Computational graphs for matrix functions

Jarlebring, Elias and Fasi, Massimiliano and Ringh, Emil

2021

MIMS EPrint: **2021.12**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Computational graphs for matrix functions*

Elias Jarlebring†   Massimiliano Fasi‡   Emil Ringh§

Abstract

Many numerical methods for evaluating matrix functions can be naturally viewed as computational graphs. Rephrasing these methods as directed acyclic graphs (DAGs) is a particularly effective way to study existing techniques, improve them, and eventually derive new ones. As the accuracy of these matrix techniques is determined by the accuracy of their scalar counterparts, the design of algorithms for matrix functions can be viewed as a scalar-valued optimization problem. The derivatives needed during the optimization can be calculated automatically by exploiting the structure of the DAG, in a fashion akin to backpropagation. The Julia package `GraphMatFun.jl` offers the tools to generate and manipulate computational graphs, to optimize their coefficients, and to generate Julia, MATLAB, and C code to evaluate them efficiently. The software also provides the means to estimate the accuracy of an algorithm and thus obtain numerically reliable methods. For the matrix exponential, for example, using a particular form (degree-optimal) of polynomials produces algorithms that are cheaper, in terms of computational cost, than the Padé-based techniques typically used in mathematical software. The optimized graphs and the corresponding generated code are available online.

Key words. Polynomials of matrices, functions of matrices, computational graphs .

1   Introduction

The scalar function $f : \Omega \subset \mathbb{C} \to \mathbb{C}$ can be extended to square matrices in a customary fashion. Matrix functions play an important role in numerical linear algebra as well as in matrix theory, and results in the literature range from theory and computation to applications [21]. Formally, $f(A)$ can be defined in a number of ways: one can use, for example, the Jordan decomposition, a Taylor series expansion, Cauchy integrals, or Hermite interpolation. For a matrix A whose spectrum lies in a region where f is analytic, these four definitions are equivalent.

Numerical algorithms for evaluating functions of $n \times n$ matrices are based on a variety of different approaches, but for a large class of methods the approximation is formed by relying on three basic operations:

O1. linear combination of matrices,

---

†Department of Mathematics, KTH Royal Institute of Technology, SeRC Swedish e-science research center, Lindstedtsvägen 25, Stockholm SE-100 44, Sweden, eliasj@kth.se.

‡School of Science and Technology, Örebro University, Örebro SE-701 82, Sweden, massimiliano.fasi@oru.se.

§Department of Mathematics, KTH Royal Institute of Technology, SeRC Swedish e-science research center, Lindstedtsvägen 25, Stockholm SE-100 44, Sweden, eringh@kth.se.

O2. matrix multiplication, and

O3. solution of a linear systems with n right-hand sides.

Our work focuses on algorithms constructed by applying O1–O3. This approach does not include methods based on the Schur decomposition, such as the algorithm of Björck and Hammarling for the square root [6, 18] or the Schur–Parlett algorithm for analytic matrix functions [26]. Restricting oneself to O1–O3 yields matrix functions that are particularly easy to work with: loosely speaking, if the expression for $f(z)$ features only the scalar counterparts of O1–O3 and the function is defined on the spectrum of A [21, Definition 1.1], then a formula for $f(A)$ can be obtained by simply replacing all occurrences of z in the formula for $f(z)$ with A [16, Chapter 9].

By interpreting functions constructed in this way as computational graphs where every node is a matrix, we can improve the accuracy and performance of state-of-the-art algorithms for the evaluation of matrix functions. The contribution of this work is twofold.

1. We present a new software package, `GraphMatFun.jl`, designed to work with computational graphs featuring only O1–O3. The code offers tools to create and manipulate such graphs, modify their topology, optimize the matrix functions they represent, and evaluate them at scalar, vector, and matrix arguments. A framework to generate efficient Julia, MATLAB, and C code that implements these functions is also provided. The software is written in the Julia[1] language [4], is released under the terms of the MIT license, and is freely available on GitHub.[2] Pull requests are welcome, and user contribution is facilitated by the availability of a complete set of unit tests and of a continuous integration infrastructure.

2. We provide a set of graphs that were obtained by using our optimization strategy on algorithms available in the literature. In many cases of practical interest, the resulting algorithms are more efficient and accurate than those they originated from, and in fact of any existing alternative. These specific graphs represent a scientific contribution in their own right and are available on GitHub in a separate repository.[3]

The features of the package are presented by means of examples accompanied by extended code snippets.

- The facilities for graph generation and basic node handling are illustrated in Examples 2 and 4.

- The data structure that underpins the graph and the numerical evaluation of the function underlying a graph are illustrated in Examples 1 and 2.

- The graph manipulation and automatic optimization techniques are illustrated in Example 3.

- The degree-optimal form for evaluating polynomial with a reduced number of non-scalar multiplications is described in Section 3 and illustrated in Example 6.

- The features associated with the use of optimization to design graph algorithms are outlined in Section 4 and illustrated in Example 10.

- The generation of code that evaluates efficiently the function underlying a graph is discussed in Section 5.1 and illustrated in Example 13.

- The functions for saving and loading graph objects in a language-independent format are introduced in Section 5.2 and illustrated in Example 15.

---

[1]https://julialang.org/

[2]https://github.com/matrixfunctions/GraphMatFun.jl

[3]https://github.com/matrixfunctions/GraphMatFunData

- The backward error analysis of methods for approximating the matrix exponential is given in Section 6.1 and illustrated in Example 16.

- A simple strategy to compute running error bounds on the round-off error occurring during the evaluation of a graph are given in Section 6.2 and illustrated in Example 17.

While developing our package, we noticed that some of the Julia Base routines for evaluating scalar polynomials and for computing the exponential of a matrix were not always optimal in terms of performance. We addressed these issues in pull requests that have now been merged into the master branch of the Julia GitHub repository.[4]

Section 2 introduces our graph framework and shows how our package can be used to generate and manipulate graphs that represent matrix algorithms. In Section 3, we discuss the polynomials in degree-optimal form, which are the most general polynomials of degree 2^m that can be evaluated with m multiplications. Degree-optimal polynomials can be represented naturally using our graph framework. Optimizing the coefficients of a graph so that the underlying algorithm approximates a function that cannot be expressed as a computational graph is the subject of Section 4. Section 5 is devoted to the description of two important features of our package: the generation of Julia, MATLAB, and C code for evaluating a computational graph efficiently, and the custom file format for storing computational graphs. Using a graph to approximate a function numerically is prone to truncation as well as round-off errors; these are discussed in Section 6, where we obtain a priori bounds on the former and a posteriori running bounds on the latter. In Section 7 we explain how the features described in previous sections can be integrated into a workflow for enhancing state-of-the-art algorithms, and present two instances in which such improvement can be encountered in practice. Finally, we summarize our contribution and outline possible directions of future investigation in Section 8.

2 Algorithms as graphs

2.1 Graph notation and operations

We regard matrix computations as graphs, specifically as directed acyclic graphs (DAGs), in which every node corresponds to a matrix. The leaves of the graph coincide with the input nodes, and the corresponding matrices are assumed to be known in advance. Every non-input node is constructed by performing an operation on two other matrices which can be identified by following the two incoming edges. Hence, input nodes have only outgoing edges, whereas every non-input node has exactly two incoming edges and zero or more outgoing edges. The output nodes represent the output (or outputs) of the computation.

Although many operations could potentially be considered in such a framework, in this work and in our software we have restricted our attention to the following:

- +, the linear combination of two matrices, $C = \alpha A + \beta B$;

- ∗, the product of two matrices, $C = AB$; and

- \, the solution of a linear systems with multiple right-hand sides, $C = A^{-1}B$.

With these three building blocks one can construct many algorithms of practical interest for evaluating matrix functions. Moreover, having only three operations all acting on exactly two matrices has several implementation advantages.

Let V be the set of nodes. For every node $i \in V$ we define:

- $p_i \in \{+, *, \backslash\}$, the operation required to evaluate node i;

---

[4]

- $e_i^{(1)} \in V$ and $e_i^{(2)} \in V$, the two nodes representing the matrices needed compute node i; and

- $c_i^{(1)}$ and $c_i^{(2)}$, the coefficients of the linear combination, defined only when $p_i = +$.

In order to evaluate the function underlying the computational graph, we need to explicitly construct the matrices each node represents. To this end, we define:

- Z_i, the content (matrix or scalar) of node i.

We will sometimes use the functional notation $Z_i(\cdot)$ to stress that Z_i depends on the content of the input nodes.

For notational convenience, here we denote the nodes by integers and write $V = \{0, \ldots, N\}$. In the software the set of nodes is implemented as an (unordered) dictionary lookup table for efficiency. The two input nodes are the identity matrix and the A matrix, which are labelled $Z_0 = I$ and $Z_1 = A$ in the matrix case, and $Z_0 = 1$ and $Z_1 = x$ in the scalar case.

The values of the other nodes are given by

$$Z_i(x) = \begin{cases} c_i^{(1)} Z_\ell(x) + c_i^{(2)} Z_r(x), & p_i = +, & \text{(1a)} \\ Z_\ell(x) Z_r(x), & p_i = *, & \text{(1b)} \\ Z_\ell(x)^{-1} Z_r(x), & p_i = \backslash, & \text{(1c)} \end{cases}$$

where $\ell = e_i^{(1)}$ and $r = e_i^{(2)}$.

The set of (directed) edges of the DAG is

$$\left\{ \left(e_2^{(1)}, 2\right), \left(e_2^{(2)}, 2\right), \left(e_3^{(1)}, 3\right), \left(e_3^{(2)}, 3\right), \ldots, \left(e_N^{(1)}, N\right), \left(e_N^{(2)}, N\right) \right\}.$$

These edges form a parent list. Therefore, if we assume that the graph has no directed cycles, by traversing it a in a depth-first fashion we can find a topological ordering based on the parent-child relationship. A topological ordering is a linear arrangement of the nodes in which every parent follows both of its children. The topological ordering is not unique, and the maximum number of nodes that are needed simultaneously at any stage is called the width of the DAG. Computing the width and the corresponding topological ordering is a difficult problem [8, 7], of fundamental importance, for example, in compiler optimization. In our setting, topological sorting will be of paramount importance, since a good ordering of the nodes decreases the memory footprint of the algorithm.

Example 1 (Graph evaluation and topological ordering) *The graph corresponding to the first four steps of the Denman–Beavers iteration [10] can be constructed with the command*

```julia
julia> using GraphMatFun
julia> (graph, _) = graph_denman_beavers(4); # 4 iterations
```

and can be evaluated at a matrix argument using `eval_graph`

```julia
julia> A = [0.5 0.2; 0.3 0.5];
julia> eval_graph(graph, A)
2×2 Matrix{Float64}:
 0.684065  0.146185
 0.219277  0.684065
julia> sqrt(A)   # Compare with the Julia's matrix square root
2×2 Matrix{Float64}:
 0.684065  0.146185
 0.219277  0.684065
```

(a) Denman–Beavers iteration for the square root [10].

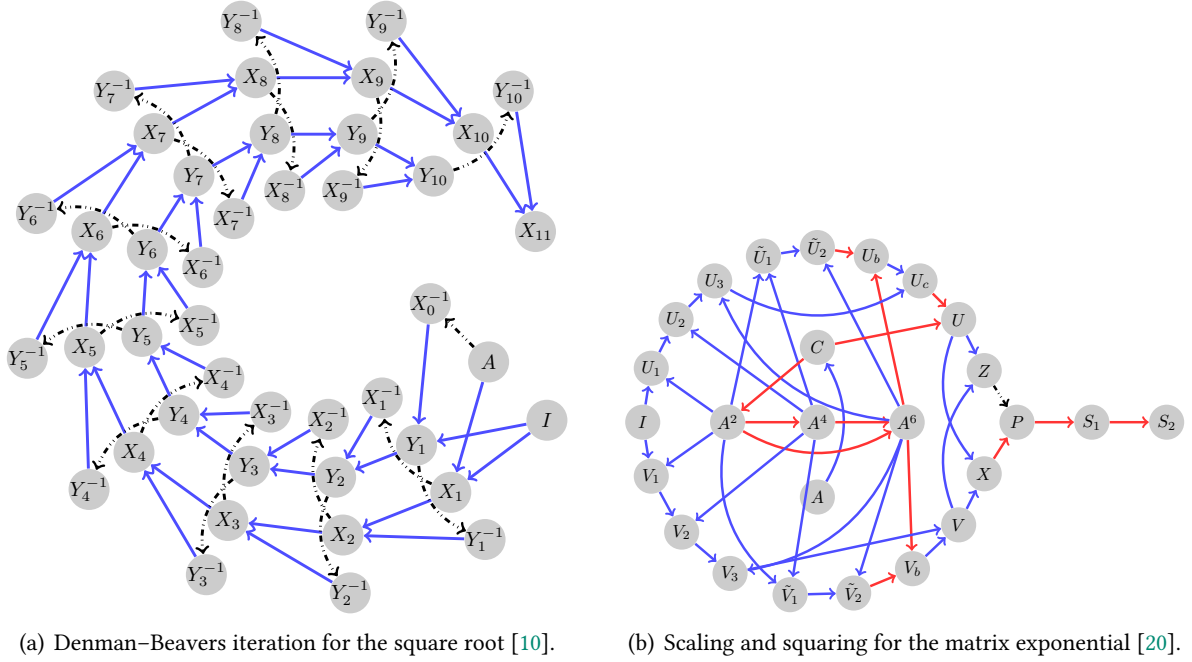(b) Scaling and squaring for the matrix exponential [20].

Figure 1: Examples of complex graphs. Blue lines represent linear combinations, red lines represent matrix multiplications, and black dash-dotted lines represent system solves. Nodes with only one visible parent are either the result of a squaring (A^2, A^4, S_1, S_2), or a suppressed trivial operation involving the identity matrix: a matrix inversion, represented as $C = A^{-1}I$, or a multiplication by a scalar, stored as $C = \alpha A + 0I$.

The graph is depicted in Figure 1(a). This evaluation relies on the ordering of the nodes computed by `get_topo_order`, *which can be examined by calling the latter directly*

```
julia> get_topo_order(graph)[1]
17-element Vector{Symbol}:
 :Xinv0
 :Y1
 :X1
 :Yinv1
 :X2
 ...
 :Xinv3
 :Y4
 :Yinv4
 :X5
```

Figure 1 shows the graphs produced by two algorithms of practical interest: the Denman–Beavers iteration for the square root [10] and the scaling and squaring algorithm for the matrix exponential based on a degree-13 Padé approximant [20].

2.2 Basic package features

The package provides a variety of functions for creating and manipulating graphs. The definition of the main data structure of the package is given in Figure 2. Procedures to generate directly graphs corresponding to several well-established algorithms are listed in Table 1. Table 2 and Table 3 list

5

```
struct Compgraph{T}
    operations::Dict{Symbol, Symbol}
    parents::Dict{Symbol, Tuple{Symbol, Symbol}}
    coeffs::Dict{Symbol, Tuple{T, T}}
    outputs::Vector{Symbol}
end
```

Figure 2: The fundamental data structure in the package is the `Compgraph`, which represents a graph and its associated operations. The node identifiers are `Symbols`. Since all operations are formed from two other nodes, the parent list is a dictionary (hash table) that associates a 2-element `Tuple` of parents to each non-input node.

functions that can be used to generate new graphs and manipulate existing ones. Their fundamental features demonstrated in the following examples.

Example 2 (Graph) *From a user's perspective, the easiest way to generate a graph is by using one of the graph generators in Table 1. The code corresponding to the graph can then be executed by calling* `eval_graph`.

```julia
julia> (g_mono, _) = graph_monomial([1.0, 0.0, 3.0]); # Create graph for 1+3x^2
julia> x = 0.1;
julia> eval_graph(g_mono, x)
1.03
julia> 1 + 3x^2
1.03
julia> A = [3.0 4.0; 5.0 6.0];
julia> P = eval_graph(g_mono, A) # can be used for scalars or matrices
2x2 Matrix{Float64}:
  88.0  108.0
 135.0  169.0
```

The graph is a `struct` *with four fields:*

```julia
julia> g_mono.parents # Every node has two parents
Dict{Symbol, Tuple{Symbol, Symbol}} with 3 entries:
  :P3 => (:P2, :A2)
  :A2 => (:A, :A)
  :P2 => (:I, :A)
julia> g_mono.operations # Every node has an associated operation
Dict{Symbol, Symbol} with 3 entries:
  :P3 => :lincomb
  :A2 => :mult
  :P2 => :lincomb
julia> g_mono.coeffs # The linear combination operations have coefficients
Dict{Symbol, Tuple{Float64, Float64}} with 2 entries:
  :P3 => (1.0, 3.0)
  :P2 => (1.0, 0.0)
julia> get_topo_order(g_mono)[1] # A computable order
3-element Vector{Symbol}:
 :A2
 :P2
 :P3
```

The definition of the data structure is given in Figure 2.

Example 3 (Compression) *Node manipulation may lead to graphs whose topology can be optimized in several ways, for example, by removing operations with zero coefficients. In Example 2, for instance, the second coefficients of node* :P2 *is zero, since the term x does not appear in the polynomial* $1 + 3x^2$ *represented by* g_mono. *We can therefore compress the graph and obtain a more efficient algorithm. We can achieve this by using* compress_graph!, *a function that optimizes the topology of the graph by removing 1) dangling nodes, i.e., non-output nodes that are not used anywhere in the graph; 2) trivial nodes, i.e., multiplication by the identity matrix or linear systems whose coefficient is the identity matrix; 3) redundant nodes, i.e., nodes that repeat computation already present in the graph; and 4) pass-through nodes, i.e., linear combinations whose coefficients include zero.*

```
julia> using BenchmarkTools;
julia> A=randn(100, 100);
julia> @btime eval_graph($g_mono, $A);
  172.168 µs (191 allocations: 330.75 KiB)
julia> compress_graph!(g_mono);
julia> g_mono.parents # The node :P2 has been removed
Dict{Symbol, Tuple{Symbol, Symbol}} with 2 entries:
  :P3 => (:I, :A2)
  :A2 => (:A, :A)
julia> @btime eval_graph($g_mono, $A); # It runs faster with fewer allocations
  133.430 µs (123 allocations: 246.20 KiB)
```

Example 4 (Manipulating graph topology) *The following illustrates how a graph can be manipulated by adding nodes with the* add_<operation> *functions in Table 2.*

```
julia> g_mono.outputs # This is the output node
1-element Vector{Symbol}:
 :P3
julia> add_mult!(g_mono, :PX, :P3, :P3); # Let's square the output node
julia> clear_outputs!(g_mono);
julia> add_output!(g_mono, :PX); # And set a new output node
julia> eval_graph(g_mono, x);
1.0609
julia> (1 + 3x^2)^2
1.0609
```

3 Degree-optimal polynomials

In numerical algorithms for matrix functions, it is customary to reduce the task of computing a function f at a matrix A to the evaluation of $p(A)$, where p is a polynomial p that approximates f. Evaluating $p(A)$ requires only matrix-matrix multiplications and linear combinations, and since the computational cost of the former asymptotically dominates that of latter, it is crucial to use polynomial evaluation schemes that require as few matrix-matrix multiplications as possible. The Paterson–Stockmeyer method [27] requires fewer matrix-matrix products than the naive scheme in which the powers of A are computed explicitly, and is currently the most efficient method for evaluating $p(A)$ using only the original monomial coefficients of p [11]. If the use of new coefficients derived from those of p is allowed, however, then even more economical schemes, such as [27, Algorithms B and C], are possible. Fuelled by the seminal work of Sastre [30], the topic has received renewed attention in recent years, and algorithms that combine this scheme with the scaling and squaring technique have been developed for the exponential [2, 31], the sine and cosine [34, 36], and the hyperbolic tangent [23].

Table 1: Graph generators. The three columns provide the name of the function in our package, the function being evaluated/approximated, and a reference for the underlying algorithm. The degree-optimal form is discussed in Section 3.

Package function	$f(x)$	Implemented algorithm
graph_monomial	$p(x)$	Standard monomial evaluation
graph_monomial_degopt	$p(x)$	Above embedded into degree-optimal form
graph_horner	$p(x)$	Horner scheme evaluation
graph_horner_degopt	$p(x)$	Above embedded into degree-optimal form
graph_ps	$p(x)$	Paterson–Stockmeyer [27]
graph_ps_degopt	$p(x)$	Above embedded into degree-optimal form
graph_rational	$r(x)$	Rational with poly eval with any of above
graph_denman_beavers	$\sqrt{x}$	Denman–Beavers iteration [10]
graph_newton_schulz	x^{-1}	Newton–Schulz iteration [35]
graph_newton_schulz_degopt	x^{-1}	Above embedded into degree-optimal form
graph_exp_native_jl	e^x	Scaling and squaring with Padé approximant [20]
graph_sastre_basic_exp	e^x	Efficient evaluation of Taylor approximant [30]
graph_sid_exp	e^x	Scaling and squaring with Taylor approximant [30, 31]
graph_bbc_basic_exp	e^x	Scaling and squaring with Taylor approximant [2]
graph_bbcs_cheb_exp	e^x	Chebyshev approximant for skew-Hermitian matrices [3]
graph_sastre_yks_degopt	$p(x)$	Transforms y_{ks}-form [30] to degree-optimal form

Table 2: Fundamental node manipulation functions. The add_sum! function is a convenience function implemented by multiple calls to add_lincomb!.

Package function	Operation
add_lincomb!	Adds a new node $Z_j = c_j^{(1)} Z_\ell + c_j^{(2)} Z_r$
add_mult!	Adds a new node $Z_j = Z_\ell Z_r$
add_ldiv!	Adds a new node $Z_j = Z_\ell^{-1} Z_r$
add_sum!	Adds a new node $Z_j = \alpha_1 Z_{m_1} + \cdots + \alpha_n A_{z_{m_n}}$
del_node!	Removes an existing node
rename_node!	Changes the label (Symbol) of an existing node in all graph data structures
add_output!	Marks a node as output node
clear_outputs!	Clears the list of output nodes

Table 3: Graph manipulation and evaluation functions.

Package function	Operation
Compgraph(T)	Creates an empty graph with coefficients of type T
Compgraph(T,orggraph)	Converts the graph orggraph so that all its coefficients have type T
eval_graph	Evaluates the graph at a given matrix, scalar, or other object
eval_jac	Evaluates the Jacobian at a point (element-wise for a vector)
compress_graph!	Removes unnecessary operations from the graph
merge_graph	Combines two graphs into one and renames the nodes accordingly

Polynomials of degree 2^m can be obtained with m multiplications by repeated squaring. This is optimal, in the sense that one cannot obtain polynomials of higher degree for a fixed number of matrix multiplications. Since matrix multiplications are computationally much more expensive than matrix additions, we can consider a larger class of polynomials obtained by taking linear combinations of previously computed matrices. This will increase the number degrees of freedom at a negligible cost. We now introduce the *degree-optimal polynomials*, which are constructed recursively by forming, at each step, the product of two linear combinations of all previously computed matrices. These reach the optimal degree, and parameterize all polynomials that can be evaluated with at most m non-scalar multiplications.

Starting with the recursion

$$A_0 = I \tag{2a}$$

$$A_1 = A \tag{2b}$$

$$A_2 = (x_{a,1,1}A_0 + x_{a,1,2}A_1)(x_{b,1,1}I + x_{b,1,2}A_1) \tag{2c}$$

$$A_4 = (x_{a,2,1}A_0 + x_{a,2,2}A_1 + x_{a,2,3}A_2)(x_{b,2,1}A_0 + x_{b,2,2}A_1 + x_{b,2,3}A_2) \tag{2d}$$

$$A_8 = (x_{a,3,1}A_0 + \cdots + x_{a,3,4}A_4)(x_{b,3,1}A_0 + \cdots + x_{b,3,4}A_4) \tag{2e}$$

$$\vdots$$

$$A_{2^m} = (x_{a,m,1}A_0 + \cdots + x_{a,m,m+1}A_{2^{m-1}})(x_{b,m,1}A_0 + \cdots + x_{b,m,m+1}A_{2^{m-1}}) \tag{2f}$$

the linear combination of all the computed matrices

$$p(A) = y_1 I + y_2 A_1 + \cdots + y_{m+2} A_{2^m}, \tag{2g}$$

is a polynomial of degree at most 2^m. Determining the coefficients in (2) is the most challenging aspect of this approach: they are, in a sense, too many and not enough at the same time.

- The coefficients $x_{a,1,1}, \ldots, x_{b,m,m+1}$ and $y_1, \ldots, y_{m+1}$ parameterize only a subset of the vector space of polynomials of degree at most 2^m. It is easy to see that (2) has only $m^2 + 4m + 2$ parameters, which is strictly less than the 2^m coefficients of the polynomial for $m \geq 6$.

- There is redundancy in the coefficients: setting $x_{a,1,1} = 1$, for example, does not change the set of parameterized polynomials. In fact, the formula (2) does not parameterize all polynomials of degree 2^m for any $m \geq 4$.

The coefficients in (2) can be represented in a compact form by means of two matrices of dimension $m \times (m + 1)$ and one vector of length $m + 2$. We define H_a, H_b, and y as

$$\left[\, H_a \,\middle|\, H_b \,\right] = \left[\begin{array}{cccc|cccc} x_{a,1,1} & x_{a,1,2} & & & x_{b,1,1} & x_{b,1,2} & & \\ x_{a,2,1} & x_{a,2,2} & x_{a,2,3} & & x_{b,2,1} & x_{b,2,2} & x_{b,2,3} & \\ \vdots & & \ddots & \ddots & \vdots & & \ddots & \ddots \\ x_{a,m,1} & \cdots & \cdots & x_{a,m,m}\; x_{a,m,m+1} & x_{b,m,1} & \cdots & \cdots & x_{b,m,m}\; x_{b,m,m+1} \end{array}\right], \tag{3a}$$

$$y = \begin{bmatrix} y_1 & \cdots & y_{m+2} \end{bmatrix}. \tag{3b}$$

Some evaluation schemes in the literature [2, 30] can be expressed in a degree-optimal form where certain coefficients are fixed to zero or one—in fact (2) is mentioned, though not used extensively, in [2]. In these cases, the free coefficients are computed from algebraic system of equations so that the resulting polynomial matches the coefficients of a Taylor expansion of the function of interest. The algebraic system is then solved symbolically, but the solver may fail to find a solution if the "wrong" coefficients are fixed, and different choices of fixed parameters can lead to an infinity of solutions, one of which has to be selected in practice [2]. The authors also point out that there may be

approaches that are more economical than those obtained by their particular choice of free and fixed parameters. As can be seen in Table 4, both approaches lead to polynomials of degree far lower than the maximum obtainable and considerably below the number of degrees of freedom in the scheme (2). New approaches to determine the coefficients using symbolic computation have recently been proposed [33].

Remark 5 (Related notation) *Some literature focusing on minimizing the number of matrix-matrix multiplications [30, 31, 34] uses a slightly different compact form to express multiplication-efficient polynomial evaluation. The authors prefer the notation y_{ks} where s is the highest power of the input matrix that has to be computed and k is a level parameter related to m in the degree-optimal form. These compact forms can always be converted to a degree-optimal form. We exemplify it with the expressions in [30, Example 3.1]:*

$$y_{02}(x) = x^2(c_4x^2 + c_3x) \tag{4a}$$
$$y_{12}(x) = (y_{02}(x) + d_2x^2 + d_1x)(y_{02}(x) + e_2x^2) + e_0y_{02}(x) + f_2x^2 + f_1x + f_0. \tag{4b}$$

For any polynomial of degree 8 with non-zero leading coefficient, there exists coefficients c_4, c_3, d_2, d_1, e_2, e_0, f_2, f_1, and f_0 such that $y_{12}(x)$ is an evaluation scheme of the polynomial [30]. In the degree-optimal form, we have one row per multiplication, and a final summation of all computed matrices. The coefficients of the degree-optimal form for (4) are

$$
\left[
\begin{array}{cccc|cccc}
x_{a,1,1} & x_{a,1,2} & & & x_{b,1,1} & x_{b,1,2} & & \\
x_{a,2,1} & x_{a,2,2} & x_{a,2,3} & & x_{b,2,1} & x_{b,2,2} & x_{b,2,3} & \\
x_{a,3,1} & x_{a,3,2} & x_{a,3,3} & x_{a,3,4} & x_{b,3,1} & x_{b,3,2} & x_{b,3,3} & x_{b,3,4}
\end{array}
\right]
=
\left[
\begin{array}{cccc|cccc}
0 & 1 & & & 0 & 1 & & \\
0 & 0 & 1 & & 0 & c_3 & c_4 & \\
0 & d_1 & d_2 & 1 & 0 & 0 & e_2 & 1
\end{array}
\right] \tag{5}
$$

with

$$
\begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & e_0 & 1 \end{bmatrix}. \tag{6}
$$

With the notation (3), we can express [30, Equation (34)-(35)] for $s = 3$ as

$$
\left[
\begin{array}{ccccc|ccccc}
0 & 1 & & & & 0 & 1 & & & \\
0 & 0 & 1 & & & 0 & 1 & 0 & & \\
0 & 0 & 0 & 1 & & 0 & c_4 & c_5 & c_6 & \\
0 & d_1 & d_2 & d_3 & 1 & 0 & 0 & e_2 & e_3 & 1
\end{array}
\right] \tag{7}
$$

with

$$
\begin{bmatrix} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 & e_0 & 1 \end{bmatrix}.
$$

The referenced work contains an extra addition of previous iterates after each multiplication. In order to transform this scheme to degree-optimal form, these terms can always be compensated for by merging into appropriate coefficients in subsequent rows in the degree-optimal form.

The function `graph_sastre_yks_degopt` *automatically converts the coefficients in the scheme [30, Equations (62)-(65)] for the evaluation of y_{ks} to degree-optimal form. We wish to stress that there is a redundancy in the degree-optimal form, and to our knowledge also in the y_{ks}-form proposed in [30]. The first row of (5) and (7), for example, can be set to*

$$
\begin{bmatrix} x_{a,1,1} & x_{a,1,2} & x_{b,1,1} & x_{b,1,2} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}, \tag{8}
$$

without loss of generality. Our software is based on the degree-optimal polynomial with arbitrary coefficients, and does not explicitly remove this and other redundancies removed in the y_{ks}-form. This is mostly due to implementation considerations: H_a and H_b having the same structure leads to more compact code. It is straightforward to impose constraints such as (8) when optimizing the coefficients of the graph, since we have the freedom to choose the parameters over which to optimize.

The general scheme (2) can clearly be phrased as a computational graph. We provide graph generators that use the coefficients tabulated in previous works for certain cases. In general, we propose to approach the choice the coefficients in (2) as an optimization problem, as discussed in details in Section 4. As we will see in the experiments, determining the coefficients by optimization can bring several advantages. In this setting, procedures such as those in [2] and [30] can be used to warm start the optimizer.

Example 6 (degree-optimal form) *The degree-optimal polynomials are represented by* Degopt *objects, which can be initialized with a constructor that takes in input the matrices in* (3). *A degree-optimal polynomial can be evaluated by using the corresponding* Compgraph, *which can be generated using the function* graph_degopt. *Here is the Paterson–Stockmeyer evaluation corresponding to the truncated Taylor expansion of the exponential.*

```
julia> c = 1 ./ factorial.(0:11); # Truncated Taylor expansion of exp(z) at z=0
julia> HA = [0.0   1.0    0.0    0.0    0.0   0.0
             0.0   0.0    1.0    0.0    0.0   0.0
             0.0   0.0    0.0    1.0    0.0   0.0
             0.0   0.0    0.0    0.0    1.0   0.0
             0.0   0.0    0.0    0.0    1.0   0.0];
julia> HB = [0.0   1.0    0.0    0.0    0.0   0.0
             0.0   1.0    0.0    0.0    0.0   0.0
             0.0   1.0    0.0    0.0    0.0   0.0
             c[9]  c[10]  c[11]  c[12]  0.0   0.0
             c[5]  c[6]   c[7]   c[8]   1.0   0.0]
julia> y=[c[1:4]; 0; 0; 1]
julia> (g, _) = graph_degopt(Degopt(HA, HB, y));
julia> A=[1 2; 3 4] / 100;
julia> eval_graph(g, A)
2x2 Matrix{Float64}:
 1.01036     0.0205091
 0.0307637   1.04112
julia> norm(exp(A) - eval_graph(g, A))
7.042759341472802e-11
```

Since the degree-optimal form contains many zeros in this case, the graph will contain many nodes corresponding to scalar multiplications by zero. These can be automatically removed by the function compress_graph!, *which substantially reduces the computation time as well as the number of allocations.*

```
julia> using BenchmarkTools;
julia> g2 = compress_graph!(deepcopy(g));
julia> @btime eval_graph($g, $A);
  1.247 ms (6271 allocations: 761.16 KiB)
julia> @btime eval_graph($g2, $A);
  99.143 µs (829 allocations: 81.49 KiB)
```

4   Algorithm design by optimization

The graphs that can be constructed from O1–O3 are polynomials and rational functions. Let g be the function underlying a graph with fixed topology. Our aim is to determine the coefficients of the linear combinations in the graph so that the function g approximates a given function f in a domain of interest. We mostly use optimization based on the forward error in exact or high-precision arithmetic. The analysis of the backward and round-off error are presented in Section 6.

Throughout this section we are going to assume that f and g are analytic in a simply connected domain $\Omega \subset \mathbb{C}$. Occasionally we will use the notation $g(\cdot\,; c)$ to emphasize that g depends on a

$$
\left[\begin{array}{ccccc|ccccc}
0 & 1 & & & & 0 & 1 & & & \\
0 & 0 & 1 & & & 0 & 1 & 0 & & \\
0 & 0 & 0 & 1 & & 0 & 1 & 0 & 0 & \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}\right]
$$

$$ y = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{bmatrix} $$

(a) Monomial evaluation of $c_0 x^0 + \cdots + c_6 x^6$.

$$
\left[\begin{array}{ccccc|ccccc}
c_5 & c_6 & & & & 0 & 1 & & & \\
c_4 & 0 & 1 & & & 0 & 1 & 0 & & \\
c_3 & 0 & 0 & 1 & & 0 & 1 & 0 & 0 & \\
c_2 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
c_1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}\right]
$$

$$ y = \begin{bmatrix} c_0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} $$

(b) Horner evaluation of $c_0 x^0 + \cdots + c_6 x^6$.

$$
\left[\begin{array}{ccccc|ccccc}
0 & 1 & & & & 0 & 1 & & & \\
0 & 0 & 1 & & & 0 & 1 & 0 & & \\
0 & 0 & 0 & 1 & & 0 & 1 & 0 & 0 & \\
0 & 0 & 0 & 0 & 1 & c_7 & c_9 & c_{10} & c_{11} & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & c_4 & c_5 & c_6 & c_7 & 0 & 1
\end{array}\right]
$$

$$ y = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & 1 \end{bmatrix} $$

(c) Paterson–Stockmeyer evaluation of $c_0 x^0 + \cdots + c_{11} x^{11}$.

$$
\left[\begin{array}{cccccc|cccccc}
0 & 1 & & & & & 0 & 1 & & & & \\
0 & 0 & 1 & & & & 0 & 1 & 0 & & & \\
0 & 0 & 0 & 1 & & & 0 & 1 & 0 & 0 & & \\
0 & 0 & 0 & 0 & 1 & & 0 & c_4 & c_3 & c_2 & c_1 & \\
0 & c_8 & c_7 & c_6 & c_5 & 1 & 0 & 0 & c_{11} & c_{10} & c_9 & 1 \\
0 & c_{16} & c_{15} & c_{14} & c_{13} & c_{12} & 1 & 0 & c_{20} & c_{19} & c_{18} & c_{17} & 1 & 0
\end{array}\right]
$$

$$ y = \begin{bmatrix} 1 & 1 & c_{23} & c_{22} & c_{21} & 0 & 0 & 1 \end{bmatrix} $$

(d) Truncated exponential using [31, Equations (26)-(28)].

Figure 3: Examples various polynomial evaluation schemes expressed in degree-optimal form. We use the matrix notation in (3).

coefficient vector $c \in \mathbb{C}^K$. The forward error is $e(A) = f(A) - g(A)$, and for any diagonalizable matrix $A \in \mathbb{C}^{n \times n}$ with spectrum in Ω we have the bound

$$ \|e(A)\| \le \kappa(V) \max_i |e(\lambda_i)| \le \kappa(V) \max_{z \in \Omega} |e(z)|, \tag{9} $$

where $V \in \mathbb{C}^{n \times n}$ is a matrix such that $\Lambda = V A V^{-1}$ is diagonal, and $\kappa(V) = \|V\| \|V^{-1}\|$ denotes the spectral condition number. Hence, we are interested in minimizing $e(z)$ over Ω. If f is analytic in Ω, then the objective function can be further simplified by applying the maximum modulus principle [28, Section 12.1], which dictates that the maximum of f over Ω is attained on the boundary. Therefore, if we denote the boundary of Ω with Γ, our task is to minimize

$$ \max_{z \in \Gamma} |f(z) - g(z; c)| $$

with respect to the K variables in the coefficient vector c. The resulting optimization problem is nevertheless difficult, thus we introduce the surrogate problem

$$ \min_{c \in \mathbb{C}^K} \|r\|^2, \tag{10} $$

where r is the vector of residuals

$$ r = \begin{bmatrix} f(z_1) - g(z_1; c) \\ f(z_2) - g(z_2; c) \\ \vdots \\ f(z_N) - g(z_N; c) \end{bmatrix}, \tag{11} $$

for a discretization $z_1, z_2, \ldots, z_N$ of Γ. The relation (10) is a nonlinear least square problem which can be tackled with methods such as the Gauss–Newton algorithm [5]; see [37] for an extension to the complex case. The method updates the current guess for c with a vector δ that is the solution to the linear least squares problem

$$ \arg \min_{\delta \in \mathbb{C}^K} \|J\delta - r\|, \tag{12} $$

Table 4: Highest-degree polynomial that can be evaluated with a given number of multiplications using different schemes. The asterisk * denotes the degree reported in the paper, but without documentation of the coefficients. Sastre's approach [30] is already optimal for $m = 0, 1, 2, 3$ since the degree of the polynomial is the same as the degree of (2g). The approach in [31] is very close to optimal for $m = 4$. The plus sign $^+$ marks the highest coefficient in the Taylor series expansion that is matched by the polynomial, the approximating polynomial is higher. Sastre [30] refers to the variants in [30, Section 3.1] and [30, Section 5].

Number of matrix products m	1	2	3	4	5	6	7	8
Paterson–Stockmeyer degree	2	4	6	9	12	16	20	25
Sastre [30] degree	2	4	8	$12^*(15^+)$	16^*	20^*	25^*	30
BBC [2] degree	2	4	8	12	18	22^*		
SID [31] degree	2	4	8	15^+	21^+	24	30	
Degree of (2g): 2^m	2	4	8	16	32	64	128	256
Number of free variables in (2): $(m+2)^2 - 2$	7	14	23	34	47	62	79	98

where J is the Jacobian of g with respect to the coefficient vector c, which can be written as

$$J = \begin{bmatrix} \partial g(z_1;c)/\partial c_1 & \partial g(z_1;c)/\partial c_2 & \ldots & \partial g(z_1;c)/\partial c_K \\ \partial g(z_2;c)/\partial c_1 & \partial g(z_2;c)/\partial c_2 & \ldots & \partial g(z_2;c)/\partial c_K \\ \vdots & \vdots & \ddots & \vdots \\ \partial g(z_N;c)/\partial c_1 & \partial g(z_N;c)/\partial c_2 & \ldots & \partial g(z_N;c)/\partial c_K \end{bmatrix}. \tag{13}$$

In our setting, the Jacobian can be computed in a manner that resembles backpropagation. This technique is commonly used to compute derivatives when training neural networks [29, 17], which can themselves be regarded as computational graphs [17, Section 6.5]. Backpropagation applies the chain rule to the operations in the computational graph, and by exploiting the structure of the DAG we can derive explicit rules for the derivatives. The derivative of (1) with respect to the coefficient c_k is

$$\frac{\partial Z_i(x)}{\partial c_k} = \begin{cases} c_i^{(1)} \dfrac{\partial Z_\ell(x)}{\partial c_k} + c_i^{(2)} \dfrac{\partial Z_r(x)}{\partial c_k} & p_i = +, \tag{14a} \\[2ex] \dfrac{\partial Z_\ell(x)}{\partial c_k} Z_r(x) + Z_\ell(x) \dfrac{\partial Z_r(x)}{\partial c_k}, & p_i = *, \tag{14b} \\[2ex] -Z_\ell(x)^{-2} \dfrac{\partial Z_\ell(x)}{\partial c_k} Z_r(x) + Z_\ell(x)^{-1} \dfrac{\partial Z_r(x)}{\partial c_k}, & p_i = \backslash. \tag{14c} \end{cases}$$

The rules can be applied recursively until a linear combination node $\bar{\imath}$ such that $c_k = c_{\bar{\imath}}^{(1)}$ or $c_k = c_{\bar{\imath}}^{(2)}$ is found. In this case, one has

$$\frac{\partial Z_{\bar{\imath}}(x)}{\partial c_k} = Z_k(x). \tag{14d}$$

The propagation formulae (14) can also be used to implement the forward evaluation of the partial derivatives, that is, to evaluate the derivative of g with respect to c_k at a point x. One can start from node $\bar{\imath}$, using (14d), and then follow the parent list of the graph according to the topological ordering. Visiting the nodes in this order guarantees that whenever one of the formulae (14a)-(14c) is evaluated, the two partial derivatives on the right-hand side have already been computed or are zero. The computation can also be carried out simultaneously for all the points $z_1, z_2, \ldots, z_N$.

Example 7 (Jacobian evaluation) *The function* `eval_jac` *evaluates the Jacobian of a graph using the method described above. We illustrate the process using as model the graph for the degree-5 Taylor approximant to the exponential in monomial form. Evaluating this approximant requires only 4 multiplications.*

```julia
julia> c = 1 ./ factorial.(0:5); # Truncated Taylor expansion of exp(z) at z=0
julia> (graph, cref) = graph_monomial(c); # Exponential coeffs
julia> discr = 0.45*exp.(1im * range(0, 2π, length=200)); # Discretization
julia> J = eval_jac(graph, discr, cref);
julia> size(J)
(200, 6)
julia> svdvals(J)
6-element Vector{Float64}:
 14.142189931772608
  6.363885389264312
  2.863711391309838
  1.2886540714299903
  0.579886987450398
  0.2609452239018225
```

The second output of the graph generator, `cref`, *is a vector of tuples* (Symbol, Int) *that references a subset of the coefficients in the graph. In this case,* `cref` *contains the 6 coefficients of the monomial.*

```julia
julia> [cref   get_coeffs(graph, cref)]
6×2 Matrix{Any}:
 (:P2, 1)   1.0
 (:P2, 2)   1.0
 (:P3, 2)   0.5
 (:P4, 2)   0.166667
 (:P5, 2)   0.0416667
 (:P6, 2)   0.00833333
```

If the same graph is embedded in degree-optimal form, then `cref` *contains all the coefficients listed in* (3).

```julia
julia> (graph, cref) = graph_monomial_degopt(c);
julia> J = eval_jac(graph, discr, cref);
julia> size(J)
(200, 34)
julia> svdvals(J)[1:10]
10-element Vector{Float64}:
 14.142241129421516
  7.886645240829699
  3.6166303096103856
  1.347686765969578
  0.590394286817795
  0.26144313322720497
  0.005193701527333443
  0.00046141908865473055
  0.0001981485633788004
  8.380655750040214e-16
```

The flexibility of the degree-optimal form is evident, as 4 multiplications are parameterized by 34 coefficients. The redundancies are also apparent: in this example only 9 singular values of the Jacobian are numerically larger than zero. These 9 degrees of freedom are more than the 6 available in the corresponding classical polynomial evaluation schemes. Note that the number of zero singular values is a local property, which depends on the graph as well as on the values in `c`. *No global conclusions should therefore be drawn from it.*

The general idea of the Gauss–Newton algorithm adapted to the minimization problem (10) is

---

Algorithm 1: Gauss–Newton algorithm for the graph nonlinear least squares (10)

    input: A graph, the coefficients c_0 corresponding to $g(\,\cdot\,;c_0)$, the step length $\gamma > 0$.
    output: A coefficient vector c such that $g(\,\cdot\,;c) \approx f$, if the iterations converge.

1 **for** $k = 1, 2, 3, \ldots$ **do**
2   Evaluate residual vector r from (11).
3   Evaluate the Jacobian J from (13) by using the rules in (14).
4   Solve (12) for δ_k using the SVD and the pseudoinverse with a drop tolerance d_k.
5   Update $c_{k+1} = c_k + \gamma \delta_k$.
6 **end**

---

summarized in Algorithm 1. Currently, our package provides a custom implementation of the Gauss–Newton algorithm which gives good control of the damping parameters and makes it easy to use the `BigFloat` (high-precision arithmetic) data type and work with real-valued coefficients, see Remark 8. In the future, we may consider integrating optimizers from the Julia ecosystem. That would allow for the use of more advanced routines for nonlinear least squares problems, such as, for example, the Levenberg–Marquardt algorithm or general trust region methods [5].

Remark 8 (Real-valued coefficients) *The nonlinear least squares problem* (10) *is a complex-valued problem. However, in many cases it is desirable to require that $c \in \mathbb{R}^K$. The problem can then be turned into a real-valued nonlinear least squares problem by reformulating the objective function as*

$$\min_{c \in \mathbb{R}^K} \left\| \begin{bmatrix} \mathrm{Re}(r) \\ \mathrm{Im}(r) \end{bmatrix} \right\|^2.$$

The corresponding linear least squares (sub-)problem, cf. (12)*, becomes*

$$\arg\min_{\delta \in \mathbb{R}^K} \left\| \begin{bmatrix} \mathrm{Re}(J) \\ \mathrm{Im}(J) \end{bmatrix} \delta - \begin{bmatrix} \mathrm{Re}(r) \\ \mathrm{Im}(r) \end{bmatrix} \right\|,$$

as described in [37].

Remark 9 (Relative error) *The minimization problem in* (10) *is derived from the absolute error. However, we often want to minimize the relative error $\|e(A)\|/\|f(A)\|$. If f has no roots inside the domain Ω, then the derivation is analogous and the result is that the Jacobian J and the residual vector r in the linear least squares problem* (12) *are diagonally scaled with $D = \mathrm{diag}(1/f(z_i))$, i.e., the linear least squares (sub-)problem in the Gauss–Newton algorithm is $\arg\min_\delta \|DJ\delta - Dr\|$. If f has roots in the domain, however, then the maximum modulus principle is not applicable to the relative error, hence we sometimes resort to the use of absolute error.*

Example 10 (Design by optimization) *We illustrate how the optimization can be used to find coefficients for a degree-optimal polynomial of the form* (2) *with 4 multiplications that approximates the exponential. This example builds on Example 7, and is a simplified version of the more extensive cases reported in Section 7. As starting guess we use the coefficients of the degree-5 Taylor approximant embedded in degree-optimal form, analogous to Figure 3(a).*

```julia
julia> (graph, cref) = graph_monomial_degopt(c); # Initial guess
julia> f = exp;
julia> A = randn(100, 100) / 40;
julia> fA = f(A);
julia> norm(eval_graph(graph, A) - fA) / norm(fA) # Test initial guess
3.521697764659818e-7
```

The second output in the graph generation, `cref`*, is described in Example 7, and represents the coefficients we use in the optimization. We apply Gauss–Newton to find an improved graph.*

```julia
julia> graphb = big(graph); # Do optimization in high precision
julia> discr = 0.45 * exp.(1im*range(0,2big(π), length=200)); # Discretization
julia> using GenericSVD # SVD on BigFloat matrices
julia> opt_gauss_newton!(graphb, f, discr,  # Run optimization
                         errtype = :relerr, # Relative error
                         stoptol = 4e-15,
                         cref = cref, # Explicitly giving target variables
                         linlsqr = :real_svd, # Force real coefficients
                         droptol = 1e-15,
                         logger = 0 # Set to 1 to see iterations
                         );
julia> graph = Compgraph(Float64, graphb); # Round to Float64
julia> norm(eval_graph(graph, A) - fA) / norm(fA) # Test result
3.3790722357852773e-16
```

However, with 4 multiplications the Paterson–Stockmeyer scheme can evaluate polynomials of degrees up to 9 [30]. Hence, we can compare the approximant thus obtained with the truncated Taylor series of degree 9.

```julia
julia> c_ps = 1 ./ factorial.(0:9); # Truncated Taylor expansion of exp(z) at z=0
julia> (ps, _) = graph_ps(c_ps); # Paterson--Stockmeyer evaluation
julia> norm(eval_graph(ps, A) - fA) / norm(fA)
2.940461227704252e-13
```

The flexible ways of evaluating the graph allows us to evaluate the equivalent monomial coefficients. We can compare these to the coefficients of the truncated Taylor series of degree 16, since that is the optimal degree for polynomials that can be evaluated using only 4 multiplications.

```julia
julia> using Polynomials
julia> x = Polynomial("x");
julia> c = coeffs(eval_graph(Compgraph(Any, graph), x)); # Coefficients of graph
julia> d = 1 ./ factorial.(0:16); # Truncated Taylor expansion of exp(z) at z=0
julia> [c  d  abs.(c - d) ./ d] # Graph | Taylor | Relative difference
17x3 Matrix{Float64}:
 1.0          1.0          2.22045e-16
 1.0          1.0          2.22045e-16
 0.5          0.5          2.22045e-16
 0.166667     0.166667     4.996e-16
 0.0416667    0.0416667    1.4988e-15
 0.00833333   0.00833333   1.39472e-14
 0.00138889   0.00138889   1.85321e-13
 0.000198413  0.000198413  2.87713e-12
 2.48016e-5   2.48016e-5   5.087e-11
 2.75573e-6   2.75573e-6   9.40872e-10
 2.75573e-7   2.75573e-7   1.83869e-9
 2.5052e-8    2.50521e-8   3.85341e-6
 2.08628e-9   2.08768e-9   0.000666946
 1.51274e-10  1.6059e-10   0.0580114
 8.45802e-12  1.14707e-11  0.262645
 3.03244e-13  7.64716e-13  0.603455
 5.15212e-15  4.77948e-14  0.892203
```

Remark 11 (Jordan form and generalization of diagonal bound (9)) *If A is defective, then defining $f(A)$ requires the ability to compute some of the derivatives of f at some eigenvalues of A, as can be seen from the definitions via Jordan canonical form [21, Definition 1.2] or via Hermite interpolation [21, Definition 1.3]. Therefore, in order to minimize $\|e(A)\|$ it might be desirable to ensure that the derivatives*

of g approximate those of f. Since f and g are analytic on Ω, so is the error e, and we can thus bound its derivatives.

To do this we consider Ω being embedded in a slightly larger region Ω_ρ. If $D(a, \rho)$ denote the disc of radius ρ centered in a, then Ω_ρ is the smallest region such that for every point $a \in \Omega$ one has $D(a, \rho) \subset \Omega_\rho$. We assume that f and g are analytic in Ω_ρ. For reasons that will become clear later, it is preferable to set $\rho \geq 1$. Since $D(a, \rho) \subset \Omega_\rho$ for all points $a \in \Omega$, by Cauchy's estimates [28, Theorem 10.26] we have that

$$\left| f^{(k)}(a) - g^{(k)}(a) \right| \leq \frac{k! M_{a,\rho}}{\rho^k},$$

where $M_{a,\rho}$ is a constant such that $\left| f(z) - g(z) \right| \leq M_{a,\rho}$ for all $z \in D(a, \rho)$. Naturally for any $a \in \Omega$ we have $M_{a,\rho} \leq M_\rho$ where M_ρ is a constant such that $|f(z) - g(z)| \leq M_\rho$ for all $z \in \Omega_\rho$. Hence, the derivative bounds can be applied uniformly giving

$$\left| f^{(k)}(z) - g^{(k)}(z) \right| \leq \frac{k! M_\rho}{\rho^k},$$

for all $z \in \Omega$. Thus, if g approximates f well in the slightly larger domain Ω_ρ, then we can bound how well the derivatives of g approximates those of f on Ω. In the context of matrix functions this implies that for a Jordan block J_k of size $k \times k$ it holds that

$$\left\| f(J_k) - g(J_k) \right\| \leq \left\| \left| f(J_k) - g(J_k) \right| \right\| \leq M_\rho \| P_k(\rho) \|,$$

where $| \cdot |$ denotes the entry-wise absolute value and $P_k(\rho)$ is the matrix

$$P_k(\rho) = \begin{bmatrix} 1 & \frac{1}{\rho} & \frac{1}{\rho^2} & \cdots & \frac{1}{\rho^{k-1}} \\ & 1 & \frac{1}{\rho} & \ddots & \vdots \\ & & 1 & \ddots & \frac{1}{\rho^2} \\ & & & \ddots & \frac{1}{\rho} \\ & & & & 1 \end{bmatrix}.$$

We can conclude that

$$\| e(A) \| \leq \kappa(S) \| P_{k'}(\rho) \| \max_{z \in \Omega_\rho} | e(z) |$$

where k' is the size of the largest Jordan block of A, and S is a transformation matrix such that $J = SAS^{-1}$ is a Jordan canonical form of A.

Remark 12 (Crouzeix's conjecture and generalization of diagonal bound ([9])) *Let Ω be a smooth, bounded, and convex domain which contains the field of values*

$$W(A) = \{ x^* A x : x \in \mathbb{C}^n, \| x \| = 1 \}.$$

By [9, Theorem 3.1], if e is analytic in Ω and continuous on its boundary, then

$$\| e(A) \| \leq \left(1 + \sqrt{2} \right) \sup_{z \in \Omega} | e(z) |,$$

Let $\Omega = D(0, r)$, and let us assume that f and g are analytic in $D(0, r)$, which implies that so is the error e. Consider the numerical radius $w(A) = \sup\{ |z| : z \in W(A) \}$. For any matrix A such that $\| A \| \leq r$ the numerical radius satisfies $w(A) \leq r$ [22, Problem 1.5.24(g)], and thus, the numerical range satisfies $W(A) \subset D(0, r)$. Hence, for all matrices $\| A \| \leq r$ we have that

$$\| e(A) \| \leq \left(1 + \sqrt{2} \right) \sup_{z \in D(0, r)} | e(z) |.$$

5 Efficiency and reproducibility features

The package is written in Julia but provides a wide range of facilities to ensure that users of other languages can benefit from our findings. Therefore, we have implemented functions for saving and loading graphs in a language-independent format and for generating multi-language code directly from a graph object.

5.1 Code generation

The package allows the user to generate fast and memory-efficient code. The code generation feature is also available for Julia, and is in general preferred over the `eval_graph` function for efficiency. The package supports code generation for

- Julia, with in-place addition to reduce the memory footprint,

- MATLAB,

- C with support for OpenBLAS, and

- C with support for the Intel Math Kernel Library (MKL).

On current HPC computer architectures, memory access and allocations have a large impact on performance. In our work, we strove to minimize the memory footprint of the generated code by choosing a topological ordering that (heuristically) tries to minimize the maximum number of matrices that the algorithms needs to store at any given time. Finding the topological ordering that is optimal in the sense of the memory footprint is essentially equivalent to determining the width of the graph, which is a difficult problem in general. For generality, the choice of topological ordering can therefore be influenced by parameters passed to `gen_code`.

Example 13 (MATLAB code generation) *The following code generates the computational graph corresponding to the Paterson–Stockmeyer evaluation of the truncated Taylor series to the cosine. We exploit the fact that the Taylor series is a polynomial in A^2.*

```julia
julia> # c is the non-zero monomial coefficients for the cosine function
julia> c = (kron(ones(5),[1, -1]) ./ factorial.(0:2:19));
julia> (g,_) = graph_ps(c); # Paterson--Stockmeyer evaluation
julia> rename_node!(g, :A, :A2tmp); # ``change'' input to A^2
julia> add_mult!(g, :A2tmp, :A, :A);
julia> gen_code("mycosm.m", g, funname = "mycosm", lang = LangMatlab())
```

This generates the file:

```
function output = mycosm(A)
    n = size(A,1);
    I = eye(n,n);
    % Computation order:
    % A2tmp B_0_1 B_1_1 B_2_1 A2 B_0_2 B_2_2 B_1_2 A3 P2 C1 P1 C0 P0
    % Computing A2tmp with operation: mult
    A2tmp = A * A;
    % Computing B_0_1 with operation: lincomb
    coeff1 = 1.0;
    coeff2 = -0.5;
    B_0_1 = coeff1*I + coeff2*A;
    % Computing B_1_1 with operation: lincomb
    coeff1 = -0.001388888888888889;
    coeff2 = 2.48015873015873e-5;
    B_1_1 = coeff1*I + coeff2*A;
    ...
```

```
        P1 = coeff1 * C1 + coeff2 * B_1_2;
        % Computing C0 with operation: mult
        C0 = P1 * A3;
        % Computing P0 with operation: lincomb
        coeff1 = 1.0;
        coeff2 = 1.0;
        P0 = coeff1 * C0 + coeff2 * B_0_2;
        output = P0;
    end
```

The function `mycosm` *can be called directly from MATLAB:*

```
>> cosm = @(S)(expm(1i*S) + expm(-1i*S)) / 2 % Euler's formula
>> A = randn(100, 100) / 100;
>> norm(cosm(A) - mycosm(A))
ans =
   2.2390e-16
```

Remark 14 (Dot fusion and in-place addition) *In terms of performance, Julia has a competitive advantage for the matrix operations considered in the package. The reason is twofold: on the one hand the extended usage of dots and dot fusion [24] reduces the number of passes through the data required to carry out a computation, while on the other the availability of in-place operations decreases the memory footprint. Our code generation is capable of taking full advantage of these two features, and can gain a considerable performance improvements. In fact, the dot fusion is the reason why the generated Julia code outperforms even C implementations in some of the experiments in Section 7. In particular, when relying on the BLAS one can only compute the sum of two matrices at a time, whereas by using the dot fusion, Julia can add an arbitrary number of matrices with only a single pass through the data. The cosine in the previous example is used as an illustration.*

```
julia> using BenchmarkTools
julia> # Dot fusion is enabled by default
julia> gen_code("mycosm.jl", g, funname = "mycosm", lang = LangJulia())
julia> # It can be disabled with a constructor argument
julia> use_dot_fusion = false;
julia> gen_code("mycosm_no_dot.jl",g,
                funname = "mycosm_no_dot",
                lang = LangJulia(true, true, use_dot_fusion))
julia> include("mycosm.jl");
julia> include("mycosm_no_dot.jl");
julia> A0 = randn(500, 500);
julia> A = copy(A0); @btime mycosm($A);
  20.576 ms (9 allocations: 7.63 MiB)
julia> A = copy(A0); @btime mycosm_no_dot($A);
  27.924 ms (11 allocations: 9.54 MiB)
```

5.2   Storing a graph in a file

Language-independent reproducibility of our results is ensured by the availability of functions for saving and loading `Compgraph` objects. The graphs are stored in a human-readable way, that can easily be interpreted without using Julia.

The functions `export_compgraph` and `import_compgraph` read and write the graph, including optional or generated graph metadata. The saved file is in computational graph (CGR) format, a storage format specifically designed to store computational graphs for matrix functions.

Example 15 (Saving and loading) *We can export the graph generated in Section 5.1 to a file with the command*

```
julia> export_compgraph(g, "cosm_graph.cgr");
```

which produces the file

```
%# Representation of a computation graph
%# Created: 2021-05-25T11:48:01.720 by user jarl

graph_coeff_type="Float64";

A2=A*A;
coeff1=2.8114572543455206e-15;
coeff2=-7.647163731819816e-13;
B_3_2=coeff1*A2+coeff2*I;
coeff1=-0.0001984126984126984;
coeff2=0.008333333333333333;
B_1_2=coeff1*A2+coeff2*I;
A3=A*A2;
coeff1=1.6059043836821613e-10;
...
```

We note that a CGR file is essentially an executable script in both MATLAB and Julia, but we stress that this is not the recommended way to generate code in either language.

6   Error analysis

6.1   Bounds on forward and backward error

Let us consider the problem of approximating the function f at a matrix A when multiple graph-represented rational functions $g_0, \dots, g_\ell$ are available. Without loss of generality, we can assume that these functions are ordered by increasing computational cost, so that evaluating $g_{i+1}(A)$ is computationally more expensive than evaluating $g_i(A)$. In practice, we will typically also have that $\left\| g_{i+1}(A) - f(A) \right\| < \left\| g_i(A) - f(A) \right\|$, but the analysis in this section does not require this.

In order to approximate f efficiently, ideally we would like to have an inexpensive strategy to choose at runtime the smallest i such that $g_i(A)$ is guaranteed to be an acceptable approximation of $f(A)$. One can gauge the quality of the approximation in several ways.

In Section 4, we described how the forward error is an effective tool to optimize the coefficients of a computational graph for a fixed graph topology. In principle, the relative forward error could be used as a means to choose a suitable approximant, as we now explain. Our analysis is based on a result of Al-Mohy and Higham [1, Theorem 4.2(a)], which we recall here as it will be crucial to our discussion. If the function $h : \Omega \subset \mathbb{C} \to \mathbb{C}$ has the series expansion

$$h(z) = \sum_{j=j_0}^{\infty} \beta_j z^j, \tag{15}$$

then for any matrix A such that

$$\alpha_p(A) = \min\left\{ \left\| A^p \right\|^{1/p}, \left\| A^{p+1} \right\|^{1/(p+1)} \right\}, \qquad p(p-1) \le j_0, \tag{16}$$

is within the disc of convergence of the series, one has that

$$\left\| f(A) \right\| \le \sum_{j=j_0}^{\infty} |\beta_j| \alpha_p(A)^j. \tag{17}$$

We note that if the coefficients β_j in (15) are one-signed, then the bound (17) reduces to

$$\left\|f(A)\right\| \leq \left|f(\alpha_p(A))\right|,$$

and also that $\alpha_p(A)$ reduces to the minimum between 1 and $\|A\|$ in the general case $j_0 = 0$.

One can then define, using the series expansion of the forward error, the series

$$E_i(z) = \sum_{j=j_0}^{\infty} |\gamma_j| z^j, \quad \text{where} \quad e_i(z) = g_i(z) - f(z) = \sum_{j=j_0}^{\infty} \gamma_j z^j, \quad j_0 \in \mathbb{N},$$

and then pre-compute the values

$$\theta_i^F = \max\{\theta \in \mathbb{R}^+ : E_i(\theta) \leq u\}, \tag{18}$$

where $\mathbb{R}^+$ and u denote the positive real semi-axis and the unit round-off of the current working precision, respectively. It is easy to see that choosing a g_i such that $\alpha_p(A) \leq \theta_i^F$ guarantees that the absolute forward error $\|e_i(A)\|$ will not be larger than u. This measure of accuracy, however, is not ideal for two reasons. On the one hand, the use of an absolute forward error disregards the magnitude of the matrix function one seeks to compute, but turning to relative errors is unfeasible in most cases, as estimates of $\|f(A)\|$ are rarely available. On the other hand, the dependency of the forward error on the condition number of f evaluated at A, makes this method not completely reliable. For these two reasons, bounds on the forward error are seldom used in algorithms for matrix functions, one exception being arbitrary-precision environments, in which the pre-computation of the θ_i^F in (18) would be impractical as the value of u is known only at runtime [12, 13].

A better alternative, originally proposed by Moler and Van Loan [25] to choose the degree of a diagonal Padé approximant to the exponential for the scaling and squaring algorithm, is based on bounds on the relative backward error. The essence of this approach is to consider the matrix E such that $g_i(A) = f(A + E)$ and then pick an approximant g_i for which $\|E\| \leq u\|A\|$. In order to do that, we can rely on the series expansion

$$F_i(z) = \sum_{j=j_0}^{\infty} |\delta_j| z^{j-1}, \quad \text{where} \quad \varphi_i(z) = \sum_{j=j_0}^{\infty} \delta_j z^j, \tag{19}$$

and the absolute backward error $\varphi_i(z)$ is defined by $g_i(z) = f(z + \varphi_i(z))$. If, when designing the algorithm, we compute the values

$$\theta_i^B = \max\{\theta \in \mathbb{R}^+ : F_i(\theta) \leq u\}, \tag{20}$$

then the approximant g_i evaluated at a matrix $A \in \mathbb{C}^{n \times n}$ such that $\alpha_p(A) \leq \theta_i^B$ will guarantee that

$$\frac{\|\varphi_i(A)\|}{\|A\|} \leq u,$$

which is entirely satisfactory.

In our package we provide the function `compute_bwd_theta` to compute the values θ_i^B in (20) for the exponential. The function exploits the identity

$$\varphi_i(z) = \log\left((e^{-z} g_i(z) - 1) + 1\right), \tag{21}$$

and uses as F_i the truncated series expansion of the expression on the right-hand side of (21). Truncated series expansion are manipulated efficiently as polynomials by relying on the functions available in the `Polynomials.jl` package. The computation of θ_i^F is then performed using the `fzero` function in arbitrary precision.

Example 16 (Backward error function) *We illustrate the usage of* `compute_bwd_theta` *by continuing Example 10. Because of how Julia handles types and how the* `Compgraph` *data structure is implemented, we can perform a symbolic evaluation of a graph to produce a* `Polynomial`.

```julia
julia> using Polynomials;
julia> x = Polynomial("x");
julia> p = eval_graph(graphb, x)
Polynomial(1.0000000000000000134967838007550967408645631852098 +
  1.0000000000000000432055652330969008190946510481787*x +
  0.5000000000000000103349460283223171196305847125560 8*x^2 +
  0.1666666666666666690001594704661822786539021333344 7*x^3 +
  0.0416666666666666718675231526549199588810357629724 03*x^4 +
  0.0083333333333333344895439682913026016254992768036 399*x^5 +
  0.0013888888888891458296210060853258251279242402015 1*x^6 +
  0.0001984126984132692106549464048193455870982386575 5*x^7 +
  0.0000248015873028489514898850556276256098485349487 6*x^8 +
  0.0000027557319249913788491117800193859147036532254 3*x^9 +
  0.0000002755731927465529049358714410326705724968219 6*x^10 +
  0.0000000250520118494317795224594544307721129338594 8*x^11 +
  0.0000000020862833319476309818454030583261187745075 5*x^12 +
  0.0000000001512743632699847697461004031636947142521 4*x^13 +
  0.0000000000084580151235493889286996942807948759610 6*x^14 +
  0.0000000000003032441896168963980874961240221155241 1*x^15 +
  0.0000000000000515212112436523524706264550043349319*x^16)
```

These coefficients are close to those of the Taylor series expansion, but not identical, neither in arbitrary (`BigFloat`*) nor in binary64* (`Float64`) *arithmetic.*

```julia
julia> a = p.coeffs;
julia> a_taylor = 1 ./ factorial.(BigFloat.(0:16));
julia> norm(abs.(a-a_taylor))
9.900938779409988e-12
julia> a_float64 = convert(Array{Float64}, a);
julia> a_taylor_float64 = convert(Array{Float64}, a_taylor);
julia> norm(abs.(a_float64 - a_taylor_float64))
9.900938779407373e-12
```

We can approximate the bound θ_i^B in (18) by using the function `compute_bwd_theta`*. For illustration we now show how this can be computed directly by using the first 100 coefficients of the series expansion in (19) and* `BigFloat` *arithmetic with 1000 significant binary digits.*

```julia
julia> setprecision(1000); # Set number of bits in fraction of BigFloat.
julia> nterms = 100; # Order of highest term in the truncated series expansions.
julia> p = convert(Polynomial{Float64, :x}, p); # Use Float64 polynomial.
julia> # Evaluate truncated series expansion of right-hand side of (21).
julia> coeffs_exp = (-1).^(0:1:nterms) ./
                    factorial.(collect(big(0.):big(1.):nterms)); # exp(-z)
julia> expminusz = Polynomial(coeffs_exp);
julia> R = expminusz * p - 1; # R_i (z) = exp(-z) p(z) - 1
julia> coeffs_log = [0; (-1) .^ (0:1:nterms-1) ./ (1:1:nterms)]; # log(x+1)
julia> logzplusone = Polynomial(coeffs_log);
julia> F = logzplusone(R); # log(R_i +1).
julia> bnd_bwd_err = Polynomial(abs.(F.coeffs)) # Absolute value of coefficients
julia> # Find point where bound on relative backward error equals tolerance.
julia> using Roots
julia> bnd_rel_bwd_err(z) = abs.(bnd_bwd_err(z)) ./ abs.(z)
julia> theta_bwd = fzero(z->bnd_rel_bwd_err(z) - eps(Float64)/2, big"0.2")
0.371875987551502714247882036196069641950651085 99084
```

The function `bnd_rel_bwd_err` *represents a bound on the relative backward error of the function underlying* `graphb` *seen as an approximant to the exponential. The plot of the function* `bnd_rel_bwd_err` *is*
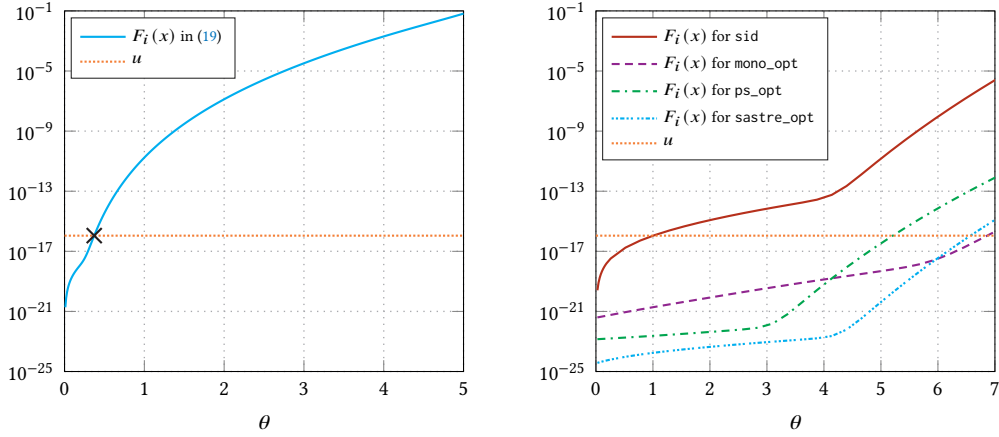
(a) Paterson–Stockmeyer evaluation for $m = 5$.

(b) Approximants in Table 6 for $m = 7$ and $r = 6.4$.

Figure 4: Series expansion (19) for various approximants to the exponential. The dashed orange line represents the unit round-off of binary64 arithmetic $u = 2^{-53} \approx 1.11 \times 10^{-16}$. The cross marks the point $\left(\theta_i^B, F_i(\theta_i^B)\right)$, where θ_i^B is defined in (20).

given in Figure 4a, where we also specified the corresponding θ_i^B. For this graph, the backward error will be smaller than machine precision for any complex scalar of modulus at most $\theta_i^B \approx 0.37$.

6.2 Round-off error

The optimization process in Section 4 and the backward error analysis are based on exact arithmetic and rely on high-precision arithmetic for practical purposes. The algorithms, however, are designed to be used in floating-point arithmetic, where rounding errors may become more prominent. The generality of the data structures in our package allows us to obtain more realistic estimates and bounds on the round-off errors that may occur during the evaluation of $g(z)$. We will rely on an a posteriori error analysis.

In particular, we will rely on a running error analysis, in which the rounding errors are bounded automatically by the algorithm as $g(z)$ is evaluated. Our goal will be to assign to each node $i \in V$ a value δ_i that represents the round-off error occurred in the evaluation of the subgraph rooted at node i. In the remainder of this section a hat denotes computed quantities, thus we will employ the notation $\widehat{Z}_i = Z_i(1 + \delta_i)$, where δ_i represents the relative round-off error [19, Theorem 2.2]. For conciseness, we will also denote the indices of the two parents of node Z_i by $\ell = e_i^{(1)}$ and $r = e_i^{(2)}$, analogous to (1).

Note that the only two ingredients the running error analysis on the graph requires are the value of δ_i for the leaf nodes, that is, the input nodes of the graph, and a rule to compute δ_i from Z_ℓ, Z_r, δ_ℓ, and δ_r. If i is an input node, then we can set $\delta_i = 0$, which is equivalent to saying that the input values 1 and z are represented exactly. If i is not an input node, we need to consider three cases.

- If $p_i = *$, then using the standard model of floating-point arithmetic [19, Equation (2.5)] we have that

$$\widehat{Z}_i(1 + \eta) = \widehat{Z}_\ell \widehat{Z}_r, \qquad |\eta| \leq u,$$

By using the fact that $Z_i = Z_\ell Z_r$, we can solve for $\widehat{Z}_i$ and obtain

$$\widehat{Z}_i = Z_i \frac{(1 + \delta_\ell)(1 + \delta_r)}{1 + \eta} \tag{22}$$

23

and therefore

$$\delta_i = \frac{(1 + \delta_\ell)(1 + \delta_r)}{1 + \eta} - 1 = \frac{\delta_\ell + \delta_r + \delta_r \delta_r - \eta}{1 + \eta} \approx \delta_\ell + \delta_r - \eta.$$

- Similarly, $p_i = \backslash$, we have $Z_i = Z_\ell^{-1} Z_r$ and

$$\widehat{Z}_i = Z_i \frac{1 + \delta_r}{(1 + \delta_\ell)(1 + \eta)}, \qquad |\eta| \leq u,$$

which implies that

$$\delta_i = \frac{\delta_r - \delta_\ell - \eta - \delta_\ell \eta}{(1 + \delta_\ell)(1 + \delta_r)} \approx \delta_r - \delta_\ell - \eta. \tag{23}$$

- Finally, if $p_i = +$, we have

$$\widehat{Z}_i = \frac{\alpha \widehat{Z}_\ell + \beta \widehat{Z}_r}{1 + \eta}, \qquad |\eta| \leq u.$$

We solve this equation for $\widehat{Z}_i$ and obtain

$$\widehat{Z}_i = Z_i \frac{\alpha Z_\ell (1 + \delta_\ell) + \beta Z_r (1 + \delta_r)}{(1 + \eta) Z_i}$$

from which we find the relative error

$$\delta_i = \frac{\delta_\ell \alpha Z_\ell + \delta_r \beta Z_r - \eta Z_i}{(1 + \eta) Z_i} \approx \frac{\delta_\ell Z_\ell + \delta_r Z_r}{Z_i} - \eta. \tag{24}$$

The formulae for the relative error in (22), (23), and (24) are exact, but they are also impractical:

1. The quantity η is not necessarily representable in the current working precision, is not readily available, and is unduly expensive to estimate.

2. The expressions for linear combinations feature exact quantities, which are not available during a genuine run of algorithm.

Point (2) can in practice handled in our setting by carrying out the analysis in high precision arithmetic, in order to estimate the error for standard rounding error arithmetic.

One way to make these formulae practical is to turn them into running error bounds. In the approximations on the right-hand side of (22), (23), and (24) we have linearized the expressions by neglecting terms of the form $\delta_r^i \delta_\ell^j \eta^k$ for $i + j + k \geq 2$. Exact quantities can be replaced by their computed counterparts. This substitution will not have a major effect, since the magnitude of δ_i should be negligible compared with that of Z_i. Then we can easily obtain running bounds on the relative error by replacing the "exact" errors δ_i in the formulae above with the approximate bounds

$$\Delta_i = \begin{cases} \dfrac{|\alpha||Z_\ell|\Delta_\ell + |\beta||Z_r|\Delta_r}{|\widehat{Z}_i|} + u, & p_i = +, \\[3mm] \Delta_\ell + \Delta_r + y, & p_i \in \{*, \backslash\}, \end{cases} \tag{25}$$

for which $\delta_i \lesssim \Delta_i$. It is important to stress that Δ_i may potentially not be a true bound because of the linearization step.

In practice, the formula (25) can be used to compute the running error (estimates) in all elements of the graph by traversing the graph in topological order, analogously to the computation of the Jacobian in (14). This is implemented in the function eval_runerr.

The first-order running error bound (25) is tighter than the worst-case error bounds produced by an a priori first-order error analysis, but can still be rather pessimistic for large graphs. An alternative is to use the linearized version of (24), (22), and (23) to estimate the value of δ_i. A stochastic estimate can be obtained, for instance, by interpreting η as a random variable uniformly distributed over the interval $[-u, u]$.

Example 17 (Numerical cancellation) *We illustrate the benefits of running error bounds on a classical example by Goldberg [15]: the evaluation of $y^2 - z^2$ for $y \approx z$, which is prone to cancellation if the two squares are not exactly representable in floating-point arithmetic. The following graph evaluates the scalar function $(1 + 2^{-27})^2 - (1 + 2^{-28})^2$*

```
julia> x = 2. .^ -27
# Construct graph for (1+x)^2 - (1+x/2)^2.
julia> graph = Compgraph(Float64);
julia> add_lincomb!(graph, :y, 1, :I, 1, :x)
julia> add_lincomb!(graph, :z, 1, :I, 1/2, :x)
julia> add_mult!(graph, :y2, :y, :y)
julia> add_mult!(graph, :z2, :z, :z)
julia> add_lincomb!(graph, :out, 1, :y2, -1, :z2)
julia> add_output!(graph, :out)
```

The relative forward error and the corresponding running error bound are of order 10^{-9} and 10^{-7}, respectively. A more accurate probabilistic running bound can also be computed.

```
# Compare forward error and running error bound.
julia> exact = 2. ^ (-2. * (i+1)) * (2. ^(i+2) + 3)
7.450580638557192 e-9
julia> computed = eval_graph(graph, x, input = :x)
7.450580596923828 e-9
julia> error = abs(exact - computed) / abs(exact)
5.5879354164678485 e-9
julia> running_error_bound = eval_runerr(graph, x, input = :x)
2.9802322765171140955271284691188744684400573666665064 e-07
# A more accurate bound estimate can be found with the random mode
julia> running_error_bound = eval_runerr(graph, x, input = :x, mode=:rand)
8.5837803426741329753118953368246018650457992705813 e-09
```

7 Applications to state-of-the-art algorithms

We now show how the graph representation and the optimization described in previous sections can be used to improve the accuracy and performance of state-of-the-art algorithms. As an illustration, we consider the computation of exponential and square root.

Matrix exponential The exponential is particularly well suited to our framework, as we can build on recent developments in polynomial methods [31, 32, 30, 2]. In this context the optimization can be seen as the second stage in a three-stage algorithm design procedure:

S1. Construct a graph using Paterson–Stockmeyer, monomials, or any method from [31, 32, 30, 2].

S2. Optimize the coefficients in the graph with the technique in Section 4 run in high precision.

S3. Compute guaranteed backward error bounds with the method discussed in Section 6.1.

For the function at hand, this approach brings an improvement over state-of-the-art polynomial methods, which in several cases already outperform the Padé-based methods typically implemented in mathematical software packages.

Table 5: Relative errors achieved after optimization within different disks with radius r for m multiplications. Optimization is carried out for the disk of radius r. The names of the graphs correspond to the graph files in the data repository. Underlined numbers correspond to optimizations initiated with squaring of the graph with one less multiplication. The dagger † indicates that the graph in the corresponding columns were obtained using as starting graph for the optimization the graph in the previous column.

Graph	$m = 4$ $r = 0.69$	$m = 5$ $r = 1.68$	$r = 1.9^\dagger$	$m = 6$ $r = 2.22$	$r = 2.7^\dagger$	$m = 7$ $r = 3.59$	$r = 6.4^\dagger$	$m = 8$ $\underline{r = 13.5}$
Graphs found by optimization								
mono_opt	$1.2 \cdot 10^{-15}$	$4.6 \cdot 10^{-13}$	$1.5 \cdot 10^{-12}$	$1.8 \cdot 10^{-16}$	$1.8 \cdot 10^{-17}$	$4.2 \cdot 10^{-18}$	$\mathbf{7.2 \cdot 10^{-17}}$	$\mathbf{3.7 \cdot 10^{-17}}$
ps_opt	$1.2 \cdot 10^{-15}$	$1.8 \cdot 10^{-13}$	$2.0 \cdot 10^{-12}$	$5.6 \cdot 10^{-14}$	$3.3 \cdot 10^{-15}$	$2.5 \cdot 10^{-20}$	$3.3 \cdot 10^{-13}$	$5.0 \cdot 10^{-14}$
sastre_opt	$1.4 \cdot 10^{-15}$	$1.8 \cdot 10^{-15}$	$\mathbf{3.7 \cdot 10^{-17}}$	$1.5 \cdot 10^{-16}$	$6.0 \cdot 10^{-17}$	$\mathbf{1.3 \cdot 10^{-21}}$	$2.6 \cdot 10^{-16}$	$1.9 \cdot 10^{-16}$
bbc_opt	$2.3 \cdot 10^{-14}$	$4.8 \cdot 10^{-14}$	$1.3 \cdot 10^{-13}$	$\times$	$\times$	$\times$	$\times$	$\times$
sid_opt	$1.1 \cdot 10^{-16}$	$\mathbf{1.8 \cdot 10^{-16}}$	$1.1 \cdot 10^{-15}$	$\mathbf{5.3 \cdot 10^{-18}}$	$\mathbf{1.6 \cdot 10^{-18}}$	$2.7 \cdot 10^{-16}$	$\times$	$\times$
Polynomial methods from the literature								
sastre	$1.5 \cdot 10^{-11}$	$\underline{5.8 \cdot 10^{-14}}$	$4.0 \cdot 10^{-13}$	$1.8 \cdot 10^{-8}$	$7.8 \cdot 10^{-7}$	$\underline{6.4 \cdot 10^{-10}}$	$\times$	$\times$
bbc	$2.5 \cdot 10^{-12}$	$7.8 \cdot 10^{-13}$	$9.9 \cdot 10^{-12}$	$\times$	$\times$	$\times$	$\times$	$\times$
sid	$\mathbf{2.6 \cdot 10^{-16}}$	$1.3 \cdot 10^{-15}$	$5.1 \cdot 10^{-15}$	$3.4 \cdot 10^{-15}$	$6.0 \cdot 10^{-14}$	$5.2 \cdot 10^{-14}$	$5.4 \cdot 10^{-8}$	$\times$

Implementations are named after the evaluation method used (mono for linear combination of monomials and ps for the Paterson–Stockmeyer scheme) or after the authors of the paper where the algorithm is first presented: sastre [30], sid [31], and bbc [2]. The suffix _opt denotes improved graphs obtained by using our optimization strategy. The same naming is used is the data repository, where the optimized graphs are available in CGR format together with the corresponding code for Julia, MATLAB, and C generated as described in Section 5.1.

Table 5 presents the results of Stage S2. The table contains the scalar relative errors on the disks of radius r for various graphs. For comparison we have included the results for the optimized algorithms (first six rows) as well as for some polynomial algorithms from the literature. For the graphs found by optimization, the symbol $\times$ denotes combinations for which the optimization routine gave unreasonably bad results; for the graphs from the literature it denotes cases for which the coefficients of the polynomial methods were not available.

In terms of relative error, the algorithm in [31] cannot be improved for $m = 4$. This is not surprising: the method is almost optimal in the sense that it matches 15 Taylor coefficients out of the optimal 16 (cf. Table 4). For $m > 4$ multiplications, however, our approach bring a substantial improvement.

All the graphs were represented in degree-optimal form (Section 3) and all free coefficients in that form were used as free variables in the optimization. For example, the rows marked mono_opt, ps_opt, and sid_opt correspond to the degree-optimal forms in Figures 3(a), 3(c), and 3(d), respectively.

The results in this section are of empirical nature. The optimization problem has many local minima, and a slight perturbation of the initial parameters can lead the optimization to converge to a different point of minimum. One consequence is the somewhat unexpected result in the third row for $m = 5$, where optimizing over a larger disk yields a smaller relative error. We were lucky. These experimental results are of scientific value since both the graphs and coefficients are publicly available.

Remark 18 (Technical details for optimization) *Even though for $m \leq 5$ optimizing in double precision led to reasonably good results, in most cases we had to resort to higher precision. This was done using Julia's* BigFloat *data type, which in turn relies on the GNU MPFR library [14]. Even when using*

Table 6: Values of θ^B in (18) for the guaranteed backward error bound computed with the method described in Section 6.1 for m multiplications (or $m-1$ multiplications and one inverse multiplication). The values marked with $*$ are those tabulated in the corresponding papers [31] and [2, Table 2]. A large value of θ^B indicates that the method is applicable in a wider domain. Boldface marks the best method for a given number of multiplications.

Graph	$m = 4$ $r = 0.69$	$m = 5$ $r = 1.68$	$r = 1.9$	$m = 6$ $r = 2.22$	$r = 2.7$	$m = 7$ $r = 3.59$	$r = 6.4$	$m = 8$ 13.5
Graphs found from optimization								
mono_opt	0.570	1.043	1.100	2.259	3.033	4.209	**6.865**	**14.707**
ps_opt	0.569	1.142	1.134	1.713	2.431	5.064	5.210	10.768
sastre_opt	0.565	1.478	**2.054**	2.247	2.868	5.832	6.572	13.884
bbc_opt	0.450	1.256	1.377	×	×	×	×	×
sid_opt	0.674	1.683	1.766	2.581	**3.313**	3.633	×	×
Methods from literature								
sid*	**0.695**	1.683	1.683	2.219	2.219	3.53	3.53	×
bbc*	0.299	1.09	1.09	2.22	2.22	×	×	×
exp_native*	0.25	0.95	0.95	2.10	2.10	5.4	5.4	10.8

higher precision, the complicated singular structure of the linear systems occurring during the optimization led to either stagnation or divergence, regardless of whether the backslash operator (based on the QR factorization) or the normal equations were used. We opted for a pseudoinverse (with adapted drop tolerance) computed using the high precision SVD routine in the Julia GenericSVD *package. The optimization routine requires a damping, which has to be adapted throughout the iteration. We were unable to find a heuristic damping strategy suitable for all cases, thus we used scripting to decide and control damping, drop tolerances, and other parameters. In order to break the symmetry in the graph, we also sometimes added a random perturbation. The sequence of parameters that led to the results below are given in the companion data package.*

These problems stem from the singularity of the Jacobian, a consequence of the redundancy of the coefficients in the degree-optimal form. Removing obvious redundancies, for instance by fixing the first row of H_a and H_b, did not improve the result, and completely characterizing all such redundancies in the degree-optimal form is a non-trivial issue beyond the scope of our package.

Table 6 reports the results of step S3, that is, the application of the technique in Section 6.1 to the graphs in Table 5. More precisely, the values correspond to the radii of the disks that produced a backward error smaller than the unit round-off u. The algorithm underlying the graph is suitable for any matrix with norm smaller than the value in the corresponding cell of the table. Note that we optimize over one disk, and obtain a backward error guarantee for another disk which may be smaller or larger. Once again the optimization brings substantial improvements over the algorithms used as initial guesses. The best method is mostly consistent with the best method in Table 5, with the only exception of the case $m = 5$. The values in this table are found as in Figure 4(b).

The table also reports the values of θ^B in (20) for the Julia implementation of the Padé-based scaling-and-squaring algorithm (exp_native), which is based on the algorithm by Higham [20]. This method always requires one multiplication by the inverse of a matrix. In the table we assume that matrix multiplication and multiplication by the inverse have same computational cost, to the disadvantage of the methods we propose.

The practical performance is illustrated by the CPU timings in Table 7. The label *Our approach* refers to the corresponding graphs in Table 6, i.e., for $\|A\| = 2.5$ we use sid_opt, and for $\|A\| = 6.0$ and $\|A\| = 13.5$ we use mono_opt. For reference we also report the CPU timings of the built-in functions for computing the matrix exponential in Julia (exp) and MATLAB (expm), and for the implementation of sid available on the website of one of the authors.[5] Among these *out-of-the-box* methods, the polynomial method in sid is the fastest. The column C reports the asymptotic computational cost normalized so that one $n \times n$ matrix multiplication has cost $C = 1$ and the solution of a linear system with n right-hand sides has cost $4/3$.

In order to gauge the benefit of code generation, we also included exp_native_jl, the code generated from a graph implementation of the built-in Julia exp function. By comparing the third row in Table 7 with the last four, we see that the code generation is competitive, being even faster than the (equivalent) native Julia implementation.

Remark 19 (Technical details for comparisons of CPU timings) *The timings were obtained following the best practices for benchmarking dense matrix computations. The processor boost was disabled in order to improve reproducibility. The reported values are median values of several runs. The experiments were run on a machine equipped with an 8-core Intel(R) Core(TM) i7-8550U CPU running at 1.80GHz. We used the development version 1.7.0 of Julia, as improvements made by the authors to the built-in Julia exp function are only available in the development branch at the time of writing. The MKL implementation (version 2020.0.166) of BLAS was used for the simulations in C.*

The matrices were made as large as possible as long as the algorithms would not run out of memory. We note that the approach based on the degree-optimal form requires more memory than the other methods, as the number of intermediate matrices that have to be stored grows linearly with the number multiplications. We also remark that larger matrices offer an advantage to the graph-based methods, which have a lower computational cost C. In some languages we could make the matrices larger. In Julia with MKL, for instance, we could run the experiment for $n = 4000$, obtaining for $\|A_1\| = 13.5$ a timing of for our approach of 11.8s, compared with 13.6s for exp and 13.1s for exp_native_jl.

Square root   The approach, both in terms of optimization and graph representation, can be applied to any analytic matrix function. We illustrate this with the function $f(x) = \sqrt{x+1}$ in the domain $\frac{1}{2} \cdot D$, for some disk D of radius r. As this function is equivalent to the square root $\sqrt{x}$ in the shifted disk $1 + \frac{1}{2} \cdot D$, we can use the Denman–Beavers iteration [10] to warm start the optimization.

```
julia> f=s->sqrt(s+1)
julia> (graph,cref)=graph_denman_beavers(2); # Use two iterations
julia> rename_node!(graph,:A,:A_shift,cref); # Introduce a shift
julia> add_lincomb!(graph,:A_shift,1.0,:A,1.0,:I);
julia> abs(eval_graph(graph,0.4im)-f(0.4im))
2.6944276809715006e-8
```

Directly optimizing the Denman–Beavers coefficients did not bring any improvement. However, the Denman–Beavers iteration can be written in a form reminiscent of the degree-optimal form, where the outer multiplication is replaced by a left division:

$$B_1 = (x_{a,1,1}I + x_{a,1,2}A)^{-1}(x_{b,1,1}I + x_{b,1,2}A) \tag{26a}$$

$$B_2 = (x_{a,2,1}I + x_{a,2,2}A + x_{a,2,3}B_1)^{-1}(x_{b,2,1}I + x_{b,2,2}A + x_{b,2,3}B_1) \tag{26b}$$

$$B_3 = (x_{a,3,1}I + x_{a,3,2}A + x_{a,3,3}B_1 + x_{a,3,4}B_2)^{-1}(x_{b,3,1}I + x_{b,3,2}A + x_{b,3,3}B_1 + x_{b,3,4}B_2) \tag{26c}$$

$$B_4 = (x_{a,4,1}I + x_{a,4,2}A + \cdots + x_{a,4,5}B_3)^{-1}(x_{b,4,1}I + x_{b,4,2}A + \cdots + x_{b,4,5}B_3) \tag{26d}$$

$$r(A) = y_1I + y_2A + y_3B_1 + \cdots + y_6B_4. \tag{26e}$$

---

[5] http://personales.upv.es/~jorsasma/software/expmpol.m

Table 7: CPU timings and normalized computational cost C for matrices of size $n = 3000$. Method that are conclusively faster are in boldface.

		Out-of-the-box					
		$\|A\| = 2.5$		$\|A\| = 6.0$		$\|A\| = 13.5$	
Implementation	Language	CPU-time	C	CPU-time	C	CPU-time	C
expm	MATLAB/MKL	4.71	$7\frac{1}{3}$	5.36	$8\frac{1}{3}$	5.93	$9\frac{1}{3}$
exp	Julia/OpeBLAS	5.62	$7\frac{1}{3}$	6.35	$8\frac{1}{3}$	6.94	$9\frac{1}{3}$
exp	Julia/MKL	4.81	$7\frac{1}{3}$	5.45	$8\frac{1}{3}$	6.03	$9\frac{1}{3}$
expmpol [31]	MATLAB/MKL	**4.47**	7	5.46	8	6.04	9
		Generated code					
		$\|A\|_1 = 2.5$		$\|A\|_1 = 6.0$		$\|A\|_1 = 13.5$	
Graph	Language	CPU-time	C	CPU-time	C	CPU-time	C
Our approach	MATLAB/MKL	4.34	6	5.20	7	6.06	8
Our approach	Julia/OpenBLAS	4.62	6	5.54	7	6.36	8
Our approach	Julia/MKL	**3.83**	6	**4.58**	7	**5.27**	8
Our approach	C/MKL	3.94	6	4.74	7	5.54	8
exp_native_jl	MATLAB/MKL	4.60	$7\frac{1}{3}$	5.18	$8\frac{1}{3}$	**5.73**	$9\frac{1}{3}$
exp_native_jl	Julia/OpenBLAS	5.07	$7\frac{1}{3}$	5.68	$8\frac{1}{3}$	6.31	$9\frac{1}{3}$
exp_native_jl	Julia/MKL	4.30	$7\frac{1}{3}$	**4.85**	$8\frac{1}{3}$	5.36	$9\frac{1}{3}$
exp_native_jl	C/MKL	4.41	$7\frac{1}{3}$	**4.95**	$8\frac{1}{3}$	5.47	$9\frac{1}{3}$

In the notation of the Denman–Beavers example in Figure 1(a) we have, if the shift is included, $B_1 = X_0^{-1} = (A + I)^{-1}$, $B_2 = Y_1^{-1} = (\frac{1}{2}I + \frac{1}{2}B_1)^{-1}$, $B_3 = X_1^{-1} = (I + \frac{1}{2}A)^{-1}$, and $B_4 = Y_2^{-1} = (\frac{1}{2}(\frac{1}{2}(I + B_1) + B_3))^{-1}$, and we can express the coefficients in the modified degree-optimal form in Figure 5(a). This formulation provides more degrees of freedom that can be used to fit to the function. We can use our optimization scheme, which is designed to work on arbitrary graphs, on these coefficients in order to obtain a better approximation. This produces the improved approximation in Figure 5(b), where the largest coefficients are normalized so that all elements in H_a and H_b have absolute value below 1.

The error of the approximation is shown in Figure 6. We note that the optimized coefficients lead to an approximation error of the order of the unit round-off u in the target domain. For comparison we also report the approximation error of the Paterson–Stockmeyer evaluation of the degree-5 Taylor approximant as well as the degree-optimal polynomial with coefficients optimized using the Paterson–Stockmeyer method to warm start the optimization. The scheme (26) clearly provides a much more accurate approximation. For fairness, the number of multiplications and multiplications by an inverse are chosen so that the methods have the same asymptotic computational cost C: the rational approximations in Figure 6 requires 4 multiplications by an inverse, whereas the polynomial approximations require 5 matrix-matrix multiplications.

8 Conclusions and outlook

We developed a Julia package for designing algorithms for matrix functions by working with computational graphs. The software offers tools to generate and manipulate such graphs, to optimize their coefficients, and to produce code to evaluate them efficiently in several programming languages.

$$
\begin{bmatrix} H_a \mid H_b \end{bmatrix} =
\left[
\begin{array}{ccccc|ccccc}
1 & 1 & & & & 1 & 0 & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & & 1 & 0 & 0 & & \\
1 & \frac{1}{2} & 0 & 0 & & 1 & 0 & 0 & 0 & \\
\frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{2} & 1 & 0 & 0 & 0 & 0
\end{array}
\right]
$$

$$
y = \begin{bmatrix} \frac{1}{4} & \frac{1}{8} & 0 & \frac{1}{4} & 0 & \frac{1}{2} \end{bmatrix}
$$

(a) Coefficients for Denman–Beavers with shift.

$$
H_a =
\begin{bmatrix}
1.0 & 0.7610413081657074 & 0.0 & 0.0 & 0.0 \\
1.0 & -0.07127247848266706 & 0.9541397430184381 & 0.0 & 0.0 \\
1.0 & 0.5152503668289159 & -0.10150239642416546 & 0.016950835467818528 & 0.0 \\
0.494421282325538585 & 0.05943417585870563 & 0.5017024742690791 & 0.5276435956563207 & 1.0
\end{bmatrix}
$$

$$
H_b =
\begin{bmatrix}
1.0 & -0.011141376535527443 & 0.0 & 0.0 & 0.0 \\
1.0 & -0.02575361014723645 & -0.004282806676350282 & 0.0 & 0.0 \\
1.0 & 0.03845623901890617 & 0.026744368550352622 & -0.029043605652931636 & 0.0 \\
1.0 & 0.12762852235945304 & -0.0015101375211018255 & -0.14486471812413823 & 0.01583234295751802
\end{bmatrix}
$$

$$
y^T =
\begin{bmatrix}
0.28457285753903816 \\
0.11495283295141033 \\
-0.0344930890633602 \\
0.34606871085737384 \\
-0.006442438999085179 \\
1.4670072325444539
\end{bmatrix}
$$

(b) Optimized coefficients.

Figure 5: Coefficients for the scheme (26) as approximation of $f(x) = \sqrt{x+1}$.

Our numerical experiments show that the approach we proposed to design algorithms can improve state-of-the-art algorithms for evaluating the exponential and the square root of a matrix.

Our approach can be extended in several directions. The fixed-topology optimization is a general technique that can be applied to any matrix function, and can potentially be used to design computational methods for matrix functions for which efficient algorithms are currently not available. Moreover, the 3-step technique discussed in Section 7 can be extended to all those functions that can be evaluated efficiently using the scaling-and-squaring technique.

The degree-optimal polynomial form leads to a complete—though not minimal—parameterization of the space of polynomials that can be evaluated with a fixed number of non-scalar multiplications. A minimal complete parameterization would be of interest as it would likely address the issues we experienced in the numerical simulations with singular Jacobians. From a more theoretical point of view, establishing sufficient and practical conditions for a polynomial of degree at most 2^m to belong to the set of optimal-degree polynomial that can be evaluated with m matrix multiplications remains an open problem.

In Section 7 we presented a modified degree-optimal form, suitable for rational matrix functions, which proved to be even more efficient at approximating non-entire functions. It may be worth investigating other variations of this already general form, such as, for example, sparse degree-optimal polynomials, where some coefficients are set to zero and not considered in the optimization. The main advantage of these variants would be a reduction of the memory footprint of the generated code.

By construction, algorithms obtained by using only O1–O3 evaluate rational functions. Our setting, however, is rather different from that of rational approximation theory, where the coefficients of a rational function of fixed degree are optimized. The rational functions we consider have an additional constraint: a fixed topology or equivalently a *fixed number of multiplications and divisions,*

(a) Denman–Beavers (4 steps).

(b) Scheme (26) with optimized coefficients.

(c) Taylor approximant (degree 5).
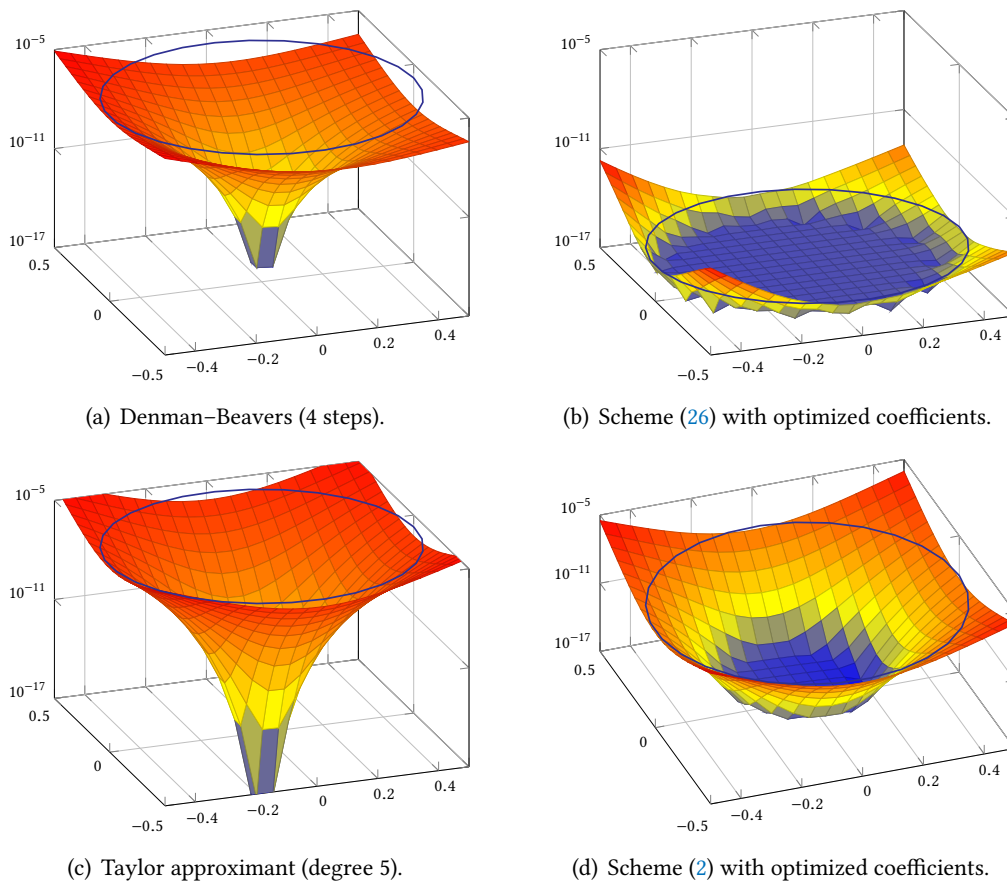

(d) Scheme (2) with optimized coefficients.

Figure 6: Relative forward error of four schemes for approximating the function $f(x) = \sqrt{x+1}$. The plots on the left correspond to 4 steps of the Denman–Beavers iteration (top) and to the Paterson–Stockmeyer evaluation of the truncated Taylor polynomial of degree $m = 5$ (bottom). The plots on the right correspond to graphs with optimized coefficients; the computational graphs on the left in the same row were used as warm start for the optimization. The circles have radius $1/2$ and z-value equal to the maximum relative error within the disk, which is $1.5 \cdot 10^{-6}$, $1.4 \cdot 10^{-15}$, $1.9 \cdot 10^{-6}$ and $3.1 \cdot 10^{-9}$ for (a), (b), (c), and (d), respectively.

depending on the point of view. Combining the effective methods for rational approximation with (say) the modified degree-optimal form in (26) could potentially lead to an improved workflow.

Although some tools for round-off error analysis are provided, further research would be required to understand which methods for the matrix exponential can guarantee a small round-off error. The design and evaluation of graphs in Section 7 is based on high-precision arithmetic, but algorithms are typically run in binary32 or binary64 arithmetic. Therefore, despite being very rare in the literature, round-off error analysis of algorithms for matrix functions is of importance in practice.

References

[1] A. H. Al-Mohy and N. J. Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 31(3):970–989, 2009.

[2] P. Bader, S. Blanes, and F. Casas. Computing the matrix exponential with an optimized Taylor polynomial approximation. *Mathematics*, 7(12):1174, 2019.

[3] P. Bader, S. Blanes, F. Casas, and M. Seydaoğlu. An efficient algorithm to compute the exponential of skew-Hermitian matrices for the time integration of the Schödinger equation. arXiv:2103.10132 [math.NA], 2021.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.

[5] Å. Björck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, January 1996. xvii+407 pp. ISBN 9781611971484.

[6] Å. Björck and S. Hammarling. A Schur method for the square root of a matrix. *Linear Algebra Appl.*, 52/53:127–140, 1983.

[7] H. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998, page 574–583.

[8] H. L. Bodlaender. A tourist guide through Treewidth. *Acta Cybernet.*, 11:1–23, 1993.

[9] M. Crouzeix and C. Palencia. The numerical range is a $(1+\sqrt{2})$-spectral set. *SIAM J. Matrix Anal. Appl.*, 38(2):649–655, 2017.

[10] E. D. Denman and A. N. Beavers Jr. The matrix sign function and computations in systems. *Appl. Math. Comput.*, 2(1):63–94, 1976.

[11] M. Fasi. Optimality of the Paterson–Stockmeyer method for evaluating matrix polynomials and rational matrix functions. *Linear Algebra Appl.*, 574:182–200, 2019.

[12] M. Fasi and N. J. Higham. Multiprecision algorithms for computing the matrix logarithm. *SIAM J. Matrix Anal. Appl.*, 39(1):472–491, 2018.

[13] M. Fasi and N. J. Higham. An arbitrary precision scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 40(4):1233–1256, 2019.

[14] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software*, 33(2):13:1–13:15, 2007.

[15] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comp. Surv.*, 23(1):5–48, 1991.

[16] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th edition, Johns Hopkins University Press, Baltimore, MD, USA, 2013. ISBN 1-4214-0794-9.

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, Cambridge, MA, USA, 2016.

[18] N. J. Higham. Computing real square roots of a real matrix. *Linear Algebra Appl.*, 88/89:405–430, 1987.

[19] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0.

[20] N. J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.*, 26(4):1179–1193, 2005.

[21] N. J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. xx+425 pp. ISBN 978-0-898716-46-7.

[22] R. A. Horn and C. R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, Cambridge, UK, 1991.

[23] J. Ibáñez, J. M. Alonso, J. Sastre, E. Defez, and P. Alonso-Jordá. Advances in the approximation of the matrix hyperbolic tangent. *Mathematics*, 9(11):1219, 2021.

[24] S. G. Johnson. More dots: Syntactic loop fusion in julia, February 2017.

[25] C. B. Moler and C. F. Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Rev.*, 20(4):801–836, 1978.

[26] B. N. Parlett. A recurrence among the elements of functions of triangular matrices. *Linear Algebra Appl.*, 14(2):117–121, 1976.

[27] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[28] W. Rudin. *Real and Complex Analysis*. 3rd edition, McGraw-Hill, Singapore, 1987.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[30] J. Sastre. Efficient evaluation of matrix polynomials. *Linear Algebra Appl.*, 539:229–250, 2018.

[31] J. Sastre, J. Ibáñez, and E. Defez. Boosting the computation of the matrix exponential. *Appl. Math. Comput.*, 340:206–220, 2019.

[32] J. Sastre, J. Ibáñez, E. Defez, and P. Ruiz. New scaling-squaring Taylor algorithms for computing the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 37(1):A439–A455, 2015.

[33] J. Sastre and J. Ibáñez. Efficient evaluation of matrix polynomials beyond the Paterson–Stockmeyer method. *Mathematics*, 9(14):1600, 2021.

[34] J. Sastre, J. Ibáñez, P. Alonso-Jordá, J. Peinado, and E. Defez. Fast Taylor polynomial evaluation for the computation of the matrix cosine. *J. Comput. Appl. Math.*, 354:641–650, 2019.

[35] G. Schulz. Iterative Berechung der reziproken Matrix. *Z. Angew. Math. Mech.*, 13(1):57–59, 1933.

[36] M. Seydaoğlu, P. Bader, S. Blanes, and F. Casas. Computing the matrix sine and cosine simultaneously with a reduced number of products. *Appl. Numer. Math.*, 163:96–107, 2021.

[37] L. Sorber, M. V. Barel, and L. D. Lathauwer. Unconstrained optimization of real functions in complex variables. *SIAM J. Optim.*, 22(3):879–898, 2012.