.

Discrete Fourier Transforms for Turbulence

Stewart Cant

CUED/A–THERMO/TR65
September 2012

Introduction

This document is intended as a brief guide to the discrete Fourier transform with emphasis on its application to Direct Numerical Simulation (DNS) and Large Eddy Simulation (LES) of turbulent flow and turbulent combustion. The theory of the discrete Fourier transform is well covered in a number of standard texts and only the briefest outline will be given here. For further information the reader is referred in particular to the comprehensive, practical and very readable account in *Numerical Recipes* [1] and the classic work of Brigham [2]. The use of Fourier transforms in turbulence theory is discussed at length by Batchelor [3], and Fourier pseudo–spectral methods for numerical solution of the Navier–Stokes equations are presented by Canuto et al. [4] where a number of other popular spectral methods are also discussed. Even where Fourier methods are not used as the primary means of solving the governing equations it is often convenient to establish turbulent initial or boundary conditions in terms of Fourier coefficients. A good example is the method given by Orszag and Patterson [5] which is in widespread use in DNS and LES. Similarly, Fourier methods are often used in diagnostic tools for the analysis of results from DNS and LES calculations, particularly when correlation data is required.

Practical use of Fourier transform methods depends on the existence of the Fast Fourier Transform (FFT), for the purposes of DNS and LES it is necessary for the transform to be as fast as possible.

Continuous Fourier Transform

The Fourier transform $F(\overline{k})$ of a function $f(x)$ is defined together with the inverse transform as

$$F(\overline{k}) = \int_{-\infty}^{\infty} f(x) \exp{(2\pi i\overline{k}x)} dx \tag{1}$$

$$f(x) = \int_{-\infty}^{\infty} F(\overline{k}) \exp{(-2\pi i\overline{k}x)} d\overline{k} \tag{2}$$

where $\overline{k}$ is the linear wavenumber. Note that the equivalent angular wavenumber is $\hat{k} = 2\pi\overline{k}$. In general the functions $f(x)$ and $F(\overline{k})$ are complex–valued, and the above definition of the Fourier transform is valid for an infinite domain in both x–space and $\overline{k}$–space provided that the functions $f(x), F(\overline{k})$ satisfy certain mild constraints to ensure convergence of the integrals [2, 6]. For present purposes it is necessary to restrict attention to a domain of finite length L, and to assume that the function $f(x)$ is periodic. Then the transform definitions become

$$F(\overline{k}) = \int_{0}^{L} f(x) \exp{(2\pi i\overline{k}x)} dx \tag{3}$$

$$f(x) = \frac{1}{L} \sum_{k=-\infty}^{\infty} F(\overline{k}) \exp{(-2\pi i\overline{k}x)} \tag{4}$$

in which the linear wavenumber $\overline{k}$ has become a discrete variable given by $\overline{k} = k/L$ where k is an integer. The discrete nature of the wavenumber space follows naturally from the periodicity of the function $f(x)$ on the finite domain of length L, since L must be an exact integer multiple of the wavelength $\lambda = 1/\overline{k}$. Equation (4) is the definition of the classical Fourier series, with coefficients given by equation (3).

It must be noted that there are many different variants of the Fourier transform, all of which share the essential feature of projecting a function of interest onto a set of complex exponential basis functions. There is no inconsistency provided that the forward and inverse transforms are compatible, and that some care is exercised in the physical interpretation of the coefficients $F(\overline{k})$. The present definition embodied in eqns.(1–4) is chosen for simplicity and ease of computational implementation, and there is evidence that this definition is becoming the *de facto* standard. Variations occur mainly in terms of notation and normalisation, and some of the common alternative definitions are given in the Appendix.

Discrete Fourier Transform

The Discrete Fourier Tranform (DFT) is formed by sampling the continuous function $f(x)$ at N equally–spaced points along the finite interval L to yield the N values $f_j, j = 0, \ldots, N-1$. The sampling interval is $\delta x = L/N$, and the spatial location of the j-th sample is given by $x = j\delta x = jL/N$. The definition of the wavenumber $\overline{k}$ is unchanged, but the maximum wavenumber that can be resolved using N samples is now the Nyquist wavenumber $\overline{k}_{\mathrm{max}} = N/2L$, and the maximum value of the wavenumber index k is $N/2$. The Nyquist limit arises for the fundamental reason that, for a sine wave of a given wavenumber, a minimum of two samples is required in order to repesent the amplitude and phase. If the continuous function $f(x)$ contains information at higher wavenumbers than the Nyquist limit then N samples is insufficient to resolve the signal and an *aliasing error* will result. The only cure is to increase the value of N. The value of the Nyquist wavenumber is also consistent with the principle that the Fourier transform cannot generate new information, so that the N complex samples f_j must transform to exactly N complex Fourier coefficients F_k. Discretisation of the integral in (3) and truncation of the Fourier series (4) at the maximum wavenumber yields the discrete Fourier transform pair

$$\begin{aligned} F_k &= \delta x \sum_{j=0}^{N-1} f_j \exp\left(2\pi i jk/N\right) && (5) \\ f_j &= \frac{1}{N\delta x} \sum_{k=-N/2}^{N/2} F_k \exp\left(-2\pi i jk/N\right) && (6) \end{aligned}$$

Note that the sum in the discrete inverse transform (6) is written to run from $k = -N/2$ to $k = N/2$. This reflects the existence of the Nyquist limit, emphasises the symmetry properties of the transform and is computationally convenient. For even values of N the range $[-N/2 \ldots N/2]$ appears to correspond to a total of $N + 1$ rather than N transform elements, but in fact the value $F_{-N/2}$ is identically equal to the value $F_{N/2}$ due to the periodicity of the discrete transform F_k. For the case of N odd the true limits of k are $\pm(N-1)/2$, which automatically yields the correct total number N of transform elements.

Fast Fourier Transform

The computational cost of the DFT is rather high, since it requires at least N multiplications and additions to evaluate each element of the transform, and there are N transform values to compute. Thus the DFT algorithm requires $O(N^2)$ arithmetic operations, and for large datasets the computational cost is prohibitive. Fortunately there is another approach, based on decomposing the sum in the transform expression (5) or (6) into two sums each of half the length. The forward transform becomes

$$\begin{aligned} F_k &= \delta x \sum_{j=0}^{N-1} f_j \exp\left(2\pi i jk/N\right) \\ &= \delta x \sum_{j=0}^{N/2-1} f_{2j} \exp\left(2\pi i (2j)k/N\right) + \delta x \sum_{j=0}^{N/2-1} f_{2j+1} \exp\left(2\pi i (2j+1)k/N\right) \\ &= F_k^0 + \exp\left(2\pi i k/N\right) F_k^1 && (7) \end{aligned}$$

where it is clear that F_k^0 and F_k^1 $(k = 0 \ldots N/2-1)$ are respectively the DFTs of the even- and odd–numbered elements of the original dataset f_j. The decomposition (7) is called the Danielson–Lanczos formula, and it may repeated recursively on each of F_k^0 and F_k^1 to produce four reduced transforms F_k^{00}, F_k^{01}, F_k^{10} and F_k^{11} $(k = 0 \ldots N/4-1)$, and so on. The process results in a heirarchy of transforms each half the length of its predecessor until all that remains is a transform of length 1, which is simply an identity operation, i.e. $F_1^b = f_p$, where b is a string of binary digits, e.g. $b = 101 \ldots 010$, and f_p denotes a single element of the dataset f_j. It is not immediately obvious which one–point transform corresponds to which element of f_j, and this is determined by noting that the application of the Danielson–Lanczos formula at each level of the heirarchy acts to select the even (=0) and odd (=1) elements from the current reduced dataset. Each selection amounts to a test of one bit of the index j expressed as a binary number, starting from the least significant bit and moving one bit higher at each level of the heirarchy until all bits have been tested and the one–point transform level has been reached. Then the binary string b that serves to identify each one–point transform also provides a record of the even–odd selections that were taken to arrive at that point, and is nothing less than the binary representation of the single index p, expressed in bit–reversed order. The FFT algorithm due to Cooley and Tukey [7] uses this approach to rearrange the original dataset into bit–reversed index order, whereupon the Danielson–Lanczos formula is applied recursively to build up the required N-point transform. The inverse transform may be computed in precisely the same manner, with the factor $\exp(2\pi ik/N)$ replaced by $\exp(-2\pi ik/N)$. Each application of the Danielson–Lanczos formula requires $O(N)$ operations, and the number of levels in the heirarchy is equal to the length of the bit–string b which by definition is $\log_2 N$. Thus the overall cost of the basic Fast Fourier Transform (FFT) algorithm is $O(N \log_2 N)$ operations, which represents a very significant saving over the DFT.

A drawback of the basic FFT algorithm is that it is applicable only for a transform of length $N = 2^m$ where m is an integer. This is unimportant for many applications, but in other cases it is necessary to work with transforms whose length lies inconveniently between neighbouring powers of two. Variants of the FFT exist for transform lengths that are multiples of powers of small prime factors (e.g. 2, 3, 5, 7 etc.) and these can be made highly efficient. An FFT algorithm for arbitrary transform lengths has been given by Bluestein [10] and retains the $O(N \log_2 N)$ operation count.

Multidimensional Fourier Transforms

Extension of the standard one–dimensional Fourier transform to two, three or higher dimensions is straightforward. Defining f as a scalar function of a position vector $\mathbf{x}$, and F as a scalar function of a wavenumber vector $\overline{\mathbf{k}}$, the continuous Fourier transform relations between them on an infinite domain are

$$F(\overline{\mathbf{k}}) = \int_{-\infty}^{\infty} f(\mathbf{x}) \exp(2\pi i\overline{\mathbf{k}}.\mathbf{x}) d\mathbf{x} \tag{8}$$

$$f(\mathbf{x}) = \int_{-\infty}^{\infty} F(\overline{\mathbf{k}}) \exp(-2\pi i\overline{\mathbf{k}}.\mathbf{x}) d\overline{\mathbf{k}} \tag{9}$$

where $\overline{\mathbf{k}}.\mathbf{x}$ is the scalar product of the position and wavenumber vectors, and the integrals are taken over all directions. Clearly this definition allows for Fourier transforms of any dimensionality, corresponding to the dimensionality of the vectors $\mathbf{x}$ and $\overline{\mathbf{k}}$. Restricting attention to a Cartesian vector space of dimension M and finite size L_m in the m-th direction, the transform relations become

$$F(\overline{\mathbf{k}}) = \int_0^{L_1} \cdots \int_0^{L_M} f(\mathbf{x}) \exp(2\pi i\overline{\mathbf{k}}.\mathbf{x}) dx_1 \dots dx_M \tag{10}$$

$$f(\mathbf{x}) = \frac{1}{L_1} \cdots \frac{1}{L_M} \sum_{k_1=-\infty}^{\infty} \cdots \sum_{k_M=-\infty}^{\infty} F(\overline{\mathbf{k}}) \exp(-2\pi i\overline{\mathbf{k}}.\mathbf{x}) \tag{11}$$

and the components of the wavenumber vector are given by $\overline{k}_m = k/L_m$ where k is an integer. In a Cartesian or other orthogonal coordinate system the multidimensional Fourier transform may be decomposed easily and treated as a succession of one–dimensional transforms:

$$\begin{aligned} F(\overline{k}_1 \dots \overline{k}_M) = & \int_0^{L_1} dx_1 \exp(2\pi i k_1 x_1) \dots \\ & \int_0^{L_M} dx_M \exp(2\pi i k_M x_M) f(x_1 \dots x_M) \end{aligned} \tag{12}$$

$$
\begin{aligned}
f(x_1 \ldots x_M) &= \frac{1}{L_1} \sum_{k_1=-\infty}^{\infty} \exp(2\pi i k_1 x_1) \ldots \\
&\quad \frac{1}{L_M} \sum_{k_M=-\infty}^{\infty} \exp(2\pi i k_M x_M) F(\overline{k}_1 \ldots \overline{k}_M) \qquad (13)
\end{aligned}
$$

where it is implicit that there exists a succession of partially–transformed functions f and F, and it is clear that the order in which the one–dimensional transforms are carried out is irrelevant.

The multidimensional discrete Fourier transform relations are defined for a space of M dimensions $m = 1 \ldots M$ each containing N_m points. The sampling interval in each direction is $\delta x_m = L_m/N_m$ and in the general case all directions may have different sizes and numbers of points. The relations are

$$
\begin{aligned}
F_{k_1 \ldots k_M} &= \delta x_1 \sum_{j_1=0}^{N_1-1} \exp(2\pi i j_1 k_1/N_1) \ldots \\
&\quad \delta x_M \sum_{j_M=0}^{N_M-1} \exp(2\pi i j_M k_M/N_M) f_{j_1 \ldots j_M} \qquad (14)
\end{aligned}
$$

$$
\begin{aligned}
f_{j_1 \ldots j_M} &= \frac{1}{N_1 \delta x_1} \sum_{k_1=-N_1/2}^{N_1/2} \exp(-2\pi i j_1 k_1/N_1) \ldots \\
&\quad \frac{1}{N_M \delta x_M} \sum_{k_M=-N_M/2}^{N_M/2} \exp(-2\pi i j_M k_M/N_M) F_{k_1 \ldots k_M} \qquad (15)
\end{aligned}
$$

and once again it is clear that each dimension may be treated separately.

Generalised Prime Factor FFT Algorithm

An FFT algorithm suitable for dataset lengths $N = 2^a 3^b 5^c$ where a,b and c are non–negative integers has been presented by Temperton [8, 9]. The algorithm is self–sorting, requires minimal workspace, and is among the most computationally efficient FFTs yet developed. Generalisation to powers of larger prime numbers is straightforward, although the computational cost then begins to increase significantly. The algorithm is derived by considering the discrete Fourier transform of length N written as

$$
F_k = \sum_{j=0}^{N-1} f_j \omega_N^{jk} \qquad (16)
$$

where $0 \leq k \leq N-1$ and $\omega_N = \exp(2\pi i/N)$. Note that the inverse transform may be treated in the same manner.

The DFT may be expressed in matrix form by defining the transform matrix $W_N(j,k) = \omega_N^{jk}$, whereupon (16) may be rewritten as

$$
\underline{F} = W_N \underline{f}
$$

and many different FFT algorithms may be obtained by appropriate factorisation of the symmetric non–sparse matrix W_N into products of sparse matrices. The Temperton algorithm is based on the decimation–in–frequency approach. Take $N = N_1 N_2$, and denote the corresponding reduced–order DFT matrices as W_{N_1} and W_{N_2}. A permutation matrix of order N is defined as

$$
\begin{aligned}
P_{N_2}^{N_1}(j,k) &= 1 \text{ if } j = rN_1 + s \text{ and } k = sN_2 + r \\
&= 0 \text{ otherwise} \qquad (17)
\end{aligned}
$$

for integer values r and s, and a diagonal matrix of order N containing the "twiddle factors" is defined as

$$\begin{aligned} D_{N_2}^{N_1}(j,k) &= \omega_N^{sm} \text{ if } j = k = sN_2 + m \\ &= 0 \text{ otherwise.} \end{aligned} \tag{18}$$

Then, with identity matrices I_{N_1} and I_{N_2} of order N_1 and N_2 respectively, the order N DFT matrix may be factorised as

$$W_N = (W_{N_2} \times I_{N_1})\, P_{N_2}^{N_1} D_{N_2}^{N_1} (W_{N_1} \times I_{N_2})$$

where × denotes the Kronecker outer product. Using the matrix identity

$$(W_{N_2} \times I_{N_1})\, P_{N_2}^{N_1} = P_{N_2}^{N_1} (I_{N_1} \times W_{N_2})$$

the factorisation of W_N becomes

$$W_N = P_{N_2}^{N_1} (I_{N_1} \times W_{N_2})\, D_{N_2}^{N_1} (W_{N_1} \times I_{N_2})$$

Thus the original DFT has been decomposed into a sequence of operations consisting of N_2 DFTs of length N_1, multiplication by a set of twiddle factors, N_1 DFTs of length N_2, and finally a permutation. For the general case of $N = N_1 N_2 \ldots N_{n-1} N_n$ the factorisation becomes

$$W_N = P_1 P_2 \ldots P_{n-1} P_n T_n T_{n-1} \ldots T_2 T_1 \tag{19}$$

in which the permutation matrices are given for stage $i, 1 \le i \le n$ of the transform by

$$P_i = \left(I_{l_i} \times P_{m_i}^{N_i}\right),$$

and the corresponding operator matrices are

$$T_i = I_{l_i} \times \left[D_{m_i}^{N_i} (W_{N_i} \times I_{m_i})\right]$$

where $l_{i+1} = N_i l_i$ with $l_1 = 1$, and $m_i = N/l_{i+1}$. Note that no assumption has been made so far that restricts the values of the factors N_1 and N_2, and so the factorisation (19) remains completely general.

The usefulness of the approach becomes clear if N is a power of a small prime factor, e.g. the radix–2 case $N = 2^n$. Then $N_i = 2$ for all i, and the operator matrices become

$$\begin{aligned} T_1 &= D_{N/2}^2 \left(W_2 \times I_{N/2}\right) \\ T_2 &= I_2 \times \left[D_{N/4}^2 \left(W_2 \times I_{N/4}\right)\right] \\ &\cdots \\ T_{n-1} &= I_{N/4} \times \left[D_2^2 \left(W_2 \times I_2\right)\right] \\ T_n &= I_{N/2} \times W_2 \end{aligned}$$

for the n stages of the transform. Interpretation of the operator T_i at each stage is quite straightforward. The first stage operator T_1 applies a DFT of length 2 (as expressed by the matrix W_2) a total of $N/2$ times, with multiplication of each result by a twiddle factor from the list of N values, of which $N/2$ are distinct, specified in the diagonal matrix $D_{N/2}^2$. The second stage operator T_2 applies $N/4$ DFTs of length 2, with multiplication of each result by a twiddle factor from the list of $N/2$ values ($N/4$ distinct) in the matrix $D_{N/4}^2$, and carries out the entire operation twice over. At each subsequent stage the number of DFTs in the first step is halved along with the number of distinct twiddle factors, and the entire operation is repeated twice as many times. At the final stage expressed by T_n the length 2 DFT is repeated $N/2$ times without twiddle factors. It is clear that the largest DFT matrix that appears at any stage is W_2, and the advantage of the method is that a DFT of length 2 involves only a small number of additions and multiplications and hence is inherently fast. Then the transform of length N is built up by repeating the length–2 transform a total of $N/2$ times, with twiddle factors, at each of the $n = \log_2 N$ stages so that the total operation count is proportional to $N \log_2 N$.

In the algorithm as described the results emerge from the sequence of transform stages in scrambled order, and the process of unscrambling is carried out by the sequence of permutations $P_1 P_2 \ldots P_n$ performed at the end. The unscrambling may be treated as a single operation which in a radix–2 FFT corresponds to bit–reversal, as in the Cooley–Tukey algorithm. Note that the Cooley–Tukey algorithm uses decimation–in–time and hence the bit–reversal operation takes place at the beginning of the process.

In order to produce a self–sorting transform a permutation matrix Q^i_j may be defined for each single bit–reversal operation according to

$$\underline{x}' = Q^i_j \underline{x}$$

whereby the vector elements $x_{\hat{k}}$ and x_k are interchanged, and where the index $\hat{k}$ is obtained from k by interchanging bits i and j of its binary representation. Then the complete bit–reversal operation is defined as

$$P_1 P_2 \ldots P_{n-1} P_n = Q^0_{n-1} Q^1_{n-2} \ldots Q^{n/2-2}_{n/2+1} Q^{n/2-1}_{n/2}.$$

Substituting in (19) and using the distributive properties of the Kronecker product it may be shown [8] that the factorisation of the DFT becomes

$$W_N = \left(Q^0_{n-1} T_n\right) \left(Q^1_{n-2} T_{n-1}\right) \ldots \left(Q^{n/2-1}_{n/2} T_{n/2+1}\right) T_{n/2} \ldots T_2 T_1.$$

Thus the bit–reversal operation is spread over the second half of the sequence of n transform stages, noting that for odd n the middle stage is left untouched. In practice the self–sorting transform involves pairing up the length–2 DFTs that are affected by the bit–reversal at each stage. These are easily identified since they also turn out to have the same twiddle factor. If the output values from the first element of the pair are denoted by $(F^{(1)}_1, F^{(1)}_2)$ and those of the second element by $(F^{(2)}_1, F^{(2)}_2)$, then the interchange of the output values corresponds to placing them in a 2× 2 matrix and taking the transpose according to

$$\begin{bmatrix} F^{(1)}_1 & F^{(1)}_2 \\ F^{(2)}_1 & F^{(2)}_2 \end{bmatrix} \rightarrow \begin{bmatrix} F^{(1)}_1 & F^{(2)}_1 \\ F^{(1)}_2 & F^{(2)}_2 \end{bmatrix}.$$

Implementation of the radix–2 prime factor algorithm follows the factorisation (19) quite literally. Computational efficiency can be maximised by precomputing a table of twiddle factors and by taking together at each stage all length–2 DFTs having the same twiddle factor. This simplifies the indexing of the input data and allows the self–sorting procedure to take place quite naturally.

Extension to the case of $N = 3^n$ is immediate, and the largest DFT matrix appearing in the factorisation (19) then becomes W_3. A DFT of length 3 is somewhat more complicated than that for length 2 but remains inherently fast, and the complete transform is constructed as before, by repeating the length 3 transform $N/3$ times, with twiddle factors, at each stage. The unscrambling process makes use of “trit–reversal”, which is simply the base–3 analogue of bit–reversal, and the self–sorting procedure involves coupling the length–3 transforms in groups of three, with transposition of a 3× 3 matrix. Values of $N = p^n$ for $p > 3$ may be handled in the same manner, and indeed p is not explicitly constrained to be a prime number. Nevertheless it should be noted that the computational cost of the necessary length–p DFT rises strongly with the value of p, and this motivates the reduction of N to a power of the smallest possible (i.e. prime) factor.

The mixed–radix case is derived by taking $N = N_1 N_2$ where N_1 and N_2 are now mutually prime. Then it is possible to factorise the transform using a “Ruritanian map”. Each index j and k is to be associated with a pair of indices (j_1, j_2) and (k_1, k_2) respectively, such that $0 \le j_1, k_1 < N_1$ and $0 \le j_2, k_2 < N_2$. Defining four further integers p, q, r, s such that

$$pN_2 = rN_1 + 1; \qquad qN_1 = sN_2 + 1$$

where $0 < p, s < N_1$ and $0 < q, r < N_2$ the Ruritanian map is given by

$$\begin{aligned} j_1 &= (pj) \bmod N_1; \qquad & j_2 &= (qj) \bmod N_2 \\ k_1 &= (pk) \bmod N_1; \qquad & k_2 &= (qk) \bmod N_2 \end{aligned}$$

with inverse

$$\begin{aligned} j &= (j_1 N_2 + j_2 N_1) \bmod N; \\ k &= (k_1 N_2 + k_2 N_1) \bmod N. \end{aligned} \tag{20}$$

The Ruritanian map has a simple representation in tabular form. An example for the case of $N = 12$, i.e. $N_1 = 4$ and $N_2 = 3$, is shown in Table 1. For each value of j_1 down the table the values of j_2 increase (modulo N) across the table in steps of N/N_2, and vice versa. This observation allows the Ruritanian map to be implemented quite easily through suitable indexing of the transform elements.

	0	1	2	j_2
0	0	4	8	
1	3	7	11	
2	6	10	2	
3	9	1	5	
j_1				

Substituting the inverse map (20) into the DFT definition (16) yields

$$F(k_1, k_2) = \sum_{k_2=0}^{N_2-1} \left[\sum_{k_1=0}^{N_1-1} f(j_1, j_2) \omega_{N_1}^{N_2 j_1 k_1} \right] \omega_{N_2}^{N_1 j_2 j_2} \tag{21}$$

and the original one–dimensional DFT of size N has been factorised into a two–dimensional DFT of size N_1 by N_2. This may be handled in the same manner as any other two–dimensional transform. Clearly, if either of N_1 and N_2 is a power of a small prime number then the prime factor algorithm may be used to compute the corresponding set of short one–dimensional transforms. The only complication arises from the appearance of the factors N_1 and N_2 in the exponents of ω in each of the short transforms. This may be interpreted as a *rotation* of each transform. If a DFT of length N_i is modified such that ω_{N_i} is replaced by $\omega_{N_i}^r$, where r is an integer, then the transform is rotated such that the ordering of the output data is changed from $[0, 1, 2, \ldots, N_i - 1]$ to $[0, r \bmod N_i, 2r \bmod N_i, \ldots, (N_i - 1)r \bmod N_i]$, while the actual output values remain unaffected.

The result (21) for $N = N_1 N_2$ can be generalised quite easily to the case $N = N_1 N_2 \ldots N_n$ where all the N_i are mutually prime, and converts the original one–dimensional transform to a n–dimensional transform with rotations:

$$\begin{aligned} &F(k_1, k_2, \ldots, k_n) = \\ &\sum_{k_n=0}^{N_n-1} \cdots \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} f(j_1, j_2, \ldots, j_n) \omega_{N_1}^{j_1 k_1 N/N_1} \omega_{N_2}^{j_2 j_2 N/N_2} \ldots \omega_{N_n}^{j_n j_n N/N_n}. \end{aligned} \tag{22}$$

The rotated transform may be repesented by the matrix $W_N^{[r]}(j, k) = \omega_N^{rjk}$, which is simply the matrix W_N with each element raised to the power r. In this notation the factorisation (22) may be written as

$$W_N = R^{-1} \left(W_{N_n}^{[N/N_n]} \times \ldots \times W_{N_2}^{[N/N_2]} \times W_{N_1}^{[N/N_1]} \right) R$$

where R is the permutation matrix corresponding to the Ruritanian map, and the operator × denotes a Kronecker product. Note that the rotation N/N_i for each short transform operation is evaluated modulo N_i. The Ruritanian map can also be generalised to the case of $N = N_1 N_2 \ldots N_n$ simply by extending the tabular representation to n dimensions, with increments of N/N_i in the i–th dimension.

Implementation of the general case $N = N_1 N_2 \ldots N_n$, where each N_i is a power of a different small prime number, requires only a slight modification to the individual radix–i prime factor transforms in order to accommodate the necessary rotation r_i. This is accomplished by raising all twiddle factors to the power r_i, and by making use of the indexing logic to reorder the input values to each transform.

Interpolation using Fourier Transform

Once the discrete Fourier transform F_k of a sampled function f_j has been obtained, the transform may be used to evaluate the original function in a straightforward manner at points that do not necessarily coincide with the sample points. If the sample points are denoted by $x = j\delta x$, where δx is the sampling interval, then any other point may be described as $x + \Delta x = (j + \Delta j)\delta x$, where Δx is the distance from x and Δj is the corresponding number of sampling intervals. Normally $\Delta x < \delta x$ and $0 < \Delta j < 1$, but this is not a necessary restriction. The interpolation formula is a simple extension of the discrete inverse Fourier transform (6):

$$\begin{aligned} f_{j+\Delta j} &= \frac{1}{N\delta x} \sum_{k=-N/2}^{N/2} F_k \exp\left(-2\pi i k[j+\Delta j]/N\right) \\ &= \frac{1}{N\delta x} \sum_{k=-N/2}^{N/2} \left[F_k \exp\left(-2\pi i k \Delta j/N\right)\right] \exp\left(-2\pi i j k/N\right) \qquad (23) \end{aligned}$$

Thus the procedure for interpolation is to multiply each element of the transform $F(k)$ by the factor $\exp\left(-2\pi i k\Delta j/N\right)$ and then take the inverse transform. Clearly this procedure is applicable only for a fixed Δx, i.e. a fixed offset from each sample point. A common requirement is for midpoint interpolation, where $\Delta x = \delta x/2$ and so $\Delta j = 1/2$.

Convolution using Fourier Transform

The convolution of two functions $f(x)$ and $g(x)$ is given by the integral

$$c^{\text{conv}}[f,g](x) = \int_{-\infty}^{\infty} f(x-x')g(x')dx' \qquad (24)$$

In general the functions f and g are complex–valued, but in practice they are often purely real. Note that it is easy to show that $c^{\text{conv}}[f,g](x) = c^{\text{conv}}[g,f](x)$. The *convolution theorem* states that the Fourier transform $C^{\text{conv}}[f,g](\bar{k})$ of $c^{\text{conv}}[f,g](x)$ is given by

$$C^{\text{conv}}[f,g](\bar{k}) = F(\bar{k})G(\bar{k}) \qquad (25)$$

where $F(\bar{k})$ and $G(\bar{k})$ are the Fourier transforms of $f(x)$ and $g(x)$ respectively. Thus a convolution in physical space transforms to a simple product in wavenumber space (and vice versa).

In discrete form the convolution integral becomes the sum

$$c_j^{\text{conv}}[f,g] = \delta x \sum_{m=0}^{N-1} f_{j-m} g_m \qquad (26)$$

where $j = 0 \dots N-1$ and the functions f and g are each assumed to be periodic on the interval $L = N\delta x$. The *discrete convolution theorem* states that the discrete Fourier transform $C_k^{\text{conv}}[f,g]$ of $c_j^{\text{conv}}[f,g]$ is given by

$$C_k^{\text{conv}}[f,g] = F_k G_k \qquad (27)$$

where F_k and G_k are the discrete Fourier transforms of f_j and g_j respectively.

Evaluation of the discrete convolution (19) requires $O(N)$ multiplications and additions for each value of j, and the total is therefore $O(N^2)$. This operation count may be greatly reduced by using the FFT to evaluate F_k and G_k, carrying out the complex multiplication in wavenumber space, and using the FFT once again to evaluate the inverse transform. This procedure is strictly valid only if the discrete functions f_j and g_j are periodic on an interval of length N, where N is a suitable number (e.g. a power of two) for efficient use of the FFT.

For periodic functions it is possible to rewrite the discrete convolution (19) as

$$c_j^{\mathrm{conv}}[f,g] = \delta x \sum_{m=-N/2+1}^{N/2} f_{j-m} g_m \tag{28}$$

where now $j = -N/2+1 \dots N/2$. Periodicity ensures that the values of c_j in (21) for j negative are precisely the same as those for $N/2+1 \leq j \leq N-1$ in the previous expression (19). For suitable N the FFT may be applied exactly as before, with the interpretation that g_j and c_j are now defined for $-N/2+1 \leq j \leq N/2$, while f_j remains unaffected on $0 \leq j \leq N-1$.

In practice the discrete functions of interest often do not satisfy the requirements of periodicity on a suitable length N, and in order to allow the use of the FFT some zero–padding is required. The discrete convolution is used in the form (21), and the function f_j may be padded with zeros at one end from its natural length $N_f - 1$ up to $N-1$. The function g_j must be padded at both ends from its natural range $-N_g^- \dots N_g^+$ out to $-N/2+1 \dots N/2$. Care must be taken to avoid contamination of the true convolution through end effects, and a minimum number of zeros is required equal to the maximum positive or negative non–zero extent of g_j, whichever is greater. The convolution c_j evaluated using this procedure is correct over the range $0 \leq j \leq N_f - 1$.

Correlation using Fourier Transform

The correlation of two functions $f(x)$ and $g(x)$ is given by the integral

$$c^{\mathrm{corr}}[f,g](x) = \int_{-\infty}^{\infty} f(x+x')g(x')dx' \tag{29}$$

Again in general the functions f and g are complex–valued in principle, and the symmetry condition for correlation is $c^{\mathrm{corr}}[f,g](x) = c^{\mathrm{corr}}[g,f](-x)$. The *correlation theorem* states that the Fourier transform $C^{\mathrm{corr}}[f,g](\bar{k})$ of $c^{\mathrm{corr}}[f,g](x)$ is given by

$$C^{\mathrm{corr}}[f,g](\bar{k}) = F(\bar{k})G(-\bar{k}) \tag{30}$$

where $F(\bar{k})$ and $G(\bar{k})$ are the Fourier transforms of $f(x)$ and $g(x)$ respectively.

The discrete correlation is given by

$$c_j^{\mathrm{corr}}[f,g] = \delta x \sum_{m=0}^{N-1} f_{j+m} g_m \tag{31}$$

and the *discrete correlation theorem* states that the discrete Fourier transform $C_k^{\mathrm{corr}}[f,g]$ of $c_j^{\mathrm{corr}}[f,g]$ is given by

$$C_k^{\mathrm{corr}}[f,g] = F_k G_{-k} \tag{32}$$

where F_k and G_k are the discrete Fourier transforms of f_j and g_j respectively.

The operation count for the evaluation of the discrete correlation (24) is $O(N^2)$, and the FFT may be used as above to speed it up. Again, the use of the FFT is strictly applicable only for functions f and g that are periodic on a suitable length N. Zero–padding may be used as necessary, subject to the same considerations as for the discrete convolution.

Fast Fourier Transform for Arbitrary Dataset Size

A fast Fourier transform algorithm for an arbitrary dataset size N has been given by Bluestein [10], and is sometimes called the chirp–z algorithm. The forward transform is written as

$$F_k = \delta x \sum_{j=0}^{N-1} f_j W^{jk}, \; k = -N/2, \dots, N/2 \tag{33}$$

where $W = \exp(2\pi i/N)$, and is then expanded as

$$\begin{aligned} F_k &= \delta x \sum_{j=0}^{N-1} f_j W^{jk+(j^2-j^2+k^2-k^2)/2} \\ &= \delta x\ W^{k^2/2} \sum_{j=0}^{N-1} \left(f_j W^{j^2/2}\right) W^{-(k-j)^2/2} \\ &= \delta x\ W^{k^2/2} \sum_{j=0}^{N-1} g_j h_{k-j} \qquad (34) \end{aligned}$$

where $g_j = f_j W^{j^2/2}$ and $h_{k-j} = W^{-(k-j)^2/2}$. Equation (27) is in the form of a discrete convolution between the functions g and h. Note that the limits on k in (26) are written to make use of the same shorthand as the DFT in (5) and (6), and that the transform F_k consists of N complex elements. Thus the convolution must produce N uncontaminated values, and in order to do so the discrete function h_{k-j} must be of length $2N-1$ at least. The convolution may be evaluated using the FFT algorithm as described above. The dataset g_j must be evaluated for $j = 0, \ldots, N-1$ and zero–padded, first up to $2N-1$ and then up to the next size suitable for efficient use of the FFT. It is convenient to take k to run from 0 to $N-1$ so that the dataset h_m may be evaluated for the range $m = -N+1, \ldots, N-1$, and then zero–padded (at both ends) up to the required size for the FFT. With this choice of indexing the factors $W^{j^2/2}$ and $W^{k^2/2}$ are identical and also equal to the positive half of h_m. The augmented datasets containing g and h are each transformed using the FFT and the product of the transforms is obtained by direct multiplication in the transform space. Inverse transformation then yields the raw convolution which must be multiplied by $W^{k^2/2}$ to produce $F_k, k = 0, \ldots, N-1$. Periodicity ensures that the elements $k = N/2+1, \ldots, N-1$ of F_k are identically equal to those for $k = -N/2+1, \ldots, -1$.

The inverse transform may be treated similarly. Writing

$$f_j = \frac{1}{N\delta x} \sum_{k=-N/2}^{N/2} F_k \bar{W}^{jk},\ j = 0, \ldots, N-1 \qquad (35)$$

where $\bar{W} = \exp(-2\pi i/N)$, the expansion becomes

$$\begin{aligned} f_j &= \frac{1}{N\delta x} \sum_{k=-N/2}^{N/2} F_k \bar{W}^{jk+(j^2-j^2+k^2-k^2)/2} \\ &= \frac{1}{N\delta x} \bar{W}^{j^2/2} \sum_{k=-N/2}^{N/2} \left(F_k \bar{W}^{k^2/2}\right) \bar{W}^{-(j-k)^2/2} \\ &= \frac{1}{N\delta x} \bar{W}^{j^2/2} \sum_{k=-N/2}^{N/2} G_k H_{j-k} \qquad (36) \end{aligned}$$

where $G_k = F_k \bar{W}^{k^2/2}$ and $H_{j-k} = \bar{W}^{-(j-k)^2/2}$. Again, the result is in the form of a discrete convolution, and the evaluation proceeds in precisely the same manner as for the forward transform.

Fourier Transform of Two Real Functions

For real–valued datasets a worthwhile saving in computational cost may be achieved by computing two transforms simultaneously. Two real–valued datasets f_j and g_j each containing N real numbers may be combined to form a single complex–valued dataset $h_j = f_j + ig_j$, containing N complex numbers. The

DFT of h_j is H_k, which also contains N complex numbers and may be expressed as $H_k = F_k + iG_k$, where F_k and G_k are the DFTs of f_j and g_j respectively. It is then necessary to extract the individual transforms F_k and G_k from the combined transform H_k. In general F_k and G_k each contain N complex values, but since f_j and g_j are real the symmetry conditions are

$$\begin{aligned} F_k &= F^*_{N-k} \\ G_k &= G^*_{N-k} \end{aligned} \tag{37}$$

Thus $H^*_{N-k} = F^*_{N-k} - iG^*_{N-k} = F_k - iG_k$, and the real and imaginary parts of the separate transforms F_k and G_k may be expressed as

$$\begin{aligned} F^R_k + iF^I_k &= \frac{1}{2}\left(H^R_{N-k} + H^R_k\right) - \frac{1}{2}i\left(H^I_{N-k} - H^I_k\right) \\ G^R_k + iG^I_k &= \frac{1}{2}\left(H^I_{N-k} + H^I_k\right) + \frac{1}{2}i\left(H^R_{N-k} - H^R_k\right) \end{aligned} \tag{38}$$

Note that by symmetry the transform elements F_0 and $F_{N/2}$, and also the elements G_0 and $G_{N/2}$, are real and independent. Symmetry also implies that only half of each transform need be stored, since the other half may be generated as required using (30).

Evaluation of the inverse transforms is straightforward. The two sets of (complex) transform values F_k and G_k may be combined to form the single complex dataset $H_k = F_k + iG_k$, whereupon an inverse DFT at once yields $h_j = f_j + ig_j$.

Fourier Transform of Single Real Function

Another way to treat a real dataset f_j, $j = 0 \ldots N-1$, is to pack it into a complex dataset of half the length. Taking even and odd elements as the real and imaginary parts respectively produces the complex dataset $h_j = f_{2j} + if_{2j+1}$, $j = 0 \ldots N/2+1$. The DFT is $H_k = F^e_k + iF^o_k$, $k = 0 \ldots N/2+1$, where F^e_k (F^o_k) is the DFT of the even (odd) elements of the original dataset f_j. Since the even and odd elements of f_j each constitute a real dataset, the process of extracting F^e_k and F^o_k from H_k is precisely the same as that described above for two unrelated real datasets:

$$\begin{aligned} F^{eR}_k + iF^{eI}_k &= \frac{1}{2}\left(H^R_{N-k} + H^R_k\right) - \frac{1}{2}i\left(H^I_{N-k} - H^I_k\right) \\ F^{oR}_k + iF^{oI}_k &= \frac{1}{2}\left(H^I_{N-k} + H^I_k\right) + \frac{1}{2}i\left(H^R_{N-k} - H^R_k\right) \end{aligned} \tag{39}$$

Note that by symmetry the transform elements F^e_0 and $F^e_{N/2}$, and also the elements F^o_0 and $F^o_{N/2}$, are real and independent. Finally, according to the Danielson–Lanczos formula the transform F_k of the full original dataset f_j may be recovered using

$$F_k = F^e_k + \exp\left(2\pi ik/N\right)F^o_k \tag{40}$$

for $k = 0 \ldots N-1$. By symmetry, F_0 and $F_{N/2}$ are real and independent, and again by symmetry it is necessary to store only half of the transform, since $F^*_{N-k} = F_k$.

The inverse transform proceeds by evaluating the even and odd transforms according to

$$\begin{aligned} F^e_k &= \frac{1}{2}\left(F_k + F^*_{N/2-k}\right) \\ F^o_k &= \frac{1}{2}\left(F_k - F^*_{N/2-k}\right)\exp\left(-2\pi ik/N\right) \end{aligned} \tag{41}$$

and then forming the combined transform $H_k = F^e_k + iF^o_k$. The inverse DFT at once yields h_j and hence f_j.

Fourier Transforms in Parallel

The use of parallel computers for DNS and LES is becoming widespread, and there is a requirement for Fourier transform methods that are compatible with current parallel architectures. Fourier transformation is by nature an integral operation, and the DFT brings in every element of the transform dataset with equal weight into a single global summation. The heirarchical structure of the FFT algorithm is based from the outset on a global approach and again requires equal access to every element of the dataset. On shared–memory parallel computers there is no difficulty, since each processor can address all areas of memory, subject only to minor penalties in access time. On distributed–memory architectures a large dataset must be split between the memory partitions associated with many different processors, and a message–passing strategy is normally used to transfer data from one partition to another. When a processor routinely requires data from remote partitions this approach necessarily incurs much larger computational penalties, and makes Fourier transformation of large datasets an expensive and inconvenient procedure.

At present the best strategy for distributed parallel Fourier transforms appears to lie in gathering together all of the component parts of a single dataset into the memory partition of a single processor. The transform may then be carried out locally using the FFT, and the component parts of the transform may then be scattered back to their points of origin. Given a dataset of size N divided up into P approximately equal partitions the computational cost is of order $2(P-1)t + N \log_2 N$, where t is the average time to effect a data–transfer using message–passing. A drawback is that this strategy is not scalable, in the sense that it requires sufficent memory to accommodate the entire dataset to be available within the memory partition of a single processor. In practice this is reasonable for one–dimensional datasets.

The development of genuinely parallel Fourier transform algorithms remains an active area of research, with emphasis on multidimensional transforms in a massively–parallel distributed–memory environment [11].

Note on Evaluation of Trigonometrical Functions

In the computation of discrete Fourier transforms it is often necessary to evaluate terms of the form

$$\exp(ik\theta) = \cos k\theta + i \sin k\theta$$

repeatedly for successive values of the integer k, while the angle θ remains fixed. The computational cost of repeated calls to the mathematical functions *sin* and *cos* can be high, and it is significantly more efficient to use the recurrence relations

$$\begin{aligned} \cos k\theta &= \cos\theta \cos(k-1)\theta - \sin\theta \sin(k-1)\theta \\ \sin k\theta &= \sin\theta \cos(k-1)\theta + \cos\theta \sin(k-1)\theta \end{aligned}$$

The recurrences are initialised by evaluating $\cos\theta$ and $\sin\theta$ once each, and no further mathematical function calls are required. Setting $\cos k\theta = 1$ and $\sin k\theta = 0$ for $k = 0$ the recurrences may be used to obtain sines and cosines for any positive k.

References

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery: *Numerical Recipes*, Cambridge University Press, 3rd ed., 2007.

[2] E.O. Brigham: *The Fast Fourier Transform*, Prentice–Hall, 1968.

[3] G.K. Batchelor: *The Theory of Homogeneous Turbulence*, Cambridge University Press, 1953.

[4] C. Canuto, M.Y. Hussaini, A. Quarteroni, T.A. Zang: *Spectral Methods in Fluid Dynamics*, Springer–Verlag, 1988.

[5] S.A. Orszag, Numerical methods for the simulation of turbulence, Phys. Fluids (suppl. II), 250–257, 1969.

[6] M.J. Lighthill: *Fourier Analysis and Generalised Functions*, Cambridge University Press, 1972.

[7] J.W. Cooley, J.W. Tukey: An algorithm for the machine calculation of complex Fourier series, Math. Comp. **19**, 297–301, 1965.

[8] C. Temperton: Self–sorting in–place fast Fourier transforms, SIAM J. Sci. Stat. Comp. **12**, 808–823, 1991.

[9] C. Temperton: A generalised prime factor FFT algorithm for any $N = 2^p 3^q 5^r$, SIAM J. Sci. Stat. Comp. **13**, 676–686, 1992.

[10] L.I. Bluestein: A linear filtering approach to the computation of the discrete Fourier transform, Nerem Record, 218–219, 1968.

[11] R.J. Allan: Parallel application software on high performance computers: serial and parallel FFT routines, CSED Report, Daresbury Laboratory, UK, ISSN 1362–0193, 2nd ed., 1999.

Routines

A set of FORTRAN subroutines for Fourier transformation and related tasks has been developed and is described below. The interface to each routine is given together with a brief description of its purpose. Any significant workspace requirements are indicated. The source code for all of these routines is freely available from the author.

DFT routines

```
SUBROUTINE DFTF1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine carries out a discrete Fourier transform using equation (5), or alternatively a discrete inverse transform using equation (6). The array `CARRAY` is assumed to contain `NX` complex numbers arranged as (real part, imaginary part) pairs. Indexing of both physical space and Fourier space data is standard. The value of `NX` can be any positive integer, even or odd, and the execution time of the routine scales as $\texttt{NX}^2$ for any value of `NX`. The routine requires one workspace array of size `2*NFTMAX` and four workspace arrays of size `NFTMAX` which are all declared internally. The value of `NFTMAX` is equal to the maximum value of `NX` that can be handled by the routine, and is set nominally to 1024. This value may be increased or decreased as required. The parameter `IFORW` must be set to 1 for a forward transform or to -1 for an inverse transform. In either case the transform is returned in `CARRAY` as a set of `NX` complex numbers, with standard indexing. Note that the inverse transform is left unscaled, i.e. there is no division by `NX`.

```
SUBROUTINE DFTP1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine carries out a discrete Fourier transform in exactly the same manner as `DFTF1D`, but with the necessary sine and cosine functions precomputed (using subroutine `DFTPIN`) in order to save execution time where repeated transforms are required. All variables are treated in the same way as for subroutine `DFTF1D`. Workspace requirements are for one array of size `2*NFTMAX` and four of size `NFTMAX`, with two of the smaller arrays held in `COMMON` and shared with subroutine `DFTPIN`. Again, `NFTMAX` is set to a nominal value of 1024 but may be adjusted as required.

```
SUBROUTINE DFTPIN(NX,IFORW)
INTEGER NX,IFORW
```

This routine precomputes the necessary sine and cosine functions for subroutine `DFTP1D`. Workspace requirements are confined to two arrays of size `NFTMAX` held in `COMMON` and shared with `DFTP1D`. The intention is that `DFTPIN` should be called once for each new value of `NX` and/or `IFORW`, thus allowing repeated calls to `DFTP1D` until a new size of transform or a change of transform type is required.

FFT routines: Cooley–Tukey algorithm (radix 2 only)

```
SUBROUTINE FFTF1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes a 1D Fast Fourier Transform using the Cooley–Tukey radix–2 algorithm. `NX` must be an integer power of two. No workspace is required since the transform is done in place. An upper limit on `NX` is fixed by the size of the bit–reversal arrays, which are set to a nominal size `NBTMAX` = 12 integer elements corresponding to NX $= 2^{11} = 1024$. This may be changed as required. The execution time scales as NX $\log_2$ (NX) for allowed values of NX. Note that the inverse transform is left unscaled.

```
SUBROUTINE FFTB1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine carries out a 1D Fast Fourier Transform in exactly the same manner as FFTF1D, but with the bit reversal index table precomputed (using subroutine FFTBIN) in order to save execution time where repeated transforms are required. All variables are treated in the same way as for subroutine FFTF1D. In particular NX must be an integer power of two. Workspace requirements are for one integer array of size NFTMAX held in COMMON with subroutine FFTBIN and used to form the bit reversal index table. Again, NFTMAX is set to a nominal value of 1024 but may be adjusted as required.

```
SUBROUTINE FFTBIN(NX,IFORW)
INTEGER NX,IFORW
```

This routine precomputes the bit reversal index table for subroutine FFTB1D. Again, NX must be an integer power of two. Workspace requirements are confined to a single integer array of size NFTMAX held in COMMON and shared with FFTB1D. The intention is that FFTBIN should be called once for each new value of NX and/or IFORW, thus allowing repeated calls to FFTB1D until a new size of transform or a change of transform type is required.

```
SUBROUTINE FFTR1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(NX)
```

This routine computes the 1D Fast Fourier Transform of a single real–valued dataset. CARRAY is an array of NX (double precision) real numbers, and NX must be an integer power of two. Workspace requirements are for two arrays of size NFTMAX, both declared internally. The value of NFTMAX is set nominally to 1024 but may be changed if necessary. If IFORW is set to 1 the routine carries out a forward transform, taking the contents of CARRAY on entry as a real dataset in physical space. On exit, CARRAY contains the NX/2 complex values of the positive–wavenumber half of the Fourier transform. The indexing is standard, except that the first complex array element contains the (real) transform values corresponding to $k = 0$ and $k =$ NX/2. For an inverse transform (with IFORW set to -1) the data must be supplied in the same format. Note that the inverse transform is left unscaled.

```
SUBROUTINE FFTT1D(ARRAY1,ARRAY2,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(NX)
```

This routine computes the 1D Fast Fourier Transform of two real–valued datasets. The interface is where each of ARRAY1 and ARRAY2 is an array of NX (double precision) real numbers, and NX must be an integer power of two. Workspace requirements are for one array of size 2*NFTMAX, declared internally. The value of NFTMAX is set nominally to 1024 but may be changed if necessary. For a forward transform (IFORW equal to 1) the contents of ARRAY1 and ARRAY2 are each taken on entry as a real dataset in physical space. On exit, each array contains the NX/2 complex values of the positive–wavenumber half of the Fourier transform of the corresponding dataset. The indexing is standard, except that the first complex array element contains the (real) transform values corresponding to $k = 0$ and k=NX/2. For an inverse transform (with IFORW set to -1) the data must be supplied in the same format. Note that the inverse transform is left unscaled.

```
SUBROUTINE FFTF2D(CARRRE,CARRIM,NXPHYS,NYPHYS,NX,NY,IFORW)
INTEGER NXPHYS,NYPHYS,NX,NY,IFORW
DOUBLE PRECISION CARRRE(NXPHYS,NYPHYS),CARRIM(NXPHYS,NYPHYS)
```

This routine computes the Fourier transform of a two–dimensional dataset using the FFT algorithm. CARRRE and CARRIM are (double precision) real arrays of physical dimension NXPHYS by NYPHYS containing respectively the real and imaginary parts of the input dataset which is of size NX by NY. Both NX and NY must be powers of two. A single workspace array is required, of size NFTMAX with NFTMAX set nominally to 1024. This defines the maximum size of any one dimension and may be changed as required.

```
SUBROUTINE FFTF3D(CARRRE,CARRIM,NXPHYS,NYPHYS,NZPHYS,NX,NY,NZ,IFORW)
INTEGER NXPHYS,NYPHYS,NZPHYS,NX,NY,NZ,IFORW
DOUBLE PRECISION CARRRE(NXPHYS,NYPHYS,NZPHYS)
DOUBLE PRECISION CARRIM(NXPHYS,NYPHYS,NZPHYS)
```

This routine computes the Fourier transform of a three–dimensional dataset using the FFT algorithm. CARRRE and CARRIM are (double precision) real arrays of size NXPHYS by NYPHYS by NZPHYS containing respectively the real and imaginary parts of the input dataset which is of size NX by NY by NZ. All of NX, NY and NZ must be powers of two. A single workspace array is required, of size NFTMAX with NFTMAX set nominally to 1024. This defines the maximum size of any one dimension and may be changed as required.

```
SUBROUTINE FFTINT(CARRAY,NX,FRACT)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX),FRACT
```

This routine interpolates a function using the FFT. CARRAY is a complex array of size NX containing the function to be interpolated. In order to allow the use of the FFT, NX must be a power of two. The (double precision) real number FRACT contains the fractional offset Δx that defines the set of points $x_j + \Delta x$ at which to carry out the interpolation. The interpolated data is returned in CARRAY.

Routines for convolution and correlation

```
SUBROUTINE CONDIR(FFUNCT,GFUNCT,ANSWER,NX)
INTEGER NX
DOUBLE PRECISION FFUNCT(NX),GFUNCT(NX),ANSWER(NX)
```

This routine computes the discrete convolution of two real–valued functions. FFUNCT and GFUNCT are (double precision) real arrays of size NX containing the datasets whose convolution will be returned in ANSWER. There is no restriction on the value of NX. No workspace is required. The execution time required for this routine scales as $\mathtt{NX}^2$.

```
SUBROUTINE CONFFT(FFUNCT,GFUNCT,ANSWER,NX)
INTEGER NX
DOUBLE PRECISION FFUNCT(NX),GFUNCT(NX),ANSWER(NX)
```

This routine computes the discrete convolution of two real–valued functions using the FFT algorithm. FFUNCT and GFUNCT are (double precision) real arrays of size NX containing the datasets whose correlation will be returned in ANSWER. The value of NX must be an integer power of two. The workspace requirement is for two arrays of size 2*NFTMAX, both declared internally. The execution time required for this routine scales as NX $\log_2$ (NX) for allowed values of NX.

```
SUBROUTINE CORDIR(FFUNCT,GFUNCT,ANSWER,NX)
INTEGER NX
DOUBLE PRECISION FFUNCT(NX),GFUNCT(NX),ANSWER(NX)
```

This routine computes the discrete correlation between two real–valued functions. FFUNCT and GFUNCT are (double precision) real arrays of size NX containing the datasets whose correlation will be returned in ANSWER. There is no restriction on the value of NX. No workspace is required. The execution time required for this routine scales as $\mathtt{NX}^2$.

```
SUBROUTINE CORFFT(FFUNCT,GFUNCT,ANSWER,NX)
INTEGER NX
DOUBLE PRECISION FFUNCT(NX),GFUNCT(NX),ANSWER(NX)
```

This routine computes the discrete correlation between two real–valued functions using the FFT algorithm. FFUNCT and GFUNCT are (double precision) real arrays of size NX containing the datasets whose correlation will be returned in ANSWER. The value of NX must be an integer power of two. The workspace requirement is for two arrays of size 2*NFTMAX, both declared internally. The execution time required for this routine scales as NX $\log_2$ (NX) for allowed values of NX.

FFT Routines: Bluestein algorithm (arbitrary length)

```
SUBROUTINE FFTC1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes a 1D discrete Fourier transform using the convolutive method described above. CARRAY contains complex data as for DFTF1D and FFTF1D, and NX may take any value. Three workspace arrays are required, one of size -NFTMAX:NFTMAX and two of size 2*NFTMAX, and these are declared internally. The value of NFTMAX is set nominally to 1024, but this may be changed if necessary. Execution time for this routine scales as NT $\log_2$ (NT) where NT is the smallest integer power of two greater than 2*NX-1. Note that the inverse transform is left unscaled.

```
SUBROUTINE FFTP1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine carries out a discrete Fourier transform in exactly the same manner as FFTC1D, but with the necessary coefficient functions precomputed (using subroutine FFTPIN) in order to save execution time where repeated transforms are required. All variables are treated in the same way as for subroutine FFTC1D. Workspace requirements are for one array of size -NFTMAX:NFTMAX and two arrays of size 2*NFTMAX, all held in COMMON and shared with subroutine FFTPIN. Again, NFTMAX is set to a nominal value of 1024 but may be adjusted as required.

```
SUBROUTINE FFTPIN(NX,IFORW)
INTEGER NX,IFORW
```

This routine precomputes the necessary coefficient functions for subroutine FFTP1D. Workspace requirements are for one array of size -NFTMAX:NFTMAX and two arrays of size 2*NFTMAX, all held in COMMON and shared with FFTP1D. The intention is that FFTPIN should be called once for each new value of NX and/or IFORW, thus allowing repeated calls to FFTP1D until a new size of transform or a change of transform type is required.

```
SUBROUTINE FFTA1D(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes a 1D discrete Fourier transform using either the FFT or the convolutive method described above. CARRAY contains complex data as for FFTF1D and FFTC1D, and NX may take any value. Workspace arrays are required as for both of these routines. If NX is a power of two then the FFT is used, otherwise the convolutive algorithm is selected.

FFT routines: Temperton algorithm

```
SUBROUTINE FFTPF2(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes the fast Fourier transform using the Temperton prime factor algorithm. NX must be an integer power of two. No workspace is required since the transform is done in place, but two arrays each of length NFTMAX are used internally to store trigonometric factors. NFTMAX is set nominally to 1024 and may be changed if required.

```
SUBROUTINE FFTPF3(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPF2, but for NX equal to an integer power of 3.

```
SUBROUTINE FFTPF5(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPF2, but for NX equal to an integer power of 5.

```
SUBROUTINE FFTPF7(CARRAY,NX,IFORW)
INTEGER NX,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPF2, but for NX equal to an integer power of 7.

```
SUBROUTINE FFTPFA(CARRAY,NX,IFORW,NRADIX)
INTEGER NX,IFORW,NRADIX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPF2, but for NX equal to an integer power of an arbitrary radix specified as NRADIX. The maximum value of NRADIX is defined internally as NRDXMX which is set nominally to 11. This value may be changed as required.

FFT routines: Temperton algorithm with rotations

```
SUBROUTINE FFTPR2(CARRAY,NX,NI,IFORW)
INTEGER NX,NI,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes the fast Fourier transform using the Temperton prime factor algorithm with rotations. NX must be an integer power of two, and NI is the value of the required rotation. No workspace is required since the transform is done in place, but two arrays each of length NFTMAX are used internally to store trigonometric factors. NFTMAX is set nominally to 1024 and may be changed if required.

```
SUBROUTINE FFTPR3(CARRAY,NX,NI,IFORW)
INTEGER NX,NI,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPR2, but for NX equal to an integer power of 3.

```
SUBROUTINE FFTPR5(CARRAY,NX,NI,IFORW)
INTEGER NX,NI,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPR2, but for NX equal to an integer power of 5.

```
SUBROUTINE FFTPR7(CARRAY,NX,NI,IFORW)
INTEGER NX,NI,IFORW
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPR2, but for NX equal to an integer power of 7.

```
SUBROUTINE FFTPRA(CARRAY,NX,NI,IFORW,NRADIX)
INTEGER NX,NI,IFORW,NRADIX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTPR2, but for NX equal to an integer power of an arbitrary radix specified as NRADIX. The maximum value of NRADIX is defined internally as NRDXMX which is set nominally to 11. This value may be changed as required.

FFT routines: Temperton algorithm with precomputed rotations

```
SUBROUTINE FFTFR2(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes the fast Fourier transform using the Temperton prime factor algorithm with precomputed rotations, and must be initialised using subroutine FFTNR2. NX must be an integer power of two. No workspace is required since the transform is done in place, but two arrays each of length NFTMAX are held in COMMON with subroutine FFTNR2 to store trigonometric factors. NFTMAX is set nominally to 1024 and may be changed if required.

```
SUBROUTINE FFTFR3(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTFR2, but for NX equal to an integer power of 3. Initialised by subroutine FFTNR3.

```
SUBROUTINE FFTFR5(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTFR2, but for NX equal to an integer power of 5. Initialised by subroutine FFTNR5.

```
SUBROUTINE FFTFR7(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTFR2, but for NX equal to an integer power of 7. Initialised by subroutine FFTNR7.

```
SUBROUTINE FFTFRA(CARRAY,NX)
INTEGER NX,NRADIX
DOUBLE PRECISION CARRAY(2*NX)
```

As FFTFR2, but for NX equal to an integer power of an arbitrary radix. Initialised by subroutine FFTNRA. NX must be an integer power of the radix specified in the call to subroutine FFTNRA. The maximum value of the radix is defined internally as NRDXMX which is set nominally to 11. This value may be changed as required.

```
SUBROUTINE FFTNR2(NX,NI,IFORW)
INTEGER NX,NI,IFORW
```

This routine precomputes the rotated trigonometric functions for subroutine FFTNR2. NX must be an integer power of two and NI is the required rotation. Two arrays each of length NFTMAX are held in COMMON with subroutine FFTFR2. The intention is that FFTNR2 should be called once for each new value of NX, NI and/or IFORW, thus allowing repeated calls to FFTFR2 until a new size of transform, a new rotation or a change of transform type is required.

```
SUBROUTINE FFTNR3(NX,NI,IFORW)
INTEGER NX,NI,IFORW
```

As FFTNR2 but for initialisation of radix 3 subroutine FFTFR3.

```
SUBROUTINE FFTNR5(NX,NI,IFORW)
INTEGER NX,NI,IFORW
```

As FFTNR2 but for initialisation of radix 5 subroutine FFTFR5.

```
SUBROUTINE FFTNR7(NX,NI,IFORW)
INTEGER NX,NI,IFORW
```

As FFTNR2 but for initialisation of radix 7 subroutine FFTFR7.

```
SUBROUTINE FFTNRA(NX,NI,IFORW,NRADIX)
INTEGER NX,NI,IFORW,NRADIX
```

This routine precomputes that rotated trigonometric functions for subroutine FFTFRA. The maximum value of the radix NRADIX is defined internally as NRDXMX which is set nominally to 11. This value may be changed as required.

FFT routines: multiple radix Temperton algorithm

```
SUBROUTINE FFTFFG(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes a fast Fourier transform using the Temperton prime factor algorithm for dataset length NX $= 2^p 3^q 5^r 7^s 11^t$. It is initialised by subroutine FFTNFG. This routine calls subroutines FFTFR2, FFTFR3, FFTFR5, FFTFR7 and FFTFRA.

```
SUBROUTINE FFTNFG(NX,IFORW)
INTEGER NX,IFORW
```

This routine initialises subroutine FFTFFG. It calls subroutines FFTNR2, FFTNR3, FFTNR5, FFTNR7 and FFTNRA.

FFT routines: Arbitrary length datasets (Temperton/Bluestein)

```
SUBROUTINE FFTGEN(CARRAY,NX)
INTEGER NX
DOUBLE PRECISION CARRAY(2*NX)
```

This routine computes a fast Fourier transform using the Temperton prime factor algorithm for dataset length NX $= 2^p 3^q 5^r 7^s 11^t$, and the Bluestein algorithm for all other dataset lengths. It is initialised by subroutine FFTGIN, and calls subroutines FFTFR2, FFTFR3, FFTFR5, FFTFR7, FFTFRA and FFTP1D. Also requires subroutine FFTF1D.

```
SUBROUTINE FFTGIN(NX,IFORW)
INTEGER NX,IFORW
```

This routine initialises subroutine FFTGEN, and calls subroutines FFTNR2, FFTNR3, FFTNR5, FFTNR7, FFTNRA and FFTPIN. Also requires subroutine FFTF1D.