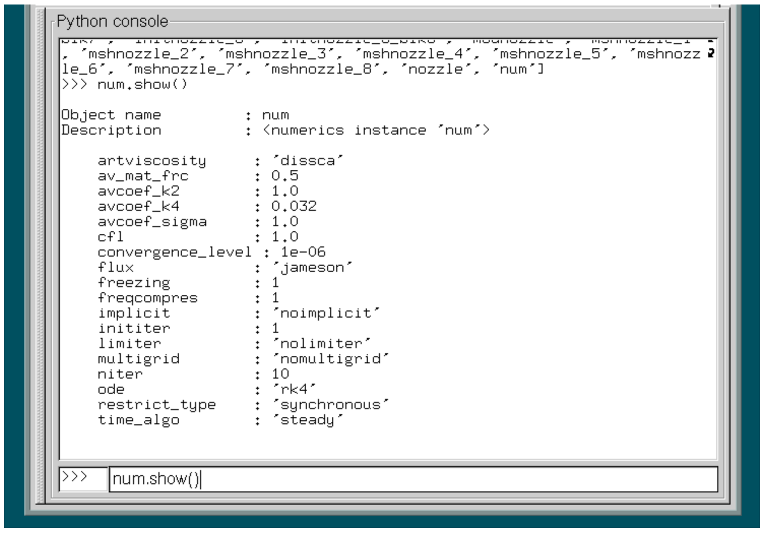
The Python console duplicates the functionality of the text-mode Python–*elsA* interface, usually accessed through the text-only mode of the `elsa` application.

Here are some examples of actions, initiated from the Python console :



```
tract_8', 'extract_8_blk8', 'initnozzle_1', 'initnozzle_1_blk1', 'init
nozzle_2', 'initnozzle_2_blk2', 'initnozzle_3', 'initnozzle_3_blk3', '
initnozzle_4', 'initnozzle_4_blk4', 'initnozzle_5', 'initnozzle_5_blk5
', 'initnozzle_6', 'initnozzle_6_blk6', 'initnozzle_7', 'initnozzle_7_
blk7', 'initnozzle_8', 'initnozzle_8_blk8', 'modnozzle', 'mshnozzle_1'
, 'mshnozzle_2', 'mshnozzle_3', 'mshnozzle_4', 'mshnozzle_5', 'mshnozz
le_6', 'mshnozzle_7', 'mshnozzle_8', 'nozzle', 'num']

>>> names()
```



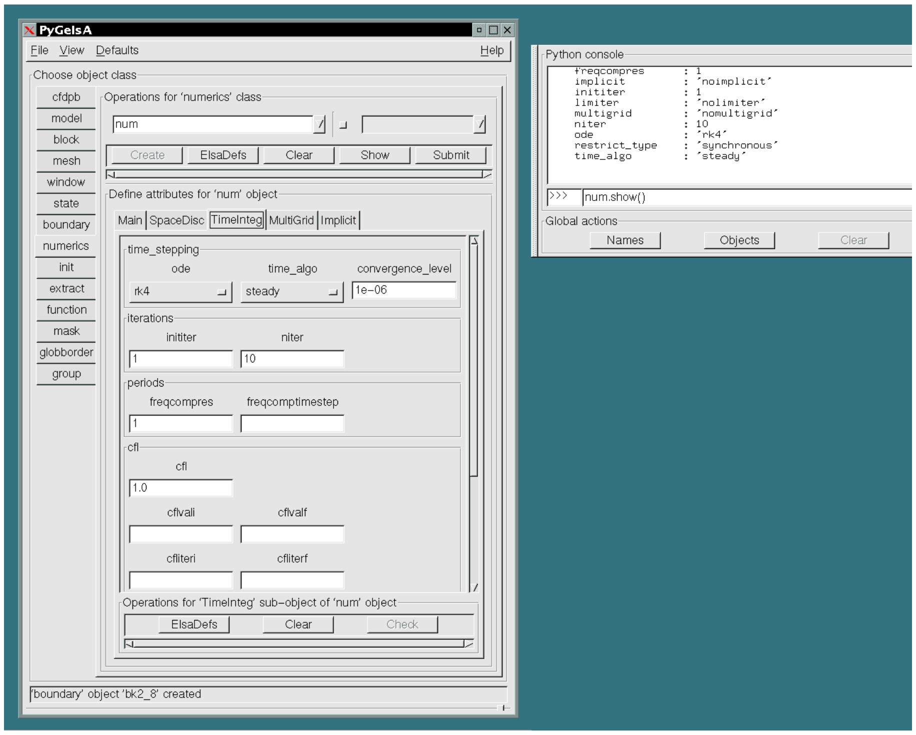
```
, 'mshnozzle_2', 'mshnozzle_3', 'mshnozzle_4', 'mshnozzle_5', 'mshnozz
le_6', 'mshnozzle_7', 'mshnozzle_8', 'nozzle', 'num']
>>> num.show()

Object name         : num
Description         : <numerics instance 'num'>

    artviscosity      : 'dissca'
    av_mat_frc        : 0.5
    avcoef_k2         : 1.0
    avcoef_k4         : 0.032
    avcoef_sigma      : 1.0
    cfl               : 1.0
    convergence_level : 1e-06
    flux              : 'jameson'
    freezing          : 1
    freqcompres       : 1
    implicit          : 'noimplicit'
    inititer          : 1
    limiter           : 'nolimiter'
    multigrid         : 'nomultigrid'
    niter             : 10
    ode               : 'rk4'
    restrict_type     : 'synchronous'
    time_algo         : 'steady'

>>> num.show()
```

If required (ex. for desktop real estate optimisation) the console may be hidden (using the `View/Toggle Python console` entry), or extracted from the interface by dragging it by its guilloched leftmost part :



num

time_stepping: ode rk4, time_algo steady, convergence_level 1e−06

iterations: inititer 1, niter 10

periods: freqcompres 1, freqcomptimestep

cfl: cfl 1.0, cflvali, cflvalf, cfliteri, cfliterf

'boundary' object 'bk2_8' created

```
    freqcompres      : 1
    implicit         : 'noimplicit'
    inititer         : 1
    limiter          : 'nolimiter'
    multigrid        : 'nomultigrid'
    niter            : 10
    ode              : 'rk4'
    restrict_type    : 'synchronous'
    time_algo        : 'steady'

>>> num.show()
```

The reverse operation is also possible, by bringing back the console to its initial location.

Scrolling the text for the console output (larger window of the console) is possible by clicking in it and using the `PageUp` and `PageDown` keyboard keys.

7. FILE-BASED OPERATIONS (PERSISTENCY)

The only means of persistent storage available at this time for *elsA* scripts is file-based, using script files with Python–*elsA* commands.

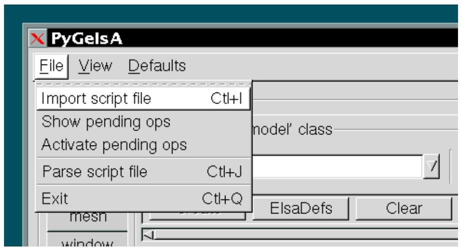
Other means could be useful, ex. databases, see **??**, or XML-based descriptions, as studied in `/ELSA/NI-01061` "Persistance des données de l'interface utilisateur *elsA*".

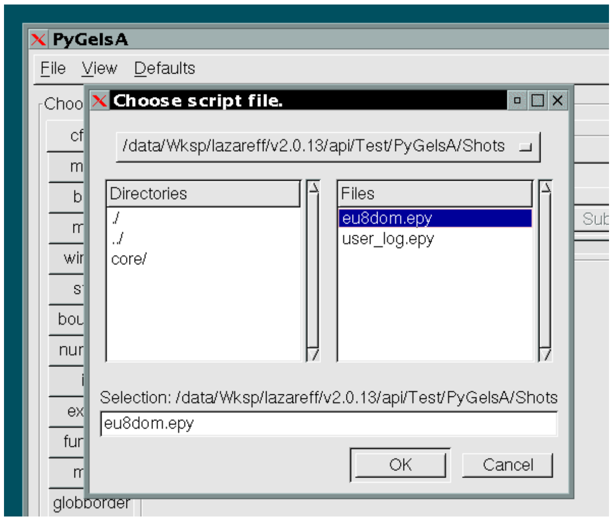
7.1. Script file import and execution

The PyG*elsA* interface may be used as an editor for existing Python–*elsA* script files, or also to launch their execution.

The `Parse script file` and `Import script file` fields in the `File` menu respectively allow :

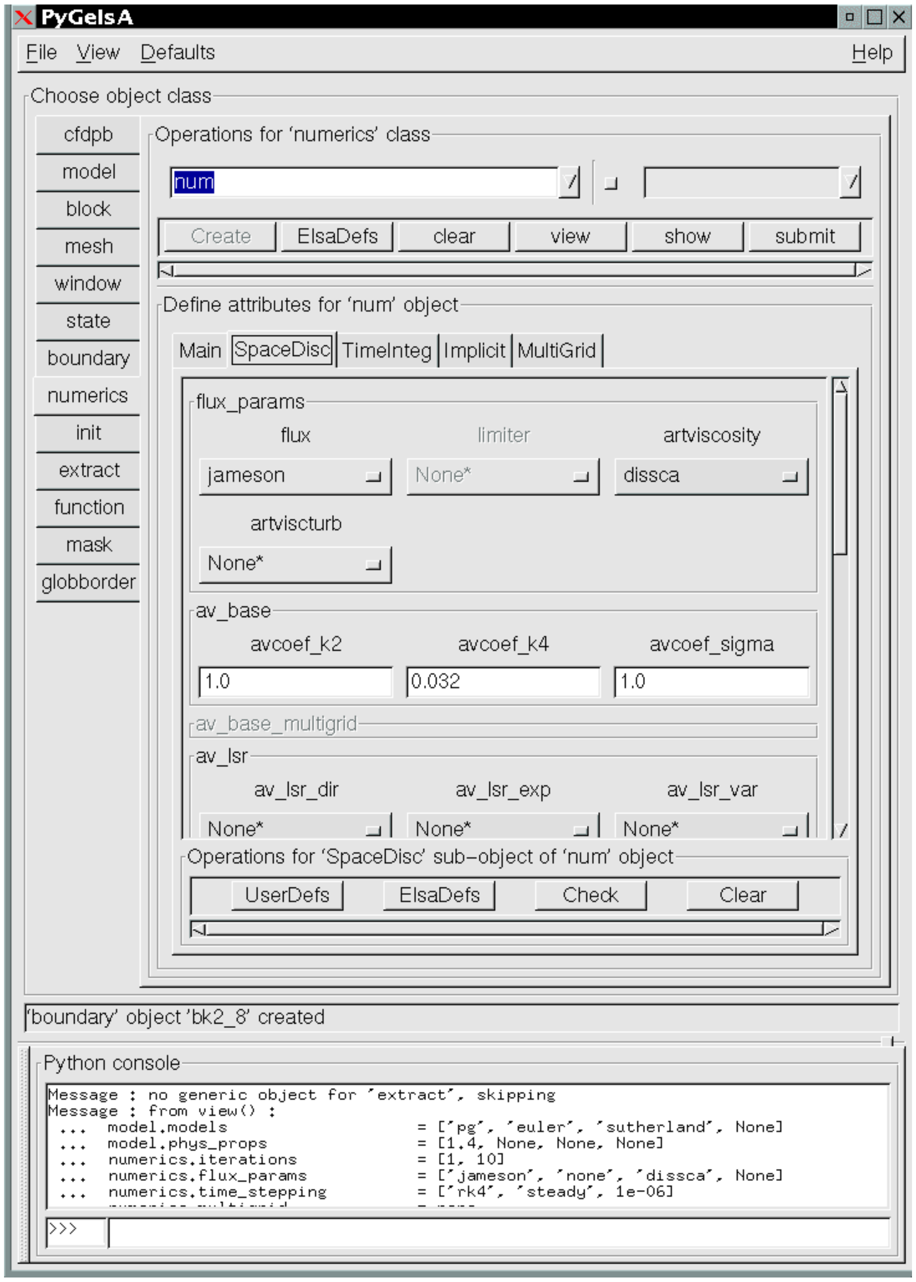
– direct execution of the script (`Parse script`) ;
– plain import, i.e. objects creation and filling, without execution (`Import script`).





Those “executive” method (`submit`, `compute`, `extract`) calls trapped during the script import may be visualised through the `File/Show pending ops` menu entry, while their execution is activated through the `File/Activate pending ops` entry.

Use of the `compute` button of the cfdpb instance implies activation of pending operations.



num

flux_params: flux jameson; limiter None*; artviscosity dissca; artviscturb None*

av_base: avcoef_k2 1.0; avcoef_k4 0.032; avcoef_sigma 1.0

av_lsr: av_lsr_dir None*; av_lsr_exp None*; av_lsr_var None*

‘boundary’ object ’bk2_8’ created

```
Message : no generic object for ’extract’, skipping
Message : from view() :
 ...   model.models                    = [’pg’, ’euler’, ’sutherland’, None]
 ...   model.phys_props                = [1.4, None, None, None]
 ...   numerics.iterations             = [1, 10]
 ...   numerics.flux_params            = [’jameson’, ’none’, ’dissca’, None]
 ...   numerics.time_stepping          = [’rk4’, ’steady’, 1e-06]
>>>
```

7.2. Trace and dump files

Just like in the case of the text interface, the trace file and optionally the dump file provide a persistent version of the state of the objects (their data values), whether created from script, graphical interface or console :

– the *trace file*, automatically created unless otherwise specified, is an image of the sequence of all operations since the interface was started ;
– the *dump file*, created by a call to the `dump` function, is an instant image of the state of all objects (or of some objects if the `dump` *method* was used), without any history character, contrary to the trace file.

In both cases, both the console and GUI's top part mirror the changes in the script file.

Please refer to the *elsA* User's Manual (`MU-98057`) for details about the trace and dump files, and their differences.

Empty page

APPENDIX I : PyG*elsA* INSTALLATION AND REQUIRED ENVIRONMENT

I.1. Required libraries

PyG*elsA* uses the Python `pygtk` bindings (by James Henstridge) of the `gtk+` graphical toolkit.

On top of the base `glib` and `gtk+` libraries [1], installation of the `pygtk` library is required, as part of the `gnome-python` package [2], to provide the Python interpreter with a graphical front-end.

The following versions are adequate (but will shortly be superseded by new components of the GNOME 2.0 development) :

```
glib  : glib-1.2.8, glib-1.2.9 ;
gtk   : gtk-1.2.8, gtk-1.2.9 ;
pygtk : gnome-python-1.0.53.
```

The GNOME environment is not required for PyG*elsA* ; this GUI should run on all Unix systems[3].

Remarks :

1. A typical installation is made of the following steps [4] :

```
cd $HOME/install
cat $HOME/incoming/<package>-<version>.tar.gz | gzip -cd | tar xvf -
cd <package>-<version>
./configure --prefix=$HOME/local && make && make install
```

2. If the installation directory is `<prefix>` (as specified by `./configure --prefix=<prefix>`), the current shell environment must contain[5] :
   `<prefix>/bin` as head of `PATH`
   `<prefix>/lib` as head of `LD_LIBRARY_PATH`
   (or their equivalents) *before* `gtk+` installation, so that installation parameters for `glib` be accessible to `configure` ;

3. If GNOME is not installed on the computer, the installation must start in the `pygtk` sub-directory of the `gnome-python` source hierarchy ; the message issued by `configure`, warning that the `imlib` is missing, may be safely ignored ;

[1] freely available on `http://www.gtk.org` ; initial authors : Peter Mattis, Spencer Kimball, Josh MacDonald

[2] `http://www.gnome.org`

[3] And maybe even on Win32 platforms, as a standalone script editor, using the Python stubs defined in `EpIntStubs.py` to mimic the *elsA* kernel

[4] on the hypothesis that the original software archives are stored in `$HOME/incoming`, unpacked in `$HOME/install` and final installation made in `$HOME/local` (`<prefix>`) parameter of the `configure` script

[5] Variable names might be different on some platforms

4. On some platforms, it may be necessary to install the GNU version of the `make` utility [6], to be able to generate the `gtkmodule_defs.c` or `gtkmodule_impl.c` files as part of the `pygtk` compilation process ;

5. The `gcc` compiler, and associated linkage utilities on the GNU/Linux platform, require the `-Xlinker -export-dynamic` options ;

6. The `gtk+` and `pygtk` libraries and Python modules will be installed in the `site-packages` directory of the Python interpreter found during the installation. This must be the same as during creation of the `elsA.x` executable ; if this is not the case, the PYTHONPATH and LD_LIBRARY_PATH environment variables will have to be defined with `<prefix>/lib/python<version>/site-packages` in front for each PyG*elsA* execution ;

7. For those platforms which support different executable formats (ex. 32/64 bits on SGI/IRIX), it may be necessary to implement sub-directories for each of them (ex. `<prefix>/lib/32` and `<prefix>/lib/64`), to be referenced in the LD_LIBRARY_PATH and PYTHONPATH variables, and to modify the `Makefile` files according to the one in the *elsA* distribution (ex. on SGI/IRIX, `-n32` or `-64` arguments for the various compilation/linking tools).

   In such a case, installation of `Python`, `glib`, `gtk+` and `pygtk` must be adapted to the chosen sub-directory structure, by modifying every `configure` invocation, as in :

   ```
       ./configure --prefix=$HOME/local/32
       ./configure --prefix=$HOME/local/64
   ```
   the `Python` modules will be installed in the following directories :
   ```
       <prefix>/32/lib/python1.5/site-packages
   ```
   and
   ```
       <prefix>/64/lib/python1.5/site-packages
   ```

   and both installations (32 and 64 bits) may be tested with :

   ```
   oneroa36: export prefix=/home/local
   oneroa36: export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$prefix/32/lib:$prefix/64/lib
   oneroa36: <prefix>/32/bin/python
   Python 1.5.2 ...
   >>> import gtk
   >>>
   oneroa36: <prefix>/64/bin/python
   Python 1.5.2 ...
   >>> import gtk
   >>> dir(gtk)
   ['ACCEL_LOCKED', 'ACCEL_MASK', 'ACCEL_SIGNAL_VISIBLE' ...
   ```

   The same tests may be carried on with the *elsA* executable instead of the plain Python one.

[6]Available on `http://ww.gnu.org`, or as `ftp://ftp.lip6.fr/pub/gnu/make/make-xxx.tar.gz`

I.2. Environment variables

The usual `$ELSAHOME/Dist/lib/py` directory (towards which the PYTHONPATH environment variable must point to) holds the `PyGelsA` sub-directory of all GUI-specific modules.

Launching this GUI is accomplished with the `elsa -g` [7] command, other command-line options being also available (see `MU-98057`, or try `elsa --help`).

Remark : Using PyG*elsA* in an overloaded computer environment (computer or network) is to be avoided. As for any graphical interface, any gain in efficiency is obtained through augmented resource usage, as needed for a greater quantity of displayed information ; this gain may be completely annihilated, even reversed, by a too low or uneven display speed.

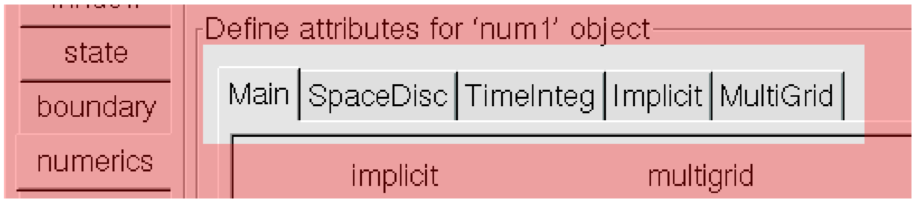
[7] Where `elsa` is a symbolic link to `$ELSAHOME/Dist/lib/py/EpelsA.py`

Empty page

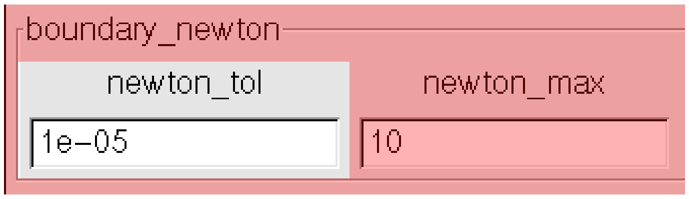
APPENDIX II : WIDGET TYPES

The PyG*elsA* interface uses a variety of widgets from the `gtk+` toolkit. In the following illustrations, taken from PyG*elsA*, the non-pertinent regions of interest are overlayed by a (red) semi-transparent veil.

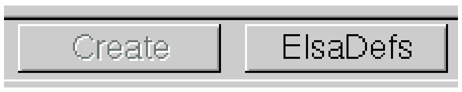
– the *tabnotebook widget* :



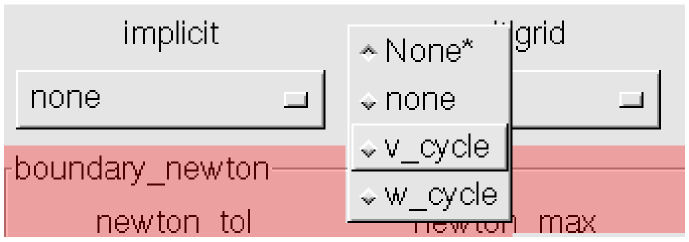
– the *entry field* ; this is used to input and display string values, or numerical values with too wide a range for a *slider widget* (not represented) :



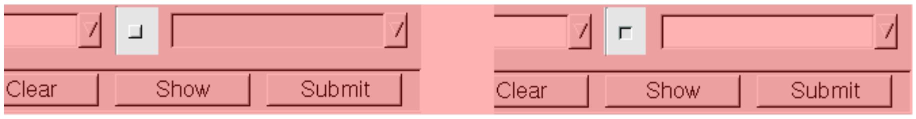
– the *button*, in *in-active state* (locked) and *active state* :



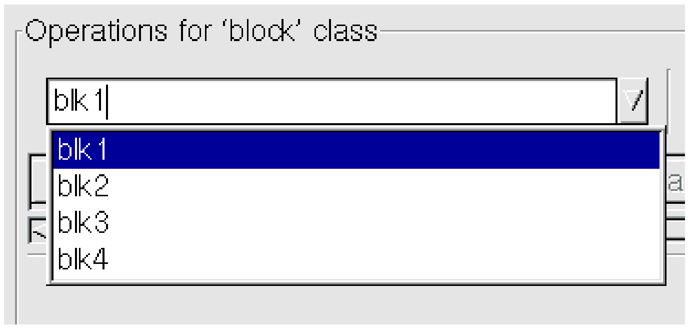
– the *radio button*, in *closed state* and *deployed state* ; this is used for a choice in a closed list of possible values :



– the *toggle button*, in *raised state* and *pressed state* :



– the *combo widget* with its *pop-down list* show in *deployed state* :



INDEX

Empty page

TYPICAL DOCUMENT DISTRIBUTION

Archives Secrétariat

Rédacteurs

Développeurs *elsA*

Utilisateurs *elsA*

Responsable Documentation

FIN de LISTE