UCLouvain

LINMA2171 - Numerical Analysis: Approximation, Interpolation, Integration

---

Homework 2

Student
Lucas Ahou (35942200)

Teacher
P-A. Absil

November 01, 2025

Polynomial interpolation: Newton & Neville

2. Computational complexities

Newton's divided differences

a) The coefficients for the Newton's algorithm are the divided differences :

$$a_j := [y_0, ..., y_j]$$

where :

$$[y_k] := y_k$$
$$[y_k, ..., y_{k+j}] := \frac{[y_{k+1}, ..., y_{k+j}] - [y_k, ..., y_{k+j-1}]}{x_{k+j} - x_k}$$

with y_k the points we want to interpolate.
For each $[y_k, ..., y_{k+j}]$, we perform 2 substractions and 1 division. The number of elements to compute to get all the coefficients is $\frac{n(n-1)}{2}$. The total number of flops to compute all the coefficients is :

$$3 \cdot \frac{n(n-1)}{2} \sim \mathcal{O}(n^2)$$

b) Assuming all the coefficients are already computed, we simply need to compute:

$$p(x) = \sum_{j=0}^{n-1} a_j n_j(x)$$
$$n_j(x) = \begin{cases} \prod_{i=0}^{j-1}(x - x_i) & j > 0 \\ 1 & j = 0 \end{cases}$$

Every $n_j(x)$ can be computed based on $n_{j-1}(x)$ by multiplying by $(x - x_j)$. Each term of the sum thus requires 1 addition, 1 substraction and 1 multiplication. The total number of operations is $n-1$ multiplications and $2(n-1)$ additions/substractions. The complexity is thus $\mathcal{O}(n)$.

c) First, let's examine the storage requirements to compute the coefficients. We know that the divided differences table contains $\frac{n(n-1)}{2}$ entries. However, the evaluated points y_k are only necessary to compute the k-th coefficient. We can thus use the given array that contains the y_k to compute the coefficients in place. The total storage requirement for the coefficient computation is $\mathcal{O}(n)$ (n for the nodes x_i and n for the evaluated points y_i).

Second, to evaluate the interpolation at m distinct points, we simply need a single variable that gets updated for each point. The storage requirement is thus $\mathcal{O}(m)$ for the evaluation part.

The total storage requirement is $\mathcal{O}(n+m)$.

Neville's algorithm

a) Neville's algorithm is designed for direct evaluation and does not precompute coefficients. So, there is no complexity for the coefficient computation.

b) The triangular table contains $\frac{n(n+1)}{2}$ and each entry $P_{i,j}(x)$ requires 2 substractions, 1 multiplication and 1 division. Since the first column of the table is given, the total number of operation is:

$$4 \cdot \left(\frac{n(n+1)}{2} - n \right) = 2n(n-1) \sim \mathcal{O}(n^2)$$

c) For a single point, we need at each step at most n slots. In fact, after computing $P_{0,1}$, we can use the slot where y_0 was stored to place it. The same goes for the other entries. The space requirement for a single point is thus $\mathcal{O}(n)$.

For m evaluation, we simply need m addition slots. The total complexity is $\mathcal{O}(n+m)$.

3. Test functions

a) After implementing both methods in Python, let's plot the interpolation for $f_1(x) = \cos(x)$ using Neville's method and for $f_2(x) = \frac{1}{1+25x^2}$ using Newton's method for $n = 5, 10$ and 15 interpolation nodes. Here are the results :

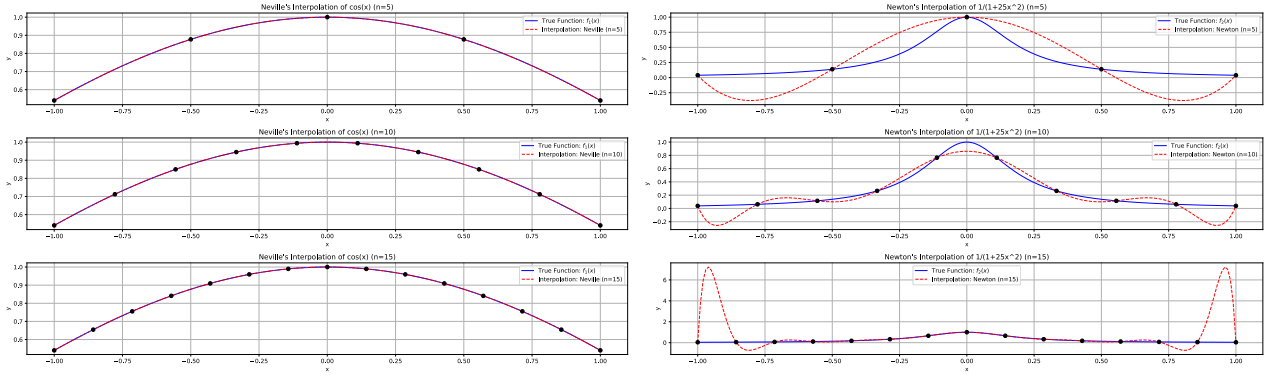

Figure 1 – Neville and Newton's methods applied to f_1 and f_2 respectively for $n = 5, 10$ and 15

On Figure 1, we can see that f_1 is being interpolated almost perfectly on the interval $[-1, 1]$, even for small values of n, using Neville's method. However, f_2 is not properly interpolated with the Newton's method. We observe a noticeable gap between the interpolation and the true function. While increasing n allows to obtain a better interpolation in the middle of the interval, we however notice a Runge's phenomenon on the borders when n becomes too large.

b) Before computing and plotting the interpolation errors, let's try to switch the methods for these two functions to see if we can visually notice a difference :

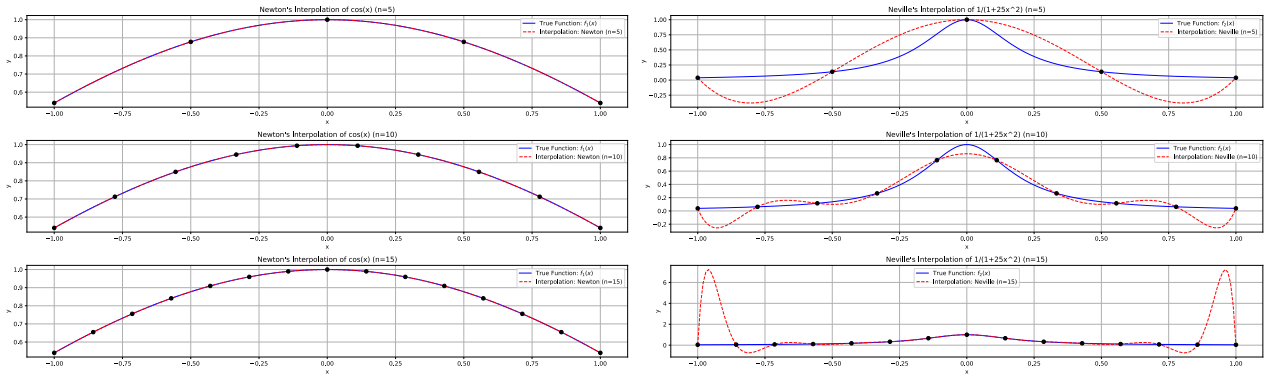

Figure 2 – Newton and Neville's methods applied to f_1 and f_2 respectively for $n = 5, 10$ and 15

On Figure 2, we can not observe any difference after swapping these two methods. We expect the errors to be relatively close between these two. Let's compute the errors using the ℓ^2-norm and plot them with respect to n:

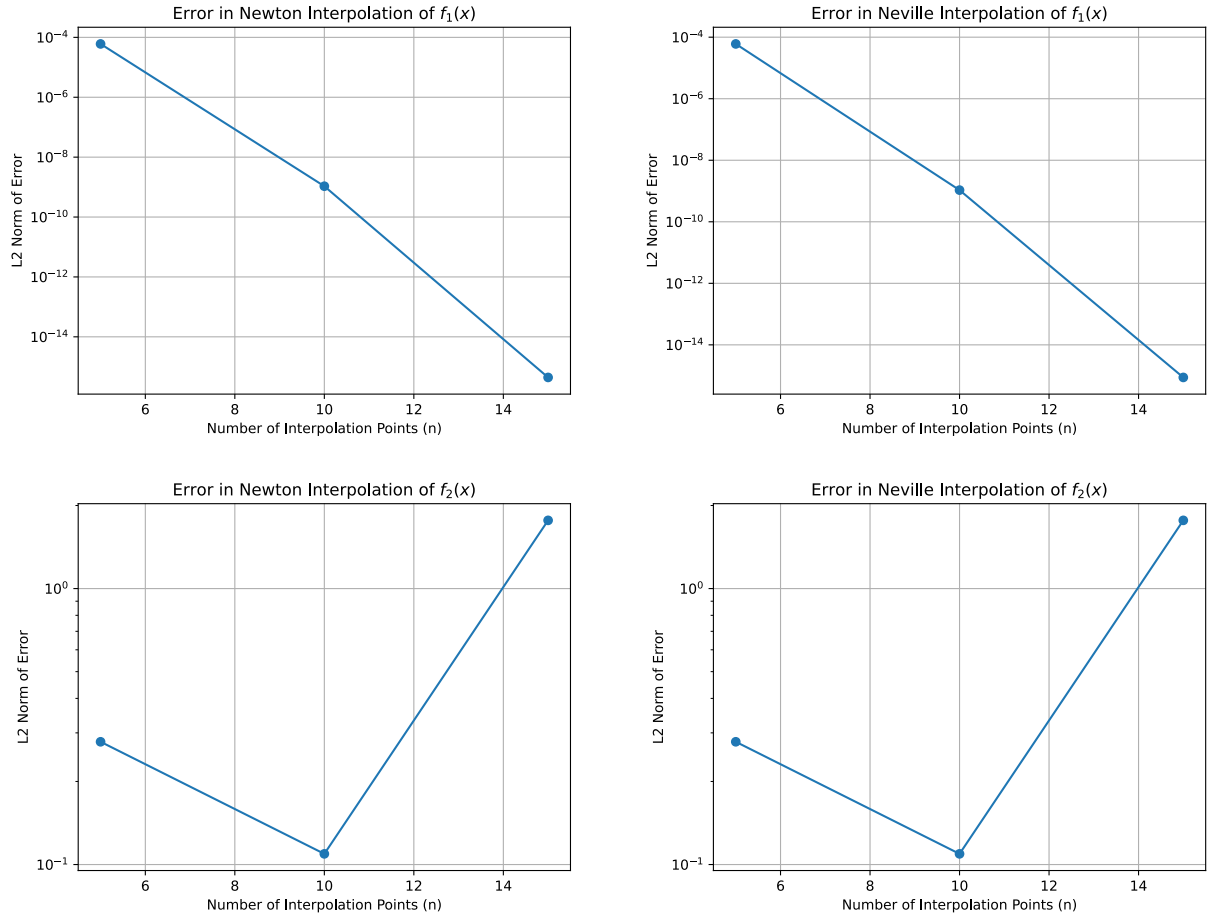

Figure 3 – Newton and Neville's methods applied to f_1 and f_2 respectively for $n = 5, 10$ and 15

On Figure 3, we see that the errors are exactly the same which indicates that both methods interpolate exactly in the same way. This is totally expected because we are trying to fit a polynomial of the same degree $n-1$ on the same nodes. The interpolation theorem tells us that there is a unique polynomial that does this, which is why both methods produce the same polynomial even though the procedure to achieve this is different.

Rational interpolation: Floater-Hormann

1. Proving r is rational

A rational function is defined as a function that can be expressed as the ratio of two polynomials. Let's analyze the numerator and denominator of $r(x)$.

Denominator $D(x)$:
We define:

$$D(x) := \sum_{i=0}^{n-d} \lambda_i(x) = \sum_{i=0}^{n-d} \frac{(-1)^i}{\prod_{j=0}^{d}(x - x_{i+j})}$$

This is a sum rational terms with a denominator of degree $d+1$. We can express this sum as a rational function but we first have to put every term to the same denominator. The common denominator is :

$$Q(x) := \prod_{k=0}^{n}(x - x_k)$$

which is a polynomial of degree $n+1$. Thus, each λ_i can be written as :

$$\lambda_i(x) = (-1)^i \frac{P_i(x)}{Q(x)}$$

where P_i is the polynomial in the numerator of the i-th term after putting it to the common denominator and so has a degree of $n-d$. The denominator $D(x)$ of $r(x)$ then becomes:

$$D(x) = \sum_{i=0}^{n-d} (-1)^i \frac{P_i(x)}{Q(x)} = \frac{\sum_{i=0}^{n-d} (-1)^i P_i(x)}{Q(x)} := \frac{\tilde{D}(x)}{Q(x)}$$

where $\tilde{D}(x)$ is a polynomial of the same degree $n-d$ as P_i.

Numerator $N(x)$:
We define:

$$N(x) := \sum_{i=0}^{n-d} \lambda_i(x) p_i(x) = \sum_{i=0}^{n-d} \frac{(-1)^i p_i(x)}{\prod_{j=0}^{d}(x - x_{i+j})}$$

Using the same common denominator $D(x)$ as before, we can rewrite the i-th term as:

$$\lambda_i(x) p_i(x) = (-1)^i \frac{p_i(x) P_i(x)}{Q(x)}$$

where P_i is the same polynomial of degree $n - d$ as before. Therefore, the numerator becomes:

$$N(x) = \sum_{i=0}^{n-d} (-1)^i \frac{p_i(x) P_i(x)}{Q(x)} = \frac{\sum_{i=0}^{n-d} (-1)^i p_i(x) P_i(x)}{Q(x)} := \frac{\tilde{N}(x)}{Q(x)}$$

Where $\tilde{N}$ is also a polynomial of the same degree as $p_i(x)P_i(x)$, i.e. $d + (n - d) = n$.

Final form of $r(x)$:
Finally, we combine these two expression to obtain the final form for $r(x)$:

$$r(x) = \frac{N(x)}{D(x)} = \frac{\frac{\tilde{N}(x)}{Q(x)}}{\frac{\tilde{D}(x)}{Q(x)}} = \frac{\tilde{N}(x)}{\tilde{D}(x)}$$

which is a ratio of two polynomials, whose numerator is of degree at most n and whose denominator is of degree at most $n - d$.

Polynomial f of degree at most d:
If f is a polynomial of degree at most d, then every p_i is exactly equal to $f(x)$. This is because, by definition, p_i is the unique polynomial of degree $\leq d$ interpolating f at $d + 1$ points. If we then substitute $p_i(x) = f(x)$ into the definition of $r(x)$, we have:

$$r(x) = \frac{\sum_{i=0}^{n-d} \lambda_i(x) p_i(x)}{\sum_{i=0}^{n-d} \lambda_i(x)} = \frac{\sum_{i=0}^{n-d} \lambda_i(x) f(x)}{\sum_{i=0}^{n-d} \lambda_i(x)} = f(x)$$

This, and the other results, however assume that the denominator $\sum_{i=0}^{n-d} \lambda_i(x)$ does not cancel. This is the case and can easily be seen from the definition of λ_i.

2. Barycentric form of $r(x)$

To express $r(x)$ in the barycentric form given in the statement, let's first express each polynomial p_i in the Lagrange form. We recall that p_j interpolates f at $x_j, ..., x_{j+d}$:

$$p_j(x) = \sum_{l=j}^{j+d} f(x_l) L_{j,l}(x)$$

where $L_{j,l}$ is the Lagrange polynomial for the nodes $x_j, ..., x_{j+d}$, defined as:

$$L_{j,l} = \prod_{\substack{m=j \\ m \neq l}}^{j+d} \frac{x - x_m}{x_l - x_m}$$

Now let's substitute p_i with this form in the numerator $N(x)$:

$$\begin{aligned} N(x) &= \sum_{j=0}^{n-d} \lambda_j(x) p_j(x) \\ &= \sum_{j=0}^{n-d} \lambda_j(x) \left(\sum_{l=j}^{j+d} f(x_l) L_{j,l}(x) \right) \\ &= \sum_{j=0}^{n-d} \sum_{l=j}^{j+d} f(x_l) \lambda_j(x) L_{j,l}(x) \end{aligned}$$

Knowing that $\lambda_j(x) = \frac{(-1)^j}{\prod_{m=j}^{j+d}(x - x_m)}$, the product $\lambda_j(x) L_{j,l}(x)$ can be simplified to:

$$\begin{aligned} \lambda_j(x) L_{j,l}(x) &= \frac{(-1)^j}{\prod_{m=j}^{j+d}(x - x_m)} \cdot \prod_{\substack{m=j \\ m \neq l}}^{j+d} \frac{x - x_m}{x_l - x_m} \\ &= (-1)^j \cdot \frac{1}{x - x_l} \cdot \prod_{\substack{m=j \\ m \neq l}}^{j+d} \frac{1}{x_l - x_m} \\ &= \frac{C_{j,l}}{x - x_l} \end{aligned}$$

where we defined

$$C_{j,l} := (-1)^j \prod_{\substack{m=j \\ m \neq l}}^{j+d} \frac{1}{x_l - x_m}$$

Our numerator thus becomes:

$$N(x) = \sum_{j=0}^{n-d} \sum_{l=j}^{j+d} f(x_l) \frac{C_{j,l}}{x - x_l}$$

To obtain the required form, we need to interchange the sums. To do that, let's introduce this notation:

$$\chi_E := \begin{cases} 1 & \text{if } E \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

This allows to rewrite the previous sums so that the inner one does not depend on the outer one:

$$\begin{aligned} N(x) &= \sum_{j=0}^{n-d} \sum_{l=j}^{j+d} f(x_l) \frac{C_{j,l}}{x - x_l} \\ &= \sum_{j=0}^{n-d} \sum_{l=0}^{n} f(x_l) \frac{C_{j,l}}{x - x_l} \chi_{\{j \le l \le j+d\}} \end{aligned}$$

The sums can now be interchanged safely:

$$\begin{aligned} N(x) &= \sum_{j=0}^{n-d} \sum_{l=0}^{n} f(x_l) \frac{C_{j,l}}{x - x_l} \chi_{\{j \le l \le j+d\}} \\ &= \sum_{l=0}^{n} \sum_{j=0}^{n-d} f(x_l) \frac{C_{j,l}}{x - x_l} \chi_{\{j \le l \le j+d\}} \\ &= \sum_{l=0}^{n} \frac{f(x_l)}{x - x_l} \left(\sum_{j=0}^{n-d} C_{j,l} \chi_{\{j \le l \le j+d\}} \right) \end{aligned}$$

Now, because j starts at 0 but must be $l - d$, the starting index can be written as $\max(0, l - d)$. The ending index can be written as $\min(l, n - d)$ because j ends at $n - d$ but must also be smaller than l. The sum thus becomes:

$$\begin{aligned} N(x) &= \sum_{i=0}^{n} \frac{f(x_i)}{x - x_i} \left(\sum_{j=\max(0,i-d)}^{\min(i,n-d)} C_{j,i} \right) \\ &= \sum_{i=0}^{n} \frac{w_i}{x - x_i} f(x_i) \end{aligned}$$

where we defined:

$$w_i = \sum_{j=\max(0,i-d)}^{\min(i,n-d)} C_{j,i} = \sum_{j=\max(0,i-d)}^{\min(i,n-d)} (-1)^j \prod_{\substack{k=j \\ k \neq i}}^{j+d} \frac{1}{x_i - x_m}$$

The same logic can be applied to the denominator $D(x)$ to obtain:

$$D(x) = \sum_{i=0}^{n} \frac{w_i}{x - x_i} \quad \square$$

4. Applying Floater-Hormann to test functions

a) We will now apply the same tests as before but using the Floater-Hormann method. We will be using the same functions, the same values of n and values of $d = \{0, 3, 5, 8\}$. However, we must have $d \leq n$ so we can not test all values of d for all values of n.

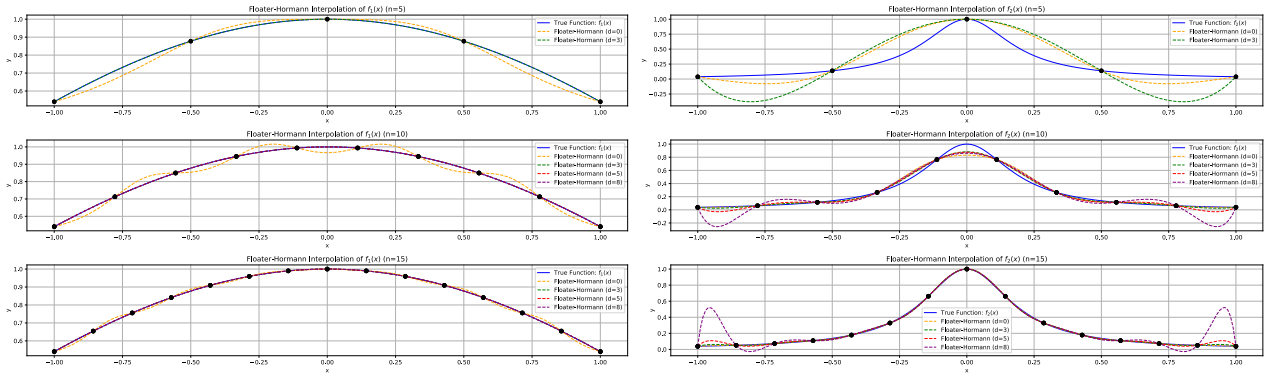

Figure 4 – Floater-Hormann method applied to f_1 and f_2 with $n = \{5, 10, 15\}$ and $d = \{0, 3, 5, 8\}$ (when applicable)

On Figure 4, we observe that the Floater-Hormann interpolation does not interpolate $f_1(x) = \cos(x)$ as well as the first two methods we used. However, as n increases, we see that it starts to interpolate it pretty well. The same can be said for f_2. However, we can see that the Runge's phenomenon is less present than before, at least for small d. In fact, when d reaches n, Floater-Hormann method becomes a polynomial interpolation like Newton's or Neville's method. We see that for high n's, small values of d like $d = 0$ or $d = 3$, the Runge's phenomenon is almost not noticeable. Let's now, like before, compute and plot the errors to further analyze the performance

b) Let's compute the errors based on the ℓ^2-norm for values of n ranging from 5 to 30 and for different values of d ranging from 0 to 15 (only for valid n's):

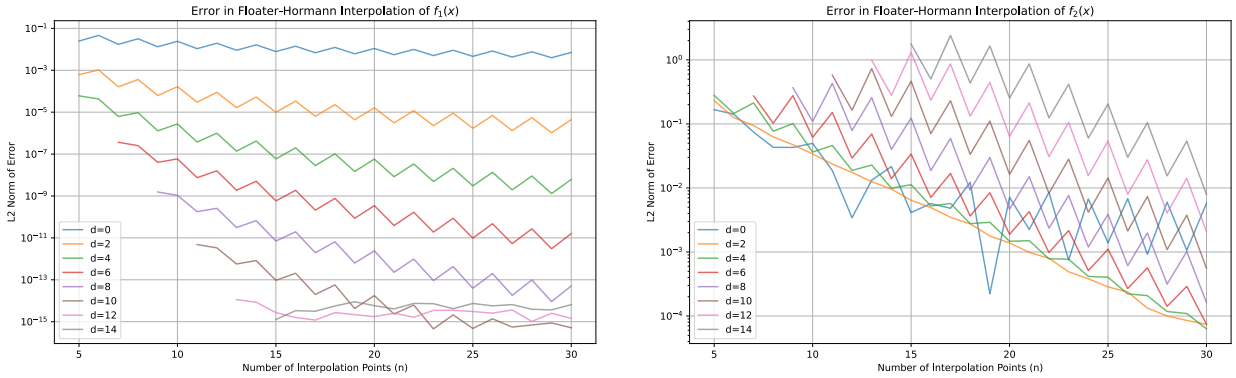

Figure 5 – Interpolation errors of the Floater-Hormann method applied to f_1 and f_2 with respect to n for different d

On Figure 5, we see that the errors seem to decrease as n increases whether we try to interpolate f_1 or f_2. However, a noticeable difference is in the choice of d. In fact, for f_1, a higher value of d seems to make the error smaller, which is not the case for f_2. This can be explained by the fact that choosing a smaller d prevents the Runge's phenomenon from happening, which is why the error is smaller in the interpolation of f_2 for small d's. For this function, it looks like the best d is $d = 2$.
Because the first function does not manifest a Runge's phenomenon when interpolated, picking a higher d allows to interpolate almost perfectly because we tend to a polynomial interpolation which, as we have seen previously, interpolates this specific function better.

c) To compare this method with the Newton and Neville's ones, we can simply look at Figure 5 for the errors when $n = d$ because this corresponds to the polynomial interpolation as said previously.
For f_1, we see that the polynomial interpolation is clearly better than Floater-Hormann's interpolation. Again, this is because there is no Runge's phenomenon so there is no point in using the latter. However, for f_2, choosing a smaller blending value makes the Runge's phenomenon vanish which decreases the error.

Conclusion

For functions that are easily interpolated by polynomials, Newton's and Neville's methods are better suited than Floater-Hormann. Amongst these two, Newton's method is better for multiple evaluations because, even though the coefficient cost is $\mathcal{O}(n^2)$, it is performed only once. Each evaluation then only has a complexity of $\mathcal{O}(n)$. Neville's method is better when we only have to compute a single evaluation.
When a Runge's phenomenon appears when we perform a polynomial interpolation, one should fall back to a Floater-Hormann interpolation with a small d.

References

[1] "Newton polynomial." [Online]. Available: *https://en.wikipedia.org/wiki/Newton_polynomial*

[2] "Neville's algorithm." [Online]. Available: *https://en.wikipedia.org/wiki/Neville%27s_algorithm*

[3] Michael S. Floater and Kai Hormann, "Barycentric rational interpolation with no poles and high rates of approximation," *Numerische Mathematik*, vol. 107.2 (2007), pp. 315–331.