2 Scalar, Deterministic, Discrete Dynamic Systems

According to our taxonomy of models on page 6, a scalar, deterministic, discrete dynamic system describes a single variable ("scalar") that varies over time ("dynamic") at discrete time instants ("discrete"), and in a way that is completely predictable ("deterministic") from enough of its past values. The variable itself, regarded as a function of time, is merely a sequence or time series.

Sequences are introduced next. Thereafter, we look at transformations of sequences and at linear transformations specifically. Linearity turns out to be a very useful property of some (but by no means all) systems. Our first examples of transformations will be in the form of convolutions, and then of discrete dynamic systems more generally.

Sequences

A sequence is a list of numbers. More formally, it is a function that has a subset of the integers \( \mathbb{Z} \) as its domain and the real numbers \( \mathbb{R} \) as its codomain:

\[
u(n) : \mathbb{Z} \subseteq \mathbb{Z} \rightarrow \mathbb{R}.
\]

The integers in the domain number the values in the list consecutively. For instance, the sequence 3, -1, 2, 4, 0 can be seen as the function from \( Z = \{0, 1, 2, 3\} \) to the reals that maps 0 to 3, 1 to -1, 2 to 4, and 3 to 0. It may at first look contrived to view a list of numbers as a function, but if you do so, you will be able to generalize what you learn here to continuous-time systems as well.

The set \( \mathbb{Z} \) of integer numbers extends all the way to negative infinity. In some cases, however, we may restrict our attention to sequences that exist starting from some point in time onwards, and that point is usually denoted as \( n = 0 \). In that case, the subset \( Z \) is the set \( \mathbb{N} \) of natural numbers 0, 1, \ldots. For instance, we could let \( u(n) \) be the sequence of daily closing values of the Dow Jones Industrial Average (DJIA). This measure of the performance of US industrial companies listed on the New York stock exchange was first published on May 26, 1896, so we could let time instant \( n = 0 \) denote this date. That day was a Tuesday, so \( n = 0 \) through 4 represent May 26, 27, 28, 29, and June 2nd, respectively. May 30 and 31 fell on a weekend, so no index was published on those days, nor was it published on Monday, June 1st, Memorial Day. Another plausible time origin \( n = 0 \) could be set on the first day for which you happen to have data available.

From this example, it is clear that the domain variable \( n \) does not have the dimensions of time: \( n \) is a dimensionless number, an index into a sequence of time values. The indices cover all of the natural numbers (at least up to the latest time we are interested in) without gaps, but the times themselves are not spread regularly: for instance, we skipped three days between \( n = 3 \) and \( n = 4 \) in the example above.

One way to represent \( u(n) \) concretely, that is, in a way that lets us determine \( u(n) \) if we give any \( n \in \mathbb{N} \), would be to list it explicitly in a table. For instance, the Resources sub-page of the class web page links to a file with all the closing values of the Dow Jones index since January 3rd,

---

8Yes, they did have Memorial Day back then. This holiday was first established to commemorate military casualties in the American Civil War, 1861-1865.
1900. So for us a good convention is to let \( n = 0 \) denote this date instead. The sequence \( u(n) \) is the last column of data in that file.

If looking at the entire sequence of 28,961 values is daunting, you may want to consider a subsequence, for instance, the closing on the last business day of every year. This subsequence, shown in Figure 2(a), has only 106 values in the file mentioned above. Alternatively, Figure 2(b) shows all the 252 closing values in 2005.

Figure 2: (a) Closing values of the Dow Jones Industrial Average on the last business day of every year from 1900 to 2005. Note the unprecedented rise associated with the dot-com bubble starting in 1995, the burst of the bubble beginning in the year 2000, and the further dip after the September 11, 2001 events. (b) This closer look shows all the daily closing values in the 252 business days in 2005.

The sequence \( u(n) \) by itself is not a dynamic system. If I had a simple formula that lets me compute future values of the index from current values, then (i) I would have a dynamic system, and (ii) I would probably be in a different line of business.\(^9\) We will get to dynamics systems after some preliminaries.

**Transformations and Linearity**

The relative daily return of a price index \( u(n) \) like the DJIA is another sequence. Let us call it \( r(n) \). It is defined as the percentage by which an investment in an index fund (i.e., an investment fund whose performance tracks the index exactly) would grow on day \( n \):

\[
r(n) = 100 \frac{u(n) - u(n - 1)}{u(n - 1)}
\]  

\(^9\)Arguably more remunerative but less enjoyable than my current one.
For instance, if you had invested in the Dow just after closing on February 15, 2006 (day $n - 1 = 28955$), you would have bought the stocks represented in it when they were worth 11,058 points (in the arbitrary unit used by the DJIA). If you had sold them after closing the day after (day $n = 28956$), the same mix of stock would have been worth 11,120 points, for a relative daily return

$$r(28956) = 100 \frac{11,120 - 11,058}{11,058} \approx 0.56\%.$$  

This seems a small number, but it is not bad at all for a single day: The average daily return over the 20th Century was 0.17 percent.\(^{10}\)

The absolute daily return would just be the daily increment (positive or negative) of the Dow,

$$a(n) = u(n) - u(n - 1)$$

(62 points in the example). This is less useful, because whether 62 points is good or bad depends on the value of the index. In 1900, when the index was around 66, a one-day gain of 62 points would have been fantastic!

However, the absolute return illustrates the idea of sequence transformation more simply. First, the absolute return is a linear operation, while the relative return is not.

A sequence transformation $z = T[u]$ is linear if for any two sequences $u$ and $u'$ and numbers $b$ and $b'$

$$T[bu + b'u'] = bT[u] + b'T[u'] \quad (2)$$

Let us try to understand what this statement means. First, a transformation

$$a = T[u]$$

takes a whole sequence $u(n)$ and produces a whole new sequence $a(n)$. The absolute rate of return (or the relative one, for that matter) is a case in point: the calculation we did earlier can be done for every value of $n$, so if we have $u(n)$ for every $n$ then we can compute $a(n)$ for every $n$.

Well, almost every $n$: to compute $a(0)$ we would have to have $u(0)$ (which we have) and $u(-1)$ (which we do not have). The conceptually correct choice here is to define $u(n)$ for $n \in \mathbb{N}$, but $a(n)$ only for $n \in \mathbb{N}^+$, the set of naturals except zero.

When programming in Matlab on entire sequences, it is natural to store these into vectors. These are lists of numbers, and come in two types: Row vectors are defined with syntax like the following:

$$v = [3 \ 1.2 \ -5];$$

Think of the three numbers as placed next to each other in a row. A column vector, on the other hand, would be defined like this:

$$v = [3; \ 1.2; \ -5];$$

---

Here, the numbers are to be thought of as placed one below the other. To access the second entry of either vector you would say $v(2)$, and to access the last you can say $v(3)$ if you know the length of $v$, or $v(\text{end})$ if you do not.

A row vector can be converted to a column vector and vice versa by the transposition symbol $\prime$ (a prime). A vector $v$ of any shape\(^{11}\) can be coerced to be a column vector by the notation $v(:)$.

When working on a sequence of data such as the Dow closing values, we store the entire sequence into a vector. In this class we choose to store sequences into column vectors.

Vectors raise a slight difficulty when dealing with time: If we define a vector $u$ to hold, say, the 252 values of $u(n)$ in the year 2005, we cannot say $u(0)$ to refer to the first entry. This is because the first entry in a Matlab vector is always numbered as entry 1, not entry 0. To keep track of this, we also store the time indices in a vector defined with the following statement:

$$n = (0:\text{length}(u) - 1)^\prime;$$

This instructs Matlab to build a vector of all the integers from 0 to one less than the length of the vector that stores the sequence $u$ (in the example, $n$ would end up being a vector of length 252 with the integers from 0 through 251). The prime in the expression makes sure this is a column vector.

You may think that it is more efficient to just remember that $u(1)$ means $u(0)$ and so forth, but this is not the case: The time and space spent in computing and storing the vector $n$ is nothing compared to the confusion arising down the line from not being clear at the outset.\(^{12}\)

A second source of awkwardness stems from the different lengths of $u$, the vector of 252 Dow closing values in 2005, and of $a$, the vector of only 251 daily returns: We assume that we do not have access to the daily closing value on the last business day of 2004, so we cannot compute $a(0)$. To address this discrepancy, we make both sequences as long as $u$, so both are consistent with the time index vector $n$, and we set

$$a(1) = \text{NaN};$$

The acronym NaN stands for “Not a Number,” and is a floating-point value that Matlab (and any language that complies with the IEEE Standard 754 floating-point convention) reserves to denote undefined numerical values.

Assuming now that the Matlab vector $u$ contains all of the 28,961 values of the Dow index, there is a slow way and a fast way to compute the absolute rate of return in Matlab. First the slow way. Write the following code into file slowAbsoluteReturn.m:

```matlab
function a = slowAbsoluteReturn(u)
a(1) = NaN;
last = length(u);
for n = 2:last
    a(n) = u(n) - u(n-1);
end

On my PC, executing slowAbsoluteReturn took about 8.5 seconds. To find out, type
```

\(^{11}\)In fact, any matrix.

\(^{12}\)Of course, we could also set the beginning of time at $n = 1$, but this contradicts standard mathematics conventions. We choose to work around standard Matlab convention instead.
tic; a = slowAbsoluteReturn(u); toc

(no semicolon at the end!) on a single line, and then press Enter. The instruction tic starts a clock, the instruction a = slowAbsoluteReturn(u) executes our code and puts the resulting sequence into the vector a, and the instruction toc stops the clock and returns the time elapsed since the last tic. Your computer may be doing other things while also running Matlab, so this is only an approximate indication of (or rather an upper bound on) running time.

There are two reasons why this is slow. First, Matlab does not know how big to make the vector a being built in the for loop. So it starts with a one-entry vector (enough to store a(1)), and every time it needs more space is extends the storage for the vector on the fly. Doing this tens of thousands of times takes time. Second, Matlab is an interpreted language: when it encounters the for loop, it reads the lines of the instructions up until the end, translates them to machine instructions, executes these, then goes back to the for line, increments n to 2, and repeats everything all over again (including translation to machine instructions). Again, this takes time. The fast way is as follows:

```matlab
function a = absoluteReturn(u)
a = NaN * ones(size(u));
a(2:end) = u(2:end) - u(1:(end-1));
```

This only took 3 milliseconds, so this version is about 2,500 times faster than the previous one! The instruction a = NaN * ones(size(u)) pre-allocates the entire vector (a vector of NaNs the same size as u), the instruction a(1) = NaN is no longer needed (all entries in a are initially set to NaN), and the rest is Matlab subscript magic: u(2:end) is the entire vector u from the second entry to the last. This is a vector with one fewer entry than u. So is u(1:(end-1)), the vector of all entries in u except the last. These two vectors have the same size, so they can be subtracted from each other. Subtracting vector w from vector v (as in v - w) means producing a vector of the entry-by-entry differences. The result goes into a(2:end), a destination sub-vector of the appropriate size. This operation does not overwrite a(1), which was initially NaN, as it should remain. Of course, the computer must compute all the differences one by one. However, this is done in a corresponding for loop that is already written in machine language, because Matlab has all useful vector (and matrix) operations compiled, that is, translated once and for all into machine language. It all happens under the hood of Matlab. The useful part of the computation (all the differences) is the same with both versions of the code, but the fast version does one memory allocation and one translation to machine language instead of 28,961.

The result of this computation is shown in Figure 3 for the year 2005 only, so we can see some detail.

Now back to mathematics. The absoluteReturn function (or its slow counterpart) takes the whole sequence u as its input, and produces a whole sequence a as its output, so is a sequence transformation. Mathematicians do not insist on the mathematical objects u and a being finite. It should not be too much a stretch of your imagination to envision the Matlab computation above continuing forever. Yet eventually we compute one entry at a time: we take two values of u, namely, u(n) and u(n - 1), and we combine them into the value of a(n) by taking their difference.

---

13In this case, the slow version makes more sense than the fast one, since end in the latter is now undefined!
The transformation $T$ acts on the whole sequence $u$ to produce the whole sequence $a$, and yet does so one entry of $a$ at a time. Thinking of a transformation in these two different ways (global and local, so to speak) is generally useful in mathematics.

So what does linearity mean? Suppose that we take two sequences $u$ and $u'$. Maybe one is the Dow and the other is the Nasdaq. We form a new sequence

$$y = bu + b'u'$$

where $b$ and $b'$ are any two numbers we are given. This means that each entry of $y$ is formed as follows:

$$y(n) = bu(n) + b'u'(n).$$

For instance, $b$ and $b'$ could represent how much money we invest in a Dow-based fund and in a Nasdaq-based fund, respectively. We can then use $T$ to transform $y$ to get yet another sequence, call it $\alpha$:

$$\alpha = T[y].$$

This new sequence would be the absolute daily return on the combined investment.

The transformation $T$ is linear if $\alpha$ can be computed alternatively by first transforming each of $u$ and $u'$ separately through $T$ (this in the example would yield the absolute returns for each index separately), and then combining them through $b$ and $b'$. That is,

$$bT[u] + b'T[u']$$

also produces $\alpha$, or, in a single line,

$$T[bu(n) + b'u'(n)] = bT[u] + b'T[u'].$$
Figure 4: A sequence transformation $T$ is linear if the same sequence $\alpha$ can be obtained in the following different ways: (i) (Diagram in the gray box) First combine two sequences $u$ and $u'$ to obtain a new sequence $y = bu + b'u'$, and then transform $y$ through $T$ to obtain $\alpha$; or (ii) (Surrounding diagram) First transform $u$ and $u'$ through $T$ to obtain $a$ and $a'$, and then combine them in the same way, $by + b'y'$, to obtain $\alpha$.

This definition is illustrated in Figure 4.

Let us now check that the absolute return sequence $a(n)$ is indeed a linear transformation. We have

$$T[u] = u(n) - u(n-1) \quad \text{and} \quad T[u'] = u'(n) - u'(n-1)$$

so that

$$bT[u] + b'T[u'] = b(u(n) - u(n-1)) + b'(u'(n) - u'(n-1)).$$

(3)

On the other hand, if we let

$$y(n) = bu(n) + b'u'(n),$$

we have

$$T[bu(n) + b'u'(n)] = T[y] = y(n) - y(n-1) = (bu(n) + b'u'(n)) - (bu(n-1) + b'u'(n-1)).$$

(4)

A simple rearrangement of terms shows that the right-hand sides of equations (3) and (4) are equal to each other, so $T$ is a linear transformation.
However, the *relative* daily return is not a linear transformation. Showing this is even easier: while proving that something is linear requires us to show that the two expressions $T[bu(n) + b'u'(n)]$ and $bT[u] + b'T[u']$ are equal for all possible choices of $b$ and $b'$ (thus, we need to prove a *universal* statement), to prove that something is not linear only requires showing a counterexample, that is, a single instance in which the equality does not hold (this is an *existential* statement). Let $b = 2$ and $b' = 0$, so $T[bu(n) + b'u'(n)]$ and $bT[u] + b'T[u']$ simplify to $T[2u(n)]$ and $2T[u]$, respectively. We have

$$T[2u(n)] = 100 \frac{2u(n) - 2u(n - 1)}{2u(n - 1)} = 100 \frac{u(n) - u(n - 1)}{u(n - 1)} = T[u(n)] ,$$

which is one half of $2T[u]$, so the relative return is not linear: Doubling the values of the input sequence doubles the absolute return (this is part of what linearity requires), but it leaves the relative return (a fraction) unaltered.

We now know what linearity is, but not quite why it is important. We will leave the answer to this question to a later time. For now, please just digest the concept itself.

**Convolution**

Suppose that we are computing the absolute return as data comes in, one value at a time. This is the “slow version” of the Matlab code. To compute the return now (*i.e.*, at time $n$) we need to the current value $u(n)$ of the Dow, as well as its previous value $u(n - 1)$. To this end, we need to have memory storage, a location in which to remember $u(n - 1)$. After we have computed

$$a(n) = u(n) - u(n - 1) ,$$

we can overwrite the stored $u(n - 1)$ with $u(n)$, which will become the “previous” value as the time index $n$ is incremented by one unit, ready for the next day’s closing of the Stock Exchange. Graphically, we can view the operation as a *First-In, First-Out (FIFO) buffer*, as illustrated in Figure 5(a). Values in the buffer are multiplied each by some number ($1$ and $-1$ in the case of the absolute return), and added together to yield the output $a(n)$.

Figure 5(b) shows a slight generalization of the same structure: instead of two cells, the FIFO buffer has an arbitrary number of them, each with its own multiplier $c_i$ (an arbitrary value). This operation is called a **convolution** of the input sequence $u(n)$ with the sequence of coefficients $c_0, c_1, \ldots, c_m$. Mathematically, we can summarize as follows.
Figure 5: (a) Dow closing values enter the First-In, First-Out (FIFO) buffer of length two from the left, and shift to the right by one cell at each time step. Data that falls off at the right end is lost forever. The two values in the buffer are multiplied by $1$ and $-1$, respectively, and added together to yield the current value of the absolute return $a(n)$. (b) A straight-forward generalization of the same structure: multipliers are arbitrary numbers, and there is an arbitrary number of cells in the buffer. The whole operation is called a convolution of the input with the sequence of coefficients $c_0, c_1, \ldots, c_m$.

Matlab has a built-in function \texttt{convn} that computes the convolution. So a third way to compute the absolute daily return in Matlab is to type

\begin{verbatim}
a = convn(u, [1, -1]', 'valid'); a = [NaN; a];
\end{verbatim}

For convolution to do what we want, Matlab requires the two input vectors to be both row vectors or both column vectors. Since in our convention $u$ is a column vector, the row vector \texttt{u(1:n)} is used as the second argument.
[1 -1] is made into a column vector by the transposition symbol '. The 'valid' argument specifies that the output sequence is computed only starting from the time that the FIFO buffer has all the values it needs (i.e., starting with \(a(1)\), rather than \(a(0)\)). Without this argument, \texttt{convn} would also compute values output by the circuitry in Figure 5(b) while the buffer fills up with actual data initially (assuming it starts filled with all zeros), and until the last data value has been flushed from the buffer, so the output sequence would be a bit longer. To make the output sequence \(a\) have the same length as the input sequence \(u\) we append the customary \texttt{NaN} at the beginning of \(a\) with the instruction \(a = [\texttt{NaN}; a]\).

This version of convolution took about ten milliseconds on my PC, a bit slower than, but comparable to, the previous "fast" version.

Since we have figured out how to use \texttt{convn} properly, it is best to wrap our thinking into a routine that does all the necessary work. In the future, we just call this routine for convolution, without having to re-think all the details\(^{14}\):

\begin{verbatim}
function y = fir(u, c)
    y = convn(u(:), c(:), 'valid');
    y = [NaN * ones(length(c)-1, 1); y];
end
\end{verbatim}

The reasoning here is the same as for the special case \(c = [1 -1]\), but generalized to an arbitrary vector \(c\) of coefficients.

A useful convolution in data analysis is the moving average, also known as the running average:

\[
\mu_N(n) = \frac{1}{N} \sum_{i=0}^{N-1} u(n-i).
\]

This is the mean of the \(N\) most recent values of the input sequence \(u(n)\). Check that this is indeed a convolution of the form (5), with \(m = N - 1\) and with \(c_i = 1/N\) for \(i = 0, \ldots, N - 1\).

For instance, to compute the moving average with 10 values we can just say

```
mu = fir(u, ones(10, 1)/10);
```

where \texttt{fir} was defined earlier on. The Matlab function call \texttt{ones(10, 1)} produces a 10 by 1 matrix of ones, that is, a column of ten ones.

The moving average computes the average of the last \(N\) values of the input sequence. For instance, Figure 6 shows the two plots obtained by transforming the sequence in Figure 2(b) with a moving average of ten (solid graph) and forty (dotted graph) values. The averaging tends to cancel the day-to-day fluctuations, and more so for longer averaging windows. However, the averaged values also tend to lag behind, because they represent average values over a past time interval whose length increases with the averaging window.

Suppose for a moment that no data comes into the convolution system of Figure 5(b). More precisely, suppose that all values \(u(n)\) are equal to zero. Then the output is zero as well. This is a

\[^{14}\text{The reason for the strange name, \texttt{fir}, will become apparent soon.}\]
(trivial) consequence of linearity, since for a linear system if we set \( b = b' = 0 \) in the definition (2) of linearity we obtain

\[
T[0u + 0u'] = 0T[u] + 0T[u'] = 0
\]

no matter what \( u \) and \( u' \) are. So our filter needs a nonzero input in order to produce a nonzero output.

Consider now a very simple input:

The sequence \( \delta(n) \) defined as follows

\[
\delta(n) = \begin{cases} 
1 & \text{for } n = 0 \\
0 & \text{for } n \neq 0 
\end{cases}
\]  

(6)

is called an \textit{impulse}. 
The impulse response for our filter is easy to determine. One way is to follow what happens as the single 1 in the input sequence \( u(n) = \delta(n) \) travels through the FIFO buffer of Figure 5(b): before time \( n = 0 \), the buffer entries are all zero, and so is therefore the output. At time \( n = 0 \), the 1 is in the first cell, so it is multiplied by \( c_0 \), and summed to all zeros from the other cells: the output is \( c_0 \) at time 0. At every time step, the 1 travels to a new cell, and multiplies the corresponding coefficient, which is thereby copied to the output. After the 1 has traveled through the FIFO buffer, the output is zero for all times. In summary:

The output of filter (5) when the input is an impulse, that is, the filter’s impulse response, is the sequence

\[
h(n) = \begin{cases} 
c_n & \text{for } 0 \leq n \leq m \\
0 & \text{for other values of } n 
\end{cases}
\]

Because the number of nonzero entries in this response is at most \( m + 1 \), the filter (5) is called a Finite Impulse Response (FIR) filter.

Thus, a FIR filter needs nonzero inputs to come to life. If an impulse shows up, the filter’s output comes to life for a finite time (at most) \( m + 1 \), and then becomes quiet again. Because of this, FIR filters seem to be rather limited as models of nontrivial system. They are useful to clean up data, like the moving average filter does. The next Section adds a small but crucial twist to make filters more interesting.

Scalar, Deterministic, Linear, Discrete Dynamic Systems

Figure 7 shows the transition from a FIR filter to a scalar, deterministic, linear, discrete dynamic system, also known as an Infinite Impulse Response (IIR) filter. The main change is that the FIFO buffer operates on the output of the filter, rather than on the input. Specifically, the input is fed into the adder, and the output \( y(n) \) (which now also includes the input value \( u(n) \)) is fed back to the first cell of the buffer. Drawing the output \( y(n) \) from the first cell of the buffer, as shown in the figure, instead of drawing it directly from the adder, is more pleasing from a circuit point of views, since the first cell then holds the value \( y(n) \) during the interval between time instants. Of course, these two solutions are perfectly equivalent from a mathematical point of view.

Mathematically:
The Infinite Impulse Response (IIR) filter of Figure 7(b) is governed by the following equation:

\[ y(n) = u(n) + \sum_{i=1}^{m} c_i y(n - i), \quad (7) \]

which is a compact form for

\[ y(n) = u(n) + c_1 y(n - 1) + \ldots + c_m y(n - m). \]

Note the summation starting at 1 rather than 0. Convince yourself that this is the correct equations by writing what comes out of the adder in the diagram of Figure 7(b). The reason why this filter is called IIR is that the effects of an impulse input \( u(n) = \delta(n) \) (see equation (6)) can persist indefinitely because of the feedback link. Let us see this in the simplest possible IIR filter, one with a two-cell buffer and therefore a single tap (Figure 8). Please verify that this filter, which has equation

\[ y(n) = u(n) + cy(n - 1), \quad (8) \]

becomes the same as the sandhill crane model in the textbook (page 17), that is,

\[ y(n) - y(n - 1) = ry(n - 1) \]

if we set

\[ u(n) = 0 \quad \text{and} \quad c = 1 + r \]

(the textbook uses \( x \) instead of \( y \), but this is of course irrelevant).
Impulse Response

Figure 8: A single-tap IIR filter. Since there is only one coefficient, we set \( c_1 = c \) for notational simplicity.

Setting \( u(n) = 0 \) for all \( n \) is obviously a drastic measure: zero input, zero output, as you can easily convince yourself by tracking what happens in Figure 8 as you feed zeros to the machine. With \( u(n) = \delta(n) \), on the other hand, the filter comes to life. The resulting output is the filter’s impulse response, which is derived next.

The input’s single nonzero value 1 will get into the FIFO at time \( n = 0 \). That 1 is also added to the zero from the only tap of the filter, and makes it to the output:

\[
y(0) = u(0) = 1
\]
as well as being fed back into the first cell of the FIFO buffer. At the next time step, \( n = 1 \), the 1 shifts into the second cell. There it is multiplied by \( c \) and added to \( u(1) \), which is now equal to zero (equation (6)), so that

\[
y(1) = c.
\]

This value of \( c \) is fed back to the first cell, and moves to the second at time \( n = 2 \). Here it multiplies by \( c \) to produce an output

\[
y(2) = c^2.
\]

You get the point. The impulse response of this filter is

\[
h(n) = \begin{cases} 
0 & \text{for } n < 0 \\
c^n & \text{for } n \geq 0 
\end{cases}
\] (9)

Figure 9 shows this response for six different filters, with \( c = \pm 1, c = \pm 1.03 \), and \( c = \pm 0.9 \).

Thus, an IIR filter, even one as simple as the single-tap filter of Figure 8, has a much more interesting life than a FIR. First, the impulse response persists indefinitely: a single 1 in the input
at some point in time excites the system, which then produces output by itself forever. Second, depending on the value of the single tap of this simple machine, a multitude of qualitatively different output behaviors can arise: constant \( (c = 1) \), oscillatory \( (c = -1) \), monotonically convergent \( (0 < c < 1) \), monotonically divergent \( (c > 1) \), damped oscillatory \( (-1 < c < 0) \), or resonant\(^{15}\) \( (c < -1) \).

The “fast” approach we used to program a FIR filter in Matlab can no longer be used for IIR filters: the output is now much longer (infinitely so!) than the nonzero part of the input or than the tap sequence, so no simple finite vector manipulation will work. Here is the more literal, slow version:

```matlab
function y = iir(c, u, x0)

if nargin < 3
    x0 = zeros(length(c), 1);
else
    if length(x0) ~= length(c)
        error('c and x0 must have the same length')
    end
end

N = length(u);

u = u(:);
c = c(:);
x = x0(:);
y = zeros(1, N);
```

\(^{15}\)That is, divergent oscillatory.
for n = 1:N
    y(n) = u(n) + c' * x;
    x = [y(n); x(1:(end-1))];
end

At this point you should be able to understand what this code is doing, with the possible exception of the nargin business. The Matlab standard variable nargin denotes the number of arguments with which the current function (iir in this case) is being called. If we call

\[ y = \text{iir}(c, u); \]

then nargin is automatically set to 2, because we passed the two arguments c and u. If we call

\[ y = \text{iir}(c, u, z); \]

then nargin is automatically set to 3. So the first two instructions say that if iir is called with fewer than three arguments, so that the parameter x0 is undefined, then x0 should be set to a vector of as many zeros as there are taps in the vector c of tap coefficients. Thus, the argument x0 is the initial content of the FIFO buffer of the filter. It is set to all zeros by default, but it can also be set to something else if desired, by passing a third argument of proper size. If this is the case, then the else clause of the if statement is executed, to check that x0 has the proper length. If it does not, then the built-in Matlab function error prints an error message and aborts execution of the function.

Note that most of the code in iir is argument processing and packaging. The real meat is in the for loop in the last four lines. This is not uncommon in programming.

Another novelty in this code is the expression

\[ c' * x \]

within the for loop. The vector c' is a row vector, because c is a column vector and the prime transposes it into a row. The x vector is a column vector. In Matlab, the product symbol ' * ' between a row vector and a column vector of same length m is the so-called inner product:

\[ \sum_{i=1}^{m} c(i)x(i). \]

Please verify that this is indeed what the adder in Figure 7(b) does (except for the addition of u(n), which is done separately in the code).

Response to a General Input

The situation is only slightly more complex when the input u(n) is general, rather than being equal to the impulse δ(n). Following again the workings of the machine in Figure 8, we obtain the following values for the output:

\[
\begin{align*}
y(0) &= u(0) \\
y(1) &= cu(0) + u(1) \\
y(2) &= c(cu(0) + u(1)) + u(2) = c^2u(0) + cu(1) + u(2) \\
y(3) &= c(c^2u(0) + cu(1) + u(2)) + u(3) = c^3u(0) + c^2u(1) + cu(2) + u(3) \\
&\vdots
\end{align*}
\]
The pattern is obvious:
\[ y(n) = \sum_{i=0}^{n} c^i u(n - i) . \] 
(10)

This should look familiar. Comparison of this expression with equation (5), which describes convolution of a signal with the sequence of coefficients of a FIR filter, and with equation (9), which shows the impulse response of the single-tap IIR filter, shows the following important result: The response of a one-tap IIR filter to a general input sequence \( u(n) \) is the convolution of \( u(n) \) with the impulse response of the filter.

**Linearity and Convolution**

Thus, at least for the single-tap IIR filter, the output is the convolution of the input with the filter’s impulse response. This result holds in fact for all filters, FIR or IIR, with one or more taps. The proof for FIR filters was given in the discussion of Figure 5.

A more general proof, which also holds for IIR filters, derives this result directly from the linearity of all filters. This proof is instructive. First, it is clear that FIR filters are linear: from equation (5), we see that an input \( bu + b'u' \) to a FIR filter produces an output
\[
\sum_{i=0}^{m} c_i [bu(n) + b'u'(n)] = b \sum_{i=0}^{m} c_i u(n) + b' \sum_{i=0}^{m} c_i u'(n) .
\]

If we denote the transformation performed by this system with \( T \), this means that

\[ y = T[u] \quad \text{and} \quad y' = T[u'] \quad \text{then} \quad by + b'y' = T[bu + b'u'] \]

as required by the definition (2) of linearity.

Linearity of IIR filters is proven in a similar although somewhat less direct fashion. The output \( y = T[u] \) of an IIR filter \( T \) starting with a zero buffer is completely determined once its input \( u \) is known. In other words, given an input sequence \( u \), there is exactly one sequence \( y \) that satisfies the recurrence equation (7) that defines an IIR filter. If a sequence \( u' \) is fed to the same system, the resulting output \( y' \) likewise satisfies the same recurrence:
\[ y'(n) = u'(n) + \sum_{i=1}^{m} c_i y'(n - i) . \]

Multiplying these two expressions by \( b \) and \( b' \) respectively and adding the result yields the following expression:
\[ by(n) + b'y'(n) = bu(n) + b'u'(n) + \sum_{i=1}^{m} c_i [by(n - i) + b'y'(n - i)] . \]

This shows that the output that filter \( T \) yields in response to the input sequence \( bu + b'u' \) is \( by(n) + b'y'(n) \), so linearity is once again verified.
Let us now use linearity of FIR and IIR filters to derive the announced convolution result. By definition, the (finite or infinite) impulse response \( h(n) \) of a filter (FIR or IIR) or, for that matter, of any linear system is the output of that system when the input is equal to \( \delta(n) \). Any input \( u(n) \) can be written by a linear combination of properly shifted \( \delta \) pulses in an apparently trivial way:

\[
 u(n) = \sum_{k=\infty}^{\infty} u(k)\delta(n - k) .
\]  

(11)

This expression is so innocent-looking and yet important that it requires some explanation. The sequence \( u(0)\delta(n) \) is zero everywhere except for \( n = 0 \), where it equals \( u(0) \) (see Figure 10). Similarly, the sequence \( u(1)\delta(n - 1) \) is nonzero only when the argument of \( \delta \) is zero, that is, when \( n = 1 \), where the sequence equals \( u(1) \). In general, the scaled impulse sequence \( u(k)\delta(n - k) \) is zero everywhere except at \( n - k \), where it equals \( u(k) \). If we add all these sequences together sample by sample, we obtain the original sequence \( u(n) \). Please look at Figure 10 again.

![Figure 10: Any sequence \( u(n) \) can be written as the sum of infinitely many \( \delta \) pulses, properly shifted and scaled.](image)

The summation endpoints \(-\infty \) and \( \infty \) are possible with the convention that if the sequence \( u(n) \) is defined on a subset \( Z \subset \mathbb{Z} \) of the integers \( \mathbb{Z} \), then it is conventionally extended to all of \( \mathbb{Z} \) by setting \( u(n) = 0 \) at the values of \( n \) for which the original sequence \( u \) is undefined. For instance, if \( u(n) \) is defined on the nonnegative integers, \( n \geq 0 \), then it is redefined so that \( u(n) = 0 \) for \( n < 0 \).

If the response of a linear system to the impulse \( \delta(n) \) is \( h(n) \), then the response to the shifted and scaled impulse \( u(k)\delta(n - k) \) is the shifted response \( u(k)h(n - k) \), so the response to the sum
(11) of shifted and scaled impulses, because of linearity, is the sum of shifted and scaled responses:

$$\sum_{k=-\infty}^{\infty} u(k)h(n-k)$$

(12)

that is, the convolution of $u$ and $h$.

The change of variable $j = n - k$ shows that convolution is symmetric in $u$ and $h$:

$$\sum_{k=-\infty}^{\infty} u(k)h(n-k) = \sum_{j=-\infty}^{\infty} u(n-j)h(j)$$

This is an important consequence of linearity for any linear system:

Let $h(n)$ be the impulse response of a linear system $T$, that is, the output from $T$ when the input is equal to the unit impulse $\delta(n)$. Assume that both $h(n)$ and the input $u(n)$ are defined for all integers (to ensure this, if a sequence is undefined at $n$, its value there is conventionally set to zero). Then, the response $y(n)$ of $T$ to input $u(n)$ is the convolution of $u$ and $h$:

$$y(n) = [u * h](n) = \sum_{k=-\infty}^{\infty} u(k)h(n-k) .$$

This is the same as

$$y(n) = [h * u](n) = \sum_{k=-\infty}^{\infty} h(k)u(n-k) .$$

**Initial Conditions and Free Evolution**

In all the preceding considerations, the input was fed to filters that started with all zeros in their FIFO buffers. In view of later generalizations, it is useful to point out that nonzero values initially present in the buffer would alter the output. For the single-tap IIR filter, these nonzero values can be easily simulated by an appropriate impulse at the input, which has the role of loading the desired value into the buffer. For IIR filters with more taps, however, this is no longer the case. While it is possible to simulate an initial buffer content with an appropriate input, the relationship between the two is nontrivial, and is best analyzed in a later Section. For now, we merely develop some notation to describe the situation.

Let $T_y$ denote an IIR filter that starts with initial buffer contents\(^\text{16}\) $y = [y_0, \ldots, y_{-m+1}]$ in the buffer (left to right in Figure 7(b)). These initial values are called the initial conditions of the system. So the same filter with zero initial conditions is $T_0$.

\(^{16}\)Note that the initial content $y_{-m}$ of the last cell cannot be chosen arbitrarily, but is determined by the recurrence
To study the response of $T_y$ it is sufficient to study its \textit{free evolution}, that is, its output $T_y[0]$ when the input sequence $u(n)$ is identically zero: Because of linearity,

$$T_y[u] = T_y[0] + T_0[u].$$

(13)

In words, the impulse response with zero buffer and the nonzero-buffer response with zero input can be studied separately, and their effects can be added because of linearity.

For the single-tap IIR filter of Figure 8, the free evolution of the nonzero-buffer filter is the same as the response of the zero-buffer to an input impulse scaled by $y = y_0 = cy_{-1}$,

$$T_{y_0}[0] = T_0[y_0 \delta(n)].$$

This is because it does not matter if the only possible initial value comes from the input or is instead initially stored in the buffer. This observation allows writing a completely general solution for a first-order recurrence. If the input is nonzero, so that in particular also $u(0) \neq 0$, the scaling factor for the free evolution is not $y_0$ but $y_0 - u(0)$, the value one would have at the output if the input were zero. We can therefore summarize as follows:

\begin{center}
\begin{tabular}{|l|}
\hline
The solution to the first-order recurrence \hfill \\
y(n) = cy(n - 1) + u(n) \hfill \\
starting at time $n = 0$ with initial condition $y(0) = y_0$ is \hfill \\
y(n) = [y_0 - u(0)]c^n + \sum_{i=0}^{n} c^i u(n - i). \\
\hline
\end{tabular}
\end{center}

In this solution, the term $y_0c^n$ is the system's \textit{free evolution} (zero input, initial condition $y_0$), and the summation is the \textit{forced evolution} (zero initial condition, input $u(n)$).

For IIR filters with more taps, there still exists an input that loads a given vector $y$ of values into a buffer that is initially zero. However, computation of this input is deferred to later in this course.

\begin{equation}
\text{equation (7) with } n = 0 \text{ and } u(0) = 0:
\end{equation}

$$y_0 = \sum_{i=1}^{m} c_i y_{-i},$$

so that

$$y_{-m} = \frac{1}{c_m} [y_0 - \sum_{i=1}^{m-1} c_i y_{-i}].$$

However, this value is irrelevant once we know $y(0) = y_0$, since at the next point in time it will be shifted out of the buffer.