Polynomial operations

Application of discrete models

The previous document introduced the basic concepts of polynomials. In this document, we will discuss how to perform various operations on polynomials – either manually, or with a computer. We will see the fundamental operations (similarly to integers): addition, subtraction, multiplication and division with remainder, but we will also see some operations that are specific to polynomials.

For every operation, we will assume that the input polynomials are given in canonical form (e.g. $p = 2x^3 - 6x + 4$, see the previous document), and we will provide the result in the same form. On computer, we only need to store the coefficients (including the zeros for the missing terms), typically starting from the constant term (e.g. for $p = 2x^3 - 6x + 4$, we store $4, -6, 0, 2$), so that the coefficient of degree k is at index k in the list (when indexing from 0).

1 Addition, subtraction

Polynomial addition ($p+q$) and subtraction ($p-q$) are among the simplest polynomial operations. They are performed simply by adding/subtracting the corresponding terms. On paper, the two polynomials should be aligned carefully so that the corresponding terms are above each other. (This means leaving spaces for the missing terms, lest the wrong terms are added together.)

Example 1. Add $p = 2x^3 + x^2 - 5$ and $q = 2x^2 - 2x + 1$ in $\mathbb{Z}[x]$:

$$\begin{array}{rr} & 2x^3 \;+ x^2 \;\;\;\;\;\;\; - 5 \\ + & \;\;\;\;\;\;\; 2x^2 - 2x + 1 \\ \hline & 2x^3 + 3x^2 - 2x - 4 \end{array}$$

Example 2. From $p = x^3 + 2x^2 + x$, subtract $q = x^3 + 3x^2 - 2x + 4$ in $\mathbb{Z}[x]$:

$$\begin{array}{rr} & x^3 + 2x^2 \;\; + x \\ -(& x^3 + 3x^2 - 2x + 4 \,) \\ \hline & \;\;\;\;\; - x^2 + 3x - 4 \end{array}$$

Example 3. The coefficients need not be integers, they can also be from $\mathbb{Z}_m$ (where we calculate modulo m, see: Congruences). Add $p = \overline{2}x^4 + x^3 + \overline{4}x^2 + \overline{3}$ and $q = \overline{3}x^4 + \overline{4}x^3 + \overline{2}x^2 + \overline{2}x + \overline{1}$ in $\mathbb{Z}_5[x]$:

$$\begin{array}{rr} & \overline{2}x^4 \;\; + x^3 + \overline{4}x^2 \;\;\;\;\;\;\; + \overline{3} \\ + & \overline{3}x^4 + \overline{4}x^3 + \overline{2}x^2 + \overline{2}x + \overline{1} \\ \hline & \; x^2 + \overline{2}x + \overline{4} \end{array}$$

We can formally describe the rules for the two operations, i.e. how to calculate each coefficient of the result from the coefficients of the two input polynomials. Let p_k be the degree-k coefficient of p (if it doesn't exist, i.e. $k > \deg p$, then use 0), and similarly for q, $(p+q)$ etc. Then:

$$(p+q)_k := p_k + q_k,$$
$$(p-q)_k := p_k - q_k.$$

Theorem 1.1 (degree of sum and difference)**.** *If $p, q \in K[x]$, then*

$$\deg(p+q) \le \max(\deg p, \deg q),$$
$$\deg(p-q) \le \max(\deg p, \deg q),$$

and if $\deg p \neq \deg q$, then it is equality (=) in both cases.

Example. In the first two examples, $\deg(p+q) = 3 = \max(3,2)$ and $\deg(p-q) = 2 < \max(3,3) = 3$.

So the result's degree cannot be higher than the higher of the input degrees. But it can be lower, if the input degrees are equal and the leading coefficients cancel out (as in the last two examples).

Time complexity: adding or subtracting two polynomials of degree $\le n$ requires at most $n+1$ additions or subtractions of coefficients, i.e. these operations run in *linear* time.

The polynomial operations are very similar to the corresponding operations of multi-digit integers. For example:

$$\begin{array}{r} 314 \\ +\ 271 \\ \hline 585 \end{array} \qquad \begin{array}{r} 31415 \\ +\ 2717 \\ \hline 34132 \end{array} \qquad \begin{array}{r} 31415 \\ -\ 2717 \\ \hline 28698 \end{array} \qquad \begin{array}{r} 98765 \\ +\ 1235 \\ \hline 100000 \end{array}$$

The reason is that integers can be written as a polynomial of the digits with $x = 10$, e.g. $314 = 3 \cdot 10^2 + 1 \cdot 10 + 4$. Indeed, the first addition can be translated to polynomials digit by digit: $(3x^2+x+4)+(2x^2+7x+1) = 5x^2+8x+5$.

But not the others. The difference is that integer addition can introduce a *carry*, i.e. if the result of a digit does not fit between 0 and 9, then it will also influence the previous digit. For example, the last digit of the second addition would be $5+7=12$, but only the 2 became the last digit, while the 1 (i.e. the carry) was added to the previous digit ($2 \to 3$). If they were polynomials, then the last two coefficients would remain 12 and 2, and no carry would happen.

Sometimes the carry influences not only one, but multiple digits in a row. For example, in the rightmost addition, the carry in the last digit had a cascading effect that influenced *all* other digits. In fact, this is an example for the fact that – unlike for polynomials – integer addition can produce longer numbers than both of the input numbers.

2 Multiplication

Multiplying two polynomials is more complicated than adding or subtracting them: we need to multiply every term by every other term, and then finally add the corresponding terms.

Example. To multiply $p = x^3 + 2x^2 - 3x + 4 \in \mathbb{Z}[x]$ and $q = 2x^2 - x + 3 \in \mathbb{Z}[x]$, we first need to multiply all four terms of p by all three terms of q. In table form:

$(\cdot)$	x^3	$2x^2$	$-3x$	4
$2x^2$	$2x^5$	$4x^4$	$\mathbf{-6x^3}$	$8x^2$
$-x$	$-x^4$	$\mathbf{-2x^3}$	$3x^2$	$-4x$
3	$\mathbf{3x^3}$	$6x^2$	$-9x$	12

Then we need to add the resulting 12 products together, grouped by exponents (notice how the terms with the same exponent are layed out diagonally, see e.g. the x^3 terms marked in bold above):

$$p \cdot q = 2x^5 + (4x^4 - x^4) + (-6x^3 - 2x^3 + 3x^3) + (8x^2 + 3x^2 + 6x^2) + (-4x - 9x) + 12 =$$
$$= 2x^5 + 3x^4 - 5x^3 + 17x^2 - 13x + 12.$$

In general, the coefficients of the product can be written as follows:

$$(p \cdot q)_k = \sum_{i=0}^{k} p_i q_{k-i} = p_0 q_k + p_1 q_{k-1} + \ldots + p_{k-1} q_1 + p_k q_0.$$

Example. In the above example, the cubic term $(-5x^3)$ has the following coefficient:

$$(p \cdot q)_3 = p_0 q_3 + p_1 q_2 + p_2 q_1 + p_3 q_0 = 4 \cdot 0 + (-3) \cdot 2 + 2 \cdot (-1) + 1 \cdot 3 = -6 - 2 + 3 = -5.$$

For manual calculation on paper, the rows should be written in such a way that the like terms are in the same column, to make the final addition easier.

Example. The product of the above p and q can be calculated as follows:

$$\begin{array}{rr} & (\ x^3 \ + 2x^2 \ - 3x \ + 4\ ) \cdot (2x^2 - x + 3) \\ \hline & 2x^5 + 4x^4 - 6x^3 + 8x^2 \\ & \quad -x^4 - 2x^3 + 3x^2 - 4x \\ + & \quad\quad 3x^3 + 6x^2 - 9x + 12 \\ \hline & 2x^5 + 3x^4 - 5x^3 + 17x^2 - 13x + 12 \end{array}$$

Theorem 2.1 (degree of product)**.** *Let $p, q \in K[x]$. If K is an integral domain, then*

$$\deg(p \cdot q) = \deg p + \deg q,$$

otherwise (i.e. if K has zero divisors):

$$\deg(p \cdot q) \leq \deg p + \deg q.$$

(Reminder: all of the usual number sets are integral domains: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, and so is $\mathbb{Z}_p$ for prime numbers, see the document called Algebraic structures.)

Example. In the above example, $\deg p = 3$ and $\deg q = 2$, and indeed: $\deg(p \cdot q) = 5 = 3 + 2$.

Example. $\mathbb{Z}_6$ is not an integral domain (e.g. zero divisors: $\overline{2} \cdot \overline{3} = \overline{0}$), so only "$\leq$" holds in $\mathbb{Z}_6[x]$:

$$p = \overline{2}x + \overline{1},\ q = \overline{3}x + \overline{1} \quad \Longrightarrow \quad p \cdot q = \overline{5}x + \overline{1} \quad \Longrightarrow \quad \deg(p \cdot q) = 1 \leq 1 + 1.$$

The reason for this is that the zero divisors made the quadratic term disappear: $\overline{2}x \cdot \overline{3}x = \overline{0}x^2$.

Example. The theorem is also true for the zero polynomial (for which $\deg 0 = -\infty$, see the previous document):

$$p = x^2,\ q = 0 \quad \Longrightarrow \quad p \cdot q = 0 \quad \Longrightarrow \quad \deg(p \cdot q) = -\infty = 2 + (-\infty).$$

Time complexity: multiplying two degree-n polynomials requires at most $(n+1)^2$ multiplications of coefficients and around the same number of additions. That is, polynomial multiplication runs in *quadratic* time.

It is worth examining a special case of polynomial multiplication, when one polynomial is constant, because then the operation becomes much easier: each coefficient is simply multiplied by that constant. Then the time complexity is *linear* (like for addition and subtraction).

Example. If $p = x^3 - 3x^2 + 2x - 4 \in \mathbb{Z}[x]$ and $q = -3$, then $p \cdot q$ can be calculated as follows:

$$\begin{array}{r} (\ x^3 \ - 3x^2 \ + 2x \ - 4\ ) \cdot (-3) \\ \hline -3x^3 + 9x^2 - 6x + 12 \end{array}$$

3 Division with remainder

Just like for integers, division doesn't generally work for polynomials either (e.g. $x^2/(x+1)$ is not a polynomial, in the same way as $2/3 \notin \mathbb{Z}$). But just like for integers, we can introduce **division with remainder** (a.k.a. **Euclidean division**) for polynomials (cf. Divisors, prime numbers, Theorem 1.1):

Theorem 3.1 (Division theorem for polynomials)**.** *If K is a field, $a, b \in K[x]$ and $b \neq 0$, then there exist unique $q, r \in \mathbb{R}[x]$ for which $a = qb + r$ and $\deg r < \deg b$.*

Definition 3.2. q is the **quotient polynomial** of a and b, and r is their **remainder polynomial**.

Example. If $a = 2x^3 + x^2 - 2x - 1$ and $b = x^2 - 2x + 1$, then $q = 2x + 5$ and $r = 6x - 6$, because $a = 2x^3 + x^2 - 2x - 1 = (2x + 5)(x^2 - 2x + 1) + (6x - 6)$ and $\deg(6x - 6) < \deg(x^2 - 2x + 1)$.

It is important that division with remainder only works over *fields* (in general). (Reminder: in fields, division is always possible, e.g. $\mathbb{Q}$ and $\mathbb{R}$ are fields, but $\mathbb{Z}$ is not, see: Algebraic structures.)
Example. Dividing $a = x^2 + 3$ by $b = 2x$ gives $q = 1/2 \cdot x$ and $r = 3$. That is, division doesn't always work in $\mathbb{Z}[x]$, only in $\mathbb{Q}[x]$, because we need to divide by b's leading coefficient (see later).
Example. But it works over other fields too: in $\mathbb{Z}_5[x]$, dividing $a = x^2 + \overline{3}$ by $b = \overline{2}x$ gives the quotient $q = \overline{3}x$ and the remainder $r = \overline{3}$ (because in $\mathbb{Z}_5$, $\overline{2}^{-1} = \overline{3}$).

In Sage, polynomial division works in the same way as integer division (`//` and `%`):

```
sage: P.<x> = QQ[]    # not ZZ[]!
sage: a = 2*x^3 + x^2 - 2*x - 1; b = x^2 - 2*x + 1
sage: a // b
2*x + 5
sage: a % b
6*x - 6
```

Attention: avoid regular division (`/`), because the type of the result is not polynomial:

```
sage: a / b          # the result is not polynomial
(2*x^2 + 3*x + 1)/(x - 1)
sage: (b*x^2) / b    # not even when it looks so!
x^2
sage: _.degree()     # doesn't work, because it is not a polynomial
[...]
AttributeError: '[...]' object has no attribute 'degree'
```

Division with remainder can be done manually using a method called **long division** (similarly to integers). Starting from the dividend (a), we need to subtract multiples of b from it to make the leading coefficients disappear, thus decreasing the degree of the polynomial step by step. This is repeated until the degree gets below $\deg b$, where we get the remainder.
Example 1. From the above example, divide $a = 2x^3 + x^2 - 2x - 1$ by $b = x^2 - 2x + 1$. The first step is to delete the leading term of a (i.e. $2x^3$). Dividing the leading terms give $2x^3/x^2 = 2x$, which means that we need to subtract $2x$ times b in the first step:

$$\begin{array}{r} (\ 2x^3 \ + x^2 - 2x - 1\ ) : (\ x^2 - 2x + 1\ ) = 2x \\ \hline -(\ 2x^3 - 4x^2 + 2x\ ) \\ \hline 5x^2 - 4x - 1 \end{array}$$

We collect the multipliers of b (e.g. $2x$) after the equals sign ($=$), to build the quotient. We multiplied b by this $2x$ in the second row, and subtracted the result from the first row. Continuing from this, $5x^2/x^2 = 5$, so we need to subtract 5 times b (and also add the term 5 to the quotient):

$$\begin{array}{r} (\ 2x^3 \ + x^2 \ - 2x - 1\ ) : (\ x^2 - 2x + 1\ ) = 2x + 5 \\ \hline -(\ 2x^3 \ - 4x^2 \ + 2x\ ) \\ \hline 5x^2 \ - 4x - 1 \\ \hline -(\ 5x^2 - 10x + 5\ ) \\ \hline 6x - 6 \end{array}$$

We can stop here, because the degree of the result went under $\deg b = 2$. Therefore, the remainder is $r = 6x - 6$, and the quotient was assembled from the multipliers: $q = 2x + 5$.

Example 2. Divide $a = 3x^2 - 2x + 1$ by $b = 2x + 1$. Now the leading coefficient of b is not 1, which slightly complicates the process, because it introduces fractions (e.g. $3x^2/2x = \frac{3}{2}x$):

$$\begin{array}{r} (\ 3x^2\ -\ 2x\ +\ 1\ ) : (\ 2x+1\ ) = \frac{3}{2}x - \frac{7}{4} \\ \underline{-(\ 3x^2\ +\ \frac{3}{2}x\ )\qquad\qquad} \\ -\frac{7}{2}x\ +\ 1 \\ \underline{-(\ -\frac{7}{2}x\ -\ \frac{7}{4}\ )} \\ \frac{11}{4} \end{array}$$

So the quotient is $q = \frac{3}{2}x - \frac{7}{4}$, and the remainder is the constant polynomial $r = \frac{11}{4}$.

Example 3. Divide $a = 3x^4 - 7x^3 - 5x + 10$ by $b = x - 2$. Make sure to leave room for the missing quadratic term ($0x^2$), lest the wrong terms are subtracted from each other:

$$\begin{array}{r} (\ 3x^4\ -\ 7x^3\qquad -\ 5x\ +\ 10\ ) : (\ x-2\ ) = 3x^3 - x^2 - 2x - 9 \\ \underline{-(\ 3x^4\ -\ 6x^3\ )\qquad\qquad\qquad\qquad} \\ -\ x^3 \\ \underline{-(\ -\ x^3\ +\ 2x^2\ )\qquad\qquad} \\ -\ 2x^2\ -\ 5x \\ \underline{-(\ -\ 2x^2\ +\ 4x\ )\qquad} \\ -\ 9x\ +\ 10 \\ \underline{-(\ -\ 9x\ +\ 18\ )} \\ -8 \end{array}$$

So the quotient is $q = 3x^3 - x^2 - 2x - 9$, and the remainder is $r = -8$.

Time complexity: if the divisor and the quotient have degree $\leq n$, then long division requires at most $(n+1)^2$ multiplications of coefficients and around the same number of subtractions, i.e. this operation also runs in *quadratic* time (like multiplication).

We will see another method for polynomial division at the end of the next section, but that works only if the divisor is a linear polynomial of the form $x - c$ (like in the above last example).

4 Evaluation

This section is about the polynomial operation called *evaluation*, which means calculating the value $p(c)$ for a given polynomial p and value c.

We can do this directly from the canonical form, i.e. $p(c) = a_n c^n + a_{n-1}c^{n-1} + \ldots + a_1 c + a_0$ – but this is not efficient: it needs n multiplications for $a_n c^n$, $n-1$ multiplications for $a_{n-1}c^{n-1}$ etc., which is $n + (n-1) + \ldots + 2 + 1 = n(n+1)/2$ multiplications in total, plus n additions. (For example, if $\deg p = 10$, then it needs 55 multiplications and 10 additions.)

This can be improved by going from right to left, and using the previous value of c^k to calculate the next c^{k+1} with only one extra multiplication: $c^{k+1} = c^k \cdot c$. This methods needs only $2n-1$ multiplications and n additions in total (which for $\deg p = 10$ is 19 multiplications and 10 additions).

But even more efficient is **Horner's method** (or *Horner's scheme*), which turns the polynomial into a form that allows a very efficient evaluation. For this, we first extract x from all the terms we can, then repeat this for the expression inside the just created parentheses, and do this again and again until all exponents disappear. For example:

$$\begin{aligned} p &= 2x^4 + 3x^3 - 2x^2 - 4x + 3 = \\ &= (2x^3 + 3x^2 - 2x - 4)x + 3 = \\ &= ((2x^2 + 3x - 2)x - 4)x + 3 = \\ &= (((2x + 3)x - 2)x - 4)x + 3. \end{aligned}$$

In general, this means the following transformation:

$$\begin{aligned} p &= a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2 x^2 + a_1 x + a_0 = \\ &= ((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_2)x + a_1)x + a_0. \end{aligned}$$

This form makes evaluation very efficient: in each step, we multiply a parenthesized subexpression (first a_n) by x (or its value, c), then add the next coefficient to it to get the next parenthesized subexpression, and continue this until we finally get the whole value $p(c)$. This process requires only n multiplications and n additions.

Horner's method:

Input: the coefficients $a_0, a_1, \ldots, a_n$ and the value c
$b := a_n$
for $k = n-1$ *to* 0 **do**
  $b := b \cdot c + a_k$
Output: b

In each step, the value of b equals to one of the parenthesized subexpressions, e.g. after the first step, $b = a_n c + a_{n-1}$, then $b = (a_n c + a_{n-1})c + a_{n-2}$ etc.

(Note: this algorithm doesn't work directly for the zero polynomial, because it doesn't have any coefficients. But this can be fixed easily by giving it a single zero coefficient: $n = 0$ and $a_0 = 0$. In general, the algorithm doesn't require that $a_n \neq 0$, i.e. we can add redundant 0's at the end of the coefficient list without changing the result.)

Example. Let $p = 3x^4 - 7x^3 - 5x + 10$ and $c = 2$. Then Horner's scheme gives

$$p = (((3x - 7)x + 0)x - 5)x + 10,$$

and so the variable b gets the following values throughout the algorithm:

$$\begin{aligned} &b := 3, \\ &b := b \cdot 2 - 7 = -1, \\ &b := b \cdot 2 + 0 = -2, \\ &b := b \cdot 2 - 5 = -9, \\ &b := b \cdot 2 + 10 = -8 \quad \longrightarrow \quad p(c) = -8. \end{aligned}$$

Manually, Horner's method can be done in a table form as follows. The first row contains the coefficients of p (including the missing 0's), the third row is filled with the values of b, calculated by adding the numbers above them in the same columns (in the first column, this is just the leading coefficient), and in the second row, the previous b values are multiplied by c.

Example. For the above polynomial $p = 3x^4 - 7x^3 - 5x + 10$, the table looks like this:

	3	−7	0	−5	10
$c = 2$		6	−2	−4	−18
	3	−1	−2	−9	−8

$\longrightarrow \quad p(2) = -8$

That is, we copied 3 down, multiplied it by $c = 2$ ($\rightarrow 6$), then added the second column together ($-7 + 6 = -1$), then multiplied the result again by $c = 2$ ($\rightarrow -2$), and so on in a zigzag line.

But Horner's method can do more than just evaluation. Compare the above b values (the third row of the table) with Example 3. from the previous section, where we divided the same polynomial by $b = x - 2$. Notice that the last b value (-8) is the remainder: $r = -8$, and the previous b values $(3, -1, -2, -9)$ are the coefficients of the quotient: $q = 3x^3 - x^2 - 2x - 9$. This is also true in general:

Theorem 4.1. *If $p \in K[x]$ and $c \in K$, then the remainder of $p = a_n x^n + \ldots + a_1 x + a_0$ and $b = x - c$ is the constant polynomial $p(c)$, and their quotient is $q = b_{n-1}x^{n-1} + \ldots + b_1 x + b_0$, where the b_k's are the b values in the algorithm of Horner's method, more precisely $b_{n-1} := a_n$ and $b_k := b_{k+1}c + a_{k+1}$.*

(It is easy to prove the remainder part of the statement: by the division theorem (see above), $p = (x - c)\, q + r$, where $\deg r < \deg b = 1$, i.e. the remainder is a constant polynomial. Substituting c into this, we get the desired statement: $p(c) = (c - c)\, q(c) + r = r$. Proving the quotient part is more complicated, so it is omitted.)

The above usage of Horner's method for polynomial division is also called **synthetic division**.

5 Formal derivative

We know the concept of **derivative** from calculus, and it has several useful properties for polynomials too. But here, we don't want to use the full machinery of calculus (limits, continuity etc.), and in some cases, we simply can't (because there is no continuity in *discrete* structures like $\mathbb{Z}$ or $\mathbb{Z}_m$). Therefore, instead of using calculus, we simply *define* the derivative for polynomials:

Definition 5.1. Let $p \in K[x]$ be a polynomial of degree n:

$$p = a_n x^n + \ldots + a_k x^k + \ldots + a_2 x^2 + a_1 x + a_0.$$

Its **formal derivative** is the following polynomial $p' \in K[x]$:

$$p' := n a_n x^{n-1} + \ldots + k a_k x^{k-1} + \ldots + 2a_2 x + a_1,$$

where the product $k a_k$ means $a_k + a_k + \ldots + a_k$ with exactly k terms. (p' is pronounced "p prime".)

Example. The derivative of $p = x^3 + 2x^2 - 3x + 5 \in \mathbb{Z}[x]$ is $p' = 3x^2 + 4x - 3$.

Note: the qualifier "formal" indicates that this definition of the derivative is not based on calculus.

Note 2: in the above definition, the meaning of $k a_k$ had to be given, because the coefficients a_k are not necessarily numbers, but merely elements of an arbitrary ring K, and we haven't yet defined how to multiply them by an integer (note that the exponents are always integers, no matter what ring the coefficients are from). For example, in $\mathbb{Z}_3[x]$, the formal derivative of $p = \overline{2}x^5$ is $p' = 5 \cdot \overline{2}x^4 = (\overline{2} + \overline{2} + \overline{2} + \overline{2} + \overline{2})x^4 = \overline{1}x^4 = x^4$.

Note 3: the definition suggests that the derivative of a degree-n polynomial has degree $n-1$. This is true for real polynomials ($\mathbb{R}[x]$), but not always in $\mathbb{Z}_m[x]$ (not even if $\mathbb{Z}_m$ is a field): e.g. the cubic $p = x^3 + \overline{2}x^2 + \overline{2}x + \overline{1} \in \mathbb{Z}_3[x]$ has only a linear derivative: $p' = x + \overline{2}$, because the quadratic term of p' vanishes in $\mathbb{Z}_3$: $3 \cdot \overline{1}x^2 = (\overline{1} + \overline{1} + \overline{1})x^2 = \overline{0}x^2$. It can also happen that the whole polynomial vanishes, e.g. the derivative of $p = x^6 + x^3 + \overline{1} \in \mathbb{Z}_3[x]$ is $p' = \overline{0}$.

In Sage, the method `p.diff()` calculates the derivative of a polynomial.

Theorem 5.2 (properties of formal derivative)**.** *For any $p, q \in K[x]$ and constant $c \in K$:*

1. $c' = 0$;
2. $x' = 1$ *(= the indentity element of K);*
3. $(p + q)' = p' + q'$;
4. $(p - q)' = p' - q'$;
5. $(c \cdot p)' = c \cdot p'$;
6. $(p \cdot q)' = p' \cdot q + p \cdot q'$;
7. $(p^n)' = n \cdot p' \cdot p^{n-1}$.

Calculating the formal derivative from the definition is trivial: just multiply each coefficient with the appropriate constant (and shift the indices by one). This requires n multiplications, i.e. this operation also runs in *linear* time.

But sometimes we don't need p' as a polynomial, only its value at some c, i.e. the value of $p'(c)$. We can of course calculate this by generating p' using the definition, and then evaluating it at c using Honner's method as described above. This requires $2n$ multiplications and n additions in total (and some memory to store p' itself, if needed).

But there is another way: Horner's method can be extended to calculate $p'(c)$ directly, along with $p(c)$, without calculating p' itself. This is allowed by the following theorem:

Theorem 5.3. *If $p \in K[x]$, $c \in K$ and the quotient of p when divided by $(x-c)$ is $q \in K[x]$, i.e. $p = (x-c)\,q + p(c)$, then $p'(c) = q(c)$.*

(Proof: $p' = ((x-c)\,q + p(c))' = q + (x-c)\,q'$, and substituting c into this gives $p'(c) = q(c)$.)

We have seen above that Horner's method calculates q automatically, so we can just repeat Horner's method to evaluate $q(c)$, which, by this theorem, is exactly $p'(c)$. Also, these two steps can be combined, so we don't need to store the whole q. (This means a total of around $2n$ multiplications and $2n$ additions, but if we already need $p(c)$ anyway, then $p'(c)$ is only an additional n multiplications and n additions, which is more efficient than calculating p' and evaluating it.)

Example. For $p = 3x^4 - 7x^3 - 5x + 10$ and $c = 2$, the extended Horner's method works like this:

	3	-7	0	-5	10	$\longleftarrow$	$p = 3x^4 - 7x^3 - 5x + 10$
$c = 2$		6	-2	-4	-18		
	3	-1	-2	-9	-8	$\longleftarrow$	$q = 3x^3 - x^2 - 2x - 9, \quad p(2) = -8$
$c = 2$		6	10	16			
	3	5	8	7		$\longleftarrow$	$q(2) = p'(2) = 7$

Extended Horner's method:

Input: the coefficients of p: $a_0, a_1, \ldots, a_n$, and the value c
$b := a_n$
$d := 0$
for $k = n-1$ *to* 0 **do**
  $d := d \cdot c + b$
  $b := b \cdot c + a_k$
Output: (b, d) – meaning $(p(c), p'(c))$

6 Cyclic redundancy check (CRC)

When data is transmitted through a physical channel (e.g. in a wire or through the air with radio signals), errors can happen, i.e. the received data may not be exactly the same as what was sent. How can we detect such errors?

The solution is *redundancy*: extend the message to include a *check value*, which is calculated from the message, i.e. it doesn't add any new information, but it can be used to check whether the data is correct. If the receiver finds that the check value doesn't correspond to the message, than they can be sure that the message or the check value (or both) was corrupted.

A simple example is the **parity bit**: the message, which is a bit sequence, is extended by a single new bit: if the message contains an even number of 1 bits, then append 0, otherwise append 1. Then the result always has an even number of 1's. For example: $0101 \to 01010$, and $1101 \to 11011$. If we get 11010 on the other end of the channel, then we know that an error must have happened, because it contains an odd number of 1's. But we don't know *where* the error is, because both of the above two example bit sequences differ from this one in only one bit. The parity bit method always detects a single-bit error, but it can't detect two bits (because then the number of 1's remains even).

In this section, we introduce an error-detection method that is more powerful than the parity bit: the **Cyclic Redundancy Check (CRC)**.

CRC treats the bit sequences as polynomials, where each coefficient is a bit (0 or 1). For example, the bit sequence 101011 corresponds to the polynomial $x^5 + x^3 + x + 1$ (the last bit is the constant term, the previous one is the linear term, and so on). The polynomials are in $\mathbb{Z}_2[x]$, i.e. all calculations are done modulo 2, e.g. $1 + 1 = 0,\ 0 - 1 = 1$ etc. As a consequence of this, for all polynomials, $p + p = 0$ (e.g. $(x^2 + 1) + (x^2 + 1) = (1 + 1)x^2 + (1 + 1) = 0$), so $p = -p$, therefore addition is the same as subtraction: $p - q = p + (-q) = p + q$. This addition/subtraction is also the same as the bitwise "exclusive or" (XOR, $\oplus$) operation (e.g. $1 + 1 = 0 \ \longrightarrow$ "true" $\oplus$ "true" = "false").

The main operation of CRC is polynomial division with remainder in $\mathbb{Z}_2[x]$. The divisor is a fixed polynomial $g \in \mathbb{Z}_2[x]$, which is called the **generator polynomial** of CRC. If $n := \deg g$ (i.e. g has $n+1$ bits), then the algorithm is called an ***n*-bit CRC** (notation: CRC-n), since the remainder by g fits into n bits (as its degree is $\leq n-1$).

CRC message sending:

Input: $m \in \mathbb{Z}_2[x]$, the message to be transmitted *(it can be treated as a bit sequence)*
Calculate the polynomial mx^n *(i.e. extend m by n zeros)*
$c := mx^n \bmod g$ *(where c is the **check value**)*
Sent **codeword**: $mx^n + c$ *(i.e. extend m by the n-bit check value c)*

CRC checking (method 1):

Input: $mx^n + c \in \mathbb{Z}_2[x]$, the received codeword (see above)
Split the codeword into m and c *(i.e. separate the last n bits)*
$c' := mx^n \bmod g$
if $c' \neq c$ **then** $\to$ *error*

CRC checking (method 2):

Input: $mx^n + c \in \mathbb{Z}_2[x]$, the received codeword (see above)
$e := (mx^n + c) \bmod g$
if $e \neq 0$ **then** $\to$ *error*

The two checking methods are equivalent, because if the remainder of mx^n is c, i.e. $mx^n = qg + c$, then we can rearrange this to $mx^n - c = qg$; but in $\mathbb{Z}_2[x]$, subtraction is the same as addition, so $mx^n - c = mx^n + c$, therefore $g \mid mx^n + c$, i.e. $mx^n + c$ indeed gives the remainder 0.

Example. Let the generator be $g = x^4 + x + 1$, or as a bit sequence, $g = 10011$ (so $n = 4$), and let the message be $m = 11000101$. For transmission, calculate the polynomial $mx^n = 11000101|0000$ (i.e. append four zeros to m), and divide it by g (see below on the left). The remainder is $c = 110$, extend it to n bits: $c = 0110$, and append it to the message: $mx^n + c = 11000101|0110$, to get the transmitted codeword. The receiver checks this value by dividing it by g (see below on the right),

and since the remainder is 0, the message is accepted as valid. (Again: these are not numbers in binary, but polynomials in $\mathbb{Z}_2[x]$, using the long division as described above, except that the x's are omitted for simplicity. Therefore, subtraction is actually XOR.)

Sending ($mx^n \bmod g$):

$$
\begin{array}{r}
11000101|0000 \bmod 10011 \\
\underline{-10011} \\
10111 \\
\underline{-10011} \\
10001 \\
\underline{-10011} \\
10000 \\
\underline{-10011} \\
110
\end{array}
$$

Checking ($mx^n + c \bmod g$):

$$
\begin{array}{r}
11000101|0110 \bmod 10011 \\
\underline{-10011} \\
10111 \\
\underline{-10011} \\
10001 \\
\underline{-10011} \\
10011 \\
\underline{-10011} \\
00
\end{array}
$$

The advantage of CRC is that this polynomial long division can be done very efficiently on computers, because it only needs bit manipulations, which computers are already very good at.

Unfortunately, it is impossible to detect *all* errors, so the check will inevitably accept some corrupted codewords too. The goal is to make this as rare as possible. Mathematically, an error means that instead of the transmitted codeword $mx^n + c$, the receiver gets $mx^n + c + e$, where e is the **error polynomial**, which has 1 for each corrupted bit (remember how XOR'ing 1 to a bit exactly flips that bit). If an error is undetected, it is because $g \mid mx^n + c + e$ (i.e. the remainder is 0), therefore $g \mid e$ (since for the correct codeword, $g \mid mx^n + c$). A good g therefore does not divide e for the most frequently occuring error patterns.

Here are some aspects of choosing a generator polynomial (g):

1. Its degree (n) equals the number of check bits, so the bigger it is, the smaller the error probability can be (a random bit sequence is accepted with $1/2^n$ probability).
2. The constant term (last bit) of g should be 1, because:
   - If it were 0, then $x \mid g$, so the last (few) bits of $c = mx^n \bmod g$ were always 0, which would reduce the number of useful bits of c.
   - Then all one-bit errors are detected (because then $e = x^k$, which would only be divisible by $g = x^l$).
   - Then all such errors are detected where the failed bits are within a single n-bit window.
3. If $x + 1 \mid g$ (or equivalently, g has an even number of 1 bits), then all errors with an odd number of bits are detected (because the multiples of g also have an even number of 1 bits).

Some frequently used CRC generator polynomials in practice:

- CRC-32: $g = 1\,00000100\,11000001\,00011101\,10110111$ (e.g. Ethernet, Wi-Fi, PNG);
- CRC-16: $g = 1\,10000000\,00000101$ (e.g. USB);
- CRC-1: $g = 11$ $(= x + 1)$ – equivalent to the parity bit (e.g. HDDs and many other hardware).

7 Author

This document was written by Uray M. János, partly using the following sources:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa (Hungarian)

[2] https://en.wikipedia.org/wiki/Polynomial_long_division

[3] https://en.wikipedia.org/wiki/Synthetic_division

[4] https://en.wikipedia.org/wiki/Cyclic_redundancy_check

[5] W.H. Press et all (2007): Numerical Recipes: The Art of Scientific Computing, Third Edition; chapter 22.4: Cyclic Redundancy and Other Checksums

Errors and suggestions can be reported here: uray.janos@inf.elte.hu.