TRIGONOMETRIC INTERPOLATION AND THE FFT

A

THOUGH trigonometry is as old as Pythagorus, its applications to modern life are numerous. Sines and cosines are periodic functions, and are useful for modeling periodic behaviors, as we saw in Chapter 4. For example, pure sound tones are carried by periodic vibrations of air, making it natural data to be modeled by sines and cosines.

Efficiency of interpolation is bound up with another concept, orthogonality. We will see that orthogonal basis functions make interpolation and least squares fitting of data much simpler and more effective. The Fourier transform exploits this orthogonality, and provides an extremely efficient means of interpolation with sines and cosines. The Fourier transform is at the basis of most methods of coding sound digitally, and of compressing sound files. Variations on the same themes lead to video coding and compression.

This chapter covers the basic ideas of the discrete Fourier transform (DFT), including a short introduction to complex numbers. The role of the DFT in trigonometric interpolation and least squares approximation is featured, and viewed as a special case of approximation by orthogonal basis functions. The computation breakthrough of Cooley and Tukey called the fast Fourier transform (FFT) meant that DFT’s could be computed very cheaply, which moved the DFT to a central place in many areas of engineering. Applications to audio and video compression are introduced in Chapter 11.

10.1 The Fourier Transform

THE French mathematician Jean Baptiste Joseph Fourier barely escaped the guillotine during the French revolution and went to war alongside Napoleon before finding time to develop a theory of heat conduction. To make the theory work he needed to expand functions not in terms of polynomials, as Taylor series, but in a revolutionary way, in terms of sine and cosine functions. Although rejected by the leading mathematicians of the time, today Fourier series pervade many areas of applied mathematics, physics, and engineering. The most profound area of impact may be modern signal processing, which carries over from this chapter to the next chapter on compression.

10.1.1 Complex arithmetic

The bookkeeping requirements of trigonometric functions can be greatly simplified by adopting the language of complex numbers. Every complex number has form \( z = a + bi \), where \( i = \sqrt{-1} \). The geometric representation of \( z \) is as a two-dimensional vector of size \( a \) along the real (horizontal) axis, and size \( b \) along the imaginary (vertical) axis, as shown in Figure 10.1. The
complex magnitude of the number \( z = a + bi \) is defined to be \( |z| = a^2 + b^2 \), and is exactly the distance of the complex number from the origin in the complex plane. The complex conjugate of a complex number \( z = a + bi \) is \( \overline{z} = a - bi \).

![Figure 10.1: Representation of a complex number.](image)

The real and imaginary parts are \( a \) and \( bi \), respectively. The polar representation is \( a + bi = re^{i\theta} \).

The celebrated **Euler formula** for complex arithmetic says \( e^{i\theta} = \cos \theta + i \sin \theta \). The complex magnitude of \( e^{i\theta} \) is therefore 1, and complex numbers of this form lie on the unit circle in the complex plane, shown in figure 10.2. Any complex number \( a + bi \) can be written in its polar representation

\[
z = a + bi = re^{i\theta}
\]

where \( r \) is the complex magnitude \( |z| = \sqrt{a^2 + b^2} \) and \( \theta = \arctan \frac{b}{a} \).

The unit circle in the complex plane corresponds to complex numbers of magnitude \( r = 1 \). Let’s multiply together two numbers on the unit circle, \( e^{i\theta} \) and \( e^{i\gamma} \). There are two ways to do it. First we can convert to trigonometric functions and multiply:

\[
e^{i\theta}e^{i\gamma} = (\cos \theta + i \sin \theta)(\cos \gamma + i \sin \gamma) = \cos \theta \cos \gamma - \sin \theta \sin \gamma + i(\sin \theta \cos \gamma + \sin \gamma \cos \theta)
\]

Recognizing the \( \cos \) addition formula and the \( \sin \) addition formula, we can rewrite this as

\[
\cos(\theta + \gamma) + i \sin(\theta + \gamma) = e^{i(\theta + \gamma)}.
\]

The second way to multiply is simply to add the exponents:

\[
e^{i\theta}e^{i\gamma} = e^{i(\theta + \gamma)}.
\]

As you can see, Euler’s formula helps to hide the trigonometry details, like the sine and cosine addition formulas, and makes the bookkeeping much easier. This is the reason we introduce complex arithmetic into the study of trigonometric interpolation - it can be done entirely in the real numbers, but the subject gets far messier without the simplifying effect of the Euler formula.

Equation (10.2) shows an important fact about multiplying complex numbers with magnitude 1. The product of two numbers on the unit circle gives a new point on the unit circle whose angle is the sum of the two angles.
Trigonometric functions like \( \sin \) and \( \cos \) are periodic, meaning they repeat the same behavior over and over. It is for this reason that we are interested in complex numbers whose powers repeat the same set of numbers over and over. We call a complex number \( z \) an \( n \)th root of unity if \( z^n = 1 \).

On the real number line, there are only two roots of unity, \(-1\) and \(1\). In the complex plane, however, there are many. For example, \( i \) itself is a 4th root of unity, because \( i^4 = (-1)^2 = 1 \). An \( n \)th root of unity is called primitive if it is not a \( k \)th root of unity for any \( k < n \). By this definition, \(-1\) is a primitive second root of unity and a non-primitive fourth root of unity. It is easy to check that for any integer \( n \), the complex number \( \omega_n = e^{-i2\pi/n} \) is a primitive \( n \)th root of unity and a non-primitive fourth root of unity. It is easy to check that for any integer \( n \), the complex number \( \omega_n = e^{-i2\pi/n} \) is a primitive \( n \)th root of unity. The number \( e^{i2\pi/n} \) is also a primitive \( n \)th root of unity, but we will follow the usual convention to use the former for the basis of the transform. Figure 10.3 shows a primitive 8th root of unity \( \omega_8 = e^{-i2\pi/8} \) and the other 7 roots of unity, which are powers of \( \omega_8 \).

Here is a key identity we need later to simplify our computations of the Discrete Fourier Transform. Let \( \omega \) denote the \( n \)th root of unity \( \omega = e^{-i2\pi/n} \). Then

\[
1 + \omega + \omega^2 + \omega^3 + \ldots + \omega^{n-1} = 0.
\] (10.3)

The proof of this identity follows from the telescoping sum

\[
(1 - \omega)(1 + \omega + \omega^2 + \omega^3 + \ldots + \omega^{n-1}) = 1 - \omega^n = 0.
\] (10.4)

Since the first term on the left is not zero, the second must be. A similar method of proof shows that

\[
1 + \omega^2 + \omega^4 + \omega^6 + \ldots + \omega^{2(n-1)} = 0,
\]
\[
1 + \omega^3 + \omega^6 + \omega^9 + \ldots + \omega^{3(n-1)} = 0,
\]
\[
\vdots
\]
\[
1 + \omega^{n-1} + \omega^{(n-1)2} + \omega^{(n-1)3} + \ldots + \omega^{(n-1)(n-1)} = 0
\] (10.5)

The next one is different:

\[
1 + \omega^n + \omega^{2n} + \omega^{3n} + \ldots + \omega^{n(n-1)} = 1 + 1 + 1 + \ldots + 1 = n.
\] (10.6)
Summarizing,

Lemma 10.1 Primitive roots of unity. Let \( \omega \) be a primitive \( n \)th root of unity. Then

\[
\sum_{j=0}^{n-1} \omega^{jk} = \begin{cases} n & \text{if } k/n \text{ is an integer} \\ 0 & \text{otherwise} \end{cases}
\]

Exercise 10.1.5 asks the reader to fill in the details of the proof.

Figure 10.3: Roots of unity. The eight 8th roots of unity are shown. They are generated by \( \omega = e^{-i2\pi/8} \), meaning that each is \( \omega^k \) for some integer \( k \). Although \( \omega \) and \( \omega^3 \) are primitive 8th roots of unity, \( \omega^2 \) is not, because it is also a 4th root of unity.

10.1.2 Discrete Fourier Transform.

Let \( x = (x_0, \ldots, x_{n-1})^T \) be a (real-valued) \( n \)-dimensional vector, and denote \( \omega = e^{-i2\pi/n} \) to be a primitive \( n \)th root of unity. Here is the fundamental definition of this chapter.

Definition 10.2 The discrete Fourier transform (DFT) of \( x = (x_0, \ldots, x_{n-1})^T \) is the \( n \)-dimensional vector \( y = (y_0, \ldots, y_{n-1}) \), where \( \omega = e^{-i2\pi/n} \) and

\[
y_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j \omega^{jk}.
\]

(10.7)

For example, Lemma 10.1 shows that the DFT of \( x = (1, 1, \ldots, 1) \) is \( y = (\sqrt{n}, 0, \ldots, 0) \). In matrix terms this definition says

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{bmatrix} =
\begin{bmatrix}
a_0 + ib_0 \\
a_1 + ib_1 \\
a_2 + ib_2 \\
\vdots \\
a_{n-1} + ib_{n-1}
\end{bmatrix}
= \frac{1}{\sqrt{n}}
\begin{bmatrix}
\omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\
\omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\
\omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_{n-1}
\end{bmatrix}.
\]

(10.8)
Each \( y_k = a_k + ib_k \) is a complex number. The \( n \times n \) matrix in (10.8) is called the **Fourier matrix**

\[
F_n = \frac{1}{\sqrt{n}} \begin{bmatrix}
\omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\
\omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\
\omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2}
\end{bmatrix}
\]  

(10.9)

Except for the top row, each row of the Fourier matrix adds to zero, and the same for the columns since \( F_n \) is a symmetric matrix. The Fourier matrix has an explicit inverse

\[
F_n^{-1} = \frac{1}{\sqrt{n}} \begin{bmatrix}
\omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\
\omega^0 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\
\omega^0 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\omega^0 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2}
\end{bmatrix}
\]  

(10.10)

called the **inverse discrete Fourier transform**. Checking that this is the inverse matrix for \( F_n \) requires Lemma 10.1 about \( n \)th roots of unity. For all points lying on the unit circle in the complex plane, including the powers of \( \omega \), the multiplicative inverse is equal to the complex conjugate. Therefore the inverse DFT is the matrix of complex conjugates of the entries of \( F_n \):

\[
F_n^{-1} = \overline{F_n}.
\]  

(10.11)

Applying the discrete Fourier transform is a matter of multiplying by the \( n \times n \) matrix \( F_n \), and therefore requires \( O(n^2) \) operations (specifically \( n^2 \) multiplications and \( n(n-1) \) additions). The inverse discrete Fourier transform, which is applied by multiplication by \( F_n^{-1} \), is also an \( O(n^2) \) process. Later in this chapter we develop a version of the DFT that requires significantly fewer operations, called the fast Fourier transform.

**Example 10.1** Find the DFT of the vector \( x = (1, 0, -1, 0)^T \).

Fix \( \omega \) to be the 4th root of unity, so \( \omega = e^{-i\pi/2} = \cos(\pi/2) - i \sin(\pi/2) = -i \). Applying the DFT, we get

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3
\end{bmatrix} = \frac{1}{2} \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 \\
1 & \omega^2 & \omega^4 & \omega^6 \\
1 & \omega^3 & \omega^6 & \omega^9
\end{bmatrix} \begin{bmatrix}
1 \\
0 \\
-1 \\
0
\end{bmatrix} = \frac{1}{2} \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -i & -i & -i \\
1 & -i & 1 & -i \\
1 & i & -1 & -i
\end{bmatrix} \begin{bmatrix}
1 \\
0 \\
-1 \\
0
\end{bmatrix} = \begin{bmatrix}
0 \\
1 \\
0 \\
1
\end{bmatrix}.
\]  

(10.12)

Matlab provides the commands `fft` and `ifft`. The Matlab command `fft(x)` multiplies the input vector \( x \) by the Fourier matrix. The result must be multiplied by \( 1/\sqrt{n} \) to get the DFT.
Likewise, the inverse DFT is given by the Matlab command \( \sqrt{n} \cdot \text{ifft}(x) \). In other words, Matlab’s \texttt{fft} and \texttt{ifft} commands are inverses of each other, although their normalization differs from our definition.

Even if the vector \( x \) has components that are real numbers, there is no reason for the components of \( y \) to be real numbers. But if the \( x_j \) are real, the complex numbers \( y_k \) have a special property:

**Lemma 10.3** Let \( \{y_k\} \) be the DFT of \( \{x_j\} \), where the \( x_j \) are real numbers. Then (a) \( y_0 \) is real, and (b) \( y_{n-k} = \overline{y}_k \) for \( k = 1, \ldots, n-1 \).

The reason for (a) is clear from the definition (10.7) of DFT; \( y_0 \) is the average of the \( x_j \)'s. Part (b) follows from the fact that

\[
\omega^{n-k} = e^{-i2\pi(n-k)/n} = e^{-i2\pi e^{i2\pi k/n}} = \cos(k/n) + i \sin(k/n),
\]

while

\[
\omega^k = e^{-i2\pi k/n} = \cos(k/n) - i \sin(k/n),
\]

implying that \( \omega^{n-k} = \overline{\omega^k} \). From the definition of Fourier transform,

\[
y_{n-k} = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j (\omega^{n-k})^j
\]

\[
= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j (\omega^k)^j
\]

\[
= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} \overline{x}_j (\overline{\omega^k})^j = \overline{y}_k.
\]

Here we have used the fact that the product of complex conjugates is the conjugate of the product.

\[\square\]

Lemma 10.3 has an interesting consequence. If the \( x_0, \ldots, x_{n-1} \) are real numbers, the Fourier transform DFT replaces them with exactly \( n \) other real numbers \( a_0, a_1, b_1, a_2, b_2, \ldots, a_{n/2}, \) the real and imaginary parts of the Fourier transform \( y_0, \ldots, y_{n-1} \). For example, the \( n = 8 \) DFT has form

\[
F_8 = \begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
\end{bmatrix}
= \begin{bmatrix}
a_0 \\
a_1 + i b_1 \\
a_2 + i b_2 \\
a_3 + i b_3 \\
a_4 \\
a_3 - i b_3 \\
a_2 - i b_2 \\
a_1 - i b_1 \\
\end{bmatrix}
= \begin{bmatrix}
y_0 \\
\vdots \\
y_{n-1} \\
\end{bmatrix},
\]

\[\text{(10.13)}\]
The achievement of Cooley and Tukey to reduce the complexity of the FFT from $O(n^2)$ operations to $O(n \log n)$ operations opened up a world of possibilities for Fourier transform methods. A method that scales “almost linearly” with the size of the problem is very valuable. For example, there is a possibility of using it for stream data, since analysis can occur approximately at the same time scale that data is acquired. The development of the FFT was followed a short time later with specialized circuitry for implementing it, now represented by DSP chips for digital signal processing, that are ubiquitous in electronic systems for analysis and control. In addition, the compression techniques of Chapter 11, while they could be done without the FFT, are much more powerful for its existence.

10.1.3 The Fast Fourier Transform

As mentioned in the last section, the Discrete Fourier Transform applied to an $n$-vector in the traditional way requires $O(n^2)$ operations. Cooley and Tukey found a way to accomplish the DFT in $O(n \log n)$ operations. When it was published, it led very fast to popularity of the FFT for data analysis. The field of signal processing went largely from analog to digital in a short time in part due to this algorithm. We will explain their method and show its superiority to the naive DFT through an operation count.

We start by showing how the $n = 6$ case works, to get the main idea across. The general case will then be clear. Let $\omega = e^{-2\pi i/6}$. The Fourier matrix is:

$$
\begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
  y_5
\end{bmatrix} =
\begin{bmatrix}
  \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\
  \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 \\
  \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} \\
  \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} \\
  \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} \\
  \omega^0 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25}
\end{bmatrix}
\begin{bmatrix}
  x_0 \\
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5
\end{bmatrix}.
$$

(10.14)

Write out the matrix product, but rearrange the order of the terms so that the even-numbered terms come first.

$$
y_0 = \omega^0 x_0 + \omega^0 x_2 + \omega^0 x_4 + \omega^0 (\omega^0 x_1 + \omega^0 x_3 + \omega^0 x_5)\\
y_1 = \omega^0 x_0 + \omega^2 x_2 + \omega^4 x_4 + \omega^1 (\omega^0 x_1 + \omega^2 x_3 + \omega^4 x_5)\\
y_2 = \omega^0 x_0 + \omega^4 x_2 + \omega^8 x_4 + \omega^2 (\omega^0 x_1 + \omega^4 x_3 + \omega^8 x_5)\\
y_3 = \omega^0 x_0 + \omega^6 x_2 + \omega^{12} x_4 + \omega^3 (\omega^0 x_1 + \omega^6 x_3 + \omega^{12} x_5)\\
y_4 = \omega^0 x_0 + \omega^8 x_2 + \omega^{16} x_4 + \omega^4 (\omega^0 x_1 + \omega^8 x_3 + \omega^{16} x_5)\\
y_5 = \omega^0 x_0 + \omega^{10} x_2 + \omega^{20} x_4 + \omega^5 (\omega^0 x_1 + \omega^{10} x_3 + \omega^{20} x_5)
$$

Using the fact that $\omega^6 = 1$, we can rewrite as

\[
\begin{align*}
y_0 &= (\omega^0 x_0 + \omega^0 x_1 + \omega^0 x_2) + \omega^0 (\omega^0 x_3 + \omega^0 x_4 + \omega^0 x_5) \\
y_1 &= (\omega^0 x_0 + \omega^2 x_1 + \omega^4 x_2) + \omega^1 (\omega^0 x_3 + \omega^2 x_4 + \omega^4 x_5) \\
y_2 &= (\omega^0 x_0 + \omega^0 x_1 + \omega^4 x_2) + \omega^2 (\omega^0 x_3 + \omega^4 x_4 + \omega^4 x_5) \\
y_3 &= (\omega^0 x_0 + \omega^0 x_1 + \omega^2 x_2 + \omega^4 x_3) + \omega^3 (\omega^0 x_4 + \omega^0 x_5 + \omega^0 x_5) \\
y_4 &= (\omega^0 x_0 + \omega^2 x_1 + \omega^4 x_2) + \omega^4 (\omega^0 x_3 + \omega^2 x_4 + \omega^4 x_5) \\
y_5 &= (\omega^0 x_0 + \omega^4 x_1 + \omega^8 x_2) + \omega^5 (\omega^0 x_4 + \omega^4 x_5 + \omega^8 x_5)
\end{align*}
\]

Notice that each term in parentheses in the top three lines is repeated verbatim in the lower three lines. Define

\[
\begin{align*}
u_0 &= \mu^0 x_0 + \mu^0 x_2 + \mu^0 x_4 \\
u_1 &= \mu^0 x_0 + \mu^1 x_2 + \mu^2 x_4 \\
u_2 &= \mu^0 x_0 + \mu^2 x_2 + \mu^4 x_4
\end{align*}
\]

and

\[
\begin{align*}
v_0 &= \mu^0 x_1 + \mu^0 x_3 + \mu^0 x_5 \\
v_1 &= \mu^0 x_1 + \mu^1 x_3 + \mu^2 x_5 \\
v_2 &= \mu^0 x_1 + \mu^2 x_3 + \mu^4 x_5
\end{align*}
\]

where $\mu = \omega^2$ is a 3rd root of unity. Both $u = (u_0, u_1, u_2)^T$ and $v = (v_0, v_1, v_2)^T$ are DFT(3)’s. We can write the original DFT(6) as

\[
\begin{align*}
y_0 &= u_0 + \omega^0 v_0 \\
y_0 &= u_1 + \omega^1 v_1 \\
y_0 &= u_2 + \omega^2 v_2 \\
y_0 &= u_0 + \omega^3 v_0 \\
y_0 &= u_1 + \omega^4 v_1 \\
y_0 &= u_2 + \omega^5 v_2.
\end{align*}
\]

In summary, the calculation of the DFT(6) has been reduced to a pair of DFT(3)’s plus some extra multiplications and additions.

In the general case, a DFT(n) can be reduced to computing two DFT(n/2)’s plus 2n-1 extra flops ($n - 1$ multiplications and $n$ additions). A careful count of the additions and multiplications necessary from the above code yields the following.

---

**Fact 10.1**

**Operation Count for FFT.** Let $n$ be a power of 2. Then the Fast Fourier Transform of size $n$ can be completed in $n(2 \log_2 n - 1) + 1$ flops.
Chapter 10  **Trigonometric Interpolation and the FFT**  379

Fact 10.1 is equivalent to saying that the DFT($2^m$) can be completed in $2^m(2m-1)+1$ additions and multiplications. In fact, we saw above how a DFT(n), where $n$ is even, can be reduced to a pair of DFT(n/2)'s. If $n$ is a power of two, say $n = 2^m$, then we can recursively break down the problem until we get to DFT(1), which is multiplication by the $1 \times 1$ identity matrix, which takes zero operations! Starting from the bottom up, DFT(1) takes no operations, and DFT(2) requires two additions and a multiplication: $y_0 = u_0 + 1v_0, y_1 = u_0 + \omega v_0$, where $u_0$ and $v_0$ are DFT(1)'s, that is $u_0 = y_0$ and $v_0 = y_1$.

DFT(4) requires two DFT(2)'s plus $2 \times 4 - 1 = 7$ further operations, for a total of $2(3) + 7 = 2^m(2m - 1) + 1$ operations, where $m = 2$. Assume this formula is correct for a given $m$. Then DFT($2^m+1$) takes two DFT($2^m$)'s, which take $2(2^m(2m - 1) + 1)$ operations, plus $2 \cdot 2^{m+1} - 1$ extras (to complete equations similar to (10.15)), for a total of

$$2(2^m(2m - 1) + 1) + 2^{m+2} - 1 = 2^{m+1}(2m - 1 + 2) + 2 - 1 = 2^{m+1}(2(m + 1) - 1) + 1.$$  

Therefore the formula $2^m(2m - 1) + 1$ flops is proved for the fast version of DFT($2^m$), from which Fact 10.1 follows.

The fast algorithm for the DFT can be exploited to make a fast algorithm for the inverse DFT without further work. The inverse DFT is $(1/n)F_n$, where the overbar stands for complex conjugate. To carry out the inverse DFT $F^{-1}$ of a complex vector $y$, just conjugate, apply the FFT, and conjugate again and divide by $n$, because

$$F^{-1}y = \frac{1}{n}F_n y = \frac{1}{n}\overline{F_n \overline{y}}$$  

(10.16)

**Exercises 10.1**

10.1.1. (a) Write down all 4th roots of unity and all primitive 4th roots of unity. (b) Write down all primitive 7th roots of unity. (c) How many primitive $p$th roots of unity exist, for $p$ a prime number?

10.1.2. Find the DFT of the following vectors.

(a) $x = (0, 1, 0, -1)$.
(b) $x = (1, 1, 1, 1)$.
(c) $x = (0, -1, 0, 1)$.
(d) $x = (0, 1, 0, -1, 0, 1, 0, -1)$.

10.1.3. Find the real numbers $a_0, a_1, b_1, a_2, b_2, \ldots , a_{2n}$ as in (10.13) for the Fourier transforms in Exercise 10.1.2.

10.1.4. Find the DFT of the following vectors.

(a) $x = (3/4, 1/4, -1/4, 1/4)$.
(b) $x = (9/4, 1/4, -3/4, 1/4)$.
(c) $x = (1, 0, -1/2, 0)$.
(d) $x = (1, 0, -1/2, 0, 1, 0, -1/2, 0)$.
(e) $x = (1/2, 0, 0, 0, 1/2, 0, 0, 0)$.

10.1.5. Prove Lemma 10.1.

10.1.6. Prove that the matrix in (10.10) is the inverse of the Fourier matrix $F_n$. 


10.2 Trigonometric interpolation.

What does the discrete Fourier transform actually do? In this section we present an interpretation of the output vector \( y \) of the Fourier transform in order to make its mysterious workings a little more understandable.

10.2.1 The DFT Interpolation Theorem

To be concrete, we restrict attention to the unit interval \([0, 1]\). Fix a positive integer \( n \) and define \( t_j = j/n \) for \( j = 0, \ldots, n-1 \). For a given input vector \( x \) to the Fourier transform, we will interpret the component \( x_j \) as the \( j \)th component of a measured signal. For example, we could think of the components of \( x \) as a time series of measurements, measured at the discrete, equally-spaced times \( t_j \), as shown in Figure 10.4.

![Figure 10.4: The components of \( x \) viewed as a time series.](image)

Let \( y = F_n x \) be the DFT of \( x \). Since \( x \) is the inverse DFT of \( y \), we can write an explicit formula for the components of \( x \) from (10.10), remembering that \( \omega = e^{-i 2\pi/n} \):

\[
x_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} y_k \omega^{-k j} = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} y_k e^{2\pi i k (j/n)} = \sum_{k=0}^{n-1} \frac{y_k}{\sqrt{n}} e^{i 2\pi k t_j}.
\]  

(10.17)

We can view this as interpolation of the points \((t_j, x_j)\) by the “trigonometric” basis functions \( e^{i 2\pi k t_j} / \sqrt{n} \), where the coefficients are \( y_k \). Theorem 10.4 is a simple restatement of (10.17), saying that data points \((t_j, x_j)\) are interpolated by basis functions \( \left\{ \frac{1}{\sqrt{n}} e^{i 2\pi k t_j} \right\}_{k = 0, \ldots, n-1} \), with interpolation coefficients given by \( F_n x \).
Theorem 10.4 DFT Interpolation Theorem. Let \( t_j = j/n \) for \( j = 0, \ldots, n - 1 \) and let \( x = (x_0, \ldots, x_{n-1}) \) denote a vector of \( n \) numbers. Define \( \vec{a} + \vec{b}i = F_n x \), where \( F_n \) is the discrete Fourier transform. Then the complex function 

\[
Q(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k)e^{i2\pi kt}
\]

satisfies \( Q(t_j) = x_j \) for \( j = 0, \ldots, n - 1 \). Furthermore, if the \( x_j \) are real, the real function 

\[
P_n(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k \cos 2\pi kt - b_k \sin 2\pi kt)
\]

satisfies \( P(t_j) = x_j \) for \( j = 0, \ldots, n - 1 \).

In other words, the Fourier transform \( F_n \) transforms data into interpolation coefficients.

The explanation for the last part of the theorem is that using Euler’s formula, the interpolation function in (10.17) can be rewritten as 

\[
Q(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k)(\cos 2\pi kt + i \sin 2\pi kt).
\]

Write the interpolating function \( Q(t) = P(t) + iI(t) \) in its real and imaginary parts. Since the \( x_j \) are real numbers, only the real part of \( Q(t) \) is needed to interpolate the \( x_j \). The real part is 

\[
P(t) = P_n(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k \cos 2\pi kt - b_k \sin 2\pi kt).
\]

(10.18)

A subscript \( n \) identifies the number of terms in the trigonometric model. We will sometimes call \( P_n \) an order \( n \) trigonometric function. Lemma 10.3 and the following Lemma 10.5 can be used to simplify the interpolating function \( P_n(t) \) further.

Lemma 10.5 Let \( t = j/n \) where \( j \) and \( n \) are integers. Let \( k \) be an integer. Then 

\[
\cos 2(n-k)\pi t = \cos 2k\pi t \quad \text{and} \quad \sin 2(n-k)\pi t = -\sin 2k\pi t.
\]

(10.19)

In fact, the cosine addition formula yields 

\[
\cos 2(n-k)\pi j/n = \cos(2\pi j - 2j\pi/n) = \cos(-2j\pi/n) = \cos(2k\pi j/n),
\]

and similarly for \( \sin \).

Lemma 10.5 says that the latter half of the trig expansion (10.18) is redundant. We can interpolate at the \( t_j \)’s using only the first half of the terms (except for a change of sign for the \( \sin \) terms).

By Lemma 10.3, the coefficients from the latter half of the expansion are the same as those from the first half (except for a change of sign for the \( \sin \) terms). Thus the changes of sign cancel one another out and we have shown that the simplified version of \( P_n \) is 

\[
P_n(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} (a_k \cos 2k\pi t - b_k \sin 2k\pi t) + \frac{a_{n/2}}{\sqrt{n}} \cos n\pi t.
\]
We have assumed \( n \) is even to write this expression. The formula is slightly different for \( n \) odd. See Exercise 10.2.6.

**Corollary 10.6** Let \( t_j = j/n \) for \( j = 0, \ldots, n - 1 \), \( n \) even, and let \( x = (x_0, \ldots, x_{n-1}) \) denote a vector of \( n \) real numbers. Define \( \vec{a} + \vec{b}i = F_n x \), where \( F_n \) is the discrete Fourier transform. Then the function

\[
P_n(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} (a_k \cos 2k\pi t - b_k \sin 2k\pi t) + \frac{a_{n/2}}{\sqrt{n}} \cos n\pi t.
\]

(10.20)

satisfies \( P_n(t_j) = x_j \) for \( j = 0, \ldots, n - 1 \).

**Example 10.2** Find the trig interpolant for Example 10.1.

Returning to our previous example, let \( x = (1, 0, -1, 0)^T \) and compute its DFT to be \( y = (0, 1, 0, 1)^T \). The interpolating coefficients are \( a_k + ib_k = y_k \). Therefore \( a_0 = a_2 = 0, a_1 = a_3 = 1, \) and \( b_0 = b_1 = b_2 = b_3 = 0 \). According to (10.20), we only need \( a_0, a_1, a_2 \) and \( b_1 \). A trig interpolating function for \( x \) is given by

\[
P_4(t) = \frac{a_0}{2} + (a_1 \cos 2\pi t - b_1 \sin 2\pi t) + \frac{a_2}{2} \cos 4\pi t
\]

(10.21)

**Example 10.3** Find the trig interpolant for \( x = (.3, .6, .8, .5, .6, .2, .3)^T \).

The Fourier transform, accurate to 4 decimal places, is

\[
y = \begin{bmatrix}
  +1.3081 \\
  -0.1061 - 0.3121i \\
  -0.0354 - 0.0707i \\
  -0.1061 + 0.1121i \\
  +0.0354 \\
  -0.1061 - 0.1121i \\
  -0.0354 + 0.0707i \\
  -0.1061 + 0.3121i
\end{bmatrix}.
\]

According to the formula (10.20), the interpolating function is

\[
P_8(t) = 0.4625 + -0.0750 \cos 2\pi t + 0.2207 \sin 2\pi t
- 0.0250 \cos 4\pi t + 0.0500 \sin 4\pi t
+ 0.0750 \cos 6\pi t - 0.0793 \sin 6\pi t
+ 0.0125 \cos 8\pi t
\]

(10.21)

Figure 10.5 shows the data points \( x \) and the interpolating function.
Figure 10.5: Trigonometric interpolation (a) The input vector $x$ is $(1, 0, -1, 0)^T$. Formula (10.20) gives the interpolating function to be $P_0(t) = \cos 2\pi t$. (b) Example 10.3. The input vector $x$ is $(.3, .6, .8, .5, .6, .2, .3)^T$. Formula (10.21) gives the interpolating function.

There is another way to evaluate and plot the trig interpolating polynomial in Figure 10.5, using the DFT to do all the work instead of plotting the sines and cosines of (10.20). After all, we know from Theorem 10.4 that multiplying the vector $x$ of data points by $F_n$ changes data to interpolation coefficients. Conversely, we can turn interpolation coefficients into data points. Instead of evaluating (10.20), just invert the DFT – multiply the vector of interpolation coefficients $\{a_k + ib_k\}$ by $F_n^{-1}$.

Of course, if we follow the operation $F_n$ by its inverse, $F_n^{-1}$, we just get the original data points back, and gain nothing. Instead, let $p \geq n$ be a larger number. We plan to view (10.20) as an order $p$ trigonometric polynomial, and then invert the Fourier transform to get the $p$ equally-spaced points on the curve. We can take $p$ large enough to get a continuous-looking plot.

In order to view the coefficients of $P_n(t)$ as the coefficients of an order $p$ trigonometric polynomial, we need to write

$$P_p(t) = P_n(t) = \frac{\sqrt{p/a_0}}{\sqrt{p}} + 2 \frac{n/2-1}{\sqrt{p}} \sum_{k=1}^{n/2-1} \left( \sqrt{\frac{p}{n}} a_k \cos 2k\pi t - \sqrt{\frac{p}{n}} b_k \sin 2k\pi t \right) + \frac{\sqrt{p/a_n/2}}{\sqrt{p}} \cos n\pi t.$$  

Therefore, the way to produce $p$ equally-spaced points lying on the curve (10.20) is to multiply the Fourier coefficients by $\sqrt{p/n}$ and then invert the DFT. We write Matlab code to implement this idea. Roughly speaking, we want to implement

$$F_p^{-1} \sqrt{\frac{p}{n}} F_n x,$$

where comparing with Matlab’s commands,

$$F_p^{-1} = \sqrt{p} \cdot \text{ifft} \text{ and } F_n = \frac{1}{\sqrt{n}} \cdot \text{fft}.$$
Putting together, this corresponds to the operations

\[
\sqrt{p} \cdot \text{ifft}[p] \sqrt{\frac{1}{n}} \cdot \text{fft}[n] = \frac{p}{n} \cdot \text{ifft}[p] \cdot \text{fft}[n].
\]

Of course, \( F_p^{-1} \) can only be applied to a length \( p \) vector, so we need to place the degree \( n \) Fourier coefficients into a length \( p \) vector before inverting. The short program dftinterp.m carries out these steps.

```matlab
% Program 10.1 Fourier interpolation
% Interpolate \( n \) data points on \([0,1]\) with trig function \( P(t) \)
% and plot interpolant at \( p (\geq n) \) evenly-spaced points.
% Input \( n, p, \) data points \( x \)
function xp = dftinterp(n,p,x)
t = (0:n-1)/n; tp = (0:p-1)/p; % time pts for data (n) and interpolant (p)
y = fft(x); % compute interpolation coefficients
yp = zeros(p,1); % will hold coefficients for ifft
yp(1:n/2+1) = y(1:n/2+1); % move \( n \) frequencies from \( n \) to \( p \)
yp(p-n/2+2:p) = y(n/2+2:n); % same for upper tier
xp = real(ifft(yp))*(p/n); % invert fft to recover data from interp coeffs
plot(t,x,'o',tp,xp) % plot original data points and interpolant
```

Running the function `dftinterp(8,100,[.3 .6 .8 .5 .6 .4 .2 .3])`, for example, produces the \( p = 100 \) plotted points in Figure 10.5. A few comments on the code are in order. The goal is to move the coefficients in the vector \( y \) from the \( n \) frequencies in \( P_n(t) \) over to a vector \( yp \) holding \( p \) frequencies, where \( p \geq n \). There are many higher frequencies among the \( p \) frequencies that are not used by \( P_n \), which leads to zeros in those high frequencies, in positions \( n/2 + 2 \) to \( p/2 + 1 \). The upper half of the entries in \( yp \) gives a recapitulation of the lower half, with complex conjugates. After inverting the DFT with the \text{ifft} \ command, although theoretically the result is real, computationally there may be a small imaginary part due to rounding. This is removed by applying the \text{real} \ command.

The interpolation formula (10.20) was restricted to the domain \([0,1]\). To interpolate evenly-spaced data points on an arbitrary interval \([c,d]\), define the points \( s_j = c + j(d-c)/n \) for \( j = 0,\ldots,n-1 \), and denote the data points by \((s_j, x_j)\). We can put this in the earlier context by solving for \( t_j = j/n = (s_j - c)/(d-c) \). Assume \( n \) is even, as usual. Let \( y \) be the DFT of \( x \), and define \( a_k + ib_k = y_k \). Then the function

\[
P_n(s) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left( a_k \cos \frac{2k\pi(s-c)}{d-c} - b_k \sin \frac{2k\pi(s-c)}{d-c} \right) + \frac{a_{n/2}}{\sqrt{n}} \cos \frac{n\pi(s-c)}{d-c}
\]

interpolates the points \((s_j, x_j)\) on \([c,d]\).

A particularly simple and useful case is \( c = 0, d = n \). The data points \( x_j \) are collected at the integer interpolation nodes \( s_j = j \) for \( j = 0,\ldots,n-1 \). The points \((j,x_j)\) are interpolated by the trigonometric function

\[
P_n(s) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left( a_k \cos \frac{2k\pi}{n}s - b_k \sin \frac{2k\pi}{n}s \right) + \frac{a_{n/2}}{\sqrt{n}} \cos \pi s
\]
10.2.2 Orthogonality and interpolation

The DFT Interpolation Theorem 10.4 is just one special case of an extremely useful method. In this section we look at interpolation from a more general point of view, which will in addition show how to find least squares approximations using trigonometric functions. These ideas will also come in handy in Chapter 11, where it will apply to the discrete cosine transform.

The deceptively simple interpolation result of the theorem was made possible by the fact that $F_n^{-1} = F_n^T$. A matrix whose inverse is its conjugate transpose is called a **unitary** matrix. If furthermore the matrix is real, there is a more specialized term. We say that a square matrix $A$ with real entries is **orthogonal** if $A^{-1} = A^T$. Note that this implies $A^T A = A A^T = I$. In other words, the rows of an orthogonal matrix are pairwise orthogonal unit vectors, and the same is true of its columns. For example, any permutation matrix is orthogonal, as is the $2 \times 2$ matrix

$$
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
$$

for any angle $\theta$.

Now we study a particular form for an orthogonal matrix that will translate immediately into a good interpolator.

**Assumption 10.1** Let $f_0(t), \ldots, f_{n-1}(t)$ be functions of $t$ and $t_0, \ldots, t_{n-1}$ be real numbers. Assume that $A$ is a real $n \times n$ orthogonal matrix of the form

$$
A = \begin{bmatrix}
f_0(t_0) & f_0(t_1) & \cdots & f_0(t_{n-1}) \\
f_1(t_0) & f_1(t_1) & \cdots & f_1(t_{n-1}) \\
\vdots & \vdots & \ddots & \vdots \\
f_{n-1}(t_0) & f_{n-1}(t_1) & \cdots & f_{n-1}(t_{n-1})
\end{bmatrix}.
$$

(10.24)

**Example 10.4** Let $n$ be a positive even integer. Show that Assumption 10.1 is satisfied for $t_k = k/n$, and

$$
\begin{align*}
f_0(t) &= \frac{1}{\sqrt{n}} \\
f_1(t) &= \sqrt{\frac{2}{n}} \cos 2\pi t \\
f_2(t) &= \sqrt{\frac{2}{n}} \sin 2\pi t \\
f_3(t) &= \sqrt{\frac{2}{n}} \cos 4\pi t \\
f_4(t) &= \sqrt{\frac{2}{n}} \sin 4\pi t \\
\vdots \\
f_{n-1}(t) &= \frac{1}{\sqrt{n}} \cos n\pi t
\end{align*}
$$
The matrix is

\[
A = \sqrt{\frac{2}{n}} \begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\
1 & \cos \frac{2\pi}{n} & \cdots & \cos \frac{2\pi(n-1)}{n} \\
0 & \sin \frac{2\pi}{n} & \cdots & \sin \frac{2\pi(n-1)}{n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \cos \pi t & \cdots & \frac{1}{\sqrt{2}} \cos (n - 1)\pi 
\end{bmatrix}
\]  

(10.25)

Lemma 10.7 shows that the rows of \( A \) are pairwise orthogonal.

\[\square\]

**Lemma 10.7** Let \( n \geq 1 \) and \( k, l \) be integers.

\[
\begin{align*}
\sum_{j=0}^{n-1} \cos \frac{2\pi jk}{n} \cos \frac{2\pi jl}{n} &= \begin{cases} 
n & \text{if both } (k - l)/n \text{ and } (k + l)/n \text{ are integers} \\
\frac{n}{2} & \text{if exactly one of } (k - l)/n \text{ and } (k + l)/n \text{ is an integer} \\
0 & \text{if neither is an integer}
\end{cases} \\
\sum_{j=0}^{n-1} \cos \frac{2\pi jk}{n} \sin \frac{2\pi jl}{n} &= 0 \\
\sum_{j=0}^{n-1} \sin \frac{2\pi jk}{n} \sin \frac{2\pi jl}{n} &= \begin{cases} 
n & \text{if both } (k - l)/n \text{ and } (k + l)/n \text{ are integers} \\
\frac{n}{2} & \text{if } (k - l)/n \text{ is an integer and } (k + l)/n \text{ is not} \\
-\frac{n}{2} & \text{if } (k + l)/n \text{ is an integer and } (k - l)/n \text{ is not} \\
0 & \text{if neither is an integer}
\end{cases}
\]

The proof of this lemma follows from Lemma 10.1. See Exercise 10.2.7.

In this situation, interpolation of the data points \((t_k, x_k)\) using the basis functions \(f_j(t)\) is readily computable. Let

\[
y = Ax
\]

be the transform variable; then inverting the \( A \)-transform is a fact about interpolation:

\[
x = A^{-1} y = \sum_{j=0}^{n-1} a_{kj} y_j = \sum_{j=0}^{n-1} y_j f_j(t_k) \quad \text{for } k = 0, \ldots, n - 1.
\]

We have proved:

**Theorem 10.8** Orthogonal Function Interpolation Theorem. Under Assumption 10.1, with \( y = Ax \), the function

\[F(t) = \sum_{j=0}^{n-1} y_j f_j(t)\]

interpolates \((t_0, x_0), \ldots, (t_{n-1}, x_{n-1})\), i.e. \(F(t_k) = x_k\) for \(k = 0, \ldots, n - 1\).
Returning to Example 10.4, let \( y = Ax \). Theorem 10.8 immediately gives the interpolating function

\[
F(t) = \frac{1}{\sqrt{n}}y_0 + \sqrt{\frac{2}{n}}y_1 \cos 2\pi t + \sqrt{\frac{2}{n}}y_2 \sin 2\pi t + \sqrt{\frac{2}{n}}y_3 \cos 4\pi t + \sqrt{\frac{2}{n}}y_4 \sin 4\pi t + \ldots + \frac{1}{\sqrt{n}}y_{n-1} \cos n\pi t
\]

for the points \((k/n, x_k)\), just as in (10.20).

**Example 10.5** Use Theorem 10.8 to interpolate the data points \( x = (.3, .6, .8, 1.5, 1.6, .4, .2, .3)^T \) from Example 10.3.

Computing the product of the \(8 \times 8\) matrix \( A \) with \( x \) yields

\[
Ax = \sqrt{\frac{2}{n}} \begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \ldots & \frac{1}{\sqrt{2}} \\
1 & \cos 2\pi \frac{1}{8} & \cos 2\pi \frac{3}{8} & \ldots & \cos 2\pi \frac{7}{8} \\
0 & \sin 2\pi \frac{1}{8} & \sin 2\pi \frac{3}{8} & \ldots & \sin 2\pi \frac{7}{8} \\
1 & \cos 4\pi \frac{1}{8} & \cos 4\pi \frac{3}{8} & \ldots & \cos 4\pi \frac{7}{8} \\
0 & \sin 4\pi \frac{1}{8} & \sin 4\pi \frac{3}{8} & \ldots & \sin 4\pi \frac{7}{8} \\
1 & \cos 6\pi \frac{1}{8} & \cos 6\pi \frac{3}{8} & \ldots & \cos 6\pi \frac{7}{8} \\
0 & \sin 6\pi \frac{1}{8} & \sin 6\pi \frac{3}{8} & \ldots & \sin 6\pi \frac{7}{8} \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \ldots & \frac{1}{\sqrt{2}}
\end{bmatrix} \begin{bmatrix} 0.3 \\ 0.6 \\ 0.8 \\ 0.5 \\ 0.6 \\ 0.4 \\ 0.2 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 1.3081 \\ -0.1500 \\ 0.4414 \\ -0.0500 \\ 0.1000 \\ -0.1500 \\ -0.1586 \\ 0.0354 \end{bmatrix}.
\]

Theorem 10.8 gives the interpolating function

\[
P(t) = 0.4625 - 0.075 \cos 2\pi t + 0.2207 \sin 2\pi t - 0.025 \cos 4\pi t + 0.05 \sin 4\pi t - 0.075 \cos 6\pi t - 0.0793 \sin 6\pi t + 0.0125 \cos 8\pi t
\]

in agreement with Example 10.3.

\[
\]

**10.2.3 Least squares fitting with trigonometric functions**

The previous section showed how the DFT makes it easy to interpolate \( n \) evenly-spaced data points on \([0, 1]\) by a trigonometric function of form

\[
P_n(t) = \frac{a_0}{\sqrt{n}} + \sum_{k=1}^{n/2-1} \left( \frac{a_k}{\sqrt{n}} \cos 2k\pi t - \frac{b_k}{\sqrt{n}} \sin 2k\pi t \right) + \frac{a_{n/2}}{\sqrt{n}} \cos n\pi t. \tag{10.27}
\]
Spotlight on: Orthogonality

In Chapter 4, we established the normal equations $A^T A = A^T b$ for solving least squares approximation to data by basis functions. The points of Assumption 10.1 is to find special cases that make the normal equations trivial, greatly simplifying the least squares procedure. This leads to an extremely useful theory of so-called orthogonal functions that we barely touch upon in this book. We will concentrate on two major examples, generated by the Fourier transform in this Chapter and the Cosine transform in Chapter 11.

Note that the number of terms is $n$, equal to the number of data points. (As usual in this chapter, we assume $n$ is even.) The more data points there are, the more cosines and sines are added to help with the interpolation.

As we found in Chapter 3, when the number of data points $n$ is large, it becomes less common to want to fit a model function exactly. In fact, a typical purpose of a model is to forget a few details (lossy compression!) in order to simplify matters. A second reason to move away from interpolation, discussed in Chapter 4, is the case where the data points themselves are assumed to be inexact, so that rigorous enforcement of an interpolating function is inappropriate.

A better choice in either of these situations is to do a least squares fit with a function of type (10.27). Since the coefficients $a_k$ and $b_k$ occur linearly in the model, we can proceed with the same program described in Chapter 4, using the normal equations to solve for the best coefficients. However, when we try this we find a surprising result, which will send us right back to the DFT!

Return to Assumption 10.1. We continue to use $n$ to denote the number of data points $x_j$, which we think of as occurring at evenly-spaced times $t_j = j/n$ in $[0, 1]$, the same as before. We will introduce the even positive integer $m$ to denote the number of basis functions to use in the least squares fit. That is, we will fit to the first $m$ of the basis functions, $f_1(t), \ldots, f_m(t)$. The function used to fit the $n$ data points will be

$$P_m(t) = \sum_{j=0}^{m-1} c_j f_j(t),$$

where the $c_k$ are to be determined. When $m = n$, the problem is still interpolation. When $m < n$, we have changed to the compression problem. In this case, we expect to miss the data points with $P_m$, but by as little as possible – with minimum possible squared error.

The least squares problem is to find coefficients $c_0, \ldots, c_{n-1}$ such that the equality

$$\sum_{j=0}^{n-1} c_j f_j(t_k) = x_k$$

is met with as little error as possible. In matrix terms,

$$A_{m}^T c = x,$$
where the $A_m$ is the matrix of the first $m$ rows of $A$. By Assumption 10.1, $A_m^T$ has pairwise orthonormal columns. When we solve the normal equations

$$A_mA_m^T \vec{c} = A_mx$$

for $c$, $A_mA_m^T$ is the identity matrix. Therefore

$$\vec{c} = A_mx$$

is easy to calculate. We have proved the following useful result, which extends Theorem 10.8.

**Theorem 10.9 Orthogonal Function Least Squares Approximation Theorem.** Let $m \leq n$ be integers, and assume data $(t_0, x_0), \ldots, (t_{n-1}, x_{n-1})$ are given. Under Assumption 10.1 with $y = Ax$, the interpolating polynomial for basis functions $f_0(t), \ldots, f_{n-1}(t)$ is

$$F_n(t) = \sum_{j=0}^{n-1} y_j f_j(t)$$

and the best least squares approximation using the first $m$ functions only is

$$F_m(t) = \sum_{j=0}^{m-1} y_j f_j(t)$$

This is an amazing and useful fact. It says that given $n$ data points, to find the best least squares trigonometric function with $m < n$ terms fitting the data, it suffices to compute the actual interpolant with $n$ terms, and keep only the first $m$ terms. In other words, the interpolating coefficients $Ax$ for $x$ degrade as gracefully as possible when terms are dropped from the highest frequencies. Keeping the $m$ lowest terms in the $n$-term expansion guarantees the best fit possible with $m$ terms. This property of function satisfying Assumption 10.1 reflects the “orthogonality” of the basis functions.

Note that the reasoning preceding Theorem 10.9 actually proves something more general. We showed how to find the least squares solution for the first $m$ basis functions, but in truth we could have specified any subset of the basis functions. The least squares solution is found simply by dropping all terms in (10.31) that are not included in the subset. As written in (10.32), it is a “low-pass” filter, assuming the lower index functions go with lower “frequencies”, but by changing the subset of basis functions kept, one can pass any frequencies of interest, by simply dropping the undesired coefficients.

Now we return to the trigonometric polynomial (10.27) and ask how to fit an order $m$ version to $n$ data points, where $m < n$. Note that the basis functions used are the functions of Example 10.4, for which we know Assumption 10.1 is satisfied. Theorem 10.9 shows that whatever the interpolating coefficients, the coefficients of the best least squares approximation of order $m$ are found by dropping all terms above order $m$. We have arrived at the following application.
Corollary 10.10 Let \( m \leq n \) be even positive integers, \( x = (x_0, \ldots, x_{n-1}) \) a vector of \( n \) real numbers, and let \( t_j = j/n \) for \( j = 0, \ldots, n-1 \). Let \( \{a_0, a_1, b_1, a_2, b_2, \ldots, a_{n/2-1}, b_{n/2-1}, a_{n/2}\} = F_n x \) be the interpolating coefficients for \( x \), i.e. such that

\[
x_j = P_n(t_j) = a_0 \sqrt{n} + \frac{2}{\sqrt{n}} \sum_{k=1}^{\frac{n}{2}-1} (a_k \cos 2k\pi t_j - b_k \sin 2k\pi t_j) + \frac{a_{n/2}}{\sqrt{n}} \cos \frac{n\pi t_j}{2}
\]

for \( j = 0, \ldots, n-1 \). Then

\[
P_m(t) = a_0 \sqrt{n} + \frac{2}{\sqrt{n}} \sum_{k=1}^{\frac{m}{2}-1} (a_k \cos 2k\pi t - b_k \sin 2k\pi t) + \frac{a_{m/2}}{\sqrt{n}} \cos m\pi t
\]

is the best least squares fit of order \( m \) to the data \( (t_j, x_j) \) for \( j = 0, \ldots, n-1 \).

Another way of appreciating the power of Theorem 10.9 is to compare it with the monomial basis functions we have used previously for least squares models. The best least squares parabola fit to the points \((0,3), (1,3), (2,5)\) is \( y = x^2 - x + 3 \). In other words, the best coefficients for the model \( y = a + bx + cx^2 \) for this data are \( a = 3, b = -1 \) and \( c = 1 \) (in this case because the squared error is zero – this is the interpolating parabola). Now let’s fit to a subset of the basis functions, say change the model to \( y = a + bx \). We calculate the best line fit to be \( a = 8/3, b = 1 \). Note that the coefficients for the degree one fit have no apparent relation to their corresponding coefficients for the degree two fit. This is exactly what doesn’t happen for trigonometric basis functions. An interpolating fit, or any least squares fit to the form (10.28), explicitly contains all the information about lower order least squares fits.

Because of the extremely simple answer DFT has for least squares, it is especially simple to write a computer program to carry out the steps. Let \( m < n < p \) be integers, where \( n \) is the number of data points, \( m \) is the order of the least squares trigonometric model, and \( p \) governs the resolution of the readout of the best model. We can think of least squares as “filtering out” the highest frequency contributions of the order \( n \) interpolant, and leaving only the lowest \( m \) frequency contributions. That explains the name of the following Matlab function.

% Program 10.2 Least squares Fourier fit
% Least squares fit of n data points on [0,1] with trig function P_m(t)
% where 2 <= m <= n. Plot best fit at p (>= n) evenly-spaced points.
% Input: m, n, p, data points x, a vector of length n
% Output: filtered points xp
function xp=dftfilter(m,n,p,x)
t=(0:n-1)/n; % time points for data (n)
tp=(0:p-1)/p; % time points for interpolant (p)
y=fft(x); % compute interpolation coefficients
yp=zeros(p,1); % will hold coefficients for ifft
yp(1:m/2)=y(1:m/2); % keep only first m frequencies from the n
yp(m/2+1)=real(y(m/2+1)); % since m is even, keep cos term, not sin
yp(p-m/2+2:p)=y(n-m/2+2:n); % same as above, for upper tier
xp=real(ifft(yp))*(p/n); % invert fft to recover data
plot(t,x,’o’,tp,xp) % plot original data and least square approx

Example 10.6 Fit the data from Example 10.3 by least squares trigonometric functions of order 2, 4 and 6.
The point of Corollary 10.10 is that we can just interpolate the data points by applying $\frac{1}{n}F_n$, and chop off terms at will to get the least squares fit of lower orders. The result from Example 10.3 was that

$$P_8(t) = 0.4625 - 0.075 \cos 2\pi t + 0.2207 \sin 2\pi t - 0.025 \cos 4\pi t + 0.05 \sin 4\pi t - 0.075 \cos 6\pi t - 0.025 \sin 6\pi t + 0.0125 \cos 8\pi t.$$  \hfill (10.33)

Therefore the least squares models of order 2, 4, and 6 are

$$P_2(t) = 0.4625 - 0.075 \cos 2\pi t$$
$$P_4(t) = 0.4625 - 0.075 \cos 2\pi t + 0.2207 \sin 2\pi t - 0.025 \cos 4\pi t$$
$$P_6(t) = 0.4625 - 0.075 \cos 2\pi t + 0.2207 \sin 2\pi t - 0.025 \cos 4\pi t + 0.05 \sin 4\pi t - 0.075 \cos 6\pi t.$$  \hfill (10.34)

Figure 10.6 shows the three least squares fits, generated by `dftfilter.m` for $m = 2, 4$, and 6.

There is an obvious inefficiency built into `dftfilter.m`. It computes the order $n$ interpolant, and then ignores $n - m$ coefficients. Of course, one look at the Fourier matrix $F_n$ shows that if we only want to know the first $m$ Fourier coefficients of $n$ data points, we can multiply $x$ only by the top $m$ rows of $F_n$ and leave it at that. In other words, it would suffice to replace the $n \times n$ matrix $F_n$ by an $m \times n$ submatrix. A more efficient version of `dftfilter.m` would make use of this fact.

Matlab provides a music file of the first 9 seconds of Handel’s Hallelujah Chorus for us to practice filtering. The dotted curve in Figure 10.7 shows the first $2^8 = 256$ values of the file, which
consists of sampled sound intensities. The sampling rate of the music is 8192 Hz, meaning that $2^{13}$ samples, evenly-spaced in time, were collected per second. Using \texttt{dftfilter.m} with the number of retained basis functions set at $m = 50$ results in the solid curve of Figure 10.7.

Filtering can be used in two major ways. Here we are using it to try to match the original sound wave as closely as possible with a simpler function. This is a form of compression. Instead of using 256 numbers to store the wave, we could instead just store the lowest $m$ frequency components, and then reconstruct the wave when needed using Corollary 10.10. (Actually, we need to store $m/2 + 1$ complex numbers, but since the first and last are real, we can consider this as a total of $m$ real numbers.) In Figure 10.7 we have used $m = 50$ real numbers in place of the original 256, a better than 5:1 compression ratio. Note that the compression is lossy, in that the original wave has not been reproduced exactly.

The second major application of filtering is to remove noise. Given a music file where the music or speech was corrupted by high-frequency noise (or hiss), eliminating the higher-frequency contributions may be important to enhancing the sound. Of course, so-called "low-pass" filters are blunt hammers - a desired high-frequency part of the desired sound, possibly in overtones not even obvious to the listener, may be deleted as well. The topic of filtering is vast, and we will not do justice to it in this discussion. Some further investigation into filtering is done in the Reality Check at the end of the section.

### 10.2.4 Sound, noise, and filtering

The \texttt{dftfilter.m} code of the last section is an introduction to the vast area of digital signal processing. We are using the Fourier Transform as a way to transfer the information of a signal \( \{x_0, \ldots, x_{n-1}\} \) from the “time domain” to the “frequency domain”, where it is easier to operate on. When we finish changing what we want to change, we send the signal back to the time domain by
an inverse FFT.

Part of the reason this is a good idea has to do with the way our hearing system is constructed. The human ear hears frequencies, and so the building blocks in the frequency domain are more directly meaningful. We will introduce some basic concepts and a few of Matlab’s commands for handling audio signals.

Let \( c \) be a clean sound signal, and let \( r \) be a noise vector of the same length. Consider the noisy signal \( x = c + r \). What is allowed to be called noise? This is a complicated question. If we set \( r = c \), we would not consider \( r \) noise, since the result would be a louder but still clean version of \( c \).

One property noise has, by the usual definition, is that noise is uncorrelated with the signal, in other words that the expected value of the inner product \( c^T r \) is zero. We will use this lack of correlation below.

In a typical application we are presented with \( x \) and asked to find \( c \). The signal \( c \) might be the value of an important system variable, being monitored in a noisy environment. Or as in our example below, \( c \) might be an audio sample that we want to bring out of noise. Norbert Wiener suggested looking for the optimal filter in the frequency domain for removing the noise from \( x \), in the sense of least squares error. That is, he wanted to find a real, diagonal matrix \( \Phi \) such that the Euclidean norm of

\[
F^{-1} \Phi F x - c
\]

is as small as possible, where \( F \) denotes the discrete Fourier transform. Then we would clean up the signal \( x \) by applying the Fourier transform, multiplying by \( \Phi \), and then inverting the Fourier transform. This is called filtering in the frequency domain, since we are changing the Fourier transformed version of \( x \) rather than \( x \) itself.

Since \( F \) is a unitary matrix (satisfies \( F^T F = I \), the complex version of an orthogonal matrix), for a real vector \( v \) it follows that \( ||Fv||^2 = v^T F^T F v = ||v||^2 \), where we use the Euclidean norm. To find the best \( \Phi \), note that

\[
||F^{-1} \Phi F x - c|| = ||\Phi F x - F c|| \\
= ||\Phi F (c + r) - F c|| \\
= ||(\Phi - I) C + \Phi R||
\]

(10.35)

where we set \( C = F c \) and \( R = F r \) to be the Fourier transforms. Note that the definition of noise implies

\[
C^T R = F c^T F r = c^T \Phi^T F r = c^T r = 0.
\]

Therefore the norm reduces to

\[
(\Phi - I) C + \Phi R \cdot ((\Phi - I) C + \Phi R) = C^T (\Phi - I)^2 C + R^T \phi^2 R \\
= \sum_{i=1}^{n} (\phi_i - 1)^2 |c_i|^2 + \phi_i^2 |R_i|^2.
\]

(10.36)

To find the \( \phi_i \) that minimize this expression, differentiate with respect to each \( \phi_i \) separately:

\[
2(\phi_i - 1)|c_i|^2 + 2\phi_i |R_i|^2 = 0
\]
for each $i$, or solving for $\phi_i$,

$$\phi_i = \frac{|C_i|^2}{|C_i|^2 + |R_i|^2}. \quad (10.37)$$

This formula gives the optimal values for the entries of the diagonal matrix $\Phi$, to minimize the difference between the filtered version $F^{-1}\Phi F x$ and the clean signal $c$. The only problem is that in typical cases we don’t know $C$ or $R$, and must make some approximations to apply the formula.

Your job is to investigate ways of putting together an approximation. Again using the uncorrelatedness of signal and noise, approximate

$$|X_i|^2 \approx |C_i|^2 + |R_i|^2.$$

Then we can write the optimal choice as

$$\phi_i \approx \frac{|X_i|^2 - |R_i|^2}{|X_i|^2}, \quad (10.38)$$

and use our best knowledge of the noise level. For example, if the noise is uncorrelated Gaussian noise (modeled by adding a normal random number independently to each sample of the clean signal), we could replace $|R_i|^2$ in (10.38) with the constant $(p\sigma)^2$ where $\sigma$ is the standard deviation of the noise and $p$ is a parameter near one to be chosen. Note that

$$\sum_{i=1}^{n} |R_i|^2 = R^T R = rF^T F r = r^T r = \sum_{i=1}^{n} r_i^2.$$

In the code below we use $p = 1.3$ standard deviations to approximate $R_i$.

%code for Chapter 10 Reality Check
load handel % y is clean signal
c=y(1:40000); % work with first 40K samples
p=1.3; % parameter for cutoff
noise=std(c)*.50; % 50 percent noise
n=length(c); % n is length of signal
r=noise*randn(n,1); % pure noise
x=c+r; % noisy signal
fx=fft(x);sfx=conj(fx).*fx; % take fft of signal, and
sfcapprox=max(sfx-n*(p*noise)^2,0); % apply cutoff
phi=sfcapprox./sfx; % define phi as derived
xout=real(ifft(phi.*fx)); % invert the fft
% then compare sound(x) and sound(xout)

Suggested activities

1. Run the code as is to form the filtered signal $y_f$, and use Matlab’s sound command to compare the input and output signals.

2. Compute the mean squared error of the input ($y_s$) and output ($y_f$) by comparing to the clean signal ($y_c$).

3. Find the best value of the parameter $p$ for 50% noise. Choose a criterion: will you minimize MSE or compare the remaining noise by ear?

4. Change the noise level to 10%, 25%, 100%, 200%, and repeat Step 3. Summarize your conclusions.
5. Design a fair comparison of the Wiener filter with the low-pass filter described in Section 10.2, and carry out the comparison.

6. Download a .wav file of your choice, add noise, and carry out the steps above.

Exercises 10.2

10.2.1. Find the trigonometric interpolant for the data
(a) \((0, 0), (1/4, 1), (1/2, 0), (3/4, -1)\)
(b) \((0, 1), (1/4, 1), (1/2, -1), (3/4, -1)\)
(c) \((0, -1), (1/4, 1), (1/2, -1), (3/4, 1)\)
(d) \((0, 0), (1/8, 1), (1/4, 0), (3/8, -1), (1/2, 0), (5/8, 1), (3/4, 0), (7/8, -1)\)
(e) \((0, 1), (1/8, 2), (1/4, 1), (3/8, 0), (1/2, 1), (5/8, 2), (3/4, 1), (7/8, 0)\)

10.2.2. Find the best least squares approximation to the data in Exercise 10.2.1, using the basis functions \(1, \cos 2\pi t\) and \(\sin 2\pi t\).

10.2.3. Apply (10.23) to find the trigonometric interpolant for the data points
(a) \((0, 0), (1, 1), (2, 0), (3, -1)\)
(b) \((0, 1), (1, 1), (2, -1), (3, -1)\)
(c) \((0, 1), (1, 2), (2, 4), (3, 1)\)
(d) \((0, 3), (1, 4), (2, 5), (3, 6)\)
(e) \((0, 1), (1, 2), (2, 2), (3, 1), (4, 1), (5, 2), (6, 2), (7, 1)\)
(f) \((0, 1), (1, 0), (2, 0), (3, 0), (4, 1), (5, 0), (6, 0), (7, 0)\)

10.2.4. Find the best least squares approximation to the data in Exercise 10.2.3, using the basis functions \(1, \cos 2\pi t/n\) and \(\sin 2\pi t/n\).

10.2.5. Apply (10.22) to interpolate the data points
(a) \((0, 0), (2, 1), (4, 0), (6, -1)\)
(b) \((-4, 0), (0, 1), (4, 0), (8, -1)\)
(c) \((1, 1), (5, 1), (9, -1), (13, -1)\)
(d) \((1, 1), (2, 2), (3, 4), (4, 1)\)

10.2.6. Find versions of (10.20) and (10.22) that give formulas for the interpolating function in the case \(n\) is odd.

10.2.7. Prove Lemma 10.7. (Hint: Express \(\cos 2\pi jk/n\) as \((e^{i2\pi jk/n} + e^{-i2\pi jk/n})/2\), and write everything in terms of \(\omega = e^{-i2\pi/n}\) so that Lemma 10.1 can be applied.)

10.2.8. Expand Assumption 10.1 to cover the complex Fourier transform case. You will need to use complex conjugate transpose in place of transpose. Prove analogues of Theorems 10.8 and 10.9.

Computer Problems 10.2

10.2.1. Find the trigonometric interpolant for the data points
\((t, x) = (0, 0), (.125, 3), (.25, 0), (.375, -3), (.5, 0), (.625, 3), (.75, 0), (.875, -3)\).

10.2.2. Find the trigonometric interpolant for the data points \((t, x) = (0, 4), (1, 3), (2, 5), (3, 9), (4, 2), (5, 6)\). Plot the data points and the interpolating function.
10.2.3. Find the trigonometric interpolant for the data points $(t, x) = (1, 4), (2, 3), (3, 5), (4, 9), (5, 2), (6, 6)$. What are $c$ and $d$ in formula (10.22)? Plot the data points and the interpolating function.

10.2.4. Find the least squares approximating functions of all orders for the given data points. Using the trick of `dftfilter.m`, plot the data points and the approximating functions, as in Figure 10.6.

(a) $(t, x) = (0, 3), (1/4, 0), (1/2, -3), (3/4, 0)$.

(b) $(t, x) = (0, 2), (1/8, 0), (1/4, -2), (3/8, 0), (1/2, 2), (5/8, 0), (3/4, -2), (7/8, 0)$.

(c) $(t, x) = (0, 2), (1/4, 0), (1/2, 5), (3/4, 1)$.

(d) $(t, x) = (0, 2), (1/8, 0), (1/4, -2), (3/8, 1), (1/2, 2), (5/8, 3), (3/4, 0), (7/8, 1)$.

(e) $(t, x) = (1, 4), (2, 3), (3, 5), (4, 9), (5, 2), (6, 6)$.

(f) $(t, x) = (0, 2), (1, 3), (2, 5), (3, 1), (4, 2), (5, 1), (6, 2), (7, 1)$.

(g) $(t, x) = (0, 0), (1, 0), (2, 1), (3, -1), (4, -2), (5, 1), (6, -2), (7, -4)$.