Idempotents and Nilpotents Illustrated Using Modular Numbers

Kurt Nalty

July 21, 2015

Abstract

Modular arithmetic is a historically valuable playground for idempotents and nilpotents. This note provides C code showing step by step the process of going from a factored number to listing and demonstrating the idempotents and nilpotents associated with that modular space.

Modular Numbers

Modular numbers are the set of integers which roll-over to zero at a maximum count. Modular numbers are seen in computer architectures, where unsigned bytes span 0-255, then roll-over to zero, unsigned words span 0-65535, then roll-over, and so on. Modular numbers are not restricted to binary powers of two, but can be any value. An example of mod 5 counting is the set $0, 1, 2, 3, 4$, which then rolls over to zero.

In common practice, modular numbers are implemented using division by the modulus, keeping the remainder and discarding the quotient. Notice the inherent loss of information in modular numbers. Similar loss of information also occurs with modular behavior associated with rotations, where we keep the phase mod 2π, and lose the number of rotations (winding number).

In C (and C++), the mod operator is the percentage sign.

```
b = a % 5;     // b is (a mod 5)
```

In C, we have the detail that if the incoming number is negative, the returned remainder will be negative. Conventionally, I want all positive values for mod results, so I test the returned result when there is a chance of negative values, and add the modulus to negative remainders when necessary, thus keeping results positive. The following code demonstrates this behavior.

```
a = -17;
b = a % 5;              // b is (a mod 5) = -2
printf("a = %d,  b = %d \n",a,b);
if(b < 0) b += 5;     //convert to positive format
printf("a = %d,  b = %d \n",a,b);
```

Addition and Subtraction with Modular Numbers

Addition and subtraction are well defined with modulo numbers. Modulo 18 addition of two integers is illustrated by

```
for (i=0;i<18;i++) {
    printf("%3d : ",i);
    for (j=0;j<18;j++) {
        k = (i+j) % 18;
        printf("%3d ",k);
    }
    printf("\n");
}
```

The modulo 18 addition table is

```
Addition
  + |   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  ------------------------------------------------------------------------
  0 :   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  1 :   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17   0
  2 :   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17   0   1
  3 :   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17   0   1   2
  4 :   4   5   6   7   8   9  10  11  12  13  14  15  16  17   0   1   2   3
  5 :   5   6   7   8   9  10  11  12  13  14  15  16  17   0   1   2   3   4
  6 :   6   7   8   9  10  11  12  13  14  15  16  17   0   1   2   3   4   5
  7 :   7   8   9  10  11  12  13  14  15  16  17   0   1   2   3   4   5   6
  8 :   8   9  10  11  12  13  14  15  16  17   0   1   2   3   4   5   6   7
  9 :   9  10  11  12  13  14  15  16  17   0   1   2   3   4   5   6   7   8
 10 :  10  11  12  13  14  15  16  17   0   1   2   3   4   5   6   7   8   9
 11 :  11  12  13  14  15  16  17   0   1   2   3   4   5   6   7   8   9  10
 12 :  12  13  14  15  16  17   0   1   2   3   4   5   6   7   8   9  10  11
 13 :  13  14  15  16  17   0   1   2   3   4   5   6   7   8   9  10  11  12
 14 :  14  15  16  17   0   1   2   3   4   5   6   7   8   9  10  11  12  13
 15 :  15  16  17   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 16 :  16  17   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 17 :  17   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
```

We notice that each row contains every mod 18 number. The inverse of addition (being subtraction) is guaranteed to work.

Looking at subtraction, a code snippet to do the subtraction with negative range correction is

```
for (i=0;i<18;i++) {
    printf("%3d : ",i);
    for (j=0;j<18;j++) {
        k = (i-j) % 18;
        if (k<0) k += 18;
        printf("%3d ",k);
    }
    printf("\n");
}
```

The modulo 18 subtraction table is

Subtraction

-	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0 :	0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1 :	1	0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
2 :	2	1	0	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
3 :	3	2	1	0	17	16	15	14	13	12	11	10	9	8	7	6	5	4
4 :	4	3	2	1	0	17	16	15	14	13	12	11	10	9	8	7	6	5
5 :	5	4	3	2	1	0	17	16	15	14	13	12	11	10	9	8	7	6
6 :	6	5	4	3	2	1	0	17	16	15	14	13	12	11	10	9	8	7
7 :	7	6	5	4	3	2	1	0	17	16	15	14	13	12	11	10	9	8
8 :	8	7	6	5	4	3	2	1	0	17	16	15	14	13	12	11	10	9
9 :	9	8	7	6	5	4	3	2	1	0	17	16	15	14	13	12	11	10
10 :	10	9	8	7	6	5	4	3	2	1	0	17	16	15	14	13	12	11
11 :	11	10	9	8	7	6	5	4	3	2	1	0	17	16	15	14	13	12
12 :	12	11	10	9	8	7	6	5	4	3	2	1	0	17	16	15	14	13
13 :	13	12	11	10	9	8	7	6	5	4	3	2	1	0	17	16	15	14
14 :	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	17	16	15
15 :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	17	16
16 :	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	17
17 :	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Again, we note that each row contains all characters for this modular scheme. An inverse function (addition) is guaranteed to exist and work.

Multiplication with Modular Numbers

A code snippet to perform modular multiplication is

```
for (i=0;i<18;i++) {
    printf("%3d : ",i);
    for (j=0;j<18;j++) {
        k = (i*j) % 18;
        printf("%3d ",k);
    }
    printf("\n");
}
```

The resulting multiplication table is

```
Multiplication
  * |   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  ------------------------------------------------------------------------------
  0 :   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  1 :   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  2 :   0   2   4   6   8  10  12  14  16   0   2   4   6   8  10  12  14  16
  3 :   0   3   6   9  12  15   0   3   6   9  12  15   0   3   6   9  12  15
  4 :   0   4   8  12  16   2   6  10  14   0   4   8  12  16   2   6  10  14
  5 :   0   5  10  15   2   7  12  17   4   9  14   1   6  11  16   3   8  13
  6 :   0   6  12   0   6  12   0   6  12   0   6  12   0   6  12   0   6  12
  7 :   0   7  14   3  10  17   6  13   2   9  16   5  12   1   8  15   4  11
  8 :   0   8  16   6  14   4  12   2  10   0   8  16   6  14   4  12   2  10
  9 :   0   9   0   9   0   9   0   9   0   9   0   9   0   9   0   9   0   9
 10 :   0  10   2  12   4  14   6  16   8   0  10   2  12   4  14   6  16   8
 11 :   0  11   4  15   8   1  12   5  16   9   2  13   6  17  10   3  14   7
 12 :   0  12   6   0  12   6   0  12   6   0  12   6   0  12   6   0  12   6
 13 :   0  13   8   3  16  11   6   1  14   9   4  17  12   7   2  15  10   5
 14 :   0  14  10   6   2  16  12   8   4   0  14  10   6   2  16  12   8   4
 15 :   0  15  12   9   6   3   0  15  12   9   6   3   0  15  12   9   6   3
 16 :   0  16  14  12  10   8   6   4   2   0  16  14  12  10   8   6   4   2
 17 :   0  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1
```

We see that the rows which are relatively prime to 18, being 1, 5, 7, 11, 13 and 17, contain the full character set with permutated elements. However, rows which share common factors with 18 have a reduced character set, and repeating subsequences.

Division with Modular Numbers

Division is not guaranteed to work for most elements in this algebra. To do division, we use the multiplication table in reverse. As an example, let's find 3/11. We use the equation $3/11 = x$, transpose to find $11x = 3$, and examine the row multiplying by eleven and see $11 * 15 = 3 \pmod{18}$. We thus have $3/11 = 15 \pmod{18}$. In a similar fashion, we find $11/7 = 17 \pmod{18}$.

The important case of reciprocals necessarily follows the same process, with reciprocals not guaranteed to exist. $(1/n)$ only exists for the n relatively prime to modulus elements. For example, $1/5 = 11 \bmod 18$, but $1/4$ does not exist.

Idempotents

Idempotents are defined by $P^2 = P$. In regular arithmetic, 0 and 1 are idempotents. Looking along the descending diagonal of the multiplication table, we find additional idempotents in mod 18 multiplication, being 9 and 10. We point out that the product of 9*10 is $90 = 0 \mod 18$.

Nilpotents

Nilpotents are defined by $N^n = 0$. The simplest nilpotents are of order 2, and defined by $N^2 = 0$. Looking along the descending diagonal of the multiplication table, we see simple nilpotents of 0, 6 and 12.

Relatively Prime Numbers

In theory, we could print out multiplication tables of our modular number system of interest, and find idempotents and nilpotents by inspection. This is impractical, of course, for larger numbers, and we turn to Euclid's Extended greatest common denominator algorithm to find idempotents. Two references for this approach are Garret Sobczyk [1] and wikipedia [2].

Before dealing with Euclid's algorithm, we should look at the modulo arithmetic properties of a number, and its relatively prime factors. We can express any number as a set of relative prime factors. For example, $18 = 2 * 3^2$ has relatively prime factors of 2 and 9. We will find the number of idempotents will equal the number of unique prime factors, and the presense of powers of a prime beyond one will lead to nilpotents associated that factor.

Look at the set of numbers given by h divided by individual relatively prime factors. Using $h = 18$ as our example, we have two numbers, $18/9 = 2$ and $18/2 = 9$. We see that each number is missing a factor of h. We note then, that the product of two different numbers, each missing different factors of h, will now have a complete factor of h times another integer as a result. We illustrate this concept with a generic three factor number below.

$$\begin{aligned}
h &= a^k c^m d^n \\
b_1 &= \frac{h}{a^k} \\
b_2 &= \frac{h}{c^m} \\
b_3 &= \frac{h}{d^n} \\
b_1 * b_2 &= \frac{h}{a^k} * \frac{h}{c^m} = h * \left(\frac{h}{a^k c^m}\right) = h * \left(\frac{a^k c^m d^n}{a^k c^m}\right) \\
&= h * d^n = 0 \mod h
\end{aligned}$$

Each of these b_n terms acts like an independent, orthogonal basis in a vector space. When we form a linear combination of these terms, and raise that combination to a power, all the cross terms will drop out, leaving a very simple expression. (The symbol $\stackrel{h}{=}$ implies modulo h arithmetic on both sides.)

$$(b_1 f_1 + b_2 f_2 + \cdots + b_n f_n)^r \quad \stackrel{h}{=} \quad (b_1 f_1)^r + (b_2 f_2)^r + \cdots + (b_n f_n)^r$$

We will now use this knowledge of the power law nature for our partial fractions with the specific linear relationship found in Euclid's extended algorithm to find the idempotents inside our modulo arithmetic family.

Euclid's Extended Algorithm

Euclid's algorithm is intended to find the greatest common denominator between two numbers. It is easily extended to find the greatest common denominator among a set of numbers. By keeping track of the quotients and remainders formed along the way, we can reconstitute a linear relationship between these numbers and their greatest common denominator.

Our first task is to find the greatest common denominator of two numbers as a linear combination of these two numbers. (This is Euclid's algorithm.) In the routine GCD2, (based upon python code at WikiCode [2]), my incoming pair of numbers are i and j, with the returned scale factors of the linear relationship at out_x and out_y, and the actual GCD returned as the function result.

We begin by arranging our two incoming numbers by size. We divide the larger by the smaller, getting a quotient and remainder. If the remainer is zero, then our GCD is the smaller number from the division. Otherwise, we replace the larger number with the smaller, the smaller with the remainder, and continue dividing.

At the end of the first division, we have Larger/Smaller = (Quotient, Remainder), or Remainder = Larger - (Quotient*Smaller). This is our linear relationship between the two incoming numbers and their GCD candidate at this point. Each time we do a division, we can update our candidate relationship. This is done with variables x, y, m, n, u and v in the code below.

```
long GCD2(long i, long j, long *out_x, long *out_y)
{
    long quotient, remainder, u, v, m, n, gcd, x, y;
    long larger, smaller;

    if(i > j) {larger = i;  smaller = j;    }  else {larger = j;  smaller = i; }

    x = 0; y = 1; u = 1; v = 0; n = 0;

    if (smaller == 1) {x = 1 - larger; y = 1;} else
    do {
        quotient = larger/smaller;
        remainder = larger % smaller;
```

```
        larger = smaller;      // update the terms for
        smaller = remainder;  // the next division

        m = x - u*quotient;
        n = y - v*quotient;

        x = u;
        y = v;
        u = m;
        v = n;

    } while (remainder != 0);

    if (smaller != 1) gcd = larger; else gcd = 1;

    *out_x = x;
    *out_y = y;
    if (i > j) {*out_x = y;  *out_y = x;}
    return gcd;
}
```

Let's walk through this process by hand using our number $18 = 3^2 * 2$. We create our partial fractions, just as before, getting $f_1 = 2$ and $f_2 = 9$. As these numbers are relatively prime, we know beforehand that their GCD will be one.

$$\begin{aligned} 9/2 &= 4 \text{ r } 1 \\ 4/1 &= 4 \text{ r } 0 \quad \text{done!} \\ 1 * 9 - 4 * 2 &= 1 \qquad \text{This is our desired relationship.} \end{aligned}$$

We notice that the 9 in the first times the 2 in the second term will make all cross terms have a factor of 18, leading to zero in mod 18.

We can apply our mod operator to both sides of our equations, and to each individual term to specialize our equation above for modulo math.

$$\begin{aligned} 1 * 9 - 4 * 2 &= 1 \\ 1 * 9 - 4 * 2 &\overset{18}{=} 1 \\ -4 &\overset{18}{=} 14 \\ 1 * 9 + 14 * 2 &\overset{18}{=} 1 \end{aligned}$$

How about the case where we have more than one pair of relatively prime products? We simply start with our first pair. Their GCD will not be one. We use this GCD with the third number to make another pair, keeping track of the return factors, and continue in this fashion until we get a CGD of 1. This process is illustrated in the code snippet below. The partial fraction terms are in Basis[n], while the factors for the linear relationship will be placed in Factors[n].

```
//  count the number of input elements
```

```
    n = 0;
    while ((Basis[n] != 0) && (n<10)) n++; // assumes zero terminated array.
    n--; // number of adjacent pairs is one less than number of entries
    printf("%ld pairs \n",n);

//  process initial pair. This could be absorbed into the loop by setting Factors[0] = 1
//  and initial gcd to Basis[0]

    gcd = GCD2(Basis[0], Basis[1], &x, &y);
    Factors[0] = x;
    Factors[1] = y;

//  finish the other pairs

    for (i=1;i<n;i++) {
        gcd = GCD2(gcd, Basis[i+1], &x, &y);
        for (j=0;j<(i+1);j++) Factors[j] *= x;
        Factors[i+1] = y;
    }
```

Idempotents via Euclid

We now identify our idempotents using the extended Euclid algorithm from above. In symbolic form, we have the linear relationship

$$b_1 f_1 + b_2 f_2 + \cdots + b_n f_n \quad \stackrel{h}{=} \quad 1$$

where b_i are the Basis[i] partial fractions from above, while f_i are the Factors[i]. We now square this expression, and apply our knowledge of the annihilating cross terms.

$$\begin{aligned} (b_1 f_1 + b_2 f_2 + \cdots + b_n f_n)^2 &\stackrel{h}{=} 1 \\ (b_1 f_1)^2 + (b_2 f_2)^2 + \cdots + (b_n f_n)^2 &\stackrel{h}{=} 1 \end{aligned}$$

We now subtract the original equation from its square.

$$\begin{aligned} \left[(b_1 f_1)^2 + (b_2 f_2)^2 + \cdots + (b_n f_n)^2\right] - \left[b_1 f_1 + b_2 f_2 + \cdots + b_n f_n\right] &\stackrel{h}{=} 1 - 1 = 0 \\ \left[(b_1 f_1)^2 - b_1 f_1\right] + \left[(b_2 f_2)^2 - b_2 f_2\right] + \cdots + \left[(b_n f_n)^2 - b_n f_n\right] &\stackrel{h}{=} 0 \end{aligned}$$

We return now to our example of a generic, three term product for h.

$$\begin{aligned} h &= a^k c^m d^n \\ b_1 &= \frac{h}{a^k} = c^m d^n \\ b_2 &= \frac{h}{c^m} = a^k d^n \\ b_3 &= \frac{h}{d^n} = a^k c^m \end{aligned}$$

We see that the first term b_1 is lacking the factor a, but everyone else has this term. Consequently, if I take mod a on my equation above, I get

$$\begin{aligned} \Big[(b_1 f_1)^2 - b_1 f_1\Big] + \Big[(b_2 f_2)^2 - b_2 f_2\Big] + \cdots + \Big[(b_n f_n)^2 - b_n f_n\Big] &\stackrel{h}{=} 0 \\ \Big[(b_1 f_1)^2 - b_1 f_1\Big] &\stackrel{a}{=} 0 \end{aligned}$$

In a similar fashion,

$$\begin{aligned} \Big[(b_1 f_1)^2 - b_1 f_1\Big] &\stackrel{a}{=} 0 \\ \Big[(b_2 f_2)^2 - b_2 f_2\Big] &\stackrel{c}{=} 0 \\ \Big[(b_3 f_3)^2 - b_3 f_3\Big] &\stackrel{d}{=} 0 \end{aligned}$$

As the defining characteristic of an idempotent is $P^2 = P \rightarrow P^2 - P = 0$, we have the pleasant result that the $b_i f_i$ terms are the idempotents for our modulo system.

Nilpotents via Euclid

Nilpotents of order n are defined by $N^n = 0$. When the relatively prime factors of h involve a power, there will be an associated nilpotent with that term's idempotent. As an example, let

$$\begin{aligned} h &= a^3 c^4 d^2 \\ b_1 &= \frac{h}{a^3} = c^4 d^2 \\ b_2 &= \frac{h}{c^4} = a^3 d^2 \\ b_3 &= \frac{h}{d^2} = a^3 c^4 \end{aligned}$$

From the extended Euclid algorithm, we will have factors f_1, f_2 and f_3 which will give us idempotents $P_1 = b_1 f_1$, $P_2 = b_2 f_2$ and $P_3 = b_3 f_3$. For programming convenience, I also define an array of radices, with $r_1 = a$, $r_2 = c$, and $r_3 = d$.

We define our nilpotents as $P_i r_i$, and now see the effect of raising these nilpotents to ascending powers.

$$\begin{aligned} N_1 &\stackrel{h}{=} P_1 r_1 = f_1 b_1 r_1 = f_1 c^4 d^2 a \neq 0 \mod h \\ N_1^2 &\stackrel{h}{=} (P_1 r_1)^2 = P_1 r_1^2 = f_1 b_1 r_1 = f_1 c^4 d^2 a^2 \neq 0 \mod h \\ N_1^3 &\stackrel{h}{=} (P_1 r_1)^3 = P_1 r_1^3 = f_1 b_1 r_1 = f_1 c^4 d^2 a^3 = 0 \mod h \end{aligned}$$

We see that the projection operators (idempotents) stay at unity power during the higher order product powers. When the product is raised to the corresponding power of the prime factor, we then get a zero mod h.

We now return to our brave number 18, and check these formulas against his tabulated behavior.

$$\begin{aligned} 18 &= 3^2 * 2 \\ 1 * 9 - 4 * 2 &= 1 \\ 1 * 9 - 4 * 2 &\stackrel{18}{=} 1 \\ -4 &\stackrel{18}{=} 14 \\ 1 * 9 + 14 * 2 &\stackrel{18}{=} 1 \\ 28 &\stackrel{18}{=} 10 \\ 9 + 10 &\stackrel{18}{=} 1 \end{aligned}$$

and so we have idempotents of 9 and 10, as seen in the tables at the start of this article.

In our list of relative prime factors, we had $9 = 3^2$, so we have a nilpotent of order 2 associated with $b_1 = 18/3^2 = 2$. Let's find this idempotent.

The associated idempotent is $14 * 2 = 28 = 10 \mod 18$. We multiply this idempotent by our prime factor of 3, and find the associated nilpotent of order 2 is $3 * 10 = 30 = 12 \mod 18$. We check our multiplication $12 * 12 = 144 = 0 \mod 18$.

Checking our previous tabulation, we found simple nilpotents at 12, 6 and 0. So now we ask, where did the six come from? To find our answer, we note that any number times a nilpotent also yields a nilpotent. Using a generic example of order 2, we have

$$\begin{aligned} N * N &\stackrel{h}{=} 0 \\ (k * N) * (k * N) &\stackrel{h}{=} k * k * N * N = k * k * 0 = 0 \end{aligned}$$

This means that all along row 12 in our example using $h = 18$, each of these terms is also an idempotent of order 2. As we look at this row, we have the repeating pattern 0 12 6. We see that $12 * 2 = 24 \stackrel{18}{=} 6$. And, in hindsight, we say of course $6 * 6 = 12 * 2 * 12 * 2 \stackrel{18}{=} 0$. All is fun with modular arithmetic.

Spectral Basis

The most common spectral basis in engineering is the use of sinusoids in to represent periodic signals. Each sinusoid at a fixed frequency is orthogonal to sinusoids at other frequencies, and the time history of a complicated peridoic waveform can be represented as a weighted sum of frequencies (with associated phase shifts).

In the case of modular numbers, each modular number can be represented as a weighted sum of that systems idempotents. In the related system of modular polynomials, each polynomial can be represented as a weighted sum of the idempotents and nilpotents. As a consequence, we commonly call out the spectral basis of a system as the collection of idempotents and nilpotents, even though, in our specific case of modular numbers, idempotents alone suffice.

So, how we represent a modular number in terms of its idempotents? The answer is to multiply the sum of the idempotents by number of interest, and then reduce each coefficient in the sum to it's minimal value.

The extended Euclid algorithm provides the values for the idempotents in a modular system, and the sum of all these idempotents sums to one.

$$P_1 + P_2 + \cdots + P_n \stackrel{h}{=} 1$$

We multiply this equation by our number of interest, say C.

$$C * P_1 + C * P_2 + \cdots + C * P_n \stackrel{h}{=} C$$

Now each idempotent P_i includes a term h/f_i, where f_i is one of the relatively prime factors of h. Consequently, multiplying this term by f_i results in a zero, mod h. This in turn means that each coefficient can be reduced mod f_i, and the sum remain unchanged.

$$(C \mod f_1) * P_1 + (C \mod f_2) * P_2 + \cdots + (C \mod f_n) * P_n \stackrel{h}{=} C$$

or equivalently

$$C \stackrel{h}{=} (C \mod f_1) * P_1 + (C \mod f_2) * P_2 + \cdots + (C \mod f_n) * P_n$$

Each term $(C \mod f_n)$ is a weight applied to a spectral basis P_i.

Let's do an example. Represent 437 in the spectral basis associated with mod 900.

We factor $900 = 2^2 * 3^2 * 5^2$. Our relatively prime factors f_n and basis b_n

are

$$\begin{aligned} f_1 &= 2^2 = 4 \\ f_2 &= 3^2 = 9 \\ f_3 &= 5^2 = 25 \\ b_1 &= 900/2^2 = 225 \\ b_2 &= 900/3^2 = 100 \\ b_3 &= 900/5^2 = 36 \end{aligned}$$

Our idempotents and sum are

$$\begin{aligned} P_1 &= 225 \\ P_2 &= 100 \\ P_3 &= 576 \\ P_1 + P_2 + P_3 &= 901 \overset{900}{=} 1 \end{aligned}$$

Our weights for 437 are $(C \mod f_n)$

$$\begin{aligned} 437 \mod 4 &= 1 \\ 437 \mod 9 &= 5 \\ 437 \mod 25 &= 12 \end{aligned}$$

Our resulting representation for 437 in the spectral basis for Z_{900} is

$$1 * 225 + 5 * 100 + 12 * 576 = 7637 \overset{900}{=} 437$$

Application

Why would anyone want to breakup a perfectly good number into these individual, incomplete parts?

$$C \overset{h}{=} (C \mod f_1) * P_1 + (C \mod f_2) * P_2 + \cdots + (C \mod f_n) * P_n$$

One scenario is the storage of a decryption key C among several parties. Only when all parties provide their partial information $(C \mod f_n)$ can the key C be recovered.

Another scenario is where a document is encoded using key C. Information about the encoded key is sent, along with the cryptotext, via $(C \mod f_1)$. In essense, the public key is f_1. The receiving party uses their privileged knowledge of P_1, P_2 and f_2 to recover the cryptokey C, and thence the document material.

References

[1] Garret Sobczyk, *New Foundations in Mathematics*, Birkhauser, ISBN 978-0-8176-8384-9

[2] Wikipedia, *Extended Euler Algorithm* `http://en.wikipedia.org/wiki/Extended_Euler_algorithm`