Notes on FFT-based differentiation

Steven G. Johnson, MIT Applied Mathematics

Created April, 2011, updated May 4, 2011.

Abstract

A common numerical technique is to differentiate some sampled function $y(x)$ via fast Fourier transforms (FFTs). Equivalently, one differentiates an approximate Fourier series. Equivalently, one differentiates a trigonometric interpolation. These are also known as *spectral* differentiation methods. However, the implementation of such methods is prey to common confusions due to the *aliasing* phenomenon inherent in sampling, and the treatment of the maximum-frequency (*Nyquist*) component is especially tricky. In this note, we review the basic ideas of spectral differentiation (for equally spaced samples) and a correct resolution of the aliasing problems for implementing operations $\frac{dy}{dx}$, $\frac{d^2y}{dx^2}$, $\frac{d}{dx}\left[c(x)\frac{dy}{dx}\right]$, and ∇^2. One surprising consequence of aliasing for the Nyquist component is that the sampled second-derivative operation is *not* equivalent to two sampled first-derivative operations! The operation $\frac{d}{dx}\left[c(x)\frac{dy}{dx}\right]$ is particularly subtle because of the interplay of aliasing and desired algebraic properties of the differential operator. (Readers who simply want to know what algorithms to follow, rather than the details of the derivations, can read the first two introductory paragraphs and then skip to the labelled **Algorithm** boxes.)

1 Background

In spectral methods for differential equations, considering one dimension here for simplicity, one has a periodic function $y(x)$ with period L that one conceptually expands as a Fourier series:

$$y(x) = \sum_{k=-\infty}^{\infty} Y_k e^{\frac{2\pi i}{L}kx}$$

for Fourier coefficients $Y_k = \frac{1}{L}\int_0^L e^{-\frac{2\pi i}{L}kx} y(x)\,dx$. One then wishes to apply differential operators like $\frac{d}{dx}$, $\frac{d^2}{dx^2}$, and more generally $\frac{d}{dx}c(x)\frac{d}{dx}$ [for some periodic function $c(x)$]. Differentiation is performed term-by-term[1] in Fourier domain, and multiplication by functions $c(x)$ is done by transforming back to space domain for the multiplication (equivalent to convolution in Fourier domain). For example,

$$\frac{d}{dx}y(x) = y'(x) = \sum_{k=-\infty}^{\infty} \left(\frac{2\pi i}{L}k \cdot Y_k\right) e^{\frac{2\pi i}{L}kx},$$

[1] You may have been disturbed by rumors from real analysts that differentiating a Fourier series term-by-term is not always valid. Don't worry. For one thing, even if we required pointwise convergence of the Fourier series, differentiating term-by-term is valid for the $y(x)$ that typically appear in practice, where y is usually continuous and y' is at least piecewise differentiable. More generally, in practical applications of Fourier analysis, such as for PDEs, we are ordinarily not interested in pointwise convergence—we only care about "weak" convergence (equality when both sides are integrated against smooth, localized test functions), in which case differentiation term-by-term is always valid for any generalized function y.

which is just a pointwise multiplication of each Y_k by a term proportional to k.

To implement this on a computer, one approximates the Fourier series by a *discrete Fourier transform* (*DFT*). That is, we replace the function $y(x)$ by N discrete *samples* $y_n = y(nL/N)$ for $n = 0, 1, \ldots, N-1$, and Y_k is approximated by:

$$Y_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n e^{-\frac{2\pi i}{N} nk},$$

a DFT (although sign and normalization conventions for the DFT vary[2]). The inverse transform (IDFT) to compute y_n from Y_k is very similar:

$$y_n = \sum_{k=0}^{N-1} Y_k e^{+\frac{2\pi i}{N} nk}.$$

On a computer, the nice thing about these expressions is that all the Y_k can be computed from the y_n, or vice versa, in $\Theta(N \log N)$ operations by a *fast Fourier transform* (*FFT*) algorithm. This makes working with Fourier series practical: we can quickly transform back and forth between space domain [where multiplying by functions like $c(x)$ is easy] and Fourier domain [where operations like derivatives are easy]. Almost invariably, FFT implementations compute DFTs and IDFTs in forms similar to these equations, with the Y_k coefficients arranged "in order" from $k = 0$ to $N-1$, and this ordering turns out to make the correct implementation of FFT-based differentiation more obscure.

In order to compute derivatives like $y'(x)$, we need to do more than express y_n. We need to use the IDFT expression to define a continuous interpolation between the samples y_n—this is called *trigonometric interpolation*—and then differentiate this interpolation. At first glance, interpolating seems very straightforward: one simply evaluates the IDFT expression at non-integer $n \in \mathbb{R}$. This indeed defines *an* interpolation, but it is not the *only* interpolation, nor is it the *best* interpolation for this purpose. The reason that there is more than one interpolation is due to *aliasing*: any term $e^{+\frac{2\pi i}{N} nk} Y_k$ in the IDFT can be replaced by $e^{+\frac{2\pi i}{N} n(k+mN)} Y_k$ for any integer m and still give the *same* samples y_n, since $e^{\frac{2\pi i}{N} nmN} = e^{2\pi i nm} = 1$ for any integers m and n. Essentially, adding the mN term to k means that the interpolated function $y(x)$ just oscillates m extra times in between the sample points, i.e. between n and $n+1$, which has no effect on y_n but has a huge effect on derivatives. To resolve this ambiguity, one imposes additional criteria—e.g. a bandlimited spectrum and/or minimizing some derivative of the interpolated $y(x)$—and we will then explore the consequences of the disambiguated interpolation for various derivative operations.

2 Trigonometric interpolation

What is the right way to do trigonometric interpolation? We can define a more arbitrary interpolated function $y(x)$ (but still limited to only N frequency components) by substituting $n = Nx/L$ into the IDFT

[2] In our FFTW implementation of FFTs, this DFT is termed an `FFTW_FORWARD` transform, but the $1/N$ normalization is omitted and must be supplied by the user. (In computing derivatives by the procedures described below, the $1/N$ factor can be combined at no cost with the multiplications by $2\pi/L$.) The IDFT is termed an `FFTW_BACKWARD` transform. In Matlab, the $1/N$ normalization is moved from the DFT (Matlab's `fft` function) to the IDFT (Matlab's `ifft` function), which doesn't affect any of the procedures in this note. Beware that Matlab uses indices that start with 1 rather than 0, however: if `z = fft(y)` in Matlab, then `z(k+1)` corresponds to NY_k.

and allowing any arbitrary aliasing integer m_k for each Y_k:[3]

$$y(x) = \sum_{k=0}^{N-1} Y_k e^{\frac{2\pi i}{L}(k+m_k N)x}.$$

Regardless of the values of m_k, this gives the same sample points $y_n = y(nL/N)$, but changing m_k greatly modifies $y(x)$ in between the samples. In order to uniquely determine the m_k, a useful criterion is that we wish to *oscillate as little as possible* between the sample points y_n. One way to express this idea is to assume that $y(x)$ is *bandlimited* to frequencies $|k + m_k N| \le N/2$. Another approach, that gives the same result given this form of $y(x)$ (i.e., what are the "best" N frequency components?), is to *minimize the mean-square slope*

$$\begin{aligned}
\frac{1}{L}\int_0^L |y'(x)|^2 dx &= \frac{1}{L}\int_0^L \left| \sum_{k=0}^{N-1} \frac{2\pi i}{L}(k+m_k N) Y_k e^{\frac{2\pi i}{L}(k+m_k N)x} \right|^2 dx \\
&= \sum_{k=0}^{N-1}\sum_{k'=0}^{N-1} \frac{1}{L}\int_0^L \overline{\frac{2\pi i}{L}(k+m_k N) Y_k e^{\frac{2\pi i}{L}(k+m_k N)x}} \frac{2\pi i}{L}(k'+m_{k'} N) Y_{k'} e^{\frac{2\pi i}{L}(k'+m_{k'} N)x} dx \\
&= \left(\frac{2\pi}{L}\right)^2 \sum_{k=0}^{N-1} |Y_k|^2 (k+m_k N)^2,
\end{aligned}$$

where in the last step we have used the orthogonality of the Fourier basis functions: $\int_0^L \overline{e^{\frac{2\pi i}{L}(k+m_k N)x}} e^{\frac{2\pi i}{L}(k'+m_{k'} N)x} dx = 0$ if $k + m_k N \ne k' + m_{k'} N$, which for $0 \le k, k' < N$ boils down to being zero for $k \ne k'$ (and giving L for $k = k'$). From this expression, for a given set of coefficients Y_k, the mean-square slope is minimized by choosing m_k that minimizes $(k + m_k N)^2$ for each k.

[3] Unfortunately, this is actually not the most general possible aliasing, because we are assigning only *one* m_k to each k. More generally, we could break each Y_k among arbitrarily many aliases $k + mN$ with amplitudes $u_{m,k} Y_k$, such that $\sum_m u_{m,k} = 1$. That is, we could write the most general trigonometric interpolation as:

$$y(x) = \sum_{k=0}^{N-1} Y_k \sum_{m=-\infty}^{\infty} u_{m,k} e^{\frac{2\pi i}{L}(k+mN)x}.$$

The mean-square slope is then $\left(\frac{2\pi}{L}\right)^2 \sum_{k=0}^{N-1} |Y_k|^2 \sum_{m=-\infty}^{\infty} |u_{m,k}|^2 (k+mN)^2$. We can eliminate the $\sum_m u_{m,k} = 1$ constraint by solving for $u_{0,k} = 1 - \sum_{m\ne 0} u_{m,k}$, yielding a mean-square slope

$$\left(\frac{2\pi}{L}\right)^2 \sum_{k=0}^{N-1} |Y_k|^2 \left[\left| 1 - \sum_{m\ne 0} u_{m,k} \right|^2 k^2 + \sum_{m\ne 0} |u_{m,k}|^2 (k+mN)^2 \right].$$

Minimizing this unconstrained expression is a straightforward calculus exercise, and yields $u_{m,k} = a_k/(k+mN)^2$, where $a_k = 1/\sum_m \frac{1}{(k+mN)^2}$: an infinite number of frequency components! (In fact, this interpolation is especially problematic, because its *second* derivative y'' actually diverges! By allowing infinitely many aliases, we are allowing *any* periodic function, and the minimial-slope periodic interpolant is simply *piecewise-linear* interpolation, with infinite y'' at the "kinks" at each sample.) To restrict ourselves to a single nonzero $u_{m,k}$ for each k, one possiblity is to restrict ourselves to one frequency $|k + mN|$ per Y_k so that we are asking for the "best" choice of N frequency components in the minimal-slope sense. Of course, if we further assume that the spectrum is bandlimited to $|k + mN| \le N/2$, that completely determines the aliasing question by itself (except for the $k = N/2$ component, which can be determined by e.g. requiring that the interpolant be real given real y_n). Alternatively, rather than any *a priori* bandlimiting assumption, we can obtain the desired result below if we minimize the mean-square p-th derivative $\int |y^{(p)}(x)|^2 dx$, and then take the $p \to \infty$ limit.

If $0 \le k < N/2$, then $(k + m_k N)^2$ is minimized for $m_k = 0$. If $N/2 < k < N$, then $(k + m_k N)^2$ is minimized for $m_k = -1$. If $k = N/2$ (for even N), however, there is an ambiguity: either $m_k = 0$ or -1 gives the same value $(k + m_k N)^2 = (N/2)^2$. For this $Y_{N/2}$ term (the "Nyquist" term), we need to consider a more general class of aliasing: we can arbitrarily split up the $Y_{N/2}$ term between $m = 0$ [$\frac{2\pi i}{L}\frac{N}{2}x = +\frac{\pi i}{L}Nx$, positive frequency] and $m = -1$ [$\frac{2\pi i}{L}(\frac{N}{2} - N)x = -\frac{\pi i}{L}Nx$, negative frequency]:

$$Y_{N/2}\left[ue^{+\frac{\pi i}{L}Nx} + (1-u)e^{-\frac{\pi i}{L}Nx}\right]$$

where u is an arbitrary complex number to be determined, so that at sample points $x = nL/N$ we get a coefficient $uY_{N/2}+(1-u)Y_{N/2} = Y_{N/2}$ multiplying $e^{\pm i\pi n} = (-1)^n$ and so recover the IDFT. The contribution to the mean-square slope from this term is then

$$\left(\frac{\pi N}{L}\right)^2 |Y_{N/2}|^2 \left[|u|^2 + |1-u|^2\right],$$

which is minimized for $u = 1/2$: the $Y_{N/2}$ term should be *equally split* between the frequencies $\pm\frac{\pi N}{L}$, giving a $\cos(\pi Nx/L)$ term. This results in the **unique "minimal-oscillation" trigonometric interpolation** of order N:

$$\boxed{y(x) = Y_0 + \sum_{0<k<N/2}\left(Y_k e^{+\frac{2\pi i}{L}kx} + Y_{N-k}e^{-\frac{2\pi i}{L}kx}\right) + Y_{N/2}\cos\left(\frac{\pi}{L}Nx\right),}$$

where the $N/2$ (Nyquist) term is absent for odd N.

As a useful side effect, this choice of trigonometric interpolation has the property that real-valued samples y_n (for which Y_0 is real and $Y_{N-k} = \overline{Y_k}$) will result in a purely real-valued interpolation $y(x)$ for all x. (A number of FFT implementations, such as FFTW, offer specialized functions for the case where y_n is real, in which case they only compute Y_k for $k \le N/2$ since the remaining Y_k are redundant.)

3 First and second derivatives

Let us consider how to compute the derivatives $y'_n = y'(nL/N)$ and $y''_n = y''(nL/N)$ at the sample points, using FFTs to compute the trigonometric interpolation coefficients. Counterintuitively, because of the $Y_{N/2}$ term for even N, the second derivative is *not* equivalent to performing the first-derivative operation twice!

The first derivative of $y(x)$ is:

$$y'(x) = \sum_{0<k<N/2}\frac{2\pi i}{L}k\left(Y_k e^{+\frac{2\pi i}{L}kx} - Y_{N-k}e^{-\frac{2\pi i}{L}kx}\right) - \frac{\pi}{L}NY_{N/2}\sin\left(\frac{\pi}{L}Nx\right).$$

When we evaluate this at the sample points $x = nL/N$, however, we obtain:

$$y'_n = y'(nL/N) = \sum_{0<k<N/2}\frac{2\pi i}{L}k\left(Y_k e^{+\frac{2\pi i}{N}nk} - Y_{N-k}e^{-\frac{2\pi i}{N}nk}\right) = \sum_{k=0}^{N-1} Y'_k e^{\frac{2\pi i}{N}nk},$$

where the $Y_{N/2}$ term has vanished because $\sin(\pi n) = 0$.[4] The resulting procedure to compute y'_n via FFTs is given in Algorithm 1.

[4]This is quite important, because any nonzero imaginary coefficient for $Y_{N/2}$ in $Y'_{N/2}$ would have caused problems, most obviously that it would have made the derivative of real y_n give a complex y'_n. A nonzero *real* coefficient of $Y_{N/2}$ would break another symmetry: it would spoil the property that the derivative of an even function should be odd and vice versa; more subtly, it would spoil a skew-symmetric property of the underlying d/dx operator.

Algorithm 1 Compute the sampled first derivative $y'_n \approx y'(nL/N)$ from samples $y_n = y(nL/N)$.

1. Given y_n for $0 \le n < N$, use an FFT to compute Y_k for $0 \le k < N$.
2. Multiply Y_k by $\frac{2\pi i}{L}k$ for $k < N/2$, by $\frac{2\pi i}{L}(k-N)$ for $k > N/2$, and by *zero* for $k = N/2$ (if N is even), to obtain Y'_k.
3. Compute y'_n from Y'_k via an inverse FFT.

Algorithm 2 Compute the sampled second derivative $y''_n \approx y''(nL/N)$ from samples $y_n = y(nL/N)$.

1. Given y_n for $0 \le n < N$, use an FFT to compute Y_k for $0 \le k < N$.
2. Multiply Y_k by $-[\frac{2\pi}{L}k]^2$ for $k \le N/2$ and by $-[\frac{2\pi}{L}(k-N)]^2$ for $k > N/2$ to obtain Y''_k.
3. Compute y''_n from Y''_k via an inverse FFT.

On the other hand, the second derivative of $y(x)$ is

$$y''(x) = -\sum_{0<k<N/2} \left[\frac{2\pi}{L}k\right]^2 \left(Y_k e^{+\frac{2\pi i}{L}kx} + Y_{N-k} e^{-\frac{2\pi i}{L}kx}\right) - \left[\frac{\pi}{L}N\right]^2 Y_{N/2} \cos\left(\frac{\pi}{L}Nx\right),$$

which at the sample points gives

$$y''_n = y''(nL/N) = -\sum_{0<k<N/2} \left[\frac{2\pi}{L}k\right]^2 \left(Y_k e^{+\frac{2\pi i}{N}nk} + Y_{N-k} e^{-\frac{2\pi i}{N}nk}\right) - \left[\frac{\pi}{L}N\right]^2 Y_{N/2}(-1)^n = \sum_{k=0}^{N-1} Y''_k e^{\frac{2\pi i}{N}nk},$$

where the $Y_{N/2}$ term has *not* vanished. The resulting procedure to compute y''_n via FFTs is given in Algorithm 2. Note that the $Y_{N/2}$ term is multiplied by $-[\frac{\pi}{L}N]^2$ regardless of whether we use $-[\frac{2\pi}{L}k]^2$ or $-[\frac{2\pi}{L}(k-N)]^2$ for that term (i.e., it doesn't matter whether we assign a positive or negative frequency to $k = N/2$). Moreover, this procedure is *not* equivalent to performing the spectral first-derivative procedure (Algorithm 1) twice (unless N is odd so that there is no $Y_{N/2}$ term) because the first-derivative operation omits the $Y_{N/2}$ term entirely.[5]

Similarly for higher derivatives: the odd-order derivatives are treated similarly to the first derivative, and the even-order derivatives are treated similarly to the second derivative.

[5] You might object that the $Y_{N/2}$ term goes to zero as $N \to \infty$ anyway, assuming a convergent Fourier series, so why not just drop it and make our spectral second derivative equivalent to two spectral first derivatives? This wouldn't change the asymptotic convergence rate of the approximation, but it would change a fundamental algebraic property of the $\frac{d^2}{dx^2}$ operator on periodic functions: its nullspace. The nullspace of the exact operator consists only of constant functions (since other affine functions are not periodic), corresponding to the zero coefficient of Y_0 in y''_n. Discarding $Y_{N/2}$ would add another vector to the nullspace, making the matrix representing our spectral derivative of y_n qualitatively different from the operator it is supposed to represent. This, in turn, would cause qualitative changes in any PDE or ODE that uses $\frac{d^2}{dx^2}$. For example, the heat equation $\frac{\partial y}{\partial t} = \frac{\partial^2 y}{\partial x^2}$ is supposed to have solutions that decay to constants, but discarding $Y_{N/2}$ means that the Nyquist component would not decay—as $t \to \infty$, you would get solutions that are the sum of a constant and a $(-1)^n$ oscillation. (Of course, the lack of a $Y_{N/2}$ contribution in the *first* derivative means that we *have* changed the nullspace there, but for the first derivative there is no good solution except to use an odd N, since any nonzero $Y'_{N/2}$ would cause other problems as noted above.)

4 Position-varying differential operators $\frac{d}{dx}c(x)\frac{d}{dx}$

A more complicated problem is the implementation of position-varying operators of the form $\frac{d}{dx}c(x)\frac{d}{dx}$, i.e. when computing $u(x) = [c(x)y'(x)]'$ for a periodic coefficient function $c(x)$. This kind of operator shows up, for example, in the heat/diffusion equation with spatially varying diffusion coefficients, in wave equations with spatially varying wave speeds, and in Poisson's equation with a spatially varying permittivity.

Generally speaking, the technique to compute such operations efficiently in a spectral method is to transform back and forth between space domain and Fourier domain: perform derivatives in Fourier domain (where you just multiply Y_k by the right factor) and multiply by $c(x)$ in space domain [where you just multiply pointwise by $c_n = c(nL/N)$]. However, this is complicated by the $Y_{N/2}$ element: as explained in the previous section, the special handling of $Y_{N/2}$ means that the second-derivative operation is *not* the same as two first-derivative operations in sequence.

Let us consider what happens to the $Y_{N/2}$ term in $y(x)$ when computing $[c(x)y'(x)]'$. This term contributes:

$$\left[-c(x)\frac{\pi}{L}NY_{N/2}\sin\left(\frac{\pi}{L}Nx\right)\right]' = -c'(x)\frac{\pi}{L}NY_{N/2}\sin\left(\frac{\pi}{L}Nx\right) - c(x)\left[\frac{\pi}{L}N\right]^2 Y_{N/2}\cos\left(\frac{\pi}{L}Nx\right),$$

which at a sample point $x = nL/N$ gives:

$$-c_n\left[\frac{\pi}{L}N\right]^2 Y_{N/2}(-1)^n,$$

where $c_n = c(nL/N)$, since the sine term vanishes. However, the obvious approach of simply adding this term into the result is problematic for a subtle reason: it breaks the key *self-adjointness* (Hermitian) property of the exact operator $\frac{d}{dx}c(x)\frac{d}{dx}$ for real $c(x)$. That is, this $Y_{N/2}$ term corresponds to multiplying y_n by a non-Hermitian matrix

$$-\left[\frac{\pi}{L}N\right]^2 \begin{bmatrix} c_0 & & & & \\ & c_1 & & & \\ & & \ddots & & \\ & & & c_{N-2} & \\ & & & & c_{N-1} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ \vdots \\ (-1)^{N-2} \\ (-1)^{N-1} \end{bmatrix} \begin{bmatrix} 1 & -1 & \cdots & (-1)^{N-2} & (-1)^{N-1} \end{bmatrix}/N,$$

where the $[\cdots]/N$ row vector multiplied by a y_n column vector gives $Y_{N/2}$. Breaking the Hermitian property is a terrible thing for many applications, because it alters the qualitative behavior the operator in a PDE or ODE setting. Hermitian matrices are also very desirable for many linear-algebra techniques. We could simply throw out the $Y_{N/2}$ term entirely, but that causes other problems (it increases the nullspace of the operator, as mentioned in the previous section). What went wrong? The problem is that we have implicitly done something non-symmetrical in the way we approximated the operator: by multiplying $Y_{N/2}\cos(\pi Nx/L)$ by $c(x)$, we have allowed the unsampled/interpolated output to contain larger Fourier components than the input. Instead, since $\cos(\pi Nx/L)$ is already at the maximum allowed frequency (the Nyquist frequency), it makes sense to discard all but the zeroth Fourier component of $c(x)$ when multiplying by $\cos(\pi Nx/L)$. The zeroth Fourier component is just the average of $c(x)$, and hence we obtain a (low-pass filtered) contribution:

$$-\frac{\sum_{m=0}^{N-1} c_m}{N}\left[\frac{\pi}{L}N\right]^2 Y_{N/2}(-1)^n.$$

The resulting procedure to compute $u_n = u(nL/N) = [cy']'|_{nL/N}$ by FFTs is given in Algorithm 3. We have

Algorithm 3 Compute the sampled position-varying second-derivative operation $u_n \approx u(nL/N) = [cy']'|_{nL/N}$ from samples $y_n = y(nL/N)$ and a given coefficient function $c_n = c(nL/N)$.

1. Compute y'_n by applying Algorithm 1 to y_n, but **save** the original $Y_{N/2}$ Fourier coefficient (if N is even).
2. Compute $v_n = c_n y'_n$ for $0 \le n < N$.
3. Compute $u_n = v'_n$ by applying Algorithm 1 to v_n. However, **before** performing the inverse FFT of V'_k (step 3 of Algorithm 1), **change** $V'_{N/2}$ (for N even) to $V'_{N/2} = -c_{\text{mean}} \left[\frac{\pi}{L}N\right]^2 Y_{N/2}$, using the $Y_{N/2}$ from step 1, where $c_{\text{mean}} = \frac{1}{N}\sum_{m=0}^{N-1} c_m$.

Algorithm 4 Alternative to Algorithm 3 to compute the sampled position-varying second-derivative operation $u_n \approx u(nL/N) = [cy']'|_{nL/N}$ from samples $y_n = y(nL/N)$ and a given coefficient function $c_n = c(nL/N)$.

1. Compute y'_n by applying Algorithm 1 to y_n, *except* that we set $Y'_{N/2} = \frac{\pi i}{L} N Y_{N/2}$ for even N.
2. Compute $v_n = c_n y'_n$ for $0 \le n < N$.
3. Compute $u_n = v'_n$ by applying Algorithm 1 to v_n, *except* that we set $V'_{N/2} = \frac{\pi i}{L} N V_{N/2}$ for even N.

applied an additional trick to save some arithmetic: instead of adding the $Y_{N/2}$ correction separately to each output n, we equivalently add it *once* to the Nyquist component $V'_{N/2}$ before the last inverse FFT. As desired, Algorithm 3 corresponds to multiplying y_n by a Hermitian matrix for the case of real c_n, and its nullspace consists only of constant vectors for $c_n > 0$ (in which case the matrix is also negative semi-definite); in both these senses it resembles the original differential operator. Note that for the case of $c_n = 1$, Algorithm 3 gives *exactly* the same results as Algorithm 2 (neglecting roundoff errors).

You could object to this procedure, however, in that we were apparently inconsistent: why didn't we also worry about discarding Fourier component beyond the Nyquist frequency when we computed $c_n y'_n$ in step 2? (Our procedure allows the small high-frequency Fourier components of the product, those beyond the Nyquist frequency, to add to the low-frequency components by aliasing.) Of course, we could have explicitly done some low-pass filtering before multiplying, or alternatively zero-padded both Y_k and C_k in Fourier domain to a length $\ge 2N - 1$ (corresponding to using a roughly doubled-resolution interpolation when multiplying $c_n y'_n$) so that we have room for the extra Fourier coefficients, only truncating back to the coarser grid at the end. (Signal-processing people will recognize such zero-padding as turning a cyclic convolution of the Fourier coeffients into a linear convolution.) However, this more complicated (and more expensive) procedure doesn't alter the asymptotic convergence rate of u_n to the exact $u(x)$ as $N \to \infty$, nor is it necessary to preserve the Hermitian, definiteness, or nullspace properties.

Actually, there is an algorithm even slightly simpler than Algorithm 3 which also preserves the Hermitian, definiteness, and nullspace properties of $\frac{d}{dx}c\frac{d}{dx}$, has the same asymptotic convergence rate, and similarly reduces to Algorithm 2 for $c_n = 1$. In this alternative, we perform the first derivative, multiply by c_n, and then perform another first derivative as before, but we modify the first-derivative operation from Algorithm 1 to multiply the $Y_{N/2}$ coefficient by $+\frac{\pi i}{L}N$ (i.e., we treat it as a "positive" frequency, abandoning minimal-oscillation interpolation for this term) instead of zero. Because we perform *two* first derivatives, this doesn't

cause the problems it would for a single first derivative (e.g. it still produces real outputs for real inputs[6]), and because this only differs from minimal-oscillation interpolation by a multiple of $Y_{N/2}$, it doesn't change the asymptotic convergence rate. This algorithm is given in Algorithm 4.

In Algorithm 4, we arbitrarily assigned a positive frequency $+\frac{\pi i}{L}N$ to the $N/2$ components. We could equally well have assigned the $N/2$ components a negative frequency $-\frac{\pi i}{L}N$, and this arbitrariness may feel a bit unsatisfactory even if it doesn't hurt the convergence rate or algebraic properties. In order to restore the symmetry of the spectrum, we might adopt an approach inspired by the minimal-oscillation interpolation, and *average* the two cases: that is, perform both the $+\frac{\pi i}{L}N$ and the $-\frac{\pi i}{L}N$ variants of Algorithm 4, and then average the two results. This seems like twice as much work, but if we are clever we can work through the consequences of this averaging analytically and come up with a combined algorithm. The resulting "averaged" algorithm, however, turns out to be *exactly* Algorithm 3! (The proof is left as an exercise for the reader.) That is, this perspective gives us an another justification, besides the low-pass filtering argument above, for using the c_{mean} factor in Algorithm 3.

One might naively imagine another alternative: apply the product rule to compute $u = c'y' + cy''$, i.e. $u_n \approx c'_n y'_n + c_n y''_n$ where the individual derivatives are computed as described in Algorithms 1–2. However, because we are approximating the derivative operations via sampling, the product rule is no longer exact, and this approach gives *different* results from either of the algorithms above. Worse, this method does *not* preserve the Hermitian property of the operator for real $c(x)$, and so I would not recommend it.

5 Higher spatial dimensions

Analogous procedures apply to rectangular domains in higher spatial dimensions, because the higher-dimensional DFT and Fourier series are just direct products—they correspond to successive Fourier transforms along each dimension in sequence. So, you just apply the same rules along each dimension. To illustrate this, consider two dimensions: a periodic function $y(x_1, x_2)$ in a rectangular domain $[0, L_1] \times [0, L_2]$, discretized into $N_1 \times N_2$ points as $y_{n_1,n_2} = y(n_1 L_1/N_1, n_2 L_2/N_2)$. The corresponding two-dimensional DFT and inverse DFT are:

$$Y_{k_1,k_2} = \frac{1}{N_1 N_2} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} y_{n_1,n_2} e^{-\frac{2\pi i}{N_1} n_1 k_1 - \frac{2\pi i}{N_2} n_2 k_2},$$

$$y_{n_1,n_2} = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} Y_{k_1,k_2} e^{+\frac{2\pi i}{N_1} n_1 k_1 + \frac{2\pi i}{N_2} n_2 k_2}.$$

To compute a single derivative like $\frac{\partial y}{\partial x_1}$ or $\frac{\partial^2 y}{\partial x_1^2}$, we only need 1D FFTs: we can apply Algorithm 1 or 2 along the n_1 direction, once for each n_2. It is more interesting to consider a differential operatosr that involves derivatives along both directions at once, like the Laplacian $\nabla^2 = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$. The computation of $u = \nabla^2 y$ follows straightforwardly from appling Algorithm 2 to Y along both directions, and is given in Algorithm 5.

An analogue of $\frac{d}{dx}[c\frac{dy}{dx}]$ in higher dimensions is $\nabla \cdot [c\nabla y]$, where $c(\vec{x})$ is in general a square matrix. The principle here is similar to that of Algorithm 3: apply first derivatives (similar to Algorithm 1) along each direction to obtain the vector field ∇y, multiply by $c(\vec{x})$ in the space domain, and then apply first derivatives

[6] For the case of real inputs y_n and c_n, it is desirable to use specialized FFT algorithms that exploit this fact to save a factor of ~ 2 in time and storage. Superficially, the intermediate steps of Algorithms 3–4 are problematic in this context because v_n is purely imaginary rather than purely real. Because of the linearity of the DFT, however, we can instead simply factor out the i coefficients from the $\frac{2\pi i}{L}$ multiplications in steps 1 and 3 and combine them into a factor of $i^2 = -1$. That is, we multiply by (e.g.) $\frac{2\pi}{L}$ factors in step 1 and by $-\frac{2\pi}{L}$ factors in step 3, which gives the same final result and ensures real arrays at step 2.

Algorithm 5 Compute the sampled 2d Laplacian $u = \nabla^2 y$ from samples $y_{n_1,n_2} = y(n_1 L_1/N_1, n_2 L_2/N_2)$.

1. Compute Y_{k_1,k_2} using a 2D FFT of y_{n_1,n_2} for $0 \le k_1 < N_1$ and $0 \le k_2 < N_2$.
2. Define $k'_{\{1,2\}} = \begin{cases} k_{\{1,2\}} & k_{\{1,2\}} \le N_{\{1,2\}}/2 \\ k_{\{1,2\}} - N_{\{1,2\}} & \text{otherwise} \end{cases}$. Set $U_{k_1,k_2} = -\left[\left(\frac{2\pi}{L_1}k'_1\right)^2 + \left(\frac{2\pi}{L_2}k'_2\right)^2\right] Y_{k_1,k_2}$.
3. Compute u_{n_1,n_2} from the inverse 2D FFT of U_{k_1,k_2}.

(again similar to Algorithm 1) to each component and sum them to compute the final divergence. As in Algorithm 3, however, the $N/2$ Nyquist components along each direction should be added in separately to preserve the key algebraic properties of $\nabla \cdot [c\nabla y]$. For example, the 2d case, with a scalar function c (rather than the more general case of a 2×2 matrix c), is given in Algorithm 6. Although the principles are similar to those of Algorithm 3, the execution here is a bit more complicated. There are several choices of how to arrange the transforms, since we can push certain terms before or after FFTs as desired (similar to how the Nyquist correction was pushed before the inverse FFT into $V'_{N/2}$ in Algorithm 3). Here, we have arranged things in terms of 1D auxiliary arrays $A^{(\{1,2\})}_{k_{\{1,2\}}}$ in order to keep all of the 2D transforms in the form of unmodified 2D FFTs (rather than breaking them up into 1D FFTs, which is both inconvenient and probably slower). To understand Algorithm 6, it might help to consider the case of $c_{n_1,n_2} = 1$: in this case, $A^{(1)}_{k_1} = Y_{k_1,N_2/2}$ and $A^{(2)}_{k_2} = Y_{N_1/2,k_2}$, making the algorithm equivalent to Algorithm 5 for $\nabla^2 y$.

The complexity of Algorithm 6, however, makes it attractive to instead consider the alternative of abandoning the original minimal-oscillation interpolation criterion, similar to Algorithm 4, and simply assign a positive frequency to the Nyquist components. This results in a far simpler algorithm in the multidimensional case, as shown in Algorithm 7, while preserving the key algebraic properties of the $\nabla \cdot c\nabla$ operator. As for Algorithm 5, because we only change the interpolation by a factor proportional to the Nyquist components, the asymptotic convergence rate is not affected, and because there are two derivatives the bad side effects of breaking the symmetry of the first derivative are avoided (real inputs give real outputs). This algorithm has the added benefit that it works without modification when c is a 2×2 matrix.

6 Non-periodic functions

All of the above applies to spectral differentiation of periodic functions $y(x)$. If the function is non-periodic, artifacts will appear in the derivatives due to the implicit discontinuities at the endpoints. If you have the freedom to sample your function anywhere you want, a much better solution is to sample $y(x)$ at $x_n = \cos(n\pi/N)$ for $n = 0, \ldots, N$, assuming a domain $x \in [-1, -1]$, and then to use *Chebyshev* interpolation. Chebyshev interpolation and differentiation also works via FFTs, and is essentially a Fourier series via a change of variables $x = \cos\theta$. Chebyshev approximation is described in many sources you can find elsewhere, such as the books by L. N. Trefethen (author of the excellent `chebfun` package) or J. P. Boyd, both freely available online.

Algorithm 6 Computed the sampled 2d Laplacian-like operation $u = \nabla \cdot [c\nabla y]$ from samples $y_{n_1,n_2} = y(n_1 L_1/N_1, n_2 L_2/N_2)$, where $c_{n_1,n_2} = c(n_1 L_1/N_1, n_2 L_2/N_2)$ is an arbitrary scalar coefficient function. (This gives identical results to Algorithm 5 when $c_{n_1,n_2} = 1$.) See also Algorithm 7 for a simpler approach that achieves similar accuracy and other properties.

1. Compute the 2-component vector field $\vec{g}_{n_1,n_2} \approx \nabla y$:
   (a) Compute Y_{k_1,k_2} using a 2D FFT of y_{n_1,n_2} for $0 \le k_1 < N_1$ and $0 \le k_2 < N_2$.
   (b) Define $k'_{\{1,2\}} = \begin{cases} k_{\{1,2\}} & k_{\{1,2\}} < N_{\{1,2\}}/2 \\ k_{\{1,2\}} - N_{\{1,2\}} & k_{\{1,2\}} > N_{\{1,2\}}/2 \\ 0 & k_{\{1,2\}} = N_{\{1,2\}}/2 \end{cases}$. Set $\vec{G}_{k_1,k_2} = \begin{bmatrix} \frac{2\pi i}{L_1} k'_1 \\ \frac{2\pi i}{L_2} k'_2 \end{bmatrix} Y_{k_1,k_2}$ (i.e. $\vec{G}$ is a 2-component vector field).
   (c) **Save** $Y_{N_1/2,k_2}$ (if N_1 is even) for all k_2, and save $Y_{k_1,N_2/2,}$ (if N_2 is even) for all k_1—these will be used again in step 3.
   (d) Compute $\vec{g}_{n_1,n_2}$ from the inverse 2D FFTs of both components of $\vec{G}_{k_1,k_2}$.
2. Compute the 2-component vector field $\vec{v}_{n_1,n_2} = c_{n_1,n_2}\vec{g}_{n_1,n_2}$.
3. Compute two auxiliary arrays $A^{(1)}_{k_1}$ (if N_2 is even) and $A^{(2)}_{k_2}$ (if N_1 is even) from the saved Nyquist data:
   (a) Compute $c_{\text{mean},n_2} = \frac{\sum_{n_1=0}^{N_1-1} c_{n_1,n_2}}{N_1}$ and $c_{n_1,\text{mean}} = \frac{\sum_{n_2=0}^{N_2-1} c_{n_1,n_2}}{N_2}$.
   (b) Perform inverse 1D FFTs of $Y_{N_1/2,k_2}$ and $Y_{k_1,N_1/2}$ to obtain $\hat{y}_{N_1/2,n_2}$ and $\hat{y}_{n_1,N_2/2}$, respectively.
   (c) Compute $a^{(1)}_{n_1} = c_{n_1,\text{mean}} \cdot \hat{y}_{n_1,N_2/2}$ and $a^{(2)}_{n_2} = c_{\text{mean},n_2} \cdot \hat{y}_{N_1/2,n_2}$.
   (d) Perform 1D FFTs of $a^{(1)}_{n_1}$ and $a^{(2)}_{n_2}$ to obtain $A^{(1)}_{k_1}$ and $A^{(2)}_{k_2}$, respectively.
4. Compute $u_{n_1,n_2} \approx \nabla \cdot [c\nabla y]$ from $\nabla \cdot \vec{v}$:
   (a) Compute $\vec{V}_{k_1,k_2}$ using 2D FFTs of both components of $\vec{v}_{n_1,n_2}$.
   (b) Define $k'_{\{1,2\}}$ as in step 1(b). Set $U_{k_1,k_2} = \begin{bmatrix} \frac{2\pi i}{L_1} k'_1 & \frac{2\pi i}{L_2} k'_2 \end{bmatrix} \vec{V}_{k_1,k_2}$ (i.e., multiply the row vector $[\cdots]$ by the column vector $\vec{V}_{k_1,k_2}$).
   (c) If N_1 is even, **add** $-\left(\frac{\pi}{L_1} N_1\right)^2 A^{(2)}_{k_2}$ to $U_{N_1/2,k_2}$. If N_2 is even, **add** $-\left(\frac{\pi}{L_2} N_2\right)^2 A^{(1)}_{k_1}$ to $U_{k_1,N_2/2}$. (Note that $U_{N_1/2,N_2/2}$ has *both* of these factors added to it.)
   (d) Compute u_{n_1,n_2} from the inverse 2D FFT of U_{k_1,k_2}.

Algorithm 7 Alternative (analogous to Algorithm 4) to Algorithm 6 to compute the sampled 2d Laplacian-like operation $u = \nabla \cdot [c\nabla y]$ from samples $y_{n_1,n_2} = y(n_1 L_1/N_1, n_2 L_2/N_2)$, where $c_{n_1,n_2} = c(n_1 L_1/N_1, n_2 L_2/N_2)$ is an arbitrary coefficient function (either a scalar or a 2×2 matrix). (This gives identical results to Algorithm 5 when $c_{n_1,n_2} = 1$.)

1. Compute the 2-component vector field $\vec{g}_{n_1,n_2} \approx \nabla y$:
   (a) Compute Y_{k_1,k_2} using a 2D FFT of y_{n_1,n_2} for $0 \le k_1 < N_1$ and $0 \le k_2 < N_2$.
   (b) Define $k'_{\{1,2\}} = \begin{cases} k_{\{1,2\}} & k_{\{1,2\}} \le N_{\{1,2\}}/2 \\ k_{\{1,2\}} - N_{\{1,2\}} & \text{otherwise} \end{cases}$. Set $\vec{G}_{k_1,k_2} = \begin{bmatrix} \frac{2\pi i}{L_1} k'_1 \\ \frac{2\pi i}{L_2} k'_2 \end{bmatrix} Y_{k_1,k_2}$ (i.e. $\vec{G}$ is a 2-component vector field).
   (c) Compute $\vec{g}_{n_1,n_2}$ from the inverse 2D FFTs of both components of $\vec{G}_{k_1,k_2}$.
2. Compute the 2-component vector field $\vec{v}_{n_1,n_2} = c_{n_1,n_2} \vec{g}_{n_1,n_2}$.
3. Compute $u_{n_1,n_2} \approx \nabla \cdot [c\nabla y]$ from $\nabla \cdot \vec{v}$:
   (a) Compute $\vec{V}_{k_1,k_2}$ using 2D FFTs of both components of $\vec{v}_{n_1,n_2}$.
   (b) Define $k'_{\{1,2\}}$ as in step 1(b). Set $U_{k_1,k_2} = \begin{bmatrix} \frac{2\pi i}{L_1} k'_1 & \frac{2\pi i}{L_2} k'_2 \end{bmatrix} \vec{V}_{k_1,k_2}$ (i.e., multiply the row vector $[\cdots]$ by the column vector $\vec{V}_{k_1,k_2}$).
   (c) Compute u_{n_1,n_2} from the inverse 2D FFT of U_{k_1,k_2}.