Midterm Project

LI,YING-SHENG
Introduction to Scientific Computing

November 2, 2025

1 Introduction and Problem Statement

1.1 Introduction

Rational approximations of real or complex functions play an essential role in numerical analysis and scientific computing. They are mainly used in two kinds of applications: (i) to provide compact representations of complicated functions, and (ii) to perform extrapolation beyond the sampled region. Compared with polynomial approximations, rational functions can often achieve high accuracy with fewer parameters and are more suitable for representing functions with singularities or rapid variations.

Despite these advantages, constructing reliable rational approximations is not straightforward. A major numerical issue is the occurrence of *spurious poles*, also known as *Froissart doublets*. In theory, such poles and zeros nearly cancel each other, but in floating-point arithmetic, they appear as numerical artifacts with residues on the order of machine precision. These so-called *numerical Froissart doublets* can cause instability and reduce the quality of the approximation.

1.2 Problem Statement

The AAA (Adaptive Antoulas–Anderson) algorithm, introduced by Nakatsukasa, Sète, and Trefethen (2018), addresses the challenge of constructing accurate and stable rational approximations in an adaptive way. The main idea is to automatically select a set of *support points* where interpolation is enforced, with the goal of minimizing the approximation error at all other sample points.

In the AAA algorithm, the rational approximant is expressed in the *barycentric form*:

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

where $m \geq 1$ is an integer, $\{z_1, \ldots, z_m\}$ are distinct real or complex support points, $\{f_1, \ldots, f_m\}$ are the corresponding function values, and $\{w_1, \ldots, w_m\}$ are barycentric weights.

However, even in this barycentric representation, numerical Froissart doublets may still appear. Specifically, when $d(z_j) = 0$ for some z_j, and simultaneously $n(z_j) \approx 0$, the pole is theoretically canceled by the numerator. In finite precision arithmetic, this near-cancellation introduces small perturbations that make the approximation numerically unstable.

To overcome this, the AAA algorithm includes a refinement step that detects and removes such spurious support points. By eliminating these nearly canceled pole–zero pairs, the algorithm maintains numerical stability and ensures that the final rational approximant achieves high accuracy with minimal artifacts.

2 Method Explanation

The AAA (Adaptive Antoulas–Anderson) algorithm constructs a rational approximation of a function $f(z)$ in an adaptive and numerically stable way. Its foundation lies in the *barycentric rational formula*:

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

where $\{z_1, \ldots, z_m\}$ are distinct support points, $f_j = f(z_j)$, and w_j are barycentric weights.

2.1 Reformulation and Degrees of Freedom

To better understand the structure of $r(z)$, we multiply both numerator and denominator by

$$\ell(z) = \prod_{j=1}^{m} (z - z_j),$$

which yields

$$r(z) = \frac{n(z)}{d(z)} = \frac{p(z)/\ell(z)}{q(z)/\ell(z)} = \frac{p(z)}{q(z)},$$

where $p(z)$ and $q(z)$ are polynomials of degree at most $m - 1$. This shows that the rational function $r(z)$ has $2m - 1$ degrees of freedom: roughly half are determined by interpolation at the support points $\{z_1, \ldots, z_m\}$, while the remaining $m - 1$ degrees of freedom are used to minimize the approximation error on the remaining sample points $Z^{(m)} = Z \setminus \{z_1, \ldots, z_m\}$.

2.2 Adaptive Selection of Support Points

Unlike classical barycentric interpolation where all support points are fixed in advance, the AAA algorithm selects them *adaptively*. The main steps are as follows:

1. **Initialization:** Start with a discrete set of sample points Z and choose an arbitrary initial support point $z_1 \in Z$.

2. **First approximation:** Construct the initial rational function $r_1(z)$ using z_1.

3. **Adaptive refinement:** At the j-th iteration, identify the point in Z where the current approximation error is maximal:

$$z_j = \arg\max_{z \in Z} ||f(z) - r_{j-1}(z)||.$$

   Add this point to the support set $\{z_1, \ldots, z_j\}$.

4. **Least-squares update:** Construct the corresponding matrix $A^{(j)}$ and compute its singular value decomposition (SVD). The barycentric weights $w^{(j)}$ are obtained from the right singular vector corresponding to the smallest singular value.

5. **Stopping criterion:** Repeat steps 3–4 until the residual error over the sample set is below a prescribed tolerance or no further improvement is observed.

3 Simple Implementation / Experiment

In this section, we present a simplified Python implementation of the AAA algorithm to demonstrate its core idea of adaptive rational approximation. The goal is to approximate the function $f(z) = e^z$ over the interval $[-1, 1]$, and observe both the resulting rational approximant and the convergence behavior of the algorithm.

3.1 Python Implementation

```
import numpy as np
import matplotlib.pyplot as plt

def aaa(F, Z, tol=1e-13, mmax=20):
    """
    Simplified version of AAA algorithm.
    Demonstrates adaptive rational approximation with stability improvements.
    """
    M = len(Z)
    F = F.astype(complex)
    Z = Z.astype(complex)
    Fdiag = np.diag(F)
    J = np.arange(M)
    z, f = [], []
    C = np.zeros((M, 0), dtype=complex)
    errvec = []

    """
    Initial approximation = constant mean value.
    """
    R = np.mean(F)

    for m in range(1, mmax + 1):
        """
```

```
        --- pick next support point (max residual) ---
        """
        j = np.argmax(np.abs(F - R))
        z.append(Z[j])
        f.append(F[j])
        J = J[J != j]

        """
        --- update Cauchy matrix ---
        """
        diff = Z - Z[j]
        diff[diff == 0] = np.inf
        c_new = 1.0 / diff
        C = np.column_stack((C, c_new))

        """
        --- form Loewner matrix ---
        """
        fdiag = np.diag(f)
        A = Fdiag @ C - C @ fdiag

        """
        --- compute right singular vector ---
        """
        U, s, Vh = np.linalg.svd(A[J, :], full_matrices=False)
        w = Vh[-1, :]

        """
        --- build rational approximation ---
        """
        N = C @ (w * f)
        D = C @ w
        R = F.copy()
        mask = np.abs(D[J]) > 1e-14
        R[J[mask]] = N[J[mask]] / D[J[mask]]

        """
        --- convergence check ---
        """
        err = np.linalg.norm(F - R, np.inf)
        errvec.append(err)
        if err <= tol * np.linalg.norm(F, np.inf):
            break
    """
    --- define final rational approximant function ---
    """
    def r(zz):
        zz = np.array(zz, ndmin=1, dtype=complex)
        Zsup = np.array(z, dtype=complex)
        diff = zz[:, None] - Zsup[None, :]
        with np.errstate(divide='ignore', invalid='ignore'):
            Cmat = np.where(np.abs(diff) < 1e-14, 0, 1.0 / diff)
        num = Cmat @ (w * f)
        den = Cmat @ w
```

```
        rvals = np.empty_like(num)
        small_den = np.abs(den) < 1e-14
        rvals[~small_den] = num[~small_den] / den[~small_den]
        rvals[small_den] = np.nan
        for k, val in enumerate(zz):
            if np.isnan(rvals[k]):
                match = np.where(np.isclose(val, Zsup, atol=1e-14))[0]
                if len(match) > 0:
                    rvals[k] = f[match[0]]
        return rvals

    return r, np.array(z), np.array(f), w, np.array(errvec)
```

Listing 1: Simplified AAA algorithm implementation in Python

3.2 Experimental Results

The experiment approximates $f(z) = e^z$ on the interval $[-1, 1]$ using the simplified AAA algorithm above. Figure 1 shows the rational approximation compared to the true function, while Figure 2 illustrates the convergence of the error norm.

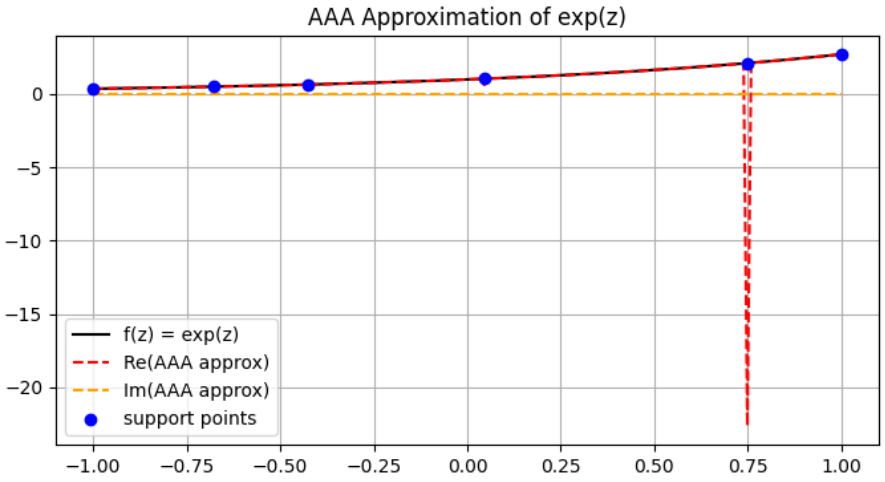

Figure 1: Approximation of $f(z) = e^z$ using the simplified AAA algorithm. Blue dots indicate selected support points.

3.3 Discussion

The implemented program effectively demonstrates how the AAA algorithm constructs a rational approximation adaptively by selecting support points where the current residual is largest. The initial support point z_1 is not chosen arbitrarily; instead, it is defined as

$$z_1 = \arg\max_{z_i \in Z} ||F(z_i) - \text{mean}(F)||,$$

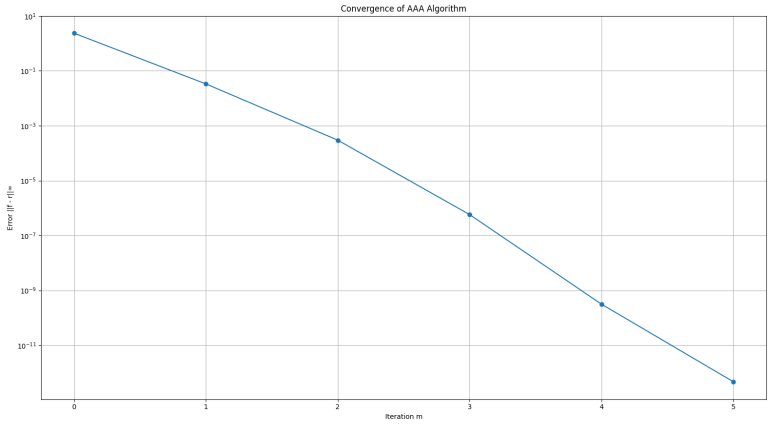

Figure 2: Convergence of the error $\|f - r_m\|_\infty$ with respect to the iteration index m.

which helps the algorithm start from the region where the function exhibits the largest variation, thereby improving convergence.

The function r(zz) in the code serves the same role as rhandle(zz, z, f, w) in the original AAA paper, computing the rational approximant at arbitrary input points. In this simplified version, we omit the auxiliary routine prz(r, z, f, w)—used in the full implementation to compute poles, residues, and zeros—since it is mainly needed for detecting and removing numerical Froissart doublets.

From Figure 1, we observe that the approximation behaves well overall, except for a small oscillation near $z = 0.75$. This local disturbance is likely caused by a *numerical Froissart doublet*, a phenomenon where a spurious pole-zero pair appears due to floating-point precision limitations. Apart from this minor artifact, the algorithm successfully achieves a stable and accurate rational approximation of e^z on the given interval.

4 Summary or Conclusion

The AAA algorithm provides an elegant and efficient approach to constructing rational approximations of complex-valued functions. By combining the barycentric representation with an adaptive selection of support points, it achieves both numerical stability and high approximation accuracy. Its ability to automatically determine interpolation points based on the residual error makes it particularly effective for functions with sharp features, oscillations, or singularities, where traditional polynomial approximations often fail.

Despite these strengths, the algorithm has several limitations. Numerical Froissart doublets can still appear due to floating-point errors, especially when approximating nearly constant or very smooth functions. The computational cost can be high for large datasets because each iteration involves building and performing SVD on a Loewner-type matrix. Performance also depends on the distribution of sample points, and the greedy, step-by-step selection of support points does not always guarantee a globally optimal approximation.