

Introduction

“[...] it is rare to have the luxury of quadratic convergence.”
(Burden-Faires, p.86^{9th})

There are a number of methods for squeezing faster convergence out of an already computed sequence of numbers.

We here explore one method which seems the have been around since the beginning of numerical analysis... Aitken’s Δ² method. It can be used to accelerate convergence of a sequence that is linearly convergent, regardless of its origin or application.

A review of extrapolation methods can be found in:

Recall: Convergence of a Sequence

Definition

Suppose the sequence \( \{p_n\}_{n=0}^{\infty} \) converges to \( p \), with \( p_n \neq p \) for all \( n \). If positive constants \( \lambda \) and \( \alpha \) exists with

\[
\lim_{n \to \infty} \frac{|p_{n+1} - p|}{|p_n - p|^\alpha} = \lambda,
\]

then \( \{p_n\}_{n=0}^{\infty} \) converges to \( p \) of order \( \alpha \), with asymptotic error constant \( \lambda \).

Linear convergence means that \( \alpha = 1 \), and \( \lambda \in (0, 1) \).
Aitken’s $\Delta^2$ Method

Assume $\{p_n\}_{n=0}^\infty$ is a linearly convergent sequence with limit $p$. Further, assume we are far out into the tail of the sequence (not large), and the signs of the successive errors agree, i.e.

$$\text{sign}(p_n - p) = \text{sign}(p_{n+1} - p) = \text{sign}(p_{n+2} - p) = \ldots$$

so that

$$\frac{p_{n+2} - p}{p_{n+1} - p} \approx \frac{p_{n+1} - p}{p_n - p} \approx \lambda \quad \text{(the asymptotic limit)}.$$

This would indicate

$$(p_{n+1} - p)^2 \approx (p_{n+2} - p)(p_n - p),$$

$$p_{n+1}^2 - 2p_{n+1}p + p^2 \approx p_{n+2}p_n - (p_{n+2} + p_n)p + p^2.$$

We solve for $p$ and get...

We solve for $p$ and get...

$$p \approx \frac{p_{n+2}p_n - p_n^2}{p_{n+2} - 2p_{n+1} + p_n}.$$

A little bit of algebraic manipulation put this into the equivalent “classical” Aitken form:

$$\hat{p}_n = p = p_n - \frac{(p_{n+1} - p_n)^2}{p_{n+2} - 2p_{n+1} + p_n}.$$  

Aitken’s $\Delta^2$ Method is based on the assumption that the $\hat{p}_n$ we compute from $p_{n+2}$, $p_{n+1}$ and $p_n$ is a better approximation to the actual limit $p$.

The analysis needed to rigorously prove this is beyond the scope of this class, see e.g. Sidi’s book.

Aitken’s $\Delta^2$ Method

The Recipe

Consider the sequence $\{p_n\}_{n=0}^N$, where the sequence is generated by the fixed point iteration $p_{n+1} = \cos(p_n)$, $p_0 = 0$.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$p_n$</th>
<th>$\hat{p}_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000000000000</td>
<td>0.685073357326045</td>
</tr>
<tr>
<td>1</td>
<td>1.000000000000000</td>
<td>0.728010361467617</td>
</tr>
<tr>
<td>2</td>
<td>0.540302305868140</td>
<td>0.73665164585231</td>
</tr>
<tr>
<td>3</td>
<td>0.857553215846393</td>
<td>0.73690294340474</td>
</tr>
<tr>
<td>4</td>
<td>0.654289790497779</td>
<td>0.73805042371664</td>
</tr>
<tr>
<td>5</td>
<td>0.93480358742566</td>
<td>0.73863696881655</td>
</tr>
<tr>
<td>6</td>
<td>0.01368787322757</td>
<td>0.73887658217136</td>
</tr>
<tr>
<td>7</td>
<td>0.736959682900654</td>
<td>0.73899243027034</td>
</tr>
<tr>
<td>8</td>
<td>0.72210245206708</td>
<td>0.739042511328159</td>
</tr>
<tr>
<td>9</td>
<td>0.750417761763761</td>
<td>0.739065949599941</td>
</tr>
<tr>
<td>10</td>
<td>0.731404042422510</td>
<td>0.739076383318956</td>
</tr>
<tr>
<td>11</td>
<td>0.744237354900557</td>
<td>0.739081177259563</td>
</tr>
<tr>
<td>12</td>
<td>0.735604740436347</td>
<td>0.73908333909684</td>
</tr>
</tbody>
</table>

Note: Bold digits are correct; $\hat{p}_{11}$ needs $p_{13}$, and $\hat{p}_{12}$ additionally needs $p_{14}$. 

#4: Solutions of Equations in One Variable — (6/59)
Steffensen’s Method: Fixed-Point Iteration on Steroids

Suppose we have a (linearly converging) fixed point iteration:

\[ p_0, \quad p_1 = g(p_0), \quad p_2 = g(p_1), \quad \ldots \]

Once we have \( p_0, p_1 \) and \( p_2 \), we can compute

\[ \hat{p}_0 = p_0 - \frac{(p_1 - p_0)^2}{p_2 - 2p_1 + p_0}. \]

At this point we “restart” the fixed point iteration with \( p_0 = \hat{p}_0 \), e.g.

\[ p_3 = \hat{p}_0, \quad p_4 = g(p_3), \quad p_5 = g(p_4), \]

and compute

\[ \hat{p}_3 = p_3 - \frac{(p_4 - p_3)^2}{p_5 - 2p_4 + p_3}. \]

3° If at some point \( p_2 - 2p_1 + p_0 = 0 \) (which appears in the denominator), then we stop and select the current value of \( p_2 \) as our approximate answer.

Both Newton’s and Steffensen’s methods give quadratic convergence. In Newton’s method we compute one function value and one derivative in each iteration. In Steffensen’s method we have two function evaluations and a more complicated algebraic expression in each iteration, but no derivative. It looks like we got something for (almost) nothing. However, in order the guarantee quadratic convergence for Steffensen’s method, the fixed point function \( g \) must be 3 times continuously differentiable, e.g. \( f \in C^3[a, b] \), (see theorem-2.15 in Burden-Faires\textsuperscript{9th}). Newton’s method “only” requires \( f \in C^2[a, b] \) (BF\textsuperscript{9th} theorem-2.6).
Aitken’s $\Delta^2$ and Steffensen’s Methods

Consider the sequence $\{p_n\}_{n=0}^{\infty}$, where the sequence is generated by the fixed point iteration $p_{n+1} = \cos(p_n)$, $p_0 = 0$.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$p_n$</th>
<th>Aitken-$\hat{p}_n$</th>
<th>Steffensen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000000000000000</td>
<td>0.685073357326045</td>
<td>0.0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>1.0000000000000000</td>
<td>0.728010361467617</td>
<td>1.0000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>0.540302305868140</td>
<td>0.73366164585231</td>
<td>0.540302305868140</td>
</tr>
<tr>
<td>3</td>
<td>0.857553215846393</td>
<td>0.736906294340474</td>
<td>0.685073357326045$^a$</td>
</tr>
<tr>
<td>4</td>
<td>0.654289790497779</td>
<td>0.738050421371664</td>
<td>0.74372633807905</td>
</tr>
<tr>
<td>5</td>
<td>0.703480358742566</td>
<td>0.738630698618155</td>
<td>0.714859871642984</td>
</tr>
<tr>
<td>6</td>
<td>0.701368773622757</td>
<td>0.738876582817136</td>
<td>0.738660156167714$^a$</td>
</tr>
<tr>
<td>7</td>
<td>0.706395682900654</td>
<td>0.73892243027034</td>
<td>0.73973136116415</td>
</tr>
<tr>
<td>8</td>
<td>0.70722102425026708</td>
<td>0.739042511328159</td>
<td>0.738892313230713</td>
</tr>
<tr>
<td>9</td>
<td>0.70750417761763761</td>
<td>0.73906594959941</td>
<td>0.739085106356719$^a$</td>
</tr>
<tr>
<td>10</td>
<td>0.70731404042422510</td>
<td>0.73907638318956</td>
<td>0.739085151307330</td>
</tr>
<tr>
<td>11</td>
<td>0.70744237354900557</td>
<td>0.739081177259563</td>
<td>0.739085121028058</td>
</tr>
<tr>
<td>12</td>
<td>0.70735604740436347</td>
<td>0.739083333909684</td>
<td>0.739085133215161$^a$</td>
</tr>
</tbody>
</table>

Note: Bold digits are correct; $\hat{p}_{11}$ needs $p_{13}$, and $\hat{p}_{12}$ additionally needs $p_{14}$.
Zeros of Polynomials

Definition: Degree of a Polynomial

A **polynomial of degree** $n$ has the form

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \quad a_n \neq 0$$

where the $a_i$’s are constants (either real, or complex) called the **coefficients** of $P$.

Why look at polynomials? — We’ll be looking at the problem $P(x) = 0$ (i.e. $f(x) = 0$ for a special class of functions.) Polynomials are the basis for many approximation methods, hence being able to solve polynomial equations fast is valuable.

We’d like to use Newton’s method, so we need to compute $P(x)$ and $P'(x)$ as efficiently as possible.
Horner’s Method: Evaluating Polynomials Quickly

Let

\[ P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0. \]

If we are looking to evaluate \( P(x_0) \) for any \( x_0 \), let

\[ b_n = a_n, \quad b_k = a_k + b_{k+1} x_0, \quad k = (n-1), (n-2), \ldots, 1, 0, \]

then \( b_0 = P(x_0) \). We have only used \( n \) multiplications and \( n \) additions.

At the same time we have computed the decomposition

\[ P(x) = (x - x_0) Q(x) + b_0, \]

where

\[ Q(x) = \sum_{k=0}^{n-1} b_{k+1} x^k. \]

Example#1: Horner’s Method

For \( P(x) = x^4 - x^3 + x^2 + x - 1 \), compute \( P(5) \):

\[
\begin{array}{ccccccc}
    x_0 = 5 & a_4 = 1 & a_3 = -1 & a_2 = 1 & a_1 = 1 & a_0 = -1 \\
    b_4 x_0 = 5 & b_3 x_0 = 20 & b_2 x_0 = 105 & b_1 x_0 = 530 \\
    b_4 = 1 & b_3 = 4 & b_2 = 21 & b_1 = 106 & b_0 = 529
\end{array}
\]

Hence, \( P(5) = 529 \), and

\[ P(x) = (x - 5)(x^3 + 4x^2 + 21x + 106) + 529 \]

Similarly we get \( P'(5) = Q(5) = 436 \)

\[
\begin{array}{ccccccc}
    x_0 = 5 & a_3 = 1 & a_2 = 4 & a_1 = 21 & a_0 = 106 \\
    b_3 x_0 = 5 & b_2 x_0 = 45 & b_1 x_0 = 330 \\
    b_3 = 1 & b_2 = 9 & b_1 = 66 & b_0 = 436
\end{array}
\]
Algorithm: Horner’s Method

Input: Degree \( n \); coefficients \( a_0, a_1, \ldots, a_n \); \( x_0 \)

Output: \( y = P(x_0), z = P'(x_0) \).

1. Set \( y = a_n, z = a_n \)
2. For \( j = (n-1), (n-2), \ldots, 1 \)
   - Set \( y = x_0y + a_j, z = x_0z + y \)
3. Set \( y = x_0y + a_0 \)
4. Output \((y, z)\)
5. End program

Deflation — Finding All the Zeros of a Polynomial

If we are solving our current favorite problem
\[
P(x) = 0, \quad P(x) \text{ a polynomial of degree } n,
\]
and we are using Horner’s method of computing \( P(x_i) \) and \( P'(x_i) \),
then after \( N \) iterations, \( x_N \) is an approximation to one of the roots of \( P(x) = 0 \).

We have
\[
P(x) = (x - x_N)Q(x) + b_0, \quad b_0 \approx 0.
\]

At this point, let \( \hat{r}_1 = x_N \) be the first root, and \( Q_1(x) = Q(x) \).
We can now find a second root by applying Newton’s method to \( Q_1(x) \).

Quality of Deflation

Now, the big question is \textit{“are the approximate roots \( \hat{r}_1, \hat{r}_2, \ldots, \hat{r}_n \) good approximations of the roots of \( P(x) \)??”}

Unfortunately, sometimes, no.

In each step we solve the equation to some tolerance, \( i.e. \)
\[
|b_0^{(k)}| < \text{tol}
\]

Even though we may solve to a tight tolerance \(10^{-8}\), the errors accumulate and the inaccuracies increase iteration-by-iteration...

Question: Is deflation therefore useless???
Improving the Accuracy of Deflation

The problem with deflation is that the zeros of \( Q_k(x) \) are not good representatives of the zeros of \( P(x) \), especially for high \( k \)'s. As \( k \) increases, the quality of the root \( \hat{r}_k \) decreases.

Maybe there is a way to get all the zeros with the same quality? The idea is quite simple... in each step of deflation, instead of just accepting \( \hat{r}_k \) as a root of \( P(x) \), we re-run Newton’s method on the full polynomial \( P(x) \), with \( \hat{r}_k \) as the starting point — a couple of Newton iterations should quickly converge to the root of the full polynomial.

Algorithm Outline: Improved Deflation

1. Apply Newton’s method to \( P(x) \) \( \rightarrow \hat{r}_1 \), \( Q_1(x) \).
2. For \( k = 2, 3, \ldots, (n - 2) \) do 3–4
3. Apply Newton’s method to \( Q_{k-1} \) \( \rightarrow \hat{r}_k \), \( Q_k(x) \).
4. Apply Newton’s method to \( P(x) \) with \( \hat{r}_k \) as the initial point \( \rightarrow \hat{r}_k \).
   Apply Horner’s method to \( Q_{k-1}(x) \) with \( x = \hat{r}_k \) \( \rightarrow Q_k(x) \).
5. Use the quadratic formula on \( Q_{n-2}(x) \) to get the two remaining roots.

Note: “Inside” Newton’s method, the evaluations of polynomials and their derivatives are also performed using Horner’s method.

Deflation & Improvement

Wilkinson Polynomials

The Wilkinson Polynomials

\[
P_n^W(x) = \prod_{k=1}^{n} (x - k)
\]

have the roots \( \{1, 2, \ldots, n\} \), but provide surprisingly tough numerical root-finding problems. (Additional details in Math 543.)

In the next few slides we show the results of Deflation and Improved Deflation applied to Wilkinson polynomials of degree 9, 10, 12, and 13.

Figure: [LEFT] The result of the two algorithms on the Wilkinson polynomial of degree 9: in this case the roots are computed so that \( |b_0^{(k)}| < 10^{-6} \). [RIGHT] The result of the two algorithms on the Wilkinson polynomial of degree 10: in this case the roots are computed so that \( |b_0^{(k)}| < 10^{-6} \). In both cases the lower line corresponds to improved deflation and we see that we get an improvement in the relative error of several orders of magnitude.
Müller’s Method

Müller’s method is an extension of the Secant method...

Recall that the secant method uses two points \( x_k \) and \( x_{k-1} \) and the function values in those two points \( f(x_k) \) and \( f(x_{k-1}) \). The zero-crossing of the linear interpolant (the secant line) is used as the next iterate \( x_{k+1} \).

Müller’s method takes the next logical step: it uses three points: \( x_k, x_{k-1} \) and \( x_{k-2} \), the function values in those points \( f(x_k), f(x_{k-1}) \) and \( f(x_{k-2}) \); a second degree polynomial fitting these three points is found, and its zero-crossing is the next iterate \( x_{k+1} \).

Next slide: \( f(x) = x^4 - 3x^3 - 1 \), \( x_{k-2} = 1.5 \), \( x_{k-1} = 2.5 \), \( x_k = 3.5 \).

One interesting / annoying feature of polynomials with real coefficients is that they may have complex roots, e.g. \( P(x) = x^2 + 1 \) has the roots \( \{-i, i\} \). Where by definition \( i = \sqrt{-1} \).

If the initial approximation given to Newton’s method is real, all the successive iterates will be real... which means we will not find complex roots.

One way to overcome this is to start with a complex initial approximation and do all the computations in complex arithmetic.

Another solution is Müllers Method...
Müller’s Method — Fitting the Quadratic Polynomial

We consider the quadratic polynomial

\[ m(x) = a(x - x_k)^2 + b(x - x_k) + c \]

at the three fitting points we get

\[ f(x_{k-2}) = a(x_{k-2} - x_k)^2 + b(x_{k-2} - x_k) + c \]
\[ f(x_{k-1}) = a(x_{k-1} - x_k)^2 + b(x_{k-1} - x_k) + c \]
\[ f(x_k) = c \]

We can solve for \( a, b, \) and \( c \):

\[
a = \frac{(x_{k-1} - x_k)(f(x_{k-2}) - f(x_k)) - (x_{k-2} - x_k)(f(x_{k-1}) - f(x_k))}{(x_{k-2} - x_k)(x_{k-1} - x_k)(x_{k-2} - x_{k-1})}
\]
\[
b = \frac{(x_{k-2} - x_k)^2(f(x_{k-1}) - f(x_k)) - (x_{k-1} - x_k)^2(f(x_{k-2}) - f(x_k))}{(x_{k-2} - x_k)(x_{k-1} - x_k)(x_{k-2} - x_{k-1})}
\]
\[
c = f(x_k)
\]

Müller’s Method — Algorithm

Algorithm: Müller’s Method

Input: \( x_0, x_1, x_2 \): tolerance \( tol \); max iterations \( N_0 \)

Output: Approximate solution \( p \), or failure message.

1. Set \( h_1 = (x_1 - x_0), h_2 = (x_2 - x_1), \delta_1 = [f(x_1) - f(x_0)]/h_1, \delta_2 = [f(x_2) - f(x_1)]/h_2, d = (\delta_2 - \delta_1)/(h_2 + h_1), j = 3. \]
2. While \( j \leq N_0 \) do 3–7
3. \( b = \delta_2 + h_2 d, D = \sqrt{b^2 - 4f(x_0)}d \) complex?
4. If \( |b - D| < |b + D| \) then set \( E = b + D \) else set \( E = b - D \)
5. Set \( h = -2f(x_0)/E, p = x_2 + h \)
6. If \( |h| < tol \) then output \( p \); stop program
7. Set \( x_0 = x_1, x_1 = x_2, x_2 = p, h_1 = (x_1 - x_0), h_2 = (x_2 - x_1), \delta_1 = [f(x_1) - f(x_0)]/h_1, \delta_2 = [f(x_2) - f(x_1)]/h_2, d = (\delta_2 - \delta_1)/(h_2 + h_1), j = j + 1 \)
8. output — “Müller’s Method failed after \( N_0 \) iterations.”

Müller’s Method — Identifying the Zero

We now have a quadratic equation for \( (x - x_k) \) which gives us two possibilities for \( x_{k+1} \):

\[ x_{k+1} - x_k = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \]

In Müller’s method we select

\[ x_{k+1} = x_k - \frac{2c}{b + \text{sign}(b)\sqrt{b^2 - 4ac}} \]

we are maximizing the (absolute) size of the denominator, hence we select the root closest to \( x_k \).

Note that if \( b^2 - 4ac < 0 \) then we automatically get complex roots.

Now We Know “Everything” About Solving \( f(x) = 0 \) !?

Let’s recap... Things to remember...

The relation between root finding \( (f(x) = 0) \) and fixed point \( (g(x) = x) \).

Key algorithms for root finding: Bisection, Secant Method, and Newton’s Method. — Know what they are (the updates), how to start (one or two points? bracketing or not bracketing the root?), can the method break, can breakage be fixed? Convergence properties.

Also, know the mechanics of the Regula Falsi method, and understand why it can run into trouble.

Fixed point iteration: Under what conditions do FP-iteration converge for all starting values in the interval?
Recap, continued...

Basic error analysis: order $\alpha$, asymptotic error constant $\lambda$. — Which one has the most impact on convergence? Convergence rate for general fixed point iterations?

Multiplicity of zeros: What does it mean? How do we use this knowledge to “help” Newton’s method when we’re looking for a zero of high multiplicity?


Zeros of polynomials: Horner’s method, Deflation (with improvement), Müllér’s method.

New Favorite Problem:

Interpolation and Polynomial Approximation

Weierstrass Approximation Theorem

The following theorem is the basis for polynomial approximation:

Theorem (Weierstrass Approximation Theorem)

Suppose $f \in C[a, b]$. Then $\forall \epsilon > 0 \exists$ a polynomial $P(x)$: $|f(x) - P(x)| < \epsilon$, $\forall x \in [a, b]$.  

Note: The bound is uniform, i.e. valid for all $x$ in the interval.

Note: The theorem says nothing about how to find the polynomial, or about its order.

Illustrated: Weierstrass Approximation Theorem

Figure: Weierstrass approximation Theorem guarantees that we (maybe with substantial work) can find a polynomial which fits into the “tube” around the function $f$, no matter how thin we make the tube.
Candidates: the Taylor Polynomials???

Natural Question:
Are our old friends, the Taylor Polynomials, good candidates for polynomial interpolation?

Answer:
No. The Taylor expansion works very hard to be accurate in the neighborhood of one point. But we want to fit data at many points (in an extended interval).

[Next slide: The approximation is great near the expansion point \( x_0 = 0 \), but get progressively worse as we get further away from the point, even for the higher degree approximations.]

Lookahead: Polynomial Approximation

Clearly, Taylor polynomials are not well suited for approximating a function over an extended interval.

We are going to look at the following:
- Lagrange polynomials — Neville’s Method. [This Lecture]
- Newton’s divided differences.
- Hermite interpolation.
- Cubic splines — Piecewise polynomial approximation.
- (Parametric curves)
- (Bézier curves)

Interpolation: Lagrange Polynomials

Idea: Instead of working hard at one point, we will prescribe a number of points through which the polynomial must pass.

As warm-up we will define a function that passes through the points \((x_0, f(x_0))\) and \((x_1, f(x_1))\). First, let’s define

\[
L_0(x) = \frac{x - x_1}{x_0 - x_1}, \quad L_1(x) = \frac{x - x_0}{x_1 - x_0},
\]

and then define the interpolating polynomial

\[
P(x) = L_0(x)f(x_0) + L_1(x)f(x_1),
\]

then \(P(x_0) = f(x_0)\), and \(P(x_1) = f(x_1)\).

- \(P(x)\) is the unique linear polynomial passing through \((x_0, f(x_0))\) and \((x_1, f(x_1))\).
An \( n \)-degree polynomial passing through \( n + 1 \) points

We are going to construct a polynomial passing through the points
\((x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_N, f(x_N))\).

We define \( L_{n,k}(x) \), the **Lagrange coefficients**:

\[
L_{n,k}(x) = \prod_{i=0, i\neq k}^{n} \frac{x-x_i}{x_k-x_i} = \frac{x-x_0}{x_k-x_0} \cdot \ldots \cdot \frac{x-x_{k-1}}{x_k-x_{k-1}} \cdot \frac{x-x_{k+1}}{x_k-x_{k+1}} \cdot \ldots \cdot \frac{x-x_n}{x_k-x_n},
\]

which have the properties

\[
L_{n,k}(x_k) = 1; \quad L_{n,k}(x_i) = 0, \quad \forall i \neq k.
\]

The \( n \)th Lagrange Interpolating Polynomial

We use \( L_{n,k}(x), k = 0, \ldots, n \) as building blocks for the Lagrange interpolating polynomial:

\[
P(x) = \sum_{k=0}^{n} f(x_k) L_{n,k}(x),
\]

which has the property

\[
P(x_i) = f(x_i), \quad \forall i = 0, \ldots, n.
\]

This is the unique polynomial passing through the points \((x_i, f(x_i)), i = 0, \ldots, n\).

Example of \( L_{n,k}(x) \)

This is \( L_{6,3}(x) \), for the points \( x_i = i, i = 0, \ldots, 6 \).

Error bound for the Lagrange interpolating polynomial

Suppose \( x_i, i = 0, \ldots, n \) are distinct numbers in the interval \([a, b]\), and \( f \in C^{n+1}[a, b] \). Then \( \forall x \in [a, b] \exists \xi(x) \in (a, b) \) so that:

\[
f(x) = P_{\text{Lagrange}}(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x-x_i),
\]

where \( P_{\text{Lagrange}}(x) \) is the \( n \)th Lagrange interpolating polynomial.

Compare with the error formula for Taylor polynomials

\[
f(x) = P_{\text{Taylor}}(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} (x-x_0)^{n+1},
\]

**Problem:** Applying the error term may be difficult...
The Lagrange and Taylor Error Terms

Just to get a feeling for the non-constant part of the error terms in the Lagrange and Taylor approximations, we plot those parts on the interval [0, 4] with interpolation points $x_i = i, i = 0, 1, \ldots, 4$:

![Graph showing the non-constant error terms for the Lagrange interpolation](image)

**Theorem**

Let $f$ be defined at $x_0, x_1, \ldots, x_k$, and $x_i$ and $x_j$ be two distinct points in this set, then

$$P(x) = \frac{(x - x_j)P_{0, \ldots, j-1, j+1, \ldots, k}(x) - (x - x_i)P_{0, \ldots, i-1, i+1, \ldots, k}(x)}{x_i - x_j}$$

is the $k^{th}$ Lagrange polynomial that interpolates $f$ at the $k + 1$ points $x_0, \ldots, x_k$. 

Increasing the degree of the Lagrange Interpolation

Applying (estimating) the error term is difficult.

The degree of the polynomial needed for some desired accuracy is not known until after cumbersome calculations — checking the error term.

**If we want to increase the degree of the polynomial (to e.g. $n + 1$) the previous calculations are of no help...**

**Building block for a fix:** Let $f$ be a function defined at $x_0, \ldots, x_n$, and suppose that $m_1, m_2, \ldots, m_k$ are $k (< n)$ distinct integers, with $0 \leq m_i \leq n \forall i$. The Lagrange polynomial that agrees with $f(x)$ the $k$ points $x_{m_1}, x_{m_2}, \ldots, x_{m_k}$ is denoted $P_{m_1, m_2, \ldots, m_k}(x)$.

**Note:** $\{m_1, m_2, \ldots, m_k\} \subset \{0, 1, \ldots, n\}$.

Recursive Generation of Higher Degree Lagrange Interpolating Polynomials

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$P_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$P_4$</td>
</tr>
</tbody>
</table>
### Neville’s Method

The notation in the previous table gets cumbersome... We introduce the notation $Q_{\text{Last,Degree}} = P_{\text{Last-Degree}...\text{Last}}$, the table becomes:

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$Q_{i,0}$</th>
<th>$Q_{i,1}$</th>
<th>...</th>
<th>$Q_{i,n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$P_{0,0}$</td>
<td>$P_{0,1}$</td>
<td>...</td>
<td>$P_{0,n}$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$P_{1,0}$</td>
<td>$P_{1,1}$</td>
<td>...</td>
<td>$P_{1,n}$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$P_{2,0}$</td>
<td>$P_{2,1}$</td>
<td>...</td>
<td>$P_{2,n}$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$P_{3,0}$</td>
<td>$P_{3,1}$</td>
<td>...</td>
<td>$P_{3,n}$</td>
</tr>
</tbody>
</table>

Compare with the old notation:

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$P_{i,0}$</th>
<th>$P_{i,1}$</th>
<th>...</th>
<th>$P_{i,n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$P_0$</td>
<td>$P_{0,1}$</td>
<td>...</td>
<td>$P_{0,n}$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$P_1$</td>
<td>$P_{1,1}$</td>
<td>...</td>
<td>$P_{1,n}$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$P_2$</td>
<td>$P_{1,2}$</td>
<td>...</td>
<td>$P_{2,n}$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$P_3$</td>
<td>$P_{2,3}$</td>
<td>...</td>
<td>$P_{3,n}$</td>
</tr>
</tbody>
</table>

### Algorithm: Neville’s Method — Iterated Interpolation

1. Initialize $Q_{i,0} = f(x_i)$.
2. 
   FOR $i = 1 : n$
     
     FOR $j = 1 : i$
     
     $Q_{i,j} = \frac{(x - x_{i-j})Q_{i,j-1} - (x - x_i)Q_{i-1,j-1}}{x_i - x_{i-j}}$
     
     END
     
     END

3. Output the $Q$-table.

### Homework #3

- Will open on 09/12/2014 at 09:30am PDT.
- Will close no earlier than 09/24/2014 at 09:00pm PDT.