Panobbgo Documentation

Release 0.0.1pre

Harald Schilly

July 09, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

Warning: It is currently work in progress and definitely not ready for any kind of usage.

Panobbgo is an open-source framework for parallel noisy black-box global optimization. The primary aim is to experiment with new ideas and algorithms. A couple of functional building blocks build the solver and exchange information via an `EventBus` among each other. This allows to rapidly prototype new modules and to combine them with existing parts. There are three basic types of parts that work together:

- the `Strategy`
- several `Heuristics`
- and `Analyzers`

Various tools for extracting statistical data and inspecting the optimization process are included (*planned*). Additionally, parallel evaluation of the objective black-box function can be archived as SMP or on a cluster via IPython [IP].

In the background, there are additional utility features for the configuration and dependency management (*planned*) available.

CHAPTER

TWO

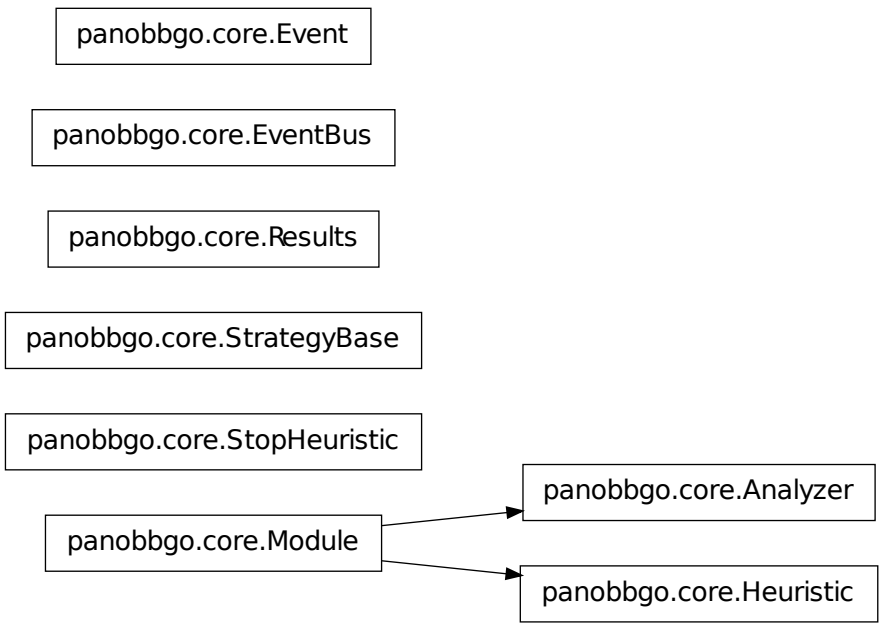
MAIN

This is the main part of Panobbgo.

2.1 Core

This is the core part. It contains the essential components and base-classes for the modules:

- `Results`: Database of all results, with some rudimentary queries and statistics.
- `EventBus`: This is the backbone for communicating between the strategy, the heuristics and the analyzers.
- "abstract" base-classes for the modules
  - `heuristics`
  - `analyzers`.
- and most importantly, the `StrategyBase` which holds everything together and subclasses in `strategies` implement the actual strategies.



class panobbgo.core.**Analyzer**(*name=None*)
Bases: panobbgo.core.Module

Abstract parent class for all types of analyzers.

class panobbgo.core.**Event**(***kwargs*)
Bases: object

This class holds the data for one single EventBus event.

class panobbgo.core.**EventBus**
Bases: object

This event bus is used to publish and send events. E.g. it is used to send information like "new best point" to all subscribing heuristics.

keys
List of all keys where you can send an Event to.

publish(*key*, *event=None*, *terminate=False*, ***kwargs*)
Publishes a new Event to all subscribers, who listen to the given key. It is either possible to send an existing event or to create an event object on the fly with the given **kwargs.

Args:

- event: if set, this given Event is sent (and not a new one created).
- **terminate: if True, the associated thread will end.** (use it for on_start and similar).
- ****kwargs: any additional keyword arguments are stored inside the Event** if event is None.

re = <module 're' from '/usr/lib/python2.7/re.pyc'>

register(*target*)
Registers a given target for this EventBus instance. It needs to have suitable on_<key> methods. For each of them, a Thread is spawn as a daemon.

subscribe(*key*, *target*)
Called by register().

Note: counterpart is unsubscribe().

---

unsubscribe(*key*, *target*)
Args:

- if key is None, the target is removed from all keys.

class panobbgo.core.**Heuristic**(*name=None*, *cap=None*)
Bases: panobbgo.core.Module

This is the "abstract" parent class for all types of point generating classes, which we call collectively "Heuristics".

Such a heuristic is capable of the following:

1. They can be parameterized by passing in optional arguments in the constructor. This should be reflected in the name!
2. The EventBus spawns a thread for each on_* method and calls them when a corresponding Event occurs.
3. Of course, they are capable of storing their state in the instance. This is also the way of how information is shared between those threads.
4. The *main purpose* of a heuristic is to emit new search points by calling either emit() or returning a list of points. The datatype must be numpy.ndarray of floats.
5. Additionally, the can get hold of other heuristics or anayzers via the strategy instance.

6. The `EventBus` inside this strategy instance allows them to publish their own events, too. This can be used to signal related heuristics something or to queue up tasks for itself.

active
This is queried by the strategy to determine, if it should still consider it. This is the case, iff there is still something in its output queue or if there is a chance that there will be something in the future (at least one thread is running).

clear_output()

emit(*points*)
This is used to send out new search points for evaluation. Args:

- `points`: Either a `numpy.ndarray` of `float64` or preferrably a list of them.

get_points(*limit=None*)
this drains the output Queue until `limit` elements are removed or the Queue is empty. For each actually emitted point, the performance value is discounted (i.e. "punishment" or "energy consumption")

class `panobbgo.core.`**Module**(*name=None*)
Bases: `object`

"Abstract" parent class for various panobbgo modules, e.g. `Heuristic` and `Analyzer`.

eventbus

name
The module's name.

Note: It should be unique, which is important for parameterized heuristics or analyzers!

---

problem

results

strategy

ui

class `panobbgo.core.`**Results**(*strategy*)
Bases: `object`

A very simple database of results with a notificaton for new results. The new results are fed directly by the `StrategyBase`, outside of the `EventBus`.

Note: Later on, maybe this will be a cool actual database which allows to persistenly store past evaluations for a given problem. This would allow resuming and further a-posteriory analysis.

---

add_results(*new_results*)
Add one single or a list of new @Result objects. Then, publish a `new_result` event.

info()

n_best(*n*)

exception `panobbgo.core.`**StopHeuristic**(*msg='stopped'*)
Bases: `exceptions.Exception`

Indicates the heuristic has finished and should be ignored/removed.

Args:

- `msg`: a custom message, will be visible in the log. (default: "stopped")

class `panobbgo.core.`**StrategyBase**(*problem*, *heurs*)
Bases: `object`

This abstract BaseStrategy is the parent class of all Strategies.

Use it this way:

1. Subclass it, write your optional initializer, *afterwards* call the initializer of this class (it will start its the main loop).
2. Overwrite the `execute()`, which returns a list of new search points (by requesting them from the `heuristics` via the `get_points()` method) and might also emit `Events`.

This `execute` method will be called repeatedly as long as there are less than the given maximum number of search points evaluated.

PROBLEM_KEY = 'problem'

add_analyzer(*a*)

add_heuristic(*h*)

analyzer(*who*)

avg_time_per_task

best

execute()
Overwrite this method when you extend this base strategy.

heuristic(*who*)

heuristics

info()

init_module(*module*)
`StrategyBase` calls this method.

name

run()

time_cpu
effective cpu time in seconds

time_start_str

time_wall
wall time in seconds

2.2 Strategies

This part outlines the coordination between the point-producing heuristics, the interaction with the cluster and the `DB of evaluated points`.

Basically, one or more threads produce points where to search, and another one consumes them and dispatches tasks. Subclass the `StrategyBase` class to implement a new strategy.

2.3 Heuristics

The main idea behind all heuristics is, ...

Each heuristic needs to listen to at least one stream of `Events` from the `EventBus`. Most likely, it is the *one-shot* event `start`, which is `published` by the `StrategyBase`.

2.4 Configuration

It's purpose is to parse a config file (create a default one if none is present) and replace values stored within it with those given via optional command-line arguments.

Note: This will also hold a class for configuring the Panobbgo framework in general. I.e. modules declare other modules as dependencies, etc...

---

class `panobbgo.config.`**Config**
Bases: `object`

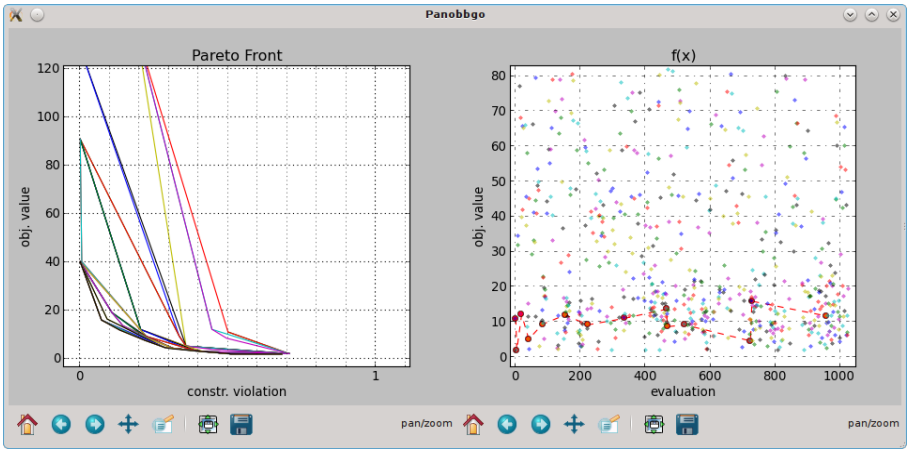
debug

get_logger(*name*, *loglevel=None*)

`panobbgo.config.`**get_config**()

2.5 User Interface

This draws a window and plots graphs.



class `panobbgo.ui.`**UI**
Bases: `panobbgo.core.Module`, `gtk.Window`, `threading.Thread`

UI

add_notebook_page(*label_text*, *frame*)

destroy(*win*)

finish()
called by base strategy in _cleanup for shutdown

static mk_canvas()
Creates a FigureCanvas, ready to be added to a gtk layout element

redraw_canvas(*c*)
If your canvas needs to be redrawn, pass it into this function.

run()

show()

CHAPTER

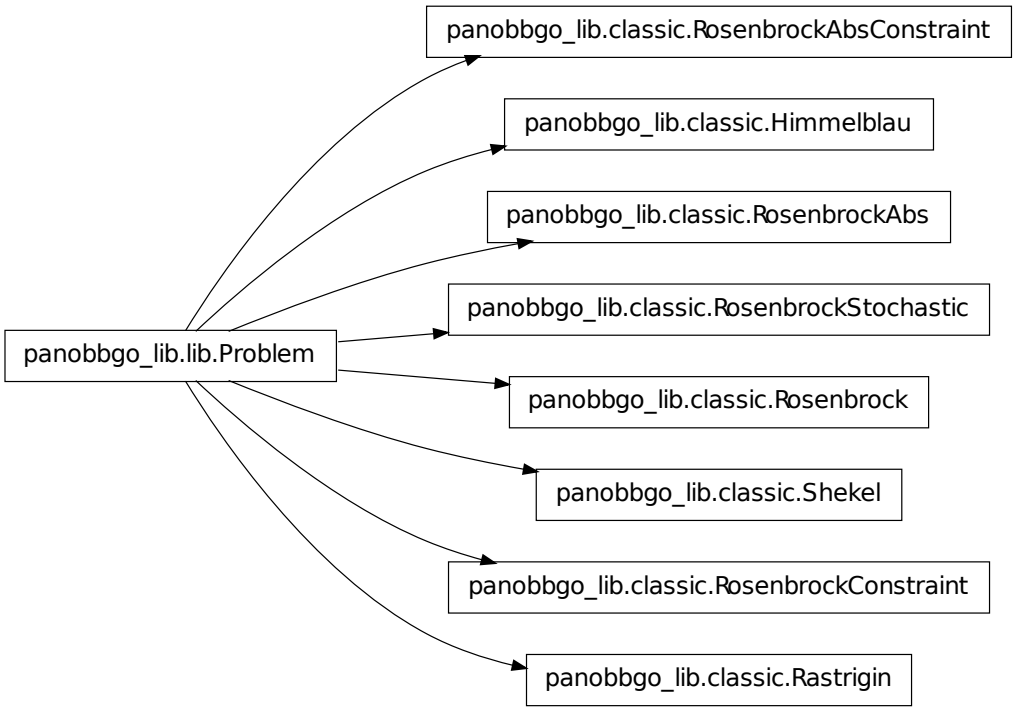
THREE

LIBRARY

This is the *library* of Panobbgo, used by the main part and the library. For example, the basic `Problem` class is defined here.

3.1 Classic Problems

This file contains the basic objects to build a problem and to do a single evaluation.



class `panobbgo_lib.classic.`**`Himmelblau`**

Bases: `panobbgo_lib.lib.Problem`

Himmelblau [HB] testproblem.

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

eval(*x*)

class panobbgo_lib.classic.**Rastrigin**(*dims*, *par1=10*, *offset=0*)
Bases: panobbgo_lib.lib.Problem

Rastrigin

$$f(x) = par_1 \cdot n + \sum_i (x_i^2 - 10\cos(2\pi x_i))$$

eval(*x*)

class panobbgo_lib.classic.**Rosenbrock**(*dims*, *par1=100*)
Bases: panobbgo_lib.lib.Problem

Rosenbrock function with parameter `par1`.

$$f(x) = \sum_i (par_1(x_{i+1} - x_i^2)^2 + (1 - x_i)^2)$$

eval(*x*)

class panobbgo_lib.classic.**RosenbrockAbs**(*dims*, *par1=100*)
Bases: panobbgo_lib.lib.Problem

Absolute Rosenbrock function.

$$f(x) = \sum_i par_1 \left\| x_{i+1} - \|x_i\| \right\| + \|1 - x_i\|$$

eval(*x*)

class panobbgo_lib.classic.**RosenbrockAbsConstraint**(*dims*, *par1=100*, *par2=0.1*)
Bases: panobbgo_lib.lib.Problem

Absolute Rosenbrock function.

$$\begin{aligned} \min f(x) &= \sum_i par_1 \left\| x_{i+1} - \|x_i\| \right\| + \|1 - x_i\| \\ s.t.\ \ & \|x_{i+1} - x_i\| \geq par_2 \ \ \forall i \in \{0, \ldots, dim - 1\} \\ & x_i \geq 0 \ \ \forall i \end{aligned}$$

eval(*x*)

eval_constraints(*x*)

class panobbgo_lib.classic.**RosenbrockConstraint**(*dims*, *par1=100*, *par2=0.25*)
Bases: panobbgo_lib.lib.Problem

Constraint Rosenbrock function with parameter `par1` and `par2`.

$$\begin{aligned} \min f(x) &= \sum_i par_1 (x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \\ s.t.\ \ & (x_{i+1} - x_i)^2 \geq par_2 \ \ \forall i \in \{0, \ldots, dim - 1\} \\ & x_i \geq 0 \ \ \forall i \end{aligned}$$

eval(*x*)

eval_constraints(*x*)

class panobbgo_lib.classic.**RosenbrockStochastic**(*dims*, *par1=100*, *jitter=0.1*)
Bases: panobbgo_lib.lib.Problem

Stochastic variant of Rosenbrock function.

$$f(x) = \sum_i (par_1 eps_i (x_{i+1} - x_i^2)^2 + (1 - x_i)^2)$$

where eps_i is a uniformly random (n-1)-dimensional vector in $[0, 1)^{n-1}$.

eval(*x*)

class panobbgo_lib.classic.**Shekel**(*dims*, *m=10*, *a=None*, *c=None*)
Bases: panobbgo_lib.lib.Problem

Shekel Function [SH].

For m minima in n dimensions:

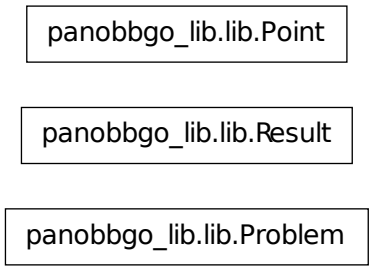
$$f($$

ec{x}) = sum_{i = 1}^{m} frac{1}{c_{i} + sumlimits_{j = 1}^{n} (x_{j} - a_{ji})^2 }

eval(*x*)

3.2 Library Classes

This file contains the basic objects to build a problem and to do a single evaluation.



Note: This is used by panobbgo and panobbgo_lib.

class panobbgo_lib.lib.**Point**(*x*, *who*)
Bases: object

This contains the x vector for a new point and a reference to who has generated it.

who
A string, which is the name of a heuristic.

To get the actual heuristic, use the strategie's heuristic method.

x
The vector x, a numpy.ndarray

class panobbgo_lib.lib.**Problem**(*box*)

Bases: object

this is used to store the objective function, information about the problem, etc.

box must be a list of tuples, which specify the range of each variable.

example: $[(-1, 1), (-100, 0), (0, 0.01)]$.

box

The bounding box for this problem, a $(dim, 2)$-array.

Note: This might change to a more sophisticated Box object.

---

dim

The number of dimensions.

eval(*x*)

This is called to evaluate the given black-box function. The problem should be called directly (__call__ special function wraps this) and the given problem should subclass this eval method.

eval_constraints(*x*)

This method is optionally overwritten by the problem to calculate the constraint violations. It has to return a numpy.ndarray of floats.

project(*point*)

projects given point into the search box. e.g. $[-1.1, 1]$ with box $[(-1, 1), (-1, 1)]$ gives $[-1, 1]$

random_point()

generates a random point inside the given search box (ranges).

ranges

The ranges along each dimension, a numpy.ndarray.

class panobbgo_lib.lib.**Result**(*point*, *fx*, *cv_vec=None*, *cv_norm=None*, *error=0.0*)

Bases: object

This represents one result, wich is a mapping of a Point $x \to f(x)$.

Additionally, there is also

- error: estimated or calculated $\Delta f(x)$.
- cv_vec: a possibly empty vector listing the constraint violation for each constraint.
- cnt: An integer counter, starting at 0.

Args:

- cv: the constraint violation vector
- cv_norm: the norm used to calculate cv. (see numpy.linalg.norm(), default None means 2-norm)

cnt

Integer ID for this result.

cv

The chosen norm of cv_vec; see cv_norm in constructor.

Note: Only the positive entries are used to calculate the norm!

---

cv_vec

Vector of constraint violations for each constraint, or None.

Note: Be aware, that entries could be negative. This is useful if you want to know how well a point is satisfied. The *.cv* property just looks at the positive entries, though.

---

error
Error margin of function evaluation, usually 0.0.

fx
The function value $f(x)$ after `evaluating` it.

point
Returns the actual `Point` object.

pp
pareto point, i.e. array([cv, fx])

who
The `name` of the heuristic, who did generate this point (String).

x
Point x where this result has been evaluated.

CHAPTER

FOUR

EXAMPLES

- ex1
- ex2

CHAPTER

FIVE

LINKS

- source repository
- short introduction talk
- Indices and Tables
  - *genindex*
  - *modindex*
  - *search*

BIBLIOGRAPHY

[HB] http://en.wikipedia.org/wiki/Himmelblau%27s_function

[SH] http://en.wikipedia.org/wiki/Shekel_function

[IP] http://www.ipython.org

PYTHON MODULE INDEX

p

INDEX

P

R

S

T

U

W

X