A Machine Learning Based Software Pipeline to Pick the Variable Ordering for Algorithms with Polynomial Inputs

Dorian Florescu and Matthew England(✉)

Faculty of Engineering, Environment and Computing,
Coventry University, Coventry CV1 5FB, UK
fdorian88@gmail.com, Matthew.England@coventry.ac.uk

Abstract. We are interested in the application of Machine Learning (ML) technology to improve mathematical software. It may seem that the probabilistic nature of ML tools would invalidate the exact results prized by such software, however, the algorithms which underpin the software often come with a range of choices which are good candidates for ML application. We refer to choices which have no effect on the mathematical correctness of the software, but do impact its performance.

In the past we experimented with one such choice: the variable ordering to use when building a Cylindrical Algebraic Decomposition (CAD). We used the Python library Scikit-Learn (`sklearn`) to experiment with different ML models, and developed new techniques for feature generation and hyper-parameter selection.

These techniques could easily be adapted for making decisions other than our immediate application of CAD variable ordering. Hence in this paper we present a software pipeline to use `sklearn` to pick the variable ordering for an algorithm that acts on a polynomial system. The code described is freely available online.

Keywords: Machine learning · Scikit-learn · Mathematical software · Cylindrical algebraic decomposition · Variable ordering

1 Introduction and Context

Mathematical Software, i.e. tools for effectively computing mathematical objects, is a broad discipline: the objects in question may be expressions such as polynomials or logical formulae, algebraic structures such as groups, or even mathematical theorems and their proofs. In recent years there have been examples of software that acts on such objects being improved through the use of artificial intelligence techniques. For example, [21] uses a Monte-Carlo tree search to find the representation of polynomials that are most efficient to evaluate; [22] uses a machine learnt branching heuristic in a SAT-solver for formulae in Boolean logic; [18] uses pattern matching to determine whether a pair of elements from a specified group are conjugate; and [1] uses deep neural networks for premise selection in an automated theorem prover. See the survey article [12] in the proceedings of ICMS 2018 for more examples.

Machine Learning (ML), that is statistical techniques to give computer systems the ability to *learn* rules from data, may seem unsuitable for use in mathematical software since ML tools can only offer probabilistic guidance, when such software prizes exactness. However, none of the examples above risked the correctness of the end-result in their software. They all used ML techniques to make non-critical choices or guide searches: the decisions of the ML carried no risk to correctness, but did offer substantial increases in computational efficiency. All mathematical software, no matter the mathematical domain, will likely involve such choices, and our thesis is that in many cases an ML technique could make a better choice than a human user, so-called magic constants [6], or a traditional human-designed heuristic.

Contribution and Outline

In Sect. 2 we briefly survey our recent work applying ML to improve an algorithm in a computer algebra system which acts on sets of polynomials. We describe how we proposed a more appropriate definition of model accuracy and used this to improve the selection of hyper-parameters for ML models; and a new technique for identifying features of the input polynomials suitable for ML.

These advances can be applied beyond our immediate application: the feature identification to any situation where the input is a set of polynomials, and the hyper-parameter selection to any situation where we are seeking to take a choice that minimises a computation time. Hence we saw value in packaging our techniques into a software pipeline so that they may be used more widely. Here, by pipeline we refer to a succession of computing tasks that can be run as one task. The software is freely available as a Zenodo repository here: https://doi.org/10.5281/zenodo.3731703

We describe the software pipeline and its functionality in Sect. 3. Then in Sect. 4 we describe its application on a dataset we had not previously studied.

2 Brief Survey of Our Recent Work

Our recent work has been using ML to select the variable ordering to use for calculating a cylindrical algebraic decomposition relative to a set of polynomials.

2.1 Cylindrical Algebraic Decomposition

A *Cylindrical Algebraic Decomposition* (CAD) is a *decomposition* of ordered $\mathbb{R}^n$ space into cells arranged *cylindrically*, meaning the projections of cells all lie within cylinders over a CAD of a lower dimensional space. All these cells are (semi)-algebraic meaning each can be described with a finite sequence of polynomial constraints. A CAD is produced for either a set of polynomials, or a logical formula whose atoms are polynomial constraints. It may be used to analyse these objects by finding a finite sample of points to query and thus understand the behaviour over all $\mathbb{R}^n$. The most important application of CAD is to perform Quantifier Elimination (QE) over the reals. I.e. given a quantified formula, a CAD may be used to find an equivalent quantifier free formula[1].

CAD was introduced in 1975 [10] and is still an active area of research. The collection [7] summarises the work up to the mid-90s while the background section of [13], for example, includes a summary of progress since. QE has numerous applications in science [2], engineering [25], and even the social sciences [23].

CAD requires an ordering of the variables. QE imposes that the ordering matches the quantification of variables, but variables in blocks of the same quantifier and the free variables can be swapped[2]. The ordering can have a great effect on the time/memory use of CAD, the number of cells, and even the underlying complexity [5]. Human designed heuristics have been developed to make the choice [3,4,11,14] and are used in most implementations.

The first application of ML to the problem was in 2014 when a support vector machine was trained to choose which of these heuristics to follow [19,20]. The machine learned choice did significantly better than any one heuristic overall.

2.2 Recent Work on ML for CAD Variable Ordering

The present authors revisited these experiments in [15] but this time using ML to predict the ordering directly (because there were many problems where none of the human-made heuristics made good choices and although the number of orderings increases exponentially, the current scope of CAD application means this is not restrictive). We also explored a more diverse selection of ML methods available in the Python library `scikit-learn` (`sklearn`) [24]. All the models tested outperformed the human made heuristics.

The ML models learn not from the polynomials directly, but from features: properties which evaluate to a floating point number for a specific polynomial set. In [20] and [15] only a handful of features were used (measures of degree and frequency of occurrence for variables). In [16] we developed a new feature generation procedure which used combinations of basic functions (average, sign, maximum) evaluated on the degrees of the variables in either one polynomial or the whole system. This allowed for substantially more features and improved the performance of all ML models. The new features could be used for any ML application where the input is a set of polynomials.

The natural metric for judging a CAD variable ordering is the corresponding CAD runtime: in the work above models were trained to pick the ordering which minimises this for a given input. However, this meant the training did not distinguish between any non-optimal ordering even though the difference between these could be huge. This led us to a new definition of accuracy in [17]: to picking an ordering which leads to a runtime within $x\%$ of the minimum possible.

[1] E.g. QE would transform $\exists x, ax^2+bx+c=0 \wedge a \neq 0$ into the equivalent $b^2-4ac \geq 0$.

[2] In Footnote 1 we must decompose (x,a,b,c)-space with x last, but the other variables can be in any order. Using $a \prec b \prec c$ requires 27 cells but $c \prec b \prec a$ requires 115.

We then wrote a new version of the `sklearn` procedure which uses cross-validation to select model hyper-parameters to minimise the total CAD runtime of its choices, rather than maximise the number of times the minimal ordering is chosen. This also improved the performance of all ML models in the experiments of [17]. The new definition and procedure are suitable for any situation where we are seeking to take a choice that minimises a computation time.

3 Software Pipeline

The input to our pipeline is given by two distinct datasets used for training and testing, respectively. An individual entry in the data set is a set of polynomials that represent an input to a symbolic computation algorithms, in our case CAD. The output is a corresponding sequence of variable ordering suggestions for each set of polynomials in the testing dataset.

The pipeline is fully automated: it generates and uses the CAD runtimes for each set of polynomials under each admissible variable ordering; uses the runtimes from the training dataset to select the hyper-parameters with cross-validation and tune the parameters of the model; and evaluates the performance of those classifiers (along with some other heuristics for the problem) for the sets of polynomials in the testing dataset.

We describe these key steps in the pipeline below. Each of the numbered stages can be individually marked for execution or not in a run of the pipeline (avoiding duplication of existing computation). The code for this pipeline, written all in Python, is freely available at: https://doi.org/10.5281/zenodo.3731703.

I. Generating a Model Using the Training Dataset

(a) Measuring the CAD Runtimes: The CAD routine is run for each set of polynomials in the training dataset. The runtimes for all possible variable orderings are stored in a different file for each set of polynomials. If the runtime exceeds a pre-defined timeout, the value of the timeout is stored instead.

(b) Polynomial Data Parsing: The training dataset is first converted to a format that is easier to process into features. For this purpose, we chose the format given by the `terms()` method from the `Poly` class located in the `sympy` package for symbolic computation in Python.

Here, each monomial is defined by a tuple, containing another tuple with the degrees of each variable, and a value defining the monomial coefficient. The polynomials are then defined by lists of monomials given in this format, and a point in the training dataset consists of a list of polynomials. For example, one entry in the dataset is the set $\{235x_1 + 42x_2^2, 2x_1^2x_3 - 1\}$ which is represented as

$$[[((1,0,0),235)\,,((0,2,0),42)]\,,[((2,0,1),2)\,,((0,0,0),-1)]]\,.$$

All the data points in the training dataset are then collected into a single file called `terms_train.txt` after being placed into this format. Subsequently, the file `y_train.txt` is created storing the index of the variable ordering with the minimum computing times for each set of polynomials, using the runtimes measured in Step I(a).

(c) Feature Generation: Here each set of polynomials in the training dataset is processed into a fixed length sequence of floating point numbers, called features, which are the actual data used to train the ML models in `sklearn`. This is done with the following steps:

i. **Raw feature generation**
We systematically consider applying all meaningful combinations of the functions `average`, `sign`, `maximum`, and `sum` to polynomials with a given number of variables. This generates a large set of feature descriptions as proposed in [16]. The new format used to store the data described above allows for an easy evaluation of these features. An example of computing such features is given in Fig. 1. In [16] we described how the method provides 1728 possible features for polynomials constructed with three variables for example. This step generates the full set of feature descriptions, saved in a file called `features_descriptions.txt`, and the corresponding values of the features on the training dataset, saved in a file called `features_train_raw.txt`.

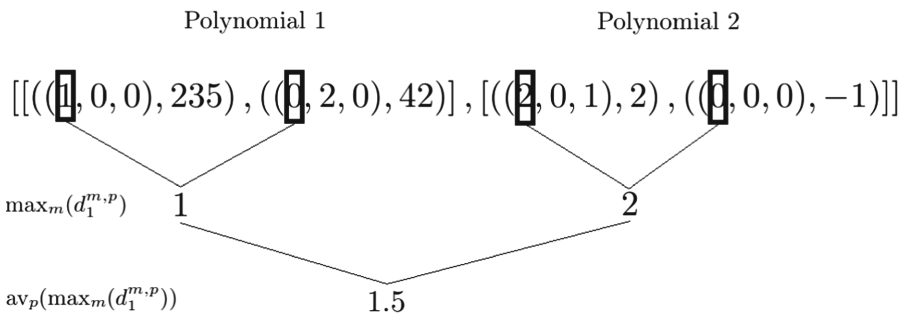

Fig. 1. Generating feature $\text{av}_p\,(\max_m\,(d_1^{m,p}))$ from data stored in the format of Section I(b). Here $d_1^{m,p}$ denotes the degree of variable x_1 in polynomial number p and monomial number m, and av_p denotes the average function computed for all polynomials [16].

ii. **Feature simplification**
After computing the numerical values of the features in Step I(c)i this step will remove those features that are constant or repetitive for the dataset in question, as described in [16]. The descriptions of the remaining features are saved in a new file called `features_descriptions_final.txt`.

iii. **Final feature generation**
The final set of features is computed by evaluating the descriptions in `features_descriptions_final.txt` for the training dataset. Even though these were already evaluated in Step I(c)i we repeat the evaluation for the

final set of feature descriptions. This is to allow the possibility of users entering alternative features manually and skipping steps i and ii. As noted above, any of the named steps in the pipeline can be selected or skipped for execution in a given run. The final values of the features are saved in a new file called `features_train.txt`.

(d) Machine Learning Classifier Training:

i. **Fitting the model hyperparameters by cross-validation**
The pipeline can apply four of the most commonly used deterministic ML models (see [15] for details), using the implementations in `sklearn` [24].
   – The K-Nearest Neighbors (KNN) classifier
   – The Multi-Layer Perceptron (MLP) classifier
   – The Decision Tree (DT) classifier
   – The Support Vector Machine (SVM) classifier

   Of course, additional models in `sklearn` and its extensions could be included with relative ease. The pipeline can use two different methods for fitting the hyperparameters via a cross-validation procedure on the training set, as described in [17]:
   – Standard cross-validation: maximizing the prediction accuracy (i.e. the number of times the model picks the optimum variable ordering).
   – Time-based cross-validation: minimizing the CAD runtime (i.e. the time taken to compute CADs with the model's choices).

   Both methods tune the hyperparameters with cross-validation using the routine `RandomizedSearchCV` from the `sklearn` package in Python (the latter an adapted version we wrote). The cross-validation results (i.e. choice of hyperparameters) are saved in a file `hyperpar_D**_**_T**_**.txt`, where `D**_**` is the date and `T**_**` denotes the time when the file was generated.

ii. **Fitting the parameters**
The parameters of each model are subsequently fitted using the standard sklearn algorithms for each chosen set of hyperparameters. These are saved in a file called `par_D**_**_T**_**.txt`.

II. Predicting the CAD Variable Orderings Using the Testing Dataset

The models in Step I are then evaluated according to their choices of variable orderings for the sets of polynomials in the testing dataset. The steps below are listed without detailed description as they are performed similarly to Step I for the testing dataset.

(a) Polynomial Data Parsing: The values generated are saved in a new file called `terms_test.txt`.

(b) Feature Generation: The final set of features is computed by evaluating the descriptions in Step I(b)ii for the testing dataset. These values are saved in a new file called `features_test.txt`.

(c) Predictions Using ML: Predictions on the testing dataset are generated using the model computed in Step I(c). The model is run with the data in Step II(a)ii, and the predictions are stored in a file called `y_D**_**_T**_**_test.txt`.

(d) Predictions Using Human-Made Heuristics: In our prior papers [15–17] we compared the performance of the ML models with the human-designed heuristics in [4] and [11]. For details on how these are applied see [15]. Their choices are saved in two files entitled `y_brown_test.txt` and `y_sotd_test.txt`, respectively.

(e) Comparative Results: Finally, in order to compare the performance of the proposed pipeline, we must measure the actual CAD runtimes on the testing dataset. The results of the comparison is saved in a file with the template: `comparative_results_D**_**_T**_**.txt`.

Adapting the Pipeline to Other Algorithms

The pipeline above was developed for choosing the variable ordering for the CAD implementation in Maple's Regular Chains Library [8,9]. But it could be used to pick the variable ordering for other procedures which take sets of polynomials as input by changing the calls to CAD in Steps I(a) and II(e) to that of another implementation/algorithm. Step II(d) would also have to be edited to call an appropriate competing heuristic.

4 Application of Pipeline to New Dataset

The pipeline described in the previous section makes it easy for us to repeat our past experiments (described in Sect. 2) for a new dataset. All that is needed to do is replace the files storing the polynomials and run the pipeline.

To demonstrate this we test the proposed pipeline on a new dataset of randomly generated polynomials. We are not suggesting that it is appropriate to test classifiers on random data: we simply mean to demonstrate the ease with which the experiments in [15–17] that originally took many man-hours can be repeated with just a single code execution.

The randomly generated parameters are: the degrees of the three variables in each polynomial term, the coefficient of each term, the number of terms in a polynomial and the number of polynomials in a set. The means and standard deviations of these parameters were extracted from the problems in the `nlsat` dataset[3], which was used in our previous work [15] so that the dataset is of a

[3] https://cs.nyu.edu/~dejan/nonlinear/.

Table 1. The comparative performance of DT, KNN, MLP, SVM, the Brown and sotd heuristics on the testing data for our randomly generated dataset. A random prediction, and the virtual best (VB) and virtual worst (VW) predictions are also included.

	DT	KNN	MLP	SVM	`Brown`	`sotd`	rand	VB	VW
Prediction time (s)	$4.8 \cdot e^{-4}$	0.68	$2.8 \cdot e^{-4}$	0.99	53.01	15 819			
Total time (s)	6 548	6 610	6 548	6 565	6 614	22 313	16 479	5 610	25 461

comparable scale. We generated 7500 sets of random polynomials, where 5000 were used for training, and the remaining 2500 for testing.

The results of the proposed processing pipeline, including the comparison with the existing human-made heuristics are given in Table 1. The prediction time is the time taken for the classifier or heuristic to make its predictions for the problems in the training set. The total time adds to this the time for the actual CAD computations using the suggested orderings. We do not report the training time of the ML as this is a cost paid only once in advance. The virtual solvers are those which always make the best/worst choice for a problem (in zero prediction time) and are useful to show the range of possible outcomes. We note that further details on our experimental methodology are given in [15–17].

As with the tests on the original dataset [15,16] the ML classifiers outperformed the human made heuristics, but for this dataset the difference compared to the Brown heuristic was marginal. We used a lower CAD timeout which may benefit the Brown heuristic as past analysis shows that when it makes suboptimal choices these tend to be much worse. We also note that the relative performance of the Brown heuristic fell significantly when used on problems with more than three variables in [17]. The results for the sotd heuristic are bad because it had a particularly long prediction time on this random dataset. We note that there is scope to parallelize sotd which may make it more competitive.

5 Conclusions

We presented our software pipeline for training and testing ML classifiers that select the variable ordering to use for CAD, and described the results of an experiment applying it to a new dataset.

The purpose of the experiment in Sect. 4 was to demonstrate that the pipeline can easily train classifiers that are competitive on a new dataset with almost no additional human effort, at least for a dataset of a similar scale (we note that the code is designed to work on higher degree polynomials but has only been tested on datasets of 3 and 4 variables so far). The pipeline makes it possible for a user to easily tune the CAD variable ordering choice classifiers to their particular application area.

Further, with only a little modification, as noted at the end of Sect. 3, the pipeline could be used to select the variable ordering for alternative algorithms that act on sets of polynomials and require a variable ordering. We thus expect the pipeline to be a useful basis for future research and plan to experiment with its use on such alternative algorithms in the near future.

Acknowledgements. This work is funded by EPSRC Project EP/R019622/1: *Embedding Machine Learning within Quantifier Elimination Procedures*. We thank the anonymous referees for their comments.

References

1. Alemi, A., Chollet, F., Een, N., Irving, G., Szegedy, C., Urban, J.: DeepMath - deep sequence models for premise selection. In: Proceedings of the NIPS 2016, pp. 2243–2251 (2016). https://doi.org/10.5555/3157096.3157347
2. Bradford, R., et al.: Identifying the parametric occurrence of multiple steady states for some biological networks. J. Symb. Comput. **98**, 84–119 (2020). https://doi.org/10.1016/j.jsc.2019.07.008
3. Bradford, R., Davenport, J.H., England, M., Wilson, D.: Optimising problem formulation for cylindrical algebraic decomposition. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) CICM 2013. LNCS (LNAI), vol. 7961, pp. 19–34. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39320-4_2
4. Brown, C.: Companion to the tutorial: cylindrical algebraic decomposition. In: Presented at ISSAC 2004 (2004). http://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/handout.pdf
5. Brown, C., Davenport, J.: The complexity of quantifier elimination and cylindrical algebraic decomposition. In: Proceedings of the ISSAC 2007, pp. 54–60. ACM (2007). https://doi.org/10.1145/1277548.1277557
6. Carette, J.: Understanding expression simplification. In: Proceedings of the ISSAC 2004, pp. 72–79. ACM (2004). https://doi.org/10.1145/1005285.1005298
7. Caviness, B., Johnson, J.: Quantifier Elimination and Cylindrical Algebraic Decomposition. Texts & Monographs in Symbolic Computation. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-7091-9459-1
8. Chen, C., Moreno Maza, M., Xia, B., Yang, L.: Computing cylindrical algebraic decomposition via triangular decomposition. In: Proceedings of the ISSAC 2009, pp. 95–102. ACM (2009). https://doi.org/10.1145/1576702.1576718
9. Chen, C., Moreno Maza, M.: Quantifier elimination by cylindrical algebraic decomposition based on regular chains. J. Symb. Comput. **75**, 74–93 (2016). https://doi.org/10.1016/j.jsc.2015.11.008
10. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975). https://doi.org/10.1007/3-540-07407-4_17. Reprinted in [7]
11. Dolzmann, A., Seidl, A., Sturm, T.: Efficient projection orders for CAD. In: Proceedings of the ISSAC 2004, pp. 111–118. ACM (2004). https://doi.org/10.1145/1005285.1005303
12. England, M.: Machine learning for mathematical software. In: Davenport, J.H., Kauers, M., Labahn, G., Urban, J. (eds.) ICMS 2018. LNCS, vol. 10931, pp. 165–174. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96418-8_20
13. England, M., Bradford, R., Davenport, J.: Cylindrical algebraic decomposition with equational constraints. J. Symb. Comput. **100**, 38–71 (2020). https://doi.org/10.1016/j.jsc.2019.07.019

14. England, M., Bradford, R., Davenport, J.H., Wilson, D.: Choosing a variable ordering for truth-table invariant cylindrical algebraic decomposition by incremental triangular decomposition. In: Hong, H., Yap, C. (eds.) ICMS 2014. LNCS, vol. 8592, pp. 450–457. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44199-2_68
15. England, M., Florescu, D.: Comparing machine learning models to choose the variable ordering for cylindrical algebraic decomposition. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) CICM 2019. LNCS (LNAI), vol. 11617, pp. 93–108. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23250-4_7
16. Florescu, D., England, M.: Algorithmically generating new algebraic features of polynomial systems for machine learning. In: Proceedings of the SC^2 2019. CEUR-WS, vol. 2460 (2019). http://ceur-ws.org/Vol-2460/
17. Florescu, D., England, M.: Improved cross-validation for classifiers that make algorithmic choices to minimise runtime without compromising output correctness. In: Slamanig, D., Tsigaridas, E., Zafeirakopoulos, Z. (eds.) MACIS 2019. LNCS, vol. 11989, pp. 341–356. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43120-4_27
18. Gryak, J., Haralick, R., Kahrobaei, D.: Solving the conjugacy decision problem via machine learning. Exp. Math. **29**(1), 66–78 (2020). https://doi.org/10.1080/10586458.2018.1434704
19. Huang, Z., England, M., Wilson, D., Bridge, J., Davenport, J., Paulson, L.: Using machine learning to improve cylindrical algebraic decomposition. Math. Comput. Sci. **13**(4), 461–488 (2019). https://doi.org/10.1007/s11786-019-00394-8
20. Huang, Z., England, M., Wilson, D., Davenport, J.H., Paulson, L.C., Bridge, J.: Applying machine learning to the problem of choosing a heuristic to select the variable ordering for cylindrical algebraic decomposition. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM 2014. LNCS (LNAI), vol. 8543, pp. 92–107. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_8
21. Kuipers, J., Ueda, T., Vermaseren, J.: Code optimization in FORM. Comput. Phys. Commun. **189**, 1–19 (2015). https://doi.org/10.1016/j.cpc.2014.08.008
22. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
23. Mulligan, C.B., Davenport, J.H., England, M.: TheoryGuru: a mathematica package to apply quantifier elimination technology to economics. In: Davenport, J.H., Kauers, M., Labahn, G., Urban, J. (eds.) ICMS 2018. LNCS, vol. 10931, pp. 369–378. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96418-8_44
24. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011). http://www.jmlr.org/papers/v12/pedregosa11a.html
25. Sturm, T.: New domains for applied quantifier elimination. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2006. LNCS, vol. 4194, pp. 295–301. Springer, Heidelberg (2006). https://doi.org/10.1007/11870814_25