On Some Applications of the (Fast) Discrete Fourier Transform

Alkiviadis G. Akritas†, Jerry Uhl‡ and Panagiotis S. Vigklas†

†University of Thessaly
Department of Computer and Communication Engineering
37 Glavani & 28th October
GR-38221, Volos
Greece
and
‡University of Illinois at Urbana-Champaign
Department of Mathematics
273 Altgeld Hall (mc 382)
1409 W. Green
Urbana, IL 61801
USA
akritas@uth.gr, juhl@cm.math.uiuc.edu, pviglas@uth.gr

Motivated by the excellent work of Bill Davis and Jerry Uhl "Differential Equations & *Mathematica*" [2], we present in detail several little known applications of the *f*ast *D*iscrete *F*ourier *T*ransform (DFT), also known as FFT. Namely, we first examine the use of FFT in: (a) multiplying univariate polynomials and integers, and (b) approximating polynomials with sines and cosines (also known as *f*ast *F*ourier *f*it or FFF). We then examine the use of the fast Fourier fit in: (c) solving differential equations with Laplace transforms, and (d) "discovering" trigonometric identities.

■ Introduction

We begin with a review of the basic definition needed.

Let R be a ring, $n \in \mathbb{Z}_{\geq 1}$, 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 $< n$ 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, i.e. $\mathrm{DFT}_\omega: \{f_0, \ldots, f_{n-1}\} \to \frac{1}{\sqrt{n}}\{f(1), f(\omega), \ldots, f(\omega^{n-1})\}$, is called the *Discrete Fourier Transform* (DFT).

In other words, the Discrete Fourier Transform 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 [5]. Keeping it simple, we mention in passing that

the **inverse** Discrete Fourier Transform 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 fast Fourier transform 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
```

■ FFT and 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 $0 \le i < n$ we have $u_i = \omega^i$; $\omega \in R$ is a primitive nth 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, with $\deg(f) + \deg(g) < n$, evaluated at n distinct points, 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, the fast Fourier transform, implemented by the function `Fourier[]`, whereas interpolation is performed with the inverse FFT, implemented by the function `InverseFourier[]`.

Example 1: 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;
```

Their product (computed the classical way) is

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

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

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 0's until their lengths equal $n = 8$.

```
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: (**a**) evaluating each polynomial at the points $u_i = \omega^i$, where $\omega = e^{\frac{2\pi i}{n}}$, $n = 8$, and (**b**) 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}
```

Exactly what we obtained with the classical multiplication.

The above 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 (a) the coefficients of the product of two polynomials with integer coefficients, or (b) 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. 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, to wit, 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 the 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 integers123456789 and 987654321 is

```
integerFFTMultiply[123456789, 987654321]
```

```
121932631112635269
```

■ FFT is the basis of fast Fourier fit (FFF)

We next turn our attention to the problem of fast Fourier fit or FFF, i.e. the problem of approximating functions with sines and/or cosines.

Definition 2: 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

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

and

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

for $k = 1, \ldots, n$, and $\omega = \frac{2\pi}{L}$ with $L > 0$ [4].

The problem of fast Fourier fit has attracted the attention of some of the best scientific minds of all times. Gauss came up with a fast Fourier fit algorithm in 1866. The modern version of the fast Fourier fit is due to John Tukey and his cohorts at IBM and Princeton [3].

We will be using the function FastFourierFit[] taken from Bill Davis and Jerry Uhl "Differential Equations & *Mathematica*" [2] to compute the approximating complex trigonometric polynomials mentioned in Definition 2 above.

```
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 $2n$ - 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. To wit, the list generated by `coeffs[]` has a total of $2n$ - 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 2: 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", "f(x)"},
  PlotStyle → {{Thickness[0.007], RGBColor[0, 0, 1]}},
  PlotLabel → "cycles" cycles,
  Epilog → {{RGBColor[1, 0, 0],
     Thickness[0.007], Line[{{0, 0}, {L, 0}}]},
    {Text["One Period", {L/2, 0.1}]}}];
```

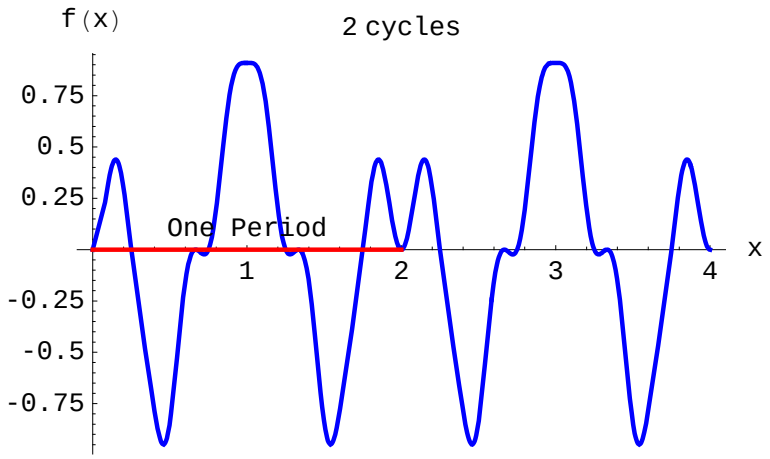

Figure 1. Approximating $f(x)$ with $n = 4$ we obtain

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

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

or its real (non-complex) version

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

$$-0.0967056 - 0.227324\,\text{Cos}[\pi t] + 0.64806\,\text{Cos}[2\pi t] - 0.227324\,\text{Cos}[3\pi t]$$

Please note that the coefficients of *fApproximation*(*t*) and *fApproximationReal*(*t*) satisfy the relations mentioned in Definition 2. 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*), $0 \le x \le L$, and *evenf*(*x*) = *f*(2*L* - *x*), $L < x \le 2L$. 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, what FastFourierFit[] does is to pick $2n - 1$ 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,
   DisplayFunction :→ Identity];

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

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

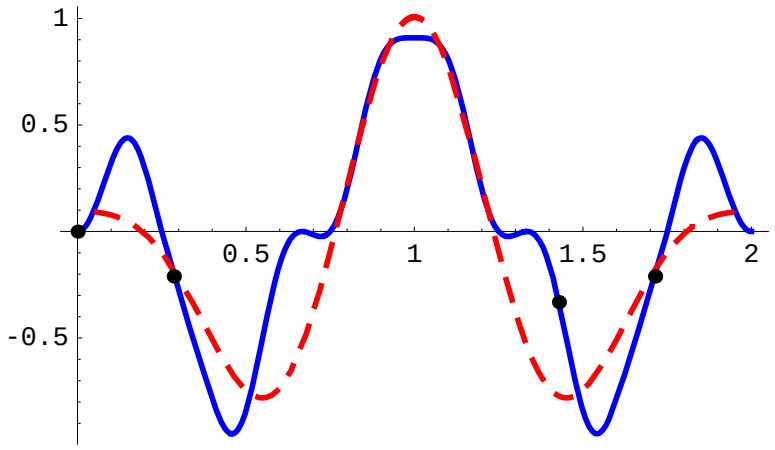

Figure 2. The dashed red plot is that of the approximating function.

As we mentioned before, the coefficients c_k of the approximating polynomial in Definition 2 are computed using the fast Fourier transform—incorporated in the function FastFourierFit[]. Another way of computing those coefficients is using the integrals

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

This 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

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

where $L > 0$, and we set

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

Then, we will definitely have

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

But

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

and, hence, the formula for the coefficients.

The two approximations resulting from the fast Fourier fit and the integral Fourier fit are pretty 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 inpractical even for numerical integration. Nonetheless, the method is useful for hand computations, whereas doing fast Fourier fit 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 fast Fourier fit.

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.

■ FFF meets Laplace transform

We recall that the Laplace transform of a given function $f(t)$ is another function, $F(s)$, given by $F(s) = \int_0^\infty e^{-st} f(t)\, dt$. The functions appropriate for the Laplace transform are all functions $f(t)$ with the property that $e^{-st} f(t) \to 0$ as $t \to \infty$ for large positive s. The functions $\sin(pt)$, $\cos(pt)$, e^{kt}, $\log(t)$ 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 $\int_0^\infty e^{-st} f(t)\, dt = \frac{1}{s^2}$. 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 albegraic means. Suppose, for example, that we are given the differential equation $y''(x) + by'(x) + cy(x) = f(x)$, 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 the above differential equation can be found algebraically if we replace all the functions involved in it by their Laplace transforms; 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]
```

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

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

$$\left\{\left\{\text{LaplaceTransform}[y[t], t, s] \to \frac{1}{c + b\,s + s^2}\right.\right.$$
$$\left.\left.(\text{LaplaceTransform}[f[t], t, s] + b\,y[0] + s\,y[0] + y'[0])\right\}\right\}$$

This tells us that if $F[s]$ and $Y[s]$ are the Laplace transforms of the functions $f(t)$ and $y(t)$, respectively, then $Y(s) = \frac{F(s)+by(0)+sy(0)+y'(0)}{s^2+bs+c}$. 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[] (FFF) and Laplace transform to come up with good approximate formulas for periodically forced oscillators. To our knowledge, save for the work by Bill Davis and Jerry Uhl [2], 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 [1].

Recall that the differential equation of the form $y''(x) + by'(x) + cy(x) = f(x)$, 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 Laplace transforming it. 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 (fast Fourier fit) of $f(x)$ (using sines and/or cosines) and then to use any of the methods mentioned above to get an approximate formula of the solution. That is, instead of solving $y''(x) + by'(x) + cy(x) = f(x)$ we will be solving the differential equation $y''(t) + by'(t) + cy(t) = fApproximationReal(t)$ with starting values $y(0)$ and $y'(0)$.

Example 3: Let us say that we have to solve the differential equation $y''(x) + 2y'(x) + 20y(x) = 1 - e^{\sin(\pi x)}$ 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", "f(x) = 1-e^sin(π x)"},
 PlotStyle → {{Thickness[0.007], RGBColor[0, 0, 1]}},
 PlotLabel → "cycles" cycles, Epilog → {{RGBColor[1, 0, 0],
    Thickness[0.007], Line[{{0, 0}, {L, 0}}]},
   {Text["One Period", {L/2, 0.1}]}}];
```

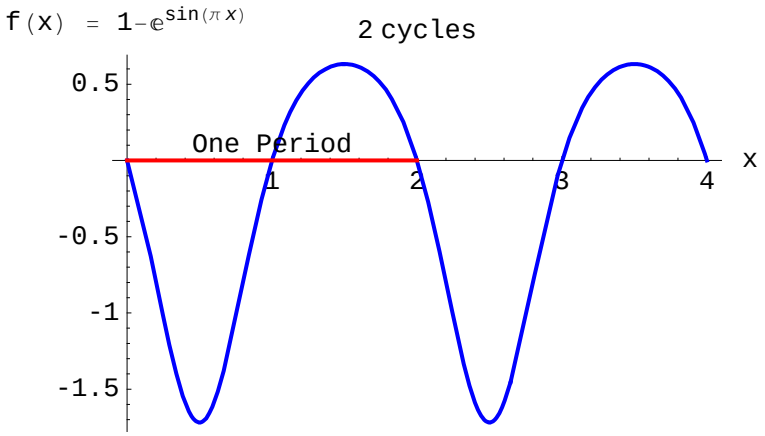

Figure 3. The periodic function $f(x) = 1 - e^{\sin(\pi x)}$.

It is impossible to find an exact solution of $y''(x) + 2y'(x) + 20y(x) = 1 - e^{\sin(\pi x)}$. *Mathematica*'s build-in function `DSolve[]` bogs down (*Mathematica* 5.0). 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
```

As we mentioned above, what we do in such cases is to first find *fApproximationReal*(*t*), a good FFF (fast Fourier fit) 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 $y''(t) + 2y'(t) + 20y(t) = fApproximationReal(t)$ 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, *yApproximation*(*t*), the formula for the approximate solution, is obtained using the Inverse Laplace Transform.

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

```
-0.0133033 + 0.0499778 Cos[3.14159 t] +
 1.97334 E^(-1. t) Cos[4.3589 t] - 0.00984358 Cos[6.28319 t] -
 0.000166123 Cos[9.42478 t] - 0.0805794 Sin[3.14159 t] -
 0.414715 E^(-1. t) Sin[4.3589 t] +
 0.00635052 Sin[6.28319 t] - 0.000606575 Sin[9.42478 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 → {{Thickness[0.008], RGBColor[0, 0, 1]},
    {Thickness[0.014], RGBColor[1, 0, 0],
     Dashing[{0.05, 0.07}]}},
  AxesLabel → {"t", "y(t)"}, AspectRatio → 1/GoldenRatio];
```

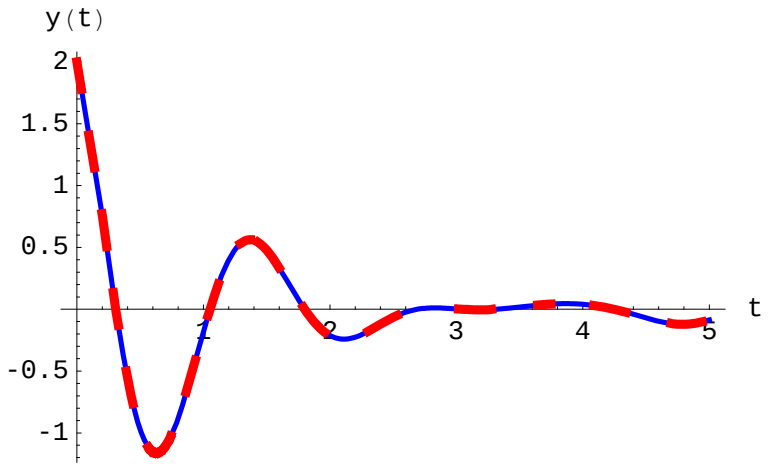

Figure 4. The red dashed line is the approximate solution.

■ FFF in "discovering" trigonometric identities

Another interesting application of FastFourierFit[] (FFF) 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 "Differential Equations & *Mathematica*" [2].

We know for example the trigonometric identity 2sin(*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]]
```

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

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 [2], 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 FFT for use in the classroom. Hopefully, the programs provided here will be of help for experimentation and further development.

■ Acknowledgements

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

■ References

1. William E. Boyce and Richard C. DiPrima: *Elementary Differential Equations and Boundary Value Problems*, John Wiley & Sons, New York, NY, 1997.

2. Bill Davis and Jerry Uhl: *Differential Equations & Mathematica*, Math Everywhere, Inc., 1999 (Part of the "Calculus & Mathematica" series of books).

3. David Kahaner, Cleve Moler and Stephen Nash: *Numerical Methods and Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

4. Walter Strampp, Victor Ganzha and Evgenij Vorozhtsov: *Höhere Mathematik mit Mathematica*, Vieweg Lehrbuch Computeralgebra, Braunschweig/Wiesbaden, 1997.

5. H. Joseph Weaver: *Applications of Discrete and Continuous Fourier Analysis*, John Wiley & Sons, New York, NY, 1983.