The Mathematica® Journal

On Some Applications of the Fast Discrete Fourier Transform

Alkiviadis G. Akritas
Jerry Uhl
Panagiotis S. Vigklas

Motivated by the excellent work of Bill Davis and Jerry Uhl's *Differential Equations & Mathematica* [1], we present in detail several little-known applications of the fast discrete Fourier transform (DFT), also known as FFT. Namely, we first examine the use of the FFT in multiplying univariate polynomials and integers and approximating polynomials with sines and cosines (also known as the fast Fourier fit or FFF). We then examine the use of the FFF in solving differential equations with Laplace transforms and rediscovering trigonometric identities.

Introduction

We begin with a review of the basic definitions needed.

Let R be a ring, $n \in \mathbb{Z}$, $n \geq 1$, and $\omega \in R$ be a primitive nth root of unity; that is, $\omega^n = 1$ and $\omega^{n/t} - 1$ is not a zero divisor (or, $\omega^{n/t} - 1 \neq 0$) for any prime divisor t of n. We represent the polynomial $f = \sum_{i=0}^{n-1} f_i x^i \in R[x]$, of degree less than n by the coefficient list, in reverse order, $\{f_0, \ldots, f_{n-1}\} \in R^n$.

Definition 1 (DFT). *The R-linear map* $\mathrm{DFT}_\omega : R^n \to R^n$*, which evaluates a polynomial at the powers of* ω*, that is,* $\mathrm{DFT}_\omega : \{f_0, \ldots, f_{n-1}\} \mapsto \frac{1}{\sqrt{n}} \left\{f(1), f(\omega), \ldots, f\left(\omega^{n-1}\right)\right\}$*, is called the discrete Fourier transform (DFT).*

In other words, the DFT is a special multipoint evaluation at the powers $1, \omega, \ldots, \omega^{n-1}$ of a primitive nth root of unity ω. The fast implementation of the DFT is known as the fast DFT, or simply as FFT; it can be performed in time $O(n \log n)$. Details can be found in the literature [2]. Keeping it simple, we mention in passing that the *inverse* DFT is defined as the problem of interpolation at the powers of ω and is easily solved.

In *Mathematica* the map DFT_ω and its inverse are implemented—for the complex numbers—by the functions `Fourier` and `InverseFourier`. The FFT is implemented in `Fourier`. So, for example, the definition is verified by

```
f[x_] := x^3 - 7 x + 7;
{Fourier[CoefficientList[f[x], x]]} ==
 {n = 4; ω = ⅇ^((2 π ⅈ)/n); 1/Sqrt[n] {f[1], f[ω], f[ω^2], f[ω^3]}}
```

```
True
```

Definition 2. *The convolution of two polynomials $f = \sum_{i=0}^{n-1} f_i\, x^i$ and $g = \sum_{k=0}^{n-1} g_k\, x^k \in R[x]$ is the polynomial*

$$h = f *_n g = \sum_{j=0}^{n-1} h_j\, x^j \in R[x],$$

where

$$h_j = \sum_{i+k\equiv j \bmod n} f_i\, g_k = \sum_{i=0}^{n-1} f_i\, g_{j-i}, \qquad \text{for } 0 \le j < n,$$

and the arithmetic at the indices of g_{j-i} (in the second summation) is done modulo n. If we regard the polynomials as vectors in R^n, then what we have is the cyclic convolution of the vectors f and g.

There is an equivalence between convolution and polynomial multiplication in the ring $R[x]/\langle x^n - 1\rangle$. Please note for the given polynomials f, g that the j^{th} coefficient of their product, $f\,g$, is $\sum_{i+k=j} f_i\, g_k$; whereas the corresponding coefficient of their convolution, $f *_n g$, is $\sum_{i+k\equiv j \bmod n} f_i\, g_k$ and hence $f *_n g \equiv f\, g \bmod x^n - 1$. Moreover, if $\deg(f\,g) < n$, then $f *_n g = f\, g$. We will exploit this equivalence to develop a fast polynomial multiplication algorithm. The following theorem holds.

Theorem. *Let R be a ring, $n \in \mathbb{Z}$, $n \ge 1$, and let $\omega \in R$ be a primitive root of unity of order n. Then for the polynomials $f, g \in R[x]$ of degree at most $n - 1$, we have*

$$\mathrm{DFT}_\omega\,(f *_n g) = \mathrm{DFT}_\omega\,(f) \cdot \mathrm{DFT}_\omega\,(g),$$

where $\cdot$ indicates "element-wise" vector multiplication.

Proof. We know that $f *_n g = f\, g + q(x^n - 1)$ for some $q \in R[x]$. Then we have

$$(f *_n g)\left(\omega^i\right) = f\left(\omega^i\right) g\left(\omega^i\right) + q\left(\omega^i\right)\left(\omega^{i\,n} - 1\right) = f\left(\omega^i\right) g\left(\omega^i\right)$$

for $0 \le i \le n - 1$. □

Example 1. Let $n = 4$, $f = x^3 - 7x + 7$ and $g = 3x^3 + 2x^2 + x + 1$. Then the cyclic convolution of the polynomials f and g (or the cyclic convolution of the vectors $\{7, -7, 0, 1\}$ and $\{1, 1, 2, 3\}$) is the polynomial

```
n = 4; fCoef = {7, -7, 0, 1}; gCoef = {1, 1, 2, 3};
powersX = Table[x^i, {i, 0, n - 1}];
powersX.ListConvolve[fCoef, gCoef, {1, 1}]
```

```
-13 + 2 x + 10 x^2 + 8 x^3
```

or the vector $\sum_{i=0}^{n-1} f_i\, g_{k-i}$, $0 \le k \le n-1$, (where arithmetic at the indices of g_{k-i} is done mod n)

```
Table[Sum[fCoef[[i + 1]] gCoef[[Mod[k - i, n] + 1]], {i, 0, n-1}], {k, 0, 3}]
```

```
{-13, 2, 10, 8}
```

Therefore, we obtain the same result with these three methods.

1. Use *Mathematica*'s function `ListConvolve`.
2. `ListConvolve[fCoef, gCoef, {1, 1}]`
3. $\{-13, 2, 10, 8\}$
4. Take successive inner products of the first row of the table with each one of the following rows. Note that we have reversed the order of g and appended its first 3:

$$\begin{pmatrix} 3 & 2 & 1 & 1 & 3 & 2 & 1 \\ & & & 7 & -7 & 0 & 1 \\ & & 7 & -7 & 0 & 1 & \\ & 7 & -7 & 0 & 1 & & \\ 7 & -7 & 0 & 1 & & & \end{pmatrix}.$$

5. Use the formula $f *_n g \equiv f\, g \bmod x^n - 1$.

```
PolynomialMod[(fCoef.powersX) (gCoef.powersX), x^n - 1]
```

```
-13 + 2 x + 10 x^2 + 8 x^3
```

■ Fast Fourier Transform for Fast Polynomial and Integer Multiplication

We begin by discussing a topic that is well known and much talked about, but for which there is little, if any at all, "hands-on" experience.

It is well known that a polynomial of degree less than n over an integral domain R, such as the integers or the rationals, can be represented either by its list of coefficients $\{f_0, \ldots, f_{n-1}\}$, taken in reverse order here, or by a list of its values at n distinct points $u_0, \ldots, u_{n-1} \in R$, where for $0 \leq i < n$ we have $u_i = \omega^i$; $\omega \in R$ is a primitive n^{th} root of unity.

The reason for considering the value representation is that multiplication in that representation is easy. To wit, if $\{f(u_0), \ldots, f(u_{n-1})\}$ and $\{g(u_0), \ldots, g(u_{n-1})\}$ are the values of two polynomials f and g, evaluated at n distinct points, with $\deg(f) + \deg(g) < n$, then the values of the product $f \cdot g$ at those points are $\{f(u_0) \cdot g(u_0), \ldots, f(u_{n-1}) \cdot g(u_{n-1})\}$. Hence, the cost of polynomial multiplication in the value representation is linear in the degree, whereas in the list of coefficients representation we do not know how to multiply in linear time.

Therefore, a fast way of doing multipoint evaluation and interpolation leads to a fast polynomial multiplication algorithm. Namely, evaluate the two input polynomials, multiply the results pointwise, and interpolate to get the product polynomial.

The multipoint evaluation is performed with FFT as implemented by the function `Fourier`, whereas interpolation is performed with the inverse FFT, implemented by the function `InverseFourier`.

Example 2. Suppose we are given the two polynomials $f(x) = x^3 - 7x + 7$ and $g(x) = 3x^2 - 7$, whose product we want to compute.

```
f[x_] = x^3 - 7 x + 7; g[x_] = 3 x^2 - 7;
```

```
f[x] g[x] // Expand
```

```
-49 + 49 x + 21 x^2 - 28 x^3 + 3 x^5
```

This is of degree $\deg(f) + \deg(g) = 5$.

We will now compute this product using FFT. Keeping in mind that FFT works best for inputs which are powers of 2, we consider the degree of the product to be less than $n = 8$.

Having fixed the value of n, we then form the lists of coefficients of f and g—padding them with zeros until their lengths equal 8.

```
n = 8;
flist = CoefficientList[f[x], x];
flist = PadRight[flist, n]

{7, -7, 0, 1, 0, 0, 0, 0}

glist = CoefficientList[g[x], x];
glist = PadRight[glist, n]

{-7, 0, 3, 0, 0, 0, 0, 0}
```

We next apply `Fourier` to these two lists and pointwise multiply the results.

```
productValues = Fourier[flist] Fourier[glist] // Chop

{-0.5, 0.415738 + 4.21599 ⅈ, -8.75 + 10. ⅈ, -12.6657 - 1.03401 ⅈ,
  -6.5, -12.6657 + 1.03401 ⅈ, -8.75 - 10. ⅈ, 0.415738 - 4.21599 ⅈ}
```

Recall, from Definition 1 and the verification following it, that what we have done here is equivalent, within a scaling factor, to evaluating each polynomial at the points $u_i = \omega^i$ (where $\omega = e^{\frac{2\pi i}{n}}$, $n = 8$) and pointwise multiplying the results.

Interpolating the result with `InverseFourier` and taking care of the scaling factor, we obtain the coefficients of the product polynomial.

```
productCoefficients =
  √n InverseFourier[productValues] // Chop // Rationalize

{-49, 49, 21, -28, 0, 3, 0, 0}
```

This is exactly what we obtained with the classical multiplication.

These ideas can be incorporated in an algorithm to do just polynomial multiplication. However, in order to avoid duplication of code—since integer FFT multiplication is very similar—we implement the function `generalFFTMultiply`, which will be used in both cases. This function is written in such a way that it computes in reverse order either the coefficients of the product of two polynomials with integer coefficients, or the integer digits—to a certain base β—of the product of two integers.

```
generalFFTMultiply[f_, g_, b_] := Module[
  {flist, glist, k = 1, m0, n, n0, productValues, var},
  If[Length[Variables[f]] ≠ 0,
   (* THEN polynomial degree > 1 *)
   var = First[Variables[f]];
   flist = CoefficientList[f, var];
   glist = CoefficientList[g, var];
   m0 = Exponent[f, x];
   n0 = Exponent[g, x],
   (*  ELSE this case is reserved  *)
   (*  for integer multiplication  *)
   flist = IntegerDigits[f, b] // Reverse;
   glist = IntegerDigits[g, b] // Reverse;
   m0 = Length[flist];
   n0 = Length[glist]];
  (*  treat polys and integers the same  *)
  While[2^k ≤ m0 + n0, ++k ]; n = 2^k;
  flist = PadRight[flist, n];
  glist = PadRight[glist, n];
  productValues = Fourier[flist] Fourier[glist] // Chop;
  √n InverseFourier[productValues] // Chop // Rationalize
 ]
```

So, to multiply the polynomials $f(x)$ and $g(x)$ we define the function

```
polyFFTMultiply[f_, g_] :=
 Module[{list}, (list = generalFFTMultiply[f, g, b]).
   Table[x^i, {i, 0, Length[list] - 1}]]
```

and their product is

```
polyFFTMultiply[f[x], g[x]]
```

$-49 + 49 x + 21 x^2 - 28 x^3 + 3 x^5$

The cost of doing polynomial multiplication this way is $O(n \log n)$ operations, which is the cost of computing the FFT and its inverse. This is a big improvement over the $O(n^2)$ cost of the classical algorithm.

Before we move on to integer multiplication it is worth mentioning that `ListConvolve` also gives us, in reverse order, the coefficient list of the product $f(x)\,g(x)$.

```
(list = ListConvolve[CoefficientList[f[x], x],
     CoefficientList[f'[x], x], {1, -1}, 0]).
 Table[x^i, {i, 0, Length[list] - 1}]
```

```
-49 + 49 x + 21 x^2 - 28 x^3 + 3 x^5
```

We next present the integer multiplication algorithm using FFT.

As we know, every integer can be represented as a "polynomial" in some base β, that is, for an integer a we have $a = (-1)^s \sum_{0\le i\le n} a_i\,\beta^i$. Therefore, integer multiplication can be considered as polynomial multiplication, where in the final result we replace the variable x by the base β.

Adjusting `polyFFTMultiply` accordingly we obtain this function.

```
integerFFTMultiply[f_Integer, g_Integer, b_ : 10] :=
 Module[{list},
  (list = generalFFTMultiply[f, g, b]).
    Table[x^i, {i, 0, Length[list] - 1}] /. x → b]
```

Then the product of the integers 123456789 and 987654321 is

```
integerFFTMultiply[123 456 789, 987 654 321]
```

```
121 932 631 112 635 269
```

■ Fast Fourier Transform Is the Basis of Fast Fourier Fit

We next turn our attention to the problem of FFF, that is, the problem of approximating functions with sines and/or cosines.

Definition 3. *Periodic functions $f:\mathbb{R}\to\mathbb{C}$, in one real variable and with values in the complex plane, can be approximated (or fitted) by complex trigonometric polynomials of the form*

$$f(t) = \sum_{k=-n}^{n} c_k\, e^{k\,\omega\, i\, t} = \frac{\alpha_0}{2} + \sum_{k=1}^{n} (\alpha_k \cos(k\,\omega\,t) + \beta_k \sin(k\,\omega\,t)),$$

where c_k are the Fourier fit coefficients satisfying

$$c_0 = \frac{\alpha_0}{2}, \qquad c_k = \frac{(\alpha_k - i\,\beta_k)}{2}, \qquad c_{-k} = \frac{(\alpha_k + i\,\beta_k)}{2}$$

and

$$\alpha_0 = 2\,c_0, \qquad \alpha_k = c_k + c_{-k}, \qquad \beta_k = i\,(c_k - c_{-k})$$

for $k = 1, \ldots, n$, *and* $\omega = \frac{2\pi}{L}$ *with* $L > 0$ [3].

The FFF problem has attracted the attention of some of the best scientific minds of all time. Gauss came up with an FFF algorithm in 1866. The modern version of the FFF is due to John Tukey and his cohorts at IBM and Princeton [4].

We will be using the function `FastFourierFit` taken from Bill Davis and Jerry Uhl's *Differential Equations & Mathematica* [1] to compute the approximating complex trigonometric polynomials mentioned in Definition 3.

```
jump[n_] := jump[n] = 1/(2 n);
Fvalues[F_, L_, n_] :=
  N[Table[F[L t], {t, 0, 1 - jump[n], jump[n]}]];

numtab[n_] := numtab[n] = Table[k, {k, 1, n}];

FourierFitters[L_, n_, t_] :=
  Table[E^(2 π I k t/L), {k, -n + 1, n - 1}];
coeffs[n_, list_] :=
  Join[Reverse[Part[Fourier[list], numtab[n]]],
    Part[InverseFourier[list], Drop[numtab[n], 1]]] /
      N[Sqrt[Length[list]]];

FastFourierFit[F_, L_, n_, t_] :=
  Chop[FourierFitters[L, n, t].coeffs[n, Fvalues[F, L, n]]];
```

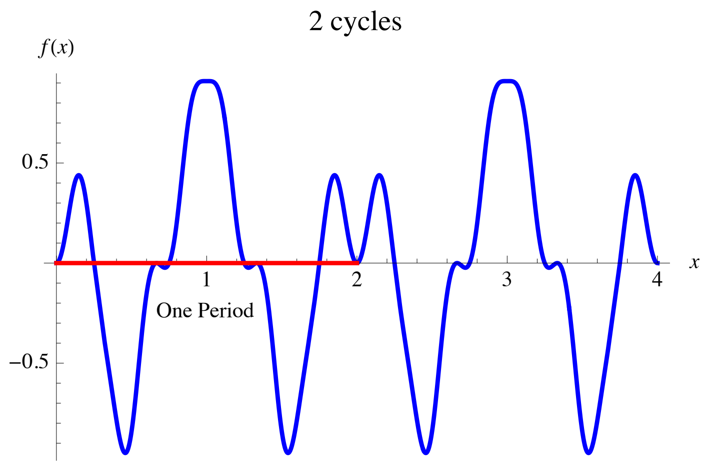
The code works as follows: the functions `jump` and `Fvalues` produce a list of $2\,n - 1$ equally spaced data points off the plot of the function $f(t)$ between $t = 0$ and $t = L$. Then, the function `numtab` creates a list of integers from 1 to n, which is used by `coeffs` to concatenate two lists. The first of these lists is the Fourier transform (taken in reversed order) of the first n points, while the second list is the inverse Fourier transform (with the first element removed) of the same n points. The list generated by `coeffs` has a total of $2\,n - 1$ points.

Finally, the function `FastFourierFit` takes the dot product of the list $\{e^{(-n+1)\,2\pi\,i\,t/L}, \ldots, 1, \ldots, e^{(n-1)\,2\pi\,i\,t/L}\}$ generated by `FourierFitters` and the list concatenated by `coeffs`. (All numbers in the list with magnitude less than 10^{-10} are rounded to 0.)

`FastFourierFit` takes four arguments: the first one is the periodic function or, in general, the list of data points which we want to fit; the second argument is the period L of the function; the third argument is the number n for the equally spaced $2n-1$ data points; and the last argument is the variable we want to use. Note that `FastFourierFit` uses the built-in functions `Fourier` and `InverseFourier`, with computational cost $n \log n$.

Example 3. To see how the function `FastFourierFit` is used, consider the periodic function $f(x) = \cos(2\pi x)\sin(1-\cos(3\pi x))$ with period $L = 2$. A plot is given in Figure 1.

```
f[x_] := Cos[2 π x] Sin[1 - Cos[3 π x]];
L = 2;
cycles = 2;
Plot[f[x], {x, 0, cycles L},
 AxesLabel → {x, HoldForm[f[x]]},
 PlotStyle → {{Thickness[0.007], RGBColor[0, 0, 1]}},
 PlotLabel → TextCell[ToString[cycles] ~~ " cycles",
   CellFrame → 0],
 Epilog →
  {{RGBColor[1, 0, 0], Thickness[0.007],
    Line[{{0, 0}, {L, 0}}]},
   {Text[TextCell["One Period", CellFrame → 0],
     {L/2, -0.25}]}}]
```



▲ **Figure 1.** The periodic function $f(x) = \cos(2\pi x)\sin(1-\cos(3\pi x))$.

Approximating $f(x)$ with $n = 4$ we obtain

```
L = 2; n = 4;
fApproximation[t_] = FastFourierFit[f, L, n, t]
```

$$-0.0967056 - 0.113662\, e^{-i\pi t} - 0.113662\, e^{i\pi t} + 0.32403\, e^{-2 i\pi t} + 0.32403\, e^{2 i\pi t} - 0.113662\, e^{-3 i\pi t} - 0.113662\, e^{3 i\pi t}$$

or its real (noncomplex) version

```
fApproximationReal[t_] =
 Chop[ComplexExpand[fApproximation[t]]]
```

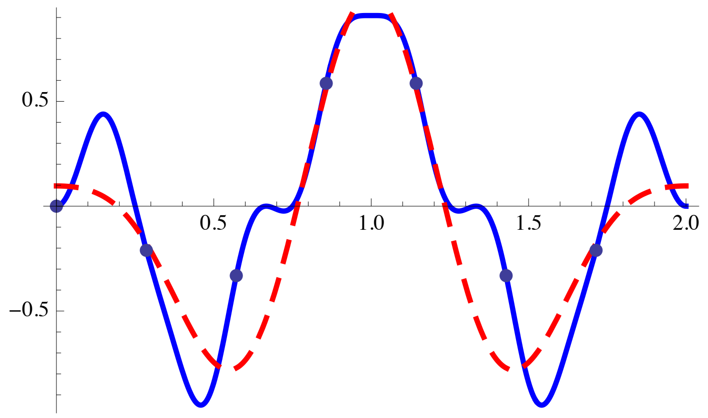
```
-0.0967056 - 0.227324 Cos[π t] +
 0.64806 Cos[2 π t] - 0.227324 Cos[3 π t]
```

Note that the coefficients of `fApproximation[t]` and `fApproximationReal[t]` satisfy the relations mentioned in Definition 3. Moreover, $f(x)$ has pure cosine fit. This was expected because the function $f(x) = \cos(2\pi x)\sin(1 - \cos(3\pi x))$ is even; that is, for the function `evenf[x]`, defined on the extended interval $0 \le x \le 2L$, we have `evenf[x] = f x, 0 ≤ x ≤ L`, and `evenf[x] = f (2 L - x), L < x ≤ 2 L`. See also its plot in Figure 1. Later on we will meet odd functions as well; those have pure sine fits.

The functions `f[x]` and `fApproximationReal[t]` are plotted together in Figure 2. As we see, `FastFourierFit` picks $2n - 1$ equally spaced data points off the plot of $f(x)$ between $x = 0$ and $x = L$; it then tries to fit these points with a combination of complex exponentials.

```
fplot = Plot[f[x], {x, 0, L},
   PlotStyle → {Thickness[0.008], RGBColor[0, 0, 1]},
   AspectRatio → 1/GoldenRatio];

fapproxPlot = Plot[fApproximationReal[t], {t, 0, L},
   PlotStyle →
    {{Thickness[0.008], RGBColor[1, 0, 0],
      Dashing[{0.03, 0.03}]}}, AspectRatio → 1/GoldenRatio];
```

```
fdata = Table[N[{x, f[x]}], {x, 0, L - L/(2 n - 1), L/(2 n - 1)}];
fdataplot = ListPlot[fdata, PlotStyle → PointSize[0.02]];
Show[fplot, fapproxPlot, fdataplot]
```



▲ **Figure 2.** The dashed red plot is that of the approximating function.

As we mentioned before, the coefficients c_k of the approximating polynomial in Definition 3 are computed using the FFT—incorporated in the function `FastFourierFit`. Another way of computing those coefficients is to use the integrals

$$c_k = \frac{1}{L}\int_0^L f(t)\, e^{-\frac{i k (2\pi) t}{L}}\, dt,$$

which results in the integral Fourier fit.

This formula for the coefficients is obtained if we assume that for a fixed n, the function $f(t)$ is being approximated by the function

$$complexApproximation(t) = \sum_{k=-n}^{n} c_k\, e^{\frac{k(2\pi) i t}{L}},$$

where $L > 0$, and we set

$$f(t) = complexApproximation(t).$$

Then, we will definitely have

$$\int_0^L complexApproximation(t)\, e^{-\frac{j(2\pi) i t}{L}}\, dt = \int_0^L f(t)\, e^{-\frac{j(2\pi) i t}{L}}\, dt.$$

But

$$\int_0^L complexApproximation(t)\, e^{-\frac{j(2\pi) i t}{L}}\, dt = L\, c_j$$

and, hence, the formula for the coefficients.

The two approximations resulting from the FFF and the integral Fourier fit are fairly close, and almost identical for large values of n.

The disadvantage of the integral Fourier fit is that the integrals that need to be computed sometimes are very hard and impracticable even for numerical integration. Nonetheless, the method is useful for hand computations, whereas doing FFF by hand is completely out of the question.

The advantage of the integral Fourier fit is that, in theoretical situations, it provides a specific formula to work with. However, after the theory is developed and calculations begin, people switch to the FFF.

Recapping, note that `FastFourierFit` is a "double" approximation. It first uses sines and cosines to approximate a continuous periodic function and then uses discrete Fourier transform to approximate integrals involving these trigonometric polynomials—in effect replacing numerical integration by sampling.

■ Fast Fourier Fit Meets Laplace Transform

We recall that the Laplace transform of a given function $f(t)$ is another function $F(s)$ given by $F(s) = \int_0^\infty e^{-s t} f(t)\, dt$. The functions appropriate for the Laplace transform are all functions $f(t)$ with the property that $e^{-s t} f(t) \to 0$ as $t \to \infty$ for large positive s. The functions $\sin(p\, t)$, $\cos(p\, t)$, $e^{k t}$, $\log(t)$, as well as any quotient of polynomials, are all appropriate candidates for the Laplace transform.

For instance, here is the Laplace transform of $f(t) = t$.

```
f[t_] := t;
F[s_] = LaplaceTransform[f[t], t, s]
```

$$\frac{1}{s^2}$$

Indeed:

```
Assuming[s > 0, ∫_0^∞ ⅇ^(-s t) f[t] ⅆt]
```

$$\frac{1}{s^2}$$

This is *Mathematica*'s way of saying that if s is real and $s > 0$ then the Laplace transform of $f(t)$ is $\int_0^\infty e^{-s\,t} f(t)\,dt = \frac{1}{s^2}$. On the other hand, if *Mathematica* is given the Laplace transform of $f(t)$, it can often recover the formula for $f(t)$:

```
InverseLaplaceTransform[F[s], s, t]
```

```
t
```

Laplace transforms are used in solving differential equations by algebraic means. Suppose, for example, that we are given the differential equation $y''(x) + b\,y'(x) + c\,y(x) = f(x)$, with starting values $y(0)$ and $y'(0)$:

```
Clear[y, t, f, b, c, s, Y];
diffeq = y''[t] + b y'[t] + c y[t] == f[t]
```

```
c y[t] + b y′[t] + y″[t] == f[t]
```

The solution $y(t)$ of this differential equation can be found algebraically if we replace all the functions involved in it by their Laplace transforms. In this way, we obtain the equation

```
laplaced = diffeq /.
    {y[t] → LaplaceTransform[y[t], t, s],
    y'[t] → LaplaceTransform[y'[t], t, s],
    y''[t] → LaplaceTransform[y''[t], t, s],
    f[t] → LaplaceTransform[f[t], t, s]}
```

```
c LaplaceTransform[y[t], t, s] +
  s^2 LaplaceTransform[y[t], t, s] +
  b (s LaplaceTransform[y[t], t, s] - y[0]) - s y[0] - y′[0] ==
 LaplaceTransform[f[t], t, s]
```

and solve it for the Laplace transform of $y(t)$ to obtain the formula:

```
sol = Solve[laplaced, LaplaceTransform[y[t], t, s]]
```

```
{{LaplaceTransform[y[t], t, s] →     1
                                  ----------
                                  c + b s + s^2
   (LaplaceTransform[f[t], t, s] + b y[0] + s y[0] + y′[0])}}
```

This tells us that if $F(s)$ and $Y(s)$ are the Laplace transforms of the functions $f(t)$ and $y(t)$, respectively, then $Y(s) = \frac{F(s)+b\,y(0)+s\,y(0)+y'(0)}{s^2+b\,s+c}$. The solution of the differential equation $y(t)$ can be obtained by taking the inverse Laplace transform of $Y(s)$—which is possible in many cases.

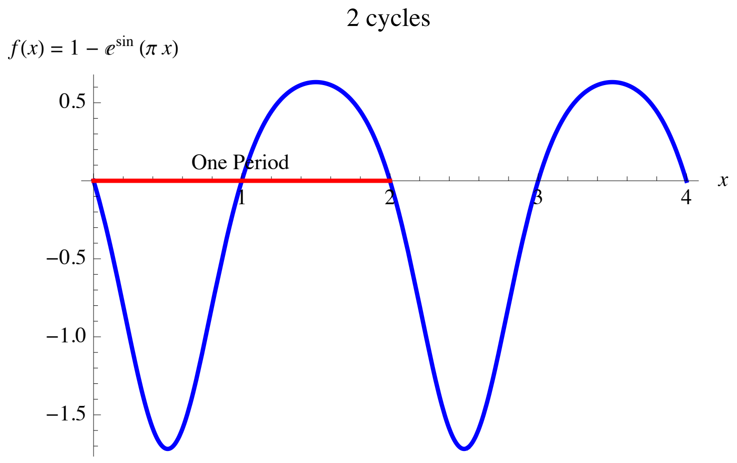
In this section we combine `FastFourierFit` and Laplace transforms to come up with good approximate formulas for periodically forced oscillators. To our knowledge, save for the work by Bill Davis and Jerry Uhl [1], this topic is totally absent from textbooks on differential equations! As a matter of fact, Fourier transforms, when discussed at all, appear only when dealing with the heat and wave equations [5].

Recall that the differential equation of the form $y''(x) + b\, y'(x) + c\, y(x) = f(x)$, with given starting values $y(0)$ and $y'(0)$, and $f(x)$ a periodic function, can be solved either by evaluating a convolution integral of $f(x)$ or by taking its Laplace transform. However, in both cases, it may happen that the integrals involving $f(x)$ are too complicated and *Mathematica* (or any other similar computer algebra package) cannot handle them.

What we want to do then is to first find a good FFF of $f(x)$ (using sines and/or cosines) and then to use any of the methods mentioned to get an approximate formula of the solution. That is, instead of solving $y''(x) + b\, y'(x) + c\, y(x) = f(x)$ we will be solving the differential equation `y'' t + b y' t + c y t = fApproximationReal[t]` with the starting values $y(0)$ and $y'(0)$.

Example 4. Let us say that we have to solve the differential equation $y''(x) + 2\, y'(x) + 20\, y(x) = 1 - e^{\sin(\pi x)}$ with $y(0) = 2$ and $y'(0) = -4$. The periodic function $f(x) = 1 - e^{\sin(\pi x)}$ can be seen in Figure 3.

```
f[x_] := 1 - e^Sin[π x];
L = 2;
cycles = 2;
Plot[f[x], {x, 0, cycles L},
 AxesLabel → {x, HoldForm[f[x] = 1 - E^sin (π x)]},
 PlotStyle → {{Thickness[0.007], RGBColor[0, 0, 1]}},
 PlotLabel → TextCell[ToString[cycles] ~~ " cycles",
   CellFrame → 0],
 Epilog →
  {{RGBColor[1, 0, 0], Thickness[0.007],
    Line[{{0, 0}, {L, 0}}]},
   {Text[TextCell["One Period", CellFrame → 0],
     {L / 2, 0.1}]}}]
```



▲ **Figure 3.** The periodic function $f(x) = 1 - e^{\sin(\pi x)}$.

It is impossible to find an exact solution of $y''(x) + 2\,y'(x) + 20\,y(x) = 1 - e^{\sin(\pi x)}$. *Mathematica*'s built-in function `DSolve` bogs down because the integrals are too complicated.

```
DSolve[{y''[x] + 2 y'[x] + 20 y[x] == 1 - E^Sin[π x], y[0] == 2,
    y'[0] == -4}, y, x] // AbsoluteTiming
```

```
{247.069239,
```

$$\left\{\left\{y \to \text{Function}\left[\{x\}, -\frac{1}{19} e^{-x}\left(-38\,\text{Cos}\left[\sqrt{19}\,x\right] + 19\,\text{Cos}\left[\sqrt{19}\,x\right]\int_1^0 \frac{e^{K[2]}\left(-1+e^{\text{Sin}[\pi K[2]]}\right)\text{Sin}\left[\sqrt{19}\,K[2]\right]}{\sqrt{19}}\,dK[2] - 19\,\text{Cos}\left[\sqrt{19}\,x\right]\int_1^x \frac{e^{K[2]}\left(-1+e^{\text{Sin}[\pi K[2]]}\right)\text{Sin}\left[\sqrt{19}\,K[2]\right]}{\sqrt{19}}\,dK[2] + 2\sqrt{19}\,\text{Sin}\left[\sqrt{19}\,x\right] + 19\left(\int_1^0 -\frac{e^{K[1]}\left(-1+e^{\text{Sin}[\pi K[1]]}\right)\text{Cos}\left[\sqrt{19}\,K[1]\right]}{\sqrt{19}}\,dK[1]\right)\text{Sin}\left[\sqrt{19}\,x\right] - 19\left(\int_1^x -\frac{e^{K[1]}\left(-1+e^{\text{Sin}[\pi K[1]]}\right)\text{Cos}\left[\sqrt{19}\,K[1]\right]}{\sqrt{19}}\,dK[1]\right)\text{Sin}\left[\sqrt{19}\,x\right]\right)\right]\right\}\right\}\right\}$$

As mentioned earlier, what we do in such cases is to first find `fApproximationReal[t]`, a good FFF of $f(x) = 1 - e^{\sin(\pi x)}$.

```
fApproximationReal[t_] =
 Chop[ComplexExpand[FastFourierFit[f, L = 2, n = 4, t] ]]

-0.266066 + 0.27154 Cos[2 π t] -
 1.13032 Sin[π t] + 0.0448798 Sin[3 π t]
```

Then, we easily obtain `LTyApproximation[s]`, the Laplace transform of the approximate formula of the solution of `y'' t + 2 y' t + 20 y t = fApproximationReal[t]` with $y(0) = 2$ and $y'(0) = -4$.

```
b = 2; c = 20;
ystarter = 2; yprimestarter = -4;
LTyApproximation[s_] =
    1
 ────────
 s^2 + b s + c
  (LaplaceTransform[fApproximationReal[t], t, s] +
    2 ystarter + s ystarter + yprimestarter)
```

$$\frac{-\frac{0.266066}{s} + 2\,s - \frac{3.55101}{\pi^2+s^2} + \frac{0.27154\,s}{4\,\pi^2+s^2} + \frac{0.422982}{9\,\pi^2+s^2}}{20 + 2\,s + s^2}$$

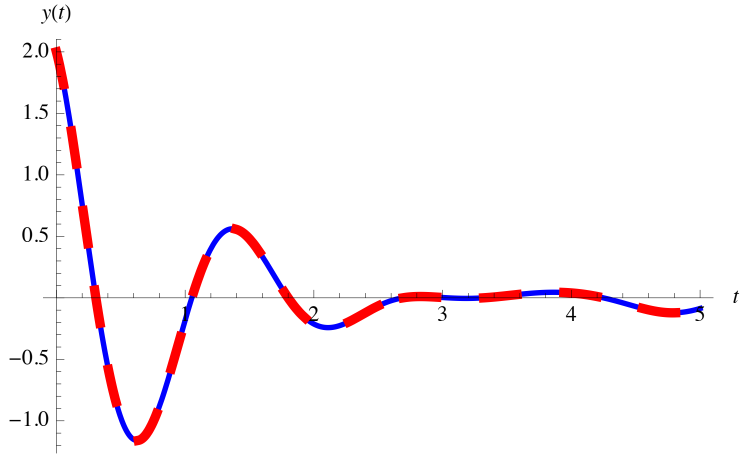
Finally, the formula for the approximate solution `yApproximation[t]` is obtained using the inverse Laplace transform.

```
yApproximation[t_] = Chop[ComplexExpand[
   InverseLaplaceTransform[LTyApproximation[s], s, t]
  ]]

-0.0133033 + 0.0249889 Cos[3.14159 t] +
 0.986668 e^(-1. t) Cos[4.3589 t] - 0.00492179 Cos[6.28319 t] +
 0.0249889 Cos[3.14159 t] Cos[6.28319 t] +
 0.986668 e^(-1. t) Cos[4.3589 t] Cos[8.7178 t] -
 0.0000830617 Cos[9.42478 t] -
 0.00492179 Cos[6.28319 t] Cos[12.5664 t] -
 0.0000830617 Cos[9.42478 t] Cos[18.8496 t] -
 0.0402897 Sin[3.14159 t] +
 0.0402897 Cos[6.28319 t] Sin[3.14159 t] -
 0.207358 e^(-1. t) Sin[4.3589 t] +
 0.207358 e^(-1. t) Cos[8.7178 t] Sin[4.3589 t] +
 0.00317526 Sin[6.28319 t] -
 0.0402897 Cos[3.14159 t] Sin[6.28319 t] -
 0.00317526 Cos[12.5664 t] Sin[6.28319 t] +
 0.0249889 Sin[3.14159 t] Sin[6.28319 t] -
```

```
0.207358 ⅇ^(-1. t) Cos[4.3589 t] Sin[8.7178 t] +
0.986668 ⅇ^(-1. t) Sin[4.3589 t] Sin[8.7178 t] -
0.000303288 Sin[9.42478 t] +
0.000303288 Cos[18.8496 t] Sin[9.42478 t] +
0.00317526 Cos[6.28319 t] Sin[12.5664 t] -
0.00492179 Sin[6.28319 t] Sin[12.5664 t] -
0.000303288 Cos[9.42478 t] Sin[18.8496 t] -
0.0000830617 Sin[9.42478 t] Sin[18.8496 t] +
ⅈ (-0.0402897 Cos[3.14159 t] -
   0.207358 ⅇ^(-1. t) Cos[4.3589 t] + 0.00317526 Cos[6.28319 t] +
   0.0402897 Cos[3.14159 t] Cos[6.28319 t] + 0.207358 ⅇ^(-1. t)
    Cos[4.3589 t] Cos[8.7178 t] - 0.000303288 Cos[9.42478 t] -
   0.00317526 Cos[6.28319 t] Cos[12.5664 t] + 0.000303288
    Cos[9.42478 t] Cos[18.8496 t] - 0.0249889 Sin[3.14159 t] -
   0.0249889 Cos[6.28319 t] Sin[3.14159 t] - 0.986668 ⅇ^(-1. t)
    Sin[4.3589 t] - 0.986668 ⅇ^(-1. t) Cos[8.7178 t] Sin[4.3589 t] +
   0.00492179 Sin[6.28319 t] + 0.0249889 Cos[3.14159 t]
    Sin[6.28319 t] + 0.00492179 Cos[12.5664 t] Sin[6.28319 t] +
   0.0402897 Sin[3.14159 t] Sin[6.28319 t] +
   0.986668 ⅇ^(-1. t) Cos[4.3589 t] Sin[8.7178 t] +
   0.207358 ⅇ^(-1. t) Sin[4.3589 t] Sin[8.7178 t] +
   0.0000830617 Sin[9.42478 t] + 0.0000830617 Cos[18.8496 t]
    Sin[9.42478 t] - 0.00492179 Cos[6.28319 t] Sin[12.5664 t] -
   0.00317526 Sin[6.28319 t] Sin[12.5664 t] -
   0.0000830617 Cos[9.42478 t] Sin[18.8496 t] +
   0.000303288 Sin[9.42478 t] Sin[18.8496 t])
```

In Figure 4 we compare the plot of the "unknown" solution obtained by `NDSolve` with the plot of the approximate formula for the solution (red, thicker dashed line). They are identical!

```
sol = NDSolve[{y''[x] + 2 y'[x] + 20 y[x] == 1 - E^Sin[π x],
    y[0] == 2, y'[0] == -4}, y, {x, 0, 5}];

Plot[{y[t] /. sol, yApproximation[t]}, {t, 0, 5},
 PlotRange → All,
 PlotStyle → {Directive[Thickness[0.008], RGBColor[0, 0, 1]],
   Directive[Thickness[0.014], RGBColor[1, 0, 0],
    Dashing[{0.05, 0.07}]]}, AxesLabel → {t, y[t]},
 AspectRatio → 1 / GoldenRatio]
```



▲ **Figure 4.** The red dashed line is the approximate solution.

■ Fast Fourier Fit for Discovering Trigonometric Identities

Another interesting application of `FastFourierFit` is in helping us "discover" trigonometric identities. Again, to our knowledge, this is mentioned only in the exercises of the work by Bill Davis and Jerry Uhl [1].

We know, for example, the trigonometric identity $2\sin(a)\sin(b) = \cos(a-b) - \cos(a+b)$. Suppose for the moment that this identity is unknown to us and that we are faced with the expression $\sin(3\,t)\sin(7\,t)$. How can we simplify it? Of course we can use *Mathematica*'s built-in function

```
TrigReduce[Sin[3 t] Sin[7 t]]
```

$$\frac{1}{2}\,(\mathrm{Cos}[4\,t]-\mathrm{Cos}[10\,t])$$

but let us write our own `trigIdentityFinder` function using FFF.

Our function `trigIdentityFinder` is based on `FastFourierFit`, which is used to approximate $\sin(3\,t)\sin(7\,t)$ for various values of n, until the result no longer changes. The final result is then the desired identity. So we have

```
trigIdentityFinder[f_] :=
 Module[{L = 2 π, old = 0, n = 2, new},
  new = Chop[ComplexExpand[FastFourierFit[f, L, n, t]]];
  While[! Chop[new - old] === 0, old = new; ++n;
   new = Chop[ComplexExpand[FastFourierFit[f, L, n, t]]]];
  Print[n - 1, "  iterations and the identity is:   ",
   f[t], " =  "]; Factor[Rationalize[new]]]
```

and the required identity for the problem at hand is found in 11 iterations.

```
f[t_] = Sin[3 t] Sin[7 t];
trigIdentityFinder[f]
```

```
11  iterations and the identity is:   Sin[3 t] Sin[7 t] =
```

$$\frac{1}{2}(\text{Cos}[4\,t] - \text{Cos}[10\,t])$$

We end this subject with one more problem from [1], comparing our identity with the result obtained from *Mathematica*.

```
f[t_] = Sin[t]^12; trigIdentityFinder[f]
```

```
13  iterations and the identity is:   Sin[t]^12 =
```

$$\frac{1}{2048}(462 - 792\,\text{Cos}[2\,t] + 495\,\text{Cos}[4\,t] - 220\,\text{Cos}[6\,t] + 66\,\text{Cos}[8\,t] - 12\,\text{Cos}[10\,t] + \text{Cos}[12\,t])$$

```
TrigReduce[f[t]]
```

$$\frac{1}{2048}(462 - 792\,\text{Cos}[2\,t] + 495\,\text{Cos}[4\,t] - 220\,\text{Cos}[6\,t] + 66\,\text{Cos}[8\,t] - 12\,\text{Cos}[10\,t] + \text{Cos}[12\,t])$$

■ Conclusions

Our goal has been to put together several difficult to access applications of the fast Fourier transform (FFT) for use in the classroom. Hopefully, the programs provided here will be of help for experimentation and further development.

■ Acknowledgments

We would like to thank two unknown referees for their most helpful comments which improved our presentation.

■ References

[1] B. Davis and J. Uhl, *Differential Equations & Mathematica*, Gahanna, OH: Math Everywhere, Inc., 1999. Part of the *Calculus & Mathematica* series of books.

[2] H. J. Weaver, *Applications of Discrete and Continuous Fourier Analysis*, New York: John Wiley & Sons, 1983.

[3] W. Strampp, V. G. Ganzha, and E. Vorozhtsov, *Höhere Mathematik mit Mathematica*, Band 4: Funktionentheorie, Fouriertransformationen and Laplacetransformationen, Braunschweig/Wiesbaden: Vieweg Lehrbuch Computeralgebra, 1997.

[4] D. K. Kahaner, C. Moler, and S. Nash, *Numerical Methods and Software*, Englewood Cliffs, NJ: Prentice Hall, 1989.

[5] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 6th ed., New York: John Wiley & Sons, 1997.

A. G. Akritas, J. Uhl, and P. S. Vigklas, “On Some Applications of the Fast Discrete Fourier Transform,” *The Mathematica Journal*, 2011. dx.doi.org/10.3888/tmj.11.1-2.

About the Authors

Alkiviadis G. Akritas taught at the University of Kansas for twenty years before he moved to Greece, where he has been teaching and doing research in the Department of Computer and Communication Engineering at the University of Thessaly, in Volos, since 1998. His research interests are in the field of symbolic and algebraic computations (a field in which he has published extensively) and in using computer algebra systems to improve the teaching of mathematics. Based on Vincent’s theorem of 1836, Akritas has developed the two fastest methods for isolating the real roots of polynomial equations; these methods have been incorporated, respectively, in the computer algebra systems Maple and *Mathematica*.

Jerry Uhl is a professor of mathematics at the University of Illinois at Urbana-Champaign. He is the author or coauthor of a number of research papers. During the 1980s, Uhl served as real analysis editor of the research journal *Proceedings of the American Mathematical Society*. He also served one term as managing editor of the same journal, as well as one term on the Council of the American Mathematical Society. Since 1988, Uhl has devoted nearly all his energies to Calculus&*Mathematica*. In 1998, he received an award for distinguished teaching from the Mathematical Association of America.

Panagiotis S. Vigklas is a Ph.D student in the Department of Computer and Communication Engineering at the University of Thessaly, in Volos. He is currently working on his dissertation under the supervision of A. G. Akritas.

Alkiviadis G. Akritas
University of Thessaly
Department of Computer and Communication Engineering
37 Glavani & 28th October
GR-38221, Volos
Greece
akritas@uth.gr
inf-server.inf.uth.gr/~akritas/index.html

Jerry Uhl
University of Illinois at Urbana-Champaign
Department of Mathematics
273 Altgeld Hall (mc 382)
1409 W. Green
Urbana, IL 61801
USA
juhl@cm.math.uiuc.edu

Panagiotis S. Vigklas
University of Thessaly
Department of Computer and Communication Engineering
37 Glavani & 28th October
GR-38221, Volos
Greece
pviglas@uth.gr