The SDEval Benchmarking Toolkit

Albert Heinle
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
aheinle@uwaterloo.ca

Viktor Levandovskyy
Lehrstuhl D für Mathematik
RWTH Aachen University
Aachen, Germany
levandov@math.rwth-aachen.de

Abstract

In this paper we will present SDeval, a software project that contains tools for creating and running benchmarks with a focus on problems in computer algebra. It is built on top of the Symbolic Data project, able to translate problems in the database into executable code for various computer algebra systems. The included tools are designed to be very flexible to use and to extend, such that they can be easily deployed even in contexts of other communities. We also address particularities of benchmarking in the field of computer algebra.

Furthermore, with SDEval, we provide a feasible and automatable way of reproducing benchmarks published in current research works, which appears to be a difficult task in general due to the customizability of the available programs.

1 Introduction

Benchmarking of software – i.e. measuring the quality of results and the time resp. memory consumption for a given, standardized set of examples as input – is a common way of evaluating implementations of algorithms in many areas of industry and academia. For example, common benchmarks for satisfiability modulo theorems (SMT) solvers are collected in the standard library SMT-LIB [BST10], and the advantages of various solvers like Z3 [DMB08] or CVC4 [BCD$^+$11] are revealed with the help of those benchmarks.

Considering the field of computer algebra, there could be various benchmarks for the different computation problems. Sometimes, one can find common problem instances throughout papers dealing with the same topics, but often there is no standard collection and authors use examples best to their knowledge. For the calculation of Gröbner bases for example, there is a collection of ideals that often appear when a new or modified approach accompanied by an implementation is presented (e.g. in [Neu12], the author uses the classical examples Katsura-n, $n \in \{11,12\}$, from [KFI$^+$87] and Cyclic-m, $m \in \{8,9\}$, from [BH08] to evaluate his new implementation). Regarding the computation on that set, the new implementation is then compared to existing and available ones. Note, that even in computations of Gröbner bases of polynomial ideals there are parameters, defining the concrete instance of the computation task, such as the ground field and the ordering on monomials. Different computer algebra systems vary in the implemented functionality, see e.g. [LRP07] for the comparison.

An outstanding systematic and transparent practice has been shown in 2001 by the computer algebra lab, lead by V. P. Gerdt, of the "Joint Institute For Nuclear Research", on their website about the progress in research of computing Janet and Gröbner bases of complicated polynomial systems (`http://invo.jinr.ru/`).

Nevertheless, in many areas, there is rarely a standard test set, and often the stated timings and computed results are hard to reconstruct due to different parameters in algorithms that can be set.

Another difficulty is the fair evaluation on how much time is consumed; we will discuss this topic detailed in section 2.

A database containing a collection of various instances of problems coming especially from the computer algebra community is given by the Symbolic Data project [Grä09]. It started more than 10 years ago, and its team of developers is steadily extending the collection of problem instances together with precise references to their origins. Furthermore, the ways of accessing the information in the database and interlinking it with other databases are being kept up to date. For the latter e.g., the techniques of the so called "semantic web" movement have been applied (for more details consider [GNJ13]). The entries are given in the `XML` data format, which makes it easy to parse them since almost every programming language nowadays provides `XML` support. All these arguments lead to the decision to use Symbolic Data as the underlying database for our project.

We are going to further discuss particularities concerning benchmarking especially for the computer algebra community and present SDEval, a benchmarking toolbox written in Python covering the following two main tasks:

(i) Creating benchmark sets with the help of the problem instances provided by the Symbolic Data database entries.

(ii) Running benchmarks, with a flexible (i.e. cross-community adaptable) interface that makes reproduction as simple as possible.

For item (i), we implemented for certain computational problems (e.g. calculation of a Gröbner basis) translators of respective problem instances from Symbolic Data into executable code for a set of computer algebra systems.

Item (ii) has a broader range of possible uses. First of all, it provides a way to run arbitrary programs on different inputs. Optionally, it monitors the computations and terminates programs automatically if they exceed a user-given time or memory limit. The results are generated and presented in a transparant and reproducible way. We envision for the future that `tar`-balls of the folders generated by SDEval would be published with computation-focused papers, so that it becomes easier to verify results of the authors.

There is an intersection between this paper and the preprint [HLN13]. In the latter, the main focus was on the interconnection between SDEval and the Symbolic Data project, whereas this paper solely focuses on the functionalities and motivations of SDEval.

This work is structured as follows. Section 2 will deal with the design principles and functionalities of SDEval. We will elaborate on item (i) and (ii) from above and show how one can use and extend/adjust it to individual computation problems if needed. We will address related work in Section 3 and finish by outlining future tasks in Section 4.

The current version of the presented toolkit SDEval can be found at `github.com/ioah86/symbolicdata`. The latest information on Symbolic Data are available at `http://symbolicdata.org`.

Note, that on our own we have used SDEval in our studies with respect to several ongoing projects, see e.g. [CCH$^+$11], [HL13] and [GHL14].

2 SDEval

2.1 Particularities about Benchmarking in Computer Algebra

2.1.1 Challenges.

Writing benchmarks in the field of computer algebra differs from other benchmarking tasks. A collection of appearing challenges is the following.

- Sometimes, the results of computations are not unique; that is, several non identically equal outputs can be equivalently correct. It is not always possible to find a canonical form for an output. Even if this is the case, the transformation of output into the canonical form can be quite costly. Moreover, the latter transformation is not necessarily provided by every single computer algebra system.
- Related to the previous item: If an answer is not unique, then the evaluation of the correctness of the output is often far from trivial. In some cases the correctness-evaluation of certain results is even subject of on-going research.
- The field of computer algebra deals with a large variety of topics, even though it can be divided into classes of areas where certain common computational problems do appear. Thus, there need to be collections of benchmarks, optimally one as a standard for each class. The benchmark creation process should be flexible to be applicable in a wide range of areas.
- Considering input formats, many computer algebra systems are going their own ways, i.e. for many computation problems, telling the respective system what to calculate differ a lot. The source of this problem is that the way of representing certain given mathematical objects may also not be unified across the community.

We tried to address these challenges as much as possible when designing our toolkit.

In particular, the first item is something that differs the creation of benchmarks for computer algebra problems from most other fields of studies.

The second item leads to one of the design decisions we made for SDEval, namely that we provide an interface for decision routines, and partially include some of them as examples how such routines could be added. Then, a particular community can deal with this question based on their problems, and provide SDEval with the information on what routine to call to obtain an answer.

2.1.2 Correct and Feasible Time Measurement.

Another seemingly trivial, yet controversial question is the correct time measure of computations, as mentioned in the introduction. It is very common in computer algebra systems to provide a time measuring functionality, and many of the timings provided in papers were calculated using those commands, since it is easily available.

Nevertheless, this methodology is questionable. Often one cannot verify their validity due to e.g. their source not being open. Furthermore, sometimes run-time-benefiting calculations are already done during the initialization phase; therefore one has to specify clearly where to start the provided time measurement. If one makes use of the implemented techniques, every program has to be analyzed in detail to find the correct spot to start the time counting in order to make the comparison fair. Hence, the use of system-provided time measuring is not practical for fair comparisons.

A widely spread method in software development is to run programs with the `time` command provided with Unix based operating systems (a similar program for Microsoft Windows is `timeit`, contained in Microsofts Server Resource Toolkit). Even though the time for parsing input – which is in general not the complex part about the computations done in computer algebra – would then also being taken into account, we decided that this method is the best choice for SDEval.

It has also another benefit: We are interested in extracting the timing results from the output files in an automated way, and there is a standard for providing timings given by the IEEE standard `IEEE Std 1003.2-1992 (‘‘POSIX.2’’)`; the `time` command can be instrumented using a parameter to provide its output according to this standard. Arranging this format for the output with the help of the included time measurement mechanisms in computer algebra systems can be regarded as an infeasible requirement for a user.

2.2 The Creation of a Benchmark Suite

2.2.1 Basic Terminology.

Let us start with defining some terminology we want to use throughout this section. This will serve the purpose of a better understanding of the design principles of SDEval.

Definition 1 (SD-Table) *An SD-Table denotes a table with computation problems given in the* Symbolic Data *project.*

Example 1 (SD-Table) *An example for an SD-Table is the table that contains instances of ideals in a polynomial ring over $\mathbb{Q}$ using integer coefficients. These instances can be used e.g. for Gröbner basis computations. The abbreviation chosen by the* Symbolic Data *project for this table is* `IntPS`.

Definition 2 (Problem Instance) *A problem instance is in our context a representation of a concrete input – aligned to the* Symbolic Data *format – that can be used for one or more algorithms. The input values for the chosen algorithm are contained in this problem instance. A problem instance is always contained in an SD-Table.*

Example 2 (Problem Instance) *A problem instance is for example the entry* `Amrhein` *(an integer polynomial system taken from [AGK96]) in the SD-Table* `IntPS`*. It contains the list of variables' names and a collection of polynomials forming the generators of the respective ideal. The concrete system is shown in Figure 1.*

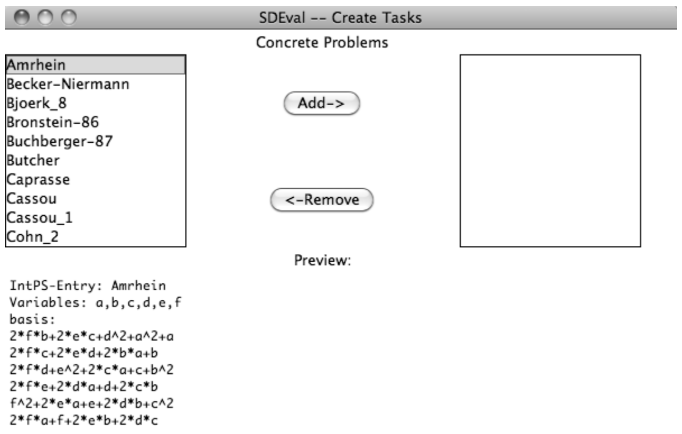
Definition 3 (Computation Problem) *A computation problem is a concrete and completely specified member of a family of algorithms. In the context of* SDEval*, it specifies which computations we want to perform on certain problem instances.*

A selection of computation problems is already provided in the SD-Table `COMP`*. The selection can be extended by the user.*

Example 3 (Computation Problem) *A computation problem is for example the computation of a Gröbner basis given an ideal over a polynomial ring over $\mathbb{Q}$ using the lexicographic ordering (abbr.* `GB_Z_lp`*).*

Definition 4 (Task) *A task consists of a computation problem, a selection of problem instances that are suitable as inputs for it and a collection of computer algebra systems that implement algorithms for the computation problem.*

Figure 1: The selection of the problem instance from integer polynomial systems



Amrhein
Becker-Niermann
Bjoerk_8
Bronstein-86
Buchberger-87
Butcher
Caprasse
Cassou
Cassou_1
Cohn_2

```
IntPS-Entry: Amrhein
Variables: a,b,c,d,e,f
basis:
2*f*b+2*e*c+d^2+a^2+a
2*f*c+2*e*d+2*b*a+b
2*f*d+e^2+2*c*a+c+b^2
2*f*e+2*d*a+d+2*c*b
f^2+2*e*a+e+2*d*b+c^2
2*f*a+f+2*e*b+2*d*c
```

2.2.2 Automated Creation of Benchmarks

Now that we have defined some basic terminology, we will address how a benchmark suite can be generated using the problem instances given in the SD-Tables. This part of SDEval addresses e.g. developers, who want to compare the running time of their implementations with those of available software without the necessity of becoming familiar with all of the available systems. Additionally, it addresses mathematicians who discovered a certain instance for a computational problem and want to examine what computer algebra systems are able to solve it and what solutions are provided, as they might differ – depending on the uniqueness of the result – for the different systems.

The SDEval project contains two Python programs that can do this job: `ctc.py` and `create_tasks_gui.py`. The first one is a command-line program, the second one provides a graphical user interface. Those scripts perform the following three steps

1. The user chooses from a set of currently supported computation problems.
2. After that, the script collects possible problem instances across the SD-Tables and presents them to the user. One can pick the desired problem instances that should be included in the benchmark. An illustration of this step is given in Figure 1.
3. In the last step, besides setting configuration parameters, the user selects from a set of computer algebra systems for which it is known that they contain implementations of the algorithms that solve the selected computation problem. Furthermore, the user enters the calling commands to execute those systems on the machine she/he wants the computation to be run.

After these three steps, the user confirms his or her choices and a folder, from now on referred to as *taskfolder*, is generated. This folder containing executable files for the selected computer algebra systems, a Python script to run all the calculations and some adjustable configuration files (e.g. if the user wants to change call parameters for a computer algebra system). The taskfolder can then be sent to the machine where the computations are intended to be run. The concrete structure is given as in Figure 2. As a recent addition, we provided a functionality that output-analyzing scripts can be included and would automatically be run after completion of the computation by the computer algebra system. For the supported computation problems and the supported computer algebra systems, we already provide scripts to do a light analysis on the output. Light analysis in this context means that it checks whether there is an output or whether the calculation has been terminated.

As outlined before, the creation tool is very flexible and easily extensible. This is due to the object oriented nature of the code written in Python. One can specify new computation problems, and declare which problem instances can be chosen as inputs. The respective code for the computer algebra systems can be added in a template-fashion and does not require familiarity with the particular concepts of Python.

Figure 2: Folder structure of a taskfolder

```
+ TaskFolder
| - runTasks.py //For Running the task
| - taskInfo.xml //Saving the Task in XML Structure
| - machinesettings.xml//The Machine Settings in XML form
| + classes //All classes of the SDEval project
| + casSources //Folder containing all executable files
| | + SomeProblemInstance1
| | | + ComputerAlgebraSystem1
| | | | - executablefile.sdc //Executable code for CAS
| | | | - template_sol.py //Script to analyze the output of the CAS
| | | + ComputerAlgebraSystem2
| | | | - executablefile.sdc
| | | + ...
| | + SomeProblemInstance2
| | | + ...
| | + ...
```

2.3 Running a Benchmark Suite

General Assumption 1: Whereas the creation of the benchmark suite is possible on any machine where PYTHON is installed, the running routine requires a machine running with a UNIX-like operating system (e.g. LINUX or MAC OS X). We require the `time` command or some equivalent to be supported, which is in general always the case on UNIX systems. If one wants to use an equivalent, it needs to be able to provide an output according to the IEEE standard `IEEE Std 1003.2-1992 (‘‘POSIX.2’’)`.
General Assumption 2: Calculations are run within a terminal. This decision was made due to the fact that calculations are often sent to a compute server. The connection to that server is in general provided through a terminal interface.

The running of a benchmark is closely connected to the taskfolder as presented in the previous section. As one can see in Figure 2, it contains a PYTHON script called `runTasks.py`. One can either generate an individual taskfolder using the design principles given in the documentation (see Figure 2 for the general structure), or one can use a taskfolder generated by the task creation scripts.

If one executes `runTasks.py`, all the stored scripts for all the contained computer algebra systems will be run consequently. Using execution parameters, one can instruct the script to do the following:

- Automatically kill a process once a user-provided CPU time limit is reached.
- Automatically kill a process once a user-provided memory consumption limit is reached.
- Run a user-provided number of processes in parallel.

Example 4 *Within a taskfolder, a call of*

```
$> python runTasks.py -c240 -m100000000 -j4
```

will start the execution process. The computer algebra systems are terminated if they take more than four minutes to run on a problem instance (indicated by `-c240`*, where 240 stands for 240s) or if they use more than approx. 100MB of memory (indicated by* `-m100000000`*, where the unit used is bytes). Futhermore, the user wants to have up to four processes to be run in parallel (indicated by* `-j4`*).*

The script will create — if not yet existent — a sub-folder within the taskfolder named `results`. Within `results`, there will be a folder named by the time stamp when `runTasks.py` was executed, where it will store the results of the computations, some monitoring information about the executed scripts (in form of HTML and XML files) and files containing information about the machine where the calculation is run on (detailed information on CPU, memory and operating system).

During the execution process, the user can feel free to terminate manually a running process without having to restart `runTasks.py`. It will simply continue with the next waiting program on the next script in the queue. If an output analyzing script is provided, there will be an error indicated in the HTML resp. XML table afterwards. Otherwise it will be just marked as “completed”.

This design of the benchmark execution part has the following benefit. Future authors that execute their scripts on certain files could provide their taskfolder with the paper they submitted. Then everyone can see the results (i.e. the outputs of the programs), and verify the timings using the calculated table. Furthermore, they can run the calculation using `runTasks.py` after adjusting the configuration to their machine (i.e. replacing the call commands for the computer algebra systems to those used on one's machine). We are already adapting this practice, as one can find the timings of the papers [GHL14] and [HL13] on one of our personal websites (`https://cs.uwaterloo.ca/~aheinle/software_projects.html`).

There are further uses of the running routines. As we can see, the execution of the benchmarks is completely detached from the creation part. This means, that a customized taskfolder can be created, defining programs one wants to run and provide the inputs and scripts to analyse the outputs inside the `casSources` folder.

Even though the routines were designed to fit especially the needs of the computer algebra community, the principles can be used for almost any kind of program.

Another use of the taskfolder and the contained PYTHON-program would be to keep track of the development process of a software project over time. Executing the `runTasks.py` script after every version change would reveal profiling information on the different examples. The profiling can be automatized since the timing-data after every run is stored in an XML file.

The following examples will illustrate the flexibility and the ease of adjustment of the taskfolder.

Example 5 *Assume the user already has a taskfolder. Now, he or she encounters a new, interesting problem instance and intends to add it to the existing problem instances in the taskfolder. There are two ways of doing it:*

- *If the user is familiar with every computer algebra system that is used in this taskfolder, the user creates the respective scripts and adds the problem with the scripts as a new subfolder to* `casSources`*. Then, it remains to add an entry in the* `taskInfo.xml` *file. In particular, the entry is given by the following lines:*
  ```
  <probleminstance>
    myNewExample
  </probleminstance>
  ```
  After that, the example will be considered with the next run.
- *If the user is not familiar with the computer algebra systems in use, then an entry in the database of* SYMBOLIC DATA *has to be made, which is a simple* XML *file. After that, the user can use our tool to automatically generate code for the computer algebra systems that have functionality for the respective computation problem.*

Example 6 *As in the previous example, assume that the user is already in possession of a taskfolder. Now he or she wants, in addition to the considered computer algebra systems, benchmark a personal, maybe self written program on the examples. All the user has to do is then generate for every problem instance in the folder* `casSources` *a subfolder with the respective script. After that, the user specifies how the program is called (parameters, options, etc.) in the* `MachineSettings.xml` *file, registers it in the* `taskInfo.xml` *and then the program will be considered in the next call of* `runTasks.py`*.*

Example 7 *By using SDEval ourselves, we also have encountered the following scenario. We generated a taskfolder with a large set of problem instances. After running the computer algebra systems on these problem instances, we realized that currently some cannot be solved in a feasible amount of time. Thus, until there is a new version of one of the used computer algebra systems, we want to exclude the example when executing* `runTasks.py`*. This can be done by simply commenting out the respective entries in the* `taskInfo.xml` *file.*

The same can be done to a computer algebra system which performs poorly in comparison to others, i.e. the user can comment it out until a new version appears.

2.4 Ways of Customizing and Contributing to SDEval

We have seen in the last section that the part of the execution of the respective programs on the problem instances is highly customizable. There are also ways for customization of the part where one creates benchmarks.

2.4.1 Adding Templates for Computer Algebra Systems.

The templates can be found in the folder `templates` in the SDEval project. One simply has to add a new folder named after the new computer algebra system, and within this folder there must be a file with the function that generates the code. Optionally, one can also write a script that analyzes the output of the respective computer algebra system. The function headers themselves can be copied from the other, already available templates, and one only has to adjust the respective code lines for the computer algebra system one wants to add. For this, there is no deep knowledge of Python needed. For more details consider the Q&A file in the documentation.

Example 8 *Let us consider a possible template for* Singular *to create executable code to calculate a Gröbner basis using the lexicographic ordering for a given problem instance coming from the SD-Table* `IntPS`*:*

```
def generateCode(vars, basis):
    """
    [Documentation lines]
    """
    result = """
ring R = 0,(%s),lp;
ideal I = %s;
ideal J = std(I);
print(J);\n\
$
""" % (",".join(vars),",\n".join(basis))
    return result
```

As one can see, all it takes is to create a formatted string, and the respective values will be provided by the input parameters of the function `generateCode`*, which is the standardized in* SDEval.

2.4.2 Adding New Sets of Problem Instances.

Communities can add new tables with problem instances into the SDEval project. For associating it with a given computation problem, this table has to be added to the supported sets of a computation problem within the project.

2.4.3 Adding New Computation Problems.

New computation problems can be divided into two kinds: the ones where the inputs of respective algorithms can be derived from existent tables, and the ones where in addition new sets of problem instances have to be added. The latter category requires more work. For the first one, a user can either choose the way of writing a representative class for the computation problem within the SDEval project, or contact the project team with an request to add this computation problem to SDEval due to its importance.

3 Related Work

StarExec [SST12]: This is an infrastructure especially for the logic solver communities. Its main focus is to provide a platform for managing benchmark libraries and run solver competitions. It is widely used in conferences based on logic solving to evaluate the benefits of new approaches. Moreover, it includes translators of problems between the different communities dealing with logic solving. Calculations are always run on the same hardware, therefore results can directly be compared to all other benchmarks that were run before without taking hardware differences into consideration. The main difference to our project is that it provides less flexibility for the individual researcher to define customized computation problems and submit problem instances. Furthermore, as the input data comes from the logic solver community, the input is standardized and every program accepts the same types of files. For computer algebra systems, this is different, as stated earlier.

homalg [BR08]: Focusing on constructive homological algebra, the homalg project provides an abstract structure for abelian categories and is distributed as a package of the computer algebra system GAP [GAP13]. For time critical computations, it allows the usage of other computer algebra systems, i.e. the task is translated to the respective system and then executed. This corresponds to the translation part of the SDEVAL project for the supported computation problems.

SAGE [S$^+$08]: The popular computer algebra system SAGE provides as an optional package an interface to the database of integer polynomial systems (`IntPS`) of the SYMBOLIC DATA project. One can directly load those problem instances as objects in SAGE for further calculations and apply the implemented/wrapped algorithms on them.

4 Conclusion and Future Work

We have presented a benchmarking tool named SDEVAL, which is built on top of the SYMBOLIC DATA project. In this paper, we addressed the particularities of benchmarking in the field of computer algebra, and with SDEVAL, we have presented a flexible, extensible and easy-to-use tool that is designed to accept the challenge.

Moreover, we introduced a practice how the reproduction and the analysis of computations with their timings would become more feasible in the future. Our approach for that is the taskfolder containing the benchmark program and the respective input files.

A future task will be to extend the benchmark creation tool to contain both more computer algebra systems and computation problems. As we have outlined in the paper, these extensions are easy tasks due to the chosen design of SDEVAL. The output interpretation routines are very basic at the current stage. In fact, they are just checking if feasible output can be extracted or not. In the future, we plan to implement thorough tests with which one can determine the correctness of the outputs. For that, one has to consider every computation problem in a detailed way and we hope for support from the communities to accomplish that.

Further possibilities and details about how to customize SDEval can be found in the documentation of SDEVAL. We would be happy to receive contributions and suggestions from users. But of course, even though we tried to simplify the processes of contributing as much as we could, it would take time from a person outside the project to obtain a basic understanding of the whole system. If one is not willing to contribute for this reason, but would like to see certain additions to the toolkit, please contact the authors. We are very thankful for any kind of input that can help to extend the functionality of SDEVAL.

About the timing-results of a benchmark, we plan to write a toolset to analyze these. By now, the timings that we are collecting are saved within XML-files. One can use this information to draw trends when applying different versions of computer algebra systems for example. An environment close to a testing-suite is on the agenda of the SDEVAL project.

As benchmarking is a very wide-ranged topic, we will continuously consider further challenges – maybe caused by not yet considered computation problems – that we have not dealt with in the present state. It remains a practically relevant and interesting problem.

Acknowledgements

We thank the “Deutsche Forschungsgesellschaft” (DFG) for partial financial support for the development of SDEVAL (“DFG Priority Project SPP 1489”). in the context of the “DFG Priority Project SPP 1489: Algorithmic and Experimental Methods in Algebra, Geometry and Number Theory.”

The authors are especially grateful to Prof. Dr. Hans-Gert Gräbe (Universität Leipzig), the major maintainer of SYMBOLICDATA project, for his encouragement and the numerous fruitful discussions we had with him. Moreover, we thank Andreas Nareike (Leipzig) for valuable discussions and Benjamin Schnitlzer (Aachen) for his collaboration on the SDEVAL project.

The authors thank Prof. Dr. Johannes Waldmann (HTWK Leipzig) for his introduction to the benchmarking practices of the logic solver communities during the SYMBOLIC DATA workshop.

Special thanks go to Prof. Dr. Mark Giesbrecht (University of Waterloo) for his helpful suggestions and valuable comments.

References

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[BCD$^+$11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.

[Neu12] Severin Neumann. Parallel reduction of matrices in Gröbner bases computations. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 7442 of *Lecture Notes in Computer Science*, pages 260–270. Springer Berlin Heidelberg, 2012. URL `http://dx.doi.org/10.1007/978-3-642-32973-9_22`.

[KFI$^+$87] S. Katsura, W. Fukuda, S. Inawashiro, N. M. Fujiki, and R. Gebauer. Distribution of effective field in the Ising spin glass of the $\pm j$ model at $T=0$. *Cell Biophysics*, 11(1):309–319, 1987.

[BH08] Goran Bjorck and Uffe Haagerup. All cyclic p-roots of index 3, found by symmetry-preserving calculations. *arXiv preprint arXiv:0803.2506*, 2008.

[LRP07] V. Levandovskyy, C. Rosenkranz, and T. Povalyayeva. Online database *gröbner bases implementations. functionality check and comparison*, 2007. URL `http://www.risc.uni-linz.ac.at/Groebner-Bases-Implementations/`.

[Grä09] Hans-Gert Gräbe. The SymbolicData project. Technical report, Technical report (2000-2009), 2009. URL `http://www.symbolicdata.org`.

[GNJ13] Hans-Gert Gräbe, Andreas Nareike, and Simon Johanning. The SymbolicData project–towards a computer algebra social network. 2013.

[HLN13] Albert Heinle, Viktor Levandovskyy, and Andreas Nareike. SymbolicData: SDEval - benchmarking for everyone. *arXiv preprint arXiv:1310.5551*, 2013.

[CCH$^+$11] Svetlana Cojocaru, Alexandru Colesnicov, Albert Heinle, Viktor Levandovskyy, Ludmila Malahov, Grischa Studzinski, and Victor Ufnarovski. Creation of a knowledge framework for non-commutative computer algebra. In *Proc. 7th International Conference on Microelectronics and Computer Science, Chişinău, Republic of Moldova*, pages 166–169, 2011.

[HL13] Albert Heinle and Viktor Levandovskyy. Factorization of $\mathbb{Z}$-homogeneous polynomials in the first (q)-Weyl algebra. *arXiv preprint arXiv:1302.5674*, 2013.

[GHL14] Mark Giesbrecht, Albert Heinle, and Viktor Levandovskyy. Factoring linear differential operators in n variables. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, pages 194–201. ACM, 2014.

[AGK96] Beatrice Amrhein, Oliver Gloor, and Wolfgang Küchlin. Walking faster. In *Design and Implementation of Symbolic Computation Systems*, pages 150–161. Springer, 1996.

[SST12] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Introducing StarExec: a cross-community infrastructure for logic solving. *Comparative Empirical Evaluation of Reasoning Systems*, page 2, 2012.

[BR08] Mohamed Barakat and Daniel Robertz. homalg — a meta-package for homological algebra. *Journal of Algebra and its Applications*, 7(03):299–317, 2008.

[GAP13] GAP. *GAP – Groups, Algorithms, and Programming, Version 4.6.3.* The GAP Group, 2013. URL `http://www.gap-system.org`.

[S$^+$08] William Stein et al. Sage: Open source mathematical software, 2008. URL `http://www.sagemath.org`.

[SEW12] Satya Swarup Samal, Hassan Errami, and Andreas Weber. PoCaB: a software infrastructure to explore algebraic methods for bio-chemical reaction networks. In *Computer Algebra in Scientific Computing*, pages 294–307. Springer, 2012.

[KG00] Minoru Kanehisa and Susumu Goto. KEGG: Kyoto encyclopedia of genes and genomes. *Nucleic acids research*, 28(1):27–30, 2000.

[LNBB$^+$06] Nicolas Le Novere, Benjamin Bornstein, Alexander Broicher, Melanie Courtot, Marco Donizelli, Harish Dharuri, Lu Li, Herbert Sauro, Maria Schilstra, Bruce Shapiro, et al. BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic acids research*, 34(suppl 1):D689–D691, 2006.