Python/C++ SOLNP

Krister S. Jakobsson

Mar 24, 2022

TABLE OF CONTENTS

1 Introduction **3**
- 1.1 SOLNP . . . 3
- 1.2 GOSOLNP . . . 4

2 Python SOLNP (pysolnp) **5**
- 2.1 Installation . . . 5
- 2.2 Method . . . 5
- 2.3 Inputs . . . 6
- 2.4 Outputs . . . 6
- 2.5 Example 1: Box Problem . . . 7
- 2.6 Use-cases and Applications . . . 7

3 Python GOSOLNP (pygosolnp) **8**
- 3.1 Installation . . . 8
- 3.2 Method . . . 8
- 3.3 Inputs . . . 9
- 3.4 Outputs . . . 10
- 3.5 Example 1: Electron Optimization Problem . . . 10
- 3.6 Example 2: Multi-process Solving . . . 12
- 3.7 Example 3: Truncated Normal Distribution . . . 13
- 3.8 Example 4: Grid Sampling . . . 14

4 C++ SOLNP (cppsolnp) **16**
- 4.1 Installation . . . 16
- 4.2 Methods . . . 16
- 4.3 Results . . . 17
- 4.4 Example 1: Box Problem . . . 18

5 Changelog **20**
- 5.1 pysolnp 2022.3.13 . . . 20
- 5.2 pysolnp 2021.4.30 . . . 20
- 5.3 Older Releases . . . 20

6 About **22**
- 6.1 Authors . . . 22
- 6.2 Acknowledgments . . . 22
- 6.3 Licensing . . . 22

[1]

[1] https://solnp.readthedocs.io/en/latest/?badge=latest

CHAPTER

ONE

INTRODUCTION

1.1 SOLNP

The SOLNP algorithm by Yinyu Ye (1989) solves the general nonlinear optimization problem below. In other words, it will find an optimum (possibly local) to it. The algorithm was originally implemented in Matlab[2], and have gained some fame through it's R implementation (RSOLNP[3]). solnp is written in C++ and with the Python wrappers (pysolnp) you have seamless integration with Python, providing high efficiency and ease of use.

$$\min_{\mathbf{x}} f(\mathbf{x})$$

$$\text{s.t.}$$

$$\mathbf{g}(\mathbf{x}) = \mathbf{e_x}$$

$$\mathbf{l_h} < \mathbf{h}(\mathbf{x}) < \mathbf{u_h}$$

$$\mathbf{l_x} < \mathbf{x} < \mathbf{u_x}$$

Here:

- $\mathbf{x}$ is the optimization parameter.
- $f(\mathbf{x})$, $\mathbf{h}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are smooth constraint functions.
- The constant-valued vector $\mathbf{e_x}$ is the target value for the equality constraint(s).
- The constant-valued vectors $\mathbf{l_h}$ and $\mathbf{u_h}$ are the lower and upper limits resp. for the inequality constraint(s).
- The constant-valued vectors $\mathbf{l_x}$ and $\mathbf{u_x}$ are the lower and upper limits resp. for the parameter constraint(s).

Bold characters indicate the variable or function can be vector-valued. All constraints are optional and vectors can be of any dimension.

[2] https://web.stanford.edu/~yyye/matlab/

[3] https://cran.r-project.org/web/packages/Rsolnp/index.html

1.2 GOSOLNP

GOSOLNP tries to find the global optimum for the given problem above. This algorithm was originally implemented in R through RSOLNP[4], but has been made available in Python through the library pygosolnp.

This is done by:

1. Generate n random starting parameters based on some specified distribution.
2. Evaluate the starting parameters based on one of two evaluation functions (lower value is better):
   a. Objective function $f(\mathbf{x})$ for all $\mathbf{x}$ that satisfies the inequality constraints $\mathbf{l_x} < \mathbf{x} < \mathbf{u_x}$
   b. Penalty Barrier function: $f(\mathbf{x}) + 100 \sum_i (\max(0, 0.9 + l_{x_i} - [\mathbf{g}(\mathbf{x})]_i)^2 + \max(0, 0.9 + [\mathbf{g}(\mathbf{x})]_i - u_{x_i})^2) + \sum_j ([h(\mathbf{x})]_j - e_{x_j})^2 / 100$
3. For the m starting parameters with the lowest evaluation function value, run pysolnp to find nearest optimum.
4. Return the best converged solution among the ones found through the various starting parameters (lowest solution value.)

[4] https://cran.r-project.org/web/packages/Rsolnp/index.html

CHAPTER

TWO

PYTHON SOLNP (PYSOLNP)

[5] 

[6] pysolnp provides Python with the power of the SOLNP algorithm explained in *Introduction* section. It is simply a Python wrapper for *C++ solnp*.

2.1 Installation

In most situations, installing with the package installer for Python, pip, will work:

```
$ pip install pysolnp
```

Precompiled Wheels are available for CPython:

- Windows: Python 3.6+
- Linux: Python 3.6+
- Mac OS: Python 3.6+

For other systems, or to have BLAS and LAPACK support, please build the wheels manually.

```
$ pip install --no-binary :all: pysolnp
```

Note that this requires CMake.

2.2 Method

```
solve(obj_func: function,
      par_start_value: List,
      par_lower_limit: object = None,
      par_upper_limit: object = None,
      eq_func: object = None,
      eq_values: object = None,
      ineq_func: object = None,
      ineq_lower_bounds: object = None,
      ineq_upper_bounds: object = None,
      rho: float = 1.0,
```

(continues on next page)

[5] https://codecov.io/gh/KristerSJakobsson/solnp
[6] https://pypi.org/project/pysolnp/

(continued from previous page)

```
    max_major_iter: int = 10,
    max_minor_iter: int = 10,
    delta: float = 1e-05,
    tolerance: float = 0.0001,
    debug: bool = False) -> pysolnp.Result
```

2.3 Inputs

Parameter	Type	Default value[7]	Description
obj_func	Callable[List, float]		The objective function $f(x)$ to minimize.
par_start_value	List		The starting parameter x_0.
par_lower_limit	List	None	The parameter lower limit x_l.
par_upper_limit	List	None	The parameter upper limit x_u.
eq_func	Callable[List, float]	None	The equality constraint function $h(x)$.
eq_values	List	None	The equality constraint values e_x.
ineq_func	Callable[List, float]	None	The inequality constraint function $g(x)$.
ineq_lower_bounds	List	None	The inequality constraint lower limit g_l.
ineq_upper_bounds	List	None	The inequality constraint upper limit g_l.
rho	float	1.0	Penalty weighting scalar for infeasability in the augmented objective function.[8]
max_major_iter	int	400	Maximum number of outer iterations.
max_minor_iter	int	800	Maximum number of inner iterations.
delta	float	1e-07	Step-size for forward differentiation.
tolerance	float	1e-08	Relative tolerance on optimality.
debug	bool	False	If set to true some debug output will be printed.

2.4 Outputs

The function returns the `pysolnp.Result` object with the below properties.

Property	Type	Description
solve_value	float	The value of the objective function at optimum $f(x*)$.
optimum	List[float]	A list of parameters for the optimum $x*$.
callbacks	int	Number of callbacks done to find this optimum.
converged	boolean	Indicates if the algorithm converged or not.
hessian_matrix	List[List[float]]	The final Hessian Matrix used by pysolnp.

[7] Defaults for configuration parameters are based on the defaults for Rsolnp.

[8] Higher values means the solution will bring the solution into the feasible region with higher weight. Very high values might lead to numerical ill conditioning or slow down convergence.

2.5 Example 1: Box Problem

The Box Problem is a common example function used for testing optimization algorithms. It has one equality constraint and variable bounds.

```
import pysolnp

def f_objective_function(x):
    return -1 * x[0] * x[1] * x[2]

def g_equality_constraint_function(x):
    return [4 * x[0] * x[1] + 2 * x[1] * x[2] + 2 * x[2] * x[0]]

x_starting_point = [1.1, 1.1, 9.0]
x_l = [1.0, 1.0, 1.0]
x_u = [10.0, 10.0, 10.0]
e_x = [100]

result = pysolnp.solve(
    obj_func=f_objective_function,
    par_start_value=x_starting_point,
    par_lower_limit=x_l,
    par_upper_limit=x_u,
    eq_func=g_equality_constraint_function,
    eq_values=e_x)

result.solve_value
result.optimum
result.callbacks
result.converged
```

Running this will yield the output:

```
>>> result.solve_value
-48.11252206814995
>>> result.optimum
[2.8867750707815447, 2.8867750713194273, 5.773407748939196]
>>> result.callbacks
118
>>> result.converged
True
```

2.6 Use-cases and Applications

- NMPC - Nonlinear model predictive controls-case studies using Matlab, REXYGEN and pysolnp NLP solver under Python environment by Štěpán Ožana. [NMPC Overhead Crane (PDF)[9]] [GitHub Source Code[10]] [Štěpán's Homepage[11]]

[9] https://github.com/StepanOzana/NMPC/raw/main/NMPC_Overhead_Crane/NMPC_overhead_crane_description.pdf
[10] https://github.com/StepanOzana/NMPC
[11] http://stepan-ozana.com/index.php?lang=EN

CHAPTER

THREE

PYTHON GOSOLNP (PYGOSOLNP)

[12] [13] pygosolnp provides Python with the power of the GOSOLNP algorithm explained in the *Introduction* section. It works as an extension on top of pysolnp by solving the problem multiple times from a randomized set of starting points. This library is implemented purely in Python.

3.1 Installation

Works on any environment that supports pysolnp and has Python 3.6+ installed.

3.2 Method

```
pygosolnp.solve(
      obj_func: Callable,
      par_lower_limit: List[float],
      par_upper_limit: List[float],
      eq_func: Optional[Callable] = None,
      eq_values: Optional[List[float]] = None,
      ineq_func: Optional[Callable] = None,
      ineq_lower_bounds: Optional[List[float]] = None,
      ineq_upper_bounds: Optional[List[float]] = None,
      number_of_restarts: int = 1,
      number_of_simulations: int = 20000,
      number_of_processes: Optional[int] = None,
      start_guess_sampling: Union[None, List[Distribution], Sampling] =␣
↪None,
      seed: Union[None, int] = None,
      evaluation_type: Union[EvaluationType, int] = EvaluationType.
↪OBJECTIVE_FUNC_EXCLUDE_INEQ,
      pysolnp_rho: float = 1.0,
      pysolnp_max_major_iter: int = 10,
      pysolnp_max_minor_iter: int = 10,
      pysolnp_delta: float = 1e-05,
      pysolnp_tolerance: float = 0.0001,
      debug: bool = False) -> Results
```

[12] https://codecov.io/gh/KristerSJakobsson/pygosolnp
[13] https://pypi.org/project/pytest/

3.3 Inputs

Parameter	Type	Default value[14]	Description
obj_func	Callable[List, float]		The objective function $f(x)$ to minimize.
par_lower_limit	List	None	The parameter lower limit x_l.
par_upper_limit	List	None	The parameter upper limit x_u.
eq_func	Callable[List, float]	None	The equality constraint function $h(x)$.
eq_values	List	None	The equality constraint values e_x.
ineq_func	Callable[List, float]	None	The inequality constraint function $g(x)$.
ineq_lower_bounds	List	None	The inequality constraint lower limit g_l.
ineq_upper_bounds	List	None	The inequality constraint upper limit g_l.
number_of_restarts	int	1	The $number_of_restarts$ best evaluation results are used to run pysolnp $number_of_restarts$ times.
number_of_simulations	int	20000	Sets how many randomly generated starting guesses we generate and evaluate with the evaluation function.
number_of_processes	int	None	Sets how many parallel processes to run when solving the problem. If None the problem is solved in the main processes.
start_guess_sampling	List[Distribution] or Sampling	None	A list of distributions for generating starting values, one distribution for each parameter. If None, the Uniform distribution is used.[16]
seed	int	None	By default the MT19937 Generator is used with timestamp-seed. Optionally an integer seed can be supplied.
evaluation_type	EvaluationType or int	EvaluationType.OBJECTIVE_FUNC_EXCLUDE_INEQ	Selects the evaluation type from the pygosolnp.EvaluationType enum.
pysolnp_rho	float	1.0	pysolnp parameter:Penalty weighting scalar for infeasability in the augmented objective function.[15]
pysolnp_max_major_iter	int	400	pysolnp parameter:Maximum number of outer iterations.
pysolnp_max_minor_iter	int	800	pysolnp parameter:Maximum number of inner iterations.
pysolnp_delta	float	1e-07	pysolnp parameter:Step-size for forward differentiation.
pysolnp_tolerance	float	1e-08	pysolnp parameter:Relative tolerance on optimality.
debug	bool	False	If set to true some debug output will be printed.

[14] Defaults for configuration parameters are based on the defaults for Rsolnp.

[16] Supply an instance of a class that inherits the abstract class $pygosolnp.sampling.Sampling$ to provide starting guesses, see *Example 3: Truncated Normal Distribution* and *Example 4: Grid Sampling*.

[15] Higher values means the solution will bring the solution into the feasible region with higher weight. Very high values might lead to numerical ill conditioning or slow down convergence.

3.4 Outputs

The function returns the `pygosolnp.Results` object with the below properties.

Property	Type	Description
best_solution	Optional[Result]	The best local optimum found for the problem.
all_results	List[Result]	All restarts and their corresponding local optimum.
starting_guesses	List[float]	All the randomized starting parameters.

Each named tuple `pygosolnp.Result` has the below properties.

Property	Type	Description
obj_value	float	The value of the objective function at optimum $f(x*)$.
parameters	List[float]	A list of parameters for the local optimum $x*$.
converged	bool	Boolean which indicates if the solution is within bounds.

3.5 Example 1: Electron Optimization Problem

This is a common benchmark problem for Global Optimization that finds the equilibrium state distribution for electrons positioned on a conducting sphere. See the COPS benchmarking suite[17] for details.

See full source code on GitHub /python_examples/example_electron.py[18]

```
import pygosolnp
from math import sqrt
import time

number_of_charges = 25


def obj_func(data):
    x = data[0:number_of_charges]
    y = data[number_of_charges:2 * number_of_charges]
    z = data[2 * number_of_charges:3 * number_of_charges]

    result = 0.0
    for i in range(0, number_of_charges - 1):
        for j in range(i + 1, number_of_charges):
            result += 1.0 / sqrt((x[i] - x[j]) ** 2 + (y[i] - y[j]) ** 2 +␣
↪(z[i] - z[j]) ** 2)

    return result


def eq_func(data):
    x = data[0:number_of_charges]
    y = data[number_of_charges:2 * number_of_charges]
    z = data[2 * number_of_charges:3 * number_of_charges]
    result = [None] * number_of_charges
```

(continues on next page)

[17] https://www.mcs.anl.gov/~more/cops/
[18] https://github.com/KristerSJakobsson/pygosolnp/blob/main/python_examples/example_electron.py

(continued from previous page)

```
    for i in range(0, number_of_charges):
        result[i] = x[i] ** 2 + y[i] ** 2 + z[i] ** 2

    return result


parameter_lower_bounds = [-1] * number_of_charges * 3
parameter_upper_bounds = [1] * number_of_charges * 3

equality_constraints = [1] * number_of_charges

if __name__ == '__main__':
    start = time.time()

    results = pygosolnp.solve(
        obj_func=obj_func,
        eq_func=eq_func,
        eq_values=equality_constraints,
        par_lower_limit=parameter_lower_bounds,
        par_upper_limit=parameter_upper_bounds,
        number_of_simulations=20000,  # This represents the number of␣
↪starting guesses to use
        number_of_restarts=20,  # This specifies how many restarts to run␣
↪from the best starting guesses
        number_of_processes=None,  # None here means to run everything␣
↪single-processed
        seed=443,  # Seed for reproducibility, if omitted the default␣
↪random seed is used (typically cpu clock based)
        pysolnp_max_major_iter=100,  # Pysolnp property
        debug=False)

    end = time.time()

    all_results = results.all_results
    print("; ".join([f"Solution {index + 1}: {solution.obj_value}" for␣
↪index, solution in enumerate(all_results)]))
    best_solution = results.best_solution
    print(f"Best solution {best_solution.obj_value} for parameters {best_
↪solution.parameters}.")
    print(f"Elapsed time: {end - start} s")
```

```
Solution 1: 244.1550118432253; Solution 2: 243.9490050190484; Solution 3:␣
↪185.78533081425041; Solution 4: 244.07921194485854; Solution 5: 216.
↪19236253370485; Solution 6: 194.1742137471891; Solution 7: 258.
↪6157748268509; Solution 8: 205.72538678938517; Solution 9: 244.
↪0944480181356; Solution 10: 217.4090464122706; Solution 11: 201.
↪58045387715478; Solution 12: 247.70691375326325; Solution 13: 243.
↪92615570955812; Solution 14: 192.3944392661305; Solution 15: 243.
↪93657263760585; Solution 16: 247.17924771908508; Solution 17: 244.
↪06529702108125; Solution 18: 244.29427536763717; Solution 19: 199.
↪69130383979302; Solution 20: 243.99315264179037
Best solution 243.92615570955812 for parameters [0.8726149386907173, 0.
↪1488320711741995, -0.8215181712229778, 0.8597822831494584, -0.
↪265961670940264, -0.6664127144955102, -0.6029702658967409, 0.
↪2867960203292267, -0.04380531711098636, 0.9519854892760677, -0.
↪39592769694574026, -0.2160514547351913, -0.21416235954836016, 0.
↪4338472533837847, -0.9411378567701716, 0.6418976636970082, 0.
↪014864034847848012, 0.6981416769347426, 0.4413252856284809, -0.
↪5267725521555919, -0.9148568048943023, -0.5831731928212042, 0.
↪47570915153781534, 0.4089885176760918, 0.008471540399374077, -0.
↪36287443863890595, 0.8618964461129363, 0.5476494687199884, -0.
↪3309316231117961, 0.9582851670742292, -0.6505818085537286, 0.
↪2793946112676732, -0.7596998666078645, 0.65142774983249, 0.
```

(continues on next page)

(continued from previous page)

```
Elapsed time: 1595.9994523525238 s
```

3.6 Example 2: Multi-process Solving

This example expands on Example 1 and solves the Electron optimization problem with 4 processes. Note that Python as a language is not great at parallell execution in general, and that spawning processes is quite expensive. *pygosolnp* spawns processes using a Multiprocessing[19] *multiprocessing.pool* and shares memory between the processes. As such, it is easy to cause bugs if you are not familiar with how this works, so please read up on it before using it! For example, you can not pass lambda functions or class functions to process pools, but global functions should work fine. If you get different results between single-processing and multi-processing execution your python functions are liekly badly defined for multi-processing.

See full source code on GitHub /python_examples/example_electron.py[20]

Below example only changes the previous one in the function call, using 4 processes to run.

```
results = pygosolnp.solve(obj_func=obj_func,
                          eq_func=eq_func,
                          eq_values=equality_constraints,
                          par_lower_limit=parameter_lower_bounds,
                          par_upper_limit=parameter_upper_bounds,
                          number_of_restarts=20,
                          number_of_simulations=20000,
                          number_of_processes=4,  # Simulations and processes␣
↪will be executed in 4 processes
                          seed=443,
                          pysolnp_max_major_iter=100,
                          debug=False)
```

Running this will yield the output:

```
Solution 1: 244.1550118432253; Solution 2: 243.9490050190484; Solution 3:␣
↪185.78533081425041; Solution 4: 244.07921194485854; Solution 5: 216.
↪19236253370485; Solution 6: 194.1742137471891; Solution 7: 258.
↪6157748268509; Solution 8: 205.72538678938517; Solution 9: 244.
↪0944480181356; Solution 10: 217.4090464122706; Solution 11: 201.
↪58045387715478; Solution 12: 247.70691375326325; Solution 13: 243.
↪92615570955812; Solution 14: 192.3944392661305; Solution 15: 243.
↪93657263760585; Solution 16: 247.17924771908508; Solution 17: 244.
↪06529702108125; Solution 18: 244.29427536763717; Solution 19: 199.
↪69130383979302; Solution 20: 243.99315264179037
Best solution 243.92615570955812 for parameters [0.8726149386907173, 0.
↪1488320711741995, -0.8215181712229778, 0.8597822831494584, -0.
↪265961670940264, -0.6664127144955102, -0.6029702658967409, 0.
↪2867960203292267, -0.04380531711098636, 0.9519854892760677, -0.
↪39592769694574026, -0.2160514547351913, -0.21416235954836016, 0.
↪4338472533837847, -0.9411378567701716, 0.6418976636970082, 0.
↪014864034847848012, 0.6981416769347426, 0.4413252856284809, -0.
↪5267725521555819, -0.9148568048943023, -0.5831731928212042, 0.
↪47570915153781534, 0.4089885176760918, 0.008471540399374077, -0.
↪3628744386389059 5, 0.8618964461129363, 0.5476494687199884, -0.
↪3309316231117961, 0.9582851670742292, -0.6505818085537286, 0.
↪2793946112676732, -0.7596998666078645, 0.65142774983249, 0.
↪3057240684166494 5, -0.1736400992779951, -0.2357569641249718, -0.
↪9762296783338298, 0.8894784482368485, -0.21768032982807542, 0.
↪44966067028074935, 0.3598982107965 23, 0.3932146838134686, -0.
↪25429508229562935, -0.0021320897149067, 0.00025657298672405 61, 0.
↪6081775900274631, -0.8731755460834034, -0.07630776960802095, -0.
↪7462707639808169, 0.32690759610807246, 0.4847543563757037, -0.
↪15870866693945487, -0.38892531575475037, -0.10466177783304143, 0.
```

(continues on next page)

[19] https://docs.python.org/3/library/multiprocessing.html

[20] https://github.com/KristerSJakobsson/pygosolnp/blob/main/python_examples/example_electron.py

(continued from previous page)

```
Elapsed time: 596.5835165977478 ms
```

As expected, we get the same result as in the single-processed approach in example 1!

This is expected as the same random starting values are used as when running single-processed.

Furthermore, the execution time is around one third of the single-processed one. Note that if we reduce the number of restarts to 4 the single-processed execution would be quicker.

3.7 Example 3: Truncated Normal Distribution

pygosolnp does not depend on any large-scale library (pandas, numpy, scipy etc.) out of box. This example shows how to overrides the logic for generating starting points by using Scipy and the Truncated Normal distribution. It is fairly trivial to modify this example to use other Scipy Distributions[21], for example Beta Distribution sampling etc.

See full source code on GitHub /python_examples/example_truncated_normal.py[22]

```
# The Sampling class is an abstract class that can be inherited and␣
↪customized as you please
class TruncatedNormalSampling(pygosolnp.sampling.Sampling):

    def __init__(self,
                 parameter_lower_bounds: List[float],
                 parameter_upper_bounds: List[float],
                 seed: Optional[int]):
        self.__generator = Generator(PCG64(seed))
        self.__parameter_lower_bounds = parameter_lower_bounds
        self.__parameter_upper_bounds = parameter_upper_bounds

    def generate_sample(self, sample_size: int) -> List[float]:
        # This function is abstract, it returns random starting values for␣
↪one sample
        return truncnorm.rvs(a=self.__parameter_lower_bounds,
                             b=self.__parameter_upper_bounds,
                             size=sample_size,
                             random_state=self.__generator)


...
# See original file for full code
...

# Instantiate sampling object
sampling = TruncatedNormalSampling(
    parameter_lower_bounds=parameter_lower_bounds,
    parameter_upper_bounds=parameter_upper_bounds,
    seed=99)

results = pygosolnp.solve(
    obj_func=permutation_function,
    par_lower_limit=parameter_lower_bounds,
```

(continues on next page)

[21] https://docs.scipy.org/doc/scipy/reference/stats.html
[22] https://github.com/KristerSJakobsson/pygosolnp/blob/main/python_examples/example_truncated_normal.py

(continued from previous page)

```
    par_upper_limit=parameter_upper_bounds,
    number_of_restarts=6,
    number_of_simulations=2000,
    pysolnp_max_major_iter=25,
    pysolnp_tolerance=1E-9,
    start_guess_sampling=sampling)
```

Running this will yield:

```
# Solution 1: 0.0016119745327847497; Solution 2: 0.005968645850086645;␣
↪Solution 3: 0.006083292803668321; Solution 4: 0.006629107105976147;␣
↪Solution 5: 0.005305936314073526; Solution 6: 0.006049589559946693
# Best solution: [1.3008954298086124, 3.181786909056148, 1.
↪3814249752478918, 3.9436695447632877]
# Objective function value: 0.0016119745327847497
# Elapsed time: 8.562503099441528 s
```

With 2000 simulations we got a fairly accurate value in 8.5 seconds. Lets compare this with Grid Sampling below.

3.8 Example 4: Grid Sampling

pygosolnp does not depend on any large-scale library (pandas, numpy, scipy etc.) out of box. This example overrides the logic for generating starting points by using Scikit-optimize Grid Sampling. It is fairly trivial to modify this example to use other Scikit-optimize Sampling Methods[23], for example Sobol, Latin hypercube sampling etc.

See full source code on GitHub /python_examples/example_grid_sampling.py[24]

```
# The Sampling class is an abstract class that can be inherited and␣
↪customized as you please
class GridSampling(pygosolnp.sampling.Sampling):

    def __init__(self,
                 parameter_lower_bounds: List[float],
                 parameter_upper_bounds: List[float],
                 seed):
        self.__space = skopt.space.Space(dimensions=zip(parameter_lower_
↪bounds, parameter_upper_bounds))
        self.__seed = seed

    def generate_all_samples(self, number_of_samples: int, sample_size:␣
↪int) -> List[float]:
        # Overwrite this function to define the behavior when generating␣
↪starting guesses for all samples
        # By default it calls `generate_sample` number_of_samples time
        grid = skopt.sampler.Grid()
        grid_values = grid.generate(dimensions=self.__space.dimensions,
                                    n_samples=number_of_samples,
                                    random_state=self.__seed)
        return list(chain.from_iterable(grid_values))
```

(continues on next page)

[23] https://scikit-optimize.github.io/stable/auto_examples/sampler/initial-sampling-method.html
[24] https://github.com/KristerSJakobsson/pygosolnp/blob/main/python_examples/example_grid_sampling.py

(continued from previous page)

```
    def generate_sample(self, sample_size: int) -> List[float]:
        # This function is abstract
        # Not needed since we are generating a grid for all samples
        pass

...
# See original file for full code
...

# Instantiate sampling object
sampling = GridSampling(
    parameter_lower_bounds=parameter_lower_bounds,
    parameter_upper_bounds=parameter_upper_bounds,
    seed=92)

results = pygosolnp.solve(
    obj_func=permutation_function,
    par_lower_limit=parameter_lower_bounds,
    par_upper_limit=parameter_upper_bounds,
    number_of_restarts=6,
    number_of_simulations=2000,
    pysolnp_max_major_iter=25,
    pysolnp_tolerance=1E-9,
    start_guess_sampling=sampling)
```

Running this will yield the output:

:: # Solution 1: 0.0006360327708392506; Solution 2: 0.006239163594915304; Solution 3: 0.006140229082904356; Solution 4: 0.006218870214655177; Solution 5: 0.005963823643719209; Solution 6: 0.13065649880545976 # Best solution: [1.1622677695732497, 1.683172007310748, 3.9509962074974956, 3.159134907203731] # Objective function value: 0.0006360327708392506 # Elapsed time: 22.986207962036133 s

With 2000 simulations Grid Sampling gave a better result than Truncated Normal but it took longer.

CHAPTER

FOUR

C++ SOLNP (CPPSOLNP)

[25] C++ solnp implements the SOLNP algorithm explained in the *Introduction* section.

4.1 Installation

C++ solnp is a header-only library, so the only thing you need to do is:

- Add dlib to your project.
- Include the C++ solnp header `solnp.hpp` in your project.

Note that the CMake script in the root of the repository is mainly intended to compile wrappers and unit tests.

4.2 Methods

The solnp function can be called both with and without Hessian Matrix, as shown below.

```
template<
        typename functor_model,
        typename parameter_input,
        typename inequality_constraint_vectors>
double cppsolnp::solnp(
             functor_model functor,
             parameter_input &parameter_data,
             const inequality_constraint_vectors &inequality_constraint_
↪data,
             const std::shared_ptr<std::vector<std::string>> &event_log =␣
↪nullptr,
             double rho = 1.0,
             int maximum_major_iterations = 400,
             int maximum_minor_iterations = 800,
             const double &delta = 1e-7,
             const double &tolerance = 1e-8
    )
```

[25] https://codecov.io/gh/KristerSJakobsson/solnp

```
template<
        typename functor_model,
        typename parameter_input,
        typename inequality_constraint_vectors>
double cppsolnp::solnp(
             functor_model functor,
             parameter_input &parameter_data,
             const inequality_constraint_vectors &inequality_constraint_
↪data,
             dlib::matrix<double> &hessian_matrix,
             const std::shared_ptr<std::vector<std::string>> &event_log =␣
↪nullptr,
             double rho = 1.0,
             int maximum_major_iterations = 400,
             int maximum_minor_iterations = 800,
             const double &delta = 1e-7,
             const double &tolerance = 1e-8
    )
```

Templates:

- functor_model: Represents a callable functor or lambda.
- parameter_input: Represents the parameter input, this should be a dynamic-height or static-height dlib::matrix with:
  - 1 column representing the start point $\mathbf{x}*$ if the problem has no parameter bounds.
  - 2 columns representing the lower limit $\mathbf{x_l}$ and upper limit $\mathbf{x_u}$, in this case th inital guess will be the middle point of the bounds.
  - 3 columns representing the start point $\mathbf{x}*$, lower limit $\mathbf{x_l}$ and upper limit $\mathbf{x_u}$.
- inequality_constraint_vectors: Represents the inequality constraint, this should be a dynamic-height or static-height dlib::matrix with:
  - 2 columns representing the lower bounds $\mathbf{l_x}$ and upper bounds $\mathbf{u_x}$.

Note: In contrast to pysolnp, C++ solnp will assume that the equality constraints equal 0, so supply a function $\mathbf{g}_{new}(\mathbf{x}) = \mathbf{g}(\mathbf{x}) - \mathbf{e_x} = \mathbf{0}$
.

4.3 Results

The solnp function will return the solve result value as a double. The parameter_data matrix will be modified in-memory and contain the corresponding solution. If a logger is supplied as a smart-pointer to an std::vector of strings, various log messages will be stored in it.

4.4 Example 1: Box Problem

The Box Problem is a common example function used for testing optimization algorithms. It has one equality constraint and variable bounds.

The below code is taken from the C++ solnp unit tests.

```
#include "catch.hpp"
#include "solnp.hpp"

dlib::matrix<double, 2, 1> box(const dlib::matrix<double, 3, 1> &m)
{
    const double x1 = m(0);
    const double x2 = m(1);
    const double x3 = m(2);

    dlib::matrix<double, 2, 1> return_values(2);
    // Function value
    return_values(0) = -1 * x1 * x2 * x3;
    // Equality constraint
    return_values(1) = 4 * x1 * x2 + 2 * x2 * x3 + 2 * x3 * x1 - 100;
    return return_values;
}


struct box_functor {
public:
    box_functor() = default;;

    dlib::matrix<double, 2, 1> operator()(const dlib::matrix<double, 3, 1>␣
↪&x) {
        return box(x);
    }
};

TEST_CASE("Optimize the Box function", "[box]") {

    dlib::matrix<double, 3, 3> parameter_data;
    parameter_data =
            1.1, 1.0, 10.0,
            1.1, 1.0, 10.0,
            9.0, 1.0, 10.0;

    dlib::matrix<double, 0, 0> ib;

    std::shared_ptr<std::vector<std::string>> logger = std::make_shared
↪<std::vector<std::string>>();

    double calculate = cppsolnp::solnp(box_functor(), parameter_data, ib,␣
↪logger, 1.0, 10, 10, 1e-5, 1e-4);

    dlib::matrix<double, 0, 1> result = dlib::colm(parameter_data, 0);

    // Check the parameters
    CHECK(result(0) == Approx(2.886775069536727));
    CHECK(result(1) == Approx(2.886775072009683));
    CHECK(result(2) == Approx(5.773407750048355));
```

(continues on next page)

(continued from previous page)

```
    REQUIRE(calculate <= -48.112522068150462);

}
```

CHAPTER

FIVE

CHANGELOG

5.1 pysolnp 2022.3.13

- Fixed bug that would give incorrect results for completely unconstrained problems.
- Removed precompiled wheels (binaries) for Python 2.7 and 3.5, and added wheels for Python 3.10.
- Migrated CI to Github Actions from a mix of Appveyor and Travis
- Started using cibuildwheels package for building wheels

5.2 pysolnp 2021.4.30

Serious issue found in releases 2021.3.8, 2021.4.25 and 2021.4.26 that caused incorrect output. This has been fixed in this release and previous releases have been deprecated.

5.3 Older Releases

pysolnp 2021.4.26 [Deprecated due to bug in output]

Fixed bug where the converged flag would only be set correctly when the debug was set to true.

pysolnp 2021.4.25 [Deprecated due to bug in output]

No changes, re-release due to issue with source code build in previous version.

pysolnp 2021.3.8 [Deprecated due to bug in output]

- Fixed issues where build would fail on Windows with newer versions of pip
- Added outputs:
    1. converged : Boolean that indicates if the algorithm converged or not
    2. hessian_matrix : A nested list of the last Hessian Matrix used by pysolnp

pysolnp 2021.1.27 [Deprecated due to build issues]

Add wheel to Python 3.9 for all platforms and fix below issues:

- A change to pip means that meta-data version must match file-name version. This broke the source code build of pysolnp with recent versions of pip.

pygosolnp 2021.1.24

Initial release of the PYGOSOLNP library to PyPi. Pure Python 3.6+ library so no precompiled binaries released.

pysolnp 2020.4.11

Initial release of the PYSOLNP library to PyPi. Release includes precompiled wheels for Python 2.7, 3.5-3.8 (excluding Python 3.5 for Windows due to compilation issues).

cppsolnp 2020.4.11

Initial release of the C++ SOLNP library.

CHAPTER

SIX

ABOUT

6.1 Authors

C++ solnp and pysolnp are developed and maintained by:

- *Krister S Jakobsson - Implementation* - krister.s.jakobsson@gmail.com

6.2 Acknowledgments

- **Yinyu Ye** - Publisher and mastermind behind the original SOLNP algorithm, Original Sources[26]
- **Alexios Ghalanos and Stefan Theussl** - The people behind RSOLNP, Github Repository[27]
- **Davis King** - The mastermind behind Dlib, check out his blog! Blog[28]

6.3 Licensing

Both pysolnp and cppsolnp makes use of software components under the Boost license below. Note that the package depend on other libraries that may not have the same generous license. If important, please investigate the dependencies and their respective licensing.

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.
```

(continues on next page)

[26] https://web.stanford.edu/~yyye/matlab/
[27] https://github.com/cran/Rsolnp
[28] http://blog.dlib.net/

(continued from previous page)

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```