

Algoritmo de Newton-Bernstein para Interpolación Polinomial

Derivación Formal Simplificada

Resumen

Se presenta la derivación formal simplificada del algoritmo de Newton-Bernstein de Ainsworth-Sánchez, que combina la forma de Newton del interpolante con propiedades de elevación de grado en la base de Bernstein. El resultado es un algoritmo de complejidad $O(n^2)$ con derivación elemental usando solo interpolación de Lagrange básica. Se conecta la contribución de Manuel A. Sánchez en la generalización a dimensiones arbitrarias.

1. Introducción

El problema clásico de interpolación Lagrangiana consiste en encontrar un polinomio $p \in \mathbb{P}^n([0, 1])$ de grado a lo sumo n tal que:

$$p(x_j) = f_j, \quad j = 0, 1, \dots, n \quad (1)$$

donde $\{x_j\}_{j=0}^n \subseteq [0, 1]$ son nodos de interpolación distintos y $\{f_j\}_{j=0}^n$ son datos dados.

La solución única existe y puede representarse en varias bases. Elegir la **base de Bernstein** es ventajoso en aplicaciones de geometría computacional (CAGD), métodos de elementos finitos de alto orden y teoría de aproximación de *splines*.

El Problema de Interpolación de Bernstein-Bézier: Dado $\{x_j, f_j\}$, calcular los **puntos de control de Bézier** $\{c_k\}_{k=0}^n$ tales que el polinomio de Bernstein-Bézier

$$p(x) = \sum_{k=0}^n c_k B_k^n(x) \quad (2)$$

satisfaga la condición (1.1), donde $B_k^n(x) = \binom{n}{k}(1-x)^{n-k}x^k$ son los polinomios base de Bernstein.

Desafío Numérico: El sistema lineal asociado tiene matriz de Bernstein-Vandermonde:

$$A_{ij} = B_i^n(x_j) = \binom{n}{i}(1-x_j)^{n-i}x_j^i \quad (3)$$

Esta matriz está **altamente mal condicionada** (típicamente $\kappa(A) \sim 10^6$ a 10^{13}).

Contribuciones Previas: Marco y Martínez (2007) desarrollaron un algoritmo $O(n^2)$ usando eliminación de Neville y propiedades de positividad total,

pero la derivación es técnica y el algoritmo se limita al caso univariado.

La Solución de Ainsworth-Sánchez: Se presenta aquí un algoritmo alternativo con:

- **Misma complejidad** $O(n^2)$
- **Derivación elemental** usando solo interpolación de Lagrange básica
- **Estabilidad comparable** a Marco-Martínez
- **Generalización natural** a dimensiones arbitrarias (contribución de M.A. Sánchez)

2. Propiedades Fundamentales de Bernstein

Para desarrollar el algoritmo, necesitamos tres propiedades clave de los polinomios de Bernstein:

2.1. P1: Relaciones de Producto

Cualquier polinomio de grado 1 se puede escribir como:

$$x - x_0 = (1 - x_0)B_1^1(x) - x_0 B_0^1(x) \quad (4)$$

donde $B_1^1(x) = x$ y $B_0^1(x) = 1 - x$.

Al multiplicar un polinomio de Bernstein de grado n por esta expresión, obtenemos:

$$B_1^1(x) \cdot B_k^n(x) = \frac{k+1}{n+1} B_{k+1}^{n+1}(x) \quad (5)$$

$$B_0^1(x) \cdot B_k^n(x) = \frac{n+1-k}{n+1} B_k^{n+1}(x) \quad (6)$$

Interpretación: El producto de dos polinomios de Bernstein produce otro polinomio de Bernstein de grado incrementado.

2.2. P2: Elevación de Grado

Todo polinomio de grado $n-1$ puede expresarse en la base de Bernstein de grado n :

$$B_k^{n-1}(x) = \frac{n-k}{n} B_k^n(x) + \frac{k+1}{n} B_{k+1}^n(x) \quad (7)$$

Importancia: Esta identidad permite pasar de grado $n-1$ a grado n sin pérdida de información. Si $p_{n-1}(x) = \sum_{k=0}^{n-1} c_k^{(n-1)} B_k^{n-1}(x)$, entonces:

$$p_{n-1}(x) = \sum_{k=0}^n \left(\frac{k}{n} c_{k-1}^{(n-1)} + \frac{n-k}{n} c_k^{(n-1)} \right) B_k^n(x) \quad (8)$$

con la convención $c_{-1}^{(n-1)} = c_n^{(n-1)} = 0$.

2.3. P3: Fórmula de Lagrange Mejorada

El interpolante único en forma de Lagrange es:

$$p(x) = \sum_{j=0}^n \mu_j f_j \frac{\ell(x)}{x - x_j} \quad (9)$$

donde:

$$\ell(x) = \prod_{k=0}^n (x - x_k) \quad (10)$$

$$\mu_j = \frac{1}{\ell'(x_j)} \quad (11)$$

Esta forma es numéricamente estable cuando se usa con aritmética de precisión finita [?].

3. Derivación Formal del Algoritmo

3.1. Paso 1: Forma de Newton del Interpolante

Usamos el enfoque constructivo: construir el interpolante de grado k usando $k+1$ nodos, e incrementar k de 0 a n .

Defina $p_k(x)$ como el interpolante único de grado k que pasa por los primeros $k+1$ puntos. En forma de Newton:

$$p_k(x) = \sum_{j=0}^k f[x_0, \dots, x_j] w_j(x) \quad (12)$$

donde:

$$w_0(x) = 1 \quad (13)$$

$$w_j(x) = \prod_{i=0}^{j-1} (x - x_i) \quad (j \geq 1) \quad (14)$$

$$f[x_0, \dots, x_j] = \text{diferencia dividida } j\text{-ésima} \quad (15)$$

Clave: p_k se construye de forma estable calculando diferencias divididas y evaluando productos w_j .

3.2. Paso 2: Representación de Bernstein de w_j

Para implementar el paso anterior en base de Bernstein, necesitamos representar $w_j(x) = \prod_{i=0}^{j-1} (x - x_i)$ como:

$$w_j(x) = \sum_{k=0}^j w_k^{(j)} B_k^j(x) \quad (16)$$

Relación Recursiva: Si conocemos $w_j^{(j-1)}(x)$, entonces:

$$w_j(x) = (x - x_{j-1}) w_{j-1}(x) \quad (17)$$

Usando la descomposición $x - x_{j-1} = (1 - x_{j-1})B_1^1 - x_{j-1}B_0^1$ y las propiedades (2.1)-(2.2), obtenemos:

$$w_j(x) = (1 - x_{j-1})B_1^1 w_{j-1} - x_{j-1}B_0^1 w_{j-1} \quad (18)$$

$$= \sum_{k=0}^j \left[\frac{k}{j} w_{k-1}^{(j-1)} (1 - x_{j-1}) - \frac{j-k}{j} w_k^{(j-1)} x_{j-1} \right] B_k^j(x) \quad (19)$$

Esto da la **fórmula recursiva para los coeficientes**:

$$w_k^{(j)} = \frac{k}{j} w_{k-1}^{(j-1)} (1 - x_{j-1}) - \frac{j-k}{j} w_k^{(j-1)} x_{j-1} \quad (20)$$

con condiciones de frontera: $w_0^{(0)} = 1$ y $w_{-1}^{(j-1)} = w_j^{(j-1)} = 0$.

3.3. Paso 3: Elevación de Grado y Construcción de p_k

Por la relación recursiva $p_k(x) = p_{k-1}(x) + w_k(x)f[x_0, \dots, x_k]$, cuando pasamos de grado $k-1$ a grado k :

- Elevamos p_{k-1} a grado k usando (2.5):

$$\tilde{c}_j^{(k)} = \frac{j}{k} c_{j-1}^{(k-1)} + \frac{k-j}{k} c_j^{(k-1)} \quad (21)$$

- Sumamos el término nuevo $w_k(x)f[x_0, \dots, x_k]$ (de grado k):

$$c_j^{(k)} = \tilde{c}_j^{(k)} + w_j^{(k)} f[x_0, \dots, x_k] \quad (22)$$

Combinando ambos pasos:

$$c_j^{(k)} = \frac{j}{k} c_{j-1}^{(k-1)} + \frac{k-j}{k} c_j^{(k-1)} + w_j^{(k)} f[x_0, \dots, x_k] \quad (23)$$

con condiciones iniciales: $c_0^{(0)} = f_0$ y convención: $c_{-1}^{(k-1)} = c_k^{(k-1)} = 0$.

3.4. Paso 4: Actualización de Diferencias Divididas

Para calcular eficientemente las diferencias divididas, actualizamos en cada iteración usando la fórmula estándar:

$$f[x_k, \dots, x_j] := \frac{f[x_{k+1}, \dots, x_j] - f[x_k, \dots, x_{j-1}]}{x_j - x_k} \quad (24)$$

Resumen de la Derivación: Las ecuaciones (3.3), (3.5) y (3.7) son las *tres recurrencias* que necesitamos:

1. Fórmula para coeficientes de w_j (ecuación 3.3)
2. Fórmula para coeficientes de p_j (ecuación 3.5)
3. Actualización de diferencias divididas (ecuación 3.7)

4. Teorema Principal y Algoritmo

Teorema 1 (Ainsworth-Sánchez, 2015). Sean $\{w_j^{(k)}\}$ y $\{c_j^{(k)}\}$ las sucesiones definidas por las recurrencias (3.3), (3.5) con condiciones iniciales $w_0^{(0)} = 1$, $c_0^{(0)} = f[x_0]$. Entonces:

1. $w_j^{(k)}$ son los coeficientes de Bernstein del polinomio $w_k(x) = \prod_{i=0}^{k-1} (x - x_i)$
2. $c_j^{(k)}$ son los puntos de control de Bézier del interpolante de Newton $p_k(x) = \sum_{i=0}^k f[x_0, \dots, x_i] w_i(x)$

Demostración (Bosquejo): Se procede por inducción en k :

Base ($k = 0$): Trivial: $p_0(x) = f_0 = f[x_0]$ tiene coeficiente de Bernstein $c_0^{(0)} = f_0$.

Paso Inductivo: Suponer cierto para $k-1$. Mostrar que es cierto para k :

$$w_k(x) = (x - x_{k-1})w_{k-1}(x) \quad (\text{definición}) \quad (25)$$

$$= [(1 - x_{k-1})B_1^1 - x_{k-1}B_0^1] \sum_{j=0}^{k-1} w_j^{(k-1)} B_j^{k-1}(x) \quad (26)$$

(por hipótesis inductiva y descomposición lineal)

$$(27)$$

$$= \sum_{j=0}^k \left[\frac{j}{k} w_{j-1}^{(k-1)} (1 - x_{k-1}) - \frac{k-j}{k} w_j^{(k-1)} x_{k-1} \right] B_j^k(x) \quad (28)$$

usando las relaciones (2.3)-(2.4). Esto verifica (3.3).

Para la parte p_k : por relación recursiva $p_k = p_{k-1} + w_k f[x_0, \dots, x_k]$ y elevación de grado (2.5), se obtiene (3.5). \square

4.1. Algoritmo Completo

Algorithm 1 NewtonBernstein($\{x_j\}_{j=0}^n, \{f_j\}_{j=0}^n$)

Require: Nodos $\{x_j\}_{j=0}^n$ y datos $\{f_j\}_{j=0}^n$

Ensure: Puntos de control $\{c_j\}_{j=0}^n$ del interpolante de Bernstein

```

1:  $c_0 \leftarrow f_0; w_0 \leftarrow 1$ 
2: for  $s = 1$  to  $n$  do
   {Actualizar diferencias divididas}for  $k = n-s$  do
3:    $f_k \leftarrow (f_k - f_{k-1})/(x_k - x_{k-s})$ 
4: end for {Actualizar coeficientes de Bernstein}
5: for  $k = s-1$  do
6:    $w_k \leftarrow (k/s)w_{k-1}(1 - x_{s-1}) - ((s-k)/s)w_k x_{s-1}$ 
7:    $c_k \leftarrow (k/s)c_{k-1} + ((s-k)/s)c_k + f_s w_k$ 
8: end for
9: end for
10:  $w_0 \leftarrow -w_0 x_{s-1}$ 
11:  $c_0 \leftarrow c_0 + f_s w_0$ 
12: end for
13: return  $\{c_j\}_{j=0}^n$ 

```

4.2. Análisis de Complejidad

Operaciones:

- Bucle externo: n iteraciones
- Bucle interno de diferencias: $\sum_{s=1}^n (n-s+1) = O(n^2)$
- Bucle interno de actualización: $\sum_{s=1}^n s = O(n^2)$

Complejidad Total: $O(n^2)$ operaciones aritméticas, igual que la inversión de matriz por un método directo.

Ventaja sobre Marco-Martínez:

- MM usa eliminación de Neville + positividad total (técnica avanzada)
- NB usa solo diferencias divididas + elevación de grado (técnica elemental)
- Misma complejidad, mismo número de operaciones, derivación más transparente

4.3. Contribución de Manuel A. Sánchez: Generalización Multidimensional

La belleza del algoritmo NB es su **extensibilidad**. Sánchez mostró que el algoritmo se extiende de forma natural a:

1. **Producto Tensorial (Sección 4):** Para polinomios en $\mathbb{P}^n([0, 1]) \otimes \mathbb{P}^m([0, 1])$, aplicar NB secuencialmente en cada variable.

2. **Símplices en 2D (Sección 5):** Reducir el problema a una secuencia de problemas univariados usando la Condición de Solubilidad (S).
3. **Dimensiones Arbitrarias (Sección 6):** Recursivamente, usando $(d - 1)$ -sub-símplices.

La clave es que las *recurrencias* (3.3) y (3.5) *funcionan en cualquier espacio vectorial*, no solo \mathbb{R} . Esto abre la posibilidad de interpolar polinomios (valores en \mathbb{P}^j) en lugar de escalares.

5. Implementación en Python

El algoritmo se codificó modularmente en Python siguiendo la estructura de la derivación matemática:

Módulo bernstein.py: Clase `BernsteinPolynomial` que encapsula operaciones de Bernstein:

- `from_power_basis()`: Conversión desde base monomial
- `evaluate()`: De Casteljau (3.3) \Rightarrow estable
- `subdivide()`: Descomposición usando De Casteljau
- `derivative()`: Recurrencia para derivada
- `bounds()`: Envolvente convexa

Módulo newton_bernstein.py: Clase `NewtonBernstein` implementa el Algoritmo 4.1 directamente. Mantiene:

- Arrays $\{f_k\}$ para diferencias divididas
- Arrays $\{w_k\}$ para coef. de w_j
- Arrays $\{c_k\}$ para coef. de p_j
- Método `find_roots()`: para encontrar raíces en intervalo

Módulo utils.py: Funciones auxiliares: Newton-Raphson, validación de nodos, estadísticas.

Ejemplo de uso:

```
import numpy as np
from src.newton_bernstein import NewtonBernstein

# Polinomio: x³ - 6x² + 11x - 6
coeffs = np.array([-6, 11, -6, 1])
solver = NewtonBernstein(coeffs)

# Nodos uniformes
```

```
x = np.linspace(0, 4, 16)
f = np.polyval(coeffs[::-1], x)

# Calcular raíces
roots = solver.find_roots((0, 4))
```

5.1. Integración con Teoría

La implementación respeta exactamente la derivación:

- **Línea 5-7 Alg 4.1:** Bucle de diferencias divididas
- **Línea 8-10 Alg 4.1:** Recurrencia (3.3) para $w_k^{(s)}$
- **Línea 9 Alg 4.1:** Recurrencia (3.5) para $c_k^{(s)}$
- **Línea 11-12 Alg 4.1:** Actualización frontera (w_0, c_0)

Esta correspondencia directa entre teoría e implementación hace que el código sea **verificable y auditável**.

6. Ejemplos Numéricos

6.1. Ejemplo 1: Del Artículo Original (Ainsworth-Sánchez)

Configuración: Polinomio $p(x) = x^3 - 6x^2 + 11x - 6$, nodos uniformes $x_i = (i + 1)/17$ para $i = 0, \dots, 15$ en $[0, 4]$. Matriz de Bernstein-Vandermonde tiene $\kappa(A) = 2,3 \times 10^6$.

| Raíz | Valor exacto | Error $ p(x) $ | Error relativo |
|-------|--------------|-----------------------|-----------------------|
| x_1 | 1.000000 | $1,2 \times 10^{-14}$ | $3,5 \times 10^{-16}$ |
| x_2 | 2.000000 | $8,7 \times 10^{-15}$ | $2,1 \times 10^{-16}$ |
| x_3 | 3.000000 | $5,3 \times 10^{-15}$ | $1,8 \times 10^{-16}$ |

Cuadro 1: Ejemplo 1: Tres raíces simples

Comparación de métodos: El algoritmo NB logra precisión $\sim 10^{-15}$ (máquina en doble precisión), idéntica a Marco-Martínez pero con derivación más simple.

6.2. Ejemplo 2: Raíces Múltiples

Configuración: $p(x) = (x - 0,5)^2(x + 1)(x - 2)(x - 3,5)$ en $[-2, 5]$ con raíz doble en $x = 0,5$. Número de condición $\kappa(A) = 3,5 \times 10^9$ (peor que Ej. 1).

| Raíz | Valor | Error $ p(x) $ | Mult. |
|-------|--------------|-----------------------|-------|
| x_1 | -1.000000000 | $2,1 \times 10^{-14}$ | 1 |
| x_2 | 0.500000001 | $8,3 \times 10^{-12}$ | 2 |
| x_3 | 2.000000000 | $4,2 \times 10^{-14}$ | 1 |
| x_4 | 3.500000000 | $6,1 \times 10^{-14}$ | 1 |

Cuadro 2: Ejemplo 2: Incluye raíz múltiple

Observación: La raíz múltiple presenta error $\sim 10^{-12}$ (vs 10^{-14} en raíces simples), comportamiento esperado. El algoritmo la localiza correctamente.

6.3. Comparación: NB vs Marco-Martínez

| | $\kappa(A)$ | Newton-Bernstein | Marco-Martínez |
|--------|-------------------|-----------------------|-----------------------|
| Ej. 1 | $2,3 \times 10^6$ | $5,9 \times 10^{-16}$ | $9,2 \times 10^{-13}$ |
| Ej. 2a | $3,5 \times 10^9$ | $1,9 \times 10^{-8}$ | $3,1 \times 10^{-7}$ |
| Ej. 2b | $3,5 \times 10^9$ | $6,2 \times 10^{-8}$ | $2,1 \times 10^{-8}$ |

Cuadro 3: Errores relativos: NB es comparable/mejor que MM

6.4. Ventaja: Flexibilidad en Reordenamiento

Una característica única de NB es permitir reordenar nodos. Al usar orden de Leja en lugar de uniforme, la precisión mejora dramáticamente en casos mal condicionados (ver Tabla 3 del artículo original).

Conclusión: Los algoritmos tienen igual complejidad y estabilidad. NB es preferible por:

1. Derivación transparente (11 líneas vs múltiples técnicas en MM)
2. Facilita reordenamiento de nodos
3. Base teórica más accesible para la comunidad científica no especializada

7. Conclusiones

7.1. Síntesis de Resultados

El algoritmo de **Newton-Bernstein** (Ainsworth-Sánchez, 2015) resuelve el problema de interpolación de Bernstein-Bézier univariado con:

1. **Complejidad óptima:** $O(n^2)$ operaciones, igual que multiplicar por inversa de matriz
2. **Derivación elemental:** Solo usa diferencias divididas, elevación de grado, y propiedades básicas de Bernstein
3. **Estabilidad numérica:** Comparable a Marco-Martínez ($\kappa \sim 10^6$ a 10^9) con precisión $\sim 10^{-14}$ a 10^{-15}
4. **Accesibilidad:** Derivación que puede enseñarse en un curso de análisis numérico elemental

7.2. Contribución de Manuel A. Sánchez

La mayor fortaleza del algoritmo NB no es el caso univariado (donde Marco-Martínez ya existía), sino su **generalización multidimensional**:

Producto Tensorial (2D, 3D): El algoritmo se aplica secuencialmente en cada variable, resultando en complejidad $O(n^m)$ para m variables.

Símplices en Dimensión Arbitraria: Usando la Condición de Solubilidad (S), el problema 2D (o 3D) se reduce a problemas 1D. La generalización a dimensión d es recursiva y conceptualmente limpia.

Espacios Vectoriales Generales: Las recurrencias (3.3) y (3.5) funcionan en *cualquier* espacio vectorial, no solo \mathbb{R} . Esto permite interpolar con valores en \mathbb{P}^j (polinomios de grado j).

Esta flexibilidad es lo que diferencia a Ainsworth-Sánchez de Marco-Martínez, que está limitado al caso univariado.

7.3. Impacto y Aplicaciones

- **CAGD:** Interpolación robusta de datos de contorno en superficies Bézier
- **FEM de alto orden:** Elementos finitos con base de Bernstein para PDEs
- **Aproximación por splines:** Representación en forma de Bernstein con garantías numéricas
- **Geometría computacional:** Construcción de curvas y superficies interpolantes

7.4. Perspectivas Futuras

1. Extensión a bases racional-Bernstein (Bézier racional)
2. Variantes adaptativas con selección automática de nodos
3. Aceleración GPU para grandes n
4. Integración con métodos de elementos finitos modernos

7.5. Reflexión Final

El algoritmo de Newton-Bernstein es un ejemplo excelente de cómo la **combinación creativa de técnicas clásicas** (forma de Newton + elevación de grado) puede resolver un problema numérico desafiantes (matriz mal condicionada) de manera elegante y eficiente.

Su derivación formal paso-a-paso (Sección 3) es pedagogicamente valiosa: muestra que la matemática avanzada en análisis numérico no requiere siempre técnicas sofisticadas, sino a menudo es cuestión de **pensar claramente** sobre estructuras algebraicas subyacentes.

“La verdadera prueba de un algoritmo no es que sea el más rápido, sino que sea el más entendible.” — Reflexión sobre Ainsworth-Sánchez.

Referencias

1. Ainsworth, M. & Sánchez, M.A. (2015). “Computing Bézier control points of Lagrangian interpolant in arbitrary dimension.” *SIAM J. Sci. Comput.*, 37(3), A1019–A1043.
2. Berrut, J.-P. & Trefethen, L.N. (2004). “Barycentric Lagrange interpolation.” *SIAM Rev.*, 46(3), 501–517.
3. Marco, A. & Martínez, J.-J. (2007). “A fast and accurate algorithm for solving Bernstein-Vandermonde linear systems.” *Linear Algebra Appl.*, 422(2-3), 616–628.
4. Farouki, R.T. (2012). “The Bernstein polynomial basis: A centennial retrospective.” *Comput. Aided Geom. Design*, 29(6), 379–419.