optipoly: A Python package for boxed-constrained multi-variable polynomial cost functions optimization

Mazen Alamir

`optipoly`: A `Python` package for boxed-constrained multi-variable polynomial cost functions optimization

Mazen Alamir*

Abstract. In this paper, a new `python` package (`optipoly`) is described that solves box-constrained optimization problem over multivariate polynomial cost functions. The principle of the algorithm is described before its performance is compared to three general purpose NLP solvers implemented in the state-of-the-art `Gekko` and `scipy` packages. The comparison show statistically better best solution provided by the algorithm with significantly less computation times. The package will be shortly made freely and easily available through the simple `pip install` process.

Key words. Non Linear Programming (NLP); Multi-variate polynomials; box-constraints; local minima; computation time.

AMS subject classifications. 26A24

1. Introduction.

Optimizing over polynomial cost functions is an important topics in computational engineering. This is because mutli-variate polynomial function basis provide a universal representation of *smooth* relationships. Moreover, optimization over multi-variate polynomial might be used as an intermediate step in a optimization process as it is the case for quadratic optimization that is commonly used in the Sequential Quadratic Programming (SQP)-based algorithm. In that sense, SQP can be viewed as a particular instance of a more general Sequential Polynomial Programming (SPP) which might be less prone to local minima as it is suggested by the comparison results shown at the end of this study.

The starting point of the investigation that led to the algorithm involved in the `optipoly` package is a newly developed sparse identification method, similar to the ones proposed in the `scikit-learn` library [3], namely `larsCV`, `lassoCV` and `lassolarsCV` but which is **scalable**[1] in the number of candidate features. This paves the way for the identification of multivariate-polynomials in a relatively high number of raw features that can represent cost functions depending on a decent number of decision variables.

Now one might ask: *What is specific about polynomial functions that makes them eligible for a special treatment that general purpose optimizers do not apply?*

The answer to that question lies in the fact that polynomials can benefit from very efficient vectorized computation algorithms that are implemented through the `scipy` module and its `interp1d` method. Indeed, this method enables, for instance, to evaluate a *scalar* polynomial

*Univ. Grenoble Alpes, CNRS, Grenoble INP, GIPSA-lab, 38000 Grenoble, France (mazen.alamir@cnrs.fr, https://www.mazenalamir.fr).

[1] Problems with 400,000 features can be solved in less than one minutes.

over 2 millions of values in less that 10 msec. This lies in the heart of the newly proposed algorithm that is presented in the current note.

This paper is organized as follows: First of all, the principle of the algorithm is presented in Section 2. Then section 3 presents the `optipoly` module and its main classes and methods. Finally, Section 4 show the comparison with some state-of-the-art solvers to underline the advantages of using `optipoly`. The paper ends with Section 5 that summarizes the message of the paper.

2. The principle of the algorithm.

The `optipoly` module is dedicated to solving **box-constrained polynomial** optimization problems of the following form:

$$\min_x P(x) \quad | \quad x \in [x_{\min}, x_{\max}] \subset \mathbb{R}^n \tag{2.1}$$

where

- $x_{\min}, x_{\max} \in \mathbb{R}^n$ stand for the vector of minimum and maximum values of the components of the decision variables $x \in \mathbb{R}^n$ so that the inclusion constraint expressed by:

  $$x \in [x_{\min}, x_{\max}] \tag{2.2}$$

  is to be interpreted component-wise.

  The terminology *Box-constrained* refers to the fact that the constraints expressed by (2.2) are the only *hard* constraints that are admissible[2].

- $P(x)$ is some cost function that is a multivariate polynomial in the decision variable $x \in \mathbb{R}^n$. More precisely, $P(x)$ takes the following general form:

  $$P(x) = \sum_{i=1}^{n_m} c_i \left[\prod_{j=1}^{n} x_j^{p_{ij}} \right] = \sum_{i=1}^{n_m} c_i [\phi_i(x)] \tag{2.3}$$

  in which n_m is the number of monomials involved in the polynomial P while ϕ_i is a multi-variate monomial while c_i is its associated coefficient in the expression of $P(x)$.

As a matter of fact, the proposed algorithm is extremely simple in that it incorporates a very old and simple principle that can be summarized as follows:

[2]This implicitly means that the cost function $P(x)$ represents a polynomial approximation of some original cost to which an exact penalty on some constraints violation penalty would have been added. The approximation polynomial would incorporate all possible non box-constraints like limitations.

PRINCIPLE OF THE ALGORITHM

Perform *several*[a] successive rounds of scalar polynomial optimization-by-enumeration in each of which, one of the components of the current solution is updated. The best value of the components is updated and the whole resulting candidate vector is used to define the new scalar polynomial in the next components based on which, the next component is updated and so on.

[a] A maximum number of iterations is defined but the iterations can be stopped before when some stopping criterion is satisfied.

Let us say it again: *Nothing is totally new here*!

This principle is well known and has been *rightly* abandoned based on efficiency arguments since the so called *optimization-by-enumeration* is generally not efficient when standard *for-loop* exploration is used. Instead descent-based iterations have been preferred since they correspond to a much lower number of function's evaluations.

Notice that, if one puts aside the computation time, the enumeration process provides the a priori advantage of being less prone to local minima (as far as the scalar optimization problem is concerned) than any descent method.

The efficient implementation of polynomial's evaluation over a high number of arguments values (through the `scipy-interp1d` method mentioned in the introduction) comes in handy in leveraging the advantages of enumeration while strongly reducing the computational cost associated to this process. This statement, supported by the numerical investigation presented below, is the main message of this paper.

Obviously, the issue of the *fight-against-the-clock* battle between descent-like approaches and cyclic scalar enumeration approaches (such as the one proposed in `optipoly`) is a quantitative question and the winner's designation is not a theoretical question. Rather, it is a practical, technological and algorithmic temporary matter.

Given the current situation of the latter determinant elements, the results proposed in this paper suggests that the second class of method outperforms the general purpose methods for the specific problem of box-constrained optimization of polynomial cost functions.

3. Description of the `optipoly` module.

Notice first of all that the expression of the polynomial function given by (2.3) shows that a multivariate polynomial is totally defined by the matrix of powers and the associated cost,

namely:

$$\texttt{powers} \in \mathbb{R}^{n_m \times n} \quad ; \quad \texttt{coefs} \in \mathbb{R}^{n_m} \tag{3.1}$$

which play respectively the roles of p and c in (2.3).

As a matter of fact, these are the only two arguments that are used to create an *instance* of the `Pol` class which is the main class of the `optipoly` module. For instance, assuming that the matrix of `powers` and the vector of coefficients `coefs` are available, an instance of the `Pol` class is obtained by the following script:

```
from optipoly import Pol

powers = [[1, 0, 2], [0,3,0]]
coefs = [1.0, 2.0]

pol = Pol(powers, coefs)
```

The above script defines the following polynomial in the three-dimensional variable $x \in \mathbb{R}^3$:

$$P(x) = x_1 x_3^2 + 2x_2^3 \tag{3.2}$$

The the following script computes the values of the polynomial at the arguments defined by the lines of the following matrix X:

$$X := \begin{bmatrix} 1 & 1 & 1 \\ -1 & 2 & 3 \\ 0 & 1 & 0 \end{bmatrix}$$

which means that the polynomial is evaluated at the arguments:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} , \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} , \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

```
X = [[1,1,1], [-1,2,3], [0,1,0]]
pol.eval(X)

>> array([3., 7., 2.])
```

Before the attributes and the methods of the class are described, let us mention the methods that enable, at a given value of x to extract the scalar polynomial in a specific in terms of a specific component of the decision variable.

In the following script, we use the `extract_sc_pol` method of the `pol` instance to compute the `scipy` scalar polynomial in x_2 at the current argument $x_0 = (-1, 2, 3)$ which obviously leads to the polynomial given by

$$2x_2^3 - 9 \tag{3.3}$$

which is identical to the result of the script.

```
x0 = np.array([-1,2,3])
sc_pol = pol.extract_sc_pol(x=x0, ix=1)
sc_pol

>> poly1d([ 2.,  0.,  0., -9.])
```

The following sections list the attributes of an instance of the class `Pol`, its methods and some useful methods of the class `Pol` itself that might be conveniently used for different purposes.

3.1. The attributes of the instances of the `Pol` class.

Once the instance `pol` of the class `Pol` is created, it exports the attributes that are listed in Table 1. These are shortly given in the following list:

`powers, coefs ,nx, ncoefs deg`

which are rather self-explanatory.

Attribute	Definition
`powers`	The matrix of powers used in the instantiation
`coefs`	The vector of coefficients of monomials used in the instantiation.
`nx`	The number of variables (dimension of x)
`ncoefs`	The number of monomials (n_m in (2.3))
`deg`	The degree of the multi-variate polynomial (the maximum values of the sum of powers over the lines of `powers`.

Table 1: List of the attributes of an instance of the class `Pol` of the `optipoly` module.

3.2. The methods exposed by an instance of the class `Pol`.

These methods are listed in Table 2. As an example, the use of the `to_df` method on the previously presented sample instance of the class `Pol` leads to the following dataframe

```
df = pol.to_df()
df
```

	x1	x2	x3	coeff
0	1	0	2	1.0
1	0	3	0	2.0

As for the `to_dict` method, it results in the following dictionary:

```
dic = pol.to_dict()
dic
```

```
{'nx': 3,
 'ncoefs': 2,
 'powers': [[1, 0, 2], [0, 3, 0]],
 'coefs': [1.0, 2.0],
 'deg': 3}
```

Since the definition of the call for the main method `solve` is more involved, the list of arguments and their description is given in the next section.

Methods	input	returns	Description
extract_sc_pol	x0, ix	sc_pol	Generate a scipy scalar polynomial sc_pol in the component indexed by ix at the current value x0 of the decision variable.
to_df	None	df	Generate a pandas dataframe df representing the polynomial data.
to_dict	None	dic	Generate a python dictionary dic representing all the information regarding the polynomial.
solve	x0, xmin, xmax, Ntrials, ngrid, iter_max, eps, psi	solution, cpu	The main method that minimizes/maximizes or find roots for the instance polynomial. The mode depends on the choice of the lambda function psi. See the detailed description of the parameters in Section 3.3.

Table 2: List of methods exported by an instance of the class `Pol` of the `optipoly` module.

3.3. Description of the main `solve` method. The last method (`solve`) invoked in Table 2 which is the *raison d'être* of the `optiply` package is more extensively described in the present section. Let us start by the discussion of the inputs arguments:

3.3.1. The input arguments for the `solve` method. The input arguments of the `solve` method are described in Table 3.

Parameters	Description
x0	The initial guess for the solver. This is a vector of dimension nx. Notice that when several starting points are used (Ntrials> 1 as explained below), the next initial guesses are randomly sampled in the admissible domain defined by xmin and xmax.
xmin	The vector of lower bounds of the decision variable.
xmax	The vector of upper bounds of the decision variable.
Ntrials	The number of different starting points used in order to enhance the avoidance of local minima (default is set to 1).
ngrid	The number of grid points used in the scalar optimization-by-enumeration in the different direction of the components of the decision variable (default is set to 1000).
iter_max	Maximum number of rounds of scalar optimization (default is set to 100). This number is rarely used since the optimization is stopped before based on the eps-related termination condition. In almost all the 1000 problems used in the benchmark discussed in this paper the number of iterations never exceeded 3 or 4.
eps	The precision that is used to stop the iterations (default value set to 10^{-2}). More precisely, the iterations stop when the last round of scalar optimization does not improve the cost function by more than eps × |J_previous| where J_previous is the cost achieved at the previous round.
psi	The lambda function that applies to the cost function in order to define the modified quantity that is to be minimized. For instance the choice: psi= lambda v : -v leads to the solver method being oriented towards the maximization of the original cost. On the other hand, the choice: psi= lambda v : abs(v) leads to the solver method being oriented towards finding a root of the polynomial. The default setting is given by psi= lambda v : v leading to solve trying to minimize the cost function. Notice that the above are only possible choices but any expression of v might be used provided that it is compatible with vectorized evaluation.

Table 3: Description of the input parameters of the solve method of the Pol class.

3.3.2. The output arguments of the solve method. The call for the solve method of an instance pol of the class Pol takes the following syntax:

```
solution, cpu = pol.solve(...)
```

where `solution` is a dictionary with the fields `x` and `f` standing respectively for the best found solution and the associated best corresponding value. On the other hand, `cpu` is the computation time required to produce the results.

The following script gives an example of a call that asks for the maximization of the polynomial defined earlier in the paper then prints the results so obtained:

```
nx = 3
x0 = np.zeros(nx)
ntrials = 6
ngrid = 1000
xmin = -1*np.ones(nx)
xmax = 2*np.ones(nx)

solution, cpu = pol.solve(x0=x0,
                          xmin=xmin,
                          xmax=xmax,
                          ngrid=ngrid,
                          Ntrials=ntrials,
                          psi=lambda v:-v
                          )

print(f'xopt = {solution.x}')
print(f'fopt = {solution.f}')
print(f'computation time = {solution.cpu}')
```

This call yields the following results:

```
xopt = [-1.  2.  0.]
fopt = 16.0
computation time = 0.0046999454498291016
```

Changing the definition of `psi` to `psi=lambda v:abs(v)` leads to the following results:

```
xopt = [-0.996997     0.58858859   0.63963964]
fopt = -9.305087356087371e-05
computation time = 0.0030119419097900039
```

Finally using the default setting `psi=lambda v:v` makes the method focused on finding the minimum of the cost function which leads to the following results:

```
xopt = [-1. -1.  2.]
fopt = -6.0
computation time = 0.005150318145751953
```

3.4. The `Pol` class's method.

Beside the previously described attributes and methods that are exported by the instances of the `Pol` class. The following method is a *class methods* that does not depend on a specific instance of the `Pol` class:

`generate_random_pol`

This method accepts the following input arguments:

`nx`	The number of variables of the polynomial.
`deg_max`	The degree of the polynomial.
`card`	The number of monomials involved in the polynomial's expression.

Based on the above input arguments, the method returns an instance of the class `Pol` that is a polynomial in `nx` variables, of degree `deg_max` involving `card` monomials. The choice of the matrix of powers and the vector of coefficients needed to instantiate the class are randomly sampled.

4. Comparison with some state-of-the-art general purpose solvers.

In this section, the performance of the above discussed algorithm in terms of the quality of the result found on one hand and the required computation time on the other hand are examined.

First of all, the alternative solvers are listed and shortly described, then the definition of the benchmark before the results are given.

4.1. The alternative solvers.

The set of alternative algorithms considered in the following comparison includes two NLP solvers called via the `Gekko` modeling framework [1] and a global optimization oriented solver proposed by the famous `scipy` python module.

More precisely, the following solvers are considered:

- ✓ **IPOPT**: The famous interior point NLP solver [2].
- ✓ **BPOPT** Another NLP general-purpose solver that is made available through the `Gekko` framework[3].
- ✓ **Dual-Annealing**: A `scipy` general-purpose solver for *global optimization* [4, 5]. This class of algorithm is interesting to consider here since the issue of local minima is one of the key issues to investigate since high degree multivariate polynomial generally present multiple local minima as it is confirmed by the results shown later on in this paper.

4.2. The benchmark's definition.

The benchmark consists in 1000 experiments of box-constrained polynomial optimization problems. For each experiment, a randomly sampled polynomial is generated using values of `nx`, `deg` and `card` that span the sets of values produced by the following script[4]:

```
set_nx = [3, 5, 10, 20, 50]
set_deg = [int(i) for i in list(np.arange(2,10))]
set_card = [int(i) for i in list(np.arange(5,30))]
configurations = list(itertools.product(set_nx, set_deg, set_card))
list_of_pols = [generate_random_pol(*c) for c in configurations]
```

This results in a list, called `configurations` in which each element is a triplet of values of (`nx`, `deg`, `card`) that can be used as calling list of arguments to the `generate_random_pol` described above.

[3]Notice that there is a third NLP algorithm proposed via the `Gekko` framework which is called APOPT that seems to produce exactly the same results than the `IPOPT` solver over the 1000 problem's instances which seems weird. That is the reason whyy this third algorithm has not be included in the comparison.

[4]Notice that the product of the cardinalities of the sets is equal to 1000.

4.3. Comparison results.

Two aspects of the performance are investigated in this section, namely:

- **The best found values of the cost function**.

  More precisely, in order to compare the performance of the proposed algorithm to the three above mentioned ones, the following differences are examined:

$$\Delta_h := f_h - f_{\text{proposed}}|_{\text{Ntrials}=1} \quad h \in \{\text{IPOPT, BPOPT, Dual-Annealing}\} \tag{4.1}$$

  where f_h is the best value of the cost function that is achieved by the algorithm h while $f_{\text{proposed}}|_{\text{Ntrials}=1}$ is the value achieved by the proposed algrothm using only one trial (`Ntrials`=1).

  Obviously, positive value of Δ_h means that the proposed algorithm achieved better solution that the alternative algorithm $h \in \{\text{IPOPT, Dual-Annealing, BPOPT}\}$.

  The results are shown in Figure 1 where the histograms of Δ_h is shown suggesting that in the majority of the problems, the proposed solver outperforms the alternative `gekko` *non global optimization-oriented* solvers (this is strictly true all the time when `bpopt` is considered).

  In the case of the IPOPT solver, the proposed solver shows better results for a large majority of problems. Moreover, when this does not happens, the difference is less important that the ones showing the superiority of the proposed algorithm. This can be more clearly seen on the cumulative histogram shown in Figure 2.

  As for the global-optimization oriented solver, it provides better results in on a slightly more frequent set of problems but the proposed algorithm still outperforms this global algorithm on a non negligible number of instances.

  Notice however that the global-optimizer needs computation times that are largely greater than the ones needed by the proposed algorithm as it is shown below (see Figure 5. Moreover, by using `Ntrials`=3 or 5, the proposed solver systematically outperforms the global `scipy` optimizer as it is shown in Figure 3 while keeping lower computation times as shown in Figure 5.

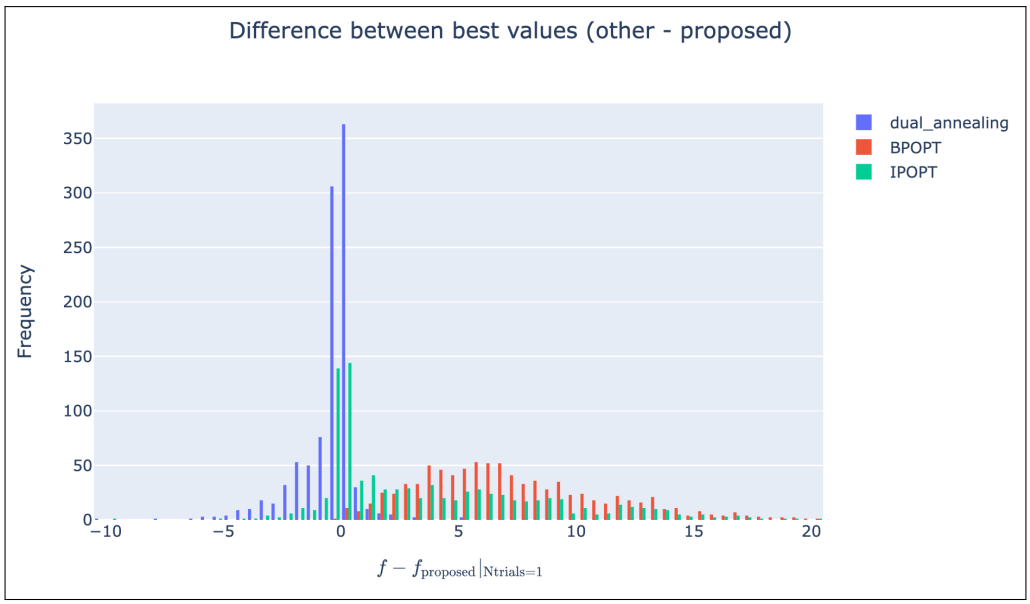


Figure 1: Histogram of the difference $f - f_{\text{proposed}}|_{\text{Ntrials}=1}$. A positive difference means that the best value obtained by the alternative solution is higher than the one delivered by the proposed reference solver using a single trial `Ntrials`=1. Although the global solver shows statistically better results, its computation times is largely higher as shown in Figure 4.

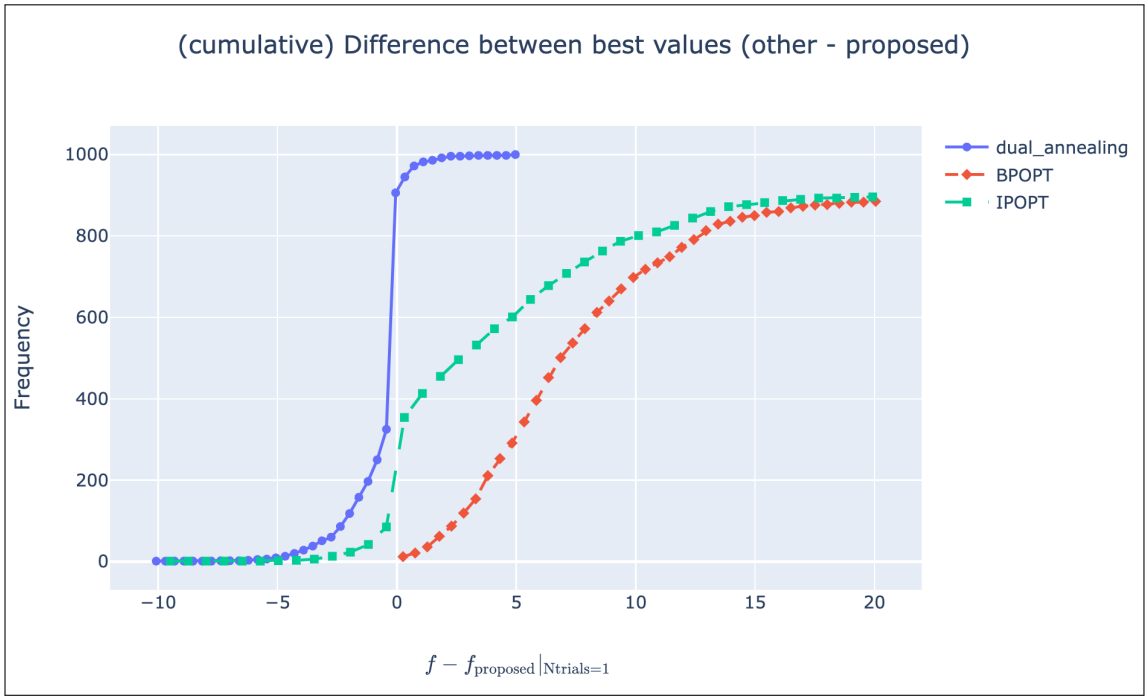
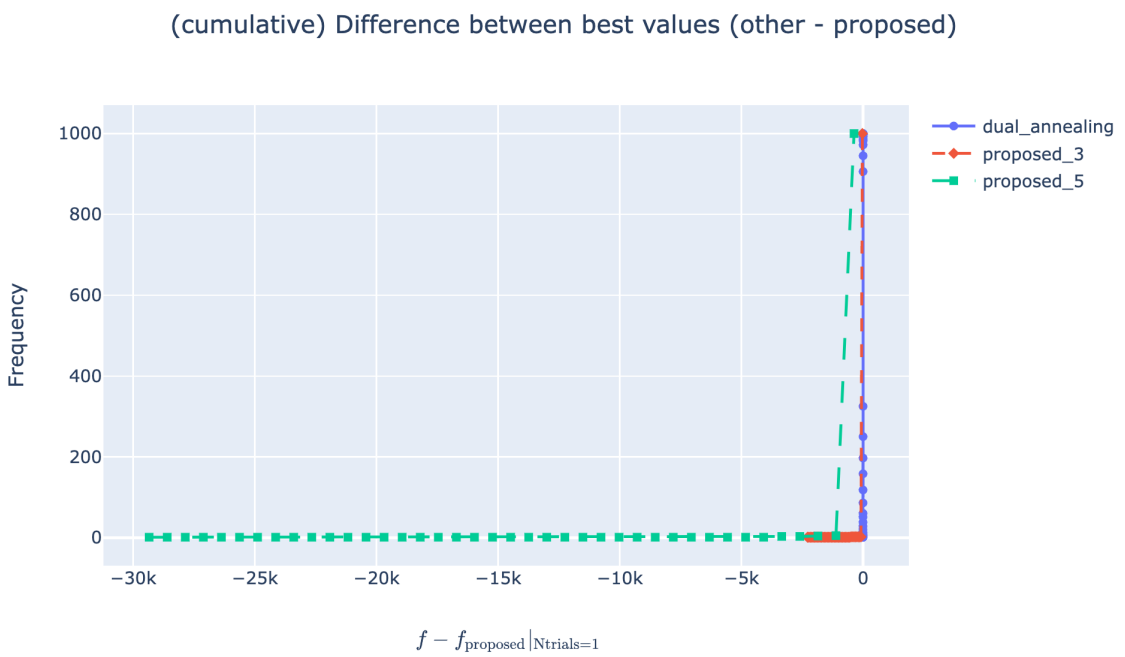


Figure 2: Cumulative version of the histogram of Figure 1. This figure shows also that the `gekko` solvers were not able to solve some of the problems considered (symbolic relationships were too long!)

(a)



(b)

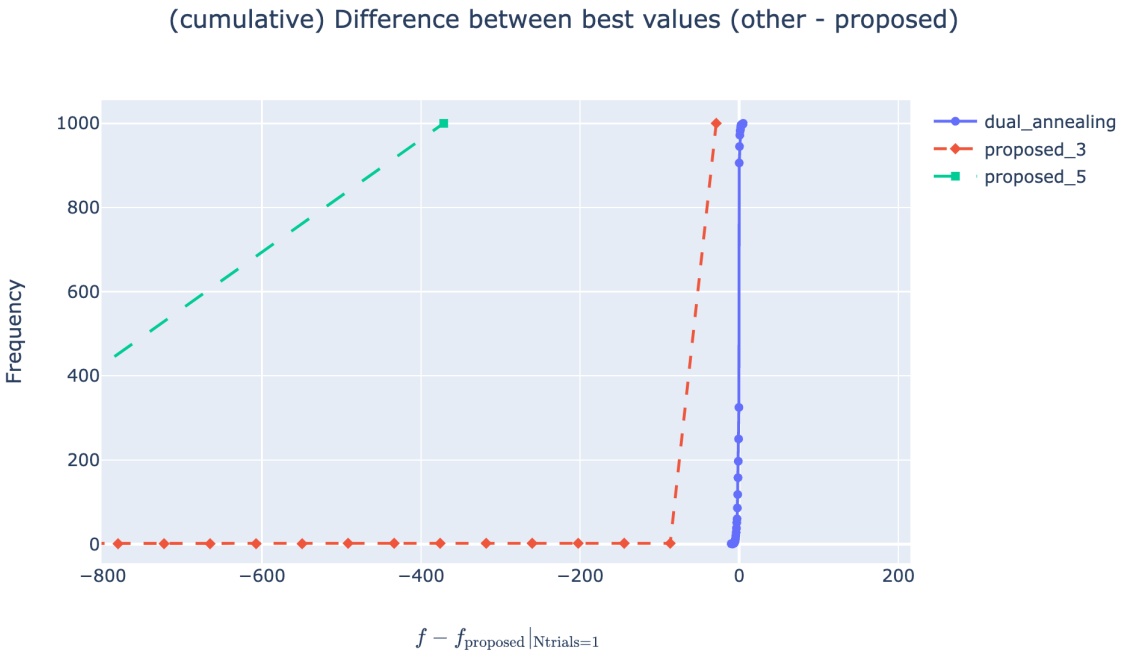


Figure 3: (a) Cumulative histograms of achieved best values (relatively to the reference solver with `Ntrials`=1) for the `dual_annealing` `scipy` optimizer and the proposed algorithm with `Ntrials`=3 and 5. (b) zoom on the plots of (a) for a better reading of the curves close to 0.

The examination of Figure 2 showing the cumulative histogram of the incremental difference in the best achieved values compared to the reference solver (the proposed one with `Ntrials`=1) shows that the solvers implemented through the `Gekko` framework were not able to solve all the problems included in the benchmark. The error message mentioned some difficulties that were due the maximum length of the symbolic relationship involved in the modeling process. This happens for high values of `nx`=50.

This being said, it is not the message of this paper to underline this shortcoming since the author is not sure whether this problem could have been by-passed by some specific manipulations (some are proposed in the warning messages).

Notice that no such remedy was searched for because the resulting qualitative conclusion would remained valid given the results already obtained on the large set of problems on which the `Gekko`-implemented solver were successful in providing solutions.

- **The computation time**. All the computations are performed on a `Apple M3 Pro, 18 Go`. The default settings of the alternative solvers implemented in `Gekko` is used with the following two options:

```
options.MAX_ITER = 5000
options.OTOL = 1e-9
```

The computation times are reported in Figure 4 where the left figure shows the ratios between the computation time of the alternative solvers and the computation time required by the proposed algorithm with `ntrials`=1. When the plots are above the horizontal gray line (corresponding to 1), this means that the computation time used by the alternative solvers is greater than the on needed by the reference proposed solver with `Ntrials`=1.

On the right subplot of Figure 4, the computation time for the proposed solver is given for all the experiments involving the reference proposed solver. Moreover Figure 5 shows the same results except that the time needed by the proposed solver with different settings of the parameter `Ntrials`=3 and 5 are shown in order to show that these last settings enables the solver to largely outperform the global `scipy` optimizer while keeping largely lower computation times.

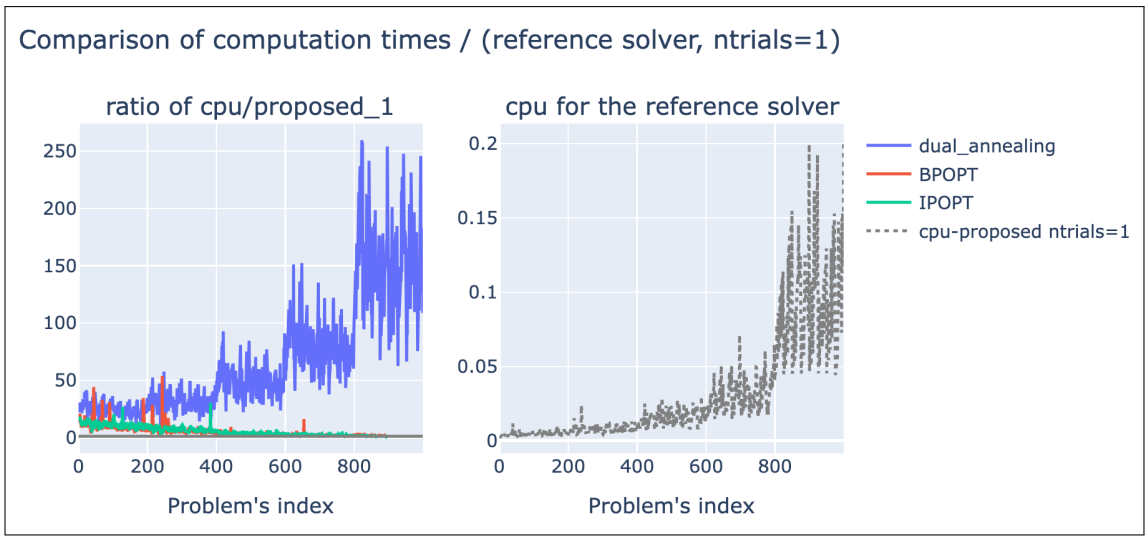


Figure 4: Histogram of the ratios of computation time relatively to the proposed one using `Ntrials=1`.(Left) when a curve is above the horizontal gray line, this means that the cpu time for the corresponding alternative solver is greater than the reference cpu time of the proposed algorithm with a single trial `Ntrials`=1. (Right) The reference cpu time of the latter algorithm.

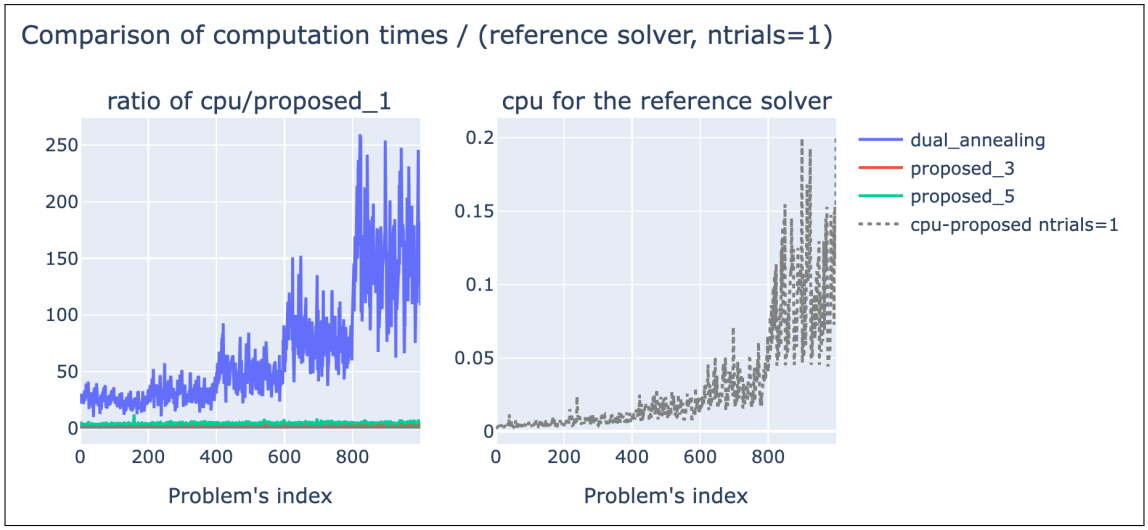


Figure 5: Histogram of the ratios of computation time relatively to the proposed one using `Ntrials=1`.(Left) when a curve is above the horizontal gray line, this means that the cpu time for the corresponding alternative solver is greater than the reference cpu time of the proposed algorithm with a single trial `Ntrials`=1. (Right) The reference cpu time of the latter algorithm.

These results coupled with the comparative performance results shown in Figures 1, 2 and 3 clearly suggest that the proposed solver, even with a single trial, provide better performance with less computation time.

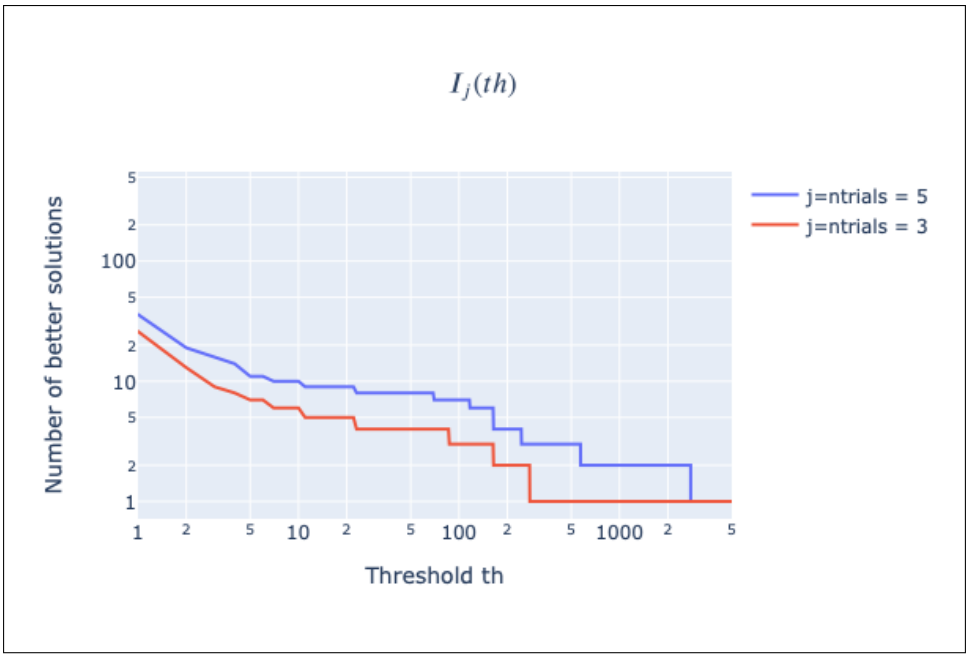


Figure 6: The number of problem's instances for which the use of `Ntrials`=3 or 5 provides better results in the sense of (4.2) depending on the threshold `th` being used.

The last investigated aspect concerns the relevance of using multiple starting points in the proposed algorithm which can be performed using higher values of the settings parameter `Ntrials`.

Recall that the enumeration aspect over each single direction is already providing a sort of partial immunity against local minima.

Nevertheless, there is no guarantee that even when no improvement is possible to obtain by changing a single direction alone, this does not mean that the current iterate is a local minimum since it would be possible to get improvement by moving more than a single component at once.

This is the *raison d'être* of the multiple-starting-point mechanism associated with the use of `Ntrials`>1.

In order to examine this aspect, the same set of 1000 experiments have been also solved using the proposed algorithm with the two following values of the `Ntrials` argument:

$$\texttt{Ntrials} \in \{3, 5\}$$

and the number of experiments for which the use of multiple starting point has improved the result by more than a threshold `th` has been reported. More precisely, the cardinality of the following `th`-dependent set:

$$I_j(\texttt{th}) := \text{card}\left\{\text{experiment} \mid \frac{f_{\text{proposed}}|_{\text{Ntrials}=j}}{f_{\text{proposed}}|_{\text{Ntrials}=1}} \leq \texttt{th}\right\} \quad j \in \{3, 5\} \tag{4.2}$$

The results are shown in Firure 6.

5. Conclusion.

In this paper, a `python` module called `optipoly` incorporating a new simple algorithm is proposed for the minimization of polynomial cost function.

The algorithm exploits the `python-interp1d` powerful computation in order to perform iterative scalar optimization along single components that are circularly updated. The algorithm is embedded inside a `Polynomial` class that exports a `solve` method that enables to run the underlying algorithm.

The comparison with some state-of-the-art freely available algorithms shows that, as far as polynomial cost is concerned, the proposed algorithm outperforms both standard non-global solvers as well as global solvers in terms of computation times as well as in terms of the quality of the best solution found.

The corresponding python module `polyopt` will be made available soon through the `pip` installation process:

```
pip install optipoly
```

Following the syntax of use mentioned throughout the paper.

REFERENCES

[1] L. Beal, D. Hill, R. Martin, and J. Hedengren, *Gekko optimization suite*, Processes, 6 (2018), p. 106, https://doi.org/10.3390/pr6080106.

[2] L. T. Biegler and V. M. Zavala, *Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization*, Computers & Chemical Engineering, 33 (2009), pp. 575–582.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., *Scikit-learn: Machine learning in python*, Journal of machine learning research, 12 (2011), pp. 2825–2830.

[4] C. Tsallis and D. A. Stariolo, *Generalized simulated annealing*, Physica A: Statistical Mechanics and its Applications, 233 (1996), pp. 395–406, https://doi.org/10.1016/0378-4371(96)00074-3.

[5] P. Virtanen et al., *Scipy 1.0: Fundamental algorithms for scientific computing in python*, Nature Methods, 17 (2020), pp. 261–272, https://doi.org/10.1038/s41592-019-0686-2.