Applications of Series: Two Examples

Students will have seen numerous instances of infinite series representing functions in Calculus II, and probably in subsequent courses (particularly in differential equations, and complex analysis). Here we supplement these by sampling a few examples in which power series arise, but where their critical role is *not* in representing functions.

1. Counting via Generating Functions

A general sequence $a_0, a_1, a_2, \ldots \in \mathbb{Q}$ is often best studied via its *generating function*

$$f(x) = \sum_{n=0}^{\infty} a_n x^n \in \mathbb{Q}[\![x]\!].$$

For the factorial sequence $a_n = n!$, the resulting generating function $f(x) = \sum_{n=1}^{\infty} n! x^n$ is not much of a function, as we remark below—so we should really call it a *generating series*; however the terminology of 'generating function' is by now standard, and we will not try to change it. The following sort of counting problem may be encountered in combinatorics courses:

(a) Let b_n denote the number of ways to divide a deck of n distinct cards into two nonempty piles, and then to permute (i.e. shuffle) each pile. Find a formula for b_n.

(b) Determine the generating function $g(x) = \sum_{n=0}^{\infty} b_n x^n$, giving the first few terms explicitly.

(c) Determine b_{52} explicitly.

For example with $n = 5$, we may first divide the deck into piles of size 1+4, 2+3, 3+2 or 4+1, giving

$$b_5 = 1{\times}24 + 2{\times}6 + 6{\times}2 + 24{\times}1 = 72.$$

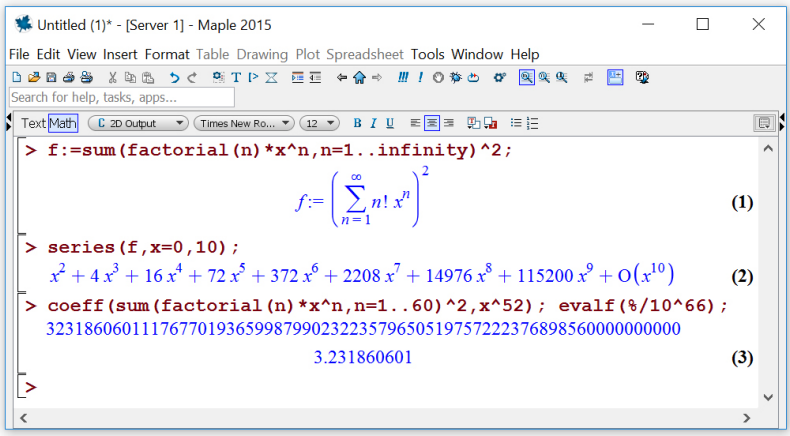
Generalizing this argument gives the formula

$$b_n = \sum_{k=1}^{n-1} k!(n-k)!\,.$$

Although the value b_{52} can be obtained from such a sum, in practice it is easier to use the generating function approach. Standard arguments for such counting problems in Math 3700 give the generating function

$$g(x) = \sum_{n=0}^{\infty} b_n x^n \;=\; \left(\sum_{n=1}^{\infty} n! x^n\right)^2 \;=\; x^2 + 4x^3 + 16x^4 + 72x^5 + 372x^6 + \cdots.$$

(Note that $b_0 = b_1 = 0$ since there is no way to obtain two nonempty piles with fewer than 2 cards.) The inner series counts ways to permute (i.e. shuffle) a single nonempty deck of n cards; and squaring represents the action of first dividing the deck into two piles. A typical Maple session for this problem is



from which we obtain the value

$$\begin{aligned} b_{52} \;&=\; 3231860601117677019365998799023223579650519757222376898560000000000 \\ &\approx\; 3.23 \times 10^{66}. \end{aligned}$$

Note that our Maple session defines a variable `f` rather than a function `f(x)`, as would typically be done using the Maple command `f:=x->sum` *etc.*. The latter would be useful if at any stage we required function values $f(1), f(2)$, etc.; however these values are undefined. Indeed the only input where $f(a)$ is defined, is at $a = 0$. Whatever use we have for $f(x)$, it is as a formal algebraic expression, *not* as a function.

2. Signal Processing

Our second example will use coefficients in $\mathbb{F}_2 = \{0, 1\}$ rather than in $\mathbb{Q}$ or $\mathbb{R}$. Given a power series

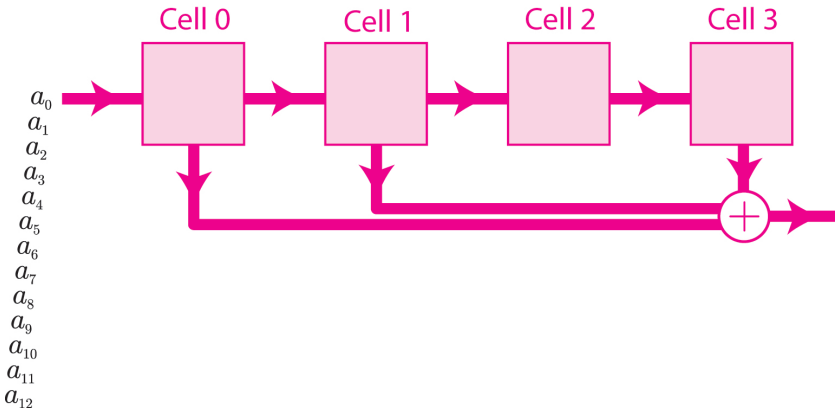
$$f(x) \;=\; \sum_{n=0}^{\infty} a_n x^n \;=\; a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots \;\in\; \mathbb{F}_2[\![x]\!],$$

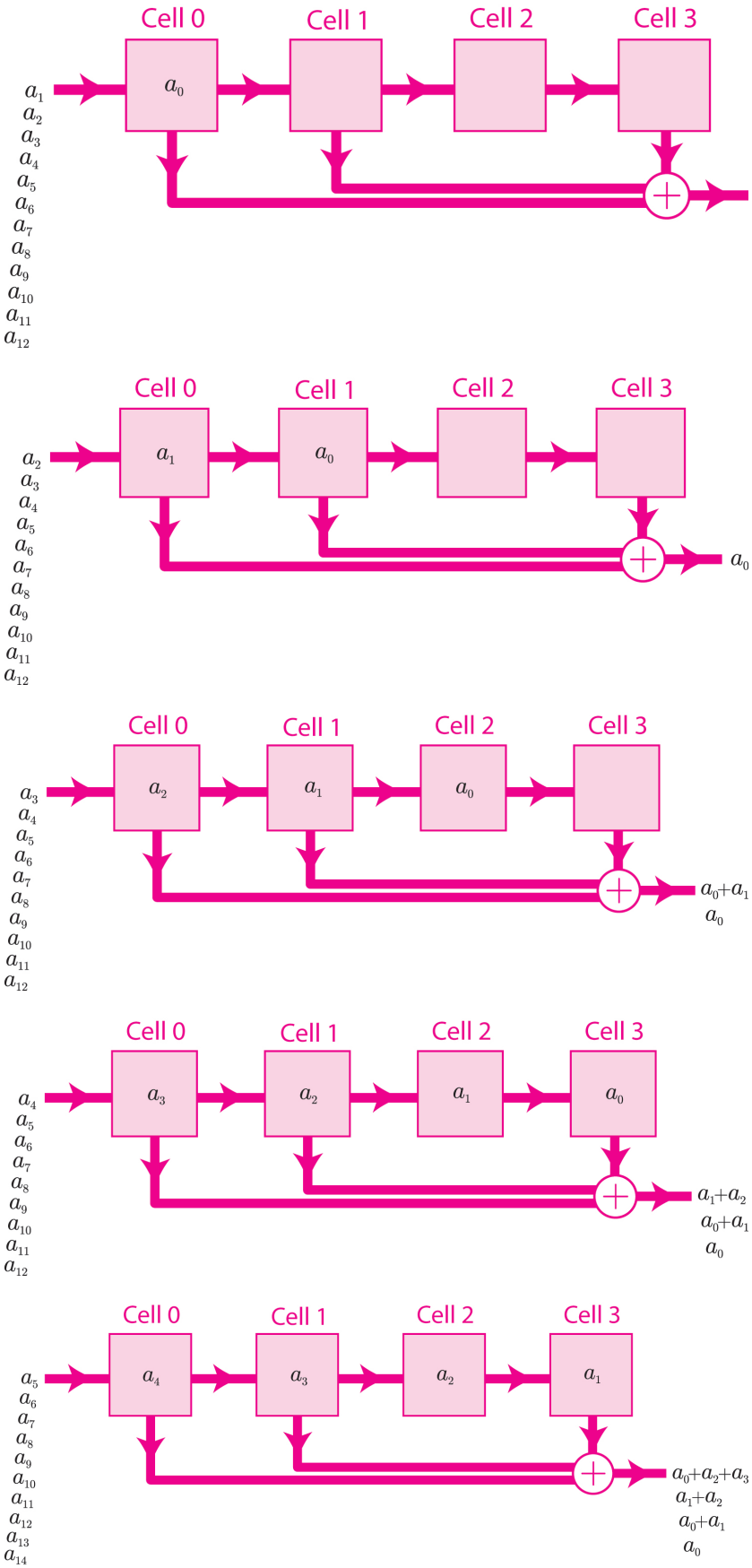
it is *immediately* apparent that we will not be viewing f as a function $\mathbb{F}_2 \to \mathbb{F}_2$. (Note that $f(1)$ is in general not a well-defined element of $\mathbb{F}_2$!) So it is not clear in what sense, if any, one may regard f as a function; fortunately, this question is not an issue for us.

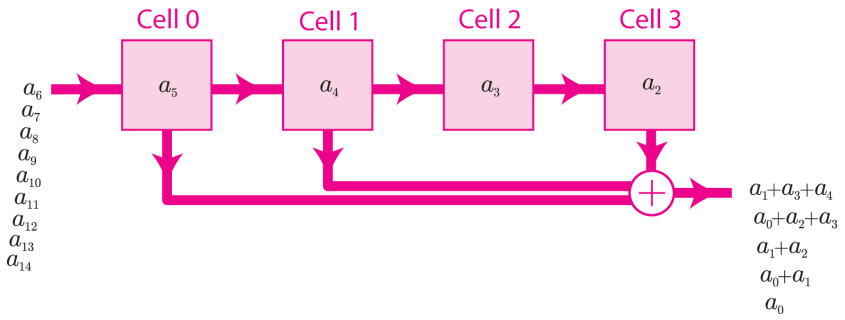
We shall regard the sequence $a_0, a_1, a_2, a_3, \ldots$ in $\mathbb{F}_2$ as a bitstream—a stream of bits of information representing, possibly, binary information from a digital camera, or a digital microphone, or a digital satellite transmission. The entire sequence is succinctly represented by the single algebraic object $f(x)$; and we are interested in transformations of this bitstream represented by algebraic operations of the form $f(x) \mapsto m(x)f(x)$ for a multiplier $m(x) \in \mathbb{F}_2[\![x]\!]$. The effect of such an operation is to scramble the original bitstream; and combinations of such operations are universally implemented in the practical error correction of streams of digital information.

Algebraic operations in $\mathbb{F}_2[\![x]\!]$ of the form we are considering could be easily implemented on a typical computer. However, to devote an entire computer (containing typically billions of transistors) to such a task would be a serious waste of hardware, since these tasks are easily accomplished using much simpler electronic circuits dedicated to the task (with correspondingly huge savings in cost and device sizes, not to mention huge improvements in response time and reliability). These circuits are known as *shift registers*; and we consider two varieties of these, arising from forward and backward shifts.

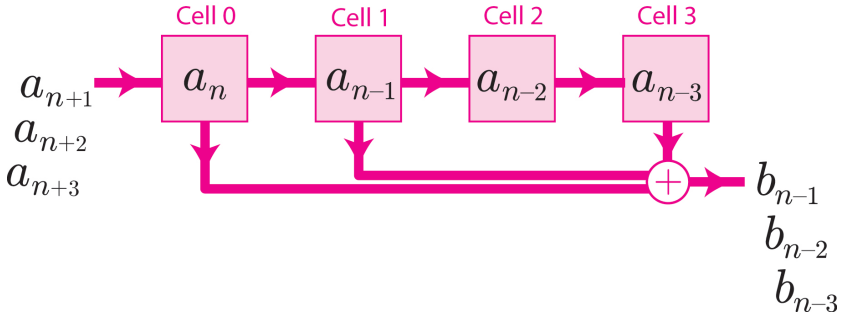
We consider first a simple example of a *forward shift register* as shown, with input $f(x) = \sum_n a_n x^n$ and output $\widehat{f}(x) = \sum_n b_n x^n$. The input bits $a_0, a_1, a_2, \ldots$ arrive at the left terminal, one bit per time step; and the bits progress through the register from left to right as shown, resulting in the output bitstream $b_0, b_1, b_2, \ldots$ on the right. Each memory cell holds a bit for one time step (as measured by a clock) and then passes its contents through the out-going arrow (or in the case of two output paths, a copy of the bit along each out-going path); and the cell then accepts the incoming bit arriving from the left. The node labelled '+' on the right performs addition (mod 2) and instantly returns the sum of incoming bits to the right as the next output bit. The state of the register at the first few successive time steps is as shown:




Cell 0
Cell 1
Cell 2
Cell 3
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}
a_0
Cell 0
Cell 1
Cell 2
Cell 3
a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}
a_1
a_0
a_0
Cell 0
Cell 1
Cell 2
Cell 3
a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}
a_2
a_1
a_0
a_0+a_1
a_0
Cell 0
Cell 1
Cell 2
Cell 3
a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}
a_3
a_2
a_1
a_0
a_1+a_2
a_0+a_1
a_0
Cell 0
Cell 1
Cell 2
Cell 3
a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14}
a_4
a_3
a_2
a_1
$a_0+a_2+a_3$
a_1+a_2
a_0+a_1
a_0



The state of the register at the instant before the n-th output bit b_n appears is

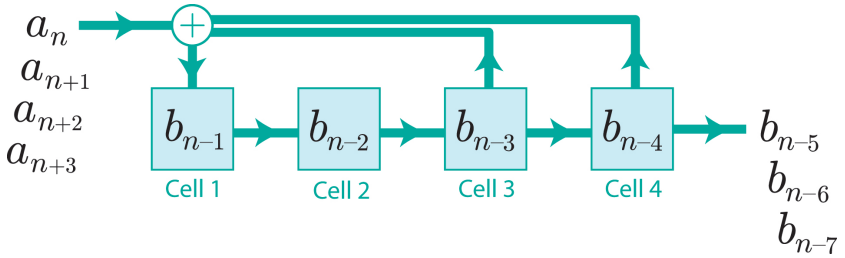


from which we see that $b_n = a_n + a_{n-1} + a_{n-3}$ (with the understanding that $a_{-1} = a_{-2} = \cdots = 0$) so the output stream is represented by

$$\begin{aligned}\widehat{f}(x) &= \sum_n b_n x^n = \sum_n (a_n + a_{n-1} + a_{n-3}) x^n \\ &= \sum_n a_n x^n + x \sum_n a_{n-1} x^{n-1} + x^3 \sum_n a_{n-3} x^{n-3} \\ &= (1 + x + x^3) f(x).\end{aligned}$$

Note that the exponents in the multiplier $1 + x + x^3$ are determined by the choice of cells (in this case, cells 0, 1 and 3) which feed into the '+' node.

A *backward shift register* works similarly but with bits jumping backwards via feedback loops, instead of forwards, as in the example:



Here is the state of such a register at one typical time step: In this case the n-th output bit will be given by $b_n = a_n + b_{n-3} + b_{n-4}$, which we solve to obtain $a_n = b_n + b_{n-3} + b_{n-4}$. The input stream is represented by

$$\begin{aligned} f(x) &= \sum_n a_n x^n = \sum_n (b_n + b_{n-3} + b_{n-4}) x^n \\ &= \sum_n b_n x^n + x^3 \sum_n b_{n-3} x^{n-3} + x^4 \sum_n b_{n-4} x^{n-4} \\ &= (1 + x^3 + x^4) \widehat{f}(x); \end{aligned}$$

in other words,

$$\widehat{f}(x) = \frac{f(x)}{1 + x^3 + x^4}.$$

Once again, the exponents arising in the denominator $1 + x^3 + x^4$ depend on the specific choice of feedback loops. (Other than the constant term 1, in our example the x^3 and x^4 terms arise from the cells 3 and 4 where the feedback loops originate.) Note that the forward and backward shift registers give implementations of multiplication and division by specified polynomials in $\mathbb{F}_2[\![x]\!]$, respectively. To implement multiplication by a rational function $\frac{m(x)}{m'(x)} \in \mathbb{F}_2(x)$, we may connect two registers in series (a forward shift register for $m(x)$ and a backward shift register for $m'(x)$); but in fact a more efficient solution is available by using a single sequence of cells with specified forward and backward loops.

In the design of effective error-correcting codes, very popular choices are convolutional codes which combine shift registers as described above, with interleavers and other primitive stream processing units. Such codes are among the best codes available—in other words they have the highest possible rate of information transfer for a given error-correcting capability.

Shift registers can also be used to give efficient implementation of the multiplication and division in finite fields of the form $\mathbb{F}_q$ where $q = 2^e$.