FOURFUN: A NEW SYSTEM FOR AUTOMATIC COMPUTATIONS USING FOURIER EXPANSIONS[†]

AUTHOR: KRISTYN N. MCLEOD[‡] AND ADVISOR: RODRIGO B. PLATTE[‡]

Abstract. Using object-oriented programming in MATLAB, a collection of functions, named Fourfun, has been created to allow quick and accurate approximations of periodic functions with Fourier expansions. To increase efficiency and reduce the number of computations of the Fourier transform, Fourfun automatically determines the number of nodes necessary for representations that are accurate to close to machine precision. Common MATLAB functions have been overloaded to keep the syntax of the Fourfun class as consistent as possible with the general MATLAB syntax. We show that the system can be used to efficiently solve differential equations. Comparisons with Chebfun, a similar system based on Chebyshev polynomial approximations, are provided.

1. Introduction

Mathematicians often have to make a choice between computing solutions symbolically through the manipulation of algebraic expressions or numerically with approximations [12]. While symbolic solutions are often preferred, they may not always be realistic to compute. For some problems, the memory required may be prohibitive while for others a symbolic solution may not even exist. In these cases, numerical computations must be used which introduces errors due to truncation and floating point arithmetic. In addition to this loss of accuracy, numerical computations often require different algorithms for standard operations. Even for a simple example such as taking the first derivative of $\sin(x)$ an algorithm for approximation must be decided on and implemented. Consequently, error analysis is essential to ensure a useful result. L.N. Trefethen and other researchers at the University of Oxford have adopted a research goal to simplify the process of numerical computing by creating a system known as Chebfun that allows a user to think symbolically while computing numerically [1, 11, 12, 14].

The name Chebfun comes from the primary reliance of the system on Chebyshev expansions to approximate functions. The system stores function values at Chebyshev points, which are defined as

$$x_j = \cos\frac{j\pi}{n}, \qquad 0 \le j \le n,$$

for computations. When it is advantageous, an FFT is used to approximate the coefficients in the Chebyshev series representation [14],

$$f(x) = {\sum_{j=0}^{\infty}}' a_j T_j(x), \quad T_j(x) = \cos(j\cos^{-1}x).$$

Function values at points other than x_j are found using a barycentric interpolation formula. Many familiar MATLAB operators have been overloaded to allow for the manipulation of the approximations which are referred to as chebfuns. Technical implementation of these ideas has created a system that allows the user to construct a chebfun that automatically determines how many coefficients should be used for acceptable accuracy, i.e. that takes care of the numerical complications for them. The following code shows an example of how to construct a chebfun of the function $f = x^3$.

```
>> f = chebfun(@(x) x.^3)
f =
   chebfun column (1 smooth piece)
       interval       length   endpoint values
[      -1,       1]        4       -1        1
```

[†] This work was supported in part by NSF-DMS-MCTP 1148771.

[‡] School of Mathematical and Statistical Sciences, Arizona State University; email addresses: knmcleod@asu.edu and rbp@asu.edu.

```
vertical scale =   1
```

Chebfun used four coefficients which is indicated by the length column. After construction, the user can apply common MATLAB operators to the chebfun and the result will be another chebfun. We can also apply many of these operators to two chebfuns and again end up with a chebfun.

```
>> g = chebfun(@(x) sin(x));
>> a = f.*g
a=
   chebfun column (1 smooth piece)
       interval       length   endpoint values
[      -1,       1]       17     0.84     0.84
vertical scale = 0.84
```

Ultimately, the Chebfun system has a wide range of potential applications because it allows users to focus on the problem they are applying it to rather than on the numerical accuracy of the method they are using.

Inspired by the Chebfun system, the Fourfun system borrows many of its fundamental ideas. Fourfun automatically determines the number of nodes needed to approximate a periodic function on a bounded domain with a Fourier expansion. Unlike the Chebyshev polynomials, the Fourier expansion depends on equally spaced points. An immediate consequence of this spacing is that the FFT can be used to quickly compute the Fourier coefficients. A second less direct consequence arises in applications of the Fourfun system to partial differential equations.

The basic method for using Chebfun and Fourfun to solve differential equations is to create chebfuns or fourfuns[1] for the equations defining the spatial dimension. Both Fourfun and Chebfun have overloaded methods that make the spatial derivatives simple to compute. These chebfuns or fourfuns can then be used with standard time-stepping schemes to compute the time derivatives. The time step often needs to be determined experimentally, but for some partial differential equations we can look to the Courant-Friedrichs-Lewy (CFL) condition as a guide. For the solutions to the transport and wave equations presented in this paper, we want $|c| \leq 1$ where $c = \frac{v\Delta t}{\Delta x}$ and v is the wave speed [5]. Consequently, the space between the sampling points, Δx, for both the Chebfun and Fourfun systems must be considered when determining an appropriate time step. Because the Chebfun system has nonuniformly spaced points that are concentrated close to the ends of the interval, Δt must be chosen smaller than the closest points. For Fourfun however, the uniform nodes lead to an improved CFL condition and a larger possible time step.

At this point, the development of the Chebfun system far exceeds that of the Fourfun system. However, Fourfun may be the better alternative for maintaining this "symbolic"-like feeling while solving PDEs with periodic boundary conditions numerically. This paper will begin with a discussion of the theoretical concepts used to construct the Fourfun system. The second section will address the format of the system and a description of the implementation. The code itself is available to download through MATLAB Central File Exchange: `http://www.mathworks.com/matlabcentral/fileexchange/46999-fourfun`. Basic examples of syntax and solutions to differential equations are included in the third section along with a comparison of speed between Fourfun and Chebfun.

2. Fourier expansions

The following section briefly explains the major mathematical concepts and formulas used in Fourfun starting with the Fourier series, a discussion of discretization, and some of the computational implications. An explanation of the related formula for barycentric trigonometric representation is included and preferred by the Fourfun system for certain computations.

2.1. **Fourier Series.**

The main method used for approximations in Fourfun is the Fourier series. To begin, we consider a periodic function of period P where $f(x) = f(x + P)$. Throughout this discussion of Fourier series and the Fourfun system, the default period will be assumed to be 2π. From here, we can define the Fourier series as an infinite sum of the trigonometric functions sine and cosine [3, 4, 8]:

$$f(x) = c_0 + \sum_{n=1}^{\infty} (a_n \cos nx + b_n \sin nx) \tag{1}$$

[1]Capitalization of Fourfun is used very specifically. "Fourfun" refers to the system as a whole or more specifically the Fourfun class, while "fourfun" is an object of the Fourfun class. The same convention is used for Chebfun.

with the coefficients a_n, b_n and c_0 defined as :

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos nx\,dx$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin nx\,dx$$

$$c_0 = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\,dx.$$

It is easy to see from (1) that a trigonometric polynomial is the result. Manipulation of this formula with Euler's formula [13],

$$e^{\pm i\theta} = cos\theta \pm i\sin\theta, \tag{2}$$

and the resulting identities for cosine and sine,

$$\cos\theta = \frac{e^{i\theta}+e^{-i\theta}}{2}, \tag{3}$$

$$\sin\theta = \frac{e^{i\theta}-e^{-i\theta}}{2i}, \tag{4}$$

leads to an equivalent form. Substituting these identities into (1) and simplifying results in the complex form of the Fourier series [13],

$$f(x) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{ikx}, \tag{5}$$

where the coefficients are defined as

$$\hat{f}_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)e^{-ikx}\,dx, \quad k\in\mathbb{Z}. \tag{6}$$

Note that when $k = 0$, this equation simply becomes

$$\hat{f}_0 = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\,dx, \tag{7}$$

which is the average value of the function on the interval $[-\pi,\pi]$. Equation (6) is known as the Fourier transform.

One of the main features of Fourier expansions is their fast convergence. The fast decay of the Fourier coefficients $\hat{f}_k$ for smooth functions is evident if we apply integration by parts multiple times to (6). To this end, assume that $f\in C^{(n)}$ and that $f^{(k)}(-\pi) = f^{(k)}(\pi)$, $k = 0,\ldots,n-1$. In this case,

$$\hat{f}_k = \frac{1}{2\pi}\frac{1}{(ik)^n}\int_{-\pi}^{\pi} f^{(n)}(x)e^{-ikx}\,dx,$$

and

$$\left|\hat{f}_k\right| \le C\frac{1}{|k|^n}, \text{ where } C = \max|f^{(n)}(x)|, \ \ x\in[-\pi,\pi]. \tag{8}$$

We point out that sharper bounds can be derived with additional assumptions (see [8, 15] for details).

2.2. **The Discrete Case.** While using the continuous Fourier transform would be ideal, for the Fourfun system and more generally for numerical computations, we will be working with discrete data points. Consequently, to effectively apply the Fourier series and transform the discretized forms of these formulas must be used.

To do this, consider an interval $[-\pi, \pi]$ with points $x_0 = -\pi$ to $x_N = \pi$ with spacing $\Delta x = \frac{2\pi}{N}$. Knowing $f(x)$ for each of the x values and calling $g(x) = f(x)e^{-ikx}$ gives an integral which can be approximated with the familiar trapezoidal rule [4, 8],

$$\frac{1}{2\pi}\int_{-\pi}^{\pi} g(x)\,dx \approx \left(\frac{1}{2\pi}\right)\frac{\Delta x}{2}\Big[g(-\pi) + 2\sum_{j=1}^{N-1} g(x_j) + g(\pi)\Big].$$

As a condition of being periodic, assume that $g(x_0) = g(x_N)$. From here, the discrete Fourier transform can be written [16]

$$\tilde{f}_k = \frac{\Delta x}{2\pi}\sum_{j=0}^{N-1} f(x_j)e^{-ikx_j} = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j)e^{-ikx_j}, \quad k = -\frac{N}{2}, \ldots, \frac{N}{2}. \tag{9}$$

The discrete Fourier sum is then given by

$$f(x_j) = {\sum_{k=-\frac{N}{2}}^{\frac{N}{2}}}' \tilde{f}_k e^{ikx_j}, \quad x_j = -\pi + \frac{2\pi j}{N}, \quad j = 0, \ldots, N-1, \tag{10}$$

where the coefficients are described by (9) and the prime indicates that $k = \pm N/2$ are multiplied by $1/2$ [4, 16]. Conveniently, we can use the fast Fourier transform to quickly calculate these Fourier coefficients.

A well-known relationship between (6) and (9) is given by the aliasing formula [8, 15],

$$\tilde{f}_k = \hat{f}_k + \sum_{j\neq 0} \hat{f}_{k+jN}, \quad k = -\frac{N}{2}, \ldots, \frac{N}{2}. \tag{11}$$

Using the bound established in (8), we have that

$$\left|\sum_{j\neq 0} \hat{f}_{k+jN}\right| \leq C\sum_{j\neq 0}\frac{1}{|k+jN|^n} = \frac{C}{N^n}\sum_{j\neq 0}\frac{1}{|\frac{k}{N}+j|^n} = C\frac{C_2}{N^n}, \tag{12}$$

where $C_2 = 2\sum_{j=1}^{\infty}\frac{1}{|\frac{1}{2}-j|^n}$, which is bounded for $n > 1$. Combining (11) and (12), the fast convergence of the Fourier series is evident in the discrete case as well.

2.3. **Fast Fourier Transform.** The discrete Fourier transform can be expensive to compute requiring $\mathcal{O}(N^2)$ operations where an operation is considered a complex multiplication followed by a complex addition [6]. However, by taking advantage of the number of nodes used, the number of operations can be reduced to $\mathcal{O}(N\log_2 N)$. This optimized algorithm is known as the fast Fourier transform or the FFT. When the number of nodes, N, is a power of two the algorithm is most efficient.

When N is even, (9) can be divided into two summations [8, 10],

$$\tilde{f}_k = \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j})e^{-ikx_{2j}} + \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j+1})e^{-ikx_{2j+1}}.$$

It is easy to see that we have one summation that evaluates the even x_{2j} and one that evaluates the odd x_{2j+1}. Keeping in mind that $x_j = \frac{2\pi j}{N}$ and letting $\omega_N = e^{\frac{-2\pi i}{N}}$ we can simplify the formula to [10]

$$\tilde{f}_k = \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j})\omega_N^{2jk} + \omega_N^k \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j+1})\omega_N^{2jk}. \tag{13}$$

This results in only $\frac{N}{2}$ $\tilde{f}_k$'s so we must also look at $\tilde{f}_{k+N/2}$ for the remaining coefficients,

$$\tilde{f}_{k+N/2} = \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j}) e^{-i(k+N/2)x_{2j}} + \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j+1}) e^{-i(k+N/2)x_{2j+1}}.$$

Following the method from above we can write this in an analogous notation to (13),

$$\tilde{f}_{k+N/2} = \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j}) \omega_N^{2jk} - \omega_N^k \sum_{j=0}^{\frac{N}{2}-1} f(x_{2j+1}) \omega_N^{2jk}.$$

Consequenlty, we compute two different FFT's with $\frac{N}{2}$ terms each, rather than one FFT with N terms. This is summarized succinctly in the algorithm borrowed from [10].

```
If n = length(y) is even,
        omega = exp(-2*pi*i/n);
        k = (0:n/2-1)';
        w = omega .^ k;
        u = fft(y(1:2:n-1));
        v = w.*fft(y(2:2:n));
then
        fft(y) = [u+v; u-v];
```

If N is not only even but also a power of two, we can continue to divide the summations until we end up with N FFT's of length one. These N FFT's can then be concatenated to give the FFT of original length. The Fourfun system uses the FFT with N even to compute Fourier coefficients. Whenever possible, Fourfun utilizes the added advantage of evaluating the FFT with N as a power of two.

2.4. **Barycentric Formula.** The DFT and FFT that have been discussed rely on discrete function values at equally spaced points along the domain. That is, the exact value of the function is known at the sampling points. In order for an approximation to be useful, we need to be able to calculate the value of the approximated function at any point in the domain. This could be accomplished by evaluating the trigonometric polynomial that the Fourier series produces using the FFT to calculate Fourier coefficients. However, a method known as barycentric representation offers an alternative [9]:

$$t(\tau) = \frac{\sum_{k=0}^{n-1} (-1)^k f_k \cot[\pi(\tau - \tau_k)]}{\sum_{k=0}^{n-1} (-1)^k \cot[\pi(\tau - \tau_k)]}$$

if n is even and by

$$t(\tau) = \frac{\sum_{k=0}^{n-1} (-1)^k f_k \csc[\pi(\tau - \tau_k)]}{\sum_{k=0}^{n-1} (-1)^k \csc[\pi(\tau - \tau_k)]}$$

if n is odd.

Note that the Fourier coefficients are not used in either the odd or the even case. Although, at times it is advantageous or necessary to calculate the Fourier coefficients, these formulas provide a option that does not rely on them. Moreover, they are numerically stable even for values that are very close to the sampling points, τ_k [9]. In other words, even where $(\tau - \tau_k)$ is very small an approximation can be made.

2.5. **Differentiation and Integration of Fourier Series.** Keeping in mind the goal of applying the Fourfun system to differential equations, the abilities to differentiate and integrate are essential. Let f be the function to be approximated by a fourfun. Supposing $f \in C^{(n)}$, that its derivative $f^{(n)}$ is piecewise smooth, and $f^{(k)}(-\pi) = f^{(k)}(\pi)$ for $k = 0, \ldots, n$, than the Fourier series can be differentiated term by term and will result in the Fourier series of $f^{(n)}$ [7, 8]. This is described in the discrete case by the equation

$$f^{(n)}(x) \approx \sum_{k=\frac{-N}{2}}^{\frac{N}{2}} {}'(ik)^n \tilde{f}_k e^{ikx}, \quad x \in [-\pi, \pi], \tag{14}$$

where the prime indicates that the $k = \pm\frac{N}{2}$ terms are multiplied by 1/2.

It is important to note that the derivative is not as accurate as the original function. Equation (5) suggests that we can write the exact representation of the original function as

$$h(x) = \sum_{k=-N/2}^{N/2} \tilde{f}_k e^{ikx} + \sum_{|k|>N/2} \tilde{f}_k e^{ikx}. \tag{15}$$

The second term can be viewed as the error due to truncating the summation at $\pm N/2$. Differentiating (15) term by term q times gives

$$h^{(q)}(x) = \sum_{k=-N/2}^{N/2} (ik)^q \tilde{f}_k e^{ikx} + \sum_{|k|>N/2} (ik)^q \tilde{f}_k e^{ikx}. \tag{16}$$

The second term of (16) again represents the truncation error. Keeping in mind that k is an integer, we can see that the error of the derivative is greater than the error of the original function. Moreover, we can look at the greatest error that might be contributed by a derivative using the triangular inequality,

$$\left| \sum_{|k|>N/2} (ik)^q \tilde{f}_k e^{ikx} \right| \leq \sum_{|k|>N/2} k^q \left| \tilde{f}_k \right| = \mathcal{O}\left(\frac{1}{N^{n-q+1}} \right), \quad n > q, \tag{17}$$

where the last estimate follows from (8), (11), and (12). This inequality serves as a warning that the more derivatives that are taken, the more significant the error becomes.

Integrating a Fourier series is a similar process. The Fourier series of a piecewise smooth function, f, can always be integrated term by term and the result will be a convergent series that always converges to the integral of f when $x \in [-\pi, \pi]$ [7]. However, there is a complication with integration that is not seen with differentiation. This convergent series is not necessarily a Fourier series [7]. To apply this to the Fourfun system, the Fourier coefficients are examined to determine if the indefinite integral of f can be represented as Fourier series. Recall (7)

$$\tilde{f}_0 = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)\, dx.$$

Should this equation, i.e. the average value of the function on the specified interval, evaluate to zero then the indefinite integral can be represented by a Fourier series that is determined by term by term integration. The discrete case is described by

$$\int f(x)\, dx \approx {\sum_{k=\frac{-N}{2}}^{\frac{N}{2}}}' \frac{\tilde{f}_k e^{ikx}}{ik}, \quad k \neq 0, \quad x \in [-\pi, \pi], \tag{18}$$

where again the prime indicates that the $k = \pm\frac{N}{2}$ terms are multiplied by $1/2$.

The error introduced when integrating should be discussed as well. Consider once again (15). Integrating term by term gives the indefinite integral

$$H(x) = \sum_{k=\frac{-N}{2}}^{\frac{N}{2}} \frac{\tilde{f}_k e^{ikx}}{ik} + \sum_{|k|>\frac{N}{2}} \frac{\tilde{f}_k e^{ikx}}{ik}, \quad k \neq 0, \quad x \in [-\pi, \pi].$$

Unlike with the derivative, we can find an upper bound to this error. If n integrals are computed, the upper bound of the error is

$$\left| \sum_{|k|>N/2} \frac{\tilde{f}_k e^{ikx}}{(ik)^n} \right| \leq \sum_{|k|>N/2} \frac{1}{k^n} \left| \tilde{f}_k \right| \leq \frac{1}{(\frac{N}{2})^n} \sum_{|k|>N/2} \left| \tilde{f}_k \right|.$$

This division by $(N/2)^n$ shows that taking the indefinite integral of $h(x)$ gives an approximation for $H(x)$ that is more accurate than our approximation for $h(x)$.

2.6. **Root-Finding.** Finding the roots of a function represented as a Fourier series can be expressed as an eigenvalue problem [2]. To begin, we need to represent a Fourier series as a polynomial. Consider (5) expanded,

$$f(x) = \frac{1}{2}\tilde{f}_{-N/2}e^{i(-N/2)x} + \tilde{f}_{-N/2+1}e^{i(-N/2+1)x} + \cdots + \tilde{f}_0 + \tilde{f}_1 e^{ix} + \cdots + \frac{1}{2}\tilde{f}_{N/2}e^{i(N/2)x}. \tag{19}$$

Let $z = e^{ix}$ and factor out a z from each term in order to write (19) in a more familiar polynomial form,

$$f(z) = z^{-N/2}\Big[\frac{1}{2}\tilde{f}_{-N/2} + \tilde{f}_{-N/2+1}z + \tilde{f}_{-N/2+2}z^2 + \cdots + \frac{1}{2}\tilde{f}_{N/2}z^N\Big]. \tag{20}$$

We know that $z^{-N/2} = e^{-i(N/2)x}$ will never equal zero. Consequently, the zeros of $f(z)$ can be found by finding the zeros of the polynomial in the brackets. Moreover, if we divide $f(z)$ by the largest coefficient to make it monic, $f(z)$ can be represented as

$$f(z) = (-1)^{(N/2)}\begin{vmatrix} -z & & & & & -a_0 \\ 1 & -z & & & & -a_1 \\ & 1 & -z & & & -a_2 \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & -z & -a_{N/2-2} \\ & & & & 1 & (-z - a_{N/2-1}) \end{vmatrix},$$

where a_i for $i = 0, \ldots, N/2-1$ are the new coefficients [15]. It is easy to see that the zeros of the polynomial represented by the above determinant are the eigenvalues of matrix A, the companion matrix of f [15],

$$A = \begin{bmatrix} 0 & & & & & -a_0 \\ 1 & 0 & & & & -a_1 \\ & 1 & 0 & & & -a_2 \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & 0 & -a_{N/2-2} \\ & & & & 1 & -a_{N/2-1} \end{bmatrix}.$$

These eigenvalues will give the zeros of $f(z)$. To find the x values of the zeros,$z = e^{ix}$ is solved for x.

3. The Fourfun Class

The Fourfun class implements the mathematical ideas from above to create a class in MATLAB that allows for easy interpolation. This allows numeric computations to feel as if they are done with continuous periodic functions rather than discrete data points. The Fourfun system overloads functions from MATLAB in order to keep the syntax straight-forward.

3.1. **Fourfun objects.** The Fourfun system creates objects, referred to as fourfuns, using one of the two constructor options which will be explained in more detail later. They can be constructed using the following code.

```
f = fourfun(op, [a,b], nx);
```

From this information, Fourfun objects (fourfuns) store four pieces of information. The first of these is `nx`, the number of nodes used in the interpolation process. As discussed in section 2.3, in the interest of speed `nx` Fourfun only accepts even values. After `nx` is stored, a vector `x` which stores `nx` number of points along the interval $[a, b]$ is created. These points are generated by a method `fourfunpts` which creates equally spaced points over a given interval. The second property in a Fourfun object is labeled `vals`. The variable `op` from the above snipet of code is the function that we are interpolating. The property `vals` stores the function values of `op` evaluated at the points stored in the vector `x`. The third property stored by the fourfun is `scl`. This property is used for scaling the interpolant and is simply the max of the absolute value of `vals`.

The final property is labeled `map`. This is a map of the desired interval $[a, b]$ to the interval $[-\pi, \pi]$, the default domain for all of the Fourfun calculations. The method `linear1D` is used to create the map. The map itself is a structure with five components.

TABLE 1. Summary of Fourfun object properties

`nx`	The number of nodes
`vals`	Vector of function values
`scl`	The magnitude of the function
`map`	A map between the desired interval and $[-\pi, \pi]$ and vice versa

```
>> f.map.all
ans =
     for: @(s)b*(s+pi)/(2*pi)+a*(pi-s)/(2*pi)
     inv: @(x)pi*((x-a)/(b-a)-(b-x)/(b-a))
    derx: @(x)(b-a)/(2*pi)+0*x
    name: 'linear'
     par: [0 6.283185307179586]
```

The `for` component is a function that maps the interval $[-\pi, \pi]$ to $[a, b]$. The `inv` component does the opposite, mapping the interval from $[a, b]$ to $[-\pi, \pi]$. The third component, `derx`, is the difference between any two points. The `name` component stores the name 'linear'. The final component stored is `par` which stores the original domain $[a, b]$. A summary of the structure can be printed with the command *f.map.all*, as shown above.

3.2. **Adaptive Construction.** As mentioned above, there are two options for constructing a fourfun. The difference between these choices rests on the decision of how many nodes to use in the approximation, i.e the value of `nx`. The first construction option requires the user to input the value of `nx`. Using the example function, $f = cos(e^{\sin 7x})$, this is the code to create a new Fourfun object with the first construction option:

```
f = fourfun(@(x) cos(exp(sin(7*x))),[-pi,pi], 10);
```

FIGURE 1. Plotting fourfuns of $f = \cos(e^{\sin 7x})$ constructed with different numbers of nodes demonstrates the issue of determining how many nodes are needed for interpolation. Fourfun's adaptive constructor determines this automatically.

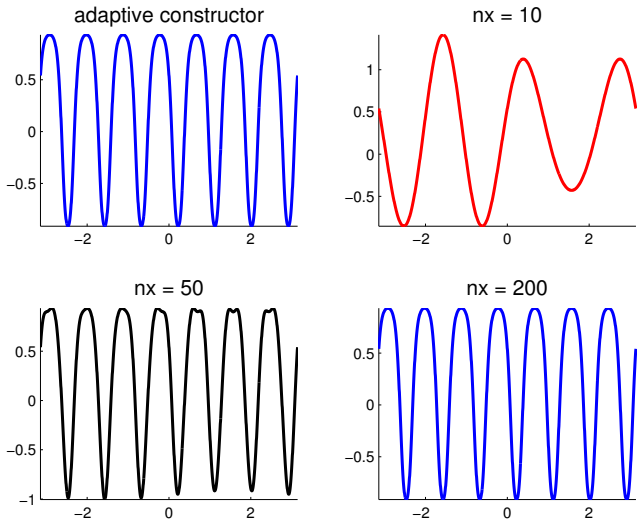


Figure 1 shows the result plotted. This figure also shows the same function constructed and plotted with `nx` = 50 and `nx` =200. It is evident from these examples that the number of nodes used to interpolate the function is essential for accurate representation. The second construction option, the adaptive constructor, handles this complication.

The adaptive constructor is designed to determine the number of points necessary to be close to computer precision. In the `ctor_adaptive` method we use `10*eps*f.scl*N` where *eps* is the MATLAB keyword for machine epsilon. Each element of this factor has a role in determining appropriate precision. This precision factor is based on the relative error of the functions,

$$\frac{||f - F_N||}{||f||_\infty} \leq eps \times N$$

where f is the function we are interpolating, F_N is the fourfun approximation, and $||f||_\infty$ is the magnitude of the function $f.scl$. Machine epsilon is multiplied by N to keep the error relative to the size of our computations. If we are dealing with a large N, then we have large frequencies. Any errors in the x direction when determining the fourfun points may lead to errors in the y direction approximations. Higher frequencies result in more egregious errors. We aim to take these errors into account by multiplying machine epsilon by the magnitude of the function, an extra factor of 10, and N. This allows for a small but reasonable range of precision.

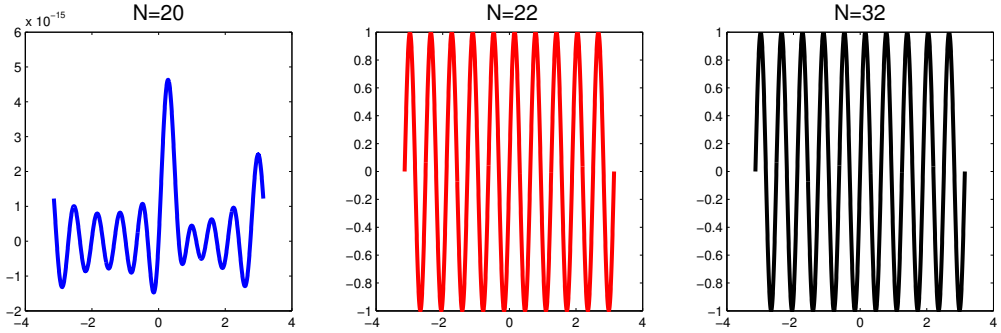
Once this range is determined, the adaptive constructor starts by constructing a fourfun g of the given function f with $N = 2$, where N is the number of nodes. A new vector `x` is made using the `fourfunpts` method with $2N$ points. Each of the functions, g and f, are evaluated at the points in `x` and the error between them is computed as an infinite norm. If this error is not within the established range, the process is repeated. That is, we reassign N as $2N$, recreate the fourfun g, and compare f and g at twice as many points. This process is repeated until the error is within an appropriate range. Then, the fourfun is saved and N represents the smallest number of points that is a power of two that can create an accurate representation of the function f.

Although an analytic determination of the number of nodes necessary for accurate interpolation is not possible for every function, we can determine this value for simple functions such as sine and cosine. A comparison between these two values is a useful tool for verifying the method. For example, using Euler's formula and the resulting identities for sine and cosine in (2), (3), and (4) we can represent $f = \sin(10x)$ as

$$\sin 10x = \frac{e^{i10x} - e^{-i10x}}{2i}. \tag{21}$$

Recalling (10), we see that the approximation of f is the sum of terms of the form $\tilde{f}_k e^{ikx}$ where k is an integer in $[-N/2, N/2]$. For $\sin(10x)$, (21) shows that we need terms with components up to $e^{i(-10)x}$ and e^{i10x}. That is, $N/2 = 10$. If we account for k=0, then this implies that we need 21 nodes. Because Fourfun accepts only even values for N, the smallest number of nodes that can be used to interpolate $\sin(10x)$ is 22. The Fourfun adaptive constructor rounds this value up to the next power of two in order to optimize the FFT computations. Thus, it uses 32 nodes for $f(x) = \sin(10x)$. This is supported graphically by Figure 2.

FIGURE 2. Graphical representations of a fourfun of $\sin 10x$ constructed with 20 nodes, 22 nodes, and the adaptive constructor which uses 32 nodes, supports that the minimum number of nodes for representing $\sin 10x$ with Fourfun is 22.

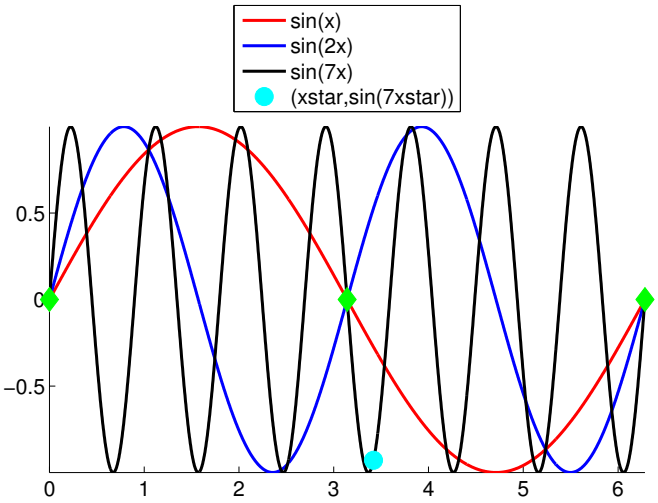


Although the adaptive constructor will always use the next highest power of two, the Fourfun system is capable of interpolating any valid periodic function with an even number of nodes. To specify the number of nodes, the first construction option should be used.

3.3. **Aliasing.** The number of nodes used to interpolate a function using the Fourfun system is closely tied to the concept of aliasing that arises when dealing with periodic functions. Figure 3 visually demonstrates this complication. The green diamonds are points that have identical values on all three functions, $\sin x$, $\sin 2x$, and $\sin 7x$. Because Fourfun's determination of the appropriate number of nodes relies on the error between the given function f and the fourfun g, it is possible that the function values at f and at g have identical values but that g is actually the interpolation of a lower frequency function called the alias of f. The adaptive constructor for the Fourfun system is designed to avoid aliasing.

In the adaptive constructor, a special value, `xstar`, is introduced. In addition to computing the error between f and g, the adaptive constructor compares the values of f and g evaluated at `xstar`. With this additional check in place, even if the functions have the same function values at the points being sampled, the error will not be within the range we ask for. The adaptive constructor will continue to increase the number of nodes used to create the fourfun until this condition is met.

FIGURE 3. This example is included to visually represent aliasing. The three functions are plotted simultaneously to show that a certain x value may have the same function value on all three functions (the green diamonds). Use of the point $xstar$ aims to avoid this complication.



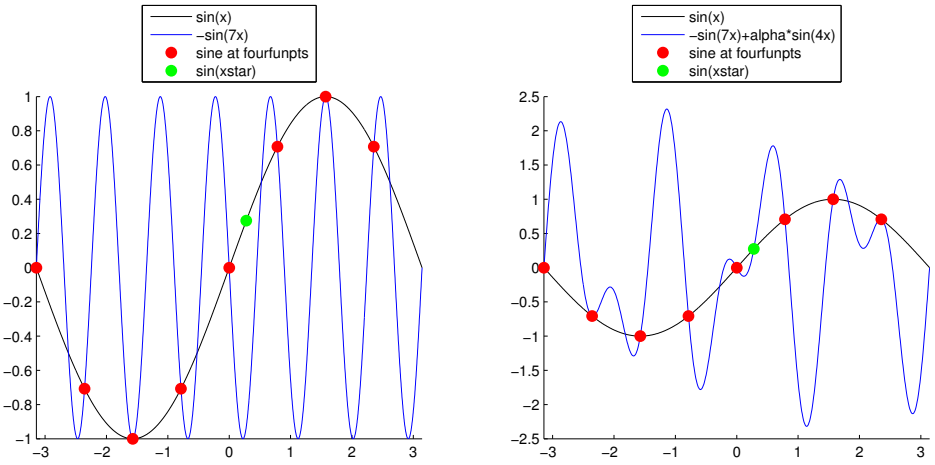
The value of `xstar` was generated once using MATLAB's *rand* command. On the interval $[-\pi, \pi]$, its value is 0.2785 and it is stored in the adaptive constructor. When constructing fourfuns on other intervals, `xstar` is mapped accordingly to that interval using the forward map stored as part of the map structure in the fourfun object. We feel comfortable choosing a single value for `xstar` considering that the probability of coming up with any single value between 0 and 1 is equal. However, there are consequences that come with this choice. Because `xstar` is known, it is possible that a function could be composed for which aliasing would occur at this value. Take for example, the equation

$$f(x) = -\sin 7x + \alpha \sin 4x, \text{ where } \alpha = \frac{\sin(xstar) + \sin(7xstar)}{\sin(4xstar)}. \tag{22}$$

This function will alias with $\sin x$. A fourfun of $\sin x$ is made with 4 nodes. Let's imagine that we wanted to make a fourfun of $-\sin 7x$. During the construction process, a fourfun of $-\sin 7x$ is initially made with 4 nodes, like $\sin x$, and is compared with the original function $-\sin 7x$ at 8 points. From Figure 4, it is clear that the value of $-\sin 7x$ will match the value of $\sin x$ at all 8 nodes. In this case, the value of the fourfun

at `xstar` would not match that of the function, $-\sin 7x$, and the Fourfun adaptive constructor would make the fourfun again with more points. However, if we add the second term of (22), the fourfun made with 4 nodes will now match $\sin x$ in value at all 8 points and at `xstar` as demonstrated in Figure 4. Thus, aliasing will occur and the constructed fourfun will represent $\sin x$ rather than (22).

FIGURE 4. One consequence of a set value for *xstar* is the ability to construct a function for which aliasing will occur. The graphs below demonstrate how adding a term to $-\sin 7x$ allow us to match 8 points between $\sin x$ and (22) as well as *xstar* which causes aliasing.



A possible solution to this problem is to generate a random number for `xstar` each time a new fourfun is made. However, because the number of nodes used for the interpolation is dependent on the error of the function value and the fourfun value at `xstar`, this could result in the same fourfun being made with a varying number of nodes each time it is computed. In the interest of consistent and repeatable results, we have opted to use a single value of `xstar`.

4. Applications of Fourfun

4.1. **Basic Examples.** This section aims to demonstrate basic uses of Fourfun that may differ from standard MATLAB syntax and that are used in applications to more meaningful problems. Moreover, it aims to highlight the way in which Fourfun allows the user to think in a more symbolic manner, while letting Fourfun handle the numerics.

- **Basic operators**
  To add or subtract two fourfuns, we create two Fourfun objects and then apply operators to them.

```
f=fourfun(@(x) sin(x));
g=fourfun(@(x) cos(4*x));
h=f+g
j=f-g
```

f uses 4 points for interpolation and g uses 16. In the outputs, we see that both h and j use 16 points.

```
h =
  fourfun with properties:
      nx: 16
    vals: [1x16 double]
     scl: 2
     map: [1x1 struct]
j =
  fourfun with properties:
```

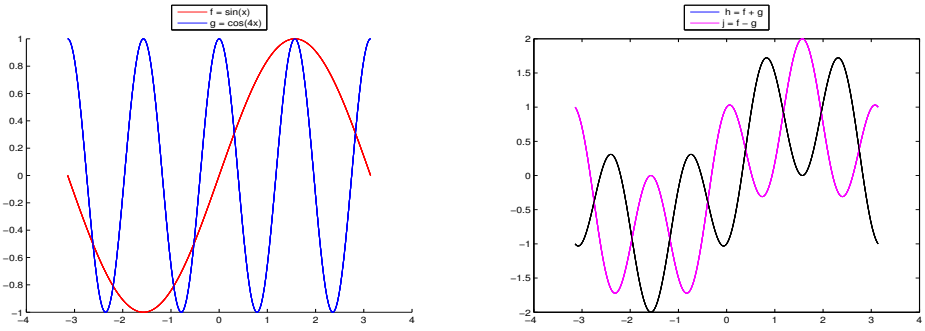
```
      nx: 16
    vals: [1x16 double]
     scl: 2
     map: [1x1 struct]
```

We can also plot these functions with the plot command and other standard MATLAB plot specifications. The output is shown in Figure 5.

```
subplot(1,2,1)
hold on
plot(f,'-r')
plot(g,'-b')
legend('f = sin(x)', 'g = cos(4x)','location','northoutside')
hold off

subplot(1,2,2)
hold on
plot(h,'m')
plot(j, 'k')
legend('h = f + g', 'j = f - g','location','northoutside')
hold off
```

FIGURE 5. Plots of $\sin(x)$, $\cos(4x)$, $\sin(x)+\cos(4x)$, & $\sin(x)-\cos(4x)$



- **Derivatives**
  Keeping f the same as in the previous example, we can easily take the derivative with this code.

```
dfdx=diff(f)
```

The output tells us that the derivative is a Fourfun object with 4 nodes as well.

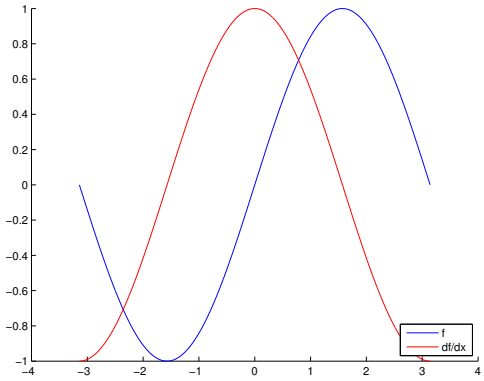
```
dfdx =
  fourfun with properties:
      nx: 4
    vals: [-1 6.1232e-17 1 -6.1232e-17]
     scl: 1
     map: [1x1 struct]
```

The `dfdx.vals` and the plot, shown in Figure 6, confirm that the derivative of $\sin(x)$ is $\cos(x)$. Moreover, the error was computed as the maximum value of the absolute values of the difference between the fourfun, dfdx, and the function $\cos(x)$ evaluated at 1000 points. The error was 5.5511×10^{-16}.

- **Antiderivatives & integrals**
  A similar code leads to the antiderivative of the fourfun f. Additionally, the command `integral` gives the definite integral of f between $[-\pi, \pi]$.

```
F=int(f)
fint= integral(f)
```

FIGURE 6. Plot of the fourfuns f and df/dx

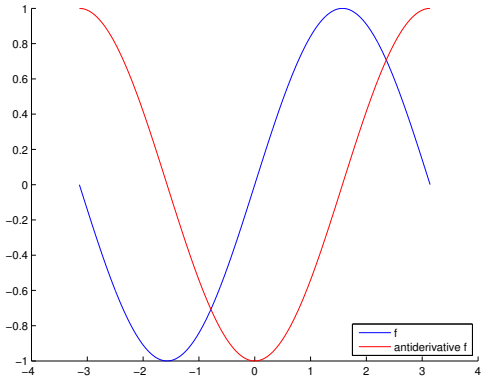


where the output is

```
F =
  fourfun with properties:
      nx: 4
    vals: [1 -6.1232e-17 -1 6.1232e-17]
     scl: 1
     map: [1x1 struct]
fint =
  -3.4879e-16
```

As expected, the integral of f is approximately zero which serves as a basic check in the error. The F.vals and plot, shown in Figure 7 confirm that the antiderivative of f is $F = -\cos(x)$. Computing the error by the same method as described for the derivative, the error comes out to be the same at 5.5511×10^{-16} which is expected due to the similarity of the integral and derivative in this situation.

FIGURE 7. Plot of the fourfuns f and the antiderivative of f



- **A complicated function**
  Fourfun is capable of handling more complicated functions as well. For an example we will consider the function $f = e^{-4x^2} \sin 20x + 0.5 \cos 2x$.

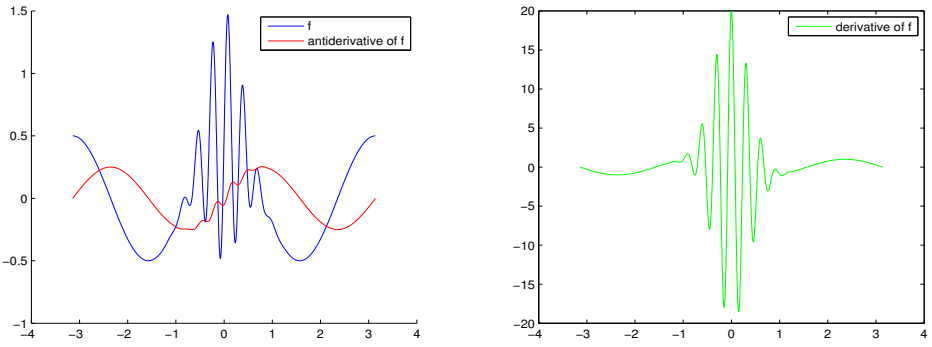
```
f=fourfun(@(x) exp(-4*x.^2).*sin(20*x)+ 0.5*cos(2*x))
g = diff(f)
h = int(f)
```

The output is below, and the plot is shown in Figure 8. Fourfun quickly and efficiently computes and plots the derivative and indefinite integral.

```
f =
  fourfun with properties:
      nx: 128
    vals: [1x128 double]
     scl: 1.3793
     map: [1x1 struct]
g =
  fourfun with properties:
      nx: 128
    vals: [1x128 double]
     scl: 20.0000
     map: [1x1 struct]
h =
  fourfun with properties:
      nx: 128
    vals: [1x128 double]
     scl: 0.2539
     map: [1x1 struct]
```

Again, the error for the derivative can be considered in the same manner as in the previous examples. The error was computed to be 2.1050×10^{-13}. Because the integral cannot be determined analytically, the error is not compared in this situation. The ease in which Fourfun can compute this integral illustrates one of the benefits of using the code.

FIGURE 8. Plots of a complicated function and its derivative and antiderivative



- **Root-Finding**
  Fourfun can also find roots of a function. Take the complicated function from above. We can find the roots of this function, its derivative, and its integral.

```
f=fourfun(@(x) exp(-4*x.^2).*sin(20*x)+ 0.5*cos(2*x));
g=int(f); h=diff(f);

z=roots(f); z1=roots(g); z2=roots(h);
```

The three plots created with this code are shown in Figure 9. Moreover, to verify that these are the proper roots, we can evaluate each function f, g, and h at its computed roots. As an example, g is evaluated at its roots.

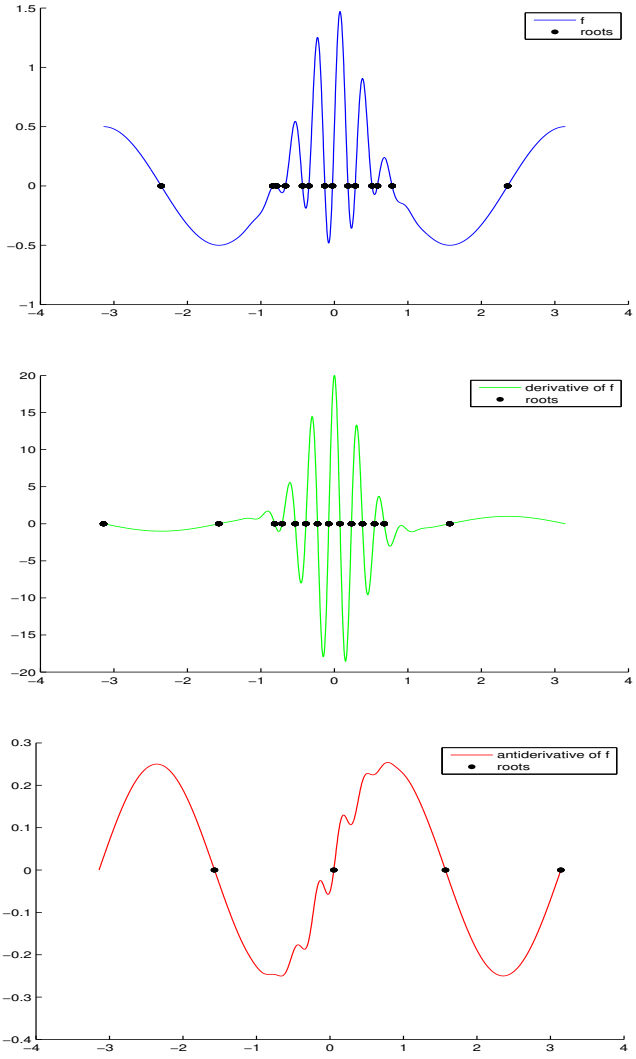
```
gzero = g(z1)
```

The output is shown below.

```
gzeros =
   1.0e-13 *
```

```
-0.4663
 0.6002
-0.4971
-0.4657
```

FIGURE 9. Plots showing the roots of a complicated function, its derivative and antiderivative



4.2. **Partial Differential Equations.** One clear motivation for the Fourfun system is its applicability to partial differential equations with periodic boundary conditions. This basic application rests on approximating the initial condition, u, the new estimate u_{new}, and the old estimate u_{old} with fourfuns. These fourfuns can then be used with standard time stepping schemes such as the Leapfrog or Runge-Kutta methods. For the spatial derivatives, the `diff` method is used. Consequently, we can think as if we have a function for the spatial derivative rather than simply discrete data. This described method for applying fourfuns to the solution of differential equations can be seen in the following examples of solutions to the well-known transport and wave equations.

4.2.1. *Transport equation.* The following code is an example of how to implement a solution to the transport equation

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}$$

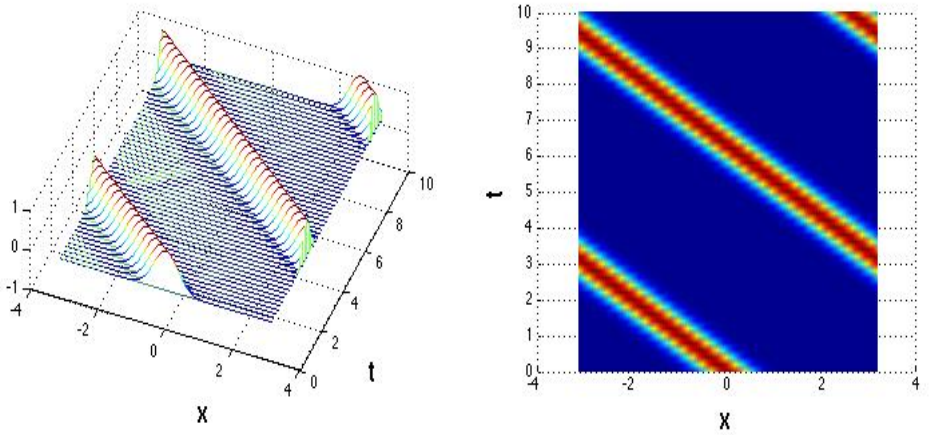
using the Fourfun system. The initial condition, given as $u(x,0) = e^{-4x^2}$, is stored as a Fourfun object and the time derivatives are computed using the Leapfrog method. The ability to easily add, subtract, and differentiate the Fourfun objects allows us to essentially write the formula of the chosen time differentiation scheme without having to think about the spatial derivatives. Fourfun methods `subsref` and `feval` make recording solutions to plot equally simple. The plotted solution is shown in Figure 10.

```
% Solution of the trasport equation with Fourfuns
% u_t = u_x, with periodic BC's
% Time-step by leapfrog.
% Initial condition:
% f = exp(-4*x^2);
dt = 0.02;
uold = fourfun(@(x) exp(-4*(x-dt).^2),[-pi pi]);
u = fourfun(@(x) exp(-4*x.^2),[-pi pi]);

xx=linspace(-pi,pi,100)'; % establish x vector for plotting
sol(:,1)=u(xx); % establish solution matrix for plotting
t(1) = 0; % establish time vector for plotting

for k = 1:50
    for j = 1:10
        unew = uold + 2*dt*diff(u);
        uold = u;
        u = unew;
    end
    2/u.nx
    sol(:,k+1) = unew(xx); % save the latest solutions at xx values
    t(k+1)=t(k)+10*dt; % save time
end
```

FIGURE 10. Plot of solution to the transport equation generated using fourfuns



4.2.2. *Wave equation.* We can implement a solution to the wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$$

using a similar method. In this problem the initial conditions are given as $u(x,0) = e^{-5(x-0.5)^2}$ and $u_t(x,0) = 0$. After creating fourfuns for the initial conditions, we can again implement a Leapfrog method for time steps. The second spatial derivative is as simple to implement as the first derivative using the `diff` command with the optional second parameter. The two plots of the solution are shown in Figure 11.

```
% Solution of the wave equation with Fourfuns
% u_tt = u_xx, with periodic BC's
```

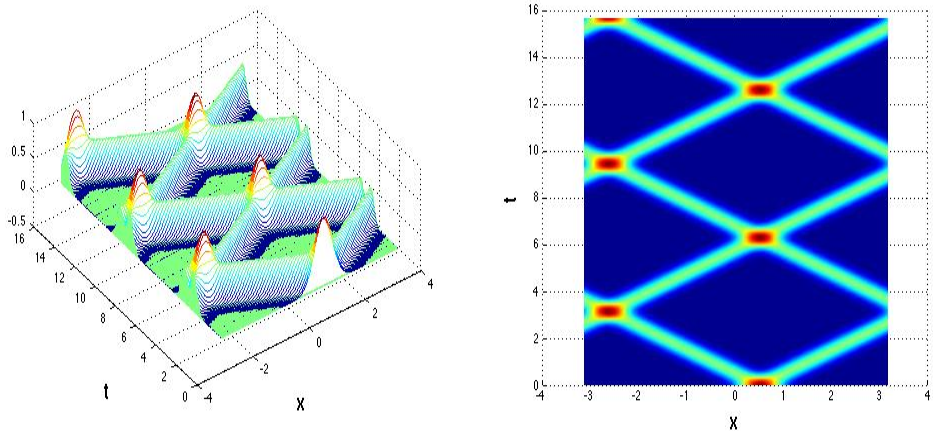
```
% Time step by leapfrog

%intial conditions
u1 = fourfun(@(x) exp(-5*(x-0.5).^2));
u0 = u1;
dt = .5/u1.nx;

%establish vectors for saving solutions
xx = linspace(-pi,pi,100)'; %
sol(:,1) = u1(xx);
t(1) = 0;

for k = 1:200
    for j = 1:10
      u = 2*u1 - u0 + dt^2*diff(u1,2);
      u0 = u1;
      u1 = u;
    end
    sol(:,k+1) = u1(xx); % save latest solution
    t(k+1) = t(k)+10*dt; % save time at latest solution
end
```

FIGURE 11. Plot of solution to the wave equation generated using fourfuns

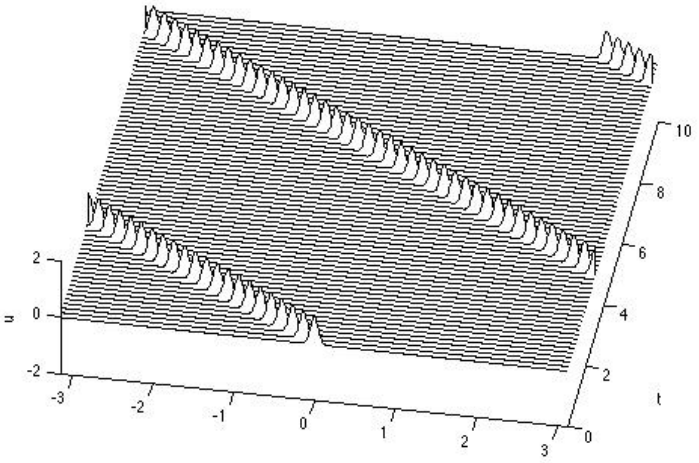


As mentioned in the introduction, one of the motivations for using Fourfun is an improved CFL condition for certain problems and consequently, the ability to take larger time steps when numerically computing solutions. We can explore this notion, by comparing the solution to a second order wave equation on a Chebyshev grid presented in [16] with one computed with Fourfun. We have adapted program 19 in [16] to solve the wave equation with periodic boundary conditions and have composed a similar program to compute the solution using Fourfun. The plot of the solution generated in Fourfun is shown in Figure 12. To generate the plot in Figure 12, we use a time step of $\Delta t = 2/N$ where $N = 512$, which is the number of points used to generate the fourfun of the initial condition, $u = e^{-200x^2}$ between $[-\pi, \pi]$. The same problem solved on a Chebyshev grid uses $\Delta t = (3.5)(8)/N^2$. That is, $\Delta t = 0.0039$ and $\Delta t = 1.0979 \times 10^{-4}$ for the solution in a Fourfun and Chebyshev domains respectively. Depending on the problem at hand, this ability to take larger times steps with Fourfun could be beneficial from both a time and storage perspective.

5. Comparison to Chebfun

Similarly to the Fourfun system, the Chebfun system aims to, as the title of [14] suggests,"[compute] numerically with functions instead of numbers." However, unlike Fourfun, the Chebfun system is not restricted to periodic functions on bounded domains. Chebfun requires only piecewise smooth functions in the univariate case. In the same way that Fourfun relies on Fourier expansions to interpolate functions, Chebfuns

FIGURE 12. Plot of solution to a second order wave equation from program 19 of [16] with periodic boundary conditions using Fourfun



utilizes Chebyshev polynomials for interpolation [11, 14]. Consequently, the Chebfun system is capable of interpolating the same periodic functions that the Fourfun system interpolates. Despite this capability, a discussion of the comparable speeds of these systems for specific computations explores a possible preference for the Fourfun interpolation system when dealing with periodic functions.

TABLE 2. Comparison of Chebfun and Fourfun: Derivatives

Function		**Fourfun**		**Chebfun**	
	no. of deriv.	no. of coeff.	time (s)	no. of coeff.	time (s)
$\sin(x)$	1	4	0.000765	21	0.007653
	3	4	0.000722	19	0.011206
	10	4	0.000754	12	0.030046
$\sin(200x)$	1	512	0.000826	713	0.006906
	3	512	0.000831	711	0.010028
	10	512	0.000855	704	0.022999
$\cos(x)e^{\sin(40x)}$	1	1024	0.000860	1643	0.007121
	3	1024	0.000913	1641	0.011344
	10	1024	0.000917	1634	0.023207

Table 2 compares the time that the Fourfun and Chebfun systems take to compute the derivatives of three different periodic functions. While taking ten derivatives is not likely in practice, the tenth derivatives are included simply for the sake of comparing speed. Additionally, both systems create new objects of the same type when taking the derivative. The table indicates the number of coefficients each method stores for these new derivative objects. Because the derivative of fourfuns are calculated from the Fourier coefficients, the number of coefficients remains constant throughout the derivative process and thus the system sees little increase in the time it takes to calculate one derivative versus ten derivatives. The time taken to compute the fourfun derivatives is approximately ten times faster than for chebfun derivatives in each of these trials.

A similar result can be seen in Table 3 when comparing the antiderivatives of fourfuns and chebfuns. Here again we see the number of coefficients used in the fourfuns remains constant for each function and the time it takes to compute the antiderivative is approximately ten times faster for the Fourfun system than the for the Chebfun system.Depending on the size the problem that either of these systems is being applied to, the quicker computation speed of the Fourfun derivatives and antiderivatives may be preferable over the more broadly applicable Chebfun system.

TABLE 3. Comparison of Chebfun and Fourfun: Antiderivatives

Function		**Fourfun**		**Chebfun**	
	no. of int.	no. of coeff.	time (s)	no. of coeff.	time (s)
$\sin(x)$	1	4	0.000852	21	0.010338
	3	4	0.001587	23	0.025502
$\sin(200x)$	1	512	0.001776	699	0.013763
	3	512	0.001067	681	0.028441
$\cos(x)e^{\sin(40x)}$	1	1024	0.001247	1403	0.012441
	3	1024	0.001976	1022	0.025990

The discrepancies in computation time between Fourfun and Chebfun can be explained by the number of times each system uses FFTs and inverse FFTs for computing derivatives and antiderivatives. The complexity of both the FFT and inverse FFT is $\mathcal{O}(N \log N)$. Recalling (9), we see that the Fourier coefficients can be found using an FFT. Equation (14) then shows that for the derivative we only need to calculate these coefficients one time, that is with one FFT, to compute $f^{(n)}$. An inverse FFT is then used to return to the initial domain. Thus, Fourfun computes just one FFT and one inverse FFT for each calculation in Table 2. On the other hand, Chebyshev polynomials used in Chebfun do not have this convenient relationship between the coefficients in each derivative. Chebyshev polynomials use points that are equally spaced on a unit circle. This property can be exploited for computing Chebyshev coefficients with an FFT. However, the same coefficients cannot be used for each derivative, therefore Chebfun must compute a FFT and an inverse FFT for each derivative it computes. That is, for three derivatives, three FFTs will be taken and three inverse FFTs will be taken. A similar argument can be made for the computation time differences in the anti differentiation process, which again is ultimately due to the number of FFTs and inverse FFTs computed.

6. CONCLUSION

Both the Fourfun and Chebfun systems aim to allow users to think more about problems and less about the numerical accommodations that must be made when working with discrete data. Fourfun is limited to periodic functions, but it may be preferable to Chebfun in this context. In our experiments, Fourfun computed derivatives and antiderivatives approximately ten times faster than Chebfun. The uniform nodes of the Fourfun method compared with the nonuniform Chebyshev points make it better suited for applications to differential equations where the time step is dependent on the grid size. For these problems, Fourfun generally results in a lower CFL condition than Chebfun, making it more optimal.

Furthermore, Fourier expansions have long been the method of choice for approximating smooth periodic functions. Besides being accurate, computations can be efficiently carried out using FFTs. The Fourfun system aims to make this powerful mathematical tool available to non-experts in scientific computing, with a friendly syntax that is an easy extension of current MATLAB commands and automated error control. It is our hope that Fourfun will provide developing mathematicians and researchers from other fields the ability to incorporate the power of Fourier expansions into their work without the need for explicit knowledge of the underlying numerical considerations.

With this goal in mind, Fourfun needs more development before becoming a self contained system. At this point, most of the basic functions are overloaded for uses with fourfuns, but a more exhaustive list of methods would surely increase possible applications of the system. Some examples would be a division method or methods to enable linear algebra computations. In our trials, the system was fast in comparison to Chebfun for computing derivatives and antiderivatives, but more time should be spent considering optimization and efficiency. This may result in the Fourier coefficients being stored as a property of the function or an innovative method for determining how many nodes should be used for interpolation. An extension of Fourfun to two dimensions is another avenue of further research.

-

Acknowledgments

I would like to sincerely thank Dr. Rodrigo Platte for his continued advice and support throughout this project. In addition, I would like to thank Dr. Nick Trefethen, Dr. Grady Wright, Hadrien Montanelli, Mohsin Javed and Dr. Anne Gelb for their commentary and suggestions on earlier versions of this manuscript. Finally, I thank the National Science Foundation for funding and Drs. Eric Kostelich and Bruno Welfert for coordinating the Mentoring through Critical Transition Points Program at Arizona State University, which provided a supportive and creative environment that was essential during the beginning stages of this research.

Appendix: Description of Fourfun Methods

Many of MATLAB's functions are overloaded for use with Fourfun objects. These methods are divided into to groups: "private" methods which are methods used in the construction of fourfuns and "public" methods which allow for interaction of fourfuns with conventional MATLAB syntax. The methods `fourfunpts` and `linear1D` have been described in the paper and so are excluded here.

- `bcinterp` is the implementation of the barycentric formula that is described in section 2.4. The function takes three input values and returns a vector. The inputs are x_k the fourfun points that make up the grid, f the function values stored in the constructor as `f.vals`, and x a vector of values for which we would like to compute the function value. A vector g is returned with the function values at the points in x. If a value in x is equal to a value in x_k, one of the fourfun points, then no computation is done and the function value in `f.vals` that corresponds to this x_k value is stored g.
- `ctor` is the implementation of what was described previously as the first construction option. It has four parameters g, the fourfun that is being constructed, *op* the function that is being interpolated, *end* the interval on which the function is to be interpolated, and nx the number of nodes used. From this information the four properties described in Table 1 are generated.
- `ctor_adaptive` is the construction option which is described in more detail in section 3.2. This method can take up to three parameters, g, *op*, and *ends* which are defined the same way as in the `ctor` method. The *ends* parameter is not required. If it is omitted, then the default domain is assumed.
- `prolong` takes two inputs: f a fourfun and Nx a number. Initally the method determines if the number of nodes in the fourfun, $f.nx$, is equal to Nx. If so, the function f is returned and nothing more is done. However, if $f.nx \neq Nx$ then the method alters f so that it uses Nx nodes. To do this, `fourfunpts` is used to make a vector of Nx points on the interval $f.map.par$, the original interval of f. The function is evaluated at these new points to update `f.vals` and ultimately `f.nx` is updated to Nx as well. The motivation for this alteration is for use in the `times` and `plus` methods that will be described more below.
- `fourfun` is the class definition of Fourfun objects which establishes the properties of Fourfun objects as nx, $vals$, scl, and map. Moreover, this class definition uses the number of inputs to determine whether the user would like the first constructor, `ctor`, or the second adaptive constructor `ctor_adaptive`.

The remaining methods are the "public" methods that have been overloaded to help make it possible to use the exact syntax with fourfuns that one would use with vectors in MATLAB.

- `feval` is the basic method for calling `bcinterp` and computing the function values at a given matrix of x values. The function takes two inputs, f the fourfun and x. This matrix x is passed as a column vector to `bcinterp`. The `bcinterp` function returns a vector that is reshaped in order to return the function values in the dimensions of the original matrix.
- `uminus & minus` are two methods with similar purposes. `uminus` is the first step in creating a subtraction method; it negates a fourfun. This is used in the `minus` method which subtracts the two fourfuns f and g by adding $f + (-g)$. The `plus` method, described next, is used for addition.
- `plus` handles both the case of adding a constant to a function and the case of adding two fourfuns together. It takes two inputs, f and g. One or both of these must be a fourfun for the method to be invoked. In the case that one of the inputs is a scalar and the other is a fourfun, the method guarantees that the second parameter is the constant by recalling the function with the parameters

switched if necessary. Following this, the scalar is added to each element of `f.vals` and `f.scl` is updated by adding the absolute value of g to it.

The process is slightly more complicated for adding two fourfuns together. First, using `prolong` both of the fourfuns are made the length of $max(f.nx, g.nx)$. In other words, we force the fourfun with less nodes use the same number of nodes as the longer fourfun. After this, we can simply add together the function values, `f.vals` and `g.vals`. The scale is updated by adding together the scales from both functions.

- `mtimes & times` deal with multiplying fourfuns. `mtimes` controls the multiplication of a fourfun by a scalar. Similarly to the `plus` function, the method takes two inputs f and g, one of which is a scalar. The function determines if f or g is a scalar and recalls the function if necessary to place the scalar is the second parameter. If g is scalar, the function values of f are multiplied by g to get the new function values. The scale is also multiplied by the constant.

  The `times` method is also similar. It is necessary to use the common MATLAB element-wise vector multiplication notation ".*" to indicate the multiplication of two fourfuns. To determine the appropriate number of nodes, $Nx = 2^{ceil(\log_2(f.nx+g.nx))}$ is evaluated. Both the f and g inputs are lengthened to Nx nodes using `prolong`. After these adjustments, the function values are multiplied together to get the new function values. The scale is updated by taking the maximum of the absolute value of the new function values.
- `cos, sin, log & exp` are very basic to implement. They simply reconstruct the fourfun calling feval inside the desired function to create the new `f.vals`. For instance, we could start with a fourfun f that interpolates $\sin x$. If we would like to take the cosine of f, we can call the constructor using the command `g = fourfun(@(x) cos(feval(f,x)))`. The resulting fourfun is returned. This same procedure can be easily adapted to the sine and exponential functions. Special consideration must be taken with implementing the `log` method due to the domain restrictions on logarithmic functions. Consequently, the `log` method is only used in cases where `f.vals` > 0.
- `int & integral` are responsible for the indefinite and definite integral of the fourfun f respectively. `int` implements (18). First, the Fourier coefficients are computed by taking the FFT of the `f.vals` and then they are divided by $(ik)^n$. It's important to note that in the computation of the FFT, MATLAB does not store the values from $[-\frac{N}{2}, \frac{N}{2}]$. Instead, the indices are rearranged during the FFT computation to be from $[0, 1, \cdots \frac{N}{2}, \frac{-N}{2}, \cdots, -2, -1]$. As discussed in the section 2.5, the first coefficient must be equal to zero in order to ensure the periodicity of the antiderivative of the function. In accordance with this and to avoid the "Inf" term that results from dividing by zero, the first Fourier coefficient, corresponding to the zero index, is set equal to zero. Using the inverse FFT, the values of the function are transformed back to the spatial domain and multiplied by the original interval divided by 2π to scale the terms appropriately. These values are then stored as the `f.vals` of the fourfun of the antiderivative. The scale of the function is updated as the maximum of the absolute value of `f.vals`.

  `integral` computes the definite integral and takes up to two inputs. The first must be the fourfun and the optional second and third are a and b defining a subinterval $[a, b]$ over which the integral is to be computed. If the default domain is chosen, then the integral is estimated using trapezoid rule. Otherwise, if $[a, b]$ is a subinterval of the original domain, the antiderivative is computed with the `int` method and the fundamental theorem of calculus is used to determine the integral.
- `diff` is an implementation of (14), and it is similar to the `int` method. The Fourier coefficients are computed using a FFT and are multiplied by $(ik)^n$. As described in the `int` explanation, consideration is made for the FFT's rearrangement of indices. The values are then transformed back to the spatial domain using the inverse FFT and the values are multiplied by a scaling factor of 2π divided by the difference in the original endpoints before being stored as `f.vals` of the fourfun of the derivative. The scale is updated as well by by computing the maximum of the absolute value of these new `f.vals`.
- `roots` finds the zeros of a fourfun. This function takes one parameter which is the fourfun we would like to find the roots of. Equation (9) is implemented to find the Fourier coefficients: take an FFT of the `f.vals` and divide by N which is the length of `f.vals`. As stated in the discussion of the integral method, the FFT rearranges the indices for computation. Here the `fftshift` command must be used to reorder the indices in increasing order. We then prune the FFT's to only include those that

are greater than `1e10^{-12} * f.scl`. These remaining Fourier coefficients are divided by the last one (the greatest) in order to give coefficients to a monic polynomial. The roots can then be solved by finding the eigenvalues of the companion matrix. The values returned are for z and $z = e^{ix}$, so we solve for x to find the roots of the fourfun.

- `plot` is relatively simple to implement and relies on the built in MATLAB plot function. The *f.map.par* reference is used to return the original domain. The `linspace` command is then used with 2000 points to create an x vector for plotting. The y vector is created using the `feval` function to evaluate the fourfun at the vector x. The result is plotted using the MATLAB plot function. Any further plotting specifications, such as color or linestyle are passed through the `varargin{:}` parameter. Consequently, these specifications can be used synonymously to the standard MATLAB plotting commands.
- `subsref` has two main purposes. First, it enables a fourfun f to be evaluated at the vector x using the notation $f(x)$. This is simply a different notation for the `feval` method, however it resembles more standard written mathematical notation.

  The second purpose of the `subsref` method is to provide access to the properties of Fourfun objects. Using the familiar object-oriented programming notation of "object.property" we can print out `f.nx, f.vals, f.scl`, each individual element of the map structure `f.map.inv`, `f.map.for`, `f.map.derx`, `f.map.name` and `f.map.par`, and a summary of the map structure `f.map.all`.

References

[1] Z. Battles and L. N. Trefethen. An extension of MATLAB to continuous functions and operators. *SIAM J. Sci. Comput.* 25(5):1743-1770, 2004.

[2] J. P. Boyd. Computing the zeros of a Fourier series or a Chebyshev series or general orthogonal polynomial series with parity symmetries. *Comput. Math. Appl.* 54: 336-349, 2007.

[3] R. N. Bracewell. *The Fourier Transform and its Applications*. McGraw Hill, 1986.

[4] W. L. Briggs and V.E. Henson. *The DFT: An owner's manual for the discrete Fourier Transform*. Society for Industrial and Applied Mathematics, 1995.

[5] D.M. Causon and C.G.Mingham. *Introductory finite difference methods for PDEs*. Ventus Publishing ApS, 2010.

[6] J.W. Cooley, and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.* 19(90):297-301, 1965.

[7] P.P.G. Dyke. *An Introduction to Laplace Transforms and Fourier Series*. Springer, 2004.

[8] J. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, 2007.

[9] P. Henrici. *Applied and computational complex analysis*. John Wiley & Sons, 1986.

[10] C. Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, 2004.

[11] R. Pachón, R. B. Platte, and L. N. Trefethen. Piecewise smooth chebfuns. *IMA J. Numer. Anal.* 30:898-916, 2010.

[12] R.B. Platte and L.N. Trefethen. Chebfun: a new kind of numerical computing. *Progress in Industrial Mathematics at ECMI 2008*. Springer:69-87, 2008.

[13] S.W. Smith. The scientist and engineer's guide to digital signal processing. California Technical Publishing, 1997.

[14] L. N. Trefethen. Computing numerically with functions instead of numbers. *Math. comput. sci.* 1:9-19, 2007.

[15] L.N. Trefethen and D. Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[16] L. N. Trefethen. *Spectral Methods in Matlab*. Society for Industrial and Applied Mathematics, 2000.