University of Pisa
Department of Computer Science

Master's Thesis in

Computer Science

Computation of Matrix Functions with Fully Automatic Schur-Parlett and Rational Krylov Methods

Supervisor
Federico Poloni

Author
Roberto Zanotto

Academic Year 2017/2018

UNIVERSITY OF PISA

DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis in

COMPUTER SCIENCE

Computation of Matrix Functions with Fully Automatic Schur-Parlett and Rational Krylov Methods

Supervisor
Federico Poloni

Author
Roberto Zanotto

Academic Year 2017/2018

First of all, I need to thank Federico Poloni; great supervisor, he was always available when I needed, I would totally recommend him to my fellow students... not that I have any anymore, I'm pretty much the last getting a degree here, Unipi must be shutting down or something after all this time. Thanks to my "telegram friends" for keeping in touch. In particular, I have to mention Marco Grandi, with whom I can sing out loud Ruggero's songs with no shame, and Andrea Bernabei (I got to be the "best" for one day). Also thanks to Marco Ponza; it seems like I can never get a degree without Ponza's intervention in one form or another. Last but not least, thanks to my parents and family, I couldn't have done it without them.

CONTENTS

CHAPTER

1 INTRODUCTION

Matrix functions play a diverse role in science and engineering, arising most frequently in connection with the solution of differential equations, with application areas including Lie group methods for geometric integration [17], control theory [4], the numerical solution of stiff ordinary differential equations [9] and nuclear magnetic resonance [8]. They have been studied for as long as matrix algebra itself and yet there is no language/package combination, to our knowledge, that provides an implementation of matrix functions that is both generic (working for any function f and matrix A) and automatic (requiring no other user input or data structure tied to f).

Julia is a relatively new programming language for numerical computing. Its design and features make possible the automatic differentiation of user-defined functions and consequently enable the implementation of a generic and automatic Schur-Parlett algorithm for the computation of $f(A)$. Still, no one had implemented it until now and the addition of generic matrix functions to the language is an open issue (`https://github.com/JuliaLang/julia/issues/5840`).

We provide a solution to this problem. Chapter 2 presents some basic theory on matrix functions, Chapter 3 discusses the Schur-Parlett algorithm for the computation of dense matrix functions [11] and Chapter 4 describes the AAA algorithm for rational approximation of functions [22]. In Chapter 5 we show how the AAA algorithm and the rational Arnoldi algorithm [14] can be combined for computing $f(A)b$ for sparse matrices. AAA is used to automatically find the poles of f, which are required for rational Arnoldi and the combination of the two algorithms is original, as far as we know.

We were also able to devise a couple of key performance improvements to Schur-Parlett. The algorithm has been modified to work mostly in real arithmetic, for real inputs (the original Schur-Parlett only works in complex arithmetic) and the Parlett recurrence step has been reworked to be cache-oblivious. These improvements make the procedure competitive, in terms of execution times, with the state-of-the-art specialized methods (scaling and squaring for `expm` [16], inverse scaling and squaring for `logm` [1] and Björk-Hammarling for `sqrtm` [7]), while Schur-Parlett retains the advantage of working on any function f.

The algorithms mentioned have been fully implemented in Julia. The implementation is freely available at `https://github.com/robzan8/MatFun.jl`.

CHAPTER

2 MATRIX FUNCTIONS

A matrix function is a function that takes a matrix as input and produces a matrix of the same dimensions as output. Given a scalar function f and a square matrix A, $f(A)$ can be defined in a way that provides a useful generalization of f.

2.1 Definitions of $f(A)$

There are many different but equivalent ways of defining matrix functions. We show three that are of particular interest (as does Higham in [15], most definitions and theorems in the fist part of this chapter come from his book).

2.1.1 Jordan Canonical Form

Every matrix $A \in \mathbb{C}^{n\times n}$ can be expressed in the Jordan canonical form

$$Z^{-1}AZ = J = \operatorname{diag}(J_1, J_2, \dots, J_p), \text{ where}$$

$$J_k = J_k(\lambda_k) = \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k \times m_k} \tag{2.1}$$

The Jordan matrix J is unique up to the ordering of the blocks J_k, the transformation matrix Z is not.

We denote by $\lambda_1, \dots, \lambda_s$ the distinct eigenvalues of A and let n_i be the *index* of λ_i, which is the order of the biggest Jordan block in which λ_i appears.

In order for $f(A)$ to be defined, f must be defined on A's spectrum. That is, the function and its derivatives must be defined on the eigenvalues of A. More precisely:

Definition 2.1.1. The function f is said to be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$ if the following values exist:

$$f^{(j)}(\lambda_i), \quad j = 0 : n_i - 1, \quad i = 1 : s$$

We have all the elements to present the first definition of $f(A)$.

Definition 2.1.2 (matrix function via Jordan canonical form)**.** Let f be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$ and let A have the Jordan canonical form (2.1). Then

$$f(A) := Zf(J)Z^{-1} = Z\operatorname{diag}(f(J_k))Z^{-1}, \text{ where}$$

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \dots & \dfrac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}$$

Before showing how the definition can be obtained, we highlight some of its properties.

The definition can be proved to be independent of the particular Jordan canonical form of A that is used (see [15, Chapter 1]).

Note that $f(A)$ is fully determined by the values of f on the spectrum of A (and by A itself, of course). The features of f outside the spectrum of A, such as the continuity of the function, are irrelevant.

Also, when A is diagonalizable the Jordan canonical form reduces to the eigendecomposition $A = ZDZ^{-1} = Z\ \mathrm{diag}(\lambda_i)Z^{-1}$, where the columns of Z are the eigenvectors of A. In this case Definition 2.1.2 yields $f(A) = Zf(D)Z^{-1} = Z\ \mathrm{diag}(f(\lambda_i))Z^{-1}$. Therefore, for diagonalizable matrices, $f(A)$ has the same eigenvectors of A and its eigenvalues are obtained by applying f to the eigenvalues of A.

One way of obtaining Definition 2.1.2 is from Taylor series considerations. Each Jordan block can be rewritten as $J_k = \lambda_k I + N_k \in \mathbb{C}^{m_k \times m_k}$, where N_k is the matrix of the superdiagonal of 1s. With $m_k = 3$, we have

$$N_k = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad N_k^2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad N_k^3 = 0$$

In general, powering N_k causes the superdiagonal of 1s to move up (or right), until it disappears with $N_k^{m_k}$. Assuming f has a convergent Taylor series expansion

$$f(t) = f(\lambda_k) + f'(\lambda_k)(t - \lambda_k) + \cdots + \frac{f^{(j)}(\lambda_k)(t-\lambda_k)^j}{j!} + \cdots$$

Substituting J_k for t yields the finite series

$$f(J_k) = f(\lambda_k)I + f'(\lambda_k)N_k + \cdots + \frac{f^{(m_k-1)}(\lambda_k)N_k^{m_k-1}}{(m_k-1)!}$$

since the higher powers of N_k are 0. The formula agrees with Definition 2.1.2 and an alternative derivation that does not rest on a Taylor series will be given in Section 2.1.4.

2.1.2 Polynomial Interpolation

When $f(t)$ is a polynomial with scalar coefficients and scalar argument t, it is natural to define $f(A)$ by simply substituting A for t. For example:

$$f(t) = 8t^4 + 5t + 9 \implies f(A) = 8A^4 + 5A + 9I$$

What makes matrix polynomials interesting for our purposes is the fact that

$p(A)$ is fully determined by the values of p on the spectrum of A.

Theorem 1. [15, Theorem 1.3] *For polynomials p and q and $A \in \mathbb{C}^{n\times n}$, $p(A) = q(A)$ if and only if p and q take the same values on the spectrum of A.*

Proof. We first remind the reader that the *minimal polynomial* of A is the unique monic polynomial of lowest degree such that $\psi(A) = 0$. It has the property of dividing any polynomial p such that $p(A) = 0$.

Suppose that p and q satisfy $p(A) = q(A)$. Then $d = p - q$ is zero at A and thus divisible by the minimal polynomial ψ. d takes only the value zero on A's spectrum, which implies that p and q take the same values on the spectrum of A.

Conversely, suppose p and q take the same values on the spectrum of A. Then $d = p - q$ is zero on the spectrum of A and must be divisible by the minimum polynomial ψ. Hence $d = \psi r$ for some polynomial r. Since $d(A) = \psi(A)r(A) = 0$, we can conclude $p(A) = q(A)$. □

By generalizing this property to arbitrary functions f, we can define $f(A)$ in such a way that its value is determined by the values of f on the spectrum of A.

Definition 2.1.3 (matrix function via Hermite interpolation)**.** Let f be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$. Then $f(A) := p(A)$, where p is any polynomial that takes the same values as f on the spectrum of A, i.e. p satisfies the interpolation conditions

$$f^{(j)}(\lambda_i) = p^{(j)}(\lambda_i), \quad j = 0 : n_i - 1, \quad i = 1 : s$$

Among such polynomials, the one with degree smaller than A's minimal polynomial is unique and known as the Hermite interpolating polynomial.

Definition 2.1.3 makes $f(A)$ a polynomial in A, but it is important to note that the polynomial depends on A (through the values of f on the spectrum of A). Given a function f it is impossible, in general, to find a polynomial p such that $f(A) = p(A) \quad \forall A$. Still, $f(A)$ being a polynomial in A results in useful properties:

- $f(A)$ commutes with A;
- $f(A^T) = f(A)^T$;

- $f(XAX^{-1}) = X f(A) X^{-1}$;
- if X commutes with A then X commutes with $f(A)$;
- if A is block triangular then $F = f(A)$ is block triangular with the same structure and $F_{ii} = f(A_{ii})$.

2.1.3 Cauchy Integral Theorem

The Cauchy integral theorem can be generalized to provide a concise definition of a function of a matrix.

Definition 2.1.4 (matrix function via Cauchy integral)**.** For $A \in \mathbb{C}^{n\times n}$,

$$f(A) := \frac{1}{2\pi\imath}\int_\Gamma f(z)(zI - A)^{-1}dz,$$

Where f is analytic inside and on a closed contour Γ that contains $\Lambda(A)$.

This definition leads to short proofs for certain theoretical results.

2.1.4 Equivalence of Definitions

Theorem 2 (Equivalence of definitions)**.** *Definition 2.1.2 (Jordan canonical form) and 2.1.3 (Hermite interpolating polynomial) are equivalent.*

Proof. The "Jordan definition" yields the property that $f(A)$ is determined by the values of f on the spectrum of A. This property is the only premise to the "Hermite definition".

Conversely, the formula for the function of a Jordan block $f(J_k)$ as seen in the "Jordan definition" can be obtained from the "Hermite definition" as follows. The only eigenvalue of J_k is λ_k, which leads to the interpolation conditions $f^{(j)}(\lambda_k) = p^{(j)}(\lambda_k), \quad j = 0 : m_k - 1$. Therefore, the Hermite interpolating polynomial reduces to the Taylor polynomial

$$p(t) = f(\lambda_k) + f'(\lambda_k)(t - \lambda_k) + \cdots + \frac{f^{(m_k-1)}(\lambda_k)(t-\lambda_k)^{m_k-1}}{(m_k-1)!}$$

Evaluating $p(J_k)$ proves the thesis. □

Definition 2.1.4 (Cauchy integral) can be proved to be equivalent to the other two, with the extra assumption about f's analyticity.

2.2 Computation of $f(A)$

Many different algorithms exist for computing matrix functions. Some are tied to a specific function, like the Björck-Hammarling method for the matrix square root [7], the scaling and squaring algorithm for the matrix exponential [16] and the inverse scaling and squaring method for the matrix logarithm [1]. We'll be covering only techniques applicable to general functions f, giving a brief survey in this section.

2.2.1 Taylor Series

The Taylor series can be a tool for approximating or (under specific conditions) computing matrix functions.

Theorem 3 (Convergence of matrix Taylor series)**.** *Assume f has a Taylor series expansion*

$$f(z) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\alpha)}{k!}(z-\alpha)^k$$

with radius of convergence r. Given $A \in \mathbb{C}^{n\times n}$, a sufficient condition for $f(A)$ to be defined is $|\lambda_i - \alpha| < r$ for $i = 1 \dots s$ and $f(A)$ is given by

$$f(A) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\alpha)}{k!}(A-\alpha I)^k$$

Proof. From Definition 2.1.2 follows that it is sufficient to prove the statement for a Jordan block $A = J(\lambda) = \lambda I + N$ with N strictly upper triangular. The evaluation of $f(J(\lambda))$ shows that the series converges on $J(\lambda)$ when it does on λ. For the complete proof, see [15, Theorem 4.7]. □

The Taylor series is mostly useful is specific situations, such as when the eigenvalues of A are clustered. In this case, the series is able to locally approximate f with good accuracy and a small number of terms. More details will be discussed in Chapter 3, where the Taylor series is used on this class of matrices as the component of a more sophisticated algorithm.

2.2.2 Rational Approximation

Rational functions

$$r_{km}(x) = \frac{p_k(x)}{q_m(x)},$$

where p_k and q_m are polynomials of degree at most k and m are powerful tools for approximation. They are able to capture the behavior of nonlinear functions much better than polynomials. r_{km} can be evaluated at a matrix A as $r_{km}(A) = q_m(A)^{-1}p_k(A)$.

It should be noted that, for approximating matrix functions, it is in general not sufficient to find a rational function that interpolates f on $\Lambda(A)$, but the derivatives of f on $\Lambda(A)$ must also be captured.

Different classes of rational functions can be used for matrix functions approximation. L_∞ (or *minimax* or *Chebyshev*) approximations and Padé approximations have historically been used in the field for specific applications, but they have some limitations when it comes to generic functions. L_∞ approximations as obtained with the Remez algorithm work for a real interval ($x \in [a, b]$) and are thus usually employed for Hermitian matrices only. Padé approximants don't always exist for a generic function f and require x to be near the origin.

The AAA algorithm for rational approximation [22] is a recently proposed method that can be used effectively for generic functions f. The algorithm is discussed in Chapter 4.

2.2.3 Diagonalization

Many methods for evaluating matrix functions exploit similarity transformations. $A = ZBZ^{-1}$ implies $f(A) = Z\,f(B)Z^{-1}$, which follows from the fact that $f(A)$ is a polynomial in A. The idea is to find a B similar to A such that computing $f(B)$ is easier than computing $f(A)$.

Assuming A is diagonalizable, we can write $A = Z\ \mathrm{diag}(\lambda_i)Z^{-1}$, where λ_i are the eigenvalues of A and Z contains the eigenvectors. This makes the computation trivial as $f(A) = Z\ \mathrm{diag}(f(\lambda_i))Z^{-1}$.

The drawback of this method is that it is often numerically unstable. Suppose the evaluation of $f(B)$ has error E. We have

$$\tilde{f}(A) = Z(f(B) + E)Z^{-1} = f(A) + ZEZ^{-1}$$

and

$$\|\tilde{f}(A) - f(A)\| \le \|Z\|\|E\|\|Z^{-1}\| = \kappa(Z)\|E\|$$

So, the error E is amplified by $\kappa(Z)$ and Z is ill conditioned for a wide class of matrices. Since the conditioning of $f(A)$ is not necessarily related to $\kappa(Z)$, this method is often numerically unstable.

If A is not diagonalizable, we can in theory employ the Jordan canonical form. In practice, the Jordan form cannot be reliably computed in floating point arithmetic and even if it could it would produce ill conditioned transformations.

The diagonalization approach can only be recommended when the diagonalizing transformation is guaranteed to be well conditioned. An important class of matrices for which this guarantee holds is the class of normal matrices: $A \in \mathbb{C}^{n\times n}$ for which $A^*A = AA^*$. Normal matrices are those diagonalizable by a unitary matrix and include unitary and Hermitian matrices.

2.2.4 Schur Decomposition

We saw in the previous section the importance of using a well conditioned similarity transformation to preserve the stability of the method. The Schur Decomposition

$$A = QTQ^*$$

is backward stable on any matrix $A \in \mathbb{C}^{n\times n}$. $Q \in \mathbb{C}^{n\times n}$ is unitary and $T \in \mathbb{C}^{n\times n}$ is upper triangular, which is the closest we can get to a diagonal matrix, sticking to unitary transformations.

The Schur decomposition effectively reduces the problem of computing $f(A)$ for a generic A to the evaluation of $f(T)$ with triangular T. How to evaluate $f(T)$ will be discussed in detail in the next chapter.

2.3 Computation of $f(A)b$ for Sparse A

When A is large and sparse, $f(A)$ is in general too expensive to compute and store, since the application of f most likely destroys A's sparsity. In this case, we are interested in the action of $f(A)$ on a vector b, which is $f(A)b$.

An overview of the methods for computing or approximating $f(A)b$ is given in [15, Chapter 13]. Many of them require specific knowledge about f and/or A. We'll focus of Krylov subspace methods, for which we were able to devise an algorithm that is generic, automatic and effective.

We assume the reader is familiar with Krylov subspaces and the Arnoldi process. As a quick reminder, the mth Krylov subspace of $A \in \mathbb{C}^{n\times n}$ and a nonzero vector $b \in \mathbb{C}^n$ is defined by

$$\mathcal{K}_m(A, b) = \operatorname{span}\{b, Ab, \ldots, A^{m-1}b\}$$

which can also be written as

$$\mathcal{K}_m(A, b) = \operatorname{span}\{p(A)b : p \text{ is a polynomial of degree} \le m-1\}$$

An orthonormal basis for $\mathcal{K}_m(A,b)$ can be obtained with the Arnoldi process: the first vector is $q_1 = b/\|b\|$ (basis of $\mathcal{K}_1(A,b)$); the next q_{j+1} is obtained by orthonormalizing Aq_j against the previously computed vectors $q_1,\dots,q_j$. $Aq_j \in \mathcal{K}_{j+1}(A,b)$ implies

$$Aq_j = \sum_{i=1}^{j+1} q_i h_{i,j}$$

and the Arnoldi process at step m produces the factorization

$$AQ_m = Q_m H_m + h_{m+1,m} q_{m+1} e_m^T$$

where $Q_m = [q_1,\dots,q_m] \in \mathbb{C}^{n\times m}$ is unitary and $H_m = (h_{i,j}) \in \mathbb{C}^{m\times m}$ is upper Hessenberg. After a certain number of steps that depends on A and b, and at most at step n, $\mathcal{K}_m(A,b)$ will be invariant w.r.t. A, meaning $\mathcal{K}_m(A,b) = \mathcal{K}_{m+1}(A,b)$ and the process "breaks down" (the breakdown is "lucky" when $m < n$). In this case, we have

$$Q_m^* A Q_m = H_m$$

which says that H_m is the orthonormal projection of A into $\mathcal{K}_m(A,b)$.

To approximate $f(A)b$ with the Arnoldi process, we can use

$$f(A)b \approx f(Q_m H_m Q_m^*)b = Q_m f(H_m) Q_m^* b = \|b\|_2 Q_m f(H_m) e_1$$

It has been proven by Saad [15, Theorem 13.5] that $\|b\|_2 Q_m f(H_m)e_1 = \tilde{p}_{m-1}(A)b$ where $\tilde{p}_{m-1}$ is the unique polynomial of degree at most $k-1$ that interpolates f on the spectrum of H_m. So, the approximation is exact in case of lucky breakdown and is good when $f(A)b$ can be approximated well by a polynomial of low degree. We say low degree because the method has acceptable performance only when $m \ll n$: it requires m sparse matrix-vector multiplications and orthogonalizations of q_j and the computation of $f(H_m)$ for a dense H_m.

For a generic function f, which may have poles and singularities, a rational function is better suited than a polynomial at providing an accurate approximation while keeping the degree low. The rational Arnoldi process is a generalization of the Arnoldi process that can be used to produce a rational approximation of $f(A)b$. The inconvenience of rational Krylov methods is that the poles of the rational approximant must be provided by the user. We solved this issue by using the AAA algorithm (discussed in Chapter 4) to automatically retrieve the poles of f. The rational Arnoldi process is described in Chapter 5.

CHAPTER

3 THE SCHUR-PARLETT ALGORITHM

The Schur-Parlett algorithm is pretty much the only numerically reliable method for computing $f(A)$ for generic f and A (there's also Kagström's algorithm [18] that uses similar ideas, we won't be covering it). This chapter describes the method as first presented in [11]. When we say "the original paper", "the original algorithm" and "Davies and Higham", we'll be referring to [11]. The method consists of four main steps

1. Compute A's Schur decomposition $A = QTQ^*$. This can be done with the QR algorithm with perfect backward stability;
2. Reorder the Schur decomposition in a way that clusters the eigenvalues of A. Informally, eigenvalues that are near to each other will appear together in a contiguous diagonal block, eigenvalues far from each other will be in separate blocks;
3. Compute $f(T_{ii})$ of the diagonal blocks using Taylor. This is possible since each block contains one cluster of near eigenvalues;
4. Reconstruct $f(T_{ij})$ for the off-diagonal blocks using the Parlett recurrence (explained later in Section 3.2).

In the next sections we will describe each step in detail, providing a complete implementation in Julia (except, of course, for the Schur decomposition step, which is a standard tool in linear algebra). We also show a couple of significant performance improvements to the original algorithm.

3.1 Reordering and blocking the Schur form

We will reorder the upper triangular matrix T into a (block) upper triangular $\tilde{T} = (\tilde{T}_{ij})$ through unitary transformations. Given a tolerance $\delta > 0$, we want to enforce the following conditions on the eigenvalues of $\tilde{T}$:

1. *separation between blocks:*

$$\forall \lambda \in \Lambda(\tilde{T}_{ii}), \mu \in \Lambda(\tilde{T}_{jj}) \quad i \neq j \implies |\lambda - \mu| > \delta \tag{3.1}$$

2. *separation within blocks:*
   for each block $\tilde{T}_{ii} \in \mathbb{C}^{m \times m}$ with $m > 1$, for each $\lambda \in \Lambda(\tilde{T}_{ii})$ there is a $\mu \in \Lambda(\tilde{T}_{ii})$ with $\mu \neq \lambda$ such that $|\lambda - \mu| \leq \delta$. This implies

$$\forall \lambda \in \Lambda(\tilde{T}_{ii}), \mu \in \Lambda(\tilde{T}_{ii}) \quad |\lambda - \mu| \leq (m-1)\delta$$

Different parts of the method have conflicting requirements for the value of δ. The Taylor series evaluation converges faster and more accurately with small δ, which translates to many small blocks with near eigenvalues. The Parlett recurrence works best with bigger δ, which results in few, well-separated big blocks, as the recurrence becomes numerically unstable when different blocks have near eigenvalues. Davies and Higham suggest a value of $\delta = 0.1$, obtained through numerical experiments.

The first step to obtaining the ordering is to assign each eigenvalue to a group, such that the groups satisfy the aforementioned conditions. The incremental algorithm to obtain such "block pattern" is straightforward: each eigenvalue λ_i is considered, assuming at first that it forms a group on its own, and then compared to the already examined eigenvalues, merging groups when appropriate. The procedure is probably best explained by code:

```
#= blockpattern groups eigenvalues in sets. The function returns S
    and p. S is the pattern, where S[i] = s means that vals[i] has
    been assigned to set s. p is the number of sets identified.
=#

delta = 0.1

function blockpattern(vals::Vector{Complex128}, schurtype::Type)

    unassigned = -1
    S = fill(unassigned, length(vals))
    p = 0

    # assign vals[i] to set s.
    function assign(i::Int64, s::Int64)
        lambda = vals[i]
        for k = i:length(vals)
            if vals[k] == lambda
                S[k] = s
            end
        end
    end

    function mergesets(s::Int64, t::Int64)
        if s == t
            return
        end
        smin, smax = minmax(s, t)
        for k = 1:length(vals)
            if S[k] == smax
                S[k] = smin
            elseif S[k] > smax
                S[k] -= 1
            end
        end
        p -= 1
    end
    for i = 1:length(vals)
        if S[i] == unassigned
            p += 1
            assign(i, p)
        end
        for j = i+1:length(vals)
            if abs(vals[i]-vals[j]) <= delta
                if S[j] == unassigned
                    assign(j, S[i])
                else
                    mergesets(S[i], S[j])
                end
            end
        end
    end

    if schurtype <: Real
        #= complex conjugate eigenvalues can't be separated in the
            real Schur factorization, so we merge the sets involving
            them:
        =#
        i = 1
        while i < length(vals)
            if imag(vals[i]) != 0
                mergesets(S[i], S[i+1])
                i += 2
            else
                i += 1
            end
        end
    end
    return S, p

end # function blockpattern
```

The last part of the function shows a potential issue when dealing with real matrices. When reordering a real Schur factorization, complex conjugate eigenvalues cannot be separated. This can be a problem since conjugate eigenvalues with big imaginary part are forced into the same block and challenge the Taylor series evaluation. The original authors suggest to use complex arithmetic for the whole method, including the Schur decomposition, even when A is real. We found a solution that is more efficient. For now, we keep complex conjugate eigenvalues together, as shown in the code. This will result in some blocks containing not one, but two conjugate clusters of eigenvalues, violating the conditions on blocks separation. We will show how we handle those blocks in Section 3.4.

Once the block pattern is obtained, our remaining task is to reorder the Schur decomposition. For that purpose, we use Julia's `ordschur(T, Q, select)`. `select` is a vector of booleans and `ordschur` reorders the Schur factors T and Q bringing to the top the eigenvalues where `select` is true. `ordschur` uses LAPACK's `trsen` method, which works by swapping pairs of adjacent eigenvalues in T (with calls to `trexc`). So, the goal is to obtain a confluent permutation of the eigenvalues, minimizing the number of swaps required in the process. By confluent permutation we mean a permutation where eigenvalues belonging to the same group are adjacent. For example, if the block pattern S is

$$[3, 1, 3, 3, 1, 3, 1, 2, 1, 2, 2]$$

a possible confluent permutation is given by

$$[2, 2, 2, 1, 1, 1, 1, 3, 3, 3, 3]$$

and another confluent permutation, requiring fewer swaps, is

$$[3, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2]$$

In general, finding an optimal swapping strategy is an NP-complete problem. In practice, the QR algorithm tends to order the eigenvalues by absolute value in the Schur form. Thus, a simple and effective strategy is the following: find the group of eigenvalues with the smallest mean position (group 3 in the example above), bring them to the top with `ordschur` and then reapply the same strategy recursively on the remaining eigenvalues. The following code implements the reordering with said strategy.

```
# analogous to ordschur/trsen, but for vectors:
function ordvec!(v::Vector{T}, select::Vector{Bool}) where {T}
    ilst = 1
    for ifst = 1:length(v)
        if select[ifst]
            if ifst != ilst
                v[ilst], v[ilst+1:ifst] = v[ifst], v[ilst:ifst-1]
            end
            ilst += 1
        end
    end
end
#= reorder reorders the Schur decomposition (T, Q, vals) according
    to pattern S. The returned vector, blockend, contains the indices
     at which each block ends (after the reordering).
=#
function reorder!(T::Matrix{N}, Q::Matrix{N}, vals::Vector{
    Complex128}, S::Vector{Int64}, p::Int64) where {
    N<:Union{Float64, Complex128}}

    # for each set, calculate its mean position in S:
    pos = zeros(Float64, p)
    cou = zeros(Int64, p)
    for i = 1:length(S)
        set = S[i]
        pos[set] += i
        cou[set] += 1
    end
    pos ./= cou

    blockend = zeros(Int64, p)
    for set = 1:p
        numdone = (set == 1) ? 0 : blockend[set-1]
        minset = indmin(pos)
        select = [i <= numdone || S[i] == minset for i = 1:length(S)]
        ordschur!(T, Q, select)
        ordvec!(vals, select)
        ordvec!(S, select)
        blockend[set] = count(select)
        pos[minset] = Inf
    end
    return blockend
end
```

As for the performance cost, `blockpattern` runs in $O(n^3)$ and `reorder` has a best case of $O(n)$ and a worst case of $O(n^3)$, depending on the number of swaps. In practice, the cost of both is small compared to the other steps of the algorithm.

3.2 Evaluation of the off-diagonal part of $f(T)$

Before showing how to evaluate $f(T_{ii})$ of the diagonal blocks, we explain how the upper part of $f(T)$ can be obtained with the Parlett recurrence, assuming the diagonal has been computed.

If $T = (T_{ij})$ is block upper triangular, $f(T) = F = (F_{ij})$ has the same block structure and F commutes with T, as seen in Section 2.1.2. Parlett [23] notes that equating (i, j) blocks in $TF = FT$ leads to

$$T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1} (F_{ik}T_{kj} - T_{ik}F_{kj}), \quad i < j$$

This recurrence can be used to compute any super-diagonal block F_{ij}, given the column of blocks below it and the row on its left, as the solution of a triangular Sylvester equation (of the form $AX - XB = C$). This is where the importance of the separation between blocks (3.1) factors in. When $A = T_{ii}$ and $B = T_{jj}$ have eigenvalues in common, the equation is singular and when they have near eigenvalues, the equation is ill-conditioned.

Davies and Higham suggest to iteratively compute each F_{ij}. This is inefficient in terms of cache behavior, no matter if the matrices are stored in column- or row-major order, as setting up each Sylvester equation requires to read two rows and two columns of blocks. We adopted a different approach. When T only has 2×2 blocks, the Parlett recurrence reduces to

$$T_{11}F_{12} - F_{12}T_{22} = F_{11}T_{12} - T_{12}F_{22}$$

This equation can be used to set up a recursive version of the recurrence. We divide T (and F) in 2×2 "superblocks" of roughly equal size (the superblocks will in general contain many smaller atomic blocks, determined by the eigenvalues of T). F_{11} and F_{22} are computed recursively and then F_{12} is found solving one Sylvester equation. The base case of the recursion is of course when a superblock coincides with an atomic block. Julia code follows.

```
#= recf computes f(T) using the block Parlett recurrence. blockend
   contains the index at which each atomic block ends.
=#
function recf(f::Func, T::Matrix{N}, vals::Vector{Complex128},
   blockend::Vector{Int64}) where {
   Func, N<:Union{Float64, Complex128}}

   @assert length(blockend) > 0
   if length(blockend) == 1
      return blockf(f, T, vals)
   end

   # split T in 2x2 superblocks of size ~n/2:
   n = size(T, 1)
   b = indmin(abs.(blockend .- n/2))
   bend = blockend[b]
   T11, T12 = T[1:bend, 1:bend], view(T, 1:bend, bend+1:n)
   T21, T22 = view(T, bend+1:n, 1:bend), T[bend+1:n, bend+1:n]

   F11 = recf(f, T11, vals[1:bend], blockend[1:b])
   F22 = recf(f, T22, vals[bend+1:n], blockend[b+1:end] .- bend)

   # T11*F12 - F12*T22 = F11*T12 - T12*F22
   F12, s = LAPACK.trsyl!('N', 'N', T11, T22, F11*T12 - T12*F22, -1)

   return [F11 F12/s; zeros(T21) F22]
end
```

The number of flops for the procedure can be computed as follows. Assume T is real with size $n \times n$ and the base cases (the atomic blocks) have size 1×1. The cost is given by a call to `trsyl` with matrices of size $n/2$ ($n^3/4$ flops) and two matrix multiplications of size $n/2$ ($2n^3/4$ flops) for a total of $3n^3/4$, plus the two recursive calls. Considering a recursion depth of $\log_2(n)$, the whole algorithm costs

$$\sum_{i=0}^{\log_2 n} 2^i \frac{3}{4}\left(\frac{n}{2^i}\right)^3 = \frac{3}{4}n^3 \sum_{i=0}^{\log_2 n}\left(\frac{1}{4}\right)^i \leq \frac{3}{4}n^3 \sum_{i=0}^{\infty}\left(\frac{1}{4}\right)^i = n^3$$

The recursive version has the advantage, over the iterative one, of being cache-oblivious. For relatively small matrices ($n = 50$) the two versions perform the same, but for larger matrices ($n = 2500$) the recursive procedure is about 3 times faster (on `Float64`). If the reader is interested in comparing the implementations, there is an old branch in the repository containing "vanilla" Schur-Parlett, without our modifications: `https://github.com/robzan8/MatFun.jl/tree/asinpaper`

3.3 Automatic differentiation

The evaluation of $f(T_{ii})$ with Taylor will require the derivatives of f. We want an efficient and accurate method for computing $f^{(i)}(x)$ for $x \in \mathbb{C}$ automatically. Forcing the user to provide the derivatives of f would be a burden for the API and would hurt the package's usability. As an example, we compare the computation of the cube root of A with MATLAB versus our package. MATLAB's `funm` for computing $f(A)$ requires the function f to be specified as a function $f(x, k)$, returning the kth derivative of f at x. In general, it is the user's responsibility to provide the implementation for $f(x, k)$. In case of the cube root, we have

```
function r = cuberoot(x, k)
    c = 1.0;
    for i = 0:k-1
        c = c*(1/3 - i);
    end
    r = c*x^(1/3 - k);
end
```

$A^{1/3}$ is then computed with `funm(A, @cuberoot)`. With our package, all that is needed is `schurparlett((x)->x^(1/3), A)`.

While the cube root is a simple example, for certain functions that involve for loops and complex control flow the implementation of $f(x, k)$ can be difficult and error-prone.

The following is an overview on how automatic differentiation can be accomplished.

3.3.1 Finite difference method

A naive approach is to use a finite difference. The derivative is computed directly from its definition with $\lim_{h\to 0}$, by choosing an appropriately small h:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

The advantage of this method is that it works out of the box with just an handle to f, the disadvantage is that it is often unstable: it has a truncation error of $O(h^2)$ and when h is too small $f(x-h)-f(x-h)$ suffers numerical cancellation.

3.3.2 Complex step method

A more sophisticated and effective method is the complex step method. The first order Taylor expansion of $f(x+ih)$ gives:

$$f(x+ih) = f(x) + f'(x)ih + O(h^2)$$

which implies

$$f'(x) = \frac{Im[f(x+ih)]}{h} + O(h^2)$$

Since the formula is not subject to cancellation errors, h can be small (like 10^{-8}) and the truncation error can be made close to machine precision in practice.

The drawback of this method is that it works only for "real" functions (for which $f(\overline{x}) = \overline{f(x)}$ holds) and real inputs. When the function does complex operations such as conjugation, the method is disrupted. Also, the function may include code like `"if x < y"`, which is defined for real inputs, but not for complex ones. These limitations can be overcome with the use of dual numbers.

3.3.3 Dual number method

Dual numbers are similar to complex numbers in the sense that they have a real part and another part, the *epsilon* part. They are defined as:

$$x + \epsilon y, \qquad \epsilon \neq 0, \quad \epsilon^2 = 0$$

ϵ represents an infinitesimal perturbation of the first order. For simplicity, we call x "real part", but note that dual numbers work when $x \in \mathbb{C}$. Operations such as multiplications and additions can be defined on dual numbers, respecting the rule $\epsilon^2 = 0$. Using Taylor, again, leads to:

$$f(x+\epsilon y) = f(x) + f'(x)\epsilon y + \frac{f''(x)}{2!}\epsilon^2 y^2 + \cdots = f(x) + f'(x)\epsilon y$$

since all terms involving ϵ^i for $i \geq 2$ are zero. This implies

$$f'(x) = Eps[f(x+\epsilon)]$$

The advantages of this method are that it has no truncation error and it works on complex inputs. The disadvantage is that it requires an implementation of dual numbers and the functions to be differentiated must accept dual inputs. This is one of the main reasons why we chose Julia as the language for our implementation. Julia has packages implementing dual numbers and the language's type system makes it possible for them to work out of the box. Dual numbers implement a certain interface and this makes them usable as a drop-in replacement for any `Real` or `Complex` number in Julia functions.

As an example, suppose we have $f(x) = (x-3)^8$ and we want to compute $f'(5) = 8(5-3)^7 = 1024$. The function can be defined in Julia as

```
function f(x)
    x -= 3
    for i = 1:3
        x *= x
    end
    return x
end
```

Evaluating `f(Dual(5, 1))`, which is $f(5+\epsilon)$, returns `Dual(256, 1024)`, which is $256 + \epsilon 1024$, indicating that $f(5) = 256$ and $f'(5) = 1024$. The operations executed during the computation are:

$$\begin{aligned}
&(5+\epsilon) - 3 = 2 + \epsilon,\\
&(2+\epsilon)(2+\epsilon) = 4 + 4\epsilon,\\
&(4+4\epsilon)(4+4\epsilon) = 16 + 32\epsilon,\\
&(16+32\epsilon)(16+32\epsilon) = 256 + 1024\epsilon.
\end{aligned}$$

Functions as `exp`, `log`, `sin` etc. are also defined on dual numbers, making them available inside any user-defined function.

Dual numbers can also be generalized for higher order derivatives. The implementation we rely on is package `TaylorSeries.jl`. The usage is `"f(3 + Taylor1(Complex128, 10))"` for computing the Taylor coefficients of order 10 of f around 3. The package can be found at `https://github.com/JuliaDiff/TaylorSeries.jl`

3.4 Evaluation of $f(T_{ii})$ of the atomic blocks

Given an upper triangular $T \in \mathbb{C}^{n \times n}$ with "close" eigenvalues, we need a method for computing $f(T)$ accurately and efficiently. One approach is to expand f with a Taylor series about the mean of the eigenvalue of T, as proposed by Stewart [21, Method 18] for the matrix exponential, investigated by Kagström [18] for general functions and of course also by Davies and Higham in [11].

Define M as T shifted by the mean of its eigenvalues:

$$T = \sigma I + M, \qquad \sigma = \operatorname{trace}(T)/n$$

Suppose f has Taylor series expansion

$$f(\sigma + z) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\sigma)}{k!} z^k$$

for z in an open disk containing $\Lambda(M)$. Then, according to Theorem 3,

$$f(T) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\sigma)}{k!} M^k \tag{3.2}$$

If T only has one eigenvalue, then M is strictly upper triangular with $M^n = 0$ and the series (3.2) is finite. More generally, if T has sufficiently close eigenvalues, the diagonal of M is small, its powers quickly decay after the $(n-1)$st and a suitable truncation of the series should achieve good accuracy. This notion is made more precise in the following lemma. Write $M = D + N$ with D diagonal and N strictly upper triangular.

Lemma 4. [11, Lemma 2.1] *Let $D \in \mathbb{C}^{n\times n}$ be diagonal with $\|D\| \le \delta$ and let $N \in \mathbb{C}^{n\times n}$ be strictly upper triangular. Then*

$$\|(D+N)^k\| \le \sum_{i=0}^{\min(k,n-1)} \binom{k}{i} \delta^{k-i} \|N\|^i$$

for any consistent matrix norm.

Proof. The bound follows from a binomial expansion of $(D+N)^k$, considering $\|D\| \le \delta$ and $N^{n-1} = 0$ (we can drop the terms N^i with $i \ge n-1$). □

If $\delta < 1$ and $\delta \ll \|N\|$, then for $k \ge n-1$

$$\|(D+N)^k\| = O(\delta^{k+1-n}\|N\|^{n-1})$$

so the powers of $M = D+N$ decay rapidly after the $(n-1)$st, irrespective of N.

We can conclude that, as long as the scalar multipliers $f^{(k)}(\sigma)/k!$ in (3.2) are not too large, we should be able to truncate the series soon after the $(n-1)$st term (and possibly sooner).

We need a reliable criterion for deciding when to truncate the Taylor series. When the terms decrease monotonically, it is safe to stop as soon as a term is found smaller than the desired error. Unfortunately, our matrix Taylor series can have non-monotonic behavior. For example, when $n = 2$, $M = T - \sigma I$ has the form

$$M = \begin{bmatrix} \epsilon & \alpha \\ 0 & -\epsilon \end{bmatrix}$$

and its powers are

$$M^{2k} = \begin{bmatrix} \epsilon^{2k} & 0 \\ 0 & \epsilon^{2k} \end{bmatrix}, \qquad M^{2k+1} = \begin{bmatrix} \epsilon^{2k+1} & \alpha\epsilon^{2k} \\ 0 & -\epsilon^{2k+1} \end{bmatrix}$$

So for $|\epsilon| < 1$, $\|M^k\| \to 0$, but $\|M^{2k+1}\| \gg \|M^{2k}\|$ for $\alpha \gg 1$.

Davies and Higham were able to develop a strict bound for the truncation error of the Taylor series:

$$\|f(\sigma I + M) - \sum_{k=0}^{s-1} \frac{f^{(k)}(\sigma)}{k!} M^k\|_\infty \le \frac{1}{s!} M^s \max_{0 \le r \le n-1} \frac{\omega_{s+r}}{r!} \|(I - |N|)^{-1}\|_\infty$$

where $\omega_{s+r} = \sup_{z\in\Omega} |f^{(s+r)}(z)|$ and Ω is a closed convex set whose interior contains $\Lambda(T)$. From this bound we can derive an easy to implement stopping criterion that works well in practice for smooth enough functions: once a small term $(f^{(k)}(\sigma)/k!)M^k$ is found, we check if the next n terms are also small; if they are, we can terminate.

The implementation of $f(T)$ using Taylor follows.

```
#=
taylorf computes f(T) using Taylor.
=#
function taylorf(f::Func, T::Mat, shift::N)::Mat where {Func,
    N<:Number, Mat<:Union{Matrix{N}, UpperTriangular{N, Matrix{N}}}}

    maxiter = 300
    lookahead = size(T, 1)
    # compute the Taylor coefficients:
    tay = f(shift + Taylor1(N, 10+lookahead))
    M = T - shift*Mat(eye(T))
    P = M
    F = tay.coeffs[1]*Mat(eye(T))
    for k = 1:maxiter
        needorder = k + lookahead
        @assert needorder <= tay.order + 1
        if needorder > tay.order
            # don't have enough Taylor coefficients, recompute:
            tay = f(shift + Taylor1(N, min(maxiter+lookahead, tay.
                order*2)))
        end

        Term = tay.coeffs[k+1] * P
        F += Term
        normP = norm(P, Inf)
        P *= M
        small = eps()*norm(F, Inf)
        if norm(Term, Inf) <= small
            #= We check that the next lookahead terms in the series
                are small, estimating ||M^(k+r)|| with max(||M^k||, ||
                M^(k+1)||). Works well in practice, at least for exp,
                log and sqrt.
            =#
            delta = maximum(abs.(tay.coeffs[k+2:k+1+lookahead]))
            if delta*max(normP, norm(P, Inf)) <= small
                return F
            end
        end
    end
    error("Taylor did not converge.")
end
```

The worst-case cost of the procedure is $O(n^4)$ as, even when M is nilpotent, the algorithm may need to compute $O(n)$ powers of M. This is an high cost (algorithms in numerical linear algebra rarely exceed $O(n^3)$ flops), but is mitigated by two factors: M is (quasi) triangular, translating to faster matrix multiplications, and the blocks typically have small size, compared to the whole matrix A.

As seen in Section 3.1, when computing $f(A)$ for real A, the diagonal blocks may contain two conjugate clusters of eigenvalues, which are not suitable for evaluation with Taylor. We deal with this by splitting such blocks with the complex Schur decomposition, resorting to complex arithmetic only at the block level, when needed. The function that takes care of it is `blockf`, which also handles other special cases, such as 1×1 and 2×2 blocks.

```
#=
blockf computes f(T) where T is a diagonal block.
=#
function blockf(f::Func, T::Matrix{N}, vals::Vector{Complex128})
    where {Func, N<:Union{Float64, Complex128}}

    n = size(T, 1)
    if n == 1
        return f.(T)
    end

    if N <: Complex
        # T is complex and triangular with one cluster of eigenvalues
        return taylorf(f, UpperTriangular(T), mean(vals))
    end

    if all(imag(vals) .== 0)
        # T is real and triangular with one cluster of real
            eigenvalues.
        return taylorf(f, UpperTriangular(T), mean(real(vals)))
    end

    if any(abs.(imag(vals)) .<= delta/2)
        # T is real and quasi-triangular with one cluster of complex
            conjugate eigenvalues (and possibly some real eigenvalue).
        return taylorf(f, T, mean(real(vals)))
    end



    if n == 2
        #= T is real and 2x2 with two well-separated complex
            conjugate eigenvalues. We use the Hermite interpolating
            polynomial obtained with the Lagrange-Hermite formula,
            simplified assuming f(conj(x)) == conj(f(x)).
        =#
        v1, v2 = vals[1], vals[2]
        @assert v1 == conj(v2)
        psi1 = f(v1)/(v1 - v2)
        return 2.0*real(psi1*(T - [v2 0; 0 v2]))
    end

    #= T is real with two well-separated, complex conjugate clusters
        of eigenvalues. We use the complex Schur factorization to
        further divide T. Doing complex Schur at the block level is
        more efficient than using complex arithmetic for the whole
        matrix.
    =#
    U, Z, vals = schur(Matrix{Complex128}(T))::Tuple{
        Matrix{Complex128}, Matrix{Complex128}, Vector{Complex128}}
    select = imag(vals) .> 0
    ordschur!(U, Z, select)
    ordvec!(vals, select)
    @assert count(select) == div(n, 2)
    F = Z*recf(f, U, vals, [div(n, 2), n])*Z'
    realF = real(F)
    if norm(imag(F), Inf) > sqrt(eps())*norm(realF, Inf)
        warn("T is real but f(T) has non-negligible imaginary part.")
        # note: we need f(T) to be real, because trsyl can't handle
            quasi-triangular complex matrices.
    end
    return realF

end #function blockf
```

It may seem like doing the real Schur decomposition on the whole A and then having to redo complex Schur on the diagonal blocks is a waste of resources, compared to the direct use of complex arithmetic on A; it is not. First of all, the QR algorithm is not cache friendly. Pushing complex operations to smaller blocks is already a win and the Schur factorization on the diagonal blocks does not show up in profiles, typically.
Second, confining complex operations to the block level allows every other step of the algorithm to work with real operations: triangularization of A,

reordering of the eigenvalues of T and the Parlett recurrence.
This speeds up the whole algorithm by a factor of 2 (results vary depending on the distribution of the eigenvalues and the size of the blocks).

3.5 Whole algorithm

As anticipated in the beginning of the chapter, the algorithm works by computing the Schur factorization $A = QTQ^*$, reordering and blocking T, computing $f(T_{ii})$ of the diagonal blocks and then evaluating the upper part of $f(T)$ with the Parlett recurrence. In Julia:

```julia
#= Computes f(A) using the Schur-Parlett algorithm. When A is a real
     matrix, computation will be done mostly in real arithmetic and
    the algorithm will assume f(conj(x)) == conj(f(x)).
=#
function schurparlett(f::Func, A::Matrix{N})::Matrix{N} where {
    Func, N<:Union{Float64, Complex128}}

    T, Q, vals = schur(A)
    return schurparlett(f, T, Q, vals)
end
```

```
#= Analogous to schurparlett(f, A), but A is provided as its Schur
    decomposition.
=#
function schurparlett(f::Func, T::Matrix{Float64}, Q::Matrix{Float64
    }, vals::Vector{Float64})::Matrix{Float64} where {Func}

    return schurparlett(f, T, Q, Vector{Complex128}(vals))
end
function schurparlett(f::Func, T::Matrix{N}, Q::Matrix{N}, vals::
    Vector{Complex128})::Matrix{N} where {
    Func, N<:Union{Float64, Complex128}}

    if size(T, 1) != size(T, 2)
        throw(DimensionMismatch("T must be square"))
    end
    if size(Q) != size(T) || length(vals) != size(T, 1)
        throw(DimensionMismatch("T, Q, vals dimension mismatch"))
    end
    if size(T, 1) == 0
        error("T is empty")
    end

    d = diag(T)
    D = diagm(d)
    if norm(T-D, Inf) <= 1000*eps()*norm(D, Inf) # T is diagonal
        return Q*diagm(f.(d))*Q'
    end

    S, p = blockpattern(vals, N)
    blockend = reorder!(T, Q, vals, S, p)
    return Q*recf(f, T, vals, blockend)*Q'
end
```

As discussed in Section 2.2.3, when A is a normal matrix ($A^*A = AA^*$) the Schur decomposition reduces to an eigendecomposition, T is diagonal and $f(T)$ can be computed trivially.

3.6 Numerical Accuracy

The modifications we proposed on the original method, oriented to improving performance, do not change its numerical accuracy. We confronted the accuracy of `funm` in MATLAB 9.4 (R2018a), implementing vanilla Schur-Parlett, and the one of our algorithm against MATLAB's Symbolic Math Toolbox as oracle. The functions tested were the matrix exponential, logarithm and square root, on several test matrices of size 13×13 from the function `matrix` of the Matrix Computation Toolbox (`http://www.ma.man.ac.uk/~higham/mctoolbox/`, the same gallery used in [11, Experiment 10]). The two versions of Schur-Parlett achieve almost identical relative errors.

The algorithm is in general competitive with specialized methods such as `expm` (scaling and squaring), `logm` (inverse scaling and squaring) and `sqrtm` (Björck-Hammarling). The typical relative error for said functions on random matrices is of the order of 10^{-14}, close to what is imposed by the condition number, with machine precision of 10^{-16}.

We won't discuss all the experiments carried by Davies and Higham, we will only show when the algorithm can behave in an unstable manner. Schur-Parlett struggles when the eigenvalues of T are difficult to cluster. An example is Experiment 4 [11], where e^T is computed and T is obtained with the following MATLAB code:

```
n = 50; randn('state', 1);
B = triu(randn(n), 1) + eye(n);
Q = gallery('orthog', n);
B = Q*B*Q'; T = schur(B, 'complex');
```

When T is formed in exact arithmetic, it has n eigenvalues 1, but the computed T has the eigenvalues approximately located on a circle, as shown in Figure 3.1.

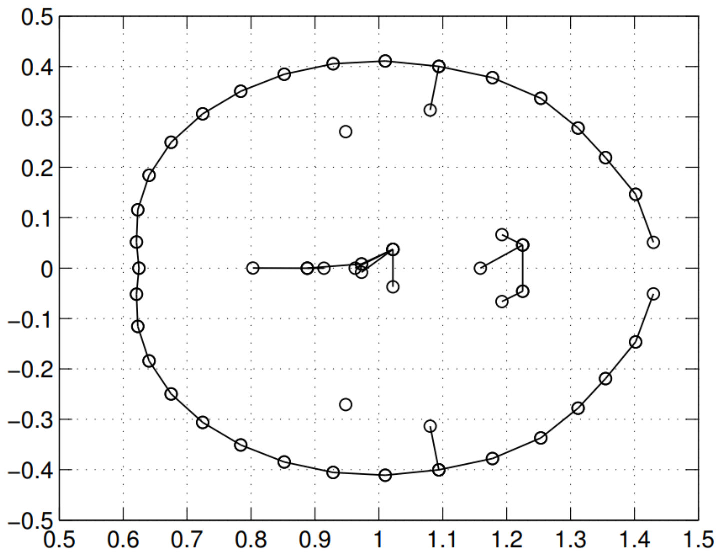

Figure 3.1: Eigenvalues of T for Experiment 4, from [11]. Eigenvalues in the same block are connected by a line.

On this experiment, the condition number is about 300 but the error is 7×10^{-4}. By using a value of $\delta = 0.2$ for the `blockpattern` algorithm, all the eigenvalues are grouped in one block and the function is evaluated entirely with Taylor, resulting in an error of 1.4×10^{-14} (less than what is dictated by the condition number). This suggests that it may be better to overshoot δ, but, depending on A, increasing δ too much can be as bad as having it too small (Experiments 5 and 6).

The issue arises from the fact that "snakes" of eigenvalues as shown in Figure 3.1 cannot be split, without compromising the numerical accuracy of the Parlett recurrence. Still, keeping such groups of eigenvalues together can challenge the evaluation via Taylor series.

3.7 Performance Results

The factor that influences Schur-Parlett's performance the most is the distribution of A's eigenvalues, which determines the size of the atomic blocks. When the eigenvalues are scattered, the atomic blocks are small, the Taylor series evaluation has negligible impact and the cost of the procedure is given by the Schur decomposition and the Parlett recurrence. When the eigenvalues are clustered, the atomic blocks are large and the Taylor series has bigger impact.

f doesn't have a big influence on the whole algorithm's performance, as will be apparent from our experiments.

We tested the performance of `expm`, `sqrtm`, `logm` and `schurparlett` on matrices cQ, where c is a scalar that allowed us to control the number and size of the resulting atomic blocks. Q has been obtained with the following code.

```
srand(666)
A = randn(1500) + im*randn(1500)
T, Q, vals = schur(A)
Q += 2*eye(Q)
```

The results of the experiments are in Table 3.1.

Input	# atomic blocks	Size of blocks	Time (s) expm	Time (s) sqrtm	Time (s) logm	Time (s) schurparlett
Q*100	1481	1 to 2	12	22	44	22
Q*13	56	2 to 88	11	22	42	22
Q*10	7	56 to 386	11	22	44	27

Table 3.1: Size and number of atomic blocks for different values of c and corresponding execution times, in seconds.

Schur-Parlett appears only in one column because its execution times stayed the same independently on the function (`exp`, `sqrt` or `log`) to be computed.

Table 3.1 shows that execution times are approximately constant, except for the case when cQ has only 7 large blocks. In this case, profiling confirms that Schur-Parlett spends more time for the Taylor series evaluation. It is worth noting that, even in this situation, the computation times do not change significantly for different functions.

Except in the aforementioned case, most of the time is in general spent for the Schur decomposition and the Parlett recurrence. The QR algorithm takes up about 2/3 of the total execution time, suggesting that there's little room for further improvements.

CHAPTER

4 THE AAA ALGORITHM FOR RATIONAL APPROXIMATION

The AAA algorithm (pronounced "triple-A") is a new algorithm for rational approximation of functions proposed at the end of 2016 [22]. It has a combination of speed, flexibility and robustness not seen in other algorithms:

- It works on any function $\mathbb{C} \to \mathbb{C}$ and $\mathbb{R} \to \mathbb{R}$, with no assumptions on f or its domain;
- It is fast and accurate;
- It is implementable in a few lines of code;
- It requires no user input, other than the function samples.

It works by combining two ideas. First, the rational approximant is represented in barycentric form, following Antoulas and Anderson [3] (AAA stands for "adaptive Antoulas-Anderson"). Second, the approximation degree is grown incrementally, choosing support points in a greedy fashion that aims to avoid numerical instabilities.

4.1 Rational Barycentric Representation

The barycentric formula consists of a quotient of two partial fractions,

$$r(z) = \frac{n(z)}{d(z)} = \sum_{j=1}^{m} \frac{w_j f_j}{z - z_j} \Bigg/ \sum_{j=1}^{m} \frac{w_j}{z - z_j}, \tag{4.1}$$

where $m \geq 1$ is an integer, $z_1, \ldots, z_m$ are distinct *support points*, $f_1, \ldots, f_m$ are *data values* and $w_1, \ldots, w_m$ are *weights*. Support points, data values and weights can be real or complex. We'll show some properties of the rational barycentric form.

First of all, note that r is a rational function of type $(m-1, m-1)$, meaning that it can be rewritten as a quotient of polynomials $p(z)/q(z)$ with p and q of degree at most $m-1$. This can be seen by taking the lowest common denominator in both $n(z)$ and $d(z)$, which then cancels out.

$$l(z) = \prod_{j=1}^{m} (z - z_j), \qquad r(z) = \frac{p(z)/l(z)}{q(z)/l(z)} = \frac{p(z)}{q(z)} \tag{4.2}$$

By looking at (4.1) it may seem like the support points z_j are poles, but they are actually interpolation points. Indeed, the discontinuity of $r(z_j)$ is removable by setting $r(z_j) = f_j$, since $\lim_{z \to z_j} r(z) = f_j$. So, r can have poles anywhere but in the support points (assuming the weights are nonzero).

Another interesting property, outside the scope of our work, is that for any given set of support points z_j, there is a particular choice of the weights w_j such that r becomes a polynomial of degree $m-1$, providing a numerically stable method for polynomial interpolation even in thousands of points [6].

The above properties can be summarized by the following theorem.

Theorem 5 (Rational barycentric representations)**.** [22, Theorem 2.1] *Let $z_1, \ldots, z_m$ be an arbitrary set of distinct complex numbers. As $f_1, \ldots, f_m$ range over all complex values and $w_1, \ldots, w_m$ range over all nonzero complex values, the functions*

$$r(z) = \frac{n(z)}{d(z)} = \sum_{j=1}^{m} \frac{w_j f_j}{z - z_j} \Bigg/ \sum_{j=1}^{m} \frac{w_j}{z - z_j}, \tag{4.3}$$

range over the set of all rational functions of type $(m-1, m-1)$ that have no poles at the points z_j. Moreover, $r(z_j) = f_j$ for each j

Proof. By (4.2), any quotient $n(z)/d(z)$ is a rational function of type $(m-1, m-1)$. Also, since $w_j \neq 0$, d has a simple pole at z_j and n has either a simple pole there (if $f_j \neq 0$) or no pole. Therefore r has no pole at z_j.

Conversely, suppose $r = p/q$ is a rational function with no poles at z_j, where p and q are polynomials of degree at most $m - 1$ with no common zeros. Then p/l is a rational function with a zero at ∞ and simple poles at the points z_j or a subset of them. Therefore, since $w_j \neq 0$, p/l can be written in the partial fraction form of a numerator n in (4.3). Similar considerations can be made for q. $\square$

The theorem shows that the support points z_j do not have influence on the set of functions described by (4.3) (except for the fact that they can't be poles). Their role is in determining the numerical quality of the formula. The barycentric form is composed of terms $1/(z - z_j)$ and for good choices of z_j these basis functions are independent enough to make the representation well-conditioned — often much better conditioned than a quotient of polynomials $p(z)/q(z)$.

The use of localized and sometimes singular basis functions is well-established in other areas of scientific computing. Examples are radial basis functions [12], the method of fundamental solutions [5] and the discretizations of the Cauchy integral formula by the trapezoidal rule on the unit circle, where the implicit basis functions are singular [20]. The AAA algorithm applies this kind of thinking to function theory.

4.2 Core AAA Algorithm

The function f to be approximated is provided to the algorithm as a finite *sample set* $Z \subseteq \mathbb{C}$ of $M \gg 1$ points, with the corresponding *data values* $F \subseteq \mathbb{C}$ (such that $f(Z_j) = F_j$ for $j = 1 \dots M$). The method doesn't need any knowledge of f, outside of the points on which the function is sampled.

The algorithm takes the form of an iteration for $m = 1, 2, \dots$ that incrementally increases the degree of the barycentric approximant r. At step m, a new support point z_m is selected, greedily avoiding numerical instabilities, and then the weights $w_1, \dots, w_m$ are recomputed, minimizing the approximation error on Z.

The selection of the next support point is as follows. Let $Z^{(m)}$ be the set of samples not selected as support points at step m, that is $Z^{(m)} = Z \setminus \{z_1, \dots, z_m\}$ and let $F^{(m)}$ be the corresponding subset of F. The new support point z_m is selected as the point $z \in Z^{(m-1)}$ that maximizes the residual $|f(z) - r(z)|$, with $r(z)$ as in step $m - 1$.

The recomputation of the weights that minimizes the approximation error on Z can be obtained as the solution of a least-squares problem. First of all, we actually want to minimize the error on $Z^{(m)}$, instead of Z, for two reasons: the rational approximation is exact on $z_1, \ldots, z_m$, which are in general interpolation points (so no error to minimize there), and excluding them from the computation is convenient as it avoids us having to deal with the poles of n and d. Our aim is an approximation

$$f(z) \approx \frac{n(z)}{d(z)}, \quad z \in Z^{(m)}$$

which in linearized form becomes

$$f(z)d(z) \approx n(z), \quad z \in Z^{(m)}$$

This translates to the least-squares problem

$$\text{minimize} \|fd - n\|_{Z^{(m)}}, \quad \|w\|_m = 1$$

where $\|\cdot\|_{Z^{(m)}}$ is the discrete 2-norm over $Z^{(m)}$ and $\|\cdot\|_m$ is the discrete 2-norm over m-vectors. To ensure that the problem makes sense, we assume that $Z^{(m)}$ has at least m points, i.e. $m \leq M/2$. Regarding Z, F, and w as column vectors, we seek a normalized $w = (w_1, \ldots, w_m)^T$ that minimizes the 2-norm of the $(M - m)$-vector whose i-th component is

$$d(Z_i^{(m)})F_i^{(m)} - n(Z_i^{(m)}) = \sum_{j=1}^{m} \frac{w_j F_i^{(m)}}{Z_i^{(m)} - z_j} - \sum_{j=1}^{m} \frac{w_j f_j}{Z_i^{(m)} - z_j} = \sum_{j=1}^{m} \frac{w_j (F_i^{(m)} - f_j)}{Z_i^{(m)} - z_j}$$

The problem can be written in matrix form as

$$\text{minimize} \|A^{(m)} w\|_{M-m}, \quad \|w\|_m = 1 \tag{4.4}$$

where $A^{(m)}$ is the $(M - m) \times m$ matrix

$$A^{(m)} = \begin{bmatrix} \dfrac{F_1^{(m)} - f_1}{Z_1^{(m)} - z_1} & \cdots & \dfrac{F_1^{(m)} - f_m}{Z_1^{(m)} - z_m} \\ \vdots & \ddots & \vdots \\ \dfrac{F_{M-m}^{(m)} - f_1}{Z_{M-m}^{(m)} - z_1} & \cdots & \dfrac{F_{M-m}^{(m)} - f_m}{Z_{M-m}^{(m)} - z_m} \end{bmatrix}$$

(4.4) will be solved using the singular value decomposition (SVD), taking w as the final right singular vector in the reduced SVD $A^{(m)} = U\Sigma V^*$. Note that the minimal singular value of $A^{(m)}$ might be nonunique or nearly so, but the algorithm does not rely on its uniqueness.

It is convenient for the computation to use the $(M-m)\times m$ *Cauchy matrix*

$$C = \begin{bmatrix} \dfrac{1}{Z_1^{(m)} - z_1} & \cdots & \dfrac{1}{Z_1^{(m)} - z_m} \\ \vdots & \ddots & \vdots \\ \dfrac{1}{Z_{M-m}^{(m)} - z_1} & \cdots & \dfrac{1}{Z_{M-m}^{(m)} - z_m} \end{bmatrix}$$

whose columns define the basis in which we approximate. Let the scaling matrices $S_F = \operatorname{diag}(F^{(m)})$ and $S_f = \operatorname{diag}(f_1,\ldots,f_m)$. $A^{(m)}$ can then be obtained from C as $A^{(m)} = S_F C - C S_f$. Once w is found, we can compute $N := n(Z^{(m)}) = C(w_1 f_1, \ldots, w_m f_m)^T$ and $D := d(Z^{(m)}) = Cw$. From there, $R := r(Z^{(m)}) = N/D$ (component-wise division).

It remains to specify how the algorithm is initialized and terminated. The algorithm starts with iteration 0, where no support points have been selected and r is the constant function with value mean(F). The procedure terminates when the nonlinear residual $f(z) - n(z)/d(z)$ is sufficiently small (on our domain Z, of course). AAA's authors have found it effective to use a default tolerance of 10^{-13} relative to the maximum of $|f(Z)|$. When the tolerance is too tight, the approximation stagnates and numerical Froissart doublets appear (more on this in Section 4.4).

Once the rational approximant has been found, the zeros of d, which are (generically) the poles of r and (hopefully) the poles of f, can be found by solving an $(m+1)\times(m+1)$ generalized eigenvalue problem, according to [19],

$$\begin{bmatrix} 0 & w_1 & w_2 & \cdots & w_m \\ 1 & z_1 & & & \\ 1 & & z_2 & & \\ \vdots & & & \ddots & \\ 1 & & & & z_m \end{bmatrix} = \lambda \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

At least two of the eigenvalues of this problem are infinite and the remaining $m-1$ are the zeros of d. The zeros of n can be computed in a similar fashion, replacing w_j with $w_j f_j$.

We now list a couple of properties of the core AAA algorithm, from [22, Proposition 3.1]. The statements refer to AAA approximants at step m and they hold in exact arithmetic for any selection of the support points among Z.

Affineness in f. For any $a \neq 0$ and b, $ar(z) + b$ is an approximant of $af(z) + b$ on Z.

Affineness in z. For any $a \neq 0$ and b, $r(az + b)$ is an approximant of $f(az + b)$ on $(Z - b)/a$.

Monotonicity. The linearized residual norm $\sigma_{\min}(A^{(m)}) = \|fd - n\|_{Z^{(m)}}$ is a nonincreasing function of m (follows from the fact that $A^{(m)}$ is obtained from $A^{(m-1)}$ by deleting one row and appending one column).

The AAA algorithm has a complexity of $O(Mm^3)$ flops, determined by the SVD operations of size $(M - j) \times j$ for $j = 1, \dots, m$. This is usually modest since m is small for most applications.

4.3 Julia Implementation

Our implementation is inspired by both the MATLAB code in the original paper and Chebfun's implementation (`https://github.com/chebfun/chebfun/blob/development/aaa.m`). Follows a simplified version of the code, that omits some checks and preprocessing of the input parameters. The full implementation is at `https://github.com/robzan8/MatFun.jl/blob/master/src/aaa.jl`.

```
#= "The AAA algorithm for rational approximation"
    (Yuji Nakatsukasa, Olivier Sete, Lloyd N. Trefethen).
=#
function aaa(F::Vector{N}, Z::Vector{N}, tol::Float64=1e-13, mmax::
    Int64=100) where {N<:Union{Float64, Complex128}}

    # initialization:
    abstol = tol*norm(F, Inf)
    J = collect(1:M) # indices of the non-support points
    z = Vector{N}(0) # support points
    f = Vector{N}(0) # corresponding data values
    w = Vector{N}(0) # weights
    C = Matrix{N}(M, 0) # Cauchy matrix
    errvec = real(Vector{N}(0))
    R = fill(mean(F), M)

    # main loop:
    for m = 1:mmax
        # Select next support point where error is largest:
        j = indmax(abs.(F - R))
        z = [z; Z[j]]
        f = [f; F[j]]
        J = J[J .!= j]
        C = [C 1./(Z .- Z[j])]

        # Compute weights:
        A = F.*C - C.*f.'
        V = svd(A[J,:], thin=true)[3]
        w = V[:,m]

        # Rational approximant on Z:
        num, den = C*(w.*f), C*w
        R = copy(F)
        R[J] = num[J]./den[J]

        err = norm(F - R, Inf)
        errvec = [errvec; err]
        if err <= abstol
            break
        end
    end


    # Remove support points with zero weight:
    keep = w .!= 0
    z = z[keep]
    f = f[keep]
    w = w[keep]

    # Construct function handle:
    r = (x) -> reval(z, f, w, x)

    # Compute poles and zeros via generalized eigenvalues:
    m = length(w)
    B = eye(N, m+1)
    B[1,1] = 0
    pol = eigvals([0 w.'; ones(N, m, 1) diagm(z)], B)
    zer = eigvals([0 (w.*f).'; ones(N, m, 1) diagm(z)], B)
    #=
    Note: some Inf poles can come up as NaN,
    since in Julia (1.+2im)/(0.+0im) is NaN+NaN*im
    (https://github.com/JuliaLang/julia/issues/9790).
    =#
    pol = pol[isfinite.(pol)]
    zer = zer[isfinite.(zer)]

    # Compute residues via discretized Cauchy integral:
    dz = (1e-5)*exp.(2im*pi*collect(1:4)/4)
    res = r(pol .+ dz.')*(dz/4)

    # We don't remove numerical Froissart doublets,
    # which are rare anyway if aaa is used correctly.

    return r, pol, res, zer, z, f, w, errvec
end # function aaa
```

Once the rational approximant r has been obtained and defined as its support points, data values and weights `z`, `f` and `w`, it can be evaluated at a set of points `x` with the following procedure.

```
#=
Evaluate rational function in barycentric form.
=#
function reval(z::Vector{N}, f::Vector{N}, w::Vector{N}, x::Vector{X
    }) where {N<:Number, X<:Number}
    C = 1./(x .- z.') # Cauchy matrix
    r = (C*(w.*f))./(C*w) # result

    # Deal with input Inf as limit:
    r[isinf.(x)] = sum(w.*f)/sum(w)

    # Force interpolation at support points, to avoid Inf/Inf:
    for j = 1:length(x)
        i = findfirst(x[j] .== z)
        if i != 0
            r[j] = f[i]
        end
    end
    return r
end
function reval(z::Vector{N}, f::Vector{N}, w::Vector{N}, x::X) where
     {N<:Number, X<:Number}
    return reval(z, f, w, [x])[1]
end
function reval(z::Vector{N}, f::Vector{N}, w::Vector{N}, A::Array{X
    }) where {N<:Number, X<:Number}
    return reshape(reval(z, f, w, A[:]), size(A))
end
```

4.4 Froissart Doublets

An issue we haven't touched upon yet is that of *spurious poles*, also called *Froissart doublets*. They are poles with very small residues or pole-zero pairs so close together as to nearly cancel [13]. The problem arises in exact arithmetic, tied to the fact that the problem of analytic continuation is ill-posed, and more often in floating point arithmetic. *Numerical froissart doublets* can be regarded as poles with residues on the order of machine precision.

The AAA algorithm typically produces no Froissart doublets when the function f to approximate is analytic or smooth enough and when the tolerance parameter is set correctly. It is possible to make the procedure generate spurious poles by using a tolerance on the order of machine precision. Even when such artifacts appear, they can usually be identified and removed by one further solution of a least squares problem, as the original paper points out [22, Section 5]. We didn't implement the extra `cleanup` step, as Froissart doublets did not arise in practice in our experiments.

4.5 Effectiveness of AAA

The AAA algorithm does not claim to achieve optimality in any particular norm such as L^2 or L^∞. However, it works effectively in practice. AAA's authors have tested the algorithm with 9 different applications, including the approximation of an analytic f in the unit disk, approximation of $|x|$ on [-1, 1], approximation of $\exp(x)$ on $(-\infty, 0)$ and approximation in connected and disconnected domains [22, Section 6]. In [22, Section 11], AAA is compared to the other most competitive methods: vector fitting and RKFIT (rational Krylov fitting). AAA comes out as the clear winner. Vector fitting and RKFIT use a set number of poles, require an initial guess of the poles' location and are in general slower, less stable and more difficult to use.

In the context of our work, AAA is used to find the poles of a function f. An example of the application of the algorithm to this problem, from the AAA paper, is in Figure 4.1.

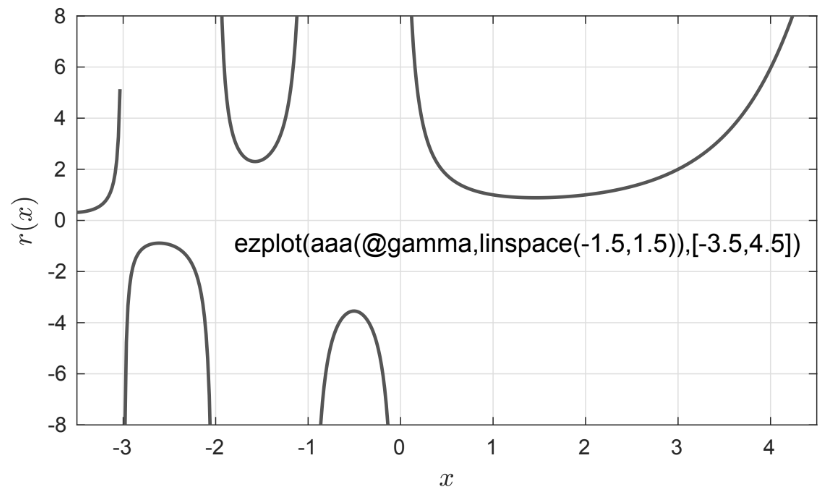

Figure 4.1: Approximation of the gamma function produced by the MATLAB code shown, from [22]. This is an approximant of type (9, 9), whose first poles match 0, -1, -2 and -3 to 15, 15, 7 and 3 digits, respectively (note that -2 and -3 are outside the domain of approximation).

Another experiment, included in our test suite for the Julia package, constructs a pseudo-random rational function with known poles, samples it and then retrieves the poles with AAA.

```julia
srand(666)
poles = randn(10) + im*randn(10)
residues = randn(10) + im*randn(10)
func = (x) -> sum(residues./(x .- poles))
rad = norm(poles, Inf)
Z = (linspace(-rad, rad, 30)' .+ im*linspace(-rad, rad, 30))[:]
F = func.(Z)
r, pol, res, zer, z, f, w, errvec = aaa(F, Z)
poles = poles[sortperm(real.(poles))]
pol = pol[sortperm(real.(pol))]
err = abs.(pol - poles)./abs.(poles)
println(err)
```

The code above outputs

```
[
3.52891e-16
5.42367e-16
2.18991e-15
6.80792e-16
8.95138e-16
3.45687e-16
6.49912e-16
7.92527e-16
1.12846e-15
1.31942e-15
]
```

showing how effective AAA can be at retrieving a function’s poles.

CHAPTER

5 RATIONAL KRYLOV + AAA FOR SPARSE MATRIX FUNCTIONS

In this chapter, we introduce some theory on rational Krylov spaces and describe how the rational Arnoldi algorithm can be used (in conjunction with AAA) for approximating $f(A)b$. The mathematics on rational Krylov methods is as presented by Güttel in [14].

5.1 Rational Krylov Spaces and the Rational Arnoldi Method

Rational Krylov methods for computing $f(A)b$ provide an approximation of the form $r_m(A)b$ where $r_m = p_{m-1}/q_{m-1}$ is a rational function of type $(m-1, m-1)$. We will assume that the denominator is factored as

$$q_{m-1}(z) = \prod_{j=1}^{m-1} (1 - z/\xi_j) \tag{5.1}$$

The poles of r_m $\xi_1, \ldots, \xi_{m-1}$ are numbers in the extended complex plane $\bar{\mathbb{C}} := \mathbb{C} \cup \{\infty\}$ different from all the eigenvalues of A and 0 (the exclusion of the pole 0 is not essential, as it can be avoided by shifting z). Note that

q_{m-1} is fully determined by the poles ξ_j and the poles are provided as input to the methods by the user, so q_{m-1} is fixed in advance.

The mth *rational Krylov space* associated with $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$ and q_{m-1} is defined as

$$\mathcal{Q}_m(A,b) := q_{m-1}(A)^{-1}\text{span}\{b, Ab, \dots, A^{m-1}b\}$$

which can also be written as

$$\mathcal{Q}_m(A,b) := q_{m-1}(A)^{-1}\text{span}\{p_{m-1}(A)b : p_{m-1} \text{ is a polynomial of degree} \leq m-1\}$$

By assumption (5.1) on the denominators q_{m-1}, rational Krylov spaces are nested and have strictly increasing dimension until some invariance index $M \leq n$ is reached:

$$\mathcal{Q}_1(A,b) \subset \mathcal{Q}_2(A,b) \subset \cdots \subset \mathcal{Q}_M(A,b) = \mathcal{Q}_{M+1}(A,b) = \dots$$

When all the poles ξ_j are set to infinity, $q_{m-1} \equiv 1$ and the rational Krylov space $\mathcal{Q}_m(A,b)$ reduces to a "regular" (polynomial) Krylov space $\mathcal{K}_m(A,b)$.

An orthonormal basis for $\mathcal{Q}_m(A,b)$ can be computed with Ruhe's *rational Arnoldi algorithm* [24]. Being a generalization of the Arnoldi process, the algorithm works in an analogous fashion. The first vector is $v_1 = b/\|b\|$ (basis of $\mathcal{Q}_1(A,b)$). Vector v_{j+1} is obtained by orthonormalizing $(I - A/\xi_j)^{-1}Av_j$ against the previously computed vectors $v_1, \dots, v_j$.
$(I - A/\xi_j)^{-1}Av_j \in \mathcal{Q}_{j+1}(A,b)$ implies

$$(I - A/\xi_j)^{-1}Av_j = \sum_{i=1}^{j+1} v_i h_{i,j}$$

and the algorithm at step m produces the factorization

$$AV_m(I + H_mD_m) + Av_{m+1}h_{m+1,m}\xi_m^{-1}e_m^T = V_mH_m + v_{m+1}h_{m+1,m}e_m^T \tag{5.2}$$

where $V_m = [v_1, \dots, v_m] \in \mathbb{C}^{n\times m}$ is unitary, $H_m = (h_{i,j}) \in \mathbb{C}^{m\times m}$ is upper Hessenberg and $D_m = \text{diag}(\xi_1^{-1}, \dots, \xi_m^{-1})$. In case of breakdown, (5.2) becomes

$$AV_m(I + H_mD_m) = V_mH_m$$

(5.2) can also be reformulated more succinctly as

$$AV_{m+1}\underline{K_m} = V_{m+1}\underline{H_m} \tag{5.3}$$

where

$$\underline{H_m} := \begin{bmatrix} H_m \\ h_{m+1,m}e_m^T \end{bmatrix}, \quad \underline{K_m} := \begin{bmatrix} I + H_mD_m \\ h_{m+1,m}\xi_m^{-1}e_m^T \end{bmatrix}$$

V_m, the orthonormal basis of $\mathcal{Q}_m(A, b)$, can be used to approximate $f(A)b$. The *rational Arnoldi approximation* of $f(A)b$ from $\mathcal{Q}_m(A, b)$ is defined as

$$f_m^{RA} := V_m f(A_m) V_m^* b \tag{5.4}$$

where $A_m := V_m^* A V_m \in \mathbb{C}^{m\times m}$ is the orthogonal projection of A into the rational Krylov space (also referred to as *compression* of A). The advantage of the rational Arnoldi approximation over its polynomial counterpart rests on the fact that f_m^{RA} can be a good approximation even for small m. It is possible to avoid the explicit projection when computing $A_m = V_m^* A V_m$. When the last pole ξ_m is set to infinity, K_m is invertible and $A_m = H_m K_m^{-1}$ (from Equation 5.3).

We'll now show a couple of key properties of f_m^{RA}. First, the approximation is exact when f is a rational function represented in the Krylov space $\mathcal{Q}_m(A, b)$, that is, f has type $(m-1, m-1)$ and q_{m-1} as denominator. This property holds for polynomial Arnoldi approximations and generalizes to the rational case.

Theorem 6 (Exactness)**.** [14, Lemma 3.1] *Let $V_m \in \mathbb{C}^{n\times m}$ be an orthonormal basis of $\mathcal{Q}_m(A, b)$ and let $A_m = V_m^* A V_m$. Then for any rational function $\tilde{r}_m \in \mathcal{P}_m/q_{m-1}$ we have*

$$(V_m V_m^*)\tilde{r}_m(A)b = V_m \tilde{r}_m(A_m) V_M^* b$$

provided that $\tilde{r}_m(A_m)$ is defined. In particular, if $r_m \in \mathcal{P}_{m-1}/q_{m-1}$, then

$$r_m(A)b = V_m r_m(A_m) V_m^* b$$

i.e., the rational Arnoldi approximation for $r_m(A)b$ is exact.

Proof. Define $q = q_{m-1}(A)^{-1}b$. We first show by induction that

$$(V_m V_m^*)A^j q = V_m A_m^j V_m^* q \quad \forall j \in \{0, \ldots, m\}. \tag{5.5}$$

Assertion (5.5) is obviously true for $j = 0$. Assume that it is true for some $j < m$. Then by the definition of a rational Krylov space we have $V_m V_m^* A^j q = A^j q$ and therefore

$$(V_m V_m^*)A^{j+1} q = (V_m V_m^*) A V_m V_m^* A^j q = (V_m V_m^*) A V_m A_m^j V_m^* q = V_m A_m^{j+1} V_m^* q$$

which establishes (5.5). Again from (5.5) we obtain by linearity

$$b = q_{m-1}(A)q = V_m q_{m-1}(A_m) V_m^* q$$

or, equivalently, $V_m^* q = q_{m-1}(A_m)^{-1} V_m^* b$. Replacing $V_m^* q$ in (5.5) completes the proof. □

The *numerical range* of a square matrix A is defined as

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

It allows to bound the norm of $f(A)$. By a theorem of Crouzeix [10], there exists a universal constant $C \leq 11.08$ such that

$$\|f(A)\| \leq C\|f\|_\Sigma, \tag{5.6}$$

where the norm on the right is the maximum norm on a compact set Σ that encloses $\mathbb{W}(A)$. With this inequality it is possible to bound the norm of the error of a rational Arnoldi approximation.

Theorem 7 (Near-optimality)**.** [14, Corollary 3.4] *Let f be analytic in a neighborhood of a compact set $\Sigma \supseteq \mathbb{W}(A)$. Then the rational Arnoldi approximation f_m^{RA} defined by (5.4) satisfies*

$$\|f(A)b - f_m^{RA}\| \leq 2C\|b\| \min_{r_m \in \mathcal{P}_{m-1}/q_{m-1}} \|f - r_m\|_\Sigma$$

with a constant $C \leq 11.08$. If A is Hermitian, the result holds even with $C = 1$ and $\Sigma \supseteq \Lambda(A) \cup \Lambda(A_m)$.

Proof. By theorem 6 we know that $r_m(A)b = V_m r_m(A_m)V_m^* b$ for every rational function $r_m \in \mathcal{P}_{m-1}/q_{m-1}$. Thus,

$$\begin{aligned}\|f(A)b - f_m^{RA}\| &= \|f(A)b - V_m f(A_m)V_m^* b - r_m(A)b + V_m r_m(A_m)V_m^* b\| \\ &\leq \|b\|(\|f(A) - r_m(A)\| + \|f(A_m) - r_m(A_m)\|) \\ &\leq 2C\|b\| \cdot \|f - r_m\|_\Sigma\end{aligned}$$

where (5.6) has been used for the last inequality. If A is Hermitian so is A_m and (5.6) holds with $C = 1$ and $\Sigma \supseteq \Lambda(A)$. The proof is completed by taking the infimum over all $r_m \in \mathcal{P}_{m-1}/q_{m-1}$ and noting that this infimum is attained on the compact set Σ. $\square$

In practice, f_m^{RA} is often very close to the orthogonal projection of $f(A)b$ onto the search space $\mathcal{Q}_m(A, b)$, much better than what predicted by the bound in Theorem 7.

For more properties and insight on rational Arnoldi approximations, see [14, Section 3.1].

5.2 Implementation and Combination with AAA

The rational Arnoldi algorithm has been ported from Güttel's Rational Krylov Toolbox for MATLAB version 1.0 (`http://guettel.com/rktoolbox/`). The algorithm works in real arithmetic when A and b are real and the poles that aren't real come in complex conjugate pairs. We slightly modified the control flow of the procedure to make it type-stable (basically, Julia is able to produce code that is more efficient when variables don't change type during runtime; see `https://docs.julialang.org/en/stable/manual/performance-tips/#Write-%22type-stable%22-functions-1`). The code is too long to be included here, it can be found at `https://github.com/robzan8/MatFun.jl/blob/master/src/ratkrylov.jl`.

To compute $f(A)b$ with poles specified by the user. all we have to do is launch the rational Arnoldi algorithm and project A into the Krylov space, as shown in the code below.

```
#= Computes f(A)*b using the rational Arnoldi approximation with
    poles p.
=#
function ratkrylov(f::Func, A::Mat, b::Vector{N}, p::Vector{
    Complex128}) where {
    Func, N<:Union{Float64, Complex128}, Mat<:Union{Matrix{N},
        SparseMatrixCSC{N}}}

    V, K, H = ratkrylov(A, b, p)

    m = size(V, 2) - 1 # may be < length(p) in case of breakdown
    Am = isinf(p[m]) ? [H/K[1:m,1:m] V'*(A*V[:,end])] : V'*A*V

    return V*(schurparlett(f, Am)*(V'*b))
end
```

In most situations, though, the user may want the poles of f to be computed automatically by AAA. In this case, instead of the poles, the parameters for calling AAA (`mmax`, `tol` and `Z`) can be optionally specified:

```julia
#= Computes f(A)*b using the rational Arnoldi approximation. The
    poles of f are found automatically by AAA, with parameters mmax,
    tol and Z. You typically want to manually set at least mmax, to
    limit the size of the resulting Krylov space. If the sample set Z
     is not provided, f will be sampled on the 0-centered disk with
    radius min(norm(A, 1), norm(A, Inf), vecnorm(A)). When A is Real,
     the algorithm will assume f(conj(x)) == conj(f(x)).
=#
function ratkrylov(f::Func, A::Mat, b::Vector{N}, mmax::Int64=100,
    tol::Float64=1e-13, Z::Vector{M}=Vector{Complex128}(0)) where {
    M<:Union{Float64, Complex128}, N<:Union{Float64, Complex128},
    Func, Mat<:Union{Matrix{N}, SparseMatrixCSC{N}}}

    if length(Z) == 0
        rad = min(norm(A, 1), norm(A, Inf), vecnorm(A))
        rad = max(rad, sqrt(eps()))
        # The number of samples attempts to balance computation times
             between AAA and rational Arnoldi.
        nsamples = Mat<:SparseMatrixCSC ? nnz(A) : prod(size(A))
        nsamples = max(nsamples, 100)
        Z = lpdisk(rad, nsamples)
    end

    pol = Vector{Complex128}(aaa(f, Z, tol, mmax)[2])
    if N <: Real
        # Real ratkrylov wants conjugated and canonically ordered
            poles.
        pos = pol[imag.(pol) .> 1e-13]
        pol = pol[abs.(imag.(pol)) .<= 1e-13]
        pol[1:end] = real.(pol)
        for i = 1:length(pos)
            pol = [pol; pos[i]; conj(pos[i])]
        end
    end
    # Inf as last pole allows for faster projection of A in the
        Krylov space:
    pol = [pol; Inf]

    return ratkrylov(f, A, b, pol)
end
```

The algorithm is able to approximate $f(A)b$ with a good accuracy and a small number of poles, in most cases, as we'll see in the next section. The disadvantage of the rational Arnoldi method, with respect to its polynomial counterpart, is that it has to solve a sparse linear system at each iteration. The system is solved with an LU factorization (in our case, the multifrontal LU factorization `umf_lufact` from UMFPACK, `http://faculty.cse.tamu.edu/davis/suitesparse.html`). The cost of the factorization depends on the sparsity of the result. In the best case scenario it can be as fast as a sparse matrix-vector multiplication, worst case the sparsity is completely destroyed and the system can go out of resources, typically memory.

5.3 Numerical Experiments

For the experiments we used matrices from the SuiteSparse Matrix Collection (`https://sparse.tamu.edu/`). The size of the matrices is about 1000×1000. This size allows to experiment with approximation degrees between 10 and 100, without the matrices being represented exactly in the Krylov space. Still, the size is small enough for using dense methods as oracles.

5.3.1 Experiment 1

This experiment approximates $e^A b$ with `ratkrylov(exp, A, b, mmax)` for various matrices. The matrices have been selected randomly from different application areas and do not share particular characteristics. b is the vector of ones. The relative condition number for the problem e^A has been estimated with `funm_condest1` from Higham's Matrix Function Toolbox. The relative error has been evaluated against `expm(full(A))*b`. We avoided using Schur-Parlett as oracle, because our rational Arnoldi method makes use of it internally for computing $f(A_m)$ of the projected matrix. The Symbolic Math Toolbox was not an option, as the matrices are too large. The results are in Table 5.1, be aware that `expm` itself is subject to errors while interpreting the results.

Problem type	name	cond(A)	cond (exp, A)	# poles	error	# poles	error
2D/3D problem	jagmesh3	1168	7.0e0	9	1.5e-6	15	6.0e-14
Fluid dynamics	sherman 4	2178	1.2e2	11	6.6e-6	19	6.7e-10
Structural problem	can_ 1072	2.0e34	3.4e1	11	7.3e-6	23	1.7e-14
Directed graph	SmaGri	Inf	7.8e1	9	2.6e-7	21	7.3e-13
Semicond device	jpwh_ 991	142	2.9e1	11	1.4e-8	21	3.4e-15
Thermal problem	lshp1009	2366	7.0e0	9	4.2e-8	15	3.5e-14

Table 5.1: Computation of $e^A b$ for various As with our rational Arnoldi + AAA method. The relative error is computed against `expm(full(A))*b`.

The experiment shows that very good accuracy, close to what is imposed by the condition number, is achieved in most cases with a small number of poles.

Similar results are obtained with other analytic functions as the matrix sine and cosine. For example, with matrix "jagmesh3", `ratkrylov` approximates $\cos(A)b$ with 19 poles and a precision of 10^{-14} (compared to `schurparlett(cos, full(A))`).

5.3.2 Experiment 2

Experiment 2 is set up as the previous one, but is designed to be more challenging for rational Arnoldi and AAA. The action of the principal square root $\sqrt{A}b$ is approximated as `ratkrylov(sqrt, A, b, mmax)` and compared against `sqrtm(full(A))*b`. The input matrices are positive definite matrices (from the same database), to avoid having to deal with negative real eigenvalues. The function should be more difficult to approximate rationally, as it has a branch cut on the negative real axis. Also, the problem is ill-conditioned when some eigenvalue of A approaches 0, but the fact that the matrices are normal should limit this kind of instability. `funm_condest1` was not able to estimate the condition number correctly for this experiment, returning always 10^8. Instead of the condition number, Table 5.2 lists the smallest eigenvalue of A and the radius of the disk on which the function is sampled, i.e. `min(norm(A, 1), norm(A, Inf), vecnorm(A))`, for reasons that will be clear in a moment.

Problem type	name	min eig(A)	norm(A)	# poles	error	# poles	error
Circuit simulatio	rajat19	1.7e-1	3.9e1	20	5.8e-6	45	3.7e-14
2D/3D problem	gr_30_30	6.1e-2	1.6e1	20	4.5e-8	30	2.8e-14
Structural problem	nos3	1.8e-2	7.7e2	50	8.8e-4	100	5.2e-9
Power network	685_bus	6.2e-2	2.6e4	50	1.5e-3	95	8.0e-5
Electro magnetic	mhdb416	5.5e-10	2.5e0	50	1.3e-3	79	5.3e-4
Electro magnetic	mhd 1280b	1.5e-11	8.0e1	50	2.5e-2	50	8.8e-3

Table 5.2: Computation of $\sqrt{A}b$ for various As with our rational Arnoldi + AAA method. The relative error is computed against `sqrtm(full(A))*b`.

The table shows that, in general, this experiment achieved worse accuracy despite the use of a larger approximation space. The rows of the table have been sorted from the smallest error to the biggest. The data suggests that larger errors can be predicted by a larger difference in magnitude between the smallest eigenvalue of A and the norm of the matrix. This seems indeed to be the cause of the problem. When A has a large norm, the samples are scattered on a large disk and AAA is not able to accurately capture the behavior of `sqrt` near 0, where the function is most sensitive. If the same matrix happens to have small eigenvalues, trouble occurs. We confirmed this hypothesis with further experimentation.

`ratkrylov` optionally accepts Z, the set of samples, as user input. We constructed Z with a bigger sample density near 0, as union of two sample sets: 500 samples evenly distributed on the disk with radius `norm(A)` and 500 even samples on the disk with radius 1. With such Z, the approximation of degree 50 for "685_bus" goes from an error of 10^{-3} to 10^{-12}. Similar results can be obtained for the other matrices in Table 5.2.

This experiment demonstrates that sometimes the fully automatic method is not good enough and additional information about the function or the eigenvalues of A may be required for accurate results. The possibility to manually specify Z has been included for this reason. The feature is easy to use, compared to the manual selection of the approximation poles and yet it can provide an advantage in case it is known that the function has particular features or the eigenvalues lie in a specific region of the complex plane.

CHAPTER

6 CONCLUSIONS AND FUTURE WORK

We solved the problem that we set out to solve.

The fully automatic rational Arnoldi + AAA method works surprisingly effectively for analytic functions. For functions with singularities or other peculiar features, the manual selection of the samples for AAA is easy to use and leads to accurate results. An optimal automatic selection of the sample set could be studied and implemented for known functions such as the square root and logarithm.

The Schur-Parlett algorithm works effectively with no user input, but there is room for improvements. A peculiar characteristic of the algorithm is that is allows for different methods of f's evaluation on the atomic blocks. The methods can have a cost of kn^3 with relatively big k, as can happen with the Taylor series evaluation, and the cost is masked by the fact that the atomic blocks are typically smaller than the whole matrix. This opens up the possibilities for a variety of algorithms to be used. One that strikes as particularly interesting is the use of a AAA approximation. AAA for the atomic blocks could solve some of the instabilities of Schur-Parlett (namely the "snake eigenvalues" problem), but, probably more importantly, it would lift Schur-Parlett's dependency on automatic differentiation or other forms of derivatives. This would enable a fully automatic Schur-Parlett implementation in languages such as MATLAB, C and Fortran (which would in turn also enable the implementation of an automatic Arnoldi method on said languages).

More research is required for exploring this and other ideas.

BIBLIOGRAPHY

[1] Awad H. Al-Mohy and Nicholas J. Higham. Improved inverse scaling and squaring algorithms for the matrix logarithm. *SIAM Journal on Scientific Computing*, 34(4):C153–C169, 2012.

[2] A. Antoulas. *Approximation of Large-Scale Dynamical Systems*. Society for Industrial and Applied Mathematics, 2005.

[3] A. C. Antoulas and B. D. Q. Anderson. On the scalar rational interpolation problem. *IMA Journal of Mathematical Control and Information*, 3(2-3):61–88, 1986.

[4] Karl Johan Astrom and Bjorn Wittenmark. *Computer-Controlled Systems: Theory and Design (3rd Edition)*. Prentice Hall, 1996.

[5] A.H. Barnett and T. Betcke. Stability and convergence of the method of fundamental solutions for helmholtz problems on analytic domains. *Journal of Computational Physics*, 227(14):7003 – 7026, 2008.

[6] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[7] Ake Bjorck and Sven Hammarling. A schur method for the square root of a matrix. *Linear Algebra and its Applications*, 52-53:127 – 140, 1983.

[8] Brandan A. Borgias, Miriam Gochin, Deborah J. Kerwood, and Thomas L. James. Relaxation matrix analysis of 2d nmr data. *Progress in Nuclear Magnetic Resonance Spectroscopy*, 22(1):83 – 100, 1990.

[9] S.M. Cox and P.C. Matthews. Exponential time differencing for stiff systems. *Journal of Computational Physics*, 176(2):430 – 455, 2002.

[10] Michel Crouzeix. Numerical range and functional calculus in hilbert space. *Journal of Functional Analysis*, 244(2):668 – 690, 2007.

[11] Philip I. Davies and Nicholas J. Higham. A schur-parlett algorithm for computing matrix functions. *SIAM Journal on Matrix Analysis and Applications*, 25(2):464–485, 2003.

[12] B. Fornberg and N. Flyer. *A Primer on Radial Basis Functions with Applications to the Geosciences.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2015.

[13] Jacek Gilewicz and Yu Kryakin. Froissart doublets in padé approximation in the case of polynomial noise. 153:235–242, 04 2003.

[14] Stefan Güttel. Rational krylov approximation of matrix functions: Numerical methods and optimal pole selection. 36, 08 2013.

[15] Nicholas J. Higham. *Functions of Matrices: Theory and Computation (Other Titles in Applied Mathematics)*. Society for Industrial & Applied Mathematics,U.S., 2008.

[16] Nicholas J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM Review*, 51(4):747–764, 2009.

[17] Arieh Iserles, Hans Munthe-Kaas, Syvert Norsett, and A Zanna. Lie group methods. 9:215–, 01 2000.

[18] B. Kagström. Numerical computation of matrix functions. *Report UMINF-58.77, Department of Information Processing, University of Umea, Sweden*, 1977.

[19] G. Klein. Applications of linear barycentric rational interpolation. *Thesis, University of Fribourg*, 2012.

[20] Andreas Klockner, Alexander Barnett, Leslie Greengard, and Michael O'Neil. Quadrature by expansion: A new method for the evaluation of layer potentials. *Journal of Computational Physics*, 252:332 – 349, 2013.

[21] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[22] Y. Nakatsukasa, O. Sète, and L. N. Trefethen. The AAA algorithm for rational approximation. *ArXiv e-prints*, December 2016.

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

[24] Axel Ruhe. Rational krylov algorithms for nonsymmetric eigenvalue problems. ii. matrix pairs. *Linear Algebra and its Applications*, 197-198:283 – 295, 1994.