The COMANDO Framework

Marco Langiu

CONTENTS:

LIST OF FIGURES

LIST OF TABLES

COMANDO is a next generation modeling framework for **Component-Oriented Modeling AND Optimization** of the design and operation of energy systems. An energy system is considered to be a collection of different interconnected components whose purpose is to satisfy demands of various commodities such as, e.g., electric power, heat and cooling, in a variety of different operating conditions.

When such a system is built (or extended), there are many design and operational decisions that need to be made. In this context, optimizing the design and operation means finding a set of decisions that results in a minimal value for some generalized costs, taking into account restrictions imposed by individual components, their connections or by other safety-related, social, political, or economic considerations.

COMANDO provides means to...

- model existing energy systems and possible extensions in a flexible, component-oriented fashion.
- use the resulting system models to create mathematical optimization problems
- solve these problems directly or use tools to automatically reformulate them to a form, more amenable to solution

To determine whether COMANDO suits your needs and avoid confusion when something doesn't work as expected, it is important to first understand some basic concepts and terminology introduced by COMANDO. We therefore recommend that you read the background section of this documentation before attempting to start hacking away. If you feel brave and know what you're doing, you can also skip ahead to the quickstart section, instead.

When using COMANDO in an academic context please cite our preprint on arXiv.org:

```
@Article{langiu2021comando,
  author        = {Marco Langiu and David Yang Shu and Florian Joseph Baader and Dominik Hering and Uwe␣
↪Bau and Andr\'{e} Xhonneux and Dirk M\"uller and Andr\'e Bardow and Alexander Mitsos and Manuel Dahmen}
↪,
  title         = {COMANDO: A Next-Generation Open-Source Framework for Energy SystemsOptimization},
  howpublished  = {\url{https://arxiv.org/abs/2102.02057v1}},
  eprint        = {http://arxiv.org/abs/1707.02514v4},
  eprintclass   = {math.OC},
  eprinttype    = {arXiv},
  optauthor     = {M.~Langiu and D.~Y.~Shu and F.~J.~Baader and D.~Hering and U.~Bau and A.~Xhonneux and␣
↪D.~M\"uller and A.~Bardow and A.~Mitsos and M.~Dahmen},
  optnote       = {submitted on 04Feb2021 to cace},
  primaryclass  = {math.OC},
  year          = {2021},
}
```

Todo:

- Extend *Quickstart Guide* examples for all 3 modeling phases.
- Find out how to change the name of the 'Parameters' and 'Variables' sections in the API to display 'Arguments' and 'Attributes' instead, to avoid confusion with *comando.Parameter* and *comando.Variable*!

CHAPTER

ONE

BACKGROUND

This section gives a brief overview of the concept behind COMANDO, the problem formulations that can be addressed with it, as well as its usage.

1.1 Concept

The aim of COMANDO is to enable object-oriented modeling of energy systems and their constituting components for the purpose of creating optimization problems related to the system's design and operation. Unlike other energy system modeling frameworks, COMANDO does not impose restrictions to maintain the resulting optimization problems in a particular class such as linear programming (LP) or mixed-integer linear programming (MILP). Instead, component and system models may contain a wide range of nonlinear expressions, including nonconvex and nonsmooth functions. Furthermore ordinary differential equations describing system dynamics can be included directly, without manual discretization. In this way COMANDO allows users to create component and system models in a natural fashion, without having to obscure them with optimization-specific implementation details.

By specifying the connections between components, different component models can be combined into a system model. Based on a finished system model, different optimization problems can be created by specifying objective functions, the operational scenarios and their time-structure to be considered, as well as associated parameter data.

To bring the resulting problem formulations into a form that solvers understand, certain reformulations may be necessary. COMANDO provides routines for substitution and reformulation of expressions, as well as for automatic approximation via linearization and time-discretization. Once an acceptable problem formulation is created, it can be translated to different algebraic modeling languages (AMLs) or be passed directly to an appropriate solver for solution.

The details of the different steps of this process are explained in the following.

1.2 Usage

The usage of COMANDO can be split into three phases

1. Modeling phase
    - Component model creation
    - System model creation
2. Problem formulation phase
    - problem generation
        - objective selection
        - time-structure selection
        - scenario-structure selection
        - providing data

- problem reformulation
  - time discretization
  - linearization
  - ...

3. Problem solution phase
   - via a solver interface (e.g., GUROBI, BARON)
   - via an algebraic modeling language (AML) interface (e.g., GAMS, PYOMO)
   - via custom algorithm
     - initialization
     - decomposition
     - ...

1.2.1 Modeling phase

In the modeling phase the system behavior is specified in terms of the component behavior and system structure.

Components

In COMANDO a component is the basic building block of an energy system. It represents a model of a generic real-world component, specified via a collection of mathematical expressions that are given in a symbolic form.

Take for example the following simple description of some generic component C:

$$C_{\text{out}} = C_{\text{in}} C_{\eta} \tag{1.1}$$

$$C_{\eta} = (C_0 + C_1\, C_{\text{pl}} + C_2\, C_{\text{pl}}^2) \tag{1.2}$$

$$C_{\text{pl}} = C_{\text{out}} / C_{\text{out,max}} \tag{1.3}$$

It contains different *symbols* that represent different design and operational quantities of C and equations that correlate these symbols with each other. Here, we may assume that $C_{\text{out,max}}$ is a design quantity, while the symbols C_{out}, C_{in}, C_{η}, and C_{pl} are operational quantities and C_0, C_1, and C_2 are some numerical coefficients.

We can create a COMANDO model of this component by deciding which role the different symbols play for some design or operation process. In COMANDO a distinction is made between quantities that are given as input data and those that are to be determined by the use of the model, i.e., by formulating some optimization problem incorporating the model and solving that problem.

Symbols that act as placeholder for input data can be modeled as a *comando.Parameter*, while those that are to be determined can be modeled as a *comando.Variable* or a *comando.VariableVector*, depending on whether they represent design- or an operational-related quantity.

Note: Objects of type *Variable* / *VariableVector* always represent scalar / vector values while those of type *Parameter* may represent both, depending on the data that is assigned to them. In the context of design and operation, design quantities are always modeled by symbols or composed expressions that represent scalar values, while operational quantities are always modeled by symbolss or composed expressions that represent vector values, with each entry representing the value of the corresponding quantity for a certain operational scenario and timestep.

We may for example model the coefficients C_0, C_1, and C_2 as objects *C_0*, *C_1*, and *C_2* of type *Parameter*, $C_{\text{out,max}}$ as a *Variable* *C_out_max* and C_{out}, C_{in} as objects *C_out*, *C_in* of type *VariableVector*.

For the definition of component COMANDO provides the *Component* class. This class defines methods to create and add the corresponding symbols to *Component* instances:

```python
# Define a new type of component
class C(comando.Component):
    """A generic component."""

    def __init__(self, label):
        """Initialize the component."""
        super().__init__(label)  # Initialize the parent class (Component)

        # Create parameters
        C_coeffs = [self.make_parameter(i) for i in range(3)]

        # Create design and operational variables
        C_out_max = self.make_design_variable('out_max')
        C_out = self.make_operational_variable('out')
        C_in = self.make_operational_variable('in')

        # Expressions formed with the defined symbols
        C_pl = self.add_expression(C_out / C_out_max, 'part_load')
        C_eta = sum(C_i * C_pl ** i for i, C_i in enumerate(C_coeffs))

        # Add constraints
        self.add_eq_constraint(C_out, C_in * C_eta, 'conversion')
        self.add_le_constraint(C_out, C_out_max, 'output_limit')

        # Add connectors
        self.add_input('INPUT', C_in)
        self.add_output('OUTPUT', C_out)

# Create an instance of our new class
c = comando.Component('C')
c_pl = c.get_expression('part_load')  # Access previously defined expression
```

Note how instead of introducing additional *VariableVector* instances for C_η and C_{pl}, we directly used their definitions and create expressions for these quantities using overloaded Python functions! The expression for C_pl is considered interesting and is therefore stored in C under the name 'part_load'. At a later point, this expression can be accessed via the `Component.get_expression()` method.

We may have similarly defined C_out as an expression, but instead we decided to make it an operational variable and introduce the correlation between input and output as an equality constraint. Similarly, we introduced an inequality constraint specifying that the component's output must be smaller than it's maximum value.

It is also possible to declare operational variables to be 'states', i.e., quantities whose time-derivative is given by some algebraic expression. This allows the consideration of dynamic effects.

Todo: Examples for `Component.declare_state()` and `Component.make_state()`

Another thing to notice are the 'INPUT' and 'OUTPUT' connectors that can be used to connect the component to others within a `comando.System` model. Components may also assign individual algebraic expressions to connectors, allowing them to be interfaced with other components. Connectors may be specified as inputs, outputs or bidirectional connectors. The former two restrict the value of the corresponding expression to be nonnegative and nonpositive, respectively, while the latter does not impose additional restrictions.

Systems

Systems are modeled as collections of interconnected components, i.e., a system model consists of a set of components and the specification of how their connectors are connected. For this purpose COMANDO provides the `comando.System` class. As the *`System`* class inherits from the *`Component`* class, systems can define additional expressions, constraints and connectors and be incorporated as subsystems within other systems.

Todo: Example for system creation

1.2.2 Problem formulation phase

Given a system model, COMANDO can currently be used to create a `comando.Problem` object, representing a mathematical optimization problem (OP) of the form:

$$
\begin{aligned}
\min_{\mathbf{x}} \quad & F_I(\mathbf{x}) + \sum_{s\in\mathcal{S}} w_s\, F^*_{II,s}(\mathbf{x}) \\
\text{s.t.} \quad & \mathbf{g}_I(\mathbf{x}) \leq \mathbf{0} \\
& \mathbf{h}_I(\mathbf{x}) = \mathbf{0} \\
& F^*_{II,s}(\mathbf{x}) = \left\{ \begin{aligned}
& \min_{\mathbf{y}_s(\cdot)} \quad F_{II,s}(\mathbf{x}, \mathbf{y}_s(\cdot)) = \textstyle\int_{\mathcal{T}_s} \dot{F}_{II}\big(\mathbf{x}, \mathbf{y}_s(t), \mathbf{p}_s(t)\big)\,\mathrm{d}t \\
& \text{s.t.} \quad \mathbf{y}^{\mathrm{d}}_s(\cdot) \in \mathbf{y}_s(\cdot) \\
& \qquad \mathbf{y}^{\mathrm{d}}_s(t=0) = \mathbf{y}^{\mathrm{d}}_{s,0} \\
& \qquad \left.\begin{aligned}
& \dot{\mathbf{y}}^{\mathrm{d}}_s(t) = \mathbf{f}\big(\mathbf{x}, \mathbf{y}_s(t), \mathbf{p}_s(t)\big) \\
& \mathbf{g}_{II}\big(\mathbf{x}, \mathbf{y}_s(t), \mathbf{p}_s(t)\big) \leq \mathbf{0} \\
& \mathbf{h}_{II}\big(\mathbf{x}, \mathbf{y}_s(t), \mathbf{p}_s(t)\big) = \mathbf{0} \\
& \mathbf{y}_s(t) = [\mathbf{y}^{\mathrm{d}}_s(t), \ldots] \\
& \mathbf{y}_s(t) \in \mathcal{Y}_s(t) \subset \mathbb{R}^{n_y} \times \mathbb{Z}^{m_y}
\end{aligned} \right\} \forall t \in \mathcal{T}_s \\
& \qquad \mathcal{T}_s = [0, T_s]
\end{aligned} \right\} \forall s \in \mathcal{S} \\
& \mathbf{x} \in \mathcal{X} \subset \mathbb{R}^{n_x} \times \mathbb{Z}^{m_x} \\
& \mathcal{S} = \{s_1, s_2, \cdots, s_{|\mathcal{S}|}\}
\end{aligned}
$$

Where $\boldsymbol{x}$ and $\boldsymbol{y}_s(\cdot)$ are the vectors of design- and operation-variables, respectively.

Problem generation

F_I and F_{II} are user-specified scalar and indexed expressions corresponding to one-time and momentary costs, $\mathcal{T}_s$ are time horizons for scenarios s in from the set $\mathcal{S}$, with corresponding probabilities w_s.

The constraints for the OP are automatically generated from the system model, i.e., all scalar relational expressions are taken as contraints and all indexed relational expressions are taken as constraints, parametrized by t, and s.

The dependence of the objective and constraint functions on time and scenario can be expressed in terms of the parameter values, which are user input. This data can currently be given only in discrete form, i.e., for a discrete time and scenario. The time-steps at which data is available

If any states were defined in the model the corresponding differential equations are currently discretized by default. An exception is the use of the `pyomo_dae` interface; here the time-continuous representation is passed and discretization via collocation can be performed. Data for the parameter values and initial guesses for the variable values can be provided based on the user-chosen sets T and S.

Note: The time- and scenario structure is only specified during *`Problem`* creation and is therefor not available during the *modeling phase*! While this may be unintuitive at first, it allows for a clean separation between component and system behavior on the one hand, and problem-specific imlementation details on the other. An important consequence of this is, that certain aspects of component behavior such as, e.g., ramping constraints must either be defined in a more general way (e.g., via limiting the derivative of the ramped quantity), or must be deferred and carried out after *`Problem`* creation (e.g. via appropriate callbacks).

Todo: Example for *Problem* creation

Todo: Examples for creating problem-specific constraints such as:

- Constraints that restrict cumulative quantities (time-integrals of quantities)
- Constraints that only consider a certain time range
- Constraints explicitly referencing quantities corresponding to multiple time-points and/or scenarios

Problem reformulation

It is possible to use manual or automated reformulations of the original problem formulation.

Todo: Example reformulations, *linearization* module and default discretization scheme in the interfaces.

1.2.3 Problem solution

Given a Problem, the user can chose to pass it directly to a solver capable of handling the corresponding problem type, transform the COMANDO Problem to a representation in an AML (currently Pyomo or GAMS), or work directly with the COMANDO representation in a custom algorithm to preprocess or solve the Problem. Currently available interfaces as well as their installarion requirements are described in *Interfaces to solvers and algebraic modeling languages (AMLs)*.

If a solution is obtained, it is loaded back into the COMANDO Problem after the solver terminates.

CHAPTER

TWO

INSTALLATION

Currently COMANDO is tested and should thus work with python 3.7 and 3.8. COMANDO itself only has `sympy` and `pandas` as its dependencies, however, while it can be used on its own for structural analysis, reformulation, or evauation of models and problems, actually solving optimization problems requires the use of an interface.

The available interfaces and their installation instructions can be found in *Interfaces to solvers and algebraic modeling languages (AMLs)*.

For the solution of optimization problems COMANDO provides interfaces to different solvers or relies on the interfaces provided by algebraic modeling languages (AMLs). Interfaces are either **text-based** (i.e., they allow for the generation of an input file for a solver or an AML) or **api-based** (i.e., they make use of a Python interface provided by the solver or AML).

Currently we provide the following interfaces.

- text-based:
  - BARON (solver)
  - GAMS (AML)
  - MAiNGO (solver)
- API-based:
  - Pyomo / Pyomo.DAE (AML)
  - Gurobi (solver)
  - MAiNGO (solver)

Until COMANDO is listed on pypi you have the following options to install COMANDO:

- Download the GitLab repository as a zip file
- clone the GitLab repository

Once you did that, you can install via

```
# from the parent directory of this repository...
pip install .
```

2.1 Uninstallation

```
# from anywhere
pip uninstall comando
```

CHAPTER

THREE

QUICKSTART GUIDE

This section contains code examples for the use of COMANDO

```
# With some energy system model ES = ...

design_objective = ...  # $F_1$ expression
operational_objective = ...  # $\dot{F}_2$ expression

P = ES.create_problem(design_objective,
                      operational_objective,
                      timesteps=timesteps,
                      name=f"min {'|'.join(objectives)}")

from comando.linearization import linearize

# Automatically linearize:
# - find all nonlinear expressions
# - evaluate them on an equidistant grid with 3 breakpoints per variable
# - create a triangulation of the expression values
# - encode the triangulation as linear constraints using the
#   convex-combination method
P_lin = linearize(P, n_bp=3, method='convex_combination')

# Use the Pyomo interface to interact with solvers
from comando.interfaces.pyomo import to_pyomo

# convert P to Pyomo 'model' objects
m = to_pyomo(P)
m_lin = to_pyomo(P_lin)

# Solve with open source solvers (need to be installed beforehand)...
res_lin = m_lin.solve('cbc')  # ...the MILP approximation globally
res = m.solve('ipopt')  # ...locally using a nonlinear solver
res_glob = m.solve('couenne')  # ...globally
```

CHAPTER

FOUR

INTERFACES TO SOLVERS AND ALGEBRAIC MODELING LANGUAGES (AMLS)

COMANDO itself allows to model energy systems and formulate optimization problems based on the resulting models. To solve these optimization problems, they need to be passed to a suitable solver. This can either be done by creating an appropriate input file and passing it to a solver or interfacing a solver's application programming interface (API) if present. Alternatively we can communicate the problem formulation to an algebraic modeling language (AML) - again via some input file or via API - which takes care of the interaction with the solver. The benefit of using an AML is that multiple solvers can be tested easily, a minor downside is that an additional representation (the one for the AML) needs to be created.

Both solver and AML interfaces are either text-based (creating input files) or use an API. Using any interface generally results in the following steps to be performed:

- translate a given problem to an appropriate representation
- instructs the solver to solve the problem with the given options
- reads back the solution from result files or API-objects
- updates the COMANDO variables with the optimal values

4.1 Available Solver interfaces

We provide the following interfaces:

4.1.1 GUROBI (API)

Problem classes:

- (MI)LP
- (MI)QP
- (MI)QCQP

Installation

Download and install the GUROBI solver (for academics you can create an account and obtain a free license for noncommercial use here)

From a commandline navigate to

- Linux: `/opt/gurobi902/linux64`
- Mac: `/Library/gurobi902/mac64`
- Windows `C:\gurobi\win64`

and run

```
python setup install
```

On Linux you may have to manually set additional paths (e.g. in `~/.bash_profile`)

```
export GUROBI_HOME="/opt/gurobi902/linux64"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

To check if everything worked properly, run python and do:

```
import gurobipy
```

4.1.2 BARON (Text-based)

Problem classes:

- (MI)LP
- (MI)QP
- (MI)QCQP
- (MI)NLP

4.1.3 SCIP (API)

Problem classes:

- (MI)LP
- (MI)QP
- (MI)QCQP
- (MI)NLP

Installation

Either download and install the prebuild binaries or download and compile the source from the SCIPOPT website.

Then install the pyscipopt python interface following these instructions.

4.1.4 MAiNGO (Text-based & API)

Problem classes:

- (MI)LP
- (MI)QP
- (MI)QCQP
- (MI)NLP

Installation

Using the MAiNGO interface requires the MAiNGO solver, which must currently build from source code. For this you will need CMake, compilers for C++11 and Fortran77.

An optional dependency is CPLEX as a MILP subsolver, alternatively the open source solver CBC is used.

To obtain the source code, we recommend using git:

```
git clone https://git.rwth-aachen.de/avt.svt/public/maingo.git
cd maingo
git submodule init
git submodule update -j 1
git submodule foreach git checkout master
git submodule foreach git pull
```

When using the commandline you can configure the cmake run via

```
mkdir build && cd build
ccmake ..
```

Now you may need to point CMake to the location of CPLEX if available. At this step you must activate the `MAiNGO_build_python_interface` switch if you want to use the API interface. Hit configure and generate to create the required files for building MAiNGO.

On Linux/Mac you can build the software with

```
make
```

On Windows you may use Visual Studio

The build process should produce the MAiNGO executable and pymaingo shared library `pymaingo.<your_python_version_number>-<your_platform>.<so OR pyd>` which needs to be added to the `PYTHONPATH` environment variable to be discoverable by python. For this add the following line to your `~/.bash_profile` or similar:

```
export PYTHONPATH="<PATH_TO...>/maingo/build:$PYTHONPATH"
```

To check if everything worked properly, run python and do:

```
import pymaingo
```

Usage

```
from comando.interfaces.maingo_api import MaingoProblem

mp = MaingoProblem(P)
solver, ret = mp.solve(epsilonA=1e-12, outstreamVerbosity=1)
# `solver` is a maingo::MAiNGO solver object that can be queried for
# solve-related information, also see `help(pymaingo.MAiNGO)`.
# ret is a maingo::RETCODE, also see `help(pymaingo.RETCODE)`.
```

4.2 Available AML interfaces

4.2.1 GAMS (Text-based)

The GAMS interface relies on the GAMS AML, which you can get (along with a trial license) here.

To use the COMANDO interface to GAMS, simply insyall the software and ensure it is found when running a commandline with

```
gams
```

Usage

```
# With a comando.Problem P...

from comando.interfaces.pyomo import to_pyomo

# NOTE: In contrast to us, pyomo refers to objects representing optimization
#       problems as 'models', hence 'm' is typically used!
m = to_pyomo(P)
# Now different locally installed solvers can be used, e.g., BARON (Requires
# standalone solver and license, not via GAMS!)
# The tee=True option results in the solver output to be displayed.
# Other, solver-specific options can be added, such as BARON's `MaxTime` here.
res = m.solve('baron', tee=True, MaxTime=300)
# res is a Pyomo results object, see the Pyomo documentation for details
```

4.2.2 Pyomo, Pyomo.DAE (API)

The Pyomo and Pyomo.DAE interfaces rely on the python package `pyomo`, which can be installed by running

```
pip install pyomo
```

Usage

```
# With a comando.Problem P...

from comando.interfaces.gams import solve

# Create a .gms file and pass it to GAMS.
# For a problem class other than MINLP, the `model_type` must be specified
# explicitly.
# Any additional GAMS options can be specified.
ret = solve(P, model_type='NLP', NLP='baron', optCR=1e-3)
# ret is the GAMS return code, also see:
# https://www.gams.com/latest/docs/UG_GAMSReturnCodes.html
if ret == 0:
    print('Εὕρηκα!')
elif ret == 7:
    raise RuntimeError("GAMS couldn't find a valid license!")
```

CHAPTER

FIVE

COMANDO PACKAGE

5.1 Subpackages

5.1.1 comando.interfaces package

Submodules

comando.interfaces.baron module

Input file generation for BARON.

`class comando.interfaces.baron.BaronParser(sym_map)`
Bases: *comando.utility.StrParser*

A class for parsing comando expressions to baron Syntax.

`comando.interfaces.baron.apply_cse(P, var_map)`
Replace reoccurring expressions with variables.

`comando.interfaces.baron.baron_pow_callback(parser, expr, idx)`
Handle special pow calls in BARON.

`comando.interfaces.baron.con_name(n=1, i=None)`
Generate n unused names of the form {prefix}{i} starting from i.

`comando.interfaces.baron.constraints_section(con_map, rel_only_cons, convex_cons, parse)`
Write the (RELAXATION_ONLY/CONVEX) EQUATIONS sections.

`comando.interfaces.baron.discretize(P, sym_map)`
Discretize differential equations in P using implicit Euler.

`comando.interfaces.baron.get_results(results_file_name='res.lst')`
Code for parsing baron results files.

`comando.interfaces.baron.get_times_and_bounds(baron_log_file)`
Code for parsing baron logs for time, and bounds.

`comando.interfaces.baron.handle_tanh(expr, tanh_definitions, var_map)`
Search the expression for occurrences of tanh and substitute them.

`comando.interfaces.baron.normalize(con)`
Bring constraints to a normal form baron can handle.

`comando.interfaces.baron.objective_section(P, parse)`

`comando.interfaces.baron.options_section(options)`
Write the OPTIONS section.

`comando.interfaces.baron.solve(P, file_name=None, silent=False, cse=False, reuse=None, **options)`
Solve the problem specified in the input_file with baron.

`comando.interfaces.baron.start_section(var_map)`

comando.interfaces.baron.var_name(*n=1*, *i=None*)
Generate n unused names of the form {prefix}{i} starting from i.

comando.interfaces.baron.variables_section(*var_map*, *prios=None*)
Write the (BINARY/INTEGER/POSITIVE) VARIABLES sections.

comando.interfaces.baron.write_bar_file(*P*, *file_name*, *options=None*, *cse=False*, *reuse=False*)
Write a baron input file for problem P.

comando.interfaces.gams module

Routines for translation to GAMS syntax.

class comando.interfaces.gams.GamsParser(*sym_map*)
Bases: *comando.utility.StrParser*

A class for parsing comando expressions to GAMS Syntax.

comando.interfaces.gams.gams_pow_callback(*parser*, *expr*, *idx*)
Handle special pow calls in GAMS.

comando.interfaces.gams.get_results(*results_file_name*)
Code for parsing gams results files.

comando.interfaces.gams.literal(*i*)
Get a gams representation for the literal index i in parentheses.

comando.interfaces.gams.populate_sym_map(*iname*, *dvars=()*, *ovars=()*, *pars=()*, *sym_map=None*)
Create a GAMS symbol map or populate an existing one with new symbols.

Parameters

- **dvars** (*iterable*) – design variables
- **ovars** (*iterable*) – operational variables
- **pars** (*iterable*) – parameters

Returns sym_map – Symbol map with GAMS representation for each passed symbol.

Return type dict

comando.interfaces.gams.solve(*P*, *input_file_name=None*, *silent=False*, ***options*)
Solve the problem specified in the input_file using GAMS.

Apart from the usual GAMS options, the model type (e.g. LP/MINLP) may be specified explixitly using the model_type option, the default is MINLP.

comando.interfaces.gams.write_equations_section(*P*, *iname*, *parse*)
Generate a string representation of a EQUATIONS section.

comando.interfaces.gams.write_gms_file(*P*, *file_name*, *model_type='MINLP'*)
Write a GAMS file based on the COMANDO Problem.

comando.interfaces.gams.write_objective(*P*, *parse*)
Create the string for the objective function of P.

comando.interfaces.gams.write_parameters_section(*P*, *iname*)
Generate a string representation of a PARAMETERS section.

comando.interfaces.gams.write_sets_section(*P*)
Generate a string representation of a SETS section.

comando.interfaces.gams.write_variables_section(*P*, *iname*)
Write the different 'VARIABLES' sections and variable bounds.

comando.interfaces.gurobi module

comando.interfaces.maingo_ale module

Code to generate an ALE input file for MAiNGO.

ALE is a library for Algebraic Logical Expressions that can be used to generate MAiNGO problems from human-readable input files. This module allows to generate such input files based on a COMANDO problem.

`class comando.interfaces.maingo_ale.AleParser(sym_map)`
Bases: *comando.utility.StrParser*

A class for parsing comando expressions to baron Syntax.

`comando.interfaces.maingo_ale.call_maingo(file_name, settings_name=None, silent=False)`
Call the maingo executable with a problem and possibly settings file.

`comando.interfaces.maingo_ale.get_results(results_file_name='MAiNGOresult.txt')`
Code for parsing MAiNGO results files.

`comando.interfaces.maingo_ale.maingo_pow_callback(parser, expr, idx)`
Handle special pow calls in MAiNGO.

`comando.interfaces.maingo_ale.solve(P, file_name=None, relaxation_only_constraints=None, squashing_constraints=None, silent=False, cse=True, outputs=None, add_intermediates_as_output=False, reuse=None, **options)`
Solve poblem P using MAiNGO.

`comando.interfaces.maingo_ale.write_ale_file(P, file_name, relaxation_only_constraints=None, squashing_constraints=None, cse=True, outputs=None, add_intermediates_as_output=False, reuse=False)`
Write the problem in ALE syntax to a file or stdout.

`comando.interfaces.maingo_ale.write_settings_file(options, settings_name='MAiNGOSettings.txt')`
Generate a settings file with the given options.

comando.interfaces.maingo_api module

comando.interfaces.pyomo module

comando.interfaces.pyomo_dae module

comando.interfaces.scip module

Module contents

Solver and algebraic modeling language interfaces for COMANDO.

5.2 Submodules

5.2.1 comando.core module

Package for generic modeling of energy system design and operation.

`class comando.core.Component(label)`

Bases: `object`

A component representing a part of an energy system.

The Component class is a model of a generic real-world component of an energy system, represented by a collection of algebraic and logical expressions which describe how the component can be controlled, its limitations and its interactions with other components.

For every component, two kinds of decisions may be taken: - design decisions a.k.a. 'here-and-now' decisions - operational decisions a.k.a. 'wait-and-see' decisions The former constitute decisions that are taken once, prior to all operation while the operational decisions need to be taken for every time-point under consideration.

The component may define named expressions that can be accessed at any time for informational purposes. These expressions may also be used to impose additional constraints at a later stage, or to aggregate information from multiple different components. In a similar way an objective for a system optimization can be generated by aggregating different types of costs defined by some subset of the system's components.

Parameters that are contained in the various expressions may or may not be given a default value that can be changed at a later stage.

Variables

- ***label*** (*str*) – A unique sting that serves as an identifier
- ***parameters*** (*set* of *Symbol*) – Set of unspecified system parameters the user can assign values to.
- ***design_variables*** (*set* of *Variable*) – Set of variables of the design stage, i.e. time-independent variables such as the number or size of newly added components.
- ***operational_variables*** (*set* of *Variable*) – Set of unique names for variables of the operational stage, e.g. the output of a given component or an operational state.
- ***states*** (*dict*) – Dictionary mapping a subset of operational_variables to automatically created Variables, representing the corresponding time derivatives. Examples are the SOC of a battery, the fill-level of a tank, etc.
- ***constraints*** (*dict* of *sympy.core.relational.Relational*) – Dictionary of relational sympy expressions that represent constraints.
- ***expressions*** (*dict*) – mapping of shared identifiers to algebraic sympy expressions. The expressions different *Component* instances associate with a shared identifier may be aggregated to additional constraints or objectives.
- `connectors` (*dict*) – mapping of strings to algebraic expressions representing in or outputs to the component. When multiple connectors are connected to each other, the connection represents the requirement that the sum of their expressions need to be equal to zero.

`add_connector(id, expr)`

Insert the connector in connectors and add it as an attribute.

`add_connectors(id=None, expr=None, **connectors)`

Add one or more named connectors to this component.

Examples

```
>>> self.add_connectors('A', sympy.S('A'))
>>> self.add_connectors(B=sympy.S('B'), C=sympy.S('C'))
```

`add_eq_constraint(`*lhs_expr*, *rhs_expr*, *name=None*`)`
Add a constraint of the form lhs_expr = rhs_expr.

`add_expression(`*identifier*, *expr*`)`
Add a named algebraic expression.

`add_ge_constraint(`*lhs_expr*, *rhs_expr*, *name=None*`)`
Add a constraint of the form lhs_expr ≥ rhs_expr.

`add_input(`*identifier*, *expr*`)`
Add a connector that corresponds to an input into the component.

The expr is assumed to always be positive and is thus bounded by 0 from below. By convention the input into a Component is positive so the new Connector's expression corresponds to expr.

`add_le_constraint(`*lhs_expr*, *rhs_expr*, *name=None*`)`
Add a constraint of the form lhs_expr ≤ rhs_expr.

`add_output(`*identifier*, *expr*`)`
Add a connector that corresponds to an output from the component.

The expr is assumed to always be positive and is thus bounded by 0 from below. By convention the output from a Component is negative so the new Connector's expression corresponds to the negated expr.

`property constraints`
Get a set of the Component's constraints.

`property constraints_dict`
Get a dictionary of the Component's constraints.

`declare_state(`*var*, *rate_of_change=None*, *init_state=None*, *der_bounds=None, None*, *der_init_val=None*`)`
Declare var to be a state.

A state is an operational variable whose time derivative is described by the rate_of_change expression, beginning at an initial_state.

`property design_variables`
Get a set of the Component's design_variables.

`property design_variables_dict`
Get a dictionary of the Component's design_variables.

`existing_components = {}`

`property expressions`
Get a set of the Component's expressions.

`property expressions_dict`
Get a dictionary of the Component's expressions.

`get_expression(`*identifier*, *default=None*`)`
Get the expression corresponding to the identifier.

`property label`
Get the Component's unique label.

`make_design_variable(`*name*, *domain=<Domain.REAL: 1>*, *bounds=(None, None)*, *init_val=None*`)`
Create a design variable with a localized name & store it.

`make_operational_variable(`*name*, *domain=<Domain.REAL: 1>*, *bounds=(None, None)*, *init_val=None*`)`
Create an operational variable with a localized name & store it.

`make_parameter(`*name*, *value=None*`)`
Create a parameter with a localized name & store it.

make_state(*name*, *rate_of_change=None*, *init_state=None*, *domain=<Domain.REAL: 1>*, *bounds=(None, None)*, *der_bounds=(None, None)*, *init_val=None*, *der_init_val=None*)
Create a state with a localized name, its derivative and store them.

A state is an operational variable whose time derivative is described by the rate_of_change expression, beginning at an initial_state.

property operational_variables
Get a set of the Component's operational_variables.

property operational_variables_dict
Get a dictionary of the Component's operational_variables.

property parameters
Get a set of the Component's parameters.

property parameters_dict
Get a dictionary of the Component's parameters.

property states

property states_dict

class comando.core.Connector(*component*, *name*, *expr*)
Bases: object

An interface used to connect one or more *Component* objects.

class comando.core.ConnectorUnion(*component*, *name*, **connectors*)
Bases: *comando.core.Connector*

A union of multiple Connectors.

property expr
Return an expression for the flow through the ConnectorUnion.

class comando.core.DataProxy(*getter: Callable[Optional[object]]*, *setter: Callable[str, [int, float]]*)
Bases: comando.SlotSerializationMixin

A proxy object to access and set data.

Callable = typing.Callable

Optional = typing.Optional

getter

setter

exception comando.core.ImpossibleConstraintException
Bases: Exception

class comando.core.Problem(*design_objective=0*, *operational_objective=0*, *constraints=None*, *states=None*, *timesteps=None*, *scenarios=None*, *data=None*, *name='Unnamed Problem'*)
Bases: object

A simple optimization problem.

property T
Get the Problem's end time.

add_symbols(*syms*)
Sort symbols into parameters, design- and operational variables.

property data
Aggregate and return the parameter data.

property design
Collect the design variable values in a DataFrame.

property design_objective

get_constraint_violations(*larger_than=0*)
Collect the current constraint violations in a DataFrame.

property index
Get the index of the Problem.

load_variable_values(*filename*)
Collect variable values from file.

property num_cons
Get the total number of variables.

property num_vars
Get the total number of variables.

property objective
Get the objective expression of the problem.

property operation
Collect the operational variable values in a DataFrame.

property operational_objective

property scenario_weights
Get the Problem's scenario weights.

property scenarios
Get the Problem's scenarios.

store_variable_values(*filename*)
Serialize current variable values and store in file.

subs(*sym=None*, *rep=None*, ***reps*)
Substitute an individual or multiple symbols in the problem.

property timesteps
Get the length of the Problem's timesteps.

weighted_sum(*op_expr*, *symbolic=True*)
Compute the scenario- and/or time-weighted sum of an expression.

Parameters

- **op_expr** (*Expression*) – an operational expression that is to be weighted
- **symbolic** (*bool*) – if True creates a new expression (default) if False evaluate numerically

class comando.core.System(*label*, *components=None*, *connections=None*)
Bases: *comando.core.Component*

A class for a generic system, made up of individual components.

Note that a system is itself a component and can therefore function as a subsystem for a larger system, allowing for nested structures!

Variables

- *components* (iterable of *Component*) – components that form part of the considered energy system
- connections (*dict*) – mapping of *str* to an iterable of *Connector*. The in- and outputs of all connectors within an iterable are assumed to balance each other.

add(*component*)
Add a *Component* to the system.

aggregate_component_expressions(*id*, *aggregator=<built-in function sum>*)
Aggregate expressions from the components matching the identifier.

The passed identifier is used to look up matching expressions in the components of this system. The resulting expressions are then aggregated using the passed aggregator function and the resulting expression is returned.

`close_connector(`*connector*, *expr=0*`)`
Specify the flow over the connector (default 0).

`property components`
Get the set of components that are part of the system.

`connect(`*bus_id*, *connectors*`)`
Connect all elements of *connectors* to a bus with id *bus_id*.

`property constraints`
Get a set of the System's constraints.

`property constraints_dict`
Get a dictionary of the System's constraints.

`create_problem(`*design_objective=0*, *operational_objective=0*, *timesteps=None*, *scenarios=None*, *data=None*, *name=None*`)`
Create a problem with the specified time and scenario structure.

Parameters

- **T** (*The end time of the operational period (default 8760)*) – If T is not specified, timesteps needs to be a mapping (see below).
- **timesteps** (*tuple, Mapping or Series.*) – If timesteps is tuple it is assumed to consist of timestep labels and data for the time horizon T. This data may be a scalar numeric value, a Mapping or a Series. In the latter two cases T maps from different scenarios s to corresponding time horizons T[s]. If timesteps is a Mapping, it can either be mapping from timestep labels to timestep lengths or from scenarios to a corresponding specification of the time structure, consisting of either the tuple representation or the timstep mapping.
- **scenarios** (*None or an iterable of timestep labels.*) – If scenarios is a Mapping or pandas Series, the values are interpreted as the probabilities / relative likelyhoods of the individual scenarios.

`property design_variables`
Get a set of the System's design_variables.

`property design_variables_dict`
Get a dictionary of the System's design_variables.

`detach(`*bus_id*, *connectors=None*`)`
Detach all *Connector`s in `connectors* from the specified bus.

`expose_connector(`*connector*, *alias=None*`)`
Expose an existing connector to the outside of the system.

`property expressions`
Get a set of the System's expressions.

`property expressions_dict`
Get a dictionary of the System's expressions.

`extend_connection(`*bus_id*, *alias=None*`)`
Extend the connection to the outside of the system.

`get_open_connectors()`
Return the set of all connectors that are not connected yet.

`property operational_variables`
Get a set of the System's operational_variables.

`property operational_variables_dict`
Get a dictionary of the System's operational_variables.

property parameters
Get a set of the System's parameters.

property parameters_dict
Get a dictionary of the System's parameters.

remove(*component*)
Remove a *Component* from the system.

property states
Get a set of the System's states.

property states_dict
Get a set of the System's states.

comando.core.is_trivial(*constraint*, *con_repr=None*, *warn=True*)
Handle constraints that are trivially true or false.

5.2.2 comando.linearization module

Methods to generate (possibly mixed-integer) linear problem formulations.

class comando.linearization.Namer(*prefix*, *i_0=0*)
Bases: object

Generator for names of the form {prefix}{i}.

get()
Get the next generated name.

exception comando.linearization.NoGloverReformulationPossible
Bases: Exception

comando.linearization.cached(*reformulator*)
Equip the *reformulator* function with a cache.

comando.linearization.glover_reformulation(**args*)

comando.linearization.linearize(*P*, *n_bp*, *method*)
Return a (possibly mixed-integer) linear approximation of problem P.

In the linearization process, all expressions that will form part of an optimization problem will be analyzed and reformulated into linear forms if necessary. In this process of reformulation new variables and constraints may need to be introduced. To refrain from modifying the passed energy system instance the new set of variables, a dictionaty of constraints and an objective expression are generated and returned.

Parameters

- **P** (*comando.Problem*) – The problem to be linearized
- **n_bp** (*number*) – the number of breakpoints used per variable
- **method** (*str*) – method used to encode the triangulation over the generated simplices currently either 'convex_combination' or 'multiple_choice'.

Returns P_lin – A linear reformulation of problem P possibly with additional variables and constraints.

Return type comando.Problem

comando.linearization.linearize_expr(**args*)

comando.linearization.piecewise_linear(*points*)
Approximate a univariate function via a discrete mapping.

compute the axis intersects y0_i and slopes dydx_i for the linear segments connecting the given points in sorted order.

Parameters points (*Mapping*) – mapping from input to output values

Returns

- **x** (*list*) – sorted list of x values
- **y0** (*list*) – len(points) - 1 axis intersections for line segments
- **dydx** (*list*) – len(points) - 1 slopes for line segments

`comando.linearization.piecewise_linear_reformulation(*args)`

5.2.3 comando.utility module

Utility functions used in various parts of COMANDO.

`class comando.utility.DefaultStringMap`
Bases: `collections.defaultdict`

A default dict implementation whose factory is called with the key.

`exception comando.utility.RootFindingError`
Bases: `Exception`

`class comando.utility.StrParser(sym_map, str_map=None, override_default_str_map=True, add_callback=None, mul_callback=None, pow_callback=None)`
Bases: `object`

A class for creating interface-specific string representations.

Callbacks must have a signature consisting of three arguments: The parser the expression to be parsed and an optional index Arguments: ==========

sym_map str_map add_callback mul_callback pow_callback

`cached_parse(expr, idx=None)`
Look up previously parsed expressions in self.cache.

`parse_args(args, idx, parent_prec=0)`
Parse arguments of an expression with parentheses as appropriate.

`comando.utility.as_numer_denom(expr)`
Normalize as_numer_denom for sympy and symengine.

Also see: https://github.com/symengine/symengine/issues/681#issuecomment-724039779

`comando.utility.bEq(x, y)`
Return the truth value of x == y, if it can be determined, else None.

`comando.utility.bGe(x, y)`
Return the truth value of x >= y, if it can be determined, else None.

`comando.utility.bLe(x, y)`
Return the truth value of x <= y, if it can be determined, else None.

`comando.utility.bounds(expr)`
Propagate variable bounds through the given expression.

`comando.utility.bounds_cost_turton(x, c1, c2, c3)`
Compute bounds for the the Guthrie cost function.

cost_turton(x, c1, c2, c3) = pow(10, c1 + c2 * log10(x) + c3 * log10(x) ** 2)

`comando.utility.canonical_file_name(base_name, suffix, file_name=None) -> (<class 'str'>, <class 'str'>)`
Get a canonical file name and the corresponding base name.

Parameters

- **base_name** (*str*) – the fallback if filename is None
- **suffix** (*str*) – desired suffix

- **file_name** (*str or None*) – a desired name with or without the desired suffix or None

Returns

- **base_name** (*str*) – the canonical base_name
- **file_name** (*str*) – the canonical file_name

`comando.utility.check_reuse_or_overwrite(`*file_name*, *reuse*`)` → None
Check whether a file exists for reuse or overwriting.

Parameters

- **file_name** (*str*) – the name of the file to reuse or overwrite
- **reuse** (*bool or None*) – whether to reuse the given file. There are three cases:
  1. reuse is False -> do nothing
  2. reuse is True -> check if file_name is a valid file
  3. reuse is None -> if file_name is a file, ask before overwriting

`comando.utility.cleanup()`
Take a snapshot of all files on current path and remove new ones.

`comando.utility.cont_univ_bounds(`*expr*, *var*`)`
Compute tight bounds for bounded continuous univariate expressions.

A bounded continuous expression of a single variable can be bounded exactly by looking at its function values at 'critical points', i.e., the union of its bounds and the points at which its derivative is zero that are in the interval implied by the bounds.

`comando.utility.define_function(`*name*, *implementation*`)`
Define a new symbolic function with a name and an implementation.

The given implementation will be used when the function is called with nonsymbolic arguments only.

Parameters

- **name** (*str*) – the function's name
- **implementation** (*callable*) – a callable that serves as the numerical implementation and is used when no symbols are within the arguments. Must have a fixed number of positional arguments.

Returns **new_function** – the newly defined function

Return type sympy.FunctionClass

`comando.utility.depth(`*expr*`)`
Compute the depth of the given expression.

`comando.utility.evaluate(`*expr*, *idx=None*`)`
Evaluate the given expression at the symbols' current values.

`comando.utility.floor_substitute_bounds(`*x*, *LB*, *UB*`)`

`comando.utility.fuzzy_not(`*v*`)`
Return None if *v* is None else *not v*.

`comando.utility.get_index(`*expr*`)`
Get the index of the current expression or None if it is not indexed.

`comando.utility.get_latest(`*path_glob*`)`
Get the path to the latest file matching the path_glob.

`comando.utility.get_pars(`*expr*`)`
Get a set of all parameters in *expr*.

`comando.utility.get_previous(`*state*, *initial_state*`)`
Create a VariableVector for the state at previous timesteps.

comando.utility.get_type_name(*expr*)
Get the type name of an expression.

comando.utility.get_vars(*expr*)
Get a set of all variables in *expr*.

comando.utility.identity(*expr*)
Return expr.

comando.utility.implicitEuler(*P*)
Discretize differential equations in P using implicit Euler.

comando.utility.indexed(*expr*)
Test whether the given expression is indexed.

comando.utility.is_negated(*expr*)
Check if expr is a negated expression or a negative number.

comando.utility.lambdify(*expr*, *vars=None*, *eval_params=True*)
Create a function for evaluation of expr with numpy.

A variant of sympy's lambdify which creates a function for fast numerical evaluation of an expression.

Parameters

- **expr** (*Expression*) – the expression to be turned into a function
- **eval_params** (*bool*) – Whether to replace parameters by their values (if expr contains indexed parameters, evaluating the function will thus return a Series) default: True

Returns **f** – a function returning the value of the expression for given variable values

Return type callable

comando.utility.lmtd_bounds(*dT1*, *dT2*)
Calculate the logarithmic mean temperature difference.

lmtd(dT1, dT2) = (dT1 - dT2) / log(dT1 / dT2)

comando.utility.make_function(*expr*)
Create a callable function from the given expression.

The user can either specify values for all Variables in the order specified by the returned function's '__doc__' (in alphabetical order), or pass keyword arguments in the form of identifier-value pairs. The values are taken as alternatives for the default values used in the evaluation for the Variables or Parameters with matching identifiers.

comando.utility.make_mayer_objective(*system*)
Create a Mayer term objective.

An option for dynamic optimization phi(t_f) is minimized and phi_dot = f(operation)

comando.utility.make_str_func(*name*)
Create a default string representation of functions.

Parameters **name** (*str*) – name of the function to be represented

Returns **func** – function that returns a string representation for the function with given string arguments

Return type callable

comando.utility.make_tac_objective(*system*, *ic_labels=None*, *fc_labels=None*, *vc_labels=None*, *n=10*, *i=0.08*)
Create the objective components corresponding to total annualized cost.

Parameters

- **ic_labels** (iterable of *str*) – labels for the investment cost expressions
- **fc_labels** (iterable of *str*) – labels for the fixed cost expressions

- **vc_labels** (iterable of *str*) – labels for the variable cost expressions
- **n** (*int*) – number of repetitions of the time-period specified by *timesteps*.
- **i** (*float*) – interest_rate

Returns

- *Expressions for design_objective and operational_objective forming part*
- *of the total annualized costs.*

`comando.utility.mul_bounds(a, b)`
Return the result of a * b by interval arithmetic.

`comando.utility.parse(expr, sym_map=None, op_map=None, idx=None, const_fun=<class 'float'>)`
Parse the sympy expression *expr*, replacing symbols and operators.

The *sym_map* provides the possibility to replace symbols (Parameters and Variables) with new objects, if the symbol map is empty or there is no mapping for a given symbol, it is replaced with its *value* attrubute; if *value* is *None*, the symbol is not replaced. The *op_map* provides the possibility to replace sympy's operator objects with other operators. If no op map is provided, the default python operators are used.

`comando.utility.pow_bounds(a, b)`
Return the result of a ** b if a[0] == a[1] or b[0] == b[1].

The power function is monotonous on any domain that excludes zero! We therefore only need to consider special cases explicitly and can return the sorted tuple of powers of the base boundaries otherwise.

`comando.utility.precedence(expr)`
Get the operator precedence of the expression.

If the argument of an expression has a higher precedence than the expression itself, precedence needs to be represented explicitly in string representations. An example is (a + b) * c, where dropping the parentheses changes the result. We define precedences as:

±x, f(x): 0 x ** y: 1 x * y, x/y: 2 x ± y: 3

`comando.utility.prod(*args)`
Return the product of the elements in args.

`comando.utility.prod_bounds(*bounds)`
Return the product of a sequence of intervals.

`comando.utility.rlmtd_bounds(dT1, dT2)`
Calculate bounds for the inverse of the LMTD.

rlmtd(dT1, dT2) = log(dT1 / dT2) / (dT1 - dT2)

`comando.utility.silence()`
Execute code without output to stdout.

`comando.utility.smooth_abs(x, delta=0.0001)`
Smooth approximation of abs(x).

`comando.utility.smooth_max(a, b, delta=0.0001)`
Smooth approximation of max(a, b).

`comando.utility.smooth_min(a, b, delta=0.0001)`
Smooth approximation of min(a, b).

`comando.utility.split(iterable, condition, c_type=None)`
Sort *iterable*'s elements into two containers based on *condition*.

Parameters

- **iterable** (*iterable*) – iterable to be sorted
- **condition** (*callable*) – function used for sorting, return value must convert to bool

- **c_type** (*type*) – type of the resulting containers; if unset will attempt to use *type(iterable)* to create the containers or default to *list* if that fails. If after this both *isinstance* and *c_type()* are mappings, the new containers will be submappings of *iterable* and *condition* is only applied to the values of *iterable*.

Returns **cond_true, cond_false** – tuple containing: - the container of elements that do **not** satisfy *condition* - the container of elements that do satisfy *condition*

Return type tuple

`comando.utility.str_parse(`*expr*, *sym_map=None*, *str_map=None*, *idx=None*`)`
Parse the sympy expression *expr*, replacing symbols and operators.

The *sym_map* provides the possibility to replace symbols (Parameters and Variables) with their string representations, if the symbol map is empty or there is no mapping for a given symbol, it is replaced with its *value* attribute. The *str_map* provides the possibility to replace sympy's operator objects with string representations.

`comando.utility.sum_bounds(`**bounds*`)`
Return the sum of a sequence of intervals.

`comando.utility.syscall(`*executable*, **args*, *log_name='/dev/null'*, *silent=False*`)`
Issue a system call.

Parameters

- **executable** (*str*) – the name of the executable to be called. Needs to be an executable in one of the directories listed in the PATH environment variable.
- **args** (*list of str*) – the arguments passed to the executable
- **log_name** (*str*) – the location of a file to which the output of the system call is mirrored in parralel to execution (default: os.devnull)
- **silent** (*bool*) – whether the output should be hidden from stdout (default: False)

Returns **ret** – the return code of the system call

Return type int

5.2.4 comando.visualization module

Collection of routines for visualizing data, expressions and results.

`comando.visualization.convergence_plot(`*times*, *lb_data*, *ub_data*, *color='b'*, *label=''*`)`
Make a simple performance plot.

`comando.visualization.plot_expr(`*expr*, *label=None*, *axis=0*, *fig=None*, *cutoff=10000000000.0*, *x=None*, *kind='surface'*, ***kwargs*`)`
Plot an expression.

`comando.visualization.plot_incidence(`*exprs*, *vars=None*`)`
Visualize which variables are contained in which expression.

5.3 Module contents

Configuration of the backends for COMANDO.

class comando.Domain(*value*)
Bases: enum.Enum

Simple Enum for variable domains, specify other types via bounds.

BINARY = 3

INTEGER = 2

REAL = 1

class comando.Symbol(*name*, ***assumptions*)
Bases: comando.SlotSerializationMixin, sympy.core.symbol.Symbol

A placeholder for a value which can occur within expressions.

property indexed
Check if the Symbol is indexed.

property is_Parameter

property is_Variable

property value
Get the Symbol's value.

comando.cartes
alias of itertools.product

comando.prod(**args*)
Return the product of the elements in args.

class comando.Parameter(*name*, *value=nan*, *parent=None*)
Bases: *comando.Symbol*

A *Symbol* representing a parameter whose value is known.

property elements

expand(*data*)
Expand the *Parameter* with indexed data.

expansion

property indexed
Check whether the Parameter is indexed or not.

property indices

property is_Parameter

property items

property parent
Return the parent of this parameter.

property value
Return the value or values of the Parameter.

class comando.Variable(*name*, *domain=<Domain.REAL: 1>*, *bounds=(None, None)*, *init_val=None*, *indexed=False*, *parent=None*)
Bases: *comando.Symbol*

A *Symbol* representing a variable whose value is unknown.

property bounds

property domain

`fix(value=None)`
Fix the variable by setting both bounds to *value*.

`property init_val`

`property is_Variable`

`property is_binary`

`property is_integer`

`property is_negative`
Check if all possible values of the variable are negative.

We can assert negativity if the upper bound is negative, otherwise we can assert nonnegativity if the lower bound is nonnegative. If we cannot assert either of these facts, the variable may contain both positive and negative values. To reflect this we return None.

`property is_nonnegative`
Check if all possible values of the variable are negative.

This is the fuzzy not of self.is_negative

`property is_nonpositive`
Check if all possible values of the variable are negative.

This is the fuzzy not of self.is_negative

`property is_positive`
Check if all possible values of the variable are positive.

We can assert positivity if the lower bound is positive, otherwise we can assert nonpositivity if the upper bound is nonpositive. If we cannot assert either of these facts, the variable may contain both positive and negative values. To reflect this we return None.

`property lb`

`property parent`
Return the parent of this variable.

`property ub`

`unfix()`
Recover the original bounds.

`property value`
Get the Symbol's value.

`class comando.VariableVector(name, domain=<Domain.REAL: 1>, bounds=(None, None), init_val=None)`
Bases: *comando.Symbol*

A *Symbol* representing a vector of *Variables*.

`property bounds`

`property domain`

`property elements`

`expansion`

`fix(value=None)`
Fix *self.elements* by setting both bounds to *value*.

`property indexed`
Check if the Symbol is indexed.

`property indices`

`property init_val`

`instantiate(`*index*`)`
Create a *Variable* instance for every element in *index*.

`property is_binary`

`property is_expanded`

`property is_integer`

`property is_negative`
Check if all possible values of the variable are negative.

We can assert negativity if the upper bound is negative, otherwise we can assert nonnegativity if the lower bound is nonnegative. If we cannot assert either of these facts, the variable may contain both positive and negative values. To reflect this we return None.

`property is_nonnegative`
Check if all possible values of the variable are negative.

This is the fuzzy not of self.is_negative

`property is_nonpositive`
Check if all possible values of the variable are negative.

This is the fuzzy not of self.is_negative

`property is_positive`
Check if all possible values of the variable are positive.

We can assert positivity if the lower bound is positive, otherwise we can assert nonpositivity if the upper bound is nonpositive. If we cannot assert either of these facts, the variable may contain both positive and negative values. To reflect this we return None.

`property items`

`property lb`

`property ub`

`unfix()`
Recover the original bounds.

`property value`
Get the Symbol's value.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W