

CHAPTER 1

Preliminaries

1.1. Introduction

The traditional way to introduce spectral methods starts by approximating the function as a sum of very smooth basis functions:

$$u(x) \approx \sum_{k=0}^N a_k \Phi_k(x)$$

where $\Phi_k(x)$ are polynomials or trigonometric functions. In practice, there have many choices of the basis functions such as

- $\Phi_k(x) = e^{ikx}$ (the Fourier spectral method);
- $\Phi_k(x) = T_k(x)$ (the Chebyshev spectral method);
- $\Phi_k(x) = L_k(x)$ (the Legendre spectral method);

where in the above $T_k(x)$ and $L_k(x)$ are the Chebyshev and the Legendre polynomials, respectively. In this section, we will introduce some basic ideas of spectral methods. For ease of exposition, we consider the Fourier spectral method (i.e. the basis functions are chosen as e^{ikx}) which is very efficient for periodic problems. For non-periodic problems, the Chebyshev and Legendre spectral methods are frequently employed.

1.1.1. Comparison with the finite difference/element method.

We may compare the spectral method (before actually describing it) to the finite difference method. Finite Difference (FD) methods approximate derivatives of a function by *local* arguments (such as $u'(x) \approx (u(x+h) - u(x-h))/2h$, where h is a small grid spacing) - these methods are typically designed to be exact for polynomials of low orders. This approach is very reasonable: since the derivative is a local property of a function, it makes little sense (and is costly) to invoke many function values far away from the point of interest. In contrast, spectral methods are *global*. In approximating the derivatives, spectral methods use information throughout the solution interval.

We may also compare the spectral method to the finite element method. One difference is this: the trial functions τ_k in finite element method is usually 1 at the meshpoint

$$x_k = kh, \quad h = 2\pi/N$$

and 0 at the others, whereas e^{ikx} is nonzero everywhere. That is not such an important distinction. We could produce from the exponentials an interpolating function like τ_k , which is zero at all meshpoints but one:

$$F_k(x) = \frac{1}{N} \frac{\sin\left(N(x-x_k)/2\right)}{\sin\left((x-x_k)/2\right)} \cos \frac{1}{2}(x-x_k). \quad (1.1.1)$$

Of course it is not a piecewise polynomial; that distinction is genuine. A consequence of this difference is the following: (i) Each function F_k spreads over the whole solution interval, whereas τ_k is zero in all elements not containing x_k ; (ii) The stiffness matrix is sparse for finite element methods; in the spectral it is full.

1.1.2. The computational efficiency. Since the matrix associated with spectral method is full, the spectral method seems more time consuming than finite differences or finite elements. In fact, spectral method had not been used widely for a long time. The main reason is the expensive cost in computational time. However, the discovery of the Fast Fourier Transform (FFT) due to Cooley and Tukey solves this problem. We will describe the Cooley-Tukey algorithm in Chapter 2. The main idea is the following. Let $w_N = e^{2\pi i/N}$ and

$$(\mathcal{F}_N)_{jk} = w_N^{jk} = \cos \frac{2\pi jk}{N} + i \sin \frac{2\pi jk}{N}, \quad 0 \leq j, k \leq N-1.$$

Then for any N -dimensional vector v_N , the usual N^2 operations in computing $\mathcal{F}_N v_N$ are reduced to $N \log_2 N$. The significant improvement can be seen from the following table:

N	N^2	$N \log_2 N$
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240

The algorithm of FFT and its application to some differential equations will be discussed in Chapter x and Chapter xx.

1.1.3. A simple example. Unlike finite differences or finite elements, which replace the right side u_{xx} by differences at nodes, the spectral method uses u_{xx}^N exactly. In spectral method, there is no Δx . The derivatives with respect to space variables are computed explicitly and correctly. To see this, we consider the heat equation

$$u_t = u_{xx}, \quad \text{with } u(x, 0) = u_0(x), \quad (1.1.2)$$

where $u_0(x)$ is 2π -periodic. Since the exact solution u is periodic, it can be written as an infinite Fourier series. The approximate solution u^N can be as a *finite* series:

$$u^N(x, t) = \sum_{k=0}^{N-1} a_k(t) e^{ikx}, \quad x \in [0, 2\pi),$$

where each $a_k(t)$ is to be determined. In other words, the Fourier approximation u^N is a combination of oscillations e^{ikx} up to frequency $N - 1$. Assume u^N satisfy (1.1.2). We obtain

$$\sum_{k=0}^{N-1} a'_k(t) e^{ikx} = \sum_{k=0}^{N-1} a_k(t) (ik)^2 e^{ikx}.$$

Since frequencies are uncoupled, we have

$$a'_k(t) = -k^2 a_k, \quad \text{or} \quad a_k(t) = e^{-k^2 t} a_k(0),$$

where $a_k(0)$ are determined by using the initial function:

$$a_k(0) = \frac{1}{2\pi} \int_0^{2\pi} u_0(x) e^{-ikx} dx.$$

It is an easy matter to show that

$$\begin{aligned} |u(x, t) - u^N(x, t)| &= \left| \sum_{k=N+1}^{\infty} a_k(0) e^{ikx} e^{-k^2 t} \right| \\ &\leq \max_k |a_k(0)| \sum_{k=N+1}^{\infty} e^{-k^2 t} \leq \max_{0 \leq x \leq 2\pi} |u_0(x)| \int_N^{\infty} e^{-tx^2} dx. \end{aligned} \quad (1.1.3)$$

Therefore, the error goes to zero very rapidly as N is reasonably large. The convergence rate is determined by the integral term

$$J(t, N) := \int_N^{\infty} e^{-tx^2} dx = \sqrt{\frac{\pi}{4t}} \operatorname{erfc}(\sqrt{t}N),$$

where $\operatorname{erfc}(x)$ is the complementary error function (both Fortran and MATLAB have this function). The following table lists the value of $J(t, N)$ at several values of t :

N	J(0.1, N)	J(0.5, N)	J(1, N)
1	1.8349e+00	3.9769e-01	1.3940e-01
2	1.0400e+00	5.7026e-02	4.1455e-03
3	5.0364e-01	3.3837e-03	1.9577e-05
4	2.0637e-01	7.9388e-05	1.3663e-08
5	7.1036e-02	7.1853e-07	1.3625e-12
6	2.0431e-02	2.4730e-09	1.9071e-17
7	4.8907e-03	3.2080e-12	3.7078e-23
8	9.7140e-04	1.5594e-15	9.9473e-30
9	1.5973e-04	2.8290e-19	3.6663e-37

10	2.1703e-05	1.9100e-23	1.8509e-45
----	------------	------------	------------

The above table, together with (1.1.3), suggest that the spectral approximation u^N converges to the exact solution of (1.1.2) in a rate much higher than the usual algebraic rate.

In more general problems, the equation in time will not be solved exactly. It needs a difference method with time step Δt , such as Runge-Kutta method. For derivatives with respect to space variables, there are two ways:

- 1. Stay with the harmonics e^{ikx} or $\sin kx$ or $\cos kx$, and use FFT to go between coefficients a_k and mesh values $u^N(x_j, t)$. Only the mesh values enter the difference equation in time.
- 2. Use an expansion $U = \sum U_k(t)F_k(x)$, where $F_k(x)$ is given by (1.1.1), that works directly with values U_k at mesh points (where $F_k = 1$). There is a *differentiation matrix* D that gives mesh values of the derivatives, $D_{jk} = F'_k(x_j)$. Then the approximate heat equation becomes $U_t = D^2U$.

The fact that x -derivatives are exact makes spectral methods free of phase error. Differentiation multipliers e^{ikx} gives the right factor ik while differences give the wrong factor iK :

$$\frac{e^{ik(x+h)} - e^{ik(x-h)}}{2h} = iK e^{ikx}, \quad \text{where} \quad K = \frac{\sin kh}{h}.$$

For low frequencies, when kh is small and there are enough mesh points in a wavelength, K is close to k . For higher frequencies K is significantly smaller. In the heat equation it means a slower wave velocity. For details, we refer to Richtmyer and Morton [4] and LeVeque [3].

1.2. Mathematical preliminaries

1.2.1. Hilbert space and Lax-Milgram Lemma.

1.2.2. Sobolev spaces and weighted Sobolev spaces.

1.2.3. Some useful inequalities.

1.3. Basic iterative methods

1.4. Runge-Kutta methods

Consider a scalar ordinary differential equation

$$\frac{du}{dt} = f(u, t), \quad u(t_0) = u_0. \tag{1.4.1}$$

In this section, we will introduce several Runge-Kutta type methods for the above initial value problem. The Runge-Kutta methods imitate the Taylor-series method by means of clever combinations of values of $f(u, t)$. We illustrate by deriving a second-order Runge-Kutta (RK2) procedure.

1.4.1. Second-order Runge-Kutta method. Let us begin with the Taylor series for $u(t + \Delta t)$:

$$u(t + \Delta t) = u(t) + \Delta t u'(t) + \frac{\Delta t^2}{2!} u''(t) + \frac{\Delta t^3}{3!} u'''(t) + \dots, \quad (1.4.2)$$

where $\Delta t > 0$ is a fixed number. From (1.4.1), we have

$$\begin{aligned} u'(t) &= f(u, t) \\ u''(t) &= f_t + f_u u' = f_t + f_u f \\ u'''(t) &= f_{tt} + f_{tu} f + (f_t + f_u f) f_u + f(f_{ut} + f_{uu} f) \end{aligned}$$

Here subscripts denote partial derivatives, and the chain rule of differentiation is used repeatedly. The first three terms in (1.4.2) can be written now in the form

$$\begin{aligned} u(t + \Delta t) &= u + \Delta t f + \frac{\Delta t^2}{2} (f_t + f f_u) + O(\Delta t^3) \\ &= x + \frac{\Delta t}{2} f + \frac{\Delta t}{2} (f + \Delta t f_t + \Delta t f f_u) + O(\Delta t^3) \end{aligned} \quad (1.4.3)$$

where u means $u(t)$, f means $f(u, t)$, and so on. We are able to eliminate the partial derivatives with the aid of the following result:

$$f(u + \Delta t f, t + \Delta t) = f + \Delta t f_t + \Delta t f f_u + O(\Delta t^2).$$

Equation (1.4.3) can be rewritten as

$$u(t + \Delta t) = u(t) + \frac{\Delta t}{2} f + \frac{\Delta t}{2} f(u + \Delta t f, t + \Delta t) + O(\Delta t^3).$$

Hence, the formula for advancing the solution is

$$u(t + \Delta t) = u(t) + \frac{\Delta t}{2} f(u, t) + \frac{\Delta t}{2} f(u + \Delta t f(u, t), t + \Delta t)$$

or equivalently,

$$\begin{aligned} u(t + \Delta t) &= u(t) + \frac{\Delta t}{2} (K_1 + K_2) \\ K_1 &= f(u, t), \quad K_2 = f(u + \Delta t K_1, t + \Delta t). \end{aligned}$$

This formula can be used repeatedly to advance the solution one step at a time. It is called a *second-order Runge-Kutta method* (RK2). It is also known as *Heun's method*.

In general, second-order Runge-Kutta formulas are of the form

$$u(t + \Delta t) = u + w_1 \Delta t f + w_2 \Delta t f(u + \beta \Delta t f, t + \alpha \Delta t) + O(\Delta t^3)$$

where w_1, w_2, α and β are parameters at our disposal. The above equation can be rewritten with the aid of the Taylor series in two variables as

$$u(t + \Delta t) = u + w_1 \Delta t f + w_2 \Delta t (f + \alpha \Delta t f_t + \beta \Delta t f f_u) + O(\Delta t^3). \quad (1.4.4)$$

Comparing (1.4.3) and (1.4.4), we see that we should impose the following conditions:

$$w_1 + w_2 = 1, \quad w_2\alpha = \frac{1}{2}, \quad w_2\beta = \frac{1}{2}.$$

One solution is $w_1 = w_2 = 1/2, \alpha = \beta = 1$, which is the one corresponding to Heun's method. The system of equations above has solutions other than this one, such as the one obtained by letting $w_1 = 0, w_2 = 1, \alpha = \beta = 1/2$. The resulting formula is called the *modified Euler method*:

$$\begin{aligned} u(t + \Delta t) &= u(t) + \Delta t K_2 \\ K_1 &= f(u, t), \quad K_2 = f\left(u + \frac{1}{2}\Delta t K_1, t + \frac{1}{2}\Delta t\right). \end{aligned}$$

The general RK2 formulas contain a nonzero parameter $\alpha \in (0, 1]$:

$$u(t + \Delta t) = u(t) + \left(1 - \frac{1}{2\alpha}\right) \Delta t f(u, t) + \frac{1}{2\alpha} \Delta t f\left(u + \alpha \Delta t f(u, t), t + \alpha \Delta t\right). \quad (1.4.5)$$

In the above, we considered the scalar ODE (1.4.1) and its corresponding second-order Runge-Kutta method. In practice, we often have a system of ODEs:

$$\frac{dU}{dt} = F(U, t) \quad (1.4.6)$$

where $U \in \mathbf{R}^N, F \in \mathbf{R}^N$. Similar to the ideas used above, we can derive the second-order Runge-Kutta formulas for the above system. The RK2 formulas are the same as (1.4.5), except that the u and f there be replaced by U and F , respectively.

In practice, it is important to save computer storage so we should try to keep the number of levels required as small as possible. To this end, we write the RK2 formulas in the following form

$$\begin{aligned} U &= U^n \\ G &= F(U, t_n) \\ U &= U + \alpha \Delta t G \\ G &= (-1 + 2\alpha - 2\alpha^2)G + F(U, t_n + \alpha \Delta t) \\ U^{n+1} &= U + \frac{\Delta t}{2\alpha} G, \end{aligned} \quad (1.4.7)$$

where $U^n \approx U(\cdot, t_n)$. Only two levels of storage (U and G) are required for the above algorithm. The choice $\alpha = 1/2$ produces the modified Euler method and $\alpha = 1$ corresponds to the Heun method.

1.4.2. General Runge-Kutta methods. The higher-order Runge-Kutta formulas are very tedious to derive, and we shall not do so. The formulas are rather elegant, however, and are easily programmed once they have been derived. The classical fourth-order Runge-Kutta (RK4) method

is

$$\begin{aligned}
 K_1 &= F(U^n, t_n) \\
 K_2 &= F\left(U^n + \frac{\Delta t}{2}K_1, t_n + \frac{1}{2}\Delta t\right) \\
 K_3 &= F\left(U^n + \frac{\Delta t}{2}K_2, t_n + \frac{1}{2}\Delta t\right) \\
 K_4 &= F(U^n + \Delta t K_3, t_{n+1}) \\
 U^{n+1} &= U^n + \frac{\Delta t}{6}(K_1 + 2K_2 + 2K_3 + K_4). \quad (1.4.8)
 \end{aligned}$$

The above formula required four levels of storage, i.e. K_1, K_2, K_3 and K_4 . An equivalent formulation is:

$$\begin{aligned}
 U &= U^n \\
 G &= U \\
 P &= F(U, t_n) \\
 U &= U + \frac{1}{2}\Delta t P \\
 G &= P \\
 P &= F\left(U, t_n + \frac{1}{2}\Delta t\right) \\
 U &= U + \frac{1}{2}\Delta t(P - G) \\
 G &= \frac{1}{6}G \\
 P &= F\left(U, t_n + \frac{1}{2}\Delta t\right) - \frac{1}{2}P \\
 U &= U + \Delta t P \\
 G &= G - P \\
 P &= F(U, t_{n+1}) + 2P \\
 U^{n+1} &= U + \Delta t\left(G + \frac{1}{6}P\right). \quad (1.4.9)
 \end{aligned}$$

This version of the RK4 method requires only three levels (U, G and P) of storage.

As we saw in the derivation of the Runge-Kutta method of order 2, a number of parameters must be selected. A similar process occurs in establishing higher-order Runge-Kutta methods. Consequently, there is not just one Runge-Kutta method for each order, but a family of methods. As shown in the following table, the number of required *function evaluations* increases more rapidly than the order of the Runge-Kutta methods:

Number of function evaluations	1	2	3	4	5	6	7	8
Maximum order of RK method	1	2	3	4	4	5	6	6

Unfortunately, this makes the higher-order Runge-Kutta methods less attractive than the classical fourth-order method, since they are more expensive to use. Furthermore, all RK methods of a given order have similar stability properties. The stability regions expand as the order increases.

1.4.3. Runge-Kutta methods for autonomous system. The Runge-Kutta procedure for systems of first-order equations is most easily written down in the case when the system is *autonomous*; that is, it has the form

$$\frac{dU}{dt} = F(U).$$

The classical RK4 formulas, in vector form, are

$$U^{n+1} = U^n + \frac{\Delta t}{6} (K_1 + 2K_2 + 2K_3 + K_4), \quad (1.4.10)$$

where

$$\begin{cases} K_1 = F(U^n), \\ K_2 = F\left(U^n + \frac{\Delta t}{2}K_1\right), \\ K_3 = F\left(U^n + \frac{\Delta t}{2}K_2\right), \\ K_4 = F\left(U^n + \Delta tK_3\right). \end{cases}$$

For problems without source terms, we will end up with an autonomous system. The above RK4 method, or its equivalent form similar to (1.4.9), can be used.

If $F(u)$ is *linear*, then the following algorithm can be applied:

$$\begin{aligned} &\text{Set} && U = U^n \\ &\text{For} && k = s, 1, -1 \\ &&& U = U^n + \frac{1}{k} \Delta t F(U) \\ &\text{end} \\ &&& U^{n+1} = U. \end{aligned} \quad (1.4.11)$$

It yields a Runge-Kutta method of order s (for linear problems) and most three levels of storage are required for this algorithm.

1.5. Fast Fourier transform

This section is devoted to the computational aspects of trigonometric interpolation. Suppose that the coefficients c_0, c_1, \dots, c_{N-1} are defined as following:

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ikx_j}, \quad 0 \leq k \leq N-1, \quad (1.5.1)$$

where $x_j = 2\pi j/N$.

In 1965, a paper by Cooley and Tukey [1] described a different method of calculating the coefficients c_k , $0 \leq k \leq N-1$. The method requires only $O(N \log_2 N)$ multiplications and $O(N \log_2 N)$ additions, provided N is chosen in an appropriate manner. For a problem with thousands of data

points, this reduces the number of calculations to thousands compared to millions for the direct technique.

The method described by Cooley and Tukey has become to be known either as the Cooley-Tukey Algorithm or the Fast Fourier Transform (FFT) Algorithm, and has led to a revolution in the use of interpolatory trigonometric polynomials. We follow the exposition of Kincaid and Cheney [2] to introduce the algorithm.

1.5.1. Computational cost. Much of this section will be using complex exponentials. We first recall *Euler's formula*:

$$e^{i\theta} = \cos \theta + i \sin \theta.$$

It is also known that the functions E_k defined by

$$E_k(x) = e^{ikx}, \quad k = 0, \pm 1, \pm 2, \dots$$

form an orthogonal system of functions in the complex space $L_2[-\pi, \pi]$ provided that we define the inner product to be

$$\langle f, g \rangle = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \overline{g(x)} dx.$$

This means that $\langle E_k, E_m \rangle = 0$ when $k \neq m$ and that $\langle E_k, E_k \rangle = 1$. The following two lemmas are important in analyzing the computational cost and in designing the FFT algorithm.

LEMMA 1.1. *Let p and q be exponential polynomials of degree $N - 1$ such that, for the points $y_j = \pi j/N$, we have*

$$p(y_{2j}) = f(y_{2j}), \quad q(y_{2j}) = f(y_{2j+1}), \quad 0 \leq j \leq N - 1. \quad (1.5.2)$$

Then the exponential polynomial of degree $\leq 2N - 1$ that interpolates f at the points $y_j, 0 \leq j \leq 2N - 1$, is given by

$$P(x) = \frac{1}{2} \left(1 + e^{iNx} \right) p(x) + \frac{1}{2} \left(1 - e^{iNx} \right) q(x - \pi/N). \quad (1.5.3)$$

PROOF. Since p and q have degrees $\leq N - 1$, whereas e^{iNx} is of degree N , it is clear that P has degree $\leq 2N - 1$. It remains to show that P interpolates f at the nodes. We have, for $0 \leq j \leq 2N - 1$,

$$P(y_j) = \frac{1}{2} \left(1 + E_N(y_j) \right) p(y_j) + \frac{1}{2} \left(1 - E_N(y_j) \right) q(y_j - \pi/N).$$

Notice that $E_N(y_j) = (-1)^j$. Thus for even j , we infer that $P(y_j) = p(y_j) = f(y_j)$, whereas for odd j , we have

$$P(y_j) = q(y_j - \pi/N) = q(y_{j-1}) = f(y_j).$$

This completes the proof of Lemma 1.1. □

LEMMA 1.2. *Let the coefficients of the polynomials described in Lemma 1.1 be as follows:*

$$p = \sum_{j=0}^{N-1} \alpha_j E_j, \quad q = \sum_{j=0}^{N-1} \beta_j E_j, \quad P = \sum_{j=0}^{2N-1} \gamma_j E_j.$$

Then, for $0 \leq j \leq N-1$,

$$\gamma_j = \frac{1}{2} \alpha_j + \frac{1}{2} e^{-ij\pi/N} \beta_j, \quad (1.5.4)$$

$$\gamma_{j+N} = \frac{1}{2} \alpha_j - \frac{1}{2} e^{-ij\pi/N} \beta_j. \quad (1.5.5)$$

PROOF. To prove (1.5.4) and (1.5.5), we will be using (1.5.3) and will require a formula for $q(x - \pi/N)$:

$$q(x - \pi/N) = \sum_{j=0}^{N-1} \beta_j E_j(x - \pi/N) = \sum_{j=0}^{N-1} \beta_j e^{ij(x - \pi/N)} = \sum_{j=0}^{N-1} \beta_j e^{-i\pi j/N} E_j(x).$$

Thus, from equation (1.5.3),

$$\begin{aligned} P &= \frac{1}{2} \sum_{j=0}^{N-1} \left\{ (1 + E_N) \alpha_j E_j + (1 - E_N) \beta_j e^{-i\pi j/N} E_j \right\} \\ &= \frac{1}{2} \sum_{j=0}^{N-1} \left\{ (\alpha_j + \beta_j e^{-i\pi j/N}) E_j + (\alpha_j - \beta_j e^{-i\pi j/N}) E_{N+j} \right\}. \end{aligned}$$

The formulas for the coefficients γ_j can now be read from this equation. This completes the proof of Lemma 1.2. \square

It follows from (1.5.1), (1.5.2) and (1.5.3) that

$$\begin{aligned} \alpha_j &= \frac{1}{N} \sum_{j=0}^{N-1} f(x_{2j}) e^{-2\pi i j/N}, \\ \beta_j &= \frac{1}{N} \sum_{j=0}^{N-1} f(x_{2j+1}) e^{-2\pi i j/N}, \\ \gamma_j &= \frac{1}{2N} \sum_{j=0}^{2N-1} f(x_j) e^{-\pi i j/N}. \end{aligned}$$

For the further analysis, let $R(N)$ denote the minimum number of multiplications necessary to compute the coefficients in an interpolating exponential polynomial for the set of points $\{2\pi j/N : 0 \leq j \leq N-1\}$.

First, we can show that

$$R(2N) \leq 2R(N) + 2N. \quad (1.5.6)$$

It is seen that $R(2N)$ is the minimum number of multiplications necessary to compute γ_j , and $R(N)$ is the minimum number of multiplications necessary

to compute α_j or β_j . By Lemma 1.2, the coefficients γ_j can be obtained from α_j and γ_j at the cost of $2N$ multiplications. Indeed, we require N multiplications to compute $\frac{1}{2}\alpha_j$ for $0 \leq j \leq N-1$, and another N multiplications to compute $(\frac{1}{2}e^{-ij\pi/N})\beta_j$ for $0 \leq j \leq N-1$. (In the latter, we assume that the factors $\frac{1}{2}e^{-ij\pi/N}$ have already been made available.) Since the coefficients α_j cost $R(N)$ multiplications, and since the same is true for β_j , we obtain a total cost for P of at most $2R(N) + 2N$ multiplications. It follows from (1.5.6) and mathematical induction that $R(2^m) \leq m 2^m$. As a consequence of the above result, we see that if N is a power of 2, say 2^m , then the cost of computing the interpolating exponential polynomial obeys the inequality

$$R(N) \leq N \log_2 N.$$

The algorithm that carries out repeatedly the procedure in Lemma 1.1 is the fast Fourier transform.

1.5.2. Algorithms. Assume $0 \leq n \leq m, 0 \leq k \leq 2^{m-n} - 1$, and $0 \leq j \leq 2^n - 1$. Let

$$P_k^{(n)}(x) = \sum_{j=0}^{2^n-1} A_{kj}^{(n)} E_j(x) = \sum_{j=0}^{2^n-1} A_{kj}^{(n)} e^{ijx}.$$

By Lemma 1.2, the following equations hold:

$$\begin{aligned} A_{kj}^{(n+1)} &= \frac{1}{2} \left[A_{kj}^{(n)} + e^{-ij\pi/2^n} A_{k+2^{m-n-1}, j}^{(n)} \right], \\ A_{k, j+2^n}^{(n+1)} &= \frac{1}{2} \left[A_{kj}^{(n)} - e^{-ij\pi/2^n} A_{k+2^{m-n-1}, j}^{(n)} \right]. \end{aligned}$$

For a fixed n , the array $A^{(n)}$ requires $N = 2^m$ storage locations in memory because $0 \leq k \leq 2^{m-n} - 1$ and $0 \leq j \leq 2^n - 1$. One way to carry out the computations is to use two linear arrays of length N , one to hold $A^{(n)}$ and the other to hold $A^{(n+1)}$. At the next stage, one array will contain $A^{(n+1)}$ and the other $A^{(n+2)}$. Let us call these arrays C and D . The two dimensional array $A^{(n)}$ is stored in C by the rule

$$C(2^n k + j) = A_{kj}^{(n)}, \quad 0 \leq k \leq 2^{m-n} - 1, \quad 0 \leq j \leq 2^n - 1.$$

It is noted that if $0 \leq k, k' \leq 2^{m-n} - 1$ and $0 \leq j, j' \leq 2^n - 1$ satisfying $2^n k + j = 2^n k' + j'$, then $(k, j) = (k', j')$. Similarly, the array $A^{(n+1)}$ is stored in D by the rule

$$D(2^{n+1} k + j) = A_{kj}^{(n+1)}, \quad 0 \leq k \leq 2^{m-n-1} - 1, \quad 0 \leq j \leq 2^{n+1} - 1.$$

The factors $Z(j) = e^{-2\pi ij/N}$ are computed at the beginning and stored. Then we use the fact that $e^{-ij\pi/2^n} = Z(j2^{m-n-1})$.

The following is the standard FFT algorithm.

```
CODE FFT.1
% Cooley-Tukey Algorithm
```

```

Input  m
N=2m,  w=e-2πi/N
for k=0 to N-1 do
    Z(k)=wk,  C(k)=f(2πk/N)
end
for n=0 to m-1 do
    for k=0 to 2m-n-1-1 do
        for j=0 to 2n-1 do
            u=C(2nk+j)
            v=Z(j2m-n-1)*C(2nk+2m-1+j)
            D(2n+1k+j)=0.5*(u+v)
            D(2n+1k+j+2n)=0.5*(u-v)
        end
    end
end
for j=0 to N-1 do
    C(j)=D(j)
end
end
Output C(0), C(1), ..., C(N-1).

```

By scrutinizing the pseudocode, we can also verify the bound $N \log_2 N$ for the number of multiplications involved. Notice that in the nested loop of the code, n takes on m values; then k takes on 2^{m-n-1} values, and j takes on 2^n values. In this part of the code, there is really just one command involving a multiplication – namely, the one in which v is computed. This command will be encountered a number of times equal to the product $m \times 2^{m-n-1} \times 2^n = m2^{m-1}$. At an earlier point in the code, the computation of the Z -array involves $2^m - 1$ multiplications. On any binary computer, a multiplication by $1/2$ need not be counted because it is accomplished by subtracting 1 from the exponent of the floating-point number. Therefore, the total number of multiplications used in CODE FFT.1 is

$$m2^{m-1} + 2^m - 1 \leq m2^m = N \log_2 N.$$

The fast Fourier transform can also be used to evaluate the inverse transform:

$$d_k = \frac{1}{N} \sum_{j=0}^{N-1} g(x_j) e^{ikx_j}, \quad 0 \leq k \leq N-1.$$

Let $j = N - 1 - m$. It is easy to verify that

$$d_k = e^{-ix_k} \frac{1}{N} \sum_{m=0}^{N-1} g(x_{N-1-m}) e^{-ikx_m}, \quad 0 \leq k \leq N-1.$$

Thus, we apply the FFT algorithm to get $e^{ix_k} d_k$. Then extra N operations give d_k . A pseudocode for computing d_k is given below.

CODE FFT.2

```

% Fast Inverse Fourier Transform
Input m
N=2m, w=e-2πi/N
for k=0 to N-1 do
    Z(k)=wk, C(k)=g(2π(N-1-m)/N)
end
for n=0 to m-1 do
    for k=0 to 2m-n-1-1 do
        for j=0 to 2n-1 do
            u=C(2nk+j)
            v=Z(j2m-n-1)*C(2nk+2m-1+j)
            D(2n+1k+j)=0.5*(u+v)
            D(2n+1k+j+2n)=0.5*(u-v)
        end
    end
    for j=0 to N-1 do
        C(j)=D(j)
    end
end
for k=0 to N-1 do
    D(k)=Z(k)*C(k)
end
Output D(0), D(1), ..., D(N-1).

```

The fast Fourier transform can also be used to evaluate the cosine transform:

$$a_k = \sum_{j=0}^N f(x_j) \cos\left(\frac{\pi j k}{N}\right), \quad 0 \leq k \leq N,$$

where $f(x_j)$ are *real numbers*. Let $v_j = f(x_j)$ for $0 \leq j \leq N$ and $v_j = 0$ for $N+1 \leq j \leq 2N-1$. We compute

$$A_k = \frac{1}{2N} \sum_{j=0}^{2N-1} v_j e^{-ikx_j}, \quad x_j = \frac{2\pi j}{2N}, \quad 0 \leq k \leq 2N-1.$$

Since v_j are real numbers and $v_j = 0$ for $j \geq N+1$, it can be shown that the real part of A_k is

$$\operatorname{Re}(A_k) = \frac{1}{2N} \sum_{j=0}^N f(x_j) \cos\left(\frac{\pi j k}{N}\right), \quad 0 \leq k \leq 2N-1.$$

In other words, the following results hold:

$$a_k = 2N \operatorname{Re}(A_k), \quad 0 \leq k \leq N.$$

By the definition of A_k , we know that they can be computed by using the pseudocode FFT.1. When they are multiplied by $2N$, we have the values of a_k .

1.5.3. Numerical examples. To test the efficiency of the FFT algorithm, we compute the coefficients in (1.5.1) using `CODE FFT.1` and the direct method. A subroutine for computing the coefficients directly from the formulas goes as follows:

```

CODE FFT.3
% Direct method for computing the coefficients
Input  m
N=2m,  w=e-2πi/N
for k=0 to N-1 do
    Z(k)=wk,  D(k)=f(2πk/N)
end
for n=0 to N-1 do
    u=D(0)
    for k=1 to N-1
        u=u+D(k)*Z(n)k
    end
    C(n)=u/N
end
Output  C(0), C(1), ..., C(N-1)

```

The computer programmes based on `CODE FFT.1` and `CODE FFT.2` are written in FORTRAN 77 with double precision. We compute the following coefficients:

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} \cos(5x_j) e^{-ikx_j}, \quad 0 \leq k \leq N-1,$$

where $x_j = 2\pi j/N$. The cpu times used are listed in the following table.

m	N	CPU (FFT)	CPU(direct)
9	512	0.02	0.5
10	1024	0.04	2.1
11	2048	0.12	9.0
12	4096	0.28	41.0
13	8192	0.60	180.0

Fourier-Spectral Methods

2.1. Discrete Fourier Transforms and Fourier Derivative Matrices

Most of discussions so far have concentrated on the algebraic polynomial basis functions. Another class of basis functions is the trigonometric functions which are more suitable for representing *periodic* phenomena. For convenience, let us assume that the function being interpolated is periodic with period 2π . One of the basic theorems from Fourier analysis states that if f is 2π -periodic and has a continuous first derivative, then its Fourier series

$$f(x) \sim \sum_{k=-\infty}^{\infty} \hat{f}(k)e^{ikx} \quad (2.1.1)$$

where $i^2 = -1$ and

$$\hat{f}(k) = \frac{1}{2\pi} \int_0^{2\pi} f(t)e^{-ikt} dt$$

converges uniformly to f .

In this section, we will study some basic properties of the Fourier series. This will be useful for deriving the Fast Fourier Transform algorithm to be considered in Sect. 1.5. Then we will derive the differentiation matrices for the Fourier pseudospectral methods. We will also investigate the spectral radius and conditions numbers for the differentiation matrices.

2.1.1. Exponential polynomials. Much of this section will be using complex exponentials. We first recall *Euler's formula*:

$$e^{i\theta} = \cos \theta + i \sin \theta.$$

It is also known that the functions E_k defined by

$$E_k(x) = e^{ikx}, \quad k = 0, \pm 1, \pm 2, \dots$$

form an orthogonal system of functions in the complex space $L_2[0, 2\pi]$ provided that we define the inner product to be

$$\langle f, g \rangle = \frac{1}{2\pi} \int_0^{2\pi} f(x)\overline{g(x)} dx.$$

This means that $\langle E_k, E_m \rangle = 0$ when $k \neq m$ and that $\langle E_k, E_k \rangle = 1$. For discrete values, it will be convenient to use the following inner-product notation:

$$\langle f, g \rangle_N = \frac{1}{N} \sum_{j=0}^{N-1} f\left(\frac{2\pi j}{N}\right) \overline{g\left(\frac{2\pi j}{N}\right)}. \quad (2.1.2)$$

This function is not a true inner product because the condition $\langle f, f \rangle_N = 0$ does not imply that $f \equiv 0$. It implies that $f(x)$ takes the value 0 at each node $2\pi j/N$.

LEMMA 2.1. *For any $N \geq 1$, we have*

$$\langle E_k, E_m \rangle_N = \begin{cases} 1 & \text{if } k - m \text{ is divisible by } N \\ 0 & \text{otherwise} \end{cases} \quad (2.1.3)$$

PROOF. The above result can be obtained by the following observations: If $k - m$ is not divisible by N , then

$$\begin{aligned} \langle E_k, E_m \rangle_N &= \frac{1}{N} \sum_{j=0}^{N-1} E_k\left(\frac{2\pi j}{N}\right) \overline{E_m\left(\frac{2\pi j}{N}\right)} \\ &= \frac{1}{N} \sum_{j=0}^{N-1} \left[e^{2\pi i(k-m)/N} \right]^j \\ &= \frac{1}{N} \frac{e^{2\pi i(k-m)} - 1}{e^{2\pi i(k-m)/N} - 1} = 0; \end{aligned}$$

If $k - m$ is divisible by N , the second line above equals to 1. \square

An *exponential polynomial* of degree at most n is any function of the form

$$p(x) = \sum_{k=0}^n d_k e^{ikx} = \sum_{k=0}^n d_k E_k(x) = \sum_{k=0}^n d_k (e^{ix})^k.$$

The last expression in this equation explains the source of the terminology because it shows p to be a polynomial of degree $\leq n$ in the variable e^{ix} .

LEMMA 2.2. *The exponential polynomial that interpolates a prescribed function f at $x_j = 2\pi j/N$, $0 \leq j \leq N - 1$, is given by*

$$P(x) = \sum_{k=0}^{N-1} c_k E_k(x), \quad (2.1.4)$$

$$\text{with } c_k = \langle f, E_k \rangle_N. \quad (2.1.5)$$

In other words, it can be shown that $P(x_j) = f(x_j)$, $0 \leq j \leq N - 1$.

PROOF. The above result can be obtained by the direct calculations: for $0 \leq m \leq N - 1$,

$$\begin{aligned}
P(x_m) &= \sum_{k=0}^{N-1} c_k E_k(x_m) \\
&= \sum_{k=0}^{N-1} \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) \overline{E_k(x_j)} E_k(x_m) \\
&= \sum_{j=0}^{N-1} f(x_j) \langle E_m, E_j \rangle_N \\
&= f(x_m),
\end{aligned}$$

where in the last step we have used (2.1.3) and the fact $0 \leq |m - j| < N$. \square

2.1.2. Fourier series and differentiation. As mentioned earlier, if f is 2π -periodic and has a continuous first derivative then its Fourier series converges uniformly to f . In application, we truncate the infinite series in (2.1.1) to the following finite series:

$$F(x) = \sum_{k=-N/2}^{N/2-1} \alpha_k e^{ikx}. \quad (2.1.6)$$

Assume that $F(x_j) = f(x_j)$, where $x_j = 2\pi j/N$, $0 \leq j \leq N - 1$. This fact, together with the transformation $k \rightarrow k' - N/2$, yield

$$f(x_j) = \sum_{k'=0}^{N-1} \alpha_{k'-N/2} (-1)^j e^{ik'x_j}, \quad 0 \leq j \leq N - 1.$$

An application of Lemma 2.2 gives

$$\alpha_{k'-N/2} = \frac{1}{N} \sum_{j=0}^{N-1} (-1)^j f(x_j) e^{-ik'x_j}, \quad k' = 0, 1, \dots, N - 1. \quad (2.1.7)$$

Making the transformation $k' - N/2 \rightarrow k$ for the above equation leads to

$$\alpha_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ikx_j} \quad k = -N/2, \dots, N/2. \quad (2.1.8)$$

We now differentiate the truncated Fourier series $F(x)$ termwise to get the approximate derivatives. It follows from (2.1.6) that

$$F^{(m)}(x) = \sum_{k=-N/2}^{N/2} \alpha_k (ik)^m e^{ikx},$$

where m is a positive integer. We can write $F^{(m)}(x)$ in the following equivalent form:

$$F^{(m)}(x) = \sum_{k'=0}^{N-1} \alpha_{k'-N/2} \left(i(k' - N/2) \right)^m e^{i(k'-N/2)x}. \quad (2.1.9)$$

Using (2.1.7) and (2.1.9), we obtain

$$\begin{aligned} \begin{pmatrix} F^{(m)}(x_0) \\ F^{(m)}(x_1) \\ \vdots \\ F^{(m)}(x_{N-1}) \end{pmatrix} &= \left((-1)^j e^{ikx_j} (i(k - N/2))^m \right)_{j,k=0}^{N-1} \begin{pmatrix} \alpha_{-N/2} \\ \alpha_{-N/2+1} \\ \vdots \\ \alpha_{N/2-1} \end{pmatrix} \\ &= \frac{1}{N} \left((-1)^j e^{ikx_j} (i(k - N/2))^m \right)_{j,k=0}^{N-1} \left((-1)^k e^{-ijx_k} \right)_{j,k=0}^{N-1} \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{N-1}) \end{pmatrix}. \end{aligned}$$

This indicates that the m -th order differentiation matrix associated with Fourier PS methods is given by

$$D^m = \frac{1}{N} \left((-1)^j e^{ikx_j} (i(k - N/2))^m \right)_{j,k=0}^{N-1} \left((-1)^k e^{-ijx_k} \right)_{j,k=0}^{N-1} \quad (2.1.10)$$

A pseudocode for computing D^m is given below.

```

CODE FPS.1
function Dm=FDMx(m,N)
for j=0 to N-1 do
    x(j)=2*pi*j/N
end
for j=0:N-1
    for k=0:N-1
        A(j,k)=(-1)^j*exp(i*k*x(j))*(i*(k-N/2))^m
        B(j,k)=(-1)^k*exp(-i*j*x(k))
    end
end
end
Dm=(1/N)*A*B

```

To test the above code, we consider a simple example. Let $f(x) = 1/(2 + \sin x)$ and $x_j = 2\pi j/N$. Let $\mathbf{F} = (f(x_0), f(x_1), \dots, f(x_{N-1}))^T$. The matrices $D1 = \text{FDMx}(1, N)$ and $D2 = \text{FDMx}(2, N)$ are given by CODE FPS.1.

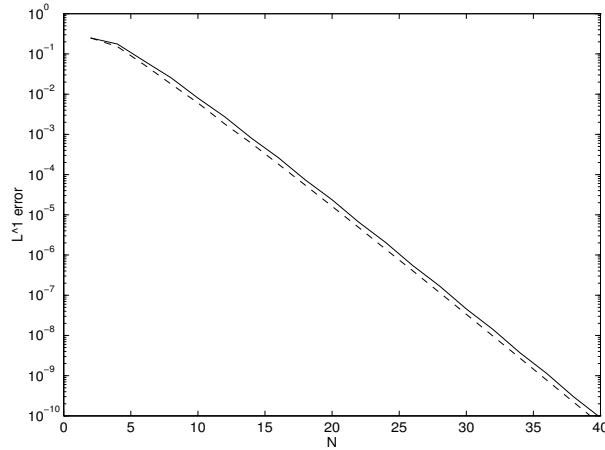


FIGURE 1. $f(x) = 1/(2 + \sin x)$. The solid line is for `err1`, and the dashed line is for `err2`.

We plot the following L^1 errors:

$$\mathbf{err1} = \frac{1}{N} \sum_{j=0}^{N-1} |(D1 * \mathbf{F})_j - f'(x_j)|,$$

$$\mathbf{err2} = \frac{1}{N} \sum_{j=0}^{N-1} |(D2 * \mathbf{F})_j - f''(x_j)|.$$

It can be seen from Fig. 1 that the above L^1 errors decay to zero very rapidly.

It should be pointed out that the convergence holds only for periodic functions. If we change the $f(x)$ above to a non-periodic function, say $f(x) = x^2$, then the errors `err1` and `err2` defined above will diverge to infinity as N becomes large.

Apart from periodic functions, the Fourier spectral methods can also be used to approximate functions which decays exponentially fast at infinities, for instance, solutions of KDV-type equations. For illustration purposes, we consider $f(x) = e^{-2(x-\pi)^2}$. In Fig. 2, we plot `err1` and `err2` for this function. It is noted that the errors will not decrease after a critical value of N , but the error for large N will be of the same magnitudes of $f'(x)$ and $f''(x)$ away from $[0, 2\pi]$.

2.1.3. Spectral radius for advection operator. We consider here the Fourier-pseudospectral approximation of the advection operator

$$Lu = \frac{du}{dx} \quad (2.1.11)$$

with collocation points $x_j = 2\pi j/N$, $j = 0, 1, \dots, N-1$. To remove the zero eigenvalue, we can fix $u(0) = 0$. It is easy to show that we will end up

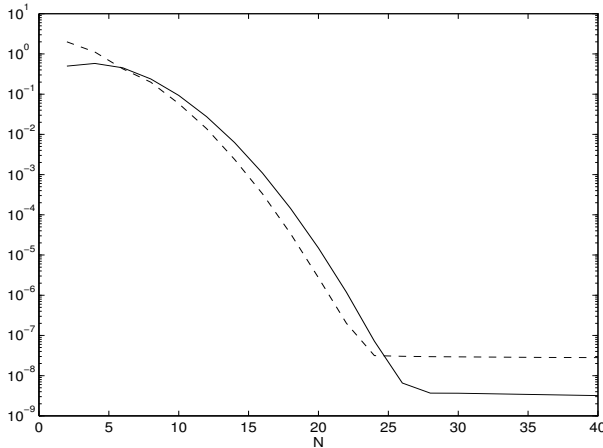


FIGURE 2. $f(x) = e^{-2(x-\pi)^2}$. The solid line is for `err1`, and the dashed line is for `err2`.

with the following eigenvalue problem:

$$AV = \lambda V,$$

$$(A)_{ij} = D1(i, j), \quad 1 \leq i, j \leq N - 1,$$

where $D1 = \text{FDMx}(1, N)$ is given by `CODE FPS.1`. It is an $(N - 1) \times (N - 1)$ matrix. It is noted that the first row $D1(0, j)$ and the first column $D1(i, 0)$ are removed from $D1$ due to the zero boundary conditions. Thus, the spectral radius and condition number for the convection operator can be easily obtained. It is seen from Fig. 2 that the spectral radius $\rho(A) = O(N)$ and the condition number $\kappa(A)$ is also of the order $O(N)$. In fact, numerical results suggest that the smallest $|\lambda|$ equals to 0.5 for all values of N .

Spectral radius for diffusion operator

We only consider the Dirichlet boundary conditions. The results obtained below for spectral radius and condition number are similar to those of Neumann boundary condition. For the diffusion operator

$$Lu = -\frac{d^2u}{dx^2} \tag{2.1.12}$$

the eigenvalues of collocation operator associated with Fourier PS methods are defined by the set of equations

$$\frac{d^2U(x_j)}{dx^2} = \lambda U(x_j), \quad 1 \leq j \leq N - 1$$

$$U(x_0) = 0,$$

where $x_j = 2\pi j/N$. It is noted that the boundary condition $U(x_N) = 0$ is imposed implicitly, due to the periodic assumption. Now we have the

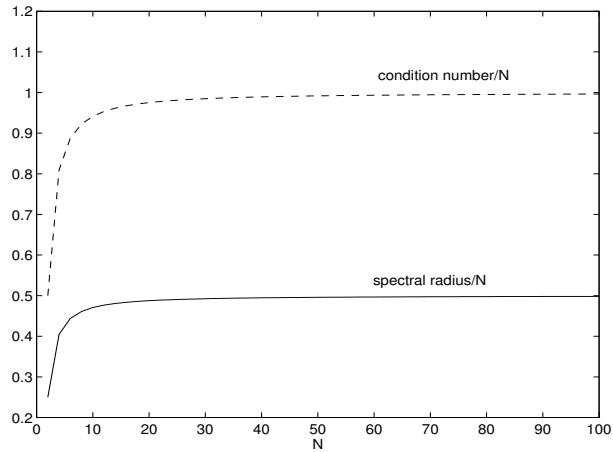


FIGURE 3. Spectral radius and condition number associated with convection operator.

following eigenvalue problem

$$AV = \lambda V,$$

$$(A)_{ij} = D2(i, j), \quad 1 \leq i, j \leq N - 1,$$

where $D2 = \text{FDMx}(2, N)$ is given by CODE FPS.1. The condition number and spectral radius again N are plotted in Fig. 4. Numerical results show that

$$0.25 \leq -\lambda \leq 0.25N^2, \quad \text{for large } N.$$

This suggests that $\rho(A) = O(N^2)$ and $\kappa(A) = O(N^2)$.

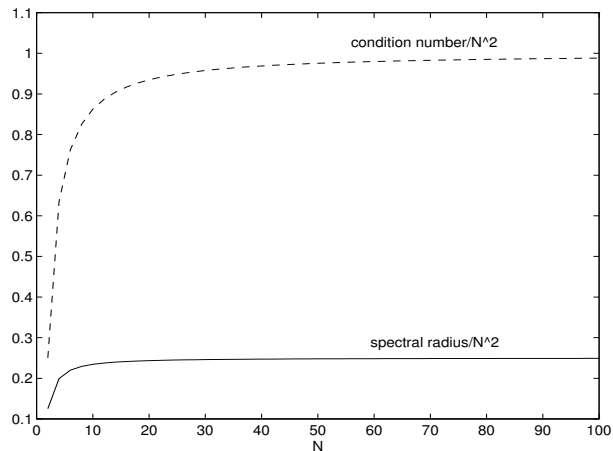


FIGURE 4. Spectral radius and condition number associated with the convection operator.

For the sake of comparison, we have for large values of N ,

- spectral radius of Fourier PS methods
- $\approx 0.4 \cdot$ spectral radius of finite difference method,
- condition number of Chebyshev PS methods
- $\approx 2.5 \cdot$ condition number of finite difference method.

The above results suggest that the Fourier spectral methods have similar asymptotic properties for the spectral radius and the condition number to the finite difference methods. In this regard, the Fourier basis function is almost the best among the standard basis functions. However, the main restriction of the Fourier spectral methods is that it works well only for function which are periodic or are of compact supports.

Bibliography

- [1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp*, 19:297–301, 1965.
- [2] D. Kincaid and E. W. Cheney. *Numerical Analysis, Mathematics of Scientific Computing*. Brooks/Cole, 2nd edition, 1996.
- [3] R. J. LeVeque. *Numerical Methods for Conservation Laws*. Birkhauser, Basel, 2nd edition, 1992.
- [4] R. D. Richtmyer and K. W. Morton. *Difference Methods for Initial-Value Problems*. Interscience, New York, 2nd edition, 1967.