GSL Technical Report

FFT Algorithms

Brian Gough, bjg@network-theory.co.uk
May, 1997

Contents

1 Introduction

Fast Fourier Transforms (FFTs) are efficient algorithms for calculating the discrete Fourier transform (DFT),

$$\begin{aligned} h_a &= \mathrm{DFT}(g_b) && (1)\\ &= \sum_{b=0}^{N-1} g_b \exp(-2\pi i a b/N) \qquad 0 \le a \le N-1 && (2)\\ &= \sum_{b=0}^{N-1} g_b W_N^{ab} \qquad W_N = \exp(-2\pi i/N) && (3) \end{aligned}$$

The DFT usually arises as an approximation to the continuous Fourier transform when functions are sampled at discrete intervals in space or time. The naive evaluation of the discrete Fourier transform is a matrix-vector multiplication $\mathbf{W}\vec{g}$, and would take $O(N^2)$ operations for N data-points. The general principle of the Fast Fourier Transform algorithms is to use a divide-and-conquer strategy to factorize the matrix W into smaller sub-matrices, typically reducing the operation count to $O(N \sum f_i)$ if N can be factorized into smaller integers, $N = f_1 f_2 \dots f_n$.

This chapter explains the algorithms used in the GSL FFT routines and provides some information on how to extend them. To learn more about the FFT you should read the review article *Fast Fourier Transforms: A Tutorial Review and A State of the Art* by [1]. There are several introductory books on the FFT with example programs, such as *The Fast Fourier Transform* by [2] and *DFT/FFT and Convolution Algorithms* by [3]. In 1979 the IEEE published a compendium of carefully-reviewed Fortran FFT programs in *Programs for Digital Signal Processing* [4] which is a useful reference for implementations of many different FFT algorithms. If you are interested in using DSPs then the *Handbook of Real-Time Fast Fourier Transforms* [5] provides detailed information on the algorithms and hardware needed to design, build and test DSP applications. Many FFT algorithms rely on results from number theory. These results are covered in the books *Fast transforms: algorithms, analyses, applications*, by [6], *Fast Algorithms for Digital Signal Processing* by [7] and *Number Theory in Digital Signal Processing* by [8]. There is also an annotated bibliography of papers on the FFT and related topics by [9].

2 Families of FFT algorithms

There are two main families of FFT algorithms: the Cooley-Tukey algorithm and the Prime Factor algorithm. These differ in the way they map the full FFT into smaller sub-transforms. Of the Cooley-Tukey algorithms there are two types of routine in common use: mixed-radix (general-N) algorithms and radix-2 (power of 2) algorithms. Each type of algorithm can be further classified by additional characteristics, such as whether it operates in-place or uses additional scratch space, whether its output is in a sorted or scrambled order, and whether it uses decimation-in-time or -frequency iterations.

Mixed-radix algorithms work by factorizing the data vector into shorter lengths. These can then be transformed by small-N FFTs. Typical programs include FFTs for small prime factors, such as 2, 3, 5, ... which are highly optimized. The small-N FFT modules act as

building blocks and can be multiplied together to make longer transforms. By combining a reasonable set of modules it is possible to compute FFTs of many different lengths. If the small-N modules are supplemented by an $O(N^2)$ general-N module then an FFT of any length can be computed, in principle. Of course, any lengths which contain large prime factors would perform only as $O(N^2)$.

Radix-2 algorithms, or "power of two" algorithms, are simplified versions of the mixed-radix algorithm. They are restricted to lengths which are a power of two. The basic radix-2 FFT module only involves addition and subtraction, so the algorithms are very simple. Radix-2 algorithms have been the subject of much research into optimizing the FFT. Many of the most efficient radix-2 routines are based on the "split-radix" algorithm. This is actually a hybrid which combines the best parts of both radix-2 and radix-4 ("power of 4") algorithms [10, 11].

The prime factor algorithm (PFA) is an alternative form of general-N algorithm based on a different way of recombining small-N FFT modules [12, 13]. It has a very simple indexing scheme which makes it attractive. However it only works in the case where all factors are mutually prime. This requirement makes it more suitable as a specialized algorithm for given lengths.

2.1 FFTs of prime lengths

Large prime lengths cannot be handled efficiently by any of these algorithms. However it may still possible to compute a DFT, by using results from number theory. Rader showed that it is possible to convert a length-p FFT (where p is prime) into a convolution of length-$(p-1)$. There is a simple identity between the convolution of length N and the FFT of the same length, so if $p-1$ is easily factorizable this allows the convolution to be computed efficiently via the FFT. The idea is presented in the original paper by [14] (also reprinted in [8]), but for more details see the theoretical books mentioned earlier.

2.2 Optimization

There is no such thing as the single fastest FFT *algorithm*. FFT algorithms involve a mixture of floating point calculations, integer arithmetic and memory access. Each of these operations will have different relative speeds on different platforms. The performance of an algorithm is a function of the hardware it is implemented on. The goal of optimization is thus to choose the algorithm best suited to the characteristics of a given hardware platform.

For example, the Winograd Fourier Transform (WFTA) is an algorithm which is designed to reduce the number of floating point multiplications in the FFT. However, it does this at the expense of using many more additions and data transfers than other algorithms. As a consequence the WFTA might be a good candidate algorithm for machines where data transfers occupy a negligible time relative to floating point arithmetic. However on most modern machines, where the speed of data transfers is comparable to or slower than floating point operations, it would be outperformed by an algorithm which used a better mix of operations (i.e. more floating point operations but fewer data transfers).

For a study of this sort of effect in detail, comparing the different algorithms on different platforms consult the paper *Effects of Architecture Implementation on DFT Algorithm Performance* by [15]. The paper was written in the early 1980's and has data for super- and mini-computers which you are unlikely to see today, except in a museum. However, the

methodology is still valid and it would be interesting to see similar results for present day computers.

3 FFT Concepts

Factorization is the key principle of the mixed-radix FFT divide-and-conquer strategy. If N can be factorized into a product of n_f integers,

$$N = f_1 f_2 ... f_{n_f}, \tag{4}$$

then the FFT itself can be divided into smaller FFTs for each factor. More precisely, an FFT of length N can be broken up into,

(N/f_1) FFTs of length f_1,
(N/f_2) FFTs of length f_2,
...
(N/f_{n_f}) FFTs of length f_{n_f}.

The total number of operations for these sub-operations will be $O(N(f_1+f_2+...+f_{n_f}))$. When the factors of N are all small integers this will be substantially less than $O(N^2)$. For example, when N is a power of 2 an FFT of length $N = 2^m$ can be reduced to $mN/2$ FFTs of length 2, or $O(N \log_2 N)$ operations. Here is a demonstration which shows this:

We start with the full DFT,

$$h_a = \sum_{b=0}^{N-1} g_b W_N^{ab} \qquad W_N = \exp(-2\pi i/N) \tag{5}$$

and split the sum into even and odd terms,

$$= \sum_{b=0}^{N/2-1} g_{2b} W_N^{a(2b)} + \sum_{b=0}^{N/2-1} g_{2b+1} W_N^{a(2b+1)}. \tag{6}$$

This converts the original DFT of length N into two DFTs of length $N/2$,

$$h_a = \sum_{b=0}^{N/2-1} g_{2b} W_{(N/2)}^{ab} + W_N^a \sum_{b=0}^{N/2-1} g_{2b+1} W_{(N/2)}^{ab} \tag{7}$$

The first term is a DFT of the even elements of g. The second term is a DFT of the odd elements of g, premultiplied by an exponential factor W_N^k (known as a *twiddle factor*).

$$\mathrm{DFT}(h) = \mathrm{DFT}(g_{even}) + W_N^k \mathrm{DFT}(g_{odd}) \tag{8}$$

By splitting the DFT into its even and odd parts we have reduced the operation count from N^2 (for a DFT of length N) to $2(N/2)^2$ (for two DFTs of length $N/2$). The cost of the splitting is that we need an additional $O(N)$ operations to multiply by the twiddle factor W_N^k and recombine the two sums.

We can repeat the splitting procedure recursively $\log_2 N$ times until the full DFT is reduced to DFTs of single terms. The DFT of a single value is just the identity operation, which costs

nothing. However since $O(N)$ operations were needed at each stage to recombine the even and odd parts the total number of operations to obtain the full DFT is $O(N \log_2 N)$. If we had used a length which was a product of factors f_1, f_2, ... we could have split the sum in a similar way. First we would split terms corresponding to the factor f_1, instead of the even and odd terms corresponding to a factor of two. Then we would repeat this procedure for the subsequent factors. This would lead to a final operation count of $O(N \sum f_i)$.

This procedure gives some motivation for why the number of operations in a DFT can in principle be reduced from $O(N^2)$ to $O(N \sum f_i)$. It does not give a good explanation of how to implement the algorithm in practice which is what we shall do in the next section.

4 Radix-2 Algorithms

For radix-2 FFTs it is natural to write array indices in binary form because the length of the data is a power of two. This is nicely explained in the article *The FFT: Fourier Transforming One Bit at a Time* by [16]. A binary representation for indices is the key to deriving the simplest efficient radix-2 algorithms.

We can write an index b ($0 \leq b < 2^{n-1}$) in binary representation like this,

$$b = [b_{n-1} \dots b_1 b_0] = 2^{n-1} b_{n-1} + \cdots + 2b_1 + b_0. \tag{9}$$

Each of the $b_0, b_1, \dots, b_{n-1}$ are the bits (either 0 or 1) of b.

Using this notation the original definition of the DFT can be rewritten as a sum over the bits of b,

$$h(a) = \sum_{b=0}^{N-1} g_b \exp(-2\pi i ab/N) \tag{10}$$

to give an equivalent summation like this,

$$h([a_{n-1} \dots a_1 a_0]) = \sum_{b_0=0}^{1} \sum_{b_1=0}^{1} \cdots \sum_{b_{n-1}=0}^{1} g([b_{n-1} \dots b_1 b_0]) W_N^{ab} \tag{11}$$

where the bits of a are $a = [a_{n-1} \dots a_1 a_0]$.

To reduce the number of operations in the sum we will use the periodicity of the exponential term,

$$W_N^{x+N} = W_N^x. \tag{12}$$

Most of the products ab in W_N^{ab} are greater than N. By making use of this periodicity they can all be collapsed down into the range $0 \dots N-1$. This allows us to reduce the number of operations by combining common terms, modulo N. Using this idea we can derive decimation-in-time or decimation-in-frequency algorithms, depending on how we break the DFT summation down into common terms. We'll first consider the decimation-in-time algorithm.

4.1 Radix-2 Decimation-in-Time (DIT)

To derive the decimation-in-time algorithm we start by separating out the most significant bit of the index b,

$$[b_{n-1} \dots b_1 b_0] = 2^{n-1} b_{n-1} + [b_{n-2} \dots b_1 b_0] \tag{13}$$

Now we can evaluate the innermost sum of the DFT without any dependence on the remaining bits of b in the exponential,

$$\begin{aligned} h([a_{n-1}\dots a_1 a_0]) &= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-1}=0}^{1} g(b) W_N^{a(2^{n-1}b_{n-1}+[b_{n-2}\dots b_1 b_0])} && (14)\\ &= \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-2}=0}^{1} W_N^{a[b_{n-2}\dots b_1 b_0]} \sum_{b_{n-1}=0}^{1} g(b) W_N^{a(2^{n-1}b_{n-1})} && (15)\end{aligned}$$

Looking at the term $W_N^{a(2^{n-1}b_{n-1})}$ we see that we can also remove most of the dependence on a as well, by using the periodicity of the exponential,

$$\begin{aligned} W_N^{a(2^{n-1}b_{n-1})} &= \exp(-2\pi i[a_{n-1}\dots a_1 a_0]2^{n-1}b_{n-1}/2^n) && (16)\\ &= \exp(-2\pi i[a_{n-1}\dots a_1 a_0]b_{n-1}/2) && (17)\\ &= \exp(-2\pi i(2^{n-2}a_{n-1}+\cdots+a_1+(a_0/2))b_{n-1}) && (18)\\ &= \exp(-2\pi i a_0 b_{n-1}/2) && (19)\\ &= W_2^{a_0 b_{n-1}} && (20)\end{aligned}$$

Thus the innermost exponential term simplifies so that it only involves the highest order bit of b and the lowest order bit of a, and the sum can be reduced to,

$$h([a_{n-1}\dots a_1 a_0]) = \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-2}=0}^{1} W_N^{a[b_{n-2}\dots b_1 b_0]} \sum_{b_{n-1}=0}^{1} g(b) W_2^{a_0 b_{n-1}}. \quad (21)$$

We can repeat this this procedure for the next most significant bit of b, b_{n-2}, using a similar identity,

$$\begin{aligned} W_N^{a(2^{n-2}b_{n-2})} &= \exp(-2\pi i[a_{n-1}\dots a_1 a_0]2^{n-2}b_{n-2}/2^n) && (22)\\ &= W_4^{[a_1 a_0]b_{n-2}}. && (23)\end{aligned}$$

to give a formula with even less dependence on the bits of a,

$$h([a_{n-1}\dots a_1 a_0]) = \sum_{b_0=0}^{1}\sum_{b_1=0}^{1}\cdots\sum_{b_{n-3}=0}^{1} W_N^{a[b_{n-3}\dots b_1 b_0]} \sum_{b_{n-2}=0}^{1} W_4^{[a_1 a_0]b_{n-2}} \sum_{b_{n-1}=0}^{1} g(b) W_2^{a_0 b_{n-1}}. \quad (24)$$

If we repeat the process for all the remaining bits we obtain a simplified DFT formula which is the basis of the radix-2 decimation-in-time algorithm,

$$h([a_{n-1}\dots a_1 a_0]) = \sum_{b_0=0}^{1} W_N^{[a_{n-1}\dots a_1 a_0]b_0} \cdots \sum_{b_{n-2}=0}^{1} W_4^{[a_1 a_0]b_{n-2}} \sum_{b_{n-1}=0}^{1} W_2^{a_0 b_{n-1}} g(b) \quad (25)$$

To convert the formula to an algorithm we expand out the sum recursively, evaluating each of the intermediate summations, which we denote by g_1, g_2, ..., g_n,

$$g_1(a_0, b_{n-2}, b_{n-3}, \ldots, b_1, b_0) = \sum_{b_{n-1}=0}^{1} W_2^{a_0 b_{n-1}} g([b_{n-1} b_{n-2} b_{n-3} \ldots b_1 b_0]) \quad (26)$$

$$g_2(a_0, \quad a_1, b_{n-3}, \ldots, b_1, b_0) = \sum_{b_{n-2}=0}^{1} W_4^{[a_1 a_0] b_{n-2}} g_1(a_0, b_{n-2}, b_{n-3}, \ldots, b_1, b_0) \quad (27)$$

$$g_3(a_0, \quad a_1, \quad a_2, \ldots, b_1, b_0) = \sum_{b_{n-3}=0}^{1} W_8^{[a_2 a_1 a_0] b_{n-2}} g_2(a_0, a_1, b_{n-3}, \ldots, b_1, b_0) \quad (28)$$

$$\ldots = \ldots \quad (29)$$

$$g_n(a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}) = \sum_{b_0=0}^{1} W_N^{[a_{n-1} \ldots a_1 a_0] b_0} g_{n-1}(a_0, a_1, a_2, \ldots, a_{n-2}, b_0) \quad (30)$$

After the final sum, we can obtain the transform h from g_n,

$$h([a_{n-1} \ldots a_1 a_0]) = g_n(a_0, a_1, \ldots, a_{n-1}) \quad (31)$$

Note that we left the storage arrangements of the intermediate sums unspecified by using the bits as function arguments and not as an index. The storage of intermediate sums is different for the decimation-in-time and decimation-in-frequency algorithms.

Before deciding on the best storage scheme we'll show that the results of each stage, g_1, g_2, ..., can be carried out *in-place*. For example, in the case of g_1, the inputs are,

$$g([\underline{b_{n-1}} b_{n-2} b_{n-3} \ldots b_1 b_0]) \quad (32)$$

for $b_{n-1} = (0, 1)$, and the corresponding outputs are,

$$g_1(\underline{a_0}, b_{n-2}, b_{n-3}, \ldots, b_1, b_0) \quad (33)$$

for $a_0 = (0, 1)$. It's clear that if we hold $b_{n-2}, b_{n-3}, \ldots, b_1, b_0$ fixed and compute the sum over b_{n-1} in memory for both values of $a_0 = 0, 1$ then we can store the result for $a_0 = 0$ in the location which originally had $b_0 = 0$ and the result for $a_0 = 1$ in the location which originally had $b_0 = 1$. The two inputs and two outputs are known as *dual node pairs*. At each stage of the calculation the sums for each dual node pair are independent of the others. It is this property which allows an in-place calculation.

So for an in-place pass our storage has to be arranged so that the two outputs $g_1(a_0, \ldots)$ overwrite the two input terms $g([b_{n-1}, \ldots])$. Note that the order of a is reversed from the natural order of b, i.e.@: the least significant bit of a replaces the most significant bit of b. This is inconvenient because a occurs in its natural order in all the exponentials, W^{ab}. We could keep track of both a and its bit-reverse, $a^{bit-reversed}$ at all times but there is a neat trick which avoids this: if we bit-reverse the order of the input data g before we start the calculation we can also bit-reverse the order of a when storing intermediate results. Since the storage involving a was originally in bit-reversed order the switch in the input g now allows us to use normal ordered storage for a, the same ordering that occurs in the exponential factors.

This is complicated to explain, so here is an example of the 4 passes needed for an $N = 16$ decimation-in-time FFT, with the initial data stored in bit-reversed order,

$$g_1([b_0 b_1 b_2 a_0]) = \sum_{b_3=0}^{1} W_2^{a_0 b_3} g([b_0 b_1 b_2 b_3]) \tag{34}$$

$$g_2([b_0 b_1 a_1 a_0]) = \sum_{b_2=0}^{1} W_4^{[a_1 a_0] b_2} g_1([b_0 b_1 b_2 a_0]) \tag{35}$$

$$g_3([b_0 a_2 a_1 a_0]) = \sum_{b_1=0}^{1} W_8^{[a_2 a_1 a_0] b_1} g_2([b_0 b_1 a_1 a_0]) \tag{36}$$

$$h(a) = g_4([a_3 a_2 a_1 a_0]) = \sum_{b_0=0}^{1} W_{16}^{[a_3 a_2 a_1 a_0] b_0} g_3([b_0 a_2 a_1 a_0]) \tag{37}$$

We compensate for the bit reversal of the input data by accessing g with the bit-reversed form of b in the first stage. This ensures that we are still carrying out the same calculation, using the same data, and not accessing different values. Only single bits of b ever occur in the exponential so we never need the bit-reversed form of b.

Let's examine the third pass in detail,

$$g_3([b_0 a_2 a_1 a_0]) = \sum_{b_1=0}^{1} W_8^{[a_2 a_1 a_0] b_1} g_2([b_0 b_1 a_1 a_0]) \tag{38}$$

First note that only one bit, b_1, varies in each summation. The other bits of b (b_0) and of a ($a_1 a_0$) are essentially "spectators" – we must loop over all combinations of these bits and carry out the same basic calculation for each, remembering to update the exponentials involving W_8 appropriately. If we are storing the results in-place (with g_3 overwriting g_2 we will need to compute the sums involving $b_1 = 0, 1$ and $a_2 = 0, 1$ simultaneously.

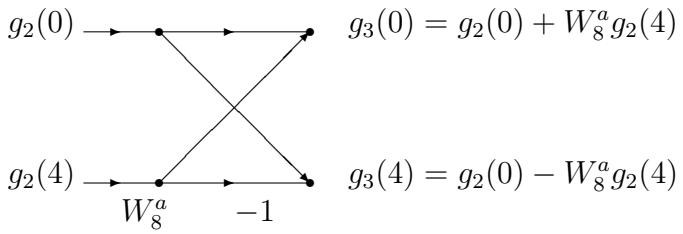
$$\begin{pmatrix} g_3([b_0 0 a_1 a_0]) \\ g_3([b_0 1 a_1 a_0]) \end{pmatrix} = \begin{pmatrix} g_2([b_0 0 a_1 a_0]) + W_8^{[0 a_1 a_0]} g_2([b_2 1 a_1 a_0]) \\ g_2([b_0 0 a_1 a_0]) + W_8^{[1 a_1 a_0]} g_2([b_2 1 a_1 a_0]) \end{pmatrix} \tag{39}$$

We can write this in a more symmetric form by simplifying the exponential,

$$W_8^{[a_2 a_1 a_0]} = W_8^{4a_2 + [a_1 a_0]} = (-1)^{a_2} W_8^{[a_1 a_0]} \tag{40}$$

$$\begin{pmatrix} g_3([b_0 0 a_1 a_0]) \\ g_3([b_0 1 a_1 a_0]) \end{pmatrix} = \begin{pmatrix} g_2([b_0 0 a_1 a_0]) + W_8^{[a_1 a_0]} g_2([b_2 1 a_1 a_0]) \\ g_2([b_0 0 a_1 a_0]) - W_8^{[a_1 a_0]} g_2([b_2 1 a_1 a_0]) \end{pmatrix} \tag{41}$$

The exponentials $W_8^{[a_1 a_0]}$ are referred to as *twiddle factors*. The form of this calculation, a symmetrical sum and difference involving a twiddle factor is called *a butterfly*. It is often shown diagrammatically, and in the case $b_0 = a_0 = a_1 = 0$ would be drawn like this,



The inputs are shown on the left and the outputs on the right. The outputs are computed by multiplying the incoming lines by their accompanying factors (shown next to the lines) and summing the results at each node.

In general, denoting the bit for dual-node pairs by Δ and the remaining bits of a and b by $\hat{a}$ and $\hat{b}$, the butterfly is,

$$\begin{pmatrix} g(\hat{b}+\hat{a}) \\ g(\hat{b}+\Delta+\hat{a}) \end{pmatrix} \leftarrow \begin{pmatrix} g(\hat{b}+\hat{a}) + W_{2\Delta}^{\hat{a}} g(\hat{b}+\Delta+\hat{a}) \\ g(\hat{b}+\hat{a}) - W_{2\Delta}^{\hat{a}} g(\hat{b}+\Delta+\hat{a}) \end{pmatrix} \tag{42}$$

where $\hat{a}$ runs from $0 \ldots \Delta - 1$ and $\hat{b}$ runs through $0 \times 2\Delta$, $1 \times 2\Delta$, $\ldots$, $(N/\Delta - 1)2\Delta$. The value of Δ is 1 on the first pass, 2 on the second pass and 2^{n-1} on the n-th pass. Each pass requires $N/2$ in-place computations, each involving two input locations and two output locations.

In the example above $\Delta = [100] = 4$, $\hat{a} = [a_1 a_0]$ and $\hat{b} = [b_0 000]$.

This leads to the canonical radix-2 decimation-in-time FFT algorithm for 2^n data points stored in the array $g(0) \ldots g(2^n - 1)$.

```
bit-reverse ordering of g
Δ ⇐ 1
for pass = 1...n do
  W ⇐ exp(−2πi/2Δ)
  for (a = 0; a < Δ; a++) do
    for (b = 0; b < N; b+ =2 ∗ Δ) do
      t0 ⇐ g(b + a) + W^a g(b + Δ + a)
      t1 ⇐ g(b + a) − W^a g(b + Δ + a)
      g(b + a) ⇐ t0
      g(b + Δ + a) ⇐ t1
    end for
  end for
  Δ ⇐ 2Δ
end for
```

4.2 Details of the Implementation

It is straightforward to implement a simple radix-2 decimation-in-time routine from the algorithm above. Some optimizations can be made by pulling the special case of $a = 0$ out of the loop over a, to avoid unnecessary multiplications when $W^a = 1$,

for $(b = 0; b < N; b+ = 2 * \Delta)$ **do**
  $t_0 \Leftarrow g(b) + g(b + \Delta)$
  $t_1 \Leftarrow g(b) - g(b + \Delta)$
  $g(b) \Leftarrow t_0$
  $g(b + \Delta) \Leftarrow t_1$
end for

There are several algorithms for doing fast bit-reversal. We use the Gold-Rader algorithm, which is simple and does not require any working space,

for $i = 0 \ldots n - 2$ **do**
  $k = n/2$
  if $i < j$ **then**
    swap $g(i)$ and $g(j)$
  end if
  while $k \leq j$ **do**
    $j \Leftarrow j - k$
    $k \Leftarrow k/2$
  end while
  $j \Leftarrow j + k$
end for

The Gold-Rader algorithm is typically twice as fast as a naive bit-reversal algorithm (where the bit reversal is carried out by left-shifts and right-shifts on the index). The library also has a routine for the Rodriguez bit reversal algorithm, which also does not require any working space [17]. There are faster bit reversal algorithms, but they all use additional scratch space [18].

Within the loop for a we can compute W^a using a trigonometric recursion relation,

$$W^{a+1} = WW^a \tag{43}$$
$$= (\cos(2\pi/2\Delta) + i\sin(2\pi/2\Delta))W^a \tag{44}$$

This requires only $2\log_2 N$ trigonometric calls, to compute the initial values of $\exp(2\pi i/2\Delta)$ for each pass.

4.3 Radix-2 Decimation-in-Frequency (DIF)

To derive the decimation-in-frequency algorithm we start by separating out the lowest order bit of the index a. Here is an example for the decimation-in-frequency $N = 16$ DFT.

$$W_{16}^{[a_3a_2a_1a_0][b_3b_2b_1b_0]} = W_{16}^{[a_3a_2a_1a_0][b_2b_1b_0]} W_{16}^{[a_3a_2a_1a_0][b_3000]} \tag{45}$$
$$= W_8^{[a_3a_2a_1][b_2b_1b_0]} W_{16}^{a_0[b_2b_1b_0]} W_2^{a_0b_3} \tag{46}$$
$$= W_8^{[a_3a_2a_1][b_2b_1b_0]} W_{16}^{a_0[b_2b_1b_0]} (-1)^{a_0b_3} \tag{47}$$

By repeating the same type of the expansion on the term,

$$W_8^{[a_3a_2a_1][b_2b_1b_0]} \tag{48}$$

we can reduce the transform to an alternative simple form,

$$h(a) = \sum_{b_0=0}^{1} (-1)^{a_3 b_0} W_4^{a_2 b_0} \sum_{b_1=0}^{1} (-1)^{a_2 b_1} W_8^{a_1 [b_1 b_0]} \sum_{b_2=0}^{1} (-1)^{a_1 b_2} W_{16}^{a_0 [b_2 b_1 b_0]} \sum_{b_3=0}^{1} (-1)^{a_0 b_3} g(b) \quad (49)$$

To implement this we can again write the sum recursively. In this case we do not have the problem of the order of a being bit reversed – the calculation can be done in-place using the natural ordering of a and b,

$$g_1([a_0 b_2 b_1 b_0]) = W_{16}^{a_0 [b_2 b_1 b_0]} \sum_{b_3=0}^{1} (-1)^{a_0 b_3} g([b_3 b_2 b_1 b_0]) \quad (50)$$

$$g_2([a_0 a_1 b_1 b_0]) = W_8^{a_1 [b_1 b_0]} \sum_{b_2=0}^{1} (-1)^{a_1 b_2} g_1([a_0 b_2 b_1 b_0]) \quad (51)$$

$$g_3([a_0 a_1 a_2 b_0]) = W_4^{a_2 b_0} \sum_{b_1=0}^{1} (-1)^{a_2 b_1} g_2([a_0 a_1 b_1 b_0]) \quad (52)$$

$$h(a) = g_4([a_0 a_1 a_2 a_3]) = \sum_{b_0=0}^{1} (-1)^{a_3 b_0} g_3([a_0 a_1 a_2 b_0]) \quad (53)$$

The final pass leaves the data for $h(a)$ in bit-reversed order, but this is easily fixed by a final bit-reversal of the ordering.

The basic in-place calculation or butterfly for each pass is slightly different from the decimation-in-time version,

$$\begin{pmatrix} g(\hat{a} + \hat{b}) \\ g(\hat{a} + \Delta + \hat{b}) \end{pmatrix} \leftarrow \begin{pmatrix} g(\hat{a} + \hat{b}) + g(\hat{a} + \Delta + \hat{b}) \\ W_\Delta^{\hat{b}} \left(g(\hat{a} + \hat{b}) - g(\hat{a} + \Delta + \hat{b}) \right) \end{pmatrix} \quad (54)$$

In each pass $\hat{b}$ runs from $0 \ldots \Delta - 1$ and $\hat{a}$ runs from $0, 2\Delta, \ldots, (N/\Delta - 1)\Delta$. On the first pass we start with $\Delta = 16$, and on subsequent passes Δ takes the values $8, 4, \ldots, 1$.

This leads to the canonical radix-2 decimation-in-frequency FFT algorithm for 2^n data points stored in the array $g(0) \ldots g(2^n - 1)$.

```
Δ ⇐ 2^(n-1)
for pass = 1 ... n do
  W ⇐ exp(-2πi/2Δ)
  for (b = 0; b < Δ; b++) do
    for (a = 0; a < N; a += 2*Δ) do
      t0 ⇐ g(b + a) + g(a + Δ + b)
      t1 ⇐ W^b (g(a + b) - g(a + Δ + b))
      g(a + b) ⇐ t0
      g(a + Δ + b) ⇐ t1
    end for
  end for
  Δ ⇐ Δ/2
end for
bit-reverse ordering of g
```

5 Self-Sorting Mixed-Radix Complex FFTs

This section is based on the review article *Self-sorting Mixed-Radix Fast Fourier Transforms* by [19]. You should consult his article for full details of all the possible algorithms (there are many variations). Here I have annotated the derivation of the simplest mixed-radix decimation-in-frequency algorithm.

For general-N FFT algorithms the simple binary-notation of radix-2 algorithms is no longer useful. The mixed-radix FFT has to be built up using products of matrices acting on a data vector. The aim is to take the full DFT matrix W_N and factor it into a set of small, sparse matrices corresponding to each factor of N.

We'll denote the components of matrices using either subscripts or function notation,

$$M_{ij} = M(i,j) \tag{55}$$

with (C-like) indices running from 0 to $N-1$. Matrix products will be denoted using square brackets,

$$[AB]_{ij} = \sum_k A_{ik} B_{kj} \tag{56}$$

Three special matrices will be needed in the mixed-radix factorization of the DFT: the identity matrix, I, a permutation matrix, P and a matrix of twiddle factors, D, as well as the normal DFT matrices W_n.

We write the identity matrix of order r as $I_r(n,m)$,

$$I_r(n,m) = \delta_{nm} \tag{57}$$

for $0 \le n, m \le r-1$.

We also need to define a permutation matrix P_b^a that performs digit reversal of the ordering of a vector. If the index of a vector $j = 0 \ldots N-1$ is factorized into $j = la+m$, with $0 \le l \le b-1$ and $0 \le m \le a-1$ then the operation of the matrix P will exchange positions $la+m$ and $mb+l$ in the vector (this sort of digit-reversal is the generalization of bit-reversal to a number system with exponents a and b).

In mathematical terms P is a square matrix of size $ab \times ab$ with the property,

$$\begin{aligned} P_b^a(j,k) &= 1 \text{ if } j = ra+s \text{ and } k = sb+r & (58)\\ &= 0 \text{ otherwise} & (59)\end{aligned}$$

Finally the FFT algorithm needs a matrix of twiddle factors, D_b^a, for the trigonometric sums. D_b^a is a diagonal square matrix of size $ab \times ab$ with the definition,

$$\begin{aligned} D_b^a(j,k) &= \omega_{ab}^{sr} \text{ if } j = k = sb+r & (60)\\ &= 0 \text{ otherwise} & (61)\end{aligned}$$

where $\omega_{ab} = e^{-2\pi i/ab}$.

5.1 The Kronecker Product

The Kronecker matrix product plays an important role in all the algorithms for combining operations on different subspaces. The Kronecker product $A \otimes B$ of two square matrices A and B, of sizes $a \times a$ and $b \times b$ respectively, is a square matrix of size $ab \times ab$, defined as,

$$[A \otimes B](tb+u, rb+s) = A(t,r)B(u,s) \tag{62}$$

where $0 \le u, s < b$ and $0 \le t, r < a$. Let's examine a specific example. If we take a 2×2 matrix and a 3×3 matrix,

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \tag{63}$$

then the Kronecker product $A \otimes B$ is,

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B \\ a_{12}B & a_{22}B \end{pmatrix} \tag{64}$$

$$= \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} \\ a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} \\ a_{11}b_{31} & a_{11}b_{32} & a_{11}b_{33} & a_{12}b_{31} & a_{12}b_{32} & a_{12}b_{33} \\ a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} \\ a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} \\ a_{21}b_{31} & a_{21}b_{32} & a_{21}b_{33} & a_{22}b_{31} & a_{22}b_{32} & a_{22}b_{33} \end{pmatrix} \tag{65}$$

When the Kronecker product $A \otimes B$ acts on a vector of length ab, each matrix operates on a different subspace of the vector. Writing the index i as $i = tb + u$, with $0 \le u \le b-1$ and $0 \le t \le a$, we can see this explicitly by looking at components,

$$[(A \otimes B)v]_{(tb+u)} = \sum_{t'=0}^{a-1} \sum_{u'=0}^{b-1} [A \otimes B]_{(tb+u,t'b+u')} v_{t'b+u'} \tag{66}$$

$$= \sum_{t'u'} A_{tt'} B_{uu'} v_{t'b+u'} \tag{67}$$

The matrix B operates on the "index" u', for all values of t', and the matrix A operates on the "index" t', for all values of u'. The most important property needed for deriving the FFT factorization is that the matrix product of two Kronecker products is the Kronecker product of the two matrix products,

$$(A \otimes B)(C \otimes D) = (AC \otimes BD) \tag{68}$$

This follows straightforwardly from the original definition of the Kronecker product.

5.2 Two factor case, $N = ab$

First consider the simplest possibility, where the data length N can be divided into two factors, $N = ab$. The aim is to reduce the DFT matrix W_N into simpler matrices corresponding to each factor. To make the derivation easier we will start from the known factorization and verify it (the initial factorization can be guessed by generalizing from simple cases). Here is the factorization we are going to prove,

$$W_{ab} = (W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b). \tag{69}$$

We can check it by expanding the product into components,

$$[(W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b)](la + m, rb + s) \tag{70}$$

$$= \sum_{u=0}^{b-1} \sum_{t=0}^{a-1} [(W_b \otimes I_a)](la + m, ua + t)[P_b^a D_b^a (W_a \otimes I_b)](ua + t, rb + s) \tag{71}$$

where we have split the indices to match the Kronecker product $0 \le m, r \le a$, $0 \le l, s \le b$. The first term in the sum can easily be reduced to its component form,

$$\begin{aligned}[(W_b \otimes I_a)](la+m, ua+t) &= W_b(l,u) I_a(m,t) &(72)\\ &= \omega_b^{lu} \delta_{mt} &(73)\end{aligned}$$

The second term is more complicated. We can expand the Kronecker product like this,

$$\begin{aligned}(W_a \otimes I_b)(tb+u, rb+s) &= W_a(t,r) I_a(u,s) &(74)\\ &= \omega_a^{tr} \delta_{us} &(75)\end{aligned}$$

and use this term to build up the product, $P_b^a D_b^a (W_a \otimes I_b)$. We first multiply by D_b^a,

$$[D_b^a (W_a \otimes I_b)](tb+u, rb+s) = \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \quad (76)$$

and then apply the permutation matrix, P_b^a, which digit-reverses the ordering of the first index, to obtain,

$$[P_b^a D_b^a (W_a \otimes I_b)](ua+t, rb+s) = \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \quad (77)$$

Combining the two terms in the matrix product we can obtain the full expansion in terms of the exponential ω,

$$[(W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b)](la+m, rb+s) = \sum_{u=0}^{b-1} \sum_{t=0}^{a-1} \omega_b^{lu} \delta_{mt} \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} \quad (78)$$

If we evaluate this sum explicitly we can make the connection between the product involving W_a and W_b (above) and the expansion of the full DFT matrix W_{ab},

$$\begin{aligned}\sum_{u=0}^{b-1} \sum_{t=0}^{a-1} \omega_b^{lu} \delta_{mt} \omega_{ab}^{tu} \omega_a^{tr} \delta_{su} &= \omega_b^{ls} \omega_{ab}^{ms} \omega_a^{mr} &(79)\\ &= \omega_{ab}^{als+ms+bmr} &(80)\\ &= \omega_{ab}^{als+ms+bmr} \omega_{ab}^{lrab} \quad \text{using } \omega_{ab}^{ab} = 1 &(81)\\ &= \omega_{ab}^{(la+m)(rb+s)} &(82)\\ &= W_{ab}(la+m, rb+s) &(83)\end{aligned}$$

The final line shows that matrix product given above is identical to the full two-factor DFT matrix, W_{ab}. Thus the full DFT matrix W_{ab} for two factors a, b can be broken down into a product of sub-transforms, W_a and W_b, plus permutations, P, and twiddle factors, D, according to the formula,

$$W_{ab} = (W_b \otimes I_a) P_b^a D_b^a (W_a \otimes I_b). \quad (84)$$

This relation is the foundation of the general-N mixed-radix FFT algorithm.

5.3 Three factor case, $N = abc$

The result for the two-factor expansion can easily be generalized to three factors. We first consider abc as being a product of two factors a and (bc), and then further expand the product (bc) into b and c. The first step of the expansion looks like this,

$$\begin{aligned}W_{abc} &= W_{a(bc)} &(85)\\ &= (W_{bc} \otimes I_a) P_{bc}^a D_{bc}^a (W_a \otimes I_{bc}). &(86)\end{aligned}$$

And after using the two-factor result to expand out W_{bc} we obtain the factorization of W_{abc},

$$\begin{aligned} W_{abc} &= (((W_c \otimes I_b)P^b_c D^b_c (W_b \otimes I_c)) \otimes I_a) P^a_{bc} D^a_{bc} (W_a \otimes I_{bc}) && (87)\\ &= (W_c \otimes I_{ab})(P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) P^a_{bc} D^a_{bc} (W_a \otimes I_c) && (88)\end{aligned}$$

We can write this factorization in a product form, with one term for each factor,

$$W_{abc} = T_3 T_2 T_1 \quad (89)$$

where we read off T_1, T_2 and T_3,

$$\begin{aligned} T_1 &= P^a_{bc} D^a_{bc} (W_a \otimes I_{bc}) && (90)\\ T_2 &= (P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) && (91)\\ T_3 &= (W_c \otimes I_{ab}) && (92)\end{aligned}$$

5.4 General case, $N = f_1 f_2 \dots f_{n_f}$

If we continue the procedure that we have used for two- and three-factors then a general pattern begins to emerge in the factorization of $W_{f_1 f_2 \dots f_{n_f}}$. To see the beginning of the pattern we can rewrite the three factor case as,

$$\begin{aligned} T_1 &= (P^a_{bc} D^a_{bc} \otimes I_1)(W_a \otimes I_{bc}) && (93)\\ T_2 &= (P^b_c D^b_c \otimes I_a)(W_b \otimes I_{ac}) && (94)\\ T_3 &= (P^c_1 D^c_1 \otimes I_{ab})(W_c \otimes I_{ab}) && (95)\end{aligned}$$

using the special cases $P^c_1 = D^c_1 = I_c$. In general, we can write the factorization of W_N for $N = f_1 f_2 \dots f_{n_f}$ as,

$$W_N = T_{n_f} \dots T_2 T_1 \quad (96)$$

where the matrix factors T_i are,

$$T_i = (P^{f_i}_{q_i} D^{f_i}_{q_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i}) \quad (97)$$

We have defined the following three additional variables p, q and m to denote different partial products of the factors,

$$\begin{aligned} p_i &= f_1 f_2 \dots f_i \quad (p_0 = 1) && (98)\\ q_i &= N/p_i && (99)\\ m_i &= N/f_i && (100)\end{aligned}$$

Note that the FFT modules W are applied before the permutations P, which makes this a decimation-in-frequency algorithm.

5.5 Implementation

Now to the implementation of the algorithm. We start with a vector of data, z, as input and want to apply the transform,

$$\begin{aligned} x &= W_N z && (101)\\ &= T_{n_f} \dots T_2 T_1 z && (102)\end{aligned}$$

where $T_i = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i})$.

The outer structure of the implementation will be a loop over the n_f factors, applying each matrix T_i to the vector in turn to build up the complete transform.

for $(i = 1 \dots n_f)$ **do**
  $v \Leftarrow T_i v$
end for

The order of the factors is not important. Now we examine the iteration $v \Leftarrow T_i v$, which we'll write as,

$$v' = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}}) \, (W_{f_i} \otimes I_{m_i}) v \tag{103}$$

There are two Kronecker product matrices in this iteration. The rightmost matrix, which is the first to be applied, is a DFT of length f_i applied to N/f_i subsets of the data. We'll call this t, since it will be a temporary array,

$$t = (W_{f_i} \otimes I_{m_i}) v \tag{104}$$

The second matrix applies a permutation and the exponential twiddle-factors. We'll call this v', since it is the result of the full iteration on v,

$$v' = (P_{q_i}^{f_i} D_{q_i}^{f_i} \otimes I_{p_{i-1}}) t \tag{105}$$

The effect of the matrix $(W_{f_i} \otimes I_{m_i})$ is best seen by an example. Suppose the factor is $f_i = 3$, and the length of the FFT is $N = 6$, then the relevant Kronecker product is,

$$t = (W_3 \otimes I_2) v \tag{106}$$

which expands out to,

$$\begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix} = \begin{pmatrix} W_3(1,1) & 0 & W_3(1,2) & 0 & W_3(1,3) & 0 \\ 0 & W_3(1,1) & 0 & W_3(1,2) & 0 & W_3(1,3) \\ W_3(2,1) & 0 & W_3(2,2) & 0 & W_3(2,3) & 0 \\ 0 & W_3(2,1) & 0 & W_3(2,2) & 0 & W_3(2,3) \\ W_3(3,1) & 0 & W_3(3,2) & 0 & W_3(3,3) & 0 \\ 0 & W_3(3,1) & 0 & W_3(3,2) & 0 & W_3(3,3) \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix} \tag{107}$$

We can rearrange the components in a computationally convenient form,

$$\begin{pmatrix} t_0 \\ t_2 \\ t_4 \\ t_1 \\ t_3 \\ t_5 \end{pmatrix} = \begin{pmatrix} W_3(1,1) & W_3(1,2) & W_3(1,3) & 0 & 0 & 0 \\ W_3(2,1) & W_3(2,2) & W_3(2,3) & 0 & 0 & 0 \\ W_3(3,1) & W_3(3,2) & W_3(3,3) & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3(1,1) & W_3(1,2) & W_3(1,3) \\ 0 & 0 & 0 & W_3(2,1) & W_3(2,2) & W_3(2,3) \\ 0 & 0 & 0 & W_3(3,1) & W_3(3,2) & W_3(3,3) \end{pmatrix} \begin{pmatrix} v_0 \\ v_2 \\ v_4 \\ v_1 \\ v_3 \\ v_5 \end{pmatrix} \tag{108}$$

which clearly shows that we just need to apply the 3×3 DFT matrix W_3 twice, once to the sub-vector of elements (v_0, v_2, v_4), and independently to the remaining sub-vector (v_1, v_3, v_5).

In the general case, if we index t as $t_k = t(\lambda, \mu) = t_{\lambda m + \mu}$ then $\lambda = 0 \dots f-1$ is an index within each transform of length f and $\mu = 0 \dots m-1$ labels the independent subsets of data. We can see this by showing the calculation with all indices present,

$$t = (W_f \otimes I_m) z \tag{109}$$

becomes,

$$t_{\lambda m+\mu} = \sum_{\lambda'=0}^{f-1}\sum_{\mu'=0}^{m-1}(W_f \otimes I_m)_{(\lambda m+\mu)(\lambda' m+\mu')} z_{\lambda' m+\mu} \quad (110)$$

$$= \sum_{\lambda'\mu'}(W_f)_{\lambda\lambda'}\delta_{\mu\mu'} z_{\lambda' m+\mu'} \quad (111)$$

$$= \sum_{\lambda'}(W_f)_{\lambda\lambda'} z_{\lambda' m+\mu} \quad (112)$$

The DFTs on the index λ will be computed using special optimized modules for each f.

To calculate the next stage,

$$v' = (P_q^f D_q^f \otimes I_{p_{i-1}})t \quad (113)$$

we note that the Kronecker product has the property of performing p_{i-1} independent multiplications of PD on q_{i-1} different subsets of t. The index μ of $t(\lambda,\mu)$ which runs from 0 to m will include q_i copies of each PD operation because $m = p_{i-1}q$, i.e.@: we can split the index μ further into $\mu = ap_{i-1} + b$, where $a = 0 \ldots q-1$ and $b = 0 \ldots p_{i-1}$,

$$\lambda m + \mu = \lambda m + ap_{i-1} + b \quad (114)$$

$$= (\lambda q + a)p_{i-1} + b. \quad (115)$$

Now we can expand the second stage,

$$v' = (P_q^f D_q^f \otimes I_{p_{i-1}})t \quad (116)$$

$$v'_{\lambda m+\mu} = \sum_{\lambda'\mu'}(P_q^f D_q^f \otimes I_{p_{i-1}})_{(\lambda m+\mu)(\lambda' m+\mu')} t_{\lambda' m+\mu'} \quad (117)$$

$$v'_{(\lambda q+a)p_{i-1}+b} = \sum_{\lambda' a' b'}(P_q^f D_q^f \otimes I_{p_{i-1}})_{((\lambda q+a)p_{i-1}+b)((\lambda' q+a')p_{i-1}+b')} t_{(\lambda' q+a')p_{i-1}+b'} \quad (118)$$

The first step in removing redundant indices is to take advantage of the identity matrix I and separate the subspaces of the Kronecker product,

$$(P_q^f D_q^f \otimes I_{p_{i-1}})_{((\lambda q+a)p_{i-1}+b)((\lambda' q+a')p_{i-1}+b')} = (P_q^f D_q^f)_{(\lambda q+a)(\lambda' q+a')}\delta_{bb'} \quad (119)$$

This eliminates one sum, leaving us with,

$$v'_{(\lambda q+a)p_{i-1}+b} = \sum_{\lambda' a'}(P_q^f D_q^f)_{(\lambda q+a)(\lambda' q+a')} t_{(\lambda' q+a')p_{i-1}+b} \quad (120)$$

We can insert the definition of D_q^f to give,

$$= \sum_{\lambda' a'}(P_q^f)_{(\lambda q+a)(\lambda' q+a')}\omega_{q_{i-1}}^{\lambda' a'} t_{(\lambda' q+a')p_{i-1}+b} \quad (121)$$

Using the definition of P_q^f, which exchanges an index of $\lambda q + a$ with $af + \lambda$, we get a final result with no matrix multiplication,

$$v'_{(af+\lambda)p_{i-1}+b} = \omega_{q_{i-1}}^{\lambda a} t_{(\lambda q+a)p_{i-1}+b} \quad (122)$$

All we have to do is premultiply each element of the temporary vector t by an exponential twiddle factor and store the result in another index location, according to the digit reversal permutation of P.

Here is the algorithm to implement the mixed-radix FFT,

for $i = 1 \ldots n_f$ **do**
  for $a = 0 \ldots q-1$ **do**
    for $b = 0 \ldots p_{i-1} - 1$ **do**
      for $\lambda = 0 \ldots f-1$ **do**
        $t_\lambda \Leftarrow \sum_{\lambda'=0}^{f-1} W_f(\lambda,\lambda') v_{b+\lambda' m + a p_{i-1}}$ {DFT matrix-multiply module}
      end for
      for $\lambda = 0 \ldots f-1$ **do**
        $v'_{(af+\lambda)p_{i-1}+b} \Leftarrow \omega_{q_{i-1}}^{\lambda a} t_\lambda$
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

5.6 Details of the implementation

First the function `gsl_fft_complex_wavetable_alloc` allocates n elements of scratch space (to hold the vector v' for each iteration) and n elements for a trigonometric lookup table of twiddle factors.

Then the length n must be factorized. There is a general factorization function `gsl_fft_factorize` which takes a list of preferred factors. It first factors out the preferred factors and then removes general remaining prime factors.

The algorithm used to generate the trigonometric lookup table is

for $a = 1 \ldots n_f$ **do**
  for $b = 1 \ldots f_i - 1$ **do**
    for $c = 1 \ldots q_i$ **do**
      trig[k++] $= \exp(-2\pi i b c p_{a-1}/N)$
    end for
  end for
end for

Note that $\sum_1^{n_f} \sum_0^{f_i-1} \sum_1^{q_i} = \sum_1^{n_f} (f_i - 1) q_i = n - 1$ so n elements are always sufficient to store the lookup table. This is chosen because we need to compute $\omega_{q_i-1}^{\lambda a} t_\lambda$ in the FFT. In terms of the lookup table we can write this as,

$$\begin{aligned} \omega_{q_{i-1}}^{\lambda a} t_\lambda &= \exp(-2\pi i \lambda a / q_{i-1}) t_\lambda && (123)\\ &= \exp(-2\pi i \lambda a p_{i-1}/N) t_\lambda && (124)\\ &= \begin{cases} t_\lambda & a = 0 \\ \text{trig[twiddle[i]} + \lambda q + (a-1)] t_\lambda & a \neq 0 \end{cases} && (125) \end{aligned}$$

The array `twiddle[i]` maintains a set of pointers into `trig` for the starting points for the outer loop. The core of the implementation is `gsl_fft_complex`. This function loops over the chosen factors of N, computing the iteration $v' = T_i v$ for each pass. When the DFT for a factor is implemented the iteration is handed-off to a dedicated small-N module, such as `gsl_fft_complex_pass_3` or `gsl_fft_complex_pass_5`. Unimplemented factors are handled by the general-N routine `gsl_fft_complex_pass_n`. The structure of one of the small-N modules is a simple transcription of the basic algorithm given above. Here is an example for

`gsl_fft_complex_pass_3`. For a pass with a factor of 3 we have to calculate the following expression,

$$v'_{(af+\lambda)p_{i-1}+b} = \sum_{\lambda'=0,1,2} \omega_{q_{i-1}}^{\lambda a} W_3^{\lambda\lambda'} v_{b+\lambda' m+ap_{i-1}} \tag{126}$$

for $b = 0 \dots p_{i-1} - 1$, $a = 0 \dots q_i - 1$ and $\lambda = 0, 1, 2$. This is implemented as,

for $a = 0 \dots q - 1$ **do**
  for $b = 0 \dots p_{i-1} - 1$ **do**

$$\begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} W_3^0 & W_3^0 & W_3^0 \\ W_3^0 & W_3^1 & W_3^2 \\ W_3^0 & W_3^2 & W_3^4 \end{pmatrix} \begin{pmatrix} v_{b+ap_{i-1}} \\ v_{b+ap_{i-1}+m} \\ v_{b+ap_{i-1}+2m} \end{pmatrix}$$

$v'_{ap_i+b} = t_0$
$v'_{ap_i+b+p_{i-1}} = \omega_{q_{i-1}}^{a} t_1$
$v'_{ap_i+b+2p_{i-1}} = \omega_{q_{i-1}}^{2a} t_2$
  end for
end for

In the code we use the variables `from0`, `from1`, `from2` to index the input locations,

$$\texttt{from0} = b + ap_{i-1} \tag{127}$$
$$\texttt{from1} = b + ap_{i-1} + m \tag{128}$$
$$\texttt{from2} = b + ap_{i-1} + 2m \tag{129}$$

and the variables `to0`, `to1`, `to2` to index the output locations in the scratch vector v',

$$\texttt{to0} = b + ap_i \tag{130}$$
$$\texttt{to1} = b + ap_i + p_{i-1} \tag{131}$$
$$\texttt{to2} = b + ap_i + 2p_{i-1} \tag{132}$$

The DFT matrix multiplication is computed using the optimized sub-transform modules given in the next section. The twiddle factors $\omega_{q_{i-1}}^{a}$ are taken out of the `trig` array.

To compute the inverse transform we go back to the definition of the Fourier transform and note that the inverse matrix is just the complex conjugate of the forward matrix (with a factor of $1/N$),

$$W_N^{-1} = W_N^*/N \tag{133}$$

Therefore we can easily compute the inverse transform by conjugating all the complex elements of the DFT matrices and twiddle factors that we apply. (An alternative strategy is to conjugate the input data, take a forward transform, and then conjugate the output data).

6 Fast Sub-transform Modules

To implement the mixed-radix FFT we still need to compute the small-N DFTs for each factor. Fortunately many highly-optimized small-N modules are available, following the work of Winograd who showed how to derive efficient small-N sub-transforms by number theoretic techniques.

The algorithms in this section all compute,[1]

$$x_a = \sum_{b=0}^{N-1} W_N^{ab} z_b \tag{134}$$

The sub-transforms given here are the ones recommended by Temperton and differ slightly from the canonical Winograd modules. According to [19] they are slightly more robust against rounding errors and trade off some additions for multiplications. For the $N = 2$ DFT,

$$x_0 = z_0 + z_1, \quad x_1 = z_0 - z_1. \tag{135}$$

For the $N = 3$ DFT,

$$t_1 = z_1 + z_2, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_1 - z_2), \tag{136}$$

$$x_0 = z_0 + t_1, \quad x_1 = t_2 + it_3, \quad x_2 = t_2 - it_3. \tag{137}$$

The $N = 4$ transform involves only additions and subtractions,

$$t_1 = z_0 + z_2, \quad t_2 = z_1 + z_3, \quad t_3 = z_0 - z_2, \quad t_4 = z_1 - z_3, \tag{138}$$

$$x_0 = t_1 + t_2, \quad x_1 = t_3 + it_4, \quad x_2 = t_1 - t_2, \quad x_3 = t_3 - it_4. \tag{139}$$

For the $N = 5$ DFT,

$$t_1 = z_1 + z_4, \quad t_2 = z_2 + z_3, \quad t_3 = z_1 - z_4, \quad t_4 = z_2 - z_3, \tag{140}$$

$$t_5 = t_1 + t_2, \quad t_6 = (\sqrt{5}/4)(t_1 - t_2), \quad t_7 = z_0 - t_5/4, \tag{141}$$

$$t_8 = t_7 + t_6, \quad t_9 = t_7 - t_6, \tag{142}$$

$$t_{10} = \sin(2\pi/5)t_3 + \sin(2\pi/10)t_4, \quad t_{11} = \sin(2\pi/10)t_3 - \sin(2\pi/5)t_4, \tag{143}$$

$$x_0 = z_0 + t_5, \tag{144}$$

$$x_1 = t_8 + it_{10}, \quad x_2 = t_9 + it_{11}, \tag{145}$$

$$x_3 = t_9 - it_{11}, \quad x_4 = t_8 - it_{10}. \tag{146}$$

The DFT matrix for $N = 6$ can be written as a combination of $N = 3$ and $N = 2$ transforms with index permutations,

$$\left(\begin{array}{c} x_0 \\ x_4 \\ \hline x_2 \\ x_3 \\ x_1 \\ x_5 \end{array}\right) = \left(\begin{array}{c|c} W_3 & W_3 \\ \hline W_3 & -W_3 \end{array}\right) \left(\begin{array}{c} z_0 \\ z_2 \\ z_4 \\ \hline z_3 \\ z_5 \\ z_1 \end{array}\right) \tag{147}$$

This simplification is an example of the Prime Factor Algorithm, which can be used because the factors 2 and 3 are mutually prime. For more details consult one of the books on number

[1] Erratum: due to a difference in sign convention, these transforms are actually for $x_a = \sum_{b=0}^{N-1} W_N^{-ab} z_b$. Thanks to Andrew Holme for the correction.

theory for FFTs [6, 7]. We can take advantage of the simple indexing scheme of the PFA to write the $N = 6$ DFT as,

$$t_1 = z_2 + z_4, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_2 - z_4), \tag{148}$$

$$t_4 = z_5 + z_1, \quad t_5 = z_3 - t_4/2, \quad t_6 = \sin(\pi/3)(z_5 - z_1), \tag{149}$$

$$t_7 = z_0 + t_1, \quad t_8 = t_2 + it_3, \quad t_9 = t_2 - it_3, \tag{150}$$

$$t_{10} = z_3 + t_4, \quad t_{11} = t_5 + it_6, \quad t_{12} = t_5 - it_6, \tag{151}$$

$$x_0 = t_7 + t_{10}, \quad x_4 = t_8 + t_{11}, \quad x_2 = t_9 + t_{12}, \tag{152}$$

$$x_3 = t_7 - t_{10}, \quad x_1 = t_8 - t_{11}, \quad x_5 = t_9 - t_{12}. \tag{153}$$

For any remaining general factors we use Singleton's efficient method for computing a DFT [20]. Although it is an $O(N^2)$ algorithm it does reduce the number of multiplications by a factor of 4 compared with a naive evaluation of the DFT. If we look at the general structure of a DFT matrix, shown schematically below,

$$\begin{pmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \\ h_{N-2} \\ h_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & W & W & \cdots & W & W \\ 1 & W & W & \cdots & W & W \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & W & W & \cdots & W & W \\ 1 & W & W & \cdots & W & W \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_{N-2} \\ g_{N-1} \end{pmatrix} \tag{154}$$

we see that the outer elements of the DFT matrix are all unity. We can remove these trivial multiplications but we will still be left with an $(N-1) \times (N-1)$ sub-matrix of complex entries, which would appear to require $(N-1)^2$ complex multiplications. Singleton's method, uses symmetries of the DFT matrix to convert the complex multiplications to an equivalent number of real multiplications. We start with the definition of the DFT in component form,

$$a_k + ib_k = \sum_{j=0} (x_j + iy_j)(\cos(2\pi jk/f) - i\sin(2\pi jk/f)) \tag{155}$$

The zeroth component can be computed using only additions,

$$a_0 + ib_0 = \sum_{j=0}^{(f-1)} x_j + iy_j \tag{156}$$

We can rewrite the remaining components as,

$$a_k + ib_k = x_0 + iy_0 + \sum_{j=1}^{(f-1)/2} (x_j + x_{f-j})\cos(2\pi jk/f) + (y_j - y_{f-j})\sin(2\pi jk/f) \tag{157}$$

$$+i \sum_{j=1}^{(f-1)/2} (y_j + y_{f-j})\cos(2\pi jk/f) - (x_j - x_{f-j})\sin(2\pi jk/f) \tag{158}$$

by using the following trigonometric identities,

$$\cos(2\pi(f-j)k/f) = \cos(2\pi jk/f) \quad (159)$$
$$\sin(2\pi(f-j)k/f) = -\sin(2\pi jk/f) \quad (160)$$

These remaining components can all be computed using four partial sums,

$$a_k + ib_k = (a_k^+ - a_k^-) + i(b_k^+ + b_k^-) \quad (161)$$
$$a_{f-k} + ib_{f-k} = (a_k^+ + a_k^-) + i(b_k^+ - b_k^-) \quad (162)$$

for $k = 1, 2, \ldots, (f-1)/2$, where,

$$a_k^+ = x_0 + \sum_{j=1}^{(f-1)/2} (x_j + x_{f-j}) \cos(2\pi jk/f) \quad (163)$$
$$a_k^- = -\sum_{j=1}^{(f-1)/2} (y_j - y_{f-j}) \sin(2\pi jk/f) \quad (164)$$
$$b_k^+ = y_0 + \sum_{j=1}^{(f-1)/2} (y_j + y_{f-j}) \cos(2\pi jk/f) \quad (165)$$
$$b_k^- = -\sum_{j=1}^{(f-1)/2} (x_j - x_{f-j}) \sin(2\pi jk/f) \quad (166)$$

Note that the higher components $k' = f - k$ can be obtained directly without further computation once a^+, a^-, b^+ and b^- are known. There are $4 \times (f-1)/2$ different sums, each involving $(f-1)/2$ real multiplications, giving a total of $(f-1)^2$ real multiplications instead of $(f-1)^2$ complex multiplications.

To implement Singleton's method we make use of the input and output vectors v and v' as scratch space, copying data back and forth between them to obtain the final result. First we use v' to store the terms of the symmetrized and anti-symmetrized vectors of the form $x_j + x_{f-j}$ and $x_j - y_{f-j}$. Then we multiply these by the appropriate trigonometric factors to compute the partial sums a^+, a^-, b^+ and b^-, storing the results $a_k + ib_k$ and $a_{f-k} + ib_{f-k}$ back in v. Finally we multiply the DFT output by any necessary twiddle factors and place the results in v'.

7 FFTs for real data

This section is based on the articles *Fast Mixed-Radix Real Fourier Transforms* by [21] and *Real-Valued Fast Fourier Transform Algorithms* by [22]. The DFT of a real sequence has a special symmetry, called a *conjugate-complex* or *half-complex* symmetry,

$$h(a) = h(N-a)^* \quad (167)$$

The element $h(0)$ is real, and when N is even $h(N/2)$ is also real. It is straightforward to prove the symmetry,

$$\begin{aligned} h(a) &= \sum W_N^{ab} g(b) && (168)\\ h(N-a)^* &= \sum W_N^{-(N-a)b} g(b)^* && (169)\\ &= \sum W_N^{-Nb} W_N^{ab} g(b) \qquad (W_N^N = 1) && (170)\\ &= \sum W_N^{ab} g(b) && (171) \end{aligned}$$

Real-valued data is very common in practice (perhaps more common that complex data) so it is worth having efficient FFT routines for real data. In principle an FFT for real data should need half the operations of an FFT on the equivalent complex data (where the imaginary parts are set to zero). There are two different strategies for computing FFTs of real-valued data:

One strategy is to "pack" the real data (of length N) into a complex array (of length $N/2$) by index transformations. A complex FFT routine can then be used to compute the transform of that array. By further index transformations the result can actually by "unpacked" to the FFT of the original real data. It is also possible to do two real FFTs simultaneously by packing one in the real part and the other in the imaginary part of the complex array. These techniques have some disadvantages. The packing and unpacking procedures always add $O(N)$ operations, and packing a real array of length N into a complex array of length $N/2$ is only possible if N is even. In addition, if two unconnected datasets with very different magnitudes are packed together in the same FFT there could be "cross-talk" between them due to a loss of precision.

A more straightforward strategy is to start with an FFT algorithm, such as the complex mixed-radix algorithm, and prune out all the operations involving the zero imaginary parts of the initial data. The FFT is linear so the imaginary part of the data can be decoupled from the real part. This procedure leads to a dedicated FFT for real-valued data which works for any length and does not perform any unnecessary operations. It also allows us to derive a corresponding inverse FFT routine which transforms a half-complex sequence back into real data.

7.1 Radix-2 FFTs for real data

Before embarking on the full mixed-radix real FFT we'll start with the radix-2 case. It contains all the essential features of the general-N algorithm. To make it easier to see the analogy between the two we will use the mixed-radix notation to describe the factors. The factors are all 2,

$$f_1 = 2, f_2 = 2, \ldots, f_{n_f} = 2 \qquad (172)$$

and the products p_i are powers of 2,

$$\begin{aligned} p_0 &= 1 && (173)\\ p_1 &= f_1 = 2 && (174)\\ p_2 &= f_1 f_2 = 4 && (175)\\ \ldots &= \ldots && (176)\\ p_i &= f_1 f_2 \ldots f_i = 2^i && (177) \end{aligned}$$

Using this notation we can rewrite the radix-2 decimation-in-time algorithm as,

bit-reverse ordering of g
for $i = 1 \ldots n$ **do**
  for $a = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q_i - 1$ **do**

$$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} + a) \end{pmatrix} = \begin{pmatrix} g(bp_i + a) + W_{p_i}^a g(bp_i + p_{i-1} + a) \\ g(bp_i + a) - W_{p_i}^a g(bp_i + p_{i-1} + a) \end{pmatrix}$$

    end for
  end for
end for

where we have used $p_i = 2\Delta$, and factored 2Δ out of the original definition of b ($b \to bp_i$).

If we go back to the original recurrence relations we can see how to write the intermediate results in a way which make the real/half-complex symmetries explicit at each step. The first pass is just a set of FFTs of length-2 on real values,

$$g_1([b_0 b_1 b_2 a_0]) = \sum_{b_3} W_2^{a_0 b_3} g([b_0 b_1 b_2 b_3]) \tag{178}$$

Using the symmetry $FFT(x)_k = FFT(x)^*_{N-k}$ we have the reality condition,

$$g_1([b_0 b_1 b_2 0]) = \text{real} \tag{179}$$
$$g_1([b_0 b_1 b_2 1]) = \text{real'} \tag{180}$$

In the next pass we have a set of length-4 FFTs on the original data,

$$g_2([b_0 b_1 b_1 a_0]) = \sum_{b_2} \sum_{b_3} W_4^{[a_1 a_0] b_2} W_2^{a_0 b_3} g([b_0 b_1 b_2 b_3]) \tag{181}$$
$$= \sum_{b_2} \sum_{b_3} W_4^{[a_1 a_0][b_3 b_2]} g([b_0 b_1 b_2 b_3]) \tag{182}$$

This time symmetry gives us the following conditions on the transformed data,

$$g_2([b_0 b_1 00]) = \text{real} \tag{183}$$
$$g_2([b_0 b_1 01]) = x + iy \tag{184}$$
$$g_2([b_0 b_1 10]) = \text{real'} \tag{185}$$
$$g_2([b_0 b_1 11]) = x - iy \tag{186}$$

We can see a pattern emerging here: the i-th pass computes a set of independent length-2^i FFTs on the original real data,

$$g_i(bp_i + a) = \sum_{a'=0}^{p_i - 1} W_{p_i}^{aa'} g(bp_i + a') \quad \text{for } b = 0 \ldots q_i - 1 \tag{187}$$

As a consequence the we can apply the symmetry for an FFT of real data to all the intermediate results – not just the final result. In general after the i-th pass we will have the symmetry,

$$g_i(bp_i) = \text{real} \tag{188}$$
$$g_i(bp_i + a) = g_i(bp_i + p_i - a)^* \qquad a = 1 \ldots p_i/2 - 1 \tag{189}$$
$$g_i(bp_i + p_i/2) = \text{real'} \tag{190}$$

In the next section we'll show that this is a general property of decimation-in-time algorithms. The same is not true for the decimation-in-frequency algorithm, which does not have any simple symmetries in the intermediate results.

Since we can obtain the values of $g_i(bp_i + a)$ for $a > p_i/2$ from the values for $a < p_i/2$ we can cut our computation and storage in half compared with the full-complex case. We can easily rewrite the algorithm to show how the computation can be halved, simply by limiting all terms to involve only values for $a \leq p_i/2$. Whenever we encounter a term $g_i(bp_i + a)$ with $a > p_i/2$ we rewrite it in terms of its complex symmetry partner, $g_i(bp_i + a')^*$, where $a' = p_i - a$. The butterfly computes two values for each value of a, $bp_i + a$ and $bp_i + p_{i-1} - a$, so we actually only need to compute from $a = 0$ to $p_{i-1}/2$. This gives the following algorithm,

for $a = 0 \ldots p_{i-1}/2$ **do**
  for $b = 0 \ldots q_i - 1$ **do**

$$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} - a)^* \end{pmatrix} = \begin{pmatrix} g(bp_i + a) + W_{p_i}^a g(bp_i + p_{i-1} + a) \\ g(bp_i + a) - W_{p_i}^a g(bp_i + p_{i-1} + a) \end{pmatrix}$$

  end for
end for

Although we have halved the number of operations we also need a storage arrangement which will halve the memory requirement. The algorithm above is still formulated in terms of a complex array g, but the input to our routine will naturally be an array of N real values which we want to use in-place.

Therefore we need a storage scheme which lays out the real and imaginary parts within the real array, in a natural way so that there is no need for complicated index calculations. In the radix-2 algorithm we do not have any additional scratch space. The storage scheme has to be designed to accommodate the in-place calculation taking account of dual node pairs.

Here is a scheme which takes these restrictions into account: On the i-th pass we store the real part of $g(bp_i+a)$ in location bp_i+a. We store the imaginary part in location bp_i+p_i-a. This is the redundant location which corresponds to the conjugate term $g(bp_i+a)^* = g(bp_i+p_i-a)$, so it is not needed. When the results are purely real (as in the case $a = 0$ and $a = p_i/2$ we store only the real part and drop the zero imaginary part).

This storage scheme has to work in-place, because the radix-2 routines should not use any scratch space. We will now check the in-place property for each butterfly. A crucial point is that the scheme is pass-dependent. Namely, when we are computing the result for pass i we are reading the results of pass $i - 1$, and we must access them using the scheme from the previous pass, i.e. we have to remember that the results from the previous pass were stored using $bp_{i-1} + a$, not $bp_i + a$, and the symmetry for these results will be $g_{i-1}(bp_{i-1} + a) = g_{i-1}(bp_{i-1} + p_{i-1} - a)^*$. To take this into account we'll write the right hand side of the iteration, g_{i-1}, in terms of p_{i-1}. For example, instead of bp_i, which occurs naturally in $g_i(bp_i + a)$ we will use $2bp_{i-1}$, since $p_i = 2p_{i-1}$.

Let's start with the butterfly for $a = 0$,

$$\begin{pmatrix} g(bp_i) \\ g(bp_i + p_{i-1})^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1}) + g((2b+1)p_{i-1}) \\ g(2bp_{i-1}) - g((2b+1)p_{i-1}) \end{pmatrix} \tag{191}$$

By the symmetry $g_{i-1}(bp_{i-1} + a) = g_{i-1}(bp_{i-1} + p_{i-1} - a)^*$ all the inputs are purely real. The input $g(2bp_{i-1})$ is read from location $2bp_{i-1}$ and $g((2b + 1)p_{i-1})$ is read from the location $(2b + 1)p_{i-1}$. Here is the full breakdown,

Term	Location		
$g(2bp_{i-1})$	real part	$2bp_{i-1}$	$= bp_i$
	imag part	—	
$g((2b+1)p_{i-1})$	real part	$(2b+1)p_{i-1}$	$= bp_i + p_{i-1}$
	imag part	—	
$g(bp_i)$	real part	bp_i	
	imag part	—	
$g(bp_i + p_{i-1})$	real part	$bp_i + p_{i-1}$	
	imag part	—	

The conjugation of the output term $g(bp_i + p_{i-1})^*$ is irrelevant here since the results are purely real. The real results are stored in locations bp_i and $bp_i + p_{i-1}$, which overwrites the inputs in-place.

The general butterfly for $a = 1 \ldots p_{i-1}/2 - 1$ is,

$$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} - a)^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1} + a) + W_{p_i}^a g((2b+1)p_{i-1} + a) \\ g(2bp_{i-1} + a) - W_{p_i}^a g((2b+1)p_{i-1} + a) \end{pmatrix} \tag{192}$$

All the terms are complex. To store a conjugated term like $g(b'p_i + a')^*$ where $a > p_i/2$ we take the real part and store it in location $b'p_i + a'$ and then take imaginary part, negate it, and store the result in location $b'p_i + p_i - a'$.

Here is the full breakdown of the inputs and outputs from the butterfly,

Term	Location		
$g(2bp_{i-1} + a)$	real part	$2bp_{i-1} + a$	$= bp_i + a$
	imag part	$2bp_{i-1} + p_{i-1} - a$	$= bp_i + p_{i-1} - a$
$g((2b+1)p_{i-1} + a)$	real part	$(2b+1)p_{i-1} + a$	$= bp_i + p_{i-1} + a$
	imag part	$(2b+1)p_{i-1} + p_{i-1} - a$	$= bp_i + p_i - a$
$g(bp_i + a)$	real part	$bp_i + a$	
	imag part	$bp_i + p_i - a$	
$g(bp_i + p_{i-1} - a)$	real part	$bp_i + p_{i-1} - a$	
	imag part	$bp_i + p_{i-1} + a$	

By comparing the input locations and output locations we can see that the calculation is done in place.

The final butterfly for $a = p_{i-1}/2$ is,

$$\begin{pmatrix} g(bp_i + p_{i-1}/2) \\ g(bp_i + p_{i-1} - p_{i-1}/2)^* \end{pmatrix} = \begin{pmatrix} g(2bp_{i-1} + p_{i-1}/2) - ig((2b+1)p_{i-1} + p_{i-1}/2) \\ g(2bp_{i-1} + p_{i-1}/2) + ig((2b+1)p_{i-1} + p_{i-1}/2) \end{pmatrix} \tag{193}$$

where we have substituted for the twiddle factor, $W_{p_i}^a = -i$,

$$\begin{aligned} W_{p_i}^{p_{i-1}/2} &= \exp(-2\pi i p_{i-1}/2p_i) && (194)\\ &= \exp(-2\pi i/4) && (195)\\ &= -i && (196)\end{aligned}$$

For this butterfly the second line is just the conjugate of the first, because $p_{i-1} - p_{i-1}/2 = p_{i-1}/2$. Therefore we only need to consider the first line. The breakdown of the inputs and outputs is,

Term			
$g(2bp_{i-1} + p_{i-1}/2)$	real part	$2bp_{i-1} + p_{i-1}/2$	$= bp_i + p_{i-1}/2$
	imag part	—	
$g((2b+1)p_{i-1} + p_{i-1}/2)$	real part	$(2b+1)p_{i-1} + p_{i-1}/2$	$= bp_i + p_i - p_{i-1}/2$
	imag part	—	
$g(bp_i + p_{i-1}/2)$	real part	$bp_i + p_{i-1}/2$	
	imag part	$bp_i + p_i - p_{i-1}/2$	

By comparing the locations of the inputs and outputs with the operations in the butterfly we find that this computation is very simple: the effect of the butterfly is to negate the location $bp_i + p_i - p_{i-1}/2$ and leave other locations unchanged. This is clearly an in-place operation.

Here is the radix-2 algorithm for real data, in full, with the cases of $a = 0$, $a = 1 \ldots p_{i-1}/2-1$ and $a = p_{i-1}/2$ in separate blocks,

bit-reverse ordering of g
for $i = 1 \ldots n$ **do**
  for $b = 0 \ldots q_i - 1$ **do**
    $\begin{pmatrix} g(bp_i) \\ g(bp_i + p_{i-1}) \end{pmatrix} \Leftarrow \begin{pmatrix} g(bp_i) + g(bp_i + p_{i-1}) \\ g(bp_i) - g(bp_i + p_{i-1}) \end{pmatrix}$
  end for
  for $a = 1 \ldots p_{i-1}/2 - 1$ **do**
    for $b = 0 \ldots q_i - 1$ **do**
      $(\mathrm{Re}\, z_0, \mathrm{Im}\, z_0) \Leftarrow (g(bp_i + a), g(bp_i + p_{i-1} - a))$
      $(\mathrm{Re}\, z_1, \mathrm{Im}\, z_1) \Leftarrow (g(bp_i + p_{i-1} + a), g(bp_i + p_i - a))$
      $t_0 \Leftarrow z_0 + W_{p_i}^a z_1$
      $t_1 \Leftarrow z_0 - W_{p_i}^a z_1$
      $(g(bp_i + a), g(bp_i + p_i - a) \Leftarrow (\mathrm{Re}\, t_0, \mathrm{Im}\, t_0)$
      $(g(bp_i + p_{i-1} - a), g(bp_i + p_{i-1} + a)) \Leftarrow (\mathrm{Re}\, t_1, -\mathrm{Im}\, t_1)$
    end for
  end for
  for $b = 0 \ldots q_i - 1$ **do**
    $g(bp_i + p_i - p_{i-1}/2) \Leftarrow -g(bp_i + p_i - p_{i-1}/2)$
  end for
end for

We split the loop over a into three parts, $a = 0$, $a = 1 \dots p_{i-1}/2 - 1$ and $a = p_{i-1}/2$ for efficiency. When $a = 0$ we have $W^a_{p_i} = 1$ so we can eliminate a complex multiplication within the loop over b. When $a = p_{i-1}/2$ we have $W^a_{p_i} = -i$ which does not require a full complex multiplication either.

7.1.1 Calculating the Inverse

The inverse FFT of complex data was easy to calculate, simply by taking the complex conjugate of the DFT matrix. The input data and output data were both complex and did not have any special symmetry. For real data the inverse FFT is more complicated because the half-complex symmetry of the transformed data is different from the purely real input data.

We can compute an inverse by stepping backwards through the forward transform. To simplify the inversion it's convenient to write the forward algorithm with the butterfly in matrix form,

for $i = 1 \dots n$ **do**
  for $a = 0 \dots p_{i-1}/2$ **do**
    for $b = 0 \dots q_i - 1$ **do**

$$\begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} + a) \end{pmatrix} = \begin{pmatrix} 1 & W^a_{p_i} \\ 1 & -W^a_{p_i} \end{pmatrix} \begin{pmatrix} g(2bp_{i-1} + a) \\ g((2b+1)p_{i-1} + a) \end{pmatrix}$$

    end for
  end for
end for

To invert the algorithm we run the iterations backwards and invert the matrix multiplication in the innermost loop,

for $i = n \dots 1$ **do**
  for $a = 0 \dots p_{i-1}/2$ **do**
    for $b = 0 \dots q_i - 1$ **do**

$$\begin{pmatrix} g(2bp_{i-1} + a) \\ g((2b+1)p_{i-1} + a) \end{pmatrix} = \begin{pmatrix} 1 & W^a_{p_i} \\ 1 & -W^a_{p_i} \end{pmatrix}^{-1} \begin{pmatrix} g(bp_i + a) \\ g(bp_i + p_{i-1} + a) \end{pmatrix}$$

    end for
  end for
end for

There is no need to reverse the loops over a and b because the result is independent of their order. The inverse of the matrix that appears is,

$$\begin{pmatrix} 1 & W^a_{p_i} \\ 1 & -W^a_{p_i} \end{pmatrix}^{-1} = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ W^{-a}_{p_i} & -W^{-a}_{p_i} \end{pmatrix} \tag{197}$$

To save divisions we remove the factor of $1/2$ inside the loop. This computes the unnormalized inverse, and the normalized inverse can be retrieved by dividing the final result by $N = 2^n$.

Here is the radix-2 half-complex to real inverse FFT algorithm, taking into account the radix-2 storage scheme,

for $i = n \ldots 1$ **do**
  for $b = 0 \ldots q_i - 1$ **do**
    $$\begin{pmatrix} g(bp_i) \\ g(bp_i + p_{i-1}) \end{pmatrix} \Leftarrow \begin{pmatrix} g(bp_i) + g(bp_i + p_{i-1}) \\ g(bp_i) - g(bp_i + p_{i-1}) \end{pmatrix}$$
  end for
  for $a = 1 \ldots p_{i-1}/2 - 1$ **do**
    for $b = 0 \ldots q_i - 1$ **do**
      $(\mathrm{Re}\, z_0, \mathrm{Im}\, z_0) \Leftarrow (g(bp_i + a), g(bp_i + p_i - a))$
      $(\mathrm{Re}\, z_1, \mathrm{Im}\, z_1) \Leftarrow (g(bp_i + p_{i-1} - a), -g(bp_i + p_{i-1} + a))$
      $t_0 \Leftarrow z_0 + z_1$
      $t_1 \Leftarrow z_0 - z_1$
      $(g(bp_i + a), g(bp_i + p_{i-1} - a)) \Leftarrow (\mathrm{Re}\, t_0, \mathrm{Im}\, t_0)$
      $(g(bp_i + p_{i-1} + a), g(bp_i + p_i - a)) \Leftarrow (\mathrm{Re}(W^a_{p_i} t_1), \mathrm{Im}(W^a_{p_i} t_1))$
    end for
  end for
  for $b = 0 \ldots q_i - 1$ **do**
    $g(bp_i + p_{i-1}/2) \Leftarrow 2g(bp_i + p_{i-1}/2)$
    $g(bp_i + p_{i-1} + p_{i-1}/2) \Leftarrow -2g(bp_i + p_{i-1} + p_{i-1}/2)$
  end for
end for
bit-reverse ordering of g

7.2 Mixed-Radix FFTs for real data

As discussed earlier the radix-2 decimation-in-time algorithm had the special property that its intermediate passes are interleaved Fourier transforms of the original data, and this generalizes to the mixed-radix algorithm. The complex mixed-radix algorithm that we derived earlier was a decimation-in-frequency algorithm, but we can obtain a decimation-in-time version by taking the transpose of the decimation-in-frequency DFT matrix like this,

$$\begin{aligned} W_N &= W_N^T && (198)\\ &= (T_{n_f} \ldots T_2 T_1)^T && (199)\\ &= T_1^T T_2^T \ldots T_{n_f}^T && (200) \end{aligned}$$

with,

$$\begin{aligned} T_i^T &= \left((P^{f_i}_{q_i} D^{f_i}_{q_i} \otimes I_{p_{i-1}})(W_{f_i} \otimes I_{m_i})\right)^T && (201)\\ &= (W_{f_i} \otimes I_{m_i})(D^{f_i}_{q_i} (P^{f_i}_{q_i})^T \otimes I_{p_{i-1}}). && (202) \end{aligned}$$

We have used the fact that W, D and I are symmetric and that the permutation matrix P obeys,

$$(P^a_b)^T = P^b_a. \tag{203}$$

From the definitions of D and P we can derive the following identity,

$$D^a_b P^b_a = P^b_a D^b_a. \tag{204}$$

This allows us to put the transpose into a simple form,

$$T_i^T = (W_{f_i} \otimes I_{m_i})(P^{q_i}_{f_i} D^{q_i}_{f_i} \otimes I_{p_{i-1}}). \tag{205}$$

The transposed matrix, T^T applies the digit-reversal P before the DFT W, giving the required decimation-in-time algorithm. The transpose reverses the order of the factors — T_{n_f} is applied first and T_1 is applied last. For convenience we can reverse the order of the factors, $f_1 \leftrightarrow f_{n_f}$, $f_2 \leftrightarrow f_{n_f-1}$, ...and make the corresponding substitution $p_{i-1} \leftrightarrow q_i$. These substitutions give us a decimation-in-time algorithm with the same ordering as the decimation-in-frequency algorithm,

$$W_N = T_{n_f} \dots T_2 T_1 \tag{206}$$

$$T_i = (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i}) \tag{207}$$

where p_i, q_i and m_i now have the same meanings as before, namely,

$$p_i = f_1 f_2 \dots f_i \quad (p_0 = 1) \tag{208}$$

$$q_i = N/p_i \tag{209}$$

$$m_i = N/f_i \tag{210}$$

Now we would like to prove that the iteration for computing $x = Wz = T_{n_f} \dots T_2 T_1 z$ has the special property interleaving property. If we write the result of each intermediate pass as $v^{(i)}$,

$$v^{(0)} = z \tag{211}$$

$$v^{(1)} = T_1 v^{(0)} \tag{212}$$

$$v^{(2)} = T_2 v^{(1)} \tag{213}$$

$$\dots = \dots \tag{214}$$

$$v^{(i)} = T_i v^{(i-1)} \tag{215}$$

then we will show that the intermediate results $v^{(i)}$ on any pass can be written as,

$$v^{(i)} = (W_{p_i} \otimes I_{q_i}) z \tag{216}$$

Each intermediate stage will be a set of q_i interleaved Fourier transforms, each of length p_i. We can prove this result by induction. First we assume that the result is true for $v^{(i-1)}$,

$$v^{(i-1)} = (W_{p_{i-1}} \otimes I_{q_{i-1}}) z \qquad \text{(assumption)} \tag{217}$$

And then we examine the next iteration using this assumption,

$$v^{(i)} = T_i v^{(i-1)} \tag{218}$$

$$= T_i (W_{p_{i-1}} \otimes I_{q_{i-1}}) z \tag{219}$$

$$= (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i})(W_{p_{i-1}} \otimes I_{q_{i-1}}) z \tag{220}$$

Using the relation $m_i = p_{i-1} q_i$, we can write I_{m_i} as $I_{p_{i-1} q_i}$ and $I_{q_{i-1}}$ as $I_{f_i q_i}$. By combining these with the basic matrix identity,

$$I_{ab} = I_a \otimes I_b \tag{221}$$

we can rewrite $v^{(i)}$ in the following form,

$$v^{(i)} = (((W_{f_i} \otimes I_{p_{i-1}})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})(W_{p_{i-1}} \otimes I_{f_i})) \otimes I_{q_i}) z. \tag{222}$$

The first part of this matrix product is the two-factor expansion of W_{ab}, for $a = p_{i-1}$ and $b = f_i$,

$$W_{p_{i-1}f_i} = ((W_{f_i} \otimes I_{p_{i-1}})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})(W_{p_{i-1}} \otimes I_{f_i})). \tag{223}$$

If we substitute this result, remembering that $p_i = p_{i-1} f_i$, we obtain,

$$v^{(i)} = (W_{p_i} \otimes I_{q_i}) z \tag{224}$$

which is the desired result. The case $i = 1$ can be verified explicitly, and induction then shows that the result is true for all i. As discussed for the radix-2 algorithm this result is important because if the initial data z is real then each intermediate pass is a set of interleaved Fourier transforms of z, having half-complex symmetries (appropriately applied in the subspaces of the Kronecker product). Consequently only N real numbers are needed to store the intermediate and final results.

7.3 Implementation

The implementation of the mixed-radix real FFT algorithm follows the same principles as the full complex transform. Some of the steps are applied in the opposite order because we are dealing with a decimation in time algorithm instead of a decimation in frequency algorithm, but the basic outer structure of the algorithm is the same. We want to apply the factorized version of the DFT matrix W_N to the input vector z,

$$\begin{aligned} x &= W_N z && (225)\\ &= T_{n_f} \dots T_2 T_1 z && (226) \end{aligned}$$

We loop over the n_f factors, applying each matrix T_i to the vector in turn to build up the complete transform,

for $(i = 1 \dots n_f)$ **do**
  $v \Leftarrow T_i v$
end for

In this case the definition of T_i is different because we have taken the transpose,

$$T_i = (W_{f_i} \otimes I_{m_i})(P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i}). \tag{227}$$

We'll define a temporary vector t to denote the results of applying the rightmost matrix,

$$t = (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i}) v \tag{228}$$

If we expand this out into individual components, as before, we find a similar simplification,

$$\begin{aligned} t_{aq+b} &= \sum_{a'b'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}} \otimes I_{q_i})_{(aq+b)(a'q+b')} v_{a'q+b'} && (229)\\ &= \sum_{a'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})_{aa'} v_{a'q+b} && (230) \end{aligned}$$

We have factorized the indices into the form $aq + b$, with $0 \le a < p_i$ and $0 \le b < q$. Just as in the decimation in frequency algorithm we can split the index a to remove the matrix multiplication completely. We have to write a as $\mu f + \lambda$, where $0 \le \mu < p_{i-1}$ and $0 \le \lambda < f$,

$$t_{(\mu f+\lambda)q+b} = \sum_{\mu'\lambda'} (P_{f_i}^{p_{i-1}} D_{f_i}^{p_{i-1}})_{(\mu f+\lambda)(\mu' f+\lambda')} v_{(\mu' f+\lambda')q+b} \quad (231)$$

$$= \sum_{\mu'\lambda'} (P_{f_i}^{p_{i-1}})_{(\mu f+\lambda)(\mu' f+\lambda')} \omega_{p_i}^{\mu'\lambda'} v_{(\mu' f+\lambda')q+b} \quad (232)$$

The matrix $P_{f_i}^{p_{i-1}}$ exchanges an index of $(\mu f + \lambda)q + b$ with $(\lambda p_{i-1} + \mu)q + b$, giving a final result of,

$$t_{(\lambda p_{i-1}+\mu)q+b} = w_{p_i}^{\mu\lambda} v_{(\mu f+\lambda)q+b} \quad (233)$$

To calculate the next stage,

$$v' = (W_{f_i} \otimes I_{m_i}) t, \quad (234)$$

we temporarily rearrange the index of t to separate the m_i independent DFTs in the Kronecker product,

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'\mu'b'} (W_{f_i} \otimes I_{m_i})_{((\lambda p_{i-1}+\mu)q+b)((\lambda' p_{i-1}+\mu')q+b')} t_{(\lambda' p_{i-1}+\mu')q+b'} \quad (235)$$

If we use the identity $m = p_{i-1}q$ to rewrite the index of t like this,

$$t_{(\lambda p_{i-1}+\mu)q+b} = t_{\lambda m+(\mu q+b)} \quad (236)$$

we can split the Kronecker product,

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'\mu'b'} (W_{f_i} \otimes I_{m_i})_{((\lambda p_{i-1}+\mu)q+b)((\lambda' p_{i-1}+\mu')q+b')} t_{(\lambda' p_{i-1}+\mu')q+b'} \quad (237)$$

$$= \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} t_{\lambda' m_i+(\mu q+b)} \quad (238)$$

If we switch back to the original form of the index in the last line we obtain,

$$= \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} t_{(\lambda p_{i-1}+\mu)q+b} \quad (239)$$

which allows us to substitute our previous result for t,

$$v'_{(\lambda p_{i-1}+\mu)q+b} = \sum_{\lambda'} (W_{f_i})_{\lambda\lambda'} w_{p_i}^{\mu\lambda'} v_{(\mu f+\lambda')q+b} \quad (240)$$

This gives us the basic decimation-in-time mixed-radix algorithm for complex data which we will be able to specialize to real data,

for $i = 1 \ldots n_f$ **do**
  for $\mu = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q - 1$ **do**
      for $\lambda = 0 \ldots f - 1$ **do**
        $t_\lambda \Leftarrow \omega_{p_i}^{\mu\lambda'} v_{(\mu f + \lambda')q + b}$
      end for
      for $\lambda = 0 \ldots f - 1$ **do**
        $v'_{(\lambda p_{i-1} + \mu)q + b} = \sum_{\lambda'=0}^{f-1} W_f(\lambda, \lambda') t_{\lambda'}$ {DFT matrix-multiply module}
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

We are now at the point where we can convert an algorithm formulated in terms of complex variables to one in terms of real variables by choosing a suitable storage scheme. We will adopt the FFTPACK storage convention. FFTPACK uses a scheme where individual FFTs are contiguous, not interleaved, and real-imaginary pairs are stored in neighboring locations. This has better locality than was possible for the radix-2 case.

The interleaving of the intermediate FFTs results from the Kronecker product, $W_p \otimes I_q$. The FFTs can be made contiguous if we reorder the Kronecker product on the intermediate passes,

$$W_p \otimes I_q \Rightarrow I_q \otimes W_p \tag{241}$$

This can be implemented by a simple change in indexing. On pass-i we store element v_{aq_i+b} in location v_{bp_i+a}. We compensate for this change by making the same transposition when reading the data. Note that this only affects the indices of the intermediate passes. On the zeroth iteration the transposition has no effect because $p_0 = 1$. Similarly there is no effect on the last iteration, which has $q_{n_f} = 1$. This is how the algorithm above looks after this index transformation,

for $i = 1 \ldots n_f$ **do**
  for $\mu = 0 \ldots p_{i-1} - 1$ **do**
    for $b = 0 \ldots q - 1$ **do**
      for $\lambda = 0 \ldots f - 1$ **do**
        $t_\lambda \Leftarrow \omega_{p_i}^{\mu\lambda'} v_{(\lambda' q + b)p_{i-1} + \mu}$
      end for
      for $\lambda = 0 \ldots f - 1$ **do**
        $v'_{bp + (\lambda p_{i-1} + \mu)} = \sum_{\lambda'=0}^{f-1} W_f(\lambda, \lambda') t_{\lambda'}$ {DFT matrix-multiply module}
      end for
    end for
  end for
  $v \Leftarrow v'$
end for

We transpose the input terms by writing the index in the form $aq_{i-1} + b$, to take account of the pass-dependence of the scheme,

$$v_{(\mu f + \lambda')q + b} = v_{\mu q_{i-1} + \lambda' q + b} \tag{242}$$

We used the identity $q_{i-1} = fq$ to split the index. Note that in this form $\lambda' q + b$ runs from 0 to $q_{i-1} - 1$ as b runs from 0 to $q - 1$ and λ' runs from 0 to $f - 1$. The transposition for the input terms then gives,

$$v_{\mu q_{i-1} + \lambda' q + b} \Rightarrow v_{(\lambda' q + b) p_{i-1} + \mu} \tag{243}$$

In the FFTPACK scheme the intermediate output data have the same half-complex symmetry as the radix-2 example, namely,

$$v^{(i)}_{bp+a} = v^{(i)*}_{bp+(p-a)} \tag{244}$$

and on the input data from the previous pass have the symmetry,

$$v^{(i-1)}_{(\lambda' q + b) p_{i-1} + \mu} = v^{(i-1)*}_{(\lambda' q + b) p_{i-1} + (p_{i-1} - \mu)} \tag{245}$$

Using these symmetries we can halve the storage and computation requirements for each pass. Compared with the radix-2 algorithm we have more freedom because the computation does not have to be done in place. The storage scheme adopted by FFTPACK places elements sequentially with real and imaginary parts in neighboring locations. Imaginary parts which are known to be zero are not stored. Here are the full details of the scheme,

Term	Location		
$g(bp_i)$	real part	bp_i	
	imag part	—	
$g(bp_i + a)$	real part	$bp_i + 2a - 1$	for $a = 1 \ldots p_i/2 - 1$
	imag part	$bp_i + 2a$	
$g(bp_i + p_i/2)$	real part	$bp_i + p_i - 1$	if p_i is even
	imag part	—	

The real element for $a = 0$ is stored in location bp. The real parts for $a = 1 \ldots p/2 - 1$ are stored in locations $bp + 2a - 1$ and the imaginary parts are stored in locations $bp + 2a$. When p is even the term for $a = p/2$ is purely real and we store it in location $bp + p - 1$. The zero imaginary part is not stored.

When we compute the basic iteration,

$$v^{(i)}_{bp+(\lambda p_{i-1} + \mu)} = \sum_{\lambda'} W_f^{\lambda\lambda'} \omega_{p_i}^{\mu\lambda'} v^{(i-1)}_{(\lambda' q + b) p_{i-1} + \mu} \tag{246}$$

we eliminate the redundant conjugate terms with $a > p_i/2$ as we did in the radix-2 algorithm. Whenever we need to store a term with $a > p_i/2$ we consider instead the corresponding conjugate term with $a' = p - a$. Similarly when reading data we replace terms with $\mu > p_{i-1}/2$ with the corresponding conjugate term for $\mu' = p_{i-1} - \mu$.

Since the input data on each stage has half-complex symmetry we only need to consider the range $\mu = 0 \ldots p_{i-1}/2$. We can consider the best ways to implement the basic iteration for each pass, $\mu = 0 \ldots p_{i-1}/2$.

On the first pass where $\mu = 0$ we will be accessing elements which are the zeroth components of the independent transforms $W_{p_{i-1}} \otimes I_{q_{i-1}}$, and are purely real. We can code the pass with $\mu = 0$ as a special case for real input data, and conjugate-complex output. When $\mu = 0$ the

twiddle factors $\omega_{p_i}^{\mu\lambda'}$ are all unity, giving a further saving. We can obtain small-N real-data DFT modules by removing the redundant operations from the complex modules. For example the $N = 3$ module was,

$$t_1 = z_1 + z_2, \quad t_2 = z_0 - t_1/2, \quad t_3 = \sin(\pi/3)(z_1 - z_2), \tag{247}$$

$$x_0 = z_0 + t_1, \quad x_1 = t_2 + it_3, \quad x_2 = t_2 - it_3. \tag{248}$$

In the complex case all the operations were complex, for complex input data z_0, z_1, z_2. In the real case z_0, z_1 and z_2 are all real. Consequently t_1, t_2 and t_3 are also real, and the symmetry $x_1 = t_1 + it_2 = x_2^*$ means that we do not have to compute x_2 once we have computed x_1.

For subsequent passes $\mu = 1 \ldots p_{i-1}/2-1$ the input data is complex and we have to compute full complex DFTs using the same modules as in the complex case. Note that the inputs are all of the form $v_{(\lambda q+b)p_{i-1}+\mu}$ with $\mu < p_{i-1}/2$ so we never need to use the symmetry to access the conjugate elements with $\mu > p_{i-1}/2$.

If p_{i-1} is even then we reach the halfway point $\mu = p_{i-1}/2$, which is another special case. The input data in this case is purely real because $\mu = p_{i-1} - \mu$ for $\mu = p_{i-1}/2$. We can code this as a special case, using real inputs and real-data DFT modules as we did for $\mu = 0$. However, for $\mu = p_{i-1}/2$ the twiddle factors are not unity,

$$\begin{aligned} \omega_{p_i}^{\mu\lambda'} &= \omega_{p_i}^{(p_{i-1}/2)\lambda'} && (249)\\ &= \exp(-i\pi\lambda'/f_i) && (250) \end{aligned}$$

These twiddle factors introduce an additional phase which modifies the symmetry of the outputs. Instead of the conjugate-complex symmetry which applied for $\mu = 0$ there is a shifted conjugate-complex symmetry,

$$t_\lambda = t^*_{f-(\lambda+1)} \tag{251}$$

This is easily proved,

$$\begin{aligned} t_\lambda &= \sum e^{-2\pi i\lambda\lambda'/f_i} e^{-i\pi\lambda'/f_i} r_{\lambda'} && (252)\\ t_{f-(\lambda+1)} &= \sum e^{-2\pi i(f-\lambda-1)\lambda'/f_i} e^{-i\pi\lambda'/f_i} r_{\lambda'} && (253)\\ &= \sum e^{2\pi i\lambda\lambda'/f_i} e^{i\pi\lambda'/f_i} r_{\lambda'} && (254)\\ &= t^*_\lambda && (255) \end{aligned}$$

The symmetry of the output means that we only need to compute half of the output terms, the remaining terms being conjugates or zero imaginary parts. For example, when $f = 4$ the outputs are $(x_0 + iy_0, x_1 + iy_1, x_1 - iy_1, x_0 - iy_0)$. For $f = 5$ the outputs are $(x_0 + iy_0, x_1 + iy_1, x_2, x_1 - iy_1, x_0 - iy_0)$. By combining the twiddle factors and DFT matrix we can make a combined module which applies both at the same time. By starting from the complex DFT modules and bringing in twiddle factors we can derive optimized modules. Here are the modules given by Temperton for $z = W\Omega x$ where x is real and z has the shifted conjugate-complex symmetry. The matrix of twiddle factors, Ω, is given by,

$$\Omega = \mathrm{diag}(1, e^{-i\pi/f}, e^{-2\pi i/f}, \ldots, e^{-i\pi(f-1)/f}) \tag{256}$$

We write z in terms of two real vectors $z = a + ib$. For $N = 2$,

$$a_0 = x_0, \quad b_0 = -x_1. \tag{257}$$

For $N = 3$,

$$t_1 = x_1 - x_2, \tag{258}$$

$$a_0 = x_0 + t_1/2, \quad b_0 = x_0 - t_1, \tag{259}$$

$$a_1 = -\sin(\pi/3)(x_1 + x_2) \tag{260}$$

For $N = 4$,

$$t_1 = (x_1 - x_3)/\sqrt{2}, \quad t_2 = (x_1 + x_3)/\sqrt{2}, \tag{261}$$

$$a_0 = x_0 + t_1, \quad b_0 = -x_2 - t_2, \tag{262}$$

$$a_1 = x_0 - t_1, \quad b_1 = x_2 - t_2. \tag{263}$$

For $N = 5$,

$$t_1 = x_1 - x_4, \quad t_2 = x_1 + x_4, \tag{264}$$

$$t_3 = x_2 - x_3, \quad t_4 = x_2 + x_3, \tag{265}$$

$$t_5 = t_1 - t_3, \quad t_6 = x_0 + t_5/4, \tag{266}$$

$$t_7 = (\sqrt{5}/4)(t_1 + t_3) \tag{267}$$

$$a_0 = t_6 + t_7, \quad b_0 = -\sin(2\pi/10)t_2 - \sin(2\pi/5)t_4, \tag{268}$$

$$a_1 = t_6 - t_7, \quad b_1 = -\sin(2\pi/5)t_2 + \sin(2\pi/10)t_4, \tag{269}$$

$$a_2 = x_0 - t_5 \tag{270}$$

For $N = 6$,

$$t_1 = \sin(\pi/3)(x_5 - x_1), \quad t_2 = \sin(\pi/3)(x_2 + x_4), \tag{271}$$

$$t_3 = x_2 - x_4, \quad t_4 = x_1 + x_5, \tag{272}$$

$$t_5 = x_0 + t_3/2, \quad t_6 = -x_3 - t_4/2, \tag{273}$$

$$a_0 = t_5 - t_1, \quad b_0 = t_6 - t_2, \tag{274}$$

$$a_1 = x_0 - t_3, \quad b_1 = x_3 - t_4, \tag{275}$$

$$a_2 = t_5 + t_1, \quad b_2 = t_6 + t_2 \tag{276}$$

8 Computing the mixed-radix inverse for real data

To compute the inverse of the mixed-radix FFT on real data we step through the algorithm in reverse and invert each operation.

This gives the following algorithm using FFTPACK indexing,

```
for i = n_f ... 1 do
  for μ = 0 ... p_{i-1} − 1 do
    for b = 0 ... q − 1 do
      for λ = 0 ... f − 1 do
        t_{λ'} = Σ_{λ'=0}^{f-1} W_f(λ, λ') v_{bp+(λp_{i-1}+μ)} {DFT matrix-multiply module}
      end for
      for λ = 0 ... f − 1 do
        v'_{(λ'q+b)p_{i-1}+μ} ⇐ ω_{p_i}^{-μλ'} t_λ
      end for
    end for
  end for
  v ⇐ v'
end for
```

When $\mu = 0$ we are applying an inverse DFT to half-complex data, giving a real result. The twiddle factors are all unity. We can code this as a special case, just as we did for the forward routine. We start with complex module and eliminate the redundant terms. In this case it is the final result which has the zero imaginary part, and we eliminate redundant terms by using the half-complex symmetry of the input data.

When $\mu = 1 \ldots p_{i-1}/2 - 1$ we have full complex transforms on complex data, so no simplification is possible.

When $\mu = p_{i-1}/2$ (which occurs only when p_{i-1} is even) we have a combination of twiddle factors and DFTs on data with the shifted half-complex symmetry which give a real result. We implement this as a special module, essentially by inverting the system of equations given for the forward case. We use the modules given by Temperton, appropriately modified for our version of the algorithm. He uses a slightly different convention which differs by factors of two for some terms (consult his paper for details [21]).

For $N = 2$,

$$x_0 = 2a_0, \quad x_1 = -2b_0. \tag{277}$$

For $N = 3$,

$$t_1 = a_0 - a_1, \quad t_2 = \sqrt{3}b_1, \tag{278}$$

$$x_0 = 2a_0 + a_1, \quad x_1 = t_1 - t_2, \quad x_2 = -t_1 - t_2 \tag{279}$$

For $N = 4$,

$$t_1 = \sqrt{2}(b_0 + b_1), \quad t_2 = \sqrt{2}(a_0 - a_1), \tag{280}$$

$$x_0 = 2(a_0 + a_1), \quad x_1 = t_2 - t_1, \tag{281}$$

$$x_2 = 2(b_1 - b_0), \quad x_3 = -(t_2 + t_1). \tag{282}$$

For $N = 5$,

$$t_1 = 2(a_0 + a_1), \quad t_2 = t_1/4 - a_2, \tag{283}$$

$$t_3 = (\sqrt{5}/2)(a_0 - a_1), \tag{284}$$

$$t_4 = 2(\sin(2\pi/10)b_0 + \sin(2\pi/5)b_1), \tag{285}$$

$$t_5 = 2(\sin(2\pi/10)b_0 - \sin(2\pi/5)b_1), \tag{286}$$

$$t_6 = t_3 + t_2, \quad t_7 = t_3 - t_2, \tag{287}$$

$$x_0 = t_1 + a_2, \quad x_1 = t_6 - t_4, \tag{288}$$

$$x_2 = t_7 - t_5, \quad x_3 = -t_7 - t_5, \tag{289}$$

$$x_4 = -t_6 - t_4. \tag{290}$$

9 Conclusions

We have described the basic algorithms for one-dimensional radix-2 and mixed-radix FFTs. It would be nice to have a pedagogical explanation of the split-radix FFT algorithm, which is faster than the simple radix-2 algorithm we used. We could also have a whole chapter on multidimensional FFTs.

Bibliography

[1] P. Duhamel and M. Vetterli. Fast fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.

[2] E. Oran Brigham. *The Fast Fourier Transform*. Prentice Hall, 1974.

[3] C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms*. Wiley, 1984.

[4] Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. *Programs for Digital Signal Processing*. IEEE Press, 1979.

[5] Winthrop W. Smith and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms*. IEEE Press, 1995.

[6] Douglas F. Elliott and K. Ramamohan Rao. *Fast transforms: algorithms, analyses, applications*. Academic Press, 1982. This book does not contain actual code, but covers the more advanced mathematics and number theory needed in the derivation of fast transforms.

[7] Richard E. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, 1984.

[8] James H. McClellan and Charles M. Rader. *Number Theory in Digital Signal Processing*. Prentice-Hall, 1979.

[9] C. S. Burrus. Notes on the FFT. Available from http://www-dsp.rice.edu/res/fft/fftnote.asc.

[10] Henrik V. Sorenson, Michael T. Heideman, and C. Sidney Burrus. On computing the split-radix FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-34(1):152–156, 1986.

[11] Pierre Duhamel. Implementation of split-radix FFT algorithms for complex, real, and real-symmetric data. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-34(2):285–295, 1986.

[12] Clive Temperton. A note on prime factor FFT algorithms. *Journal of Computational Physics*, 58:198–204, 1983.

[13] Clive Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58:283–299, 1985.

[14] Charles M. Rader. Discrete fourier transform when the number of data samples is prime. *IEEE Proceedings*, 56(6):1107–1108, 1968.

[15] Mehalic, Rustan, and Route. Effects of architecture implementation on DFT algorithm performance. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASP-33:684–693, 1985.

[16] P. B. Visscher. The FFT: Fourier transforming one bit at a time. *Computers in Physics*, 10(5):438–443, Sep/Oct 1996.

[17] Jeffrey J. Rodriguez. An improved FFT digit-reversal algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(8):1298–1300, 1989.

[18] Petr Rösel. Timing of some bit reversal algorithms. *Signal Processing*, 18:425–433, 1989.

[19] Clive Temperton. Self-sorting mixed-radix fast fourier transforms. *Journal of Computational Physics*, 52(1):1–23, 1983.

[20] Richard C. Singleton. An algorithm for computing the mixed radix fast fourier transform. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103, June 1969.

[21] Clive Temperton. Fast mixed-radix real fourier transforms. *Journal of Computational Physics*, 52:340–350, 1983.

[22] Henrik V. Sorenson, Douglas L. Jones, Michael T. Heideman, and C. Sidney Burrus. Real-valued fast fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, 1987.