

Efficient Data Structures for Representation of Polynomial Optimization Problems: Implementation in SOSTOOLS

Declan Jagt¹, Sachin Shivakumar, Peter Seiler², *Fellow, IEEE*,
and Matthew Peet¹, *Senior Member, IEEE*

Abstract—We present a new data structure for representation of polynomial variables in the parsing of sum-of-squares (SOS) programs. In SOS programs, the variables $s(x; P)$ are polynomial in the independent variables x , but linear in the decision variables P . Current SOS parsers, however, fail to exploit the semi-linear structure of the polynomial variables, treating the decision variables as independent variables in their representation. This results in unnecessary overhead in storage and manipulation of the polynomial variables. To reduce this computational overhead, we introduce a new representation of polynomial variables, the *dpvar* structure, which allows the parser to exploit the structure of the decision variables. We show that use of the *dpvar* structure significantly reduces the computational complexity of the polynomial operations required for parsing SOS programs. We further show that the memory complexity required to store polynomial variables is significantly reduced when using the *dpvar* structure, particularly when combined with the MATLAB Compressed Sparse Column (CSC) matrix representation. Finally, we incorporate the *dpvar* structure into SOSTOOLS 4.00, and test performance for several polynomial optimization problems.

Index Terms—Computational methods, large-scale systems, LMIs, stability of nonlinear systems.

I. INTRODUCTION

MANY problems in analysis and control of nonlinear systems can be formulated as polynomial optimization problems. Since testing nonnegativity of polynomials is NP-hard [1], polynomial constraints of the form $s(x) \geq 0$ for all $x \in \mathbb{R}^n$ are often tightened to sum-of-squares (SOS) constraints: $s \in \Sigma_s$, where Σ_s denotes the set of functions that may be expanded as $s(x) = \sum_i p_i(x)^2$ for some polynomials

$p_i \in \mathbb{R}[x]$. Feasibility of $s \in \Sigma_s$ in turn is equivalent to existence of a positive semidefinite matrix $Q \geq 0$ and a vector of monomials Z_d such that $s(x) = Z_d(x)^T Q Z_d(x)$ – allowing SOS constraints to be expressed as LMIs. In this manner, SOS programs (SOSPs) can be formulated as semidefinite programs (SDPs), which may be solved in polynomial time [2].

The typical process of numerically solving SOSPs consists of two stages: the *parsing* of the SOSP, i.e., the implementation of the program and conversion to an SDP; and the actual *solving* of this SDP. Unfortunately, the computational complexity associated with both of these stages increases rapidly with the size of the SOSP, as a result of which many large-scale applications of SOS programming remain unsolvable. This failure to tackle large-scale problems has prompted several variations on SOS programming to be proposed, reducing complexity of the problem by imposing more restrictive constraints on the positive semidefinite matrix Q [3]–[5]. However, the goal of these modifications is primarily to reduce the computational complexity of the solving stage of the SOS programming process, offering little to no reduction in the cost of parsing the SOSP. As such, even if larger-scale problems can be solved with these modifications, the computational cost of parsing such programs may still make numerical implementation impossible. In fact, in many cases, the computational cost of parsing the SOSP far exceeds that associated to solving the resulting SDP (see Fig. 1), a discrepancy that will only be exacerbated by reducing the complexity of the SDP.

For the greatest lower bound problem and robust stability test presented in Section V-A and V-B, Fig. 1 shows what percentage of the time required to solve each problem is spent on parsing the SOSP. Results are shown using the well-established SOS parsers SOSTOOLS 3.04 [6] and YALMIP [7] to parse the problems, using SEDUMI [8] to solve the resulting SDP. The results show that both parsers consistently require more time to construct the SDP from the SOSP than it takes to actually solve this SDP, frequently spending more than 90% of the execution time on parsing. In this letter, we show that the percentage of the time spent on parsing can be significantly reduced, proposing a new representation of polynomial variables that allows for more efficient parsing of SOSPs.

In converting an SOSP to an SDP, SOS parsers use finite monomial bases Z_d to represent the polynomial variables. Here, we let $Z_d \in \mathbb{R}^{n_1}[x]$ denote a vector containing all monomials in variables x_1, \dots, x_p of degree at most d , where $n_1 := \frac{(p+d)!}{p!d!}$. These monomials may be numerically represented as a matrix $Z_{M,d} \in \mathbb{N}^{n_1 \times p}$ containing the degrees of each variable

Manuscript received 19 March 2022; revised 17 May 2022; accepted 5 June 2022. Date of publication 16 June 2022; date of current version 5 July 2022. The work was supported by the National Science Foundation under Grant CMMI-1931270 and Grant CMMI-1935453. Recommended by Senior Editor L. Menini. (Corresponding author: Declan Jagt.)

Declan Jagt, Sachin Shivakumar, and Matthew Peet are with the School for Engineering of Matter, Transport and Energy (SEMTE), Arizona State University, Tempe, AZ 85287 USA (e-mail: djagt@asu.edu).

Peter Seiler is with the Department of Electrical Engineering and Computer Science, University of Michigan at Ann Arbor, Ann Arbor, MI 48109 USA.

Digital Object Identifier 10.1109/LCSYS.2022.3183650

2475-1456 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See <https://www.ieee.org/publications/rights/index.html> for more information.

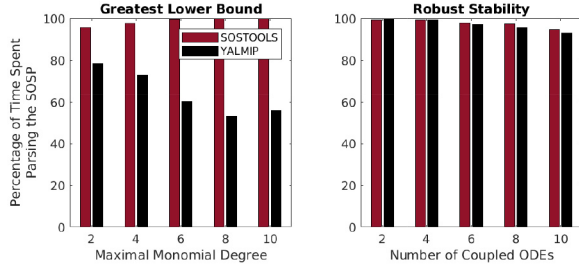


Fig. 1. Percentage of execution time spent parsing the greatest lower bound problem from Section V-A (Eqn. (6)) and the robust stability problem from Section V-B (Eqn. (7)), using SOSTOOLS 3.04 and YALMIP. Using either parser, less than 50% of the time spent on each problem is actually spent on solving the associated SDP, with the parsing of the robust stability program even taking up more than 90% of the time.

in each monomial, so that e.g.,

$$Z_2(x_1, x_2) = \begin{bmatrix} 1 \\ x_2 \\ x_2^2 \\ x_1 \\ x_1 x_2 \\ x_1^2 \\ x_1^3 \end{bmatrix} \quad \text{and} \quad Z_{M,2} = \begin{bmatrix} x_1, x_2 \\ 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 2 & 0 \end{bmatrix}.$$

Using such a monomial basis, an SOS variable $s \in \Sigma_s$ of degree at most $2d$ can be represented in the quadratic form

$$s(x; Q) = Z_d(x)^T Q Z_d(x),$$

where now $Q \in \mathbb{S}^{n_1 \times n_1}$ is a *decision variable*. Meanwhile, any polynomial $p \in \mathbb{R}[x]$ of degree $2d$ is uniquely defined by a vector of coefficients $c \in \mathbb{R}^{n_2}$ for $n_2 := \frac{(p+2d)!}{p!(2d)!}$, and may be represented in the linear *pvar* form as

$$p(x) = c^T Z_{2d}(x). \quad (1)$$

Finally, interface with SDP solvers requires polynomial constraints $g(x; \xi) = 0$, parameterized by decision variables ξ , to be expressed in the SDP format

$$0 = g(x; \xi) = (A\xi - b)^T Z(x), \quad \text{imposing} \quad A\xi = b.$$

For example, letting $s_1(x_1; \xi) = \begin{bmatrix} 1 \\ x_1 \end{bmatrix}^T \begin{bmatrix} \xi_1 \xi_2 \\ \xi_2 \xi_3 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \end{bmatrix}$ for $\begin{bmatrix} \xi_1 \xi_2 \\ \xi_2 \xi_3 \end{bmatrix} \geq 0$, and defining $p_1(x_1) := 1 - 2x_1^2$, the constraint

$$0 = g_1(x_1; \xi) := s_1(x_1; \xi)p_1(x_1) - 1 + 4x_1^4,$$

can be equivalently represented in the SDP format as

$$0 = g_1(x_1; \xi) = \left(\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ -2 & 0 & 1 \\ 0 & -4 & 0 \\ 0 & 0 & -2 \end{bmatrix}}_A \underbrace{\begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix}}_\xi - \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ -4 \end{bmatrix}}_b \right)^T \begin{bmatrix} x_1 \\ x_1^2 \\ x_1^3 \\ x_1^4 \end{bmatrix}.$$

In order to derive this expression, however, an SOS parser has to compute the product $s_1(x; \xi)p_1(x)$ without knowing the values of the decision variables ξ . To this end, current parsers treat the decision variables as independent variables, and represent SOS variables s in the linear form as

$$s(x; \xi) = c^T \bar{Z}_{2d}(x; \xi)$$

where $\bar{Z}_{2d}(x; \xi) := \begin{bmatrix} 1 \\ \xi \end{bmatrix} \otimes Z_{2d}(x)$. However, including the decision variables in the monomial basis \bar{Z}_{2d} , the computational cost of operations like multiplication increases rapidly with the number of decision variables. Moreover, substantial computational effort may be necessary to convert constraints $0 = c^T \bar{Z}(x; \xi)$ to the SDP format $0 = (A\xi - b)^T Z(x)$.

To reduce the computational complexity associated with parsing SOSPs, we propose a new representation of polynomial decision variables, representing a variable $s \in \mathbb{R}[x; \xi]$ as

$$s(x; \xi) := Z_1(\xi)^T C Z_d(x) = \begin{bmatrix} 1 \\ \xi \end{bmatrix}^T C Z_d(x), \quad (2)$$

so that, for example

$$s_1(x_1; \xi) = \begin{bmatrix} 1 \\ x_1 \end{bmatrix}^T \begin{bmatrix} \xi_1 & \xi_2 \\ \xi_2 & \xi_3 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \end{bmatrix}.$$

We refer to this variable structure as the *dpvar* representation – a generalization of the *pvar* representation for polynomials (Eqn. (1)) to polynomials with decision variables. As will be shown in Section II, use of this representation accounts for linearity with respect to the decision variables and eliminates polynomial manipulations involving decision variables. Furthermore, translation of a *dpvar* constraint $s(x; \xi) = 0$ to an SDP constraint $(C_2\xi + c_1)^T Z(x) = 0$ is trivial, in that

$$s(x; \xi) = \begin{bmatrix} 1 \\ \xi \end{bmatrix}^T \begin{bmatrix} c_1^T \\ C_2^T \end{bmatrix} Z_d(x) = (\xi^T C_2^T + c_1^T) Z_d(x).$$

Finally, by exploiting the MATLAB sparse matrix representation features, the *dpvar* structure reduces memory requirements and computational effort – see Section IV.

In the remainder of this letter, we carefully detail how an ideal parser should integrate the *dpvar* structure into the parsing of SOSPs. Specifically, an ideal parser should

- 1) Exploit structure in polynomial computations. In particular, for operations such as multiplication, the parser should exploit the affine dependence on the decision variables to reduce computational overhead.
- 2) Be based on analytic expressions for the mathematical operations.
- 3) Allow for fully dense polynomial structures.
- 4) Make efficient use of the platform-specific sparsity structure to minimize memory usage and computational complexity for sparse polynomial objects.
- 5) Be scalable to hundreds of thousands of decision variables.

In the following sections, we show how the *dpvar* structure can be used to achieve these goals in the context of the MATLAB programming language and sparsity package.

II. OPERATIONS IN THE DPVAR REPRESENTATION

We first show that, using the *dpvar* representation, standard operations on polynomial variables $s \in \mathbb{R}[x; \xi]$ may be performed at relatively low computational cost. In particular, we note that in the *dpvar* representation, as presented in Eqn. (2), the vector of linear monomials $Z_1(\xi)$ always takes the same form. Therefore, there is no need to explicitly store or account for the degrees of the monomials in $Z_1(\xi)$, and the complexity of operations will be largely independent of the number of decision variables ξ .

By contrast, in the pvar representation,

$$s(x; \xi) = c^T \bar{Z}_d(x; \xi), \quad \text{with } \bar{Z}_d(x; \xi) := \begin{bmatrix} 1 \\ \xi \end{bmatrix} \otimes Z_d(x), \quad (3)$$

the decision and independent variables are included in a single vector of monomials $\bar{Z}_d(x; \xi)$. Implementing a data structure based on the pvar representation, therefore, the degrees of the decision variables ξ have to be explicitly stored and processed when performing polynomial operations, introducing unnecessary computational overhead.

In the following subsections, we show how multiplication, differentiation, and substitution of polynomial variables may be performed efficiently using the dpvar representation. The computational complexity of each operation is indicated using big O notation, writing $\mathcal{O}(g)$ to indicate a complexity no greater than Cg for some constant $C > 0$. In each case, the complexity is also illustrated through a scalability test, comparing the time required to perform each operation using the dpvar data structure from SOSTOOLS 4.00, the pvar and syms structures from SOSTOOLS 3.04, as well as the YALMIP sdmpvar structure. For the syms tests, the presented computation times include those necessary to construct the monomial degrees and coefficients needed for further processing in SOSTOOLS 3.04. All tests were performed on a computer with Intel Core i7-5960X CPU, and 128 GB of installed RAM. Examples and additional test results can be found in the arXiv version [9].

A. Multiplication

We first consider the operation of multiplication. Since decision variables must appear linearly in any SOS program, polynomial decision variables $s \in \mathbb{R}[x; \xi]$ may only be multiplied by known polynomial functions $p \in \mathbb{R}[y]$. In dpvar format, these may be expressed as

$$s_1(x; \xi) = Z_1(\xi)^T C Z_{d_1}(x), \quad p_2(y) = b^T Z_{d_2}(y),$$

so that the product becomes

$$s_1(x; \xi) p_2(y) = Z_1(\xi)^T (b^T \otimes C) (Z_{d_2}(y) \otimes Z_{d_1}(x)).$$

Performing this operation in MATLAB, the coefficients b, C and degrees Z_{M,d_1}, Z_{M,d_2} can be stored as sparse matrices. Then, taking the Kronecker product $b^T \otimes C$ will require multiplying at most $\text{nnz}(C) \cdot \text{nnz}(b)$ elements, where $\text{nnz}(A)$ denotes the number of nonzero elements of a matrix A , invoking a worst-case complexity of

$$\mathcal{O}(\text{nnz}(C) \text{nnz}(b)).$$

To compute the product $Z_{d_2}(y) \otimes Z_{d_1}(x)$, the nonzero degrees of all the variables in each monomial in Z_{d_2} must be added to the degrees of the same variables in each of the monomials in Z_{d_1} . In the worst-case scenario (e.g., $x = y$ and $Z_{d_1} = Z_{d_2}$), this will require adding all nonzero degrees in Z_{M,d_2} to all nonzero degrees in Z_{M,d_1} amounting to a complexity of

$$\mathcal{O}(\text{nnz}(Z_{M,d_1}) \text{nnz}(Z_{M,d_2})).$$

Consider now computing the same product based on the pvar representation, $s_1(x; \xi) = c^T \bar{Z}_{d_1}(x; \xi)$, so that

$$s_1(x; \xi) p_2(y) = (b^T \otimes c^T) (Z_{d_2}(y) \otimes \bar{Z}_{d_1}(x; \xi)),$$

where $\bar{Z}_d(x; \xi)$ is as in (3). As was the case in the dpvar format, the cost of computing the new coefficients will be

$$\mathcal{O}(\text{nnz}(c) \text{nnz}(b)) = \mathcal{O}(\text{nnz}(C) \text{nnz}(b)),$$

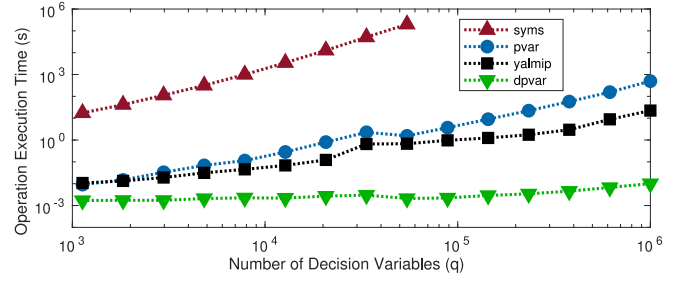


Fig. 2. Computation time for polynomial multiplication $s(x, \xi)p(y)$ using the syms, pvar, and dpvar data structures from respectively SOSTOOLS 3.04 and 4.00, and the sdmpvar structure from YALMIP to represent $s(x, \xi)$. The rate at which the computation time increases is relatively small using the dpvar representation compared to the alternatives, with the time increasing only slightly as the number of decision variables grows to 10^6 .

scaling with the product of the number of terms in the two polynomials. However, in the pvar representation, the number of nonzero degrees in \bar{Z}_{M,d_1} increases linearly with the number of decision variables q in s_1 , so that the complexity of computing $Z_{d_2} \otimes \bar{Z}_{d_1}$ will be

$$\mathcal{O}(\text{nnz}(\bar{Z}_{M,d_1}) \text{nnz}(Z_{M,d_2})) = \mathcal{O}(q \cdot \text{nnz}(Z_{M,d_1}) \text{nnz}(Z_{M,d_2})).$$

This dependence on the number of decision variables is not present when implementing the dpvar representation, resulting in a substantial difference in computational complexity for large values of q . This reduction in complexity can be observed in Fig. 2, displaying the required time for multiplying a variable $s_1(x_1, x_2; \xi_1, \dots, \xi_q)$ with a polynomial $p_2(y_1, y_2)$ using the different data structures. The presented computation time for each structure and each value of q corresponds to the average of several test results, using randomly generated coefficients to construct s_1 and p_2 . Both polynomials were of degree $d_1 = d_2 = 4$ in each case.

B. Differentiation and Substitution

We now consider the operations of differentiation and substitution. For an arbitrary polynomial $s \in \mathbb{R}[x; \xi]$ in the dpvar representation, these operations will involve only adjusting the monomial degrees $Z_{M,d}$, and associated columns in the coefficient matrix C . For example, let $z_{ij} = [Z_{M,d}]_{ij}$ denote the element in row i and column j of the degree matrix $Z_{M,d} \in \mathbb{N}^{n \times p}$, and let C_i denote the i th column of the coefficient matrix $C \in \mathbb{R}^{(q+1) \times n}$. Then, differentiation with respect to x_j may be performed by multiplying all elements in each column C_i for $i = 1, \dots, n$ with z_{ij} , and subtracting a value of 1 from all nonzero degrees in column j of $Z_{M,d} \in \mathbb{N}^{n \times p}$. The complexity of this operation depends only indirectly on the number of decision variables, as each decision variable adds a row to the coefficient matrix C .

By contrast, performing the same operations using the pvar representation, the decision variables are included in the monomial basis \bar{Z}_d . Therefore, the complexity of finding and adjusting the appropriate degrees of the monomials to account for, e.g., differentiation with respect to a variable x_j will directly increase with the number of decision variables, despite the fact that the decision variables themselves are invariant under these operations. In this sense, unnecessary computational overhead is introduced when performing operations such as differentiation in the pvar representation, which is avoided using the dpvar representation. The reduced

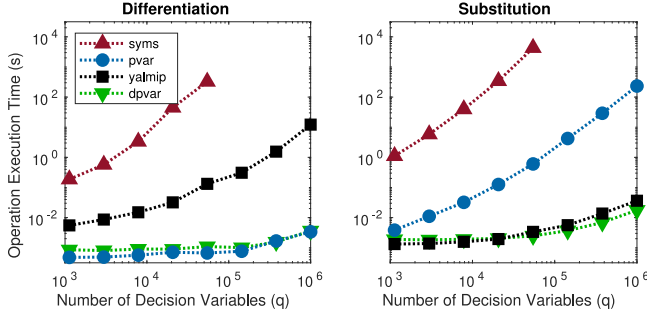


Fig. 3. Computation time for differentiation and substitution of polynomial variables $s(x; \xi)$ using different data structures to represent s . Using the `dpvar` structure, the required time for both operations increases very slowly, invoking the lowest computational complexity overall.

computation time this allows is illustrated in Fig. 3, presenting the elapsed time for differentiation and substitution of a polynomial $s_1(x_1, x_2; \xi_1, \dots, \xi_q)$ with respect to the variable x_2 , using the different SOSTOOLS and YALMIP data structures, and for increasing values of q .

III. STORAGE AND MANIPULATION OF DPVARS

Having analyzed the complexity of standard operations in the `dpvar` representation, in this section, we show how this representation also allows polynomial variables to be efficiently stored and manipulated. In particular, implementing the `dpvar` representation in MATLAB, we define a variable $S \in \mathbb{R}^{m_1 \times m_2}[x; \xi]$ using the `dpvar` data structure, storing

- The independent variables x_1, \dots, x_p .
- The decision variables ξ_1, \dots, ξ_q .
- The monomial degrees $Z_{M,d} \in \mathbb{N}^{n \times p}$.
- The coefficient matrix $C \in \mathbb{R}^{m_1(q+1) \times m_2 n}$.

Decomposing the polynomial in this manner, the greatest storage cost will be associated with the monomial degrees and coefficient matrix. However, storing both of these fields as sparse matrices in MATLAB, the memory overhead will be minimal, as we show in Section III-A. In addition, exploiting the structure of `dpvar` objects, matrix operations such as concatenation can be performed with relatively low computational overhead, as detailed in Section III-B.

A. Memory Complexity of Storing `dpvar` Objects

Exploiting linearity of the decision variables in its structure, the `dpvar` representation allows polynomial variables to be stored in programming languages with sparsity structures using minimal memory with respect to the number of decision variables. Specifically, consider storing a matrix-valued polynomial variable $S \in \mathbb{R}^{m_1 \times m_2}[x_1, \dots, x_p; \xi_1, \dots, \xi_q]$, expressed in the `dpvar` representation as

$$S(x; \xi) = (I_{m_1} \otimes Z_1(\xi))^T C (I_{m_2} \otimes Z_d(x)). \quad (4)$$

As mentioned, the greatest storage cost in representing this variable in MATLAB will be that associated to the coefficients $C \in \mathbb{R}^{m_1(q+1) \times m_2 n_1}$ and degrees $Z_{M,d} \in \mathbb{N}^{n_1 \times p}$. Storing both objects as sparse matrices, only the nonzero values are retained, invoking a memory complexity of

$$\mathcal{O}(nnz(C) + nnz(Z_{M,d})),$$

which does not depend directly on the value of q .

Consider now storing the same variable in the `pvar` format,

$$S(x; \xi) := B^T (I_{m_2} \otimes \bar{Z}_d(x; \xi)), \quad (5)$$

where $B \in \mathbb{R}^{m_1 \times m_2 n_2}$ and $\bar{Z}_d = \begin{bmatrix} 1 \\ \xi \end{bmatrix} \otimes Z_d(x) \in \mathbb{R}^{n_2}[x; \xi]$.

In this representation, the required memory for storing the variable will be $\mathcal{O}(nnz(B) + nnz(\bar{Z}_{M,d}))$. However, although the number of coefficients $nnz(B) = nnz(C)$ is the same, the number of degrees in $\bar{Z}_{M,d} \in \mathbb{N}^{(q+1)n_1 \times (p+q)}$ now depends on the number of decision variables q (see [9] for more details), so that the total memory complexity becomes

$$\mathcal{O}(nnz(C) + (q+1)nnz(Z_{M,d}) + qn_1).$$

Implementing the `pvar` representation, the required memory of storing the monomials increases directly with the number of decision variables, potentially amounting to a substantial storage cost that may be avoided using the `dpvar` structure.

B. Matrix Operations on `dpvar` Objects

In many SOS programs, the polynomial decision variables appear as matrix-valued objects, making it crucial for matrix operations such as concatenation to be efficiently implemented in any SOS parser. Using the `dpvar` representation, this can be achieved by exploiting the block structure of the coefficient matrix. In particular, for a variable $S \in \mathbb{R}^{m_1 \times m_2}[x; \xi]$, the coefficient matrix $C \in \mathbb{R}^{m_1(q+1) \times m_2 n}$ is comprised of $m_1 \times m_2$ blocks $C_{ij} \in \mathbb{R}^{(q+1) \times n}$, each corresponding to a single element of the matrix-valued variable. This allows for efficient modification of individual elements of the polynomial variable. In addition, for two matrix-valued polynomial variables $S_1, S_2 \in \mathbb{R}^{m_1 \times m_2}[x; \xi]$, defined in terms of the same monomial basis Z_d as

$$S_i(x; \xi) = (I_{m_1} \otimes Z_1(\xi))^T C_i (I_{m_2} \otimes Z_d(x)),$$

concatenation of S_1 and S_2 merely requires concatenating the coefficient matrices C_1 and C_2 . For example, vertical concatenation of S_1, S_2 may be represented as

$$\begin{bmatrix} S_1(x; \xi) \\ S_2(x; \xi) \end{bmatrix} = (I_{2m_1} \otimes Z_1(\xi))^T \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} (I_{m_2} \otimes Z_d(x)),$$

requiring almost no computational effort.

IV. EXPLOITING SPARSITY IN STORAGE AND OPERATION

Having presented the benefits of using the `dpvar` representation in parsing SOS programs, we finally show how the `dpvar` data structure exploits the MATLAB built-in sparsity structure to minimize memory and computational overhead in representing polynomial variables. In particular, in Section IV-A, we outline how sparse matrices are implemented in MATLAB and analyze how this format affects memory and computational complexity. In Section IV-B, we then show how the `dpvar` structure exploits this format in storing the coefficients and monomials, to optimize performance.

A. The Compressed Sparse Column Format

In MATLAB, sparse matrices are implemented using a Compressed Sparse Column (CSC) format [10], representing a matrix $A \in \mathbb{R}^{m \times n}$ through three arrays:

- 1) An array $a \in \mathbb{R}^{nnz(A)}$ of nonzero elements.
- 2) An array $r \in \mathbb{R}^{nnz(A)}$ of row indices.
- 3) An array $cp \in \mathbb{R}^{n+1}$ of column pointers.

In the first of these arrays, $\mathbf{a} \in \mathbb{R}^{nnz(A)}$, all nonzero elements of the matrix are collected in *column-major* order. That is, letting $\{a_1, \dots, a_n\}$ denote the columns of the matrix A , and letting $\{\bar{a}_1, \dots, \bar{a}_n\}$ denote the nonzero elements from each of these columns, the first array \mathbf{a} may be constructed as

$$\mathbf{a} = [\bar{a}_1^T, \dots, \bar{a}_n^T]^T \in \mathbb{R}^{nnz(A)}.$$

Corresponding row numbers for these nonzero elements are then stored in the array \mathbf{r} , so that the k th nonzero element $\mathbf{a}(k)$ appears in row $\mathbf{r}(k)$ of the matrix A . Finally, for each of the columns $j = 1, \dots, n$ of the matrix, a column pointer is stored in the array \mathbf{cp} , defined as

$$\mathbf{cp} = \left[1, 1 + \ell_1, \dots, 1 + \sum_{j=1}^{n-1} \ell_j, \sum_{j=1}^n \ell_j\right] \in \mathbb{R}^{n+1},$$

where $\ell_j := nnz(a_j)$. Then, $\mathbf{a}(\mathbf{cp}(j))$ provides the first nonzero element of column $j \in \{1, \dots, n\}$ of $A \in \mathbb{R}^{m \times n}$.

Using this data structure to store (sparse) matrices, the required memory will be minimal for matrices with few columns. In particular, although the cost of storing $\mathbf{a} \in \mathbb{R}^{nnz(A)}$ and $\mathbf{r} \in \mathbb{R}^{nnz(A)}$ depends only on the number of nonzero elements $nnz(A)$, the memory necessary to store the array $\mathbf{cp} \in \mathbb{R}^{n+1}$ is determined by the number of columns n of the matrix. Therefore, the memory burden for storing sparse matrices increases with the number of columns in this matrix, even if these columns contain no nonzero elements.

In addition, using the CSC storage format, the complexity of operations involving full or partial columns of the matrix will generally be smaller than those involving full or partial rows of the matrix. Indeed, for any column $j \in \{1, \dots, n\}$ of A , the nonzero elements appearing in this column are known to be stored at positions $k \in \{\mathbf{cp}(j), \mathbf{cp}(j) + 1, \dots, \mathbf{cp}(j + 1) - 1\}$ within the array \mathbf{a} , requiring minimal effort to access these elements. On the other hand, in order to access elements of a particular row $i \in \{1, \dots, m\}$ of the matrix, all indices $k \in \{1, \dots, nnz(A)\}$ with associated row index $\mathbf{r}(k) = i$ have to be found, potentially requiring the full array \mathbf{r} to be analyzed. This introduces additional computational overhead when operating on full or partial rows of the matrix, generally making “row-based” operations more computationally demanding than “column-based” equivalents.

B. Sparsity in the *dpvar* Structure

We now show how, using the *dpvar* data structure, the CSC storage format may be exploited to minimize the storage and operational cost of representing and manipulating polynomial variables. To illustrate, consider storing a variable

$$s(x; \xi) = Z_1(\xi)CZ_d(x) \in \mathbb{R}[x_1, \dots, x_p; \xi_1, \dots, \xi_q].$$

Storing the coefficient matrix $C \in \mathbb{R}^{(q+1) \times n}$ and monomial degrees $Z_{M,d} \in \mathbb{N}^{n \times p}$ using the CSC structure, the required memory will be relatively small. In particular, since p variables allow $n = \frac{(p+d)!}{p!d!}$ monomials of degree at most d , the number of rows of the degree matrix $Z_{M,d} \in \mathbb{N}^{n \times p}$ will in general vastly exceed the number of columns. In addition, in SOS programs, a monomial $[Z_d]_k$ is often paired with multiple decision variables ξ_j . As a consequence, the number of decision variables tends to exceed the number of monomials, and thus the number of rows in the coefficient matrix $C \in \mathbb{R}^{(q+1) \times n}$ also tends to be at least as large as the number of columns. Since the memory cost of storing a matrix in the CSC format increases with the number of columns, the fact that both the coefficient and monomial degree matrices contain relatively

few columns allows polynomial variables to be efficiently stored using the *dpvar* data structure.

Similarly, the complexity of performing operations on variables in the *dpvar* structure may be minimized using the sparse storage structure. In particular, the greatest computational effort in many operations comes from having to merge or adjust particular monomials in $Z_d \in \mathbb{R}^n[x]$, as well as the associated columns of the coefficient matrix $C \in \mathbb{R}^{(q+1) \times n}$. Here, although the large number of rows in $Z_{M,d} \in \mathbb{N}^{n \times p}$ makes these operations more demanding in the CSC storage format, the small number of columns ensures the complexity remains relatively small. Moreover, the CSC storage structure allows the columns of the matrix $C \in \mathbb{R}^{(q+1) \times n}$ to be adjusted with relatively high efficiency, invoking a complexity that does not depend directly on the number of rows $q+1$. Thus, exploiting the MATLAB built-in sparse storage structure, the *dpvar* data structure allows the computational cost of standard operations to be minimized with respect to the number of decision variables q .

V. INCORPORATION INTO SOSTOOLS

Having demonstrated the advantages of using the *dpvar* data structure for parsing polynomial variables, we now consider the incorporation of this structure in SOSTOOLS. Specifically, for SOSTOOLS 4.00 [11], we have modified all functions to use the *dpvar* data structure for representation of polynomial variables. To illustrate the enhanced performance this offers, in this section, we consider several optimization problems that are commonly solved with SOSTOOLS. For each problem, we compare the time required for parsing the problem using SOSTOOLS 3.04 and 4.00, as well as using the batch parser YALMIP [7], in each case using SEDUMI [8] to solve the resulting SDP. For more details on the implementation of each problem, we refer to the arXiv version [9].

A. Greatest Lower Bound

As a first problem, we seek a greatest lower bound (GLB) on a function $f(x) = x_1^4 + x_2^4 - 2x_2x_1^3 - 3x_2^2x_1^2 + 150(x_1^2 + x_2^2)$,

$$\max_{\gamma} \quad \gamma, \quad \text{s.t.} \quad \gamma \leq f(x) \quad \forall x_1, x_2 \in [-12, 12].$$

Invoking Putinar’s Positivstellensatz [12] (Psatz), we implement this problem using a single SOS constraint

$$(f - \gamma) - s_1g_1 - s_2g_2 - s_3[g_1 + g_2] \in \Sigma_s, \quad (6)$$

where $s_1, s_2, s_3 \in \Sigma_s$ are SOS variables, and we define

$$g_1(x) = 12^2 - x_1^2, \quad g_2(x) = 12^2 - x_2^2.$$

Figure 4(a) displays the time required to parse the GLB problem using the different data structures, and for increasing monomial degrees d in the variables $s_i = Z_d(x)^T P Z_d(x)$. The results show that, using the *dpvar* structure, SOSTOOLS 4.00 is able to parse the GLB problem in substantially less time than other parsers, in general spending less than 20% of the computation time on parsing the SOS.

B. Robust Stability

As a second example, we consider testing robust stability of a linear ODE

$$\dot{x}(t) = A(p)x(t), \quad \forall p \in G := \{p \in \mathbb{R}^2 \mid g(p) \geq 0\},$$

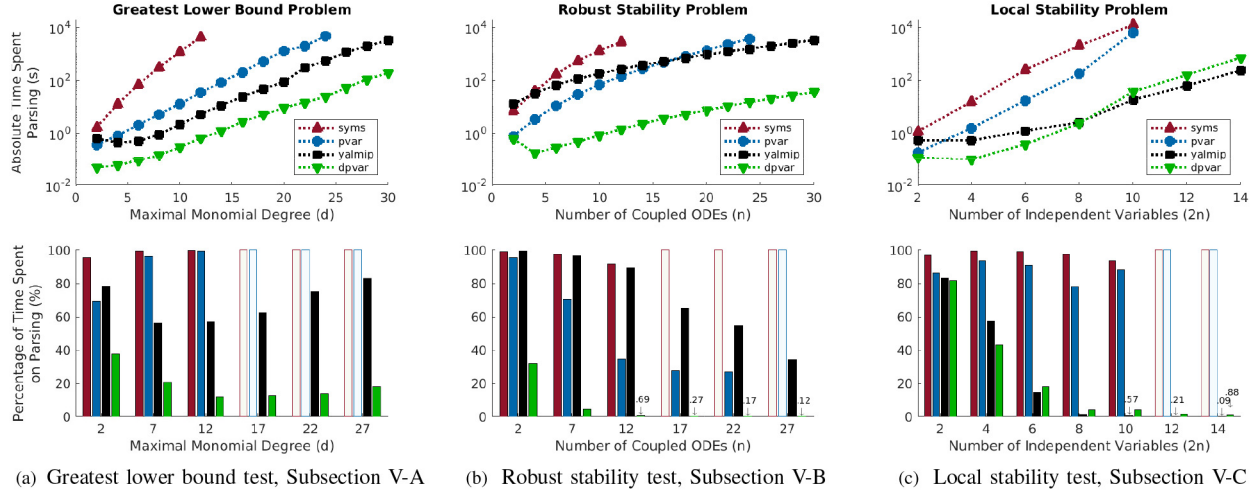


Fig. 4. Required time to parse the SOS optimization problems from Section V, using SOSTOOLS 3.04 with the `syms` and `pvar` data structures, SOSTOOLS 4.00 with the `dpvar` data structure, and using the batch parser YALMIP. Tests for the local stability problem were discontinued at $n = 14$, as the solver ran out of memory. The percentage of time spent on parsing each problem was computed by dividing the absolute time spent parsing by the sum of the time spent parsing the SOSP and solving the resulting SDP. The results show that, using the `dpvar` data structure, SOSTOOLS 4.00 is able to parse common SOS problems with an efficiency comparable to, or even greater than that of the batch parser YALMIP.

where $g(p) = 1 - p_1^2 - p_2^2$, and $A \in \mathbb{R}^{n \times n}$ is defined such that $A_{ij} = 0.25p_1$ for $i > j$, $A_{ij} = -0.25p_2$ for $i < j$, and $A_{ii} = 1$. We implement this as an SOS problem

$$P - \epsilon I_n \in \Sigma_s[p], \quad -Qg - PA - A^T P \in \Sigma_s[p], \quad (7)$$

where $Q \in \Sigma_s[p]$, and we let $\epsilon = 10^{-4}$.

The time required for parsing this problem using each data structure is displayed in Fig. 4(b), using a variable P of maximal degree $2d = 4$, and problem sizes up to $n = 30$. The results show that this problem too can be parsed in significantly less time using the `dpvar` structure than using the alternative implementations. In fact, for $n = 50$, the problem could still be parsed in 374 seconds using SOSTOOLS 4.00, a threshold exceeded by YALMIP at $n = 13$.

C. Local Stability

As a final example, we test local stability of a chain of n Van der Pol oscillators. In particular, we consider the system presented in [13], given by $\dot{x}(t) = f(x)$, where $x = (y, z) = (y_1, \dots, y_n, z_1, \dots, z_n)$ and for any $j \in \{1, \dots, n-1\}$,

$$\begin{aligned} f_j(y, z) &= 0.8y_j + 10(1.2^2 y_j^2 - 0.21)z_j + \epsilon_j z_{j+1} y_j, \\ f_n(y, z) &= 0.8y_n + 10(1.2^2 y_n^2 - 0.21)z_n, \\ f_{n+i}(y, z) &= -2z_i, \quad \forall i \in \{1, \dots, n\} \end{aligned}$$

where we let $\epsilon_j = -0.5$ for each j . We test stability inside a ball of radius $r = 0.5$, so that $x \in \{x \in \mathbb{R}^{2n} \mid g(x) \geq 0\}$, where $g(x) = r^2 - \|x\|^2$, implementing SOS constraints

$$V \in \Sigma_s[x], \quad -[\nabla V(x)]^T f(x) - s(x)g(x) \in \Sigma_s[x], \quad (8)$$

where $s \in \Sigma_s$. The times required to parse this problem using a function V of degree $2d = 4$ are presented in Fig. 4(c), showing that, for this problem, SOSTOOLS 4.00 achieves an efficiency similar to that of the batch parser YALMIP.

VI. CONCLUSION

In this letter, we have introduced a new representation of polynomial variables, which is affine in the decision variables.

We showed that, using this `dpvar` representation, computation time for polynomial operations remains relatively small, increasing favorably with the number of involved decision variables. Exploiting the MATLAB built-in sparsity structure, we also showed that the computational and memory overhead for storing and manipulating variables in the `dpvar` representation is minimal. Incorporating this representation in SOSTOOLS 4.00, performance was drastically enhanced, with significant speedup over all existing parsers achieved for most common SOS problems.

REFERENCES

- [1] L. Blum *et al.*, *Complexity and Real Computation*. New York, NY, USA: Springer, 1998.
- [2] S. Boyd *et al.*, *Linear Matrix Inequalities in System and Control Theory*. Philadelphia, PA, USA: SIAM, 1994.
- [3] A. A. Ahmadi and A. Majumdar, "DSOS and SDSOS optimization: LP and SOCP-based alternatives to sum of squares optimization," in *Proc. 48th Annu. Conf. Inf. Sci. Syst. (CISS)*, 2014, pp. 1–5.
- [4] H. Waki *et al.*, "Sums of squares and semidefinite program relaxations for polynomial optimization problems with structured sparsity," *SIAM J. Optim.*, vol. 17, no. 1, pp. 218–242, 2006.
- [5] Y. Zheng *et al.*, "Sparse sum-of-squares (SOS) optimization: A bridge between DSOS/SDSOS and SOS optimization for sparse polynomials," in *Proc. Amer. Control Conf. (ACC)*, 2019, pp. 5513–5518.
- [6] S. Prajna *et al.*, "Introducing SOSTOOLS: A general purpose sum of squares programming solver," in *Proc. 41st IEEE Conf. Decis. Control*, vol. 1, 2002, pp. 741–746.
- [7] J. Löfberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2004, pp. 284–289.
- [8] J. F. Sturm, "Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones," *Optim. Methods Softw.*, vol. 11, nos. 1–4, pp. 625–653, 1999.
- [9] D. Jagt *et al.*, "Efficient data structures for exploiting sparsity and structure in representation of polynomial optimization problems: Implementation in SOSTOOLS," 2022, *arXiv:2203.01910*.
- [10] J. R. Gilbert *et al.*, "Sparse matrices in MATLAB: Design and implementation," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 333–356, 1992.
- [11] A. Papachristodoulou *et al.*, "SOSTOOLS version 4.00 sum of squares optimization toolbox for MATLAB," 2021, *arXiv:1310.4716*.
- [12] M. Putinar, "Positive polynomials on compact semi-algebraic sets," *Indiana Univ. Math. J.*, vol. 42, no. 3, pp. 969–984, 1993.
- [13] M. Tacchi *et al.*, "Approximating regions of attraction of a sparse polynomial differential system," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 3266–3271, 2020.