International Journal of Applied Science and Mathematics
Volume 4, Issue 5, ISSN (Online): 2394-2894

Numerical Comparisons of Polynomial Interpolations

Jen-Yuan Chen[1*] and Ya-Fang Wang[2]
[1]Department of Mathematics, National Kaohsiung Normal University, Taiwan.
[2] Kaohsiung Municipal Nanzih Junior High School, Taiwan.
*Corresponding author email id: jchen@nknu.edu.tw

Date of publication (dd/mm/yyyy): 02/09/2017

Abstract **– We introduce several polynomial interpolations such as Newton polynomial interpolation, Lagrange polynomial interpolation and Barycentric Lagrange interpolation. Except for comparing the operations of several polynomial interpolations, we also compute the errors of each polynomial interpolation for Runge function and Natural Logarithm function.**

Keywords **– Barycentric Lagrange interpolation, Lagrange polynomial interpolation, Newton polynomial interpolation, Runge function, Natural Logarithm function.**

I. INTRODUCTION

One of the essential problems in numerical algorithms is the problem of interpolation, that is given a set of data (x_i, f_i) for $i = 0,1,\ldots,n$, how do we find a function $P(x)$ that connects the data points such that the function $P(x)$ goes through the data, *i.e.* $P(x_i) = f_i$. There are plenty of ways to thread those data points ranging from polynomial interpolation to cubic splines.

Originally, Interpolation is a tool in producing computable approximations to continuous. Due to the advent of computers and calculators, Interpolation is used to solve problems from the more general area of interpolation theory in the present day. To find the approximation of the integration or derivative of a function, We usually replace the function by simpler approximation and then integrate or differentiate the approximation of the function. These simpler approximations are often obtained by interpolation.

Many different methods have been developed to construct useful interpolation formulas. Newton's divided difference formula and Lagrange's formula are the most famous interpolation formulas for polynomial interpolation to any arbitrary degree with finite number of points.

We explain how this two types of Polynomial Interpolations work and reveal several algorithms in Section 2. In Section 3, we introduce an improved Lagrange formula and the conversion between Newton interpolating polynomial and Lagrange interpolating polynomial. In Section 4, we give Numerical experiment and compute the cost of different Polynomial Interpolations. At last, we draw a conclusion in Section 5.

One of the essential problems in numerical algorithms is the problem of interpolation, that is given a set of data (x_i, f_i) for $i = 0,1,\ldots n$, how do we find a function $P(x)$ that connects the data points such that the function $P(x)$ goes through the data, i.e. $P(x_i) = f_i$. There are plenty of ways to thread those data points ranging from polynomial interpolation to cubic splines.

Originally, Interpolation is a tool in producing computable approximations to continuous. Due to the advent of computers and calculators, Interpolation is used to solve problems from the more general area of interpolation theory in the present day. To find the approximation of the integration or derivative of a function, we usually replace the function by simpler approximation and then integrate or differentiate the approximation of the function. These simpler approximations are often obtained by interpolation.

Many different methods have been developed to construct useful interpolation formulas. Newton's divided difference formula and Lagrange's formula are the most famous interpolation formulas for polynomial interpolation to any arbitrary degree with finite number of points.

We explain how this two types of Polynomial Interpolations work and reveal several algorithms in Section 2. In Section 3, we introduce an improved Lagrange formula and the conversion between Newton interpolating polynomial and Lagrange interpolating polynomial. In Section 4, we give Numerical experiment and compute the cost of different Polynomial Interpolations. At last, we draw a conclusion in Section 5.

II. POLYNOMIAL INTERPOLATION

Given $n+1$ distinct points x_0, x_1, ..., x_n, and arbitrary values f_0, f_1, ..., f_n. The polynomial P_n of degree at most n which satisfies $P_n(x_i) = f_i$, $i = 0,...,n$ is called interpolating polynomial. From the algebraic aspects of polynomial interpolation, the existence and uniqueness of interpolating polynomial are easy to prove. [2]

There are many different ways of expressing and evaluating this interpolating polynomial depending on the choice of polynomial basis.

1) 2.1 Monomial Basis

The standard basis for polynomials of degree less than or equal to n is $1, x, x^2, \ldots, x^n$. We usually think of polynomials as a linear combination of the monomials $1, x, x^2, \ldots, x^n$, *i.e.*

$$P_n(x) = a_0 + a_1 x + \ldots + a_n x^n,$$

where $a_0, a_1, \ldots, a_n$ are undetermined coefficients.

To determine these coefficients we need to solve the $(n+1)\times(n+1)$ linear system.

$$\begin{aligned} P_n(x_0) &= f \\ P_n(x_1) &= f \\ &\vdots \\ P_n(x_n) &= f_n \end{aligned} \quad (1)$$

or in matrix-vector form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix} \quad (2)$$

as $AX = f$

It can be shown that if all the coordinates $x_0, x_1, \ldots, x_n$ are distinct, then the Vandermonde matrix A is nonsingular and therefore the coefficients are unique.

2) 2.2 Newton Polynomial Basis

Given $n+1$ distinct points x_0, x_1, ..., x_n and arbitrary values f_0, f_1, ..., f_n. A second basis for polynomials is the Newton Polynomials.

$$\begin{aligned} \phi_0(x) &= 1 \\ \phi_1(x) &= (x - x_0) \\ \phi_2(x) &= (x - x_0)(x - x_1) \\ &\vdots \\ \phi_n(x) &= (x - x_0)(x - x_1)\ldots(x - x_{n-1}) \end{aligned}$$

Thus the polynomial $P_n(x)$ can be written as

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)\cdots(x - x_{n-1})$$

Since $P_n(x)$ interpolates $n+1$ data points (x_0, f_0), (x_1, f_1), ..., (x_n, f_n).

It follows that

$$\begin{aligned} P_n(x_0) = f_0 &\Rightarrow f_0 = a_0 \\ P_n(x_1) = f_1 &\Rightarrow f_1 = a_0 + a_1(x_1 - x_0) \\ P_n(x_2) = f_2 &\Rightarrow f_2 = a_0 + a_1(x_2 - x_0) \\ &\quad + a_2(x_2 - x_0)(x_2 - x_1) \\ P_n(x_n) = f_n &\Rightarrow f_n = a_0 + a_1(x_n - x_0) + \ldots + \\ &\quad a_n(x_n - x_0)(x_n - x_1)\ldots(x_n - x_{n-1}) \end{aligned}$$

Therefore it becomes the Lower Triangular System

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & \\ 1 & (x_1 - x_0) & 0 & \cdots & \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & \varphi_1(x_n) & \varphi_2(x_n) & \cdots & \ ) \end{pmatrix}$$

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

Solving the lower triangular system by forward substitution, we have

$$a_0 = f_0$$

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$a_2 = \frac{f_2 - a_0 - (x_2 - x_0)a_1}{(x_2 - x_0)(x_2 - x_1)}$$

$$= \frac{f(x_2) - f(x_0) - (x_2 - x_0)\dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{(x_2 - x_0)(x_2 - x_1)}$$

$$= \frac{\dfrac{f(x_2) - f(x_0)}{x_2 - x_0} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{(x_2 - x_1)}$$

$$= \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{(x_2 - x_0)}$$

We see the pattern $\dfrac{f(x_j) - f(x_i)}{x_j - x_i}$ occurring in the coefficients of $\phi_j(x)$.

Define $f[x_0, x_1, \cdots, x_j]$ to be the coefficient of ϕ_j of the interpolating polynomial. Thus the Newton Interpolating polynomial could be written as

$$P_n(x) = \sum_{j=0}^{n} f[x_0, \ldots, x_j](x - x_0)\cdots(x - x_{j-1}) \quad (3)$$

where

$$f[x_0, x_1, \cdots, x_{n-1}, x_n] = \frac{f[x_1, \cdots, x_n] - f[x_0, \cdots, x_{n-1}]}{x_n - x_0}$$

The strengths and weaknesses of Newton interpolation are:

• Easy to determine $P_n(x)$ -

International Journal of Applied Science and Mathematics
Volume 4, Issue 5, ISSN (Online): 2394-2894

An efficient way to evaluate the coefficients $a_i's$ of the Newton interpolating polynomial is divided differences which takes about $n^2/2$ operations.

• Easy to evaluate $P_n(x)$ for arbitrary x -

Use Horner's scheme can accelerate the rate.

$$P_n(x) = a_0 + a_1(x-x_0) + \cdots + a_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})$$
$$= a_0 + (x-x_0)\{a_1 + (x-x_1)\{a_2 + (x-x_2)\{\cdots\{a_{n-1} + (x-x_{n-1})a_n\}\ldots\}\}\}$$

• Derivatives of $P_n(x)$ is available -

$$P_n(x) = Q_0(x) \quad \text{where} \quad Q_n(x) = a_n, \quad Q_{n-1}(x) = a_{n-1} + (x-x_{n-1})Q_n(x)$$

Once we compute all the $Q_k's$, we can get $P_{n'}(x)$.

• Allow incremental interpolation.

There are two Krogh Algorithms to evaluate the Newton form.

1. Krogh Algorithm I

$\Pi_0 = 1$
$V_{00} = f(x_0)$
$P_0 = V_{00}$
for $k = 1:n$
$V_{0k} = f(x_k)$
for $i = 0,1,\ldots,k-1$
$V_{i+1,k} = \dfrac{V_{ii} - V_{ik}}{x_i - x_k}$
end
$w_{k-1} = x - x_{k-1}$
$\Pi_k = w_{k-1}\Pi_{k-1}$
$P_k(x) = P_{k-1} + \Pi_k V_{kk}$
end

2. Krogh Algorithm II(Horner's Scheme)

$V_{00} = f(x_0)$
for $k = 1:n$
$V_{0k} = f(x_k)$
for $i = 0,1,\ldots,k-1$
$V_{i+1,k} = \dfrac{V_{ii} - V_{ik}}{x_i - x_k}$
end
end
for $k = n, n-1, \ldots, 2, 1$
$V_{k-1,k-1} = V_{k-1,k-1} + (x - x_{k-1})V_{kk}$
end
$P_n(x) = V_{00}$

3) 2.3 Lagrange Polynomial Basis

In this section we will explain how the polynomial interpolation in Lagrange form works.

Given $n+1$ distinct data points (x_0, f_0), (x_1, f_1), ..., (x_n, f_n). The idea of finding the polynomial which conform to $P_n(x_i) = f_i$, for $i = 0, \cdots, n$ is as follow :

First of all, we consider an simple situation:

$$f_i = 1, f_0 = \cdots = f_{i-1} = f_{i+1} = \cdots = f_n = 0$$

Let $l_i(x)$ be the polynomial of degree at most n which fit in with this condition. We name $l_i(x)$ as the i-th Lagrange polynomial. That is, the i-th Lagrange polynomial $l_i(x)$ of degree n must satisfy

$$l_i(x_j) = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases} \quad i = 0, \cdots, n$$

Since $l_i(x)$ is an n -degree polynomial with n -roots $x_0, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n$. We know that it must have the form

$$l_i(x) = C_i \cdot (x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)$$
$$= C_i \prod_{j=0, j\neq i}^{n} (x - x_j)$$

The condition $l_i(x_i) = 1$ leads to $C_i = \dfrac{1}{\prod_{j=0, j\neq i}^{n}(x_i - x_j)}$.

It follows that $l_i(x) = \dfrac{\prod_{j=0, j\neq i}^{n}(x - x_j)}{\prod_{j=0, j\neq i}^{n}(x_i - x_j)}$.

Second, we adjust above situation and consider the general situation.

Set $f_i =$ be any arbitrary number, $f_0 = \cdots = f_{i-1} = f_{i+1} = \cdots = f_n = 0$

It is obvious that $f_i \cdot l_i(x)$ satisfies $f_i l_i(x_i) = 1$ and $f_i l_i(x_j) = 0$ for $i \neq j$.

All we have to do is adding up all $f_i \cdot l_i(x)$.

Then we obtain the polynomial interpolation in Lagrange form :

$$P_n(x) = \sum_{i=0}^{n} f_i \cdot l_i(x),$$

$$where\ l_j(x) = \frac{\prod_{i=0, j\neq j}^{n}(x - x_i)}{\prod_{i=0, j\neq j}^{n}(x_j - x_i)} \qquad (4)$$

The strengths and weaknesses of Lagrange interpolation are :

• Easy to determine $P_n(x)$ - But it costs a lot to write as $P_n(x) = a_0 + a_1 X + \cdots + a_n x^n$.

• Easy to evaluate $P_n(x)$ for arbitrary x - It requires $O(n^2)$ operations.

• Hard to find the derivatives of $P_n(x)$ - differentiating each $l_i(x)$ is not trivial.

• Do not allow incremental interpolation.

4) 2.4 Aitken's and Neville's Interpolation Methods

Given points x_0 and x_1, let $P_{0,1}(x)$ denote the interpolation polynomial for function on this set of points. Applying Lagrange Polynomial Interpolation, we have

$$P_{0,1}(x) = f(x_0)\frac{x - x_1}{x_0 - x_1} + f(x_1)\frac{x - x_0}{x_1 - x_0}$$

So

$$P_{0,1}(x) = \frac{(x - x_0)f(x_1) - (x - x_1)f(x_0)}{x_1 - x_0}$$

In general, we have following Theorem. [7]

Theorem 1 *Let $P_{0,1,..,k}(x)$ be the interpolation polynomial for function f over set $\{x_0, x_1, \ldots, x_k\}$.*

Then $P_{0,1,..,k,k+1}(x)$ is equal to

$$\frac{(x - x_k)P_{0,1,..,k-1,k+1}(x) - (x - x_{k+1})P_{0,1,..,k}(x)}{x_{k+1} - x_k}$$

We use this theorem to construct recursive polynomial for computing the value of $P_{0,1,..,n}(x)$ systematically.

First construction is developed as follows:

Suppose that values of x_k and $f(x_k) = P_k$ have been given for $k = 0,1,\ldots,n$ where $n \geq 1$. We compute values of

$$P_{0,1} = \frac{(x - x_0)P_1 - (x - x_1)P_0}{x_1 - x_0}$$

and

$$P_{0,2} = \frac{(x - x_0)P_2 - (x - x_2)P_0}{x_2 - x_0}$$

By Theorem 1, we see that

$$P_{0,1,2} = \frac{(x - x_1)P_{0,2} - (x - x_2)P_{0,1}}{x_2 - x_1}$$

Now if we compute $P_{0,3}$, we can also compute

$$P_{0,1,3} = \frac{(x - x_1)P_{0,3} - (x - x_3)P_{0,1}}{x_3 - x_1}$$

Then it follows from Theorem 1 that

$$P_{0,1,2,3} = \frac{(x - x_2)P_{0,1,3} - (x - x_3)P_{0,1,2}}{x_3 - x_2}$$

Continuing in this way, we can compute row by row the array of the interpolation polynomials.

$$\begin{array}{ccccc} P_0 & & & & \\ P_1 & P_{0,1} & & & \\ P_2 & P_{0,2} & P_{0,1,2} & & \\ P_3 & P_{0,3} & P_{0,1,3} & P_{0,1,2,3} & \\ P_4 & P_{0,4} & P_{0,1,4} & P_{0,1,2,4} & P_{0,1,2,3,4} \end{array}$$

etc.

The process of computing the interpolated values of a function f by repeated use of Theorem 1 can be organized into a systematic programmable algorithm. We call this procedure the Aitken's Interpolation Method.

There are two Aitken Algorithms as follow:

1. Aitken Algorithm I

$P_{00} = f(x_0)$
$w_0 = x - x_0$
for $k = 1:n$
$P_{0k} = f(x_k)$
$w_k = x - x_k$
$P_{i+1,k} = \frac{w_k P_{ii} - w_i P_{ik}}{x_i - x_k}$, $i = 0,1,\ldots,k-1$
end
$P_n(x) = P_{nn}(x)$

2. Aitken Algorithm II

$P_{00} = f(x_0)$
$w_0 = x - x_0$
for $k = 1:n$
$P_{0k} = f(x_k)$
$w_k = x - x_k$
$P_{i+1,k} = P_{ii} + \frac{w_i}{x_i - x_k}(P_{ii} - P_{ik})$, $i = 0,1,\ldots,k-1$
end
$P_n(x) = P_{nn}(x)$

Second construction is developed as follows:

Suppose that values of x_k and $f(x_k) = P_k$ have been given for $k = 0,1,\ldots,n$ where $n \geq 1$. We compute values of

$$P_{0,1} = \frac{(x - x_0)P_1 - (x - x_1)P_0}{x_1 - x_0}$$

and $P_{1,2} = \frac{(x - x_1)P_2 - (x - x_2)P_1}{x_2 - x_1}$

By Theorem 1, we see that

$$P_{0,1,2}=\frac{(x-x_0)P_{1,2}-(x-x_2)P_{0,1}}{x_2-x_0}$$

Now if we compute $P_{2,3}$, we can also compute

$$P_{1,2,3}=\frac{(x-x_1)P_{2,3}-(x-x_3)P_{1,2}}{x_3-x_1}$$

Then it follows from Theorem 1 that

$$P_{0,1,2,3}=\frac{(x-x_0)P_{1,2,3}-(x-x_3)P_{0,1,2}}{x_3-x_0}$$

Continuing in this way, we can compute row by row the array of the interpolation polynomials

P_0
P_1 $P_{0,1}$
P_2 $P_{1,2}$ $P_{0,1,2}$
P_3 $P_{2,3}$ $P_{1,2,3}$ $P_{0,1,2,3}$
P_4 $P_{3,4}$ $P_{2,3,4}$ $P_{1,2,3,4}$ $P_{0,1,2,3,4}$
etc.

The process of computing the interpolated values of a function f by repeated use of Theorem 1 can be organized into a systematic programmable algorithm. We call this procedure the Neville's Interpolation Method.

5) 2.5 Runge Phenomenon

The error of interpolating polynomial is as the following Theorem.[4]

Theorem 2 *Let $f(x)$ be a real $(n+1)$ -times continuously differentiable function on the bounded interval $[a,b]$. For the interpolating polynomial $P_n(x)$ with $(n+1)$ pairwise distinct nodes $x_0,\ldots,x_n$ with $min(x_i)=a$ and $max(x_i)=b$ and values $f_i=f(x_i)$ we have for each $\bar{x}\in[a,b]$*

$$f(\bar{x})-P_n(\bar{x})=\frac{f^{(n+1)}(\xi)}{(n+1)!}\prod_{i=0}^{n}(\bar{x}-x_i)$$

where $\xi\in[a,b]$ is independent of $\bar{x}$

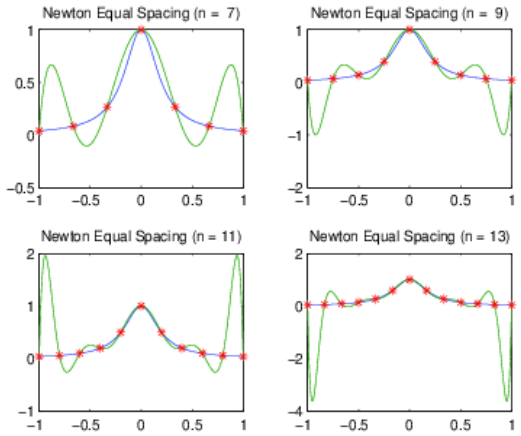


Fig. 1. Interpolate Runge function $\frac{1}{1+25x^2}$ at equal spacing points

From Theorem 2 we know that as we keep adding more and more data points, the error depends on the higher order derivatives of $f(x)$.

For some function, the values of $|f^{(n)}(x)|$ grow vastly as $k\to\infty$, like Runge's function, $f(x)=\frac{1}{1+25x^2}$. From Fig. 1 we find that the error become huge on the edge of the interval. So a possible improvement to reduce the error is to select the Chebyshev points as nodes.

6) 2.6 Chebyshev Points

The Chebyshev Polynomial are defined as $T_n(x)=\cos[n\arccos(x)]$.

By using basic trig identities we can establish the triple recursion relation

$$T_{n+1}(x)=2xT_n(x)-T_{n-1}(x),n=1,2,\ldots$$

It have been proved that the Chebyshev polynomial T_n is a polynomial of degree n. And the zeros of T_n are

$$x_k=\cos[\frac{\pi(2k+1)}{2n+2}],k=0,1,\ldots,n$$

The zeros are known as the Chebyshev-Gauss points.

It is for sure that $|f(x)-P(x)|\to 0$ as $n\to\infty$ when use Chebyshev points as long as both $f(x)$ and $f'(x)$ are bounded over[-1, 1]

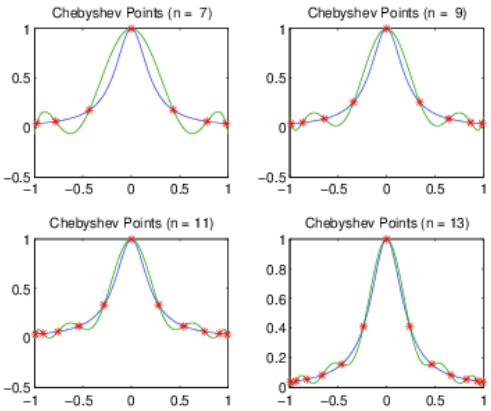


Fig. 2. Interpolate Runge function $\frac{1}{1+25x^2}$ at Chebyshev Points

But this is still not a ideal solution because of the following reasons:

- Sometimes we couldn't choose the $x_{i'}s$.
- It costs more than $O(n)$ computation to construct $P_n(x)$ as n increase.
- Local changes in the data points affect the entire extent of the interpolation.

To solve this problem, we can consider another method which was called Cubic Spline.

7) 2.7 Cubic Spline

Definition 1 *Given* $n+1$ *data points* (x_0, f_0), (x_1, f_1), …, (x_n, f_n), *a function* $S(x)$ *is called a cubic spline if*

- The domain of $S(x)$ is an interval $[a,b]$
- There is a partition $a = x_0, x_1 \ldots, x_n = b$ such that $S(x)$ is cubic on each subinterval $[x_i, x_{i+1}]$
- $S(x)$ interpolates $(x_i, f_i), i = 0,1,\ldots,n$
- $S(x)$, $S'(x)$, and $S''(x)$ are continuous on $[a,b]$

The cubic spline function $S(x)$ looks like

$$S(x) = \begin{cases} S_0(x) = a_{3,0}x^3 + a_{2,0}x^2 + a_{1,0}x + a_{0,0} & on [x_0, x_1] \\ S_1(x) = a_{3,1}x^3 + a_{2,1}x^2 + a_{1,1}x + a_{0,1} & on [x_1, x_2] \\ \vdots \quad\quad \vdots & \\ S_i(x) = a_{3,i}x^3 + a_{2,i}x^2 + a_{1,i}x + a_{0,i} & on [x_i, x_{i+1}] \\ \vdots \quad\quad \vdots & \\ S_{n-1}(x) = a_{3,n-1}x^3 + a_{2,n-1}x^2 + a_{1,n-1}x + a_{0,n-1} & on [x_{n-1}, x_n] \end{cases}$$

There are n intervals and $n+1$ knots. For each interval there are 4 unknowns, we have $4n$ unknowns it total. To satisfy the conditions of continuity and interpolation for $S(x)$ on x_0, x_n we require $2n$ constrains.

$$S_i(x_i) = f_i, S_i(x_{i+1}) = f_{i+1}, i = 0,1,\ldots,n-1$$

For the continuity of $S'(x)$ and $S''(x)$ we need $2(n-1)$ constrains.

$$S_{i'}(x_{i+1}) = S_{i+1}'(x_{i+1}), i = 0,1,\ldots,n-2$$

$$S_{i''}(x_{i+1}) = S_{i+1}''(x_{i+1}), i = 0,1,\ldots,n-2$$

In total, we shall have $4n$ unknowns and $4n-2$ conditions. This leaves 2 extra degrees of freedom. For the 2 extra degrees of freedom, there are some options:

- natural cubic spline: $S_{0''}(x_0) = S_{n''}(x_n) = 0$.
- complete cubic spline: Given $f'(x_0) = a$, $f'(x_n) = b$, we need that

$$S_{0'}(x_0) = f'(x_0), S_{n'}(x_n) = f'(x_n).$$

- not-a-Knot: $S'''(x)$ continuous at x_1 and x_{n-1}.
- periodic: S' and S'' are periodic at the ends.

Now let us take an example for the application of cubic spline. In Shortest path of a robot.[3] There are several restrictions when use a robot arm to manufacture something. The path of the robot going from one point to another point needs to be smooth so as to avoid sharp jerks in the arm that otherwise create premature wear and tears of the robot arm. For economic effects, the path of robot arm should be as short as possible. In this example, it illustrate that the cubic spline could achieve these goals.

III. Variants of Polynomial Interpolations

As we already know, "Lagrange interpolation is praised for analytic utility and beauty but deplored for numerical practice." [6] Could it be said that there is no other choice but Newton Interpolation to deal with numerical practice? The answer is absolutely No! The Barycentric Interpolation originated with Jocibi in 1825 is another choice for us to evaluate the interpolation polynomial.

1) 3.1 The first form of the Barycentric Interpolation

Given $n+1$ distinct interpolation points x_i, $i = 0,1,\ldots,n$ together with corresponding number f_i. The polynomial P_n which interpolates f at the points x_i can be written in Lagrange form:

$$P_n(x) = \sum_{i=0}^{n} f_i \cdot l_i(x), \text{ where } l_j(x) = \frac{\prod_{i=0, i\neq j}^{n} (x - x_i)}{\prod_{i=0, j\neq j}^{n} (x_j - x_i)} \quad (5)$$

Let $l(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$, and define the Barycentric weights by

$$w_j := \frac{1}{\prod_{i=0, i\neq j}^{n} (x_j - x_i)} = \frac{1}{l'(x_j)}, j = 0,1,\ldots,n$$

Then (5) can be written as

$$P_n(x) = \sum_{i=0}^{n} w_i f_i \prod_{j=0, j\neq i}^{n} (x - x_j) \quad (6)$$

Thus the Lagrange polynomial l_j could be written as

$$l_j(x) = l(x)\frac{w_j}{(x-x_j)}, j = 0,1,\ldots,n$$

Notice that $l(x)$ is the factor of all the terms of $l_j(x)$.

Therefore we can bring out the factor $l(x)$ in front of the sum in (5) and it yields

$$P_n(x) = l(x)\sum_{i=0}^{n} \frac{w_i}{(x-x_i)} f_i \quad (7)$$

That is it! Rutishauser [5] called it " the first form of the Barycentric Interpolation formula".

The strengths and weakness of the first form of the Barycentric Interpolation formula are:

- Thought we still need $O(n^2)$ flops for calculating some quantities independent of x, we need only $O(n)$ flops for evaluating $P_n(x)$ once these numbers are known.
- Allow incremental interpolation.

When incorporate a new node x_{n+1}.

- It costs $n+1$ flops to evaluate the new w_j, $j = 0,1,\ldots,n$.

All we have to do is to divide each w_j, $j=0,1,\ldots,n$ by $(x_j - x_{n+1})$

- It costs $n+1$ flops to compute w_{n+1}.

- Numerical Stability.[11]

That is, the first form of the Barycentric Interpolation could be updated with $O(n)$ flops!

2) 3.2 The second form of the Barycentric Interpolation

The first form of the Barycentric Interpolation can be modified to more elegant formula, which is often used in practice.

Consider the constant function $f(x)=1$.

Use (7) to interpolate it, it follows that

$$1=\sum_{i=0}^{n} l_i(x) = l(x)\sum_{i=0}^{n}\frac{w_i}{x-x_i} \tag{8}$$

Dividing (7) by (8), then we obtain

$$P_n(x)=\frac{\sum_{i=0}^{n}\frac{w_i}{x-x_i}f_i}{\sum_{i=0}^{n}\frac{w_i}{x-x_i}} \tag{9}$$

Rutishauser [5] called (9) " the second(true) form of the Barycentric Interpolation formula".

It needs about $O(n^2)$ flops to compute the coefficients w_i, which is independent of x. Once the coefficients w_i are obtained, one evaluation of (9) requires $2n+3$ multiplications (resp. divisions) and $3n+1$ additions(resp. subtractions). And the form (9) exhibits a numerical stability. [9]

3) 3.3 Conversion between the Newton Interpolation and the Lagrange Interpolation

Given the interpolating polynomial P_n in Newton form,

$$P_n(x)=\sum_{i=0}^{n} a_i \prod_{j=0}^{i-1}(x-x_j) \tag{10}$$

where $a_i = f[x_0,x_1,\ldots,x_i]$ and the interpolating polynomial P_n in Lagrange form,

$$P_n(x)=\sum_{i=0}^{n}\sigma_i \prod_{j=0,j\neq i}^{n}(x-x_j) \tag{11}$$

where $\sigma_i := w_i P(x_i), i=0,1,\ldots,n$

By the result on divided differences (Milne-Thompson[12], Steffensen[13]),

We have:

$$a_k=\sum_{i=0}^{k}\frac{P(x_i)}{\prod_{j=0,j\neq i}^{k}(x_i-x_j)}=\sum_{i=0}^{k}\sigma_i\prod_{j=k+1}^{n}(x_i-x_j), k=0,1,\ldots,n$$

This is equivalent to the following triangular system of linear equations[10]:

$$\begin{pmatrix} a_0\\ a_1\\ a_2\\ \vdots\\ a_{n-1}\\ a_n \end{pmatrix} = \begin{pmatrix} \prod_{j=1}^{n}(x_0-x_j) & 0 & 0 & \cdots & 0\\ \prod_{j=2}^{n}(x_0-x_j) & \prod_{j=2}^{n}(x_1-x_j) & 0 & \cdots & 0\\ \prod_{j=3}^{n}(x_0-x_j) & \prod_{j=3}^{n}(x_1-x_j) & \prod_{j=3}^{n}(x_2-x_j) & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ x_0-x_n & x_1-x_n & x_2-x_n & \cdots & 0\\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} \sigma_0\\ \sigma_1\\ \sigma_2\\ \vdots\\ \sigma_{n-1}\\ \sigma_n \end{pmatrix} \tag{12}$$

4) 3.4 Conversion of the Newton Form into the Lagrange Form

Given the interpolating polynomial $P_n(x)$ in Newton form (10).

We can compute the Lagrange form of P_n by solving (12).

(That is, given $a_{i'}s$, we can compute $\sigma_{i'}s$.)

Using Gauss-elimination, we get the algorithm

$a_k^{(0)} := a_k$	$k=0,1,\ldots,n$
$a_k^{(i)} := a_k^{(i-1)}/(x_k - x_i)$	$k=0,1,\ldots,i-1;$ $i=1,2,\ldots,n$
$a_i^{(k+1)} := a_i^{(k)} - a_k^{(i)}$	$k=0,1,\ldots,i-1;$ $i=1,2,\ldots,n$
$\sigma_i := a_i^{(n)}$	$i=0,1,\ldots,n$

It costs $\frac{1}{2}n(n+1)$ divisions and $n(n+1)$ additions for the transformation of a Newton interpolating polynomial into its Lagrangian representation.

5) 3.5 Conversion of the Lagrange Form into the Newton Form

Given the interpolating polynomial $P_n(x)$ in Lagrange form (11).

We can compute the Newton form of P_n by solving (12).

(That is, given $\sigma_{i'}s$, we can compute $a_{i'}s$.)

We get the algorithm

$\sigma_i^{(0)} := \sigma_i,$	$i=0,1,\ldots,n$
$\sigma_{n-k}^{(k+1)} := \sum_{i=0}^{n-k}\sigma_i^{(k)}$	$k=0,1,\ldots,n$
$\sigma_i^{(k+1)} := (x_i - x_{n-k})\sigma_i^{(k)}$	$k=0,1,\ldots$ $n-k-a,\ldots$ $i-k,$
$a_i := \sigma_i^{(n+1-i)}$	$i=0,1,\ldots,n$

This algorithm also requires $\frac{1}{2}n(n+1)$ divisions and $n(n+1)$ additions for the transformation of a Lagrange interpolating polynomial into its Newtonian representation.

6) 3.6 Efficient Computation of the Barycentric Weight

For $f_i = 1, i = 0,1,\ldots,n$, (6) reads

$$1 = \sum_{i=0}^{n} w_i \prod_{j=0, j\neq i}^{n} (x - x_j) \qquad (13)$$

An application of algorithm in section 0 to (13) yields an algorithm for the efficient computation of the quantities $w_i, i = 0,1,\ldots,n$. We named this algorithm as Werner's Method. [10]

• Werner's Method

$a_0^{(0)} := 1, a_k^{(0)} := 0$	$k = 1,1,\ldots,n$
$a_k^{(i)} := a_k^{(i-1)}/(x_k - x_i)$	$k = 0,1,\ldots,i-1; i = 1,1,\ldots,n$
$a_i^{(k+1)} := a_i^{(k)} - a_k^{(i)}$	$k = 0,1,\ldots,i-1; i = 1,1,\ldots,n$
$w_i := a_i^{(n)}$	$i = 0,1,\ldots,n$

Thus we have shown that the coefficients of the Lagrangian form of the interpolating polynomial can be calculated with approximately $n^2/2$ divisions and n^2 additions (resp. subtractions).

IV. Numerical Results and Discussion

In this Section, we evaluate the operations of Polynomial Interpolations and use the numerical results to compare Polynomial Interpolations .

1) 4.1 Operations of all kinds of Interpolating polynomials

First of all, we compare the operations of Divided differences and Barycentric weights.

	additions /subtrations	divisions /multiplications
Divided difference		
$f[x_0, x_1, \ldots, x_{j-1}, x_j], j = 1,\ldots,n$	$O(n^2)$	$O(\frac{n^2}{2})$
Barycentric weight		
$w_j := \frac{1}{\prod_{i=0, i\neq j}^{n} (x_j - x_i)}, j = 0,\ldots,n$	$O(n^2)$	$O(n^2)$

$$f[x_0, x_1, \ldots, x_{j-1}, x_j] := \frac{f[x_1, \cdots, x_j] - f[x_0, \cdots, x_{j-1}]}{x_j - x_0}$$

The Divided difference and Barycentric weight are both independent of x. Therefore once the Divided differences and Barycentric weights are computed respectively, the operations of Newton Interpolation and Barycentric Interpolation would decrease one degree.

The following table is the costs of all kinds of Polynomial Interpolation which we introduced in this thesis.

	additions /subtrations	divisions /multiplications
Newton Interpolation	$O(n^2)$	$O(n^2)$
Newton Interpolation (given $a_{j'}s$)	$O(n)$	$O(\frac{n^2}{2})$
Krogh Algorithm I(given $a_{j'}s$)	$O(n)$	$O(\frac{n^2}{2})$
Krogh Algorithm II(given $a_{j'}s$)	$O(n)$	$O(n)$
Lagrange Interpolation	$O(n^2)$	$O(n^2)$
Aitken's Interpolation	$O(n^2)$	$O(n^2)$
Neville's Interpolation	$O(n^2)$	$O(n^2)$
Barycentric1 Interpolation(given $w_{j'}s$)	$O(n)$	$O(n)$
Barycentric2 Interpolation(given $w_{j'}s$)	$O(n)$	$O(n)$

$$a_j = f[x_0, x_1, \ldots, x_{j-1}, x_j] = \frac{f[x_1, \cdots, x_j] - f[x_0, \cdots, x_{j-1}]}{x_j - x_0}$$

$$w_j = \frac{1}{\prod_{i=0, i\neq j}^{n} (x_j - x_i)}$$

2) 4.2 Numerical Results and Discussion

We use three examples to study the error of eight kinds of Polynomial Interpolations. The computations were performed in MATLAB.

In the first example, we interpolate the Runge function

$$f(x) = \frac{1}{1 + 25x^2}$$

at n equal spacing points with eight kinds of interpolating polynomial.

The result is listed as follows:

1. Max error for Runge function at equal spacing points

	n=7	n=9	n=11
Newton Interpolation	0.6169	1.0452	1.9156
Krogh I Algorithm	0.6169	1.0452	1.9156
Krogh II Algorithm	0.6169	1.0452	1.9156
Lagrange Interpolation	0.6169	1.0452	1.9156
Aitken's Method	0.6169	1.0452	1.9156
Neville's Method	0.6169	1.0452	1.9156
Barycentric I formula	0.6169	1.0452	1.9156
Barycentric II formula	0.6169	1.0452	1.9156

	n=15	n=17	n=19
Newton Interpolation	7.1923	14.3868	29.1862
Krogh I Algorithm	7.1923	14.3868	29.1862
Krogh II Algorithm	7.1923	14.3868	29.1862
Lagrange Interpolation	7.1923	14.3868	29.1862
Aitken's Method	7.1923	14.3868	29.1862
Neville's Method	7.1923	14.3868	29.1862
Barycentric I formula	7.1923	14.3868	29.1862
Barycentric II formula	7.1923	14.3868	29.1862

The numerical results show that no matter what kinds of interpolating polynomial we choose, the max error grow vastly as $n \to \infty$ if the nodes are equidistant.

Now we use MATLAB to plot the Runge function and the interpolating polynomial in the same figure.

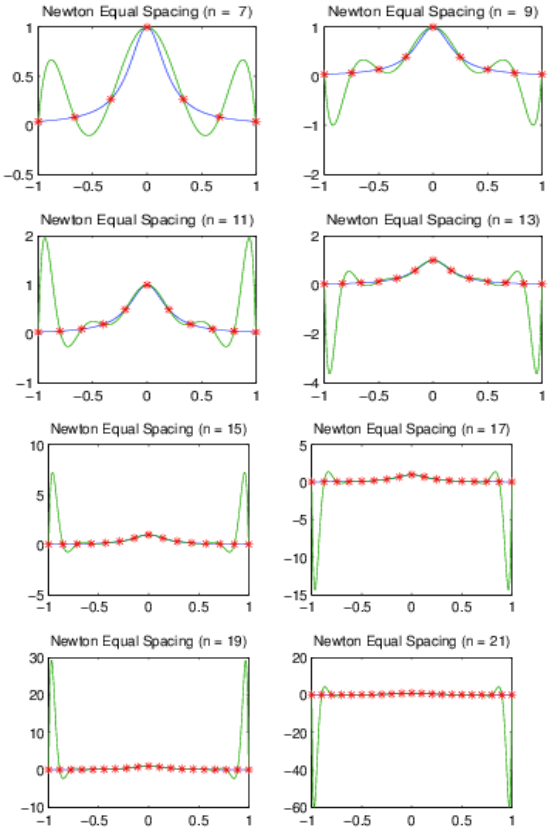
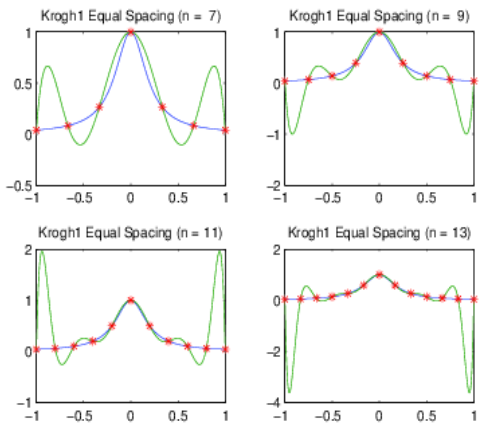


Fig. 3. Newton Interpolation of the Runge function in n equal spacing points on [-1, 1]



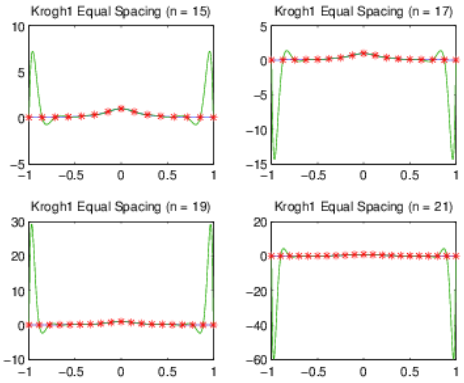


Fig. 4. Krogh 1 Interpolation of the Runge function in n equal spacing points on [-1, 1]

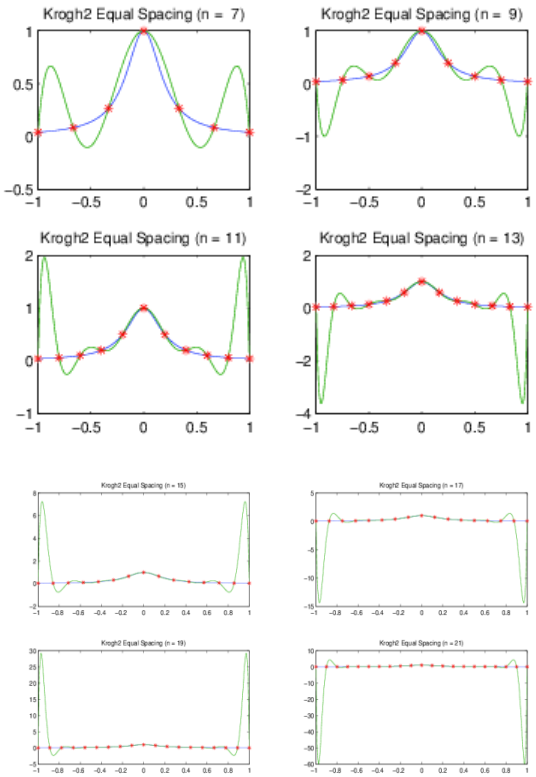
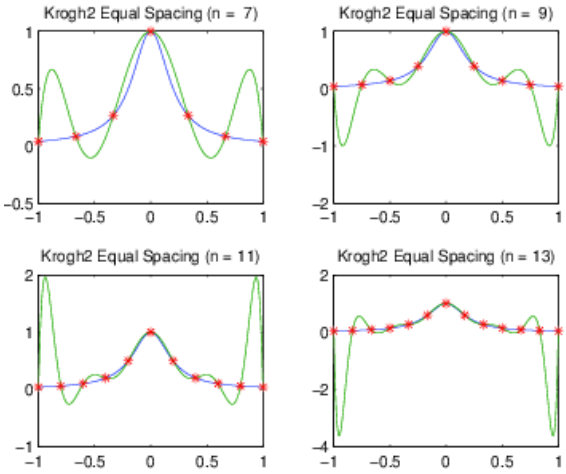


Fig. 5. Krogh 2 Interpolation of the Runge function in n equal spacing points on [-1, 1]



International Journal of Applied Science and Mathematics
Volume 4, Issue 5, ISSN (Online): 2394-2894

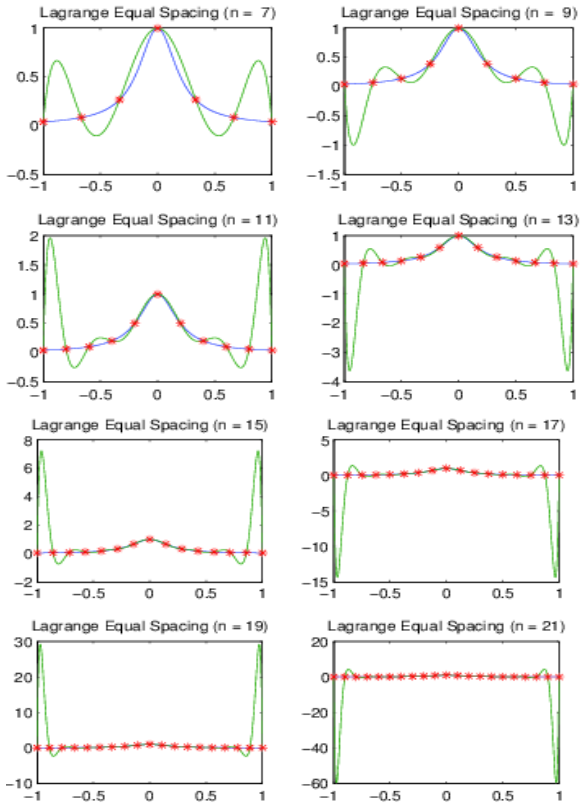


Fig. 6. Lagrange Interpolation of the Runge function in n equal spacing points on [-1, 1]

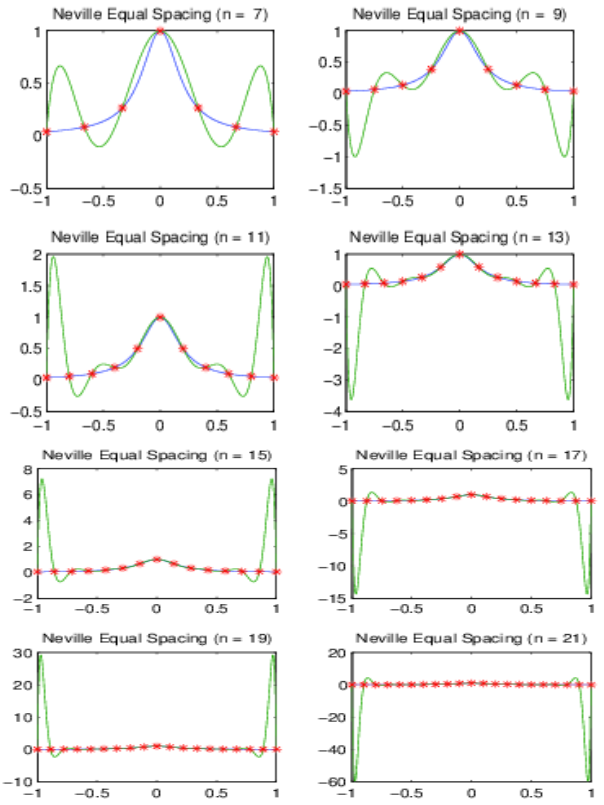


Fig. 8. Neville Interpolation of the Runge function in n equal spacing points on [-1, 1]

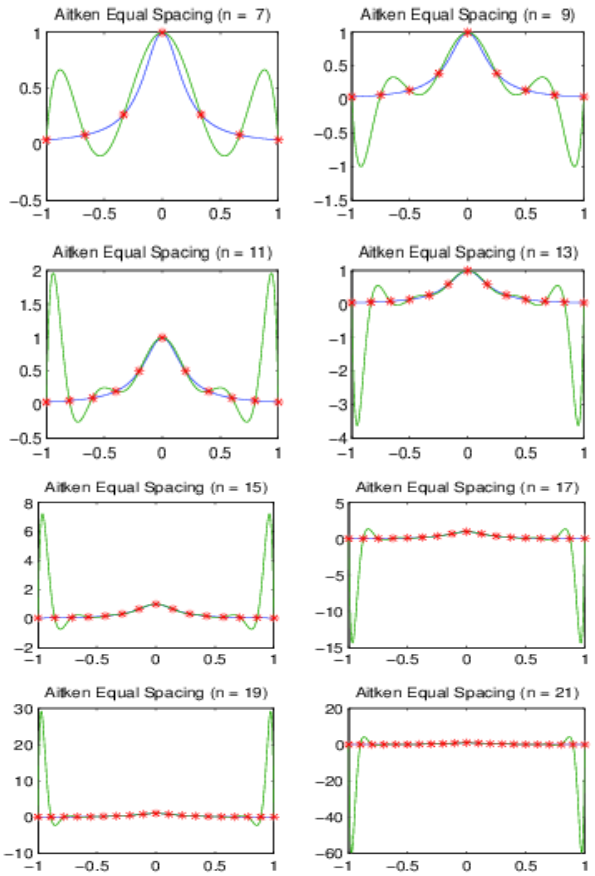


Fig. 7. Aitken Interpolation of the Runge function in n equal spacing points on [-1, 1]

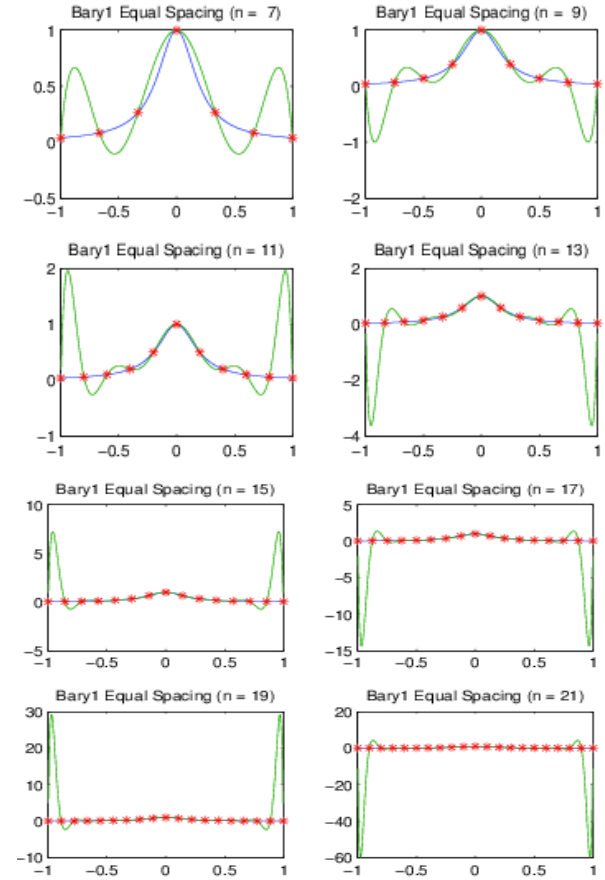


Fig. 9. Barycentric1 Interpolation of the Runge function in n equal spacing points on [-1, 1]

International Journal of Applied Science and Mathematics
Volume 4, Issue 5, ISSN (Online): 2394-2894

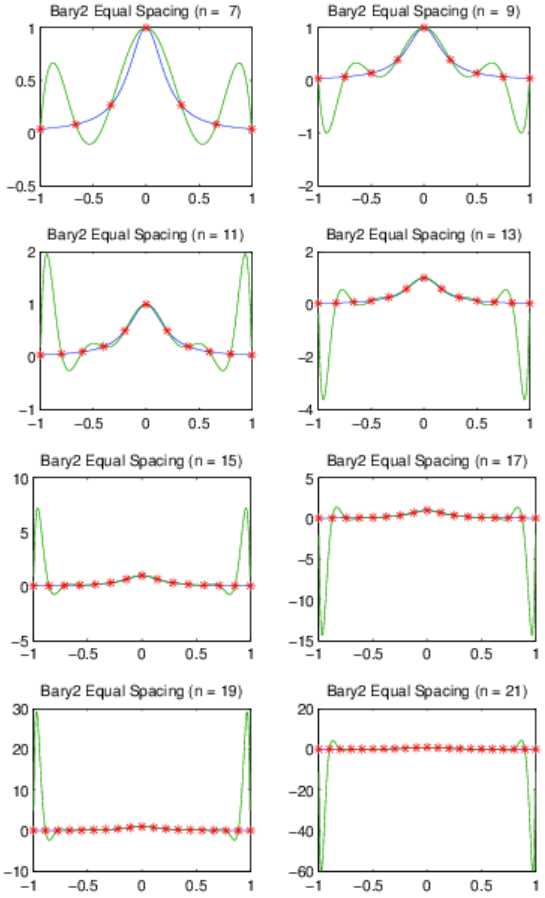


Fig. 10. Barycentric2 Interpolation of the Runge function in n equal spacing points on [-1, 1]

In Fig 3, Fig 4, Fig 5, Fig 6, Fig 7, Fig 8, Fig 9 and Fig 10.

It is obviously that all kinds of interpolating polynomial become unstable when x approaches the edge of the interval.

Now we take the Chebyshev Points as the nodes and run the numerical experiment again, The result is listed as follows:

2. Max error for Runge function at Chebyshev points

	n=7	n=9	n=11	n=13
Newton Interpolation	0.2642	0.1708	0.1092	0.0692
KroghI Algorithm	0.2642	0.1708	0.1092	0.0692
KroghII Algorithm	0.2642	0.1708	0.1092	0.0692
Lagrange Interpolation	0.2642	0.1708	0.1092	0.0692
Aitken's Method	0.2642	0.1708	0.1092	0.0692
Neville's Method	0.2642	0.1708	0.1092	0.0692
BarycentricI formula	0.2642	0.1708	0.1092	0.0692
BarycentricII formula	0.2642	0.1708	0.1092	0.0692

The numerical results show that no matter what kinds of interpolating polynomial we take, As long as we take the Chebyshev Points as nodes, the max error decrease as $n \to \infty$. That is, the interpolating polynomial of Runge function is numerical stable at Chebyshev Points. Now let us take a look at the following figures.

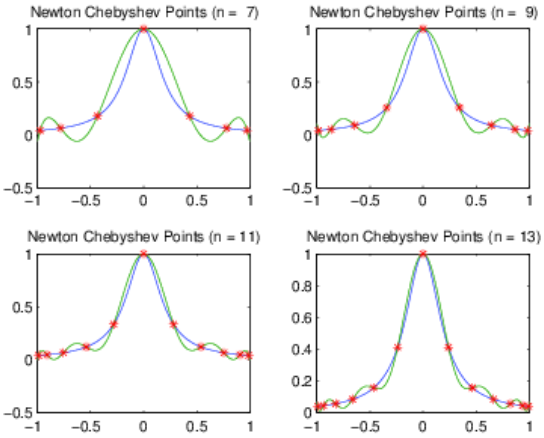


Fig. 11. Newton Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

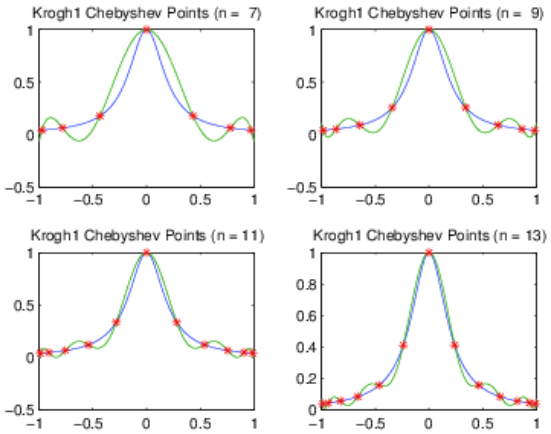


Fig. 12. Krogh1 Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

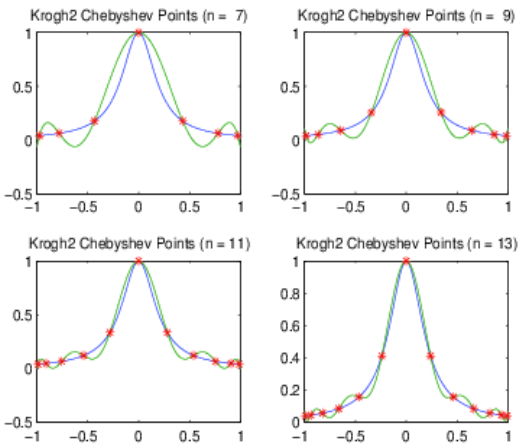


Fig. 13. Krogh2 Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

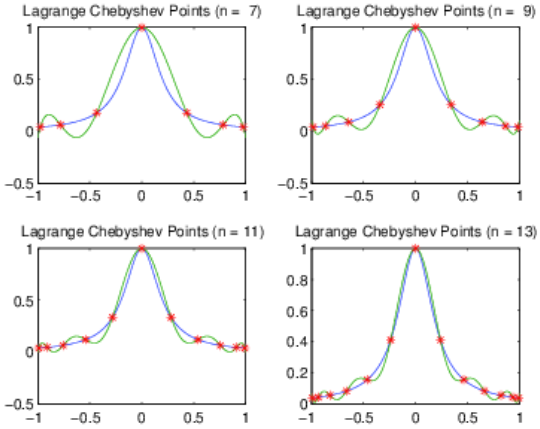


Fig. 14. Lagrange Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

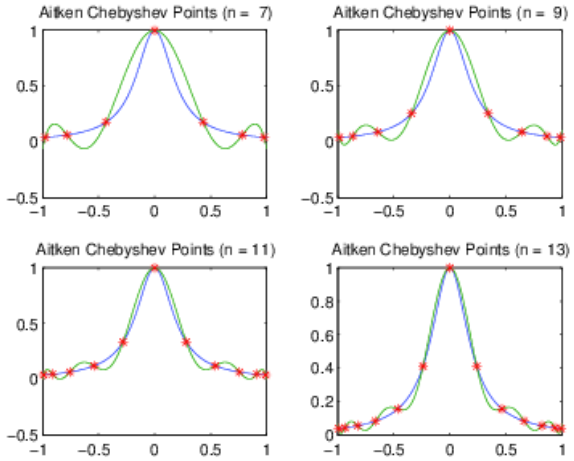


Fig. 15. Aitken Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

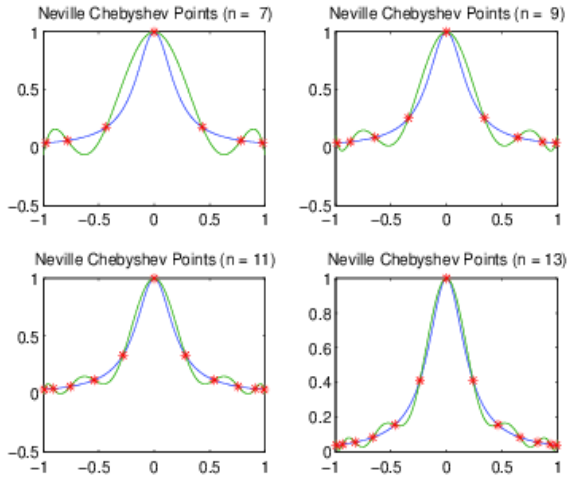


Fig. 6. Neville Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

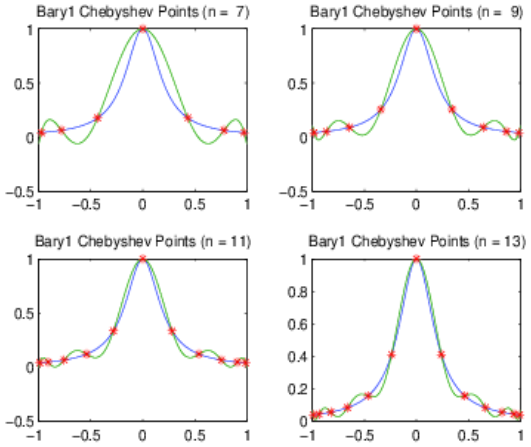


Fig. 17. Barycentric1 Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on [-1, 1]

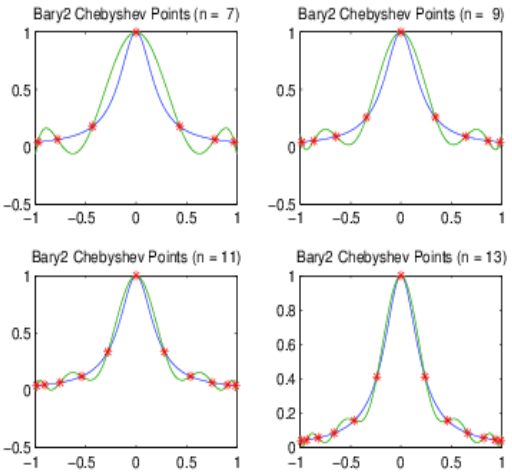


Fig. 18. Barycentric2 Interpolation of the Runge function $\frac{1}{1+25x^2}$ in n Chebyshev Points on $[-1,1]$

From Fig 11, Fig 12, Fig 13, Fig 14, Fig 15, Fig 16, Fig 17 and Fig 18. We find that each kind of interpolating polynomials performs well.

At last, we interpolate the natural logarithm function at equal spacing points with those interpolating polynomials. The numerical result is listed as follows:

3. Max error for Natural Log function at equal spacing points

	n=7	n=9	n=11	n=13
Newton Interpolation	4.4693 e-06	1.7591 e-07	7.6440 e-09	.5402 e-10
KroghI Algorithm	4.4693 e-06	.7591 e-07	7.6440 e-09	3.5402 e-10
KroghII Algorithm	4.4693 e-06	.7591 e-07	7.6440 e-09	.5402 e-10
Lagrange Interpolation	4.4693 e-06	.7591 e-07	7.6440 e-09	.5401 e-10
Aitken's Method	4.4693 e-06	.7591 e-07	7.6440 e-09	.5385 e-10
Neville's Method	4.4693 e-06	.7591 e-07	7.6440 e-09	.5402 e-10
Barycentric I formula	4.4693 e-06	1.7591 e-07	7.6440 e-09	.5402 e-10
Barycentric II formula	.4693 e-06	1.7591 e-07	7.6440 e-09	.5402 e-10

	n = 15	n = 17	n = 19
Newton Interpolation	1.7152e-11	8.5489e-13	8.5931e-14
KroghI Algorithm	1.7152e-11	8.4957e-13	9.8144e-14
KroghII Algorithm	1.7152e-11	8.4957e-13	9.8033e-14
Lagrange Interpolation	1.7154e-11	9.0844e-13	4.1300e-13
Aitken's Method	1.7865e-11	2.1104e-11	1.8385e-10
Neville's Method	1.7151e-11	8.6163e-13	1.1446e-13
Barycentric I formula	2.5815e-11	3.1330e-11	9.3865e-10

From the numerical results we know that all interpolating polynomial we discussed here are stable for natural logarithm function.

When $n \le 13$, the max error of all interpolating polynomial are similar. But along with n increase, the Newton Interpolation is more stable than others. The Newton Interpolation is slightly better than the Lagrange Interpolation. The Newton Interpolation is slightly better than Krogh1 algorithm and Krogh2 algorithm. The Neville's Method is better than the Aitken's Method. The Barycentric2 Interpolation is better than the Barycentric1 Interpolation.

Now let us take a look at the following figures:

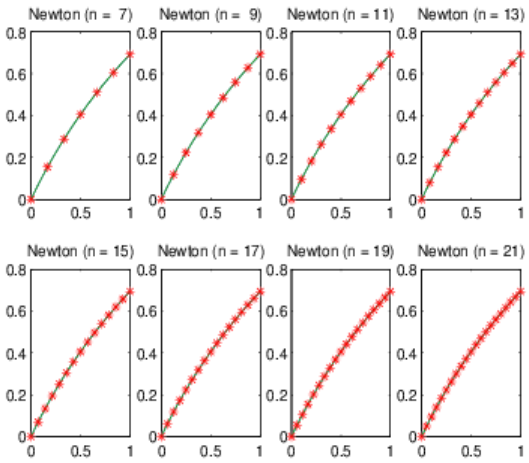


Fig. 19. Newton Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

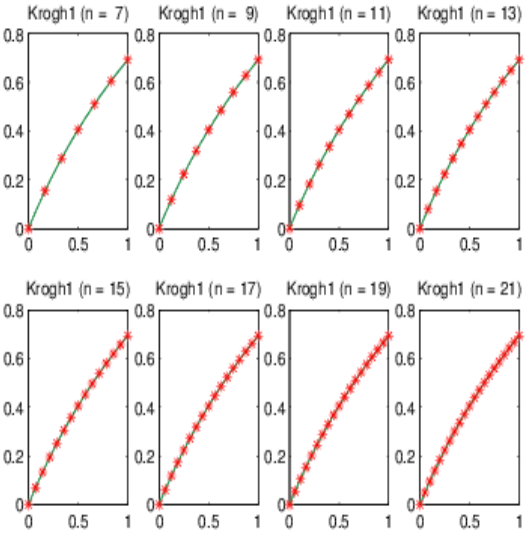


Fig. 20. Krogh1 Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

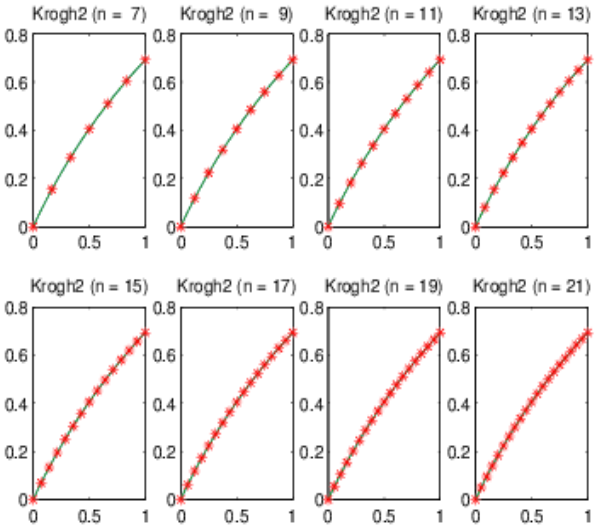


Fig. 21. Krogh2 Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

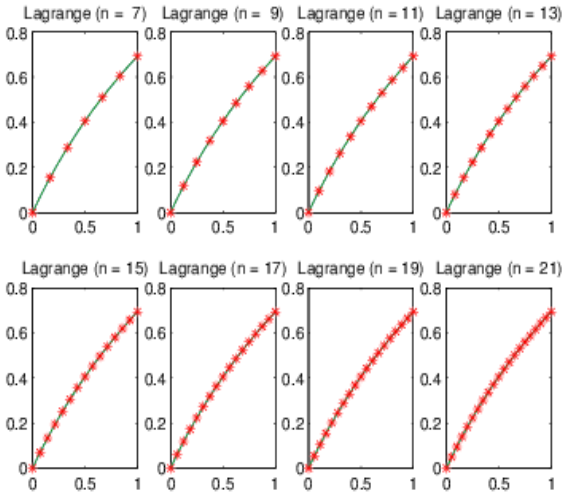


Fig. 22. Lagrange Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

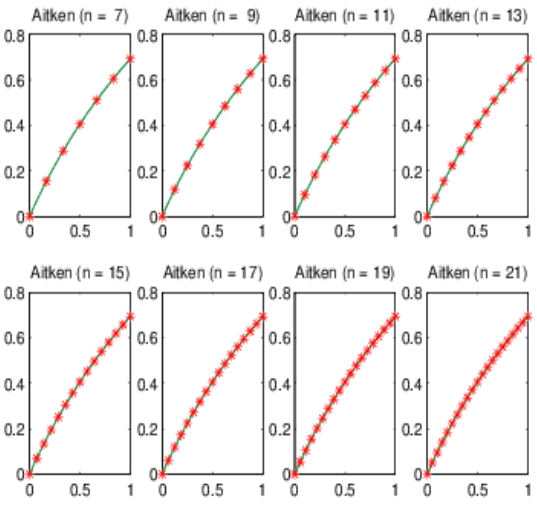


Fig. 23. Aitken Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

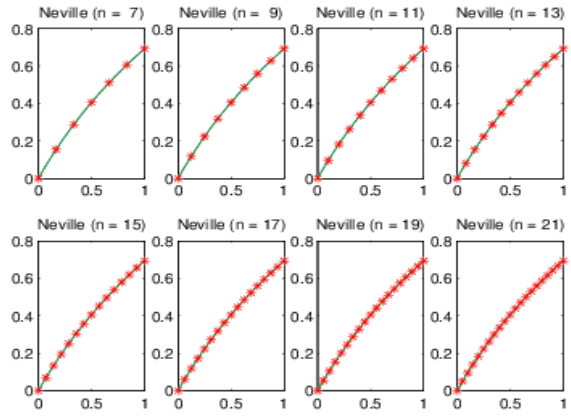


Fig. 24. Neville Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

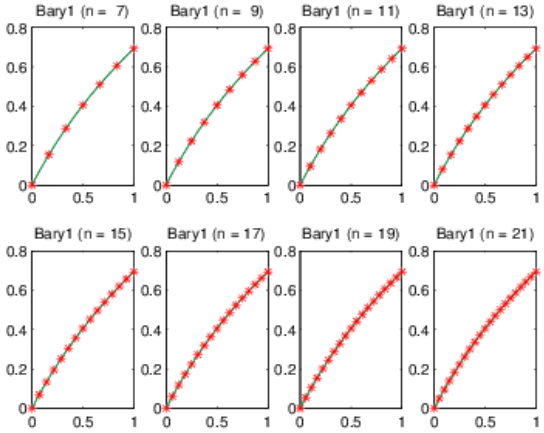


Fig. 25. Barycentric1 Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

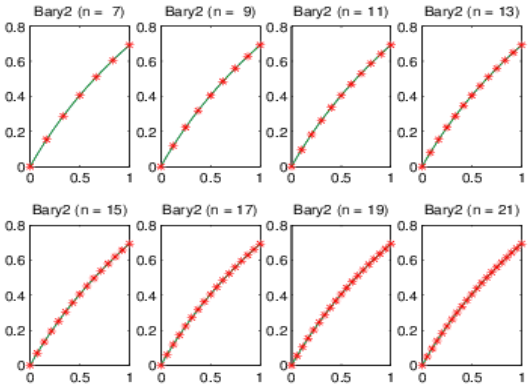


Fig. 26. Barycentric2 Interpolation of the Natural Logarithm function $ln(1+x)$ in n equal spacing points on [-1, 1]

From Fig 19, Fig 20, Fig 21, Fig 22, Fig 23, Fig 24, Fig 25 and Fig 26, we find that each kind of interpolating polynomials performs well.

Since the natural logarithm function $\log(1+x)$ is quite smooth on $[0,1]$, therefore the difference for each method is almost the same, even the degrees of polynomial ranging from $n=7$ to $n=21$.

V. Conclusions

In most textbooks of numerical analysis, Newton and Lagrange Interpolation are the most popular methods for polynomial interpolation. The textbooks often refer the superiority of Newton over Lagrange. However Berrut and Trefethen [11] pointed out that Barycentric interpolation is rarely known to most students. We study several Polynomial Interpolation Schemes and compare the error of the approximation.

Among the Polynomial Interpolations we study here, Lagrange Interpolation is praised for analytic utility and beauty but deplored for numerical practice. Newton Interpolation and Barycentric Interpolation are both good choice for numerical practice. Because they both require only $O(n)$ operations while $a_i's$ or $w_i's$ are known. Furthermore, Newton formula gives better accuracy than any other interpolating polynomials we discuss here.

References

[1] Kendall E. Atkinson, *An Introduction to Numerical Analysis*, 2nd. ed., John Wiley & Sons, 1989, New York.
[2] Wikipedia, Polynomial Interpolation Available: http://en.wikipedia.org/wiki/Polynomial_interpolation
[3] Autar Kaw and Michael Keteltas, Shortest path of a robot, Available: http://numericalmethods.eng.usf.edu .
[4] J.Behrens, Numerical Analysis I-Proof of Interpolation Error Theorem. Available:http://www.3.ma.tum.de/foswiki/pub/NumericalAnalysisI0506/Contents/proof-20051219.pdf.
[5] H.Rutishauser, Lectures on Numerical Mathematics, Boston, 1990 .
[6] F.S. Acton. Numerical Methods That [Usually] Work, AMS, Providence, 1990.
[7] Benjamin F. Plybon. An Introduction to Applied Numerical Analysis. PWS-KENT Publishing Company, Boston, 1992.
[8] W. J. Taylor, Method of Lagrangian Curvilinear Interpolation. J. Res. Nat. Bur. Standards, Vol. 35, 151-155, 1945.
[9] P. Henrici, Essentials of Numerical Analysis. Wiley, New Youk, 1982.
[10] Wilhelm Werner, Polynomial Interpolation: Lagrange versus Newton. Mathematics of Computation, Vol.43, 205-208, 1984.
[11] Jean-Paul Berrut and Lloyd N. Trefethen, Barycentric Lagrange Interpolation. SIAM Rev, Vol. 46, No 3, 501-517, 2004.
[12] L. M. Milne-Thompson. The Calculus of Finite Differences. Macmillan, 1933, London.
[13] J. F. Steffensen, Interpolation. Chelsea, New York, 1950.

Authors' Profiles

Dr. Jen-Yuan Chen received the degree of Doctor of Philosophy in 1997 in the field of Mathematics at the University of Texas at Austin USA. He specialized in matrix computation. He is working as an Associate Professor in Department of Mathematics of National Kaohsiung Normal University in Taiwan.

Ya-Fang Wang received the degree of Master of Mathematics in 2013 at the National Kaohsiung Normal University in Taiwan. Her specialty is numerical analysis. She is working as a mathematics teacher at Kaohsiung Municipal Nanzih Junior High School in Taiwan.