Sum operator

Some very long time ago, I had a problem with understanding the notion of mathematical expectation. (Those who accidentally have found my PhD dissertation, please, don't laugh!) This painful situation has stopped when somebody has shown me, how ordinary calculation of mathematical expectation looked in code.

Here, I will take the same way that once worked for me. Let's simply consider a sum operator, so as not to return to this question any more. Finally, sum is a fundamental operator for linear algebra, mathematical statistics and signal processing in application to digital data, i.e. in what we are going to address hereafter.

Simply a sum

Whenever you encounter in mathematical documents a capital Greek letter sigma, consider it designates a sum over values of a generic expression to the right of the summation symbol:

$$result = \sum_{i=0}^{N-1} expression(\mathrm{i})$$

⇔

```
for (int i = 0; i < N; i++)
    result += Expression(i);
```

Figure 1. Sum operator in general

Beneath the sum symbol, it is usual to specify the summation index and its initial value, while the upper limit for that is typically indicated above the sum. It is also possible, but conceivably less common to specify the index and its iteration values below the sum symbol:

$$\sum_{i=0}^{N-1} expression(\mathrm{i}) \quad \Leftrightarrow \quad \sum_{i=0,1\ldots \mathrm{N}-1} expression(\mathrm{i})$$

Figure 2. Sum index limits: the indicated index limits always include the upper limit!

In any case, the indicated index limits **always include the upper limit**.

Here, the notion of "generic expression" stands for an expression that interprets the sum (iteration) index values *i* into values to be summed up. Summation index in the sum operator plays exactly the same role as it does in the for-loop.

The most frequent case of index expression is indexing array elements:

$$result = \sum_{i=0}^{N-1} a_i$$

⇔

```
for (int i = 0; i < N; ++i)
    result += a[i];
```

Figure 3. Sum over array elements

Summation index can also be involved into evaluation, like, for instance, in the polynomials:

$$result = \sum_{i=0}^{N-1} a_i x^i \quad \Leftrightarrow \quad result = a_o + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}$$

$$result = \sum_{i=0}^{N-1} a_i x^i \quad \Leftrightarrow$$

```
power = 1;
for (int i = 0; i < N; ++i)
{
    result += a[i]*power;
    power *= x;
}
```

Figure 4. A sum operator for calculation of polynomials

Iteration over specific indexes (e.g. even indexes) can be specified in various ways in mathematical formulas as well as in code, for instance as below:

$$result = \sum_{i=0,2,4\ldots}^{N-1} a_i \quad \Leftrightarrow$$

```
for (int i = 0; i < N; i+=2)
    result += a[i];
```

$$result = \sum_{i=0}^{N-1} a_{2i} \quad \Leftrightarrow$$

```
for (int i = 0; i < N; ++i)
    result += a[2*i];
```

Figure 5. Sum over specific indexes

Multidimensional sum

A sum can be applied to a generic expression of more than one index. In course of mathematical manipulation, it might be sometimes convenient to write down such expressions with own sum symbol per summation index. In the final result, however, I would prefer having it in a short form:

$$result = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} expression(i,j)$$

$$result = \sum_{i,j=0}^{N-1,M-1} expression(i,j) \quad \Leftrightarrow$$

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        result += Expression(i,j);
```

Figure 6. 2-dimensional sum

Sum operators in C# LINQ expression

In the very beginning of this section, I would like to emphasize that this article is concentrated on understanding the mathematical formulas; performance issues are out of its scope.

Use of LINQ expressions instead of *for*-loops could often be a good choice, once you implement it in C#. Here, a criterion of readability should be considered.

For practical evaluation of readability, I would suggest checking, **how easily each specific LINQ expression can be reverse-engineered into a mathematical formula**.

Perception of readability is individual. It depends, amid other factors, upon your experience with LINQ. It is also significant, to what extend this or another specific sum expression and its LINQ counterpart are typical for your simulation tasks.

Another practical rule I use for myself is minimization of potential error sources in advance. So, in this specific case, I'd prefer reduce my mental effort about implementation details: **if a LINQ expression requires less thinking, I take it**, otherwise I don't.

Here below is an illustration of different sum expressions and their LINQ pictures:

$$result = \sum_{i=0}^{N-1} a_i$$

⇔

```
result = a.Sum();
```

$$result = \sum_{i,j=0}^{N-1,M-1} a_{i,j}$$

⇔

```
result = a.Sum(r => r.Sum());
```

$$result = \sum_{i=0}^{N-1} ia_i$$

⇔

```
result = a.Select((e,i) => i*e).Sum();
```

$$result = \sum_{i,j=0}^{N-1,M-1} ija_{i,j}$$

⇔

```
result = a.Select((r,i)=> r.Select((c,j)=> i*j*c)
                           .Sum()).Sum();
```

Figure 7. LINQ expression for sum expression of different complexity

Here, everybody can estimate it on his own, where the readability redline is for him or herself.