BPOLY: An object–oriented library of algorithms for polynomials in Bernstein form

Yi–Feng Tsai and Rida T. Farouki
Department of Mechanical and Aeronautical Engineering,
University of California, Davis, CA 95616.

Abstract

The design, implementation, and testing of a C++ software library for univariate polynomials in Bernstein form is described. By invoking the class environment and operator overloading, each polynomial in an expression is interpreted as an object compatible with the arithmetic operations and other common functions (subdvision, degree elevation, differentiation and integration, composition, greatest common divisor, real–root solving, etc.) for polynomials in Bernstein form. The library allows compact and intuitive implementation of lengthy manipulations of Bernstein–form polynomials, which often arise in computer graphics and computer–aided design and manufacturing applications. A series of empirical tests indicate that the library functions are typically very accurate and reliable, even for polynomials of surprisingly high degree.

Keywords: Bernstein basis, object–oriented software, polynomial algorithms, root–finding, numerical stability.

e–mail addresses: ytsai@ucdavis.edu, farouki@ucdavis.edu

1 Introduction

The Bernstein–Bézier curve and surface representations have become *de facto* standards in computer–aided geometric design (CAGD), due to the intuitive geometrical properties and recursive algorithms they entail [5, 18] — convex hull confinement, variation–diminishing property, de Casteljau subdivision algorithm, etc. The Bernstein form of a degree–n polynomial

$$P(x) \,=\, \sum_{k=0}^{n} C_k^n\, b_k^n(x)\,, \quad b_k^n(x) \,=\, \binom{n}{k}(1-x)^{n-k}x^k \tag{1}$$

on[1] $x \in [\,0,1\,]$ was introduced in 1912 by S. N. Bernstein [2] in a constructive proof for the *theorem of Weierstrass*, concerning uniform approximation of continuous functions on finite intervals by polynomials. The slow convergence of Bernstein polynomial approximants incurred a widespread perception that the form (1) is unsuitable for "practical" computations, which persisted until de Casteljau and Bézier demonstrated its advantages in geometric design.

Another advantage of the Bernstein form, of much broader significance, is its intrinsic numerical stability as a representation for polynomials over finite intervals [11]. Namely, the values (and roots) of $P(x)$ on $x \in [\,0,1\,]$ are much less sensitive to perturbations or errors in the coefficients $C_0^n,\ldots,C_n^n$ compared to other commonly used polynomial bases — e.g., the power form. Indeed, the Bernstein form is *optimally stable* [9]: it is impossible to construct a basis on $[\,0,1\,]$ that yields systematically smaller condition numbers for the values and roots of arbitrary polynomials on this interval.

Explicit conversions between different polynomial bases typically become increasingly ill–conditioned [3, 6, 8, 17] as the polynomial degree n increases. To take full advantage of its stability, it is essential to formulate the problem under consideration using the Bernstein form, and to express all intermediate steps using this form. Although perhaps less familiar, the basic algorithms for Bernstein–form polynomials (arithmetic operations, evaluation, subdivision, composition, etc.) are not significantly more complicated or expensive than their familiar power–form versions [12]. In this paper we describe an object–oriented library of functions for Bernstein–form polynomials that facilitates intuitive high–level programming of polynomial computations. Relieved from having to worry about the underlying technical details (binomial coefficient

[1] For brevity we shall focus on the Bernstein form over the unit interval $x \in [\,0,1\,]$ here; the extension to an arbitrary interval $[\,a,b\,]$ is straightforward.

operations, matching degrees of operands, etc.) the programmer can rapidly develop robust and efficient code that inherits both the intuitive geometrical features and stability properties of Bernstein–form polynomials.

A Bernstein–form polynomial is specified by its degree n and coefficients $C_0^n, \ldots, C_n^n$. A function that performs a binary operation on two polynomials might typically have the form (in a syntax similar to C or C++):

```
operation(int n1, double C1[n1+1], int n2, double C2[n2+1])
{
    ...
    return n_result, C_result[n_result+1];
}
```

where two sets of Bernstein data are received as input, and one set is returned as output.[2] Natural as it seems, this approach becomes unduly cumbersome in programming polynomial expressions that require just a single line when written by hand. Consider, for example, the formula

$$g(x) \,=\, 3\,[\,u(x)v'(x) \,-\, u'(x)v(x)\,] \,+\, [\,u^2(x) \,-\, v^2(x)\,]\,, \tag{2}$$

where $u(x)$ and $v(x)$ are Bernstein–form polynomials. To obtain the desired polynomial one might invoke a sequence of function calls, such as:

```
v_prime = differentiate(v);
u_v_prime = multiply(u,v_prime);
u_prime = differentiate(u);
u_prime_v = multiply(u_prime,v);
term1 = subtract(u_v_prime,u_prime_v);
3_term1 = premultiply(3.0,term1);
u_square = multiply(u,u);
v_square = multiply(v,v);
term2 = subtract(u_square,v_square);
g = add(3_term1,term2);
```

Apart from its verbosity, this approach incurs a proliferation of temporary variables. Expression (2) is an innocuous example of the kinds of Bernstein–form polynomials that are commonplace in geometric modelling applications

[2]Of course the input and output data sets may differ in number from the example shown here, and may also include scalar values.

— see [13] for numerous examples that arise in the context of determining maximum safe feedrates for a CNC machine under given constraints on the torque and power capacity of the drive motors.

This paper describes a software library for Bernstein–form polynomials that exploits the object–oriented programming capabilities of C++. By the *operator overloading* method, each polynomial in an expression is regarded as an "object" that comprises degree and coefficient information. Arithmetic operations and other basic functions on these polynomials are then defined, receiving "instances" of the objects as input, and returning new instances as outputs. Programming a sequence of polynomial operations such as (2) thus reduces to simply "writing out" the expression in the appropriate syntax:

```
g = 3.0 * (u * diff(v) - diff(u) * v) + (u ^ 2 - v ^ 2);
```

Since no separate function calls (or intermediate variables) are needed, there is a significant reduction in the implementation effort, and also a diminished likelihood of programming mistakes. We expect this software library to prove useful in the complex geometrical calculations often encountered in CAGD (e.g., surface intersections) and many other application contexts.

2 Bernstein data & algorithms as objects

The software library comprises a header file Bernstein.h for declaration of the functions, and a C++ program Bernstein.cpp containing all of the definitions, basic functions, and operator–overloadings. It employs the class environment in C++ to handle both Bernstein *data* and *algorithms*.

The header file Bernstein.h is listed below, showing the structure of the object definition. One can see that the object Bernstein possesses two pieces of data, cf and dgr, and a group of basic functions and algorithms. Specific definitions for each of these functions are explained in subsequent sections.

```
// Declare a class for Bernstein data and algorithms:
class Bernstein
{
    public:
        // Array for storing Bernstein coefficients.
        double *cf;
        // Degree of the polynomial.
```

```
int dgr;

// Constructors and destructors:
Bernstein();
Bernstein(double *coeff, int degree);
Bernstein(const Bernstein &u);
~Bernstein();

// Binomial coefficients and utilities:
friend Bernstein getprimes(int n, int pr_num, int mode);
friend Bernstein factorization(int n, int k);
friend double binom(int n, int k);
friend double binom(const Bernstein &e);
friend Bernstein binomult(const Bernstein &e1, const Bernstein &e2);
friend Bernstein binodiv(const Bernstein &e1, const Bernstein &e2);

// Degree elevation:
friend Bernstein DE(const Bernstein &u, int r);

// Evaluation and subdivision:
friend double EVAL(const Bernstein &u, double x);
friend Bernstein subLEFT(const Bernstein &u, double x);
friend Bernstein subRIGHT(const Bernstein &u, double x);

// Differentiation and integration:
friend Bernstein diff(const Bernstein &u);
friend Bernstein integrate(const Bernstein &u);
friend double integral(const Bernstein &u);

// Normalization:
friend double Norm(const Bernstein &u);
friend Bernstein Normalize(const Bernstein &u);

// Operator overloading:
Bernstein operator=(const Bernstein &u);
Bernstein operator+(const Bernstein &u);
Bernstein operator-(const Bernstein &u);
Bernstein operator*(const Bernstein &u);
friend Bernstein operator*(const Bernstein &u, double factor);
friend Bernstein operator*(double factor, const Bernstein &u);
Bernstein operator^(int power);
Bernstein operator/(const Bernstein &u);
friend Bernstein quo(const Bernstein &u, int degree);
friend Bernstein rem(const Bernstein &u, int degree);
Bernstein operator<<(const Bernstein &u);
```

```
        // Greatest common divisor:
        friend Bernstein GCD(const Bernstein &u, const Bernstein &v, double epsilon);

        // Bernstein root solver:
        friend Bernstein ROOT(const Bernstein &u, double delta, double eta, double epsilon);
        friend Bernstein sROOT(const Bernstein &u, double delta, double eta);
        friend Bernstein mROOT(const Bernstein &u, double delta, double eta, double
            epsilon);
};
```

3 Basic functions

3.1 Binomial coefficient computations

Related programs:
Bernstein getprimes(int n, int pr_num, int mode)
Bernstein factorization(int n, int k)
double binom(int n, int k)
double binom(const Bernstein &e)
Bernstein binomult(const Bernstein &e1, const Bernstein &e2)
Bernstein binodiv(const Bernstein &e1, const Bernstein &e2)

In processing polynomials expressed in Bernstein form, we often encounter computations involving the binomial coefficients

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}\,, \quad k = 0, \ldots, n\,. \tag{3}$$

We adopt the convention that (3) has the value 0 when $k < 0$ or $k > n$. Pascal's triangle is a well–known scheme for computing binomial coefficients:

$$\binom{m}{k} = \binom{m-1}{k} + \binom{m-1}{k-1} \tag{4}$$

for $k = 0, \ldots, m$ and $m = 2, \ldots, n$. In ordinary integer arithmetic, however, this may incur overflow at relatively small n values [12]. Goetgheluck [14, 15] describes an approach to processing binomial coefficients that circumvents this problem, based on their *factorization into primes*

$$\binom{n}{k} = \prod_{r=1}^{m} p_r^{e_r} , \tag{5}$$

$p_1, p_2, p_3, \ldots = 2, 3, 5, \ldots$ being the ordered set of primes, and $e_1, e_2, e_3, \ldots$ the corresponding exponents. Since prime factors larger than n cannot appear in the factorization, m is determined by $p_m \leq n$ and $p_{m+1} > n$.

Employing (5) as our basic representation, the multiplication and division of binomial coefficients can be performed by simply adding or subtracting the exponent vectors. Additions and subtractions also benefit from use of (5), by extracting the *highest common factor* from the operands before the addition or subtraction is performed. Although the value of an expression that involves binomial coefficients may ultimately be required as a floating–point number, performing as much of the calculation as possible on the exponents in the representation (5) minimizes the accumulation of round–off errors.

To obtain the prime factorization of a binomial coefficient, we will need the following lemma:

Lemma 1 *The power of a prime p in the factorization of $n!$ is given by*

$$\sum_{i>0\,,\,p^i \leq n} \left\lfloor \frac{n}{p^i} \right\rfloor . \tag{6}$$

To verify this, note that $m_1 = \lfloor n/p \rfloor$ is the largest integer such that $m_1 p \leq n$. Now in the expansion of $n!$, p occurs in $p, 2p, \ldots, m_1 p$, and hence m_1 is the number of "simple" appearances of p in the factors of $n!$. Similarly, setting $m_2 = \lfloor n/p^2 \rfloor$, we see that p^2 occurs in $p^2, 2p^2, \ldots, m_2 p^2$ upon expanding $n!$, and m_2 is thus the number of "double" appearances of p in the factors of $n!$. Continuing in the same manner for each successive power of p, such that $p^i \leq n$, we finally obtain (6) as the power of p in $n!$.

Applying this result to (3), and noting that multiplications and divisions correspond to additions and subtractions of prime exponents, we obtain

$$e_p(n,k) = \sum_{i>0\,,\,p^i \leq n} \left\lfloor \frac{n}{p^i} \right\rfloor - \left\lfloor \frac{k}{p^i} \right\rfloor - \left\lfloor \frac{n-k}{p^i} \right\rfloor \tag{7}$$

for the exponent of the prime p in (3). We now show that each term of the above sum is simply 0 or 1.

Theorem 1 *Let p be a prime number, and n, k, i be positive integers that satisfy $n \geq k$ and $n, k \geq p^i$. Then n and k can be expressed as*

$$n = a\,p^i + b\,, \qquad k = c\,p^i + d\,,$$

where a, b, c, d are positive integers with $0 \leq b, d < p^i$, and we have

$$\left\lfloor \frac{n}{p^i} \right\rfloor - \left\lfloor \frac{k}{p^i} \right\rfloor - \left\lfloor \frac{n-k}{p^i} \right\rfloor = \begin{cases} 0 & \text{if } b \geq d\,, \\ 1 & \text{if } b < d\,. \end{cases} \tag{8}$$

Proof : Clearly, n/p^i and k/p^i comprise integer and fractional parts,

$$\frac{n}{p^i} = a + \frac{b}{p^i}\,, \qquad \frac{k}{p^i} = c + \frac{d}{p^i}\,,$$

so that $\lfloor n/p^i \rfloor - \lfloor k/p^i \rfloor = a - c$. Thus, recalling that $0 \leq b, d < p^i$, we write

$$\frac{n-k}{p^i} = a - c + \frac{b-d}{p^i}\,. \tag{9}$$

Clearly, if $b \geq d$, the integer part of (9) is just $a - c$, which causes expression (8) to assume the value 0. When $b < d$, on the other hand, we must borrow 1 from $a - c$ to accommodate the negative fractional part on the right in (9), so that $\lfloor (n-k)/p^i \rfloor = a - c - 1$, which causes (8) to assume the value 1. ■

The relation (8) suffices to compute the exponent (7) of each prime $p \leq n$, and hence yields the complete factorization of (3). The following lists a simple implementation of the scheme (Goetgheluck gives a slightly more complicated version, employing several "rules of thumb" that help reduce the total number of floating point divisions — see [14] for details).

```
Bernstein factorization(int n, int k)
{
    int i, j;
    double p_i;
    Bernstein primes, exp;

    // Irrational case:
    if ((n < k) || (k < 0))    exp.dgr = -1;
    else
    {
        // Retrieve prime numbers in mode 1:
        primes = getprimes(n, 0, 1);
```

```
        // Set up the array for storing the exponent vector:
        exp.dgr = primes.dgr;
        delete [] exp.cf;
        exp.cf = new double[exp.dgr];
        if (!exp.cf)   {   cout << "Allocation error.\n";   exit(1);   }

        exp.cf[0] = 1.0;
        for (j=1; j <= exp.dgr; j++)
        {
            exp.cf[j] = 0.0;
            i = 1;
            p_i = pow(primes.cf[j], i);
            while (p_i <= n)
            {
                if (fmod(n, p_i) >= fmod(k, p_i))   exp.cf[j] = exp.cf[j];
                else if (fmod(n, p_i) < fmod(k, p_i))   exp.cf[j] = exp.cf[j] + 1.0;
                i = i + 1;
                p_i = pow(primes.cf[j], i);
            }
        }
    }
    // Return the exponent array and also the size of the array in
    // an instance of Bernstein object:
    return exp;
}
```

For input parameters n and k, this function returns the exponents $e_p(n,k)$ of all prime factors $p \leq n$ of (3). Note that class Bernstein is "borrowed" here to store the exponents in the cf array, and the size of the array in the variable dgr. This ensures that the return type of this function is consistent with all the other functions. Input data with $k < 0$ or $k > n$ causes the value -1 to be returned for the size of the exponent array, which alerts subsequent functions to produce the value 0 for (3).

The output of factorization serves as input to functions that evaluate, or perform arithmetic operations on, binomial coefficients. We emphasize that the prime factorization is an *exact* representation of (3), regardless of the size of n. Other functions concerned with the binomial coefficient are:

- Bernstein getprimes(int n, int pr_num, int mode):

This retrieves an ordered set of primes. When called with mode=1, it returns all primes less than or equal to n in an array, and the number of elements in that array. When called with other values for mode, the desired number of primes (starting with 2) is specified by pr_num, and the function returns these primes in ascending order.

- Bernstein binomult(const Bernstein &e1, const Bernstein &e2):
  This function multiplies two binomial coefficients by adding their prime exponent arrays e1 and e2; the resulting exponent array is returned in a Bernstein instance.

- Bernstein binodiv(const Bernstein &e1, const Bernstein &e2):
  Similar to the above, but with addition replaced by subtraction.

- double binom(int n, int k):
  Wiht input n and k, this function computes (3) by calling factorization() and then directly multiplying out the prime factorization.

- double binom(const Bernstein &e):
  With exponent array e as input, this function directly computes (3) by multiplying out the prime factorization.

3.2 Degree elevation

Related program:
Bernstein DE(const Bernstein &u, int r)

A polynomial of true degree n can be represented in the Bernstein basis of degree $n+r$, for $r \geq 1$, through a process known as *degree elevation* [5, 18]. Degree elevation is required in, for example, the addition of polynomials of different degrees. If $C_0^n, \ldots, C_n^n$ are the coefficients in the degree–n Bernstein basis, the coefficients in the basis of degree $n+r$ are given [12] by

$$C_k^{n+r} = \sum_{j=\max(0,k-r)}^{\min(n,k)} \frac{\binom{r}{k-j}\binom{n}{j}}{\binom{n+r}{k}} C_j^n \tag{10}$$

for $k = 0, \ldots, n+r$. The following function performs degree elevations.

```
Bernstein DE(const Bernstein &u, int r)
{
```

```
int j, jmin, jmax, n, k, i;
double fracBINOM;
Bernstein result, e1, e2, e12, e3, e123, CF;
n = u.dgr;
result.dgr = n + r;
delete [] result.cf;
result.cf = new double[result.dgr + 1];
if (!result.cf)   {   cout << "Allocation error.\n";   exit(1);   }

for (k=0; k <= n+r; k++)
{
    jmin = k - r;
    if (jmin < 0)   jmin = 0;
    jmax = k;
    if (jmax > n)   jmax = n;
    result.cf[k] = 0.0;

    // Compute the common factor:
    e1 = factorization(r, k - jmin);
    e2 = factorization(n, jmin);
    e3 = factorization(n+r, k);
    e12 = binomult(e1, e2);
    e123 = binodiv(e12, e3);
    CF = e123;

    for (j=jmin+1; j <= jmax; j++)
    {
        e1 = factorization(r, k - j);
        e2 = factorization(n, j);
        e3 = factorization(n + r, k);
        e12 = binomult(e1, e2);
        e123 = binodiv(e12, e3);

        // Establish the common factors:
        for (i=1; i <= e123.dgr; i++)
            if (e123.cf[i] != CF.cf[i])   CF.cf[i] = 0.0;
    }

    // Bring out the common factor:
    for (j=jmin; j <= jmax; j++)
    {
        e1 = factorization(r, k - j);
        e2 = factorization(n, j);
        e3 = factorization(n + r, k);
        e12 = binomult(e1, e2);
```

```
            e123 = binodiv(e12, e3);

            for (i=1; i <= e123.dgr; i++)
                if (e123.cf[i] == CF.cf[i])    e123.cf[i] = 0.0;

            // Direct evaluations:
            fracBINOM = binom(e123);
            result.cf[k] += fracBINOM * u.cf[j];
        }

        // Multiply by the common factor:
        result.cf[k] = result.cf[k] * binom(CF);
    }
    return result;
}
```

This function exploits the factorized binomial coefficient representations. For each term of the sum, the quantity involving three binomial coefficients is aggregated in factored form in the exponent array e123. Also, the highest common factor of all of these quantities is extracted from the sum as CF. By performing the necessary floating–point operations only after these steps, the effects of round–off error (especially for large n) is minimized.

3.3 Evaluation and subdivision

Related programs:
double EVAL(const Bernstein &u, double x)
Bernstein subLEFT(const Bernstein &u, double x)
Bernstein subRIGHT(const Bernstein &u, double x)

The standard means of evaluating a Bernstein–form polynomial is through an iterated sequence of linear interpolations among the Bernstein coefficients, known as the *de Casteljau algorithm* [5]. A triangular array of values $P_k^{(r)}$ for $r = 0, \ldots, n$ and $k = r, \ldots, n$ is populated as follows. We begin by assigning the Bernstein coefficients to the first row ($r = 0$):

$$P_k^{(0)} = C_k^n\,, \quad k = 0, \ldots, n\,.$$

Then, if x_s is the point at which the polynomial is to be evaluated, subsequent rows $r = 1, \ldots, n$ are filled recursively according to the formula

$$P_k^{(r)} \,=\, (1 - x_s)\, P_{k-1}^{(r-1)} \,+\, x_s\, P_k^{(r-1)}\,, \quad k = r, \ldots, n\,. \tag{11}$$

The final element, at the apex of the triangular array, then gives the desired value: $P(x_s) = P_n^{(n)}$. In addition to *evaluating* the polynomial at $x = x_s$, the de Casteljau algorithm also *subdivides* the polynomial at this point. Namely, the values on the left and right hand sides of the triangular array

$$P_0^{(0)}, P_1^{(1)}, \ldots, P_n^{(n)} \qquad \text{and} \qquad P_n^{(n)}, P_n^{(n-1)}, \ldots, P_n^{(0)}$$

are the Bernstein coefficients of the polynomial on $x \in [\,0, x_s\,]$ and $x \in [\,x_s, 1\,]$, when these sub–intervals are both mapped to $[\,0, 1\,]$. The functions subLEFT and subRIGHT furnish[3] the coefficients of the subdivided polynomial.

Subdivision is a very useful tool in processing Bézier curves and surfaces [5, 19]; it is also the basis of the root–solver described in §5 below. Note that the de Casteljau algorithm has $O(n^2)$ cost; in contexts where efficiency is of primary concern, an $O(n)$ Horner–like algorithm may be used instead.

3.4 Differentiation and integration

Related programs:
Bernstein diff(const Bernstein &u)
Bernstein integrate(const Bernstein &u)
double integral(const Bernstein &u)

The differentiation and integration of polynomials in Bernstein form is a straightforward matter, involving only linear combinations of the Bernstein coefficients. The above functions are based upon the following formulas [12]. The derivative of (1) is a polynomial of degree $n - 1$, given by

$$\frac{\mathrm{d}}{\mathrm{d}x} P(x) \,=\, \sum_{k=0}^{n-1} n(C_{k+1}^n - C_k^n)\, b_k^{n-1}(x)\,, \tag{12}$$

while the indefinite integral is a polynomial of degree $n + 1$,

$$\int P(x)\,\mathrm{d}x \,=\, \sum_{k=0}^{n+1} I_k^{n+1} b_k^{n+1}(x) \tag{13}$$

[3]Note that each call to EVAL, subLEFT, subRIGHT incurs execution of the de Casteljau algorithm — although a single execution can furnish all the information provided by these three functions, we prefer to keep them separate for greater programming clarity.

where we take $I_0^{n+1} = 0$ as the integration constant, and

$$I_k^{n+1} = \frac{1}{n+1} \sum_{j=0}^{k-1} C_j^n \,, \quad k = 1, \ldots, n+1 \,.$$

Finally, the definite integral of $P(x)$ over $[\,0,1\,]$ is

$$\int_0^1 P(x)\,\mathrm{d}x = I_{n+1}^{n+1} = \frac{1}{n+1} \sum_{k=0}^{n} C_k^n \,. \tag{14}$$

The functions diff(), integrate(), and integral() compute (12), (13), and (14).

3.5 Normalization

Related programs:
double Norm(const Bernstein &u)
Bernstein Normalize(const Bernstein &u)

In root–solving and other contexts, it may be advantageous to scale the Bernstein coefficients of a polynomial by a certain factor, in order to satisfy a desired normalization condition. If we choose the L_2 norm

$$\|P(x)\| = \left[\int_0^1 |P(x)|^2 \,\mathrm{d}x \right]^{1/2} , \tag{15}$$

as the scaling factor, this is given [12] by

$$\|P(x)\|^2 = \frac{1}{2n+1} \sum_{i=0}^{n} \sum_{j=0}^{n} \frac{\binom{n}{i}\binom{n}{j}}{\binom{2n}{i+j}} C_i^n C_j^n \,.$$

Dividing the coefficients $C_0^n, \ldots, C_n^n$ by $\|P(x)\|$ ensures that the root–mean–square value of the polynomial over $[\,0,1\,]$ is unity.

The function Norm computes the value of (15), while Normalize returns a Bernstein–form polynomial whose coefficients have been scaled by (15).

4 Arithmetic operations

Consider now the sum, difference, product, and quotient of two polynomials $F(x)$ and $G(x)$, of degree m and n, with Bernstein coefficients $A_0^m, \ldots, A_m^m$

and $B_0^n, \ldots, B_n^n$, respectively. These functions are embedded into the familiar operators $+$, $-$, $*$, $/$ by means of the operator overloading technique. The assignment operator $=$ is also overloaded; it simply copies the coefficients and degree from the right operand to the left. Two further binary operations for Bernstein–form polynomials are also presented — the *polynomial composition* and *greatest common divisor* algorithms.

4.1 Addition and subtraction

Related programs:
Bernstein Bernstein::operator+(const Bernstein &u)
Bernstein Bernstein::operator−(const Bernstein &u)

If $m = n$, the sum or difference of $F(x) \pm G(x)$ is of the same degree, and involves simply adding or subtracting the coefficients term–by–term. If $m \neq n$ we degree–elevate the polynomial of lower degree, so as to represent both polynomials in the basis of the same degree, and then add/subtract the coefficients. The addition function is illustrated below.

```
Bernstein Bernstein::operator+(const Bernstein &u)
{
    int k, m, n;
    m = dgr;    n= u.dgr;
    Bernstein result, temp;

    if (m == n)
    {
        result.dgr = m;
        delete [] result.cf;
        result.cf = new double[m + 1];
        if (!result.cf)   {   cout << "Allocation error.\n";   exit(1);   }

        for (k=0; k <= m; k++)
            result.cf[k] = cf[k] + u.cf[k];
    }
    else if (m > n)
    {
        result.dgr = m;
        delete [] result.cf;
        result.cf = new double[m + 1];
        if (!result.cf)   {   cout << "Allocation error.\n";   exit(1);   }
```

```
        temp = DE(u, m − n);
        for (k=0; k <= m; k++)
            result.cf[k] = cf[k] + temp.cf[k];
    }
    else if (m < n)
    {
        result.dgr = n;
        delete [] result.cf;
        result.cf = new double[n + 1];
        if (!result.cf)   {   cout << "Allocation error.\n";   exit(1);   }

        temp = DE(*this, n − m);
        for (k=0; k <= n; k++)
            result.cf[k] = temp.cf[k] + u.cf[k];
    }
    return result;
}
```

4.2 Multiplication

Related programs:
Bernstein Bernstein::operator∗(const Bernstein &u)
Bernstein Bernstein::operator∧(int power)
Bernstein operator∗(double factor, const Bernstein &u)
Bernstein operator∗(const Bernstein &u, double factor)

The product $F(x)G(x)$ is of degree $m+n$, and has Bernstein coefficients

$$C_k^{m+n} = \sum_{j=\max(0,k-n)}^{\min(m,k)} \frac{\binom{m}{j}\binom{n}{k-j}}{\binom{m+n}{k}} A_j^m \, B_{k-j}^n \tag{16}$$

for $k = 0, \ldots, m+n$ [12]. We omit the implementation, which is similar to that of the degree–elevation formula (10). The power operator $\wedge$ and pre– or post–muliplication (by a number) are natural concomitants of the product operator. To implement $\wedge$ (for integer exponents), we just apply the product operator repeatedly on the same operand. For pre/post–multiplication by a number, we modify the argument list of the $*$ overloading:

```
// Bernstein "pre-multiply" operator:
Bernstein operator*(double factor, const Bernstein &u)
```

```
{
    ... (omitted) ...
}

// Bernstein "post-multiply" operator:
Bernstein operator*(const Bernstein &u, double factor)
{
    ... (omitted) ...
}
```

so that it calls the appropriate definition of $*$ when the compiler recognizes a particular argument list. In C++, this is known as *function overloading.* Of course, the general product function could be used to multiply a polynomial by a scalar, if we are willing to define the latter as a polynomial of degree 0. However, it is much more convenient for the user to have the ability to write `k * polynomial` or `polynomial * k`, for any number `k`.

4.3 Division

Related programs:

```
Bernstein Bernstein::operator/(const Bernstein &u)
Bernstein quo(const Bernstein &u, int degree)
Bernstein rem(const Bernstein &u, int degree)
```

Assuming that $m \geq n$, the division of $F(x)$ by $G(x)$ corresponds to finding the *quotient* and *remainder* polynomials, $Q(x)$ and $R(x)$, in the equation

$$F(x) \;=\; G(x)Q(x) \;+\; R(x)\,, \tag{17}$$

Q being of degree $m-n$ and R of degree $\leq n-1$. Let $Q_0^{m-n},\ldots,Q_{m-n}^{m-n}$ and $R_0^{n-1},\ldots,R_{n-1}^{n-1}$ be the Bernstein coefficients of these polynomials. Applying the sum and product rules, equation (17) yields

$$\begin{aligned} A_k^m \;=& \sum_{j=\max(0,k-n)}^{\min(m-n,k)} \frac{\binom{m-n}{j}\binom{n}{k-j}}{\binom{m}{k}}\, B_{k-j}^n Q_j^{m-n} \\ &+ \sum_{j=\max(0,k-m+n-1)}^{\min(n-1,k)} \frac{\binom{m-n+1}{k-j}\binom{n-1}{j}}{\binom{m}{k}}\, R_j^{n-1} \end{aligned} \tag{18}$$

for $k=0,\ldots,m$. Since $A_0^m,\ldots,A_m^m$ and $B_0^n,\ldots,B_n^n$ are known, (18) amounts to a system of $m+1$ linear equations for the $m+1$ unknowns $Q_0^{m-n},\ldots,Q_{m-n}^{m-n}$ and $R_0^{n-1},\ldots,R_{n-1}^{n-1}$. Writing this system in vector–matrix form, a standard procedure — e.g., Gaussian elimination with partial pivoting [1] — suffices for its solution. Clearly, polynomial division incurs greater computational cost (and is perhaps more prone to round–off error accumulation) than addition or multiplication, since it entails solving a linear system. The following fragment illustrates the use of the division function:

```
Bernstein u, v, result, q, r;

result = u / v ;
q = quo(result,m-n);
r = rem(result,n-1);
```

4.4 Composition

Related programs:
Bernstein Bernstein::operator<<(const Bernstein &u)

The composition algorithm is concerned with computing the polynomial

$$W(x) \;=\; F(G(x))\,,$$

defined by substituting one polynomial as the argument of other. As before, we take $F(x)$ and $G(x)$ to be of degree m and n, with Bernstein coefficients $A_0^m,\ldots,A_m^m$ and $B_0^n,\ldots,B_n^n$. Then $W(x)$ is of degree mn, and its coefficients may be computed using the *product algorithm* of DeRose [4]. This populates a tetrahedral array of numbers $h_{i,j}^{(s)}$ as follows. In the first level, $s=0$, we set

$$h_{i,0}^{(0)} \;=\; A_i^m\,, \quad i=0,\ldots,m\,.$$

Successive levels $s=1,\ldots,m$ are then filled using the recursive formula

$$h_{i,j}^{(s)} \;=\; \sum_{k=\max(0,j-n)}^{\min(j,ns-n)} \frac{\binom{ns-n}{k}\binom{n}{j-k}}{\binom{ns}{j}}\,[\,(1-B_{j-k}^n)h_{i,k}^{(s-1)} + B_{j-k}^n h_{i+1,k}^{(s-1)}\,] \tag{19}$$

for $i = 0, \dots, m-s$ and $j = 0, \dots, ns$. Once the array is fully populated, the last level, $s = m$, contains the desired coefficients of $W(x)$:

$$W_j^{mn} \;=\; h_{0,j}^{(m)}\,, \quad j = 0, \dots, mn\,.$$

Although more complicated, expression (19) is somewhat similar in structure to degree elevation (10) and multiplication (16), so we omit implementation details. We adopt the operator $<<$ to denote composition — $F(u) << G(x)$ suggests "feeding" the expression $u = G(x)$ into the polynomial $F(u)$.

It should be noted that composition is relatively expensive compared to the arithmetic and other simple operations — the cost is $O(m^3n^2)$ flops.

4.5 Greatest common divisor

Related program:
Bernstein GCD(const Bernstein &u, const Bernstein &v, double epsilon)

The *greatest common divisor* $\gcd(F(x), G(x))$ of two polynomials is the polynomial of highest degree that divides *exactly* (i.e., with zero remainder) into both polynomials. The gcd of a given polynomial $P(x)$ and its derivative, $\gcd(P(x), P'(x))$, is an important example: it allows multiple roots of $P(x)$ to be identified in a root–finding procedure.

The standard method of computing $\gcd(F(x), G(x))$ is through *Euclid's algorithm* [24], which employs an iterated sequence of polynomial divisions. Assuming that $m \geq n$, we assign $\phi_0(x) = F(x)$, $\phi_1(x) = G(x)$, and compute successive polynomials $\phi_2(x), \dots, \phi_m(x)$ through the scheme

$$\begin{aligned}
\phi_0(x) &= q_1(x)\,\phi_1(x) \,+\, \phi_2(x)\,, \\
\phi_1(x) &= q_2(x)\,\phi_2(x) \,+\, \phi_3(x)\,, \\
&\cdots \\
\phi_{r-1}(x) &= q_r(x)\,\phi_r(x) \,+\, \phi_{r+1}(x)\,, \\
&\cdots \\
\phi_{m-1}(x) &= q_m(x)\,\phi_m(x)\,,
\end{aligned} \tag{20}$$

where $q_r(x)$ and $\phi_{r+1}(x)$ are the quotient and remainder upon dividing $\phi_{r-1}(x)$ by $\phi_r(x)$. The procedure is continued until we encounter a remainder $\phi_{m+1}(x)$ that vanishes identically, and we then have $\gcd(F(x), G(x)) = \phi_m(x)$.

The implementation of Euclid's algorithm in floating–point arithmetic is a delicate issue, since the vanishing–remainder termination criterion is never precisely attained due to round–off errors incurred in each division. It might seem that a reasonable termination criterion, in the context of floating–point arithmetic, is to test the remainder norm[4] $\|\phi_{r+1}(x)\|$ at each step against a prescribed tolerance ϵ, assuming that the input polynomials $F(x)$ and $G(x)$ are normalized before embarking on the Euclidean algorithm (20).

Example 1 Consider the polynomials

$$\begin{aligned} F(x) &= (x-0.19)^6(x-0.53)^4(x-0.81)^4\,, \\ G(x) &= (x-0.24)^3(x-0.53)^4(x-0.66)^4\,, \end{aligned}$$

for which $\gcd(F(x),G(x)) = (x-0.53)^4$. Representing these polynomials in Bernstein form and applying the division sequence (20), we obtain the data shown in Table 1. Since $\gcd(F(x),G(x))$ is known to be of degree 4, one might expect to observe a small remainder norm at stage 8. We see, however, that the remainder norm experiences dramatic and unpredictable changes at each successive stage: its comparison with a specified tolerance ϵ is thus an unreliable indicator of when to stop the Euclidean algorithm (20).

The inadequacy of $\|\phi_{r+1}(x)\|$ as an indicator of when to stop the division sequence (20) to obtain a satisfactory "approximate gcd" can be understood as follows. If we stop at the r–th step (i.e., the division of $\phi_{r-1}(x)$ by $\phi_r(x)$) we can show, by working backwards through (20), that the original polynomials can be written in the form

$$\begin{aligned} F(x) &= \phi_0(x) = \theta_0^r(x)\,\phi_r(x) + \theta_0^{r+1}(x)\,\phi_{r+1}(x)\,, \\ G(x) &= \phi_1(x) = \theta_1^r(x)\,\phi_r(x) + \theta_1^{r+1}(x)\,\phi_{r+1}(x)\,, \end{aligned}$$

for certain polynomials $\theta_0^r(x), \theta_0^{r+1}(x)$ and $\theta_1^r(x), \theta_1^{r+1}(x)$. When $\phi_{r+1}(x) \equiv 0$, $\phi_r(x)$ divides exactly into both $F(x)$ and $G(x)$. When $\phi_{r+1}(x) \not\equiv 0$, however, the value of $\|\phi_{r+1}(x)\|$ is *not* a good indicator of "how nearly" $\phi_r(x)$ divides into $F(x)$ and $G(x)$, since the divisions $F(x)/\phi_r(x)$ and $G(x)/\phi_r(x)$ produce remainders $\theta_0^{r+1}(x)\,\phi_{r+1}(x)$ and $\theta_1^{r+1}(x)\,\phi_{r+1}(x)$, not just $\phi_{r+1}(x)$.

[4]Since the norm (15) depends only on the polynomial behavior over $x \in [\,0,1\,]$, this can only be expected to accurately yield gcd factors corresponding to roots on this interval.

stage	dividend degree	divisor degree	remainder norm
1	14	11	1.2228×10^{0}
2	11	10	5.2620×10^{1}
3	10	9	6.6666×10^{-4}
4	9	8	3.2542×10^{1}
5	8	7	1.0953×10^{-5}
6	7	6	4.4323×10^{0}
7	6	5	3.4512×10^{-9}
8	5	4	1.2140×10^{-6}
9	4	3	1.4128×10^{-9}
10	3	2	5.5170×10^{-9}
11	2	1	1.6978×10^{-8}

Table 1: Remainder norms in applying (20) to the polynomials of Example 1.

Since a satisfactory "approximate gcd" should produce remainders with small norms when divided into the original polynomials $F(x)$ and $G(x)$, these considerations motivate the following definition (adapted from [22]):

Definition 1 The quasi–gcd of two normalized Bernstein–form polynomials $F(x)$ and $G(x)$, for a given tolerance ϵ, is the first polynomial $\phi_r(x)$ generated by (20) satisfying the condition that the divisions

$$\begin{aligned} F(x) &= q_1(x)\,\phi_r(x) \,+\, r_1(x)\,, \\ G(x) &= q_2(x)\,\phi_r(x) \,+\, r_2(x)\,, \end{aligned}$$

produce remainders with $\|r_1(x)\| < \epsilon$ and $\|r_2(x)\| < \epsilon$.

The use of this definition in an approximate gcd procedure clearly incurs additional expense, since at each step of the division sequence (20) we must divide the "candidate gcd" $\phi_r(x)$ into the original polynomials, and compute the norms of the resulting remainders (Schönhage [22] presents a complexity analysis for a somewhat different definition of the quasi–gcd). With $\epsilon = 10^{-7}$ in Definition 1, the ϵ–gcd of $F(x)$ and $G(x)$ in Example 1 is properly identified at the 8–th division stage, as shown in Table 2 below.
Table 3 compares the coefficients of the computed gcd, using the termination strategy in Definition 1, with the exact coefficients (obtained by writing the factor $(x-0.53)^4$ in Bernstein form).

stage	dividend degree	divisor degree	$\|r_1\|$	$\|r_2\|$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
6	7	6	1.8422×10^{-5}	2.8021×10^{-2}
7	6	5	2.0652×10^{-6}	6.3700×10^{-5}
8	5	4	4.9069×10^{-8}	8.1258×10^{-9}

Table 2: Remainder norms on dividing $F(x)$ and $G(x)$ by $\phi_r(x)$ in Example 1.

C_0^4	exact	3.560966909593
	computed	3.560966646640
C_1^4	exact	−3.157838580205
	computed	−3.157837897874
C_2^4	exact	2.800347420182
	computed	2.800347420182
C_3^4	exact	−2.483326957520
	computed	−2.483326561676
C_4^4	exact	2.202195603838
	computed	2.202195707067

Table 3: Comparison of computed and exact gcd coefficients for Example 1.

Since any gcd algorithm implemented in floating–point arithmetic must rely on a prescribed numerical tolerance for its termination, it is inherently less "robust" than the other polynomial operations described above. Round–off errors incurred by the gcd computation may be especially prominent when the number of division stages is large. Thus, when invoking the gcd function, careful consideration must be given to the choice of an appropriate tolerance, and the consequences of using an approximate gcd in subsequent applications (e.g., determination of multiple roots; see §5). A simple example illustrates the need for a cautionary attitude toward the use of this function:

Example 2 Given the three linear Bernstein–form polynomials

$$a(x) = 2.5(1-x) - 3.8x,\ b(x) = 4.5(1-x) - 1.8x,\ c(x) = 4(1-x) - 3x$$

we define $F(x) = a^p(x)\,c(x)$ and $G(x) = b^q(x)\,c(x)$, so that $\gcd(F(x), G(x)) = c(x)$ for any p and q. If $h(x) = h_0(1-x) + h_1 x$ is the quasi–gcd computed using the above procedure, the deviation of h_0/h_1 from the exact value $-4/3$ indicates the error in this approximate gcd. With $\epsilon = 10^{-7}$, we obtain the acceptable result $h_0/h_1 = -1.333333333333335$ when $(p, q) = (4, 3)$. With $(p, q) = (19, 18)$, however, the ratio becomes -1.3333333427725553 — i.e., the 15 extra division stages incur a loss of 7 significant digits.

To summarize, one must be realistic about the accuracy expected from a floating–point gcd function. As a simple rule of thumb, results that require more than $r = 10$ division stages in (20) should be regarded with skepticism. One must also keep in mind the important role that the tolerance ϵ plays in terminating the procedure. A truly robust floating–point gcd would entail a detailed study of the conditioning and backward error analysis of sequences of polynomial divisions, a task that is beyond our present scope.

5 Bernstein root–solver

Related programs:
Bernstein sROOT(const Bernstein &u, double delta, double eta)
Bernstein mROOT(const Bernstein &u, double delta, double eta, double epsilon)
Bernstein ROOT(const Bernstein &u, double delta, double eta, double epsilon)

The Bernstein form is ideally suited to the isolation and approximation of the real roots of a polynomial on a given interval, due to its numerical stability [9, 11] and the useful geometrical information contained in the coefficients. As a first step, the root–finder normalizes the Bernstein coefficients by calling Normalize(). A variety of root–finding methods [23] are then available, based on the following well–known properties:

1. **convex hull property**: the graph of $y = P(x)$ is confined within the *convex hull* of the control points $(x_k, y_k) = (k/n, C_k^n)$ for $k = 0, \ldots, n$;

2. **variation–diminishing property**: the number N of real roots of $P(x)$ on the open interval $(0, 1)$ is related to the number of sign changes V in its coefficients by $N = V - 2k$, where k is a non–negative integer;

3. **subdivision property**: the de Casteljau algorithm (11) splits $P(x)$ at any point $x_s \in [\,0, 1\,]$ and furnishes its Bernstein coefficients on the sub–intervals $[\,0, x_s\,]$ and $[\,x_s, 1\,]$.

Lane and Riesenfeld [20] describe a scheme combining iterated subdivision with the variation–diminishing property to isolate and approximate real roots on $[\,0,1\,]$. Although valuable in isolating the real roots, repeated subdivision is rather slow as a means of *approximating* them. A quadratically–convergent scheme, such as the Newton–Raphson method, is preferable for this purpose *provided* we can ensure convergence to a desired root before invoking it.

Henrici [16] specifies sufficient conditions for the convergence of Newton's method to a simple real root, in terms of the behavior of the first and second derivatives over the root isolating interval $[\,a,b\,]$. In terms of the Bernstein form on $[\,a,b\,]$, these conditions may be phrased as follows [7]:

Theorem 2 *Let the polynomial $P(x)$ have Bernstein coefficients $C_0^n,\ldots,C_n^n$ on $x\in[\,a,b\,]$, with first and second forward difference $\Delta C_k^n = C_{k+1}^n - C_k^n$ and $\Delta^2 C_k^n = \Delta C_{k+1}^n - \Delta C_k^n$. Then sufficient conditions for $P(x)$ to have a unique root on $[\,a,b\,]$, and for the Newton–Raphson iteration to converge to it from any starting point in the interval, may be phrased as follows*

1. $C_0^n C_n^n < 0\,;$
2. $\Delta C_0^n, \Delta C_{n-1}^n \neq 0$ *and* $V(\Delta C_0^n,\ldots,\Delta C_{n-1}^n) = 0\,;$
3. $V(\Delta^2 C_0^n,\ldots,\Delta^2 C_{n-2}^n) = 0\,;$
4. $|C_0^n| \leq n(b-a)|\Delta C_0^n|$ *and* $|C_n^n| \leq n(b-a)|\Delta C_{n-1}^n|\,,$

where $V(\cdots)$ is the number of sign variations in the indicated quantities.

Since $P(a) = C_0^n$ and $P(b) = C_n^n$, condition (1) ensures the existence of a real root on (a,b). Note that roots at the interval end–points must be tested for separately, by checking if $C_0^n = 0$ or $C_n^n = 0$. Condition (2) ensures that $P'(x) \neq 0$, and $P(x)$ is thus monotone, on $x \in [\,a,b\,]$ — hence, there is *only* one root. Finally, condition (3) guarantees that $P''(x) \geq 0$ or $P''(x) \leq 0$ for $x \in [\,a,b\,]$, while (4) requires the tangent lines to $P(x)$ at $x = a$ and $x = b$ to intersect the x–axis within the interval $[\,a,b\,]$.

The basic root–finding algorithm employs recursive binary subdivision of the interval $x \in [\,0,1\,]$ to identify sub–intervals on which the conditions of Theorem 2 are satisfied. During subdivision, the extent of each sub–interval and corresponding Bernstein coefficients for $P(x)$ are stored in stacks. Sub–intervals over which all the Bernstein coefficients are of the same sign may be discarded, since they cannot contain roots. If $P(x)$ has only simple roots on $[\,0,1\,]$, this root isolation process terminates with at most n sub–intervals on which the conditions of Theorem 2 hold. Newton–Raphson iterations may then be applied to obtain guaranteed quadratic convergence to the roots.

The ability of the root solver to isolate real roots is ultimately limited by the accumulation of floating–point errors incurred in successive subdivisions. Subdivision may be regarded as a linear map $\mathbf{M}$ transforming the Bernstein coefficients on an interval $[\,a,b\,]$ into those on any sub–interval $[\,\overline{a},\overline{b}\,]$. In the $\|\cdot\|_\infty$ norm, the condition number of $\mathbf{M}$ can be written [10] as

$$\kappa_\infty(\mathbf{M}) \,=\, [\,2f\, \max(u_{\overline{m}},v_{\overline{m}})\,]^n, \tag{21}$$

$u_{\overline{m}}=(\overline{m}-a)/(b-a)$, $v_{\overline{m}}=(b-\overline{m})/(b-a)$ being barycentric coordinates of the sub–interval midpoint $\overline{m}=\frac{1}{2}(\overline{a}+\overline{b})$, $f=(b-a)/(\overline{b}-\overline{a})$ being the "zoom factor," and n the polynomial degree. In the case $[\,a,b\,]=[\,0,1\,]$ we have $\frac{1}{2}\le \max(u_{\overline{m}},v_{\overline{m}})<1$, and hence $\kappa_\infty(\mathbf{M})\le(2f)^n$. For high–degree polynomials, this worst–case amplification of errors in the original Bernstein coefficients as subdivision proceeds to a high resolution can generate erroneous coefficient signs and compromise the root isolation process.[5] In §6.2 we show that, for the Chebyshev polynomials, this concern becomes significant at $n\approx 50$.

The above procedure is implemented in the function sROOT, intended for polynomials with only *simple* roots. Note that sROOT requires two numerical tolerances: delta is a tolerance on the magnitude of $P(x)$, used to check for roots at interval endpoints, while eta specifies the minimum allowed width of an interval generated during the recursive subdivision — if the conditions of Theorem 2 are not met on a sub–interval smaller than this tolerance, sROOT will terminate and output an error message.

Multiple roots require special treatment, since the criteria of Theorem 2 are not satisfied in any sub–interval, however small, that contains a multiple root. The identification of multiple roots may be based [24] upon a sequence of gcd computations: starting with $P_0(x)=P(x)$, we construct the sequence of polynomial defined recursively by the formula

$$P_k(x) \,=\, \frac{P_{k-1}(x)}{\gcd(P_{k-1}(x),P'_{k-1}(x))}$$

for $k=1,\ldots,m$, terminating when $P_m(x)=$ constant. Then $P(x)$ has no

[5] Since we use the *original* Bernstein coefficients of $P(x)$ on $x\in[\,0,1\,]$ for the Newton–Raphson iterations, the accuracy of a root that is successfully isolated (according to the conditions of Theorem 2) is not compromised by errors incurred in the subdivision process.

roots of multiplicity $> m$, and the polynomials defined by

$$Q_k(x) = \frac{P_k(x)}{P_{k+1}(x)} \quad \text{for } k = 1, \ldots, m-1$$

and $Q_m(x) = P_m(x)$ all have only *simple* roots, such that the roots of $Q_k(x)$ correspond to the roots of $P(x)$ of multiplicity k.

Once the polynomials $Q_k(x)$ are constructed using the gcd and polynomial division functions, we can invoke the simple–root solver sROOT and thus (in principle) determine the value and multiplicity of all the roots on $[0,1]$. This method is implemented in mROOT, which requires the numerical tolerance epsilon (for terminating the gcd computations) in addition to delta and eta.

Because of its reliance upon successive tolerance–based gcd computations, mROOT is obviously not robust when applied to polynomials of high degree with roots of high multiplicity. The determination of multiple roots is, in fact, an inherently ill–conditioned problem in finite–precision arithmetic. Caution is thus advisable when invoking the function mROOT.

Finally, ROOT checks whether a call to sROOT or mROOT is necessary, by testing (using the specified tolerance epsilon) if $\gcd(P(x), P'(x)) = \text{constant}$, and then makes the appropriate call. We emphasize again that use of sROOT is preferred whenever possible, since choosing a "loose" tolerance in the gcd computation can erroneously signal the presence of multiple roots.

6 Empirical tests of software library

A variety of empirical tests were conducted in order to assess the accuracy and robustness the software library — we describe below a few representative results from these experiments.

6.1 Arithmetic operations

The special treatment of the binomial coefficients, described in §3.1 above, is a key feature of the software library that allows accurate processing of high–degree polynomials. To illustrate this, we consider the explicit construction of Bernstein representations of successively higher degree for a rather trivial polynomial, namely, the constant $P(x) \equiv 1$. In the basis of any degree n, the Bernstein coefficients of $P(x)$ are simply $C_0^n = \cdots = C_n^n = 1$. Starting with $C_0^1 = C_1^1 = 1$, however, we explicitly compute these coefficients through successive polynomial multiplications

$$\sum_{j=0}^{m} C_j^m \, b_j^m(x) \; \times \; \sum_{k=0}^{1} C_k^1 \, b_k^1(x) \, ,$$

$m = 1, \ldots, n-1$, as invoked by the exponentiation operator $\wedge$. From (16), we see that these multiplications incur successive evaluation of the sums

$$C_k^{m+1} \;=\; \sum_{j=\max(0,k-1)}^{\min(m,k)} \frac{\binom{m}{j}\binom{1}{k-j}}{\binom{m+1}{k}} \, C_j^m C_{k-j}^1 \, .$$

Table 4 lists the root–mean–square deviations, about the nominal value of 1, of the Bernstein coefficients computed in this manner for n values up to 350. These errors are remarkably subdued, and grow very slowly with n — on the other hand, explicit evaluation of the binomial coefficients can rapidly incur enormous values: from the Stirling approximation $m! \approx \sqrt{2\pi m}\, \mathrm{e}^{-m} \, m^m$ one can estimate that, for even m, the largest binomial coefficient of order m is approximately $2^m \sqrt{2/\pi m}$ ($\approx 10^{104}$ for $m = 350$).

degree n	rms coefficient error
50	1.04×10^{-16}
100	4.65×10^{-16}
150	5.75×10^{-16}
200	1.08×10^{-15}
250	1.02×10^{-15}
300	1.34×10^{-15}
350	1.66×10^{-15}

Table 4: Root–mean–square errors in computed Bernstein coefficients.

6.2 Roots of Chebyshev polynomials

Our second example is concerned with constructing the Bernstein form of the Chebyshev polynomials on $[\,0,1\,]$ and computing their roots on this interval. The Chebyshev polynomial on $x \in [\,-1,+1\,]$ are defined [21] by

$$T_n(x) \;=\; \cos(n \cos^{-1} x) \, , \tag{22}$$

and may be constructed by the recursion formula

$$T_n(x) \;=\; 2x\,T_{n-1}(x) \;-\; T_{n-2}(x)\,, \tag{23}$$

commencing with $T_0(x) = 1$ and $T_1(x) = x$. To cast (23) in Bernstein form, we introduce the change of variables $u = \frac{1}{2}(x+1)$ that maps $x \in [\,-1,+1\,]$ into $u \in [\,0,1\,]$. With $T_0(u) = 1$ and $T_1(u) = 2u - 1$, the recursion is then

$$T_n(u) \;=\; 2\,(2u-1)\,T_{n-1}(u) \;-\; T_{n-2}(u)\,, \tag{24}$$

while (22) becomes $T_n(u) = \cos(n\cos^{-1}(2u-1))$. Hence, $T_n(u)$ has n distinct real roots between 0 and 1 — namely,

$$u_k \;=\; \frac{1}{2}\left[1 \,+\, \cos\!\left(\frac{2k+1}{2n}\,\pi\right)\right] \quad \text{for } k = 0,\ldots,n-1\,. \tag{25}$$

The following program illustrates the construction of a degree–n Chebyshev polynomial in Bernstein form by use of (24), and the computation of its roots using the root solver described in §5. The computed roots can be compared with the exact roots defined by expression (25).

```
#include "Bernstein.h"
// Demonstration for constructing a degree n Chebyshev polynomial and finding its roots:
void main()
{
    int i, n;
    n = 6;    // the polynomial degree
    double *zeros, pi = 3.14159265358979323846264;
    double R, delta, eta;
    Bernstein *T, result;

    // Set the tolerances for root solving:
    delta = 1e-13;
    eta = 1e-8;

    T = new Bernstein[n + 1];
    if (!T)   {   cout << "Allocation error.\n";   exit(1);   }
    zeros = new double[n];
    if (!zeros)   {   cout << "Allocation error.\n";   exit(1);   }

    // Set T[0] = 1:
    T[0].dgr = 0;
    T[0].cf[0] = 1.0;
```

```
    // Set T[1] = 2u – 1 = (-1.0)*(1.0 – u) + 1.0*(u):
    T[1].dgr = 1;
    T[1].cf[0] = -1.0; T[1].cf[1] = 1.0;

    // Recursion for constructing T[n]:
    for (i=2; i <= n; i++)
        T[i] = 2.0 * T[1] * T[i – 1] – T[i – 2];

    // Root solving (with the result sorted in ascending order):
    result = sort(sROOT(T[n],delta,eta));

    // Compute "standard" roots:
    for (i=n–1; i >= 0; i– –)
        zeros[n–1–i] = 0.5 + 0.5 * cos((2.0 * (double)i + 1.0) * pi / (2.0 * (double)n));

    // Compute RMS error:
    ... (omitted) ...
    // Print the results:
    ... (omitted) ...
}
```

u_0	computed	0.01703708685546585
	exact	0.01703708685546590
u_1	computed	0.14644660940672632
	exact	0.14644660940672627
u_2	computed	0.37059047744873957
	exact	0.37059047744873969
u_3	computed	0.62940952255126048
	exact	0.62940952255126037
u_4	computed	0.85355339059327373
	exact	0.85355339059327373
u_5	computed	0.98296291314453410
	exact	0.98296291314453410

Table 5: Computed and exact roots for the degree–6 Chebyshev polynomial.

Table 5 presents results for the degree–6 Chebyshev polynomial obtained using sROOT with delta $= 10^{-13}$ and eta $= 10^{-8}$. The computed values agree with the exact roots to at least 15 significant digits in all cases, and the rms error of the computed roots is 7.0×10^{-17}. Similar computations were also performed for n up to 50, and the results are summarized in Table 6. As expected, there is a steady degradation of the accuracy with increasing n. At $n = 50$, the accumulated floating–point errors incur erroneous satisfaction of the conditions of Theorem 2, that cause 3 of the 50 Newton–Raphson iterations to wander into adjacent isolating intervals. For cases with $n \leq 20$, however, the software performs remarkably well. It should be noted that the results in Tables 5 and 6 are far superior to what can be obtained using the familiar "power" form of the Chebyshev polynomials.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$
rms error	6.36×10^{-16}	5.99×10^{-13}	2.09×10^{-10}	4.45×10^{-8}

Table 6: RMS errors in computed roots of degree–n Chebyshev polynomials.

7 Closure

In applications requiring computations with polynomials over finite intervals, the Bernstein representation offers enhanced numerical stability and intuitive algorithms for many basic functions. The development of an object–oriented software library for Bernstein–form polynomials, as described herein, allows the programmer to take advantage of these features, and implement complex polynomial manipulations in an extremely concise and reliable manner that does not pre–suppose an intimate knowledge of the underlying mathematics. Empirical tests (e.g., computing the roots of Chebyshev polynomials) show that the library is capable of excellent performance in typical circumstances. One should be realistic, however, about performance on problems that are inherently ill–posed in the context of floating–point arithmetic (e.g., greatest common divisors and multiple roots). We believe that this library will help accelerate the development of software packages for a variety applications.

References

[1] K. Atkinson (1989), *An Introduction to Numerical Analysis* (2nd edition), Wiley, New York.

[2] S. N. Bernstein (1912), On the best approximation of continuous functions by polynomials of a given degree (in Russian), *Comm. Kharkow Math. Soc.* (Series 2), **13**, 49–194.

[3] M. Daniel and J. C. Daubisse (1989), The numerical problem of using Bézier curves and surfaces in the power basis, *Computer Aided Geometric Design* **6**, 121–128.

[4] A. D. DeRose (1988), Composing Bézier simplexes, *ACM Transactions on Graphics* **7**, 198–221.

[5] G. E. Farin (1997), *Curves and Surfaces for Computer–Aided Geometric Design* (4th edition), Academic Press, San Diego.

[6] R. T. Farouki (1991), On the stability of transformations between power and Bernstein polynomial forms, *Computer Aided Geometric Design* **8**, 29–36.

[7] R. T. Farouki (1991), Computing with barycentric polynomials, *The Mathematical Intelligencer* **13**, 61–69.

[8] R. T. Farouki (2000), Legendre–Bernstein basis transformations, *Journal of Computational and Applied Mathematics* **119**, 145–160.

[9] R. T. Farouki and T. N. T. Goodman (1996), On the optimal stability of the Bernstein basis, *Mathematics of Computation* **65**, 1553–1566.

[10] R. T. Farouki and C. A. Neff (1990), On the numerical condition of Bernstein–Bézier subdivision processes, *Mathematics of Computation* **55**, 637–647.

[11] R. T. Farouki and V. T. Rajan (1987), On the numerical condition of polynomials in Bernstein form, *Computer Aided Geometric Design* **4**, 191–216.

[12] R. T. Farouki and V. T. Rajan (1988), Algorithms for polynomials in Bernstein form, *Computer Aided Geometric Design* **5**, 1–26.

[13] R. T. Farouki, Y–F. Tsai, and C. S. Wilson (2000), Physical constraints on feedrates and feed accelerations along curved tool paths, *Computer Aided Geometric Design* **17**, 337–359.

[14] P. Goetgheluck (1987), Computing binomial coefficients, *American Mathematical Monthly* **94**, 360–365.

[15] P. Goetgheluck (1988), On prime divisors of binomial coefficients, *Mathematics of Computation* **51**, 325–329.

[16] P. Henrici (1964), *Elements of Numerical Analysis*, Wiley, New York.

[17] T. Hermann (1996), On the stability of polynomial transformations between Taylor, Bézier, and Hermite forms, *Numerical Algorithms* **13**, 307–320.

[18] J. Hoschek and D. Lasser (1993), *Fundamentals of Computer–Aided Geometric Design* (translated by L. L. Schumaker), AK Peters, 1993.

[19] J. M. Lane and R. F. Riesenfeld (1980), A theoretical development for the computer generation and display of piecewise polynomial surfaces, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI–2**, 35–46.

[20] J. M. Lane and R. F. Riesenfeld (1981), Bounds on a polynomial, *BIT* **21**, 112–117.

[21] T. J. Rivlin (1990), *Chebyshev Polynomials: From Approximation Theory to Algebra and Number Theory* (2nd edition), Wiley, New York.

[22] A. Schönhage (1985), Quasi–GCD computations, *Journal of Complexity* **1**, 118–137.

[23] M. R. Spencer (1994), *Polynomial Real Root Finding in Bernstein Form*, PhD Dissertation, Brigham Young University.

[24] J. V. Uspensky (1948), *Theory of Equations*, McGraw–Hill, New York.