Lecture 17: Function Representation by Fourier Series

Reading:
Kreyszig Sections: 11.1, 11.2, 11.3

Periodic Functions

Periodic functions should be familiar to everyone. The keeping of time, the ebb and flow of tides, the patterns and textures of our buildings, decorations, and vestments invoke repetition and periodicity that seem to be inseparable from the elements of human cognition.[9] Although other species utilize music for purposes that we can only imagine—we seem to derive emotion and enjoyment from making and experience of music.

[9] I hope you enjoy the lyrical quality of the prose. While I wonder again if anyone is reading these notes, my wistfulness is taking a poetic turn:

They repeat themselves
What is here, will be there
It wills, willing, to be again
spring; neap, ebb and flow, wane; wax
sow; reap, warp and woof, motif; melody.
The changed changes. We remain
Perpetually, Immutably, Endlessly.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Playing with Audible Periodic Phenomena

Several example of creating sounds using mathematical functions are illustrated for education and amusement.

Sounds will not be available on PDF or HTML versions

Let's begin by "looking" at a familiar periodic phenomena:

We index the notes and write an indexed set of frequency (in Hertz) for each of the notes for one octave above middle-c. We write a function to create a Sound for each note.

```
c = 1; d = 2; e = 3; f = 4; g = 5; a = 6; b = 7;
freq[c] = 261.6;
freq[d]  = 293.7;
freq[e] = 329.6;
freq[f] = 349.2;
freq[g] = 392.0;
freq[a] = 440.0;
freq[b] = 493.9;
purenote[note_Integer] := purenote[note] =
  Play[Sin[2 π freq[note] t], {t, 0, 1}]
```

1

We extend the function to get simultaneous notes from a List. We use Thread which takes f[{a,b,c}] to {f[a],f[b],f[c]}

```
notes[note_List] :=
 Sound[Thread[purenote[note]]]
```

2

Here are examples of their use.

```
cnote = purenote[c]
notes[{a, c, e}]
```

3

We can play with variable amplitudes for a fixed frequency, here we can hear the increased, but non-singular amplitude through zero.

```
Plot[Sin[540 x] / x, {x, -.1, .1},
 PlotRange → All, Filling → Axis]
Play[Sin[540 x] / x, {x, -1, 1}]
```

4

We can vary amplitudes and frequencies. Warning, playing with this function can become addictive...

```
Play[
 2 Sin[20 x Sin[x Exp[-x / .2]  + Sin[x] / x]] +
  Exp[(1 + Cos[x])] Sin[x Exp[x / 10]]
   Sin[1500 x], {x, 0.01, 20}]
```

5

1: The seven musical notes around middle C indexed here with integers and then their frequencies (in hertz) are defined with a `freq`. The function *Note* takes one of the seven indexed notes and creates a wave-form for that note. The function `Play` takes the waveform and produces audio output. We introduce a function, *purenote* , that takes an integer argument and plays the corresponding exact note.

2: To play a sequence of notes, a list of notes must be passed to `Sound`. We write a function, *notes* , that uses `Thread` to create a list of object created by applications of `purenote`. In other words, `Thread[function[{la,lb,lc}]]` returns `{function[la], function[lb], function[lc]}`.

3: This is an example of the use of *note* and *purenote* .

4: This is noise generated from a function; we can modulate the amplitude and frequency. Enjoy the fact that $\sin(x)/x$ is not singular at $x = 0$.

5: This is a function that I cooked up. Enjoy.

3.016 Home

Full Screen

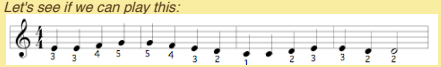
Close

Quit

Music and Instruments

Having no musical talent whatsoever, I try to write a program to make music.

Let's see if we can play this:



```
twoframes = {e, e, f, g, g, f, e, d, c, c, d, e};
```

1

We will play it, but it probably not what was intended...

```
notes[twoframes]
```

2

We create a rest of a fixed length, and then use Riffle to insert a rest after each note. We use a new function, called SoundNote, with None as the first argument, we get no sound.

```
purenote[rest] = SoundNote[None, .15];
```

3

```
notes[Riffle[twoframes, rest]]
```

4

SoundNote can take general strings for arguments as well, here we enter the musical notes from above (as characters), but one octave lower..

```
TwoFramesLower = {"E3", "E3", "F3", "G3", "G3",
   "F3", "E3", "D3", "C3", "C3", "D3", "E3"};
```

5

The default is to use a piano to make the sound; here we ask for a duration of 6/10 of a second.

```
piano[note_String] := SoundNote[note, .6]
```

6

```
Sound[Thread[piano[TwoFramesLower]]]
```

7

We can use other MIDI instruments as well; here is a bagpipe

```
bagpipe[note_String] :=
 SoundNote[note, .6, "Bagpipe"]
Sound[Thread[bagpipe[TwoFramesLower]]]
```

8

And, now for the birds.

```
avian[note_String] :=
 SoundNote[note, .2, "Bird"]
avian[rest] = SoundNote[None, .4];
Sound[
 Thread[avian[Riffle[TwoFramesLower, rest]]]]
```

9

1: Someone who knows how to read music told me what these notes were; so, I entered them into a list.

2: This is musical score with one-second duration notes played every 1 second. Oh, Joy.

3: This is probably not what Ludwig Van had in mind; so let's figure out how to insert a 'rest.' We use MATHEMATICA® 's built-in `SoundNote` for .15 seconds to define a *purenote* for rest `rest`.

4: `Riffle[{l1,l2,l3},x]` intersperses `x` into the list and returns `{l1,x,l2,x,l3,x}`. Calling *notes* on the resulting structure returns the pure notes with rests in between.

5: `SoundNote` will also play MIDI sounds and take string arguments for notes. We recreate a sting version of the musical score for notes one octave below middle-c.

6: By default, `SoundNote` returns a MIDI piano sound. We create a function, *piano* , to play a single note for a 0.6 second duration.

7: By using `Thread` again, we play the 12-note musical score on the piano.

8: There are other MIDI instruments; here we create the function *bagpipe* and play the score with a simulated bagpipe.

9: And finally, we introduce a 'cheep' little function, *avian* , to let the birds sing their own joy.

3.016 Home

Full Screen

Close

Quit

Random Notes and Instruments

Just because we can, let's see how sequences of random notes sound. We'll add random instruments and rests too.

Let's hear what random notes sound like: SoundNote[n] will play n semitones above middle C. Here we make a list of random notes and play them.

```
RandomNotes = Table[SoundNote[
    RandomInteger[{-15, 20}], .2], {36}];
Sound[RandomNotes, 10]
```

1

Here, we mix in some rests at random

```
RanRest[] := Module[{rdur = .2},
  If[RandomReal[] > 0.5, rdur = .4];
  SoundNote[None, rdur]]
RandomNotesandRests =
  Table[If[RandomReal[] ≥ .33,
    SoundNote[RandomInteger[{-15, 20}], .2],
    RanRest[]], {96}];
Sound[RandomNotesandRests, 20]
```

2

Now, we ask for random instruments as well, with "chords" of up to 5 instruments at each beat.

```
RandomInstruments =
  Table[If[RandomReal[] > 0.5, Table[
     SoundNote[
      Table[RandomInteger[{-15, 20}],
       {RandomInteger[{1, 5}]}],
      Round[RandomReal[{1, 2}], .2],
      RandomInteger[{1, 15}]],
     {RandomInteger[{1, 4}]}], RanRest[]],
   {48}];
Sound[RandomInstruments, 20]
```

3

Finally, some random percussion events.

```
percs = {SoundNote["Clap", 1],
   SoundNote["Sticks", 1], SoundNote["Shaker",
    1], SoundNote["LowWoodblock", 1],
   SoundNote["Castanets", 1],
   SoundNote[None, 1]};
perctable = Table[RandomChoice[percs], {50}];
Sound[perctable, 20]
```

4

1: *RandomNotes* creates a 36 member random set of single pitches from 15 semi-tones below to 20 semi-tones above middle-c. We play them for ten seconds.

2: To introduce some variety into the random melody, we write a program, *RanRest* , which will be used to introduce rests with lengths .2 and .4 with equal probability. A list, *RandomNotesandRests* , is created with random notes which call *RanRest* for approximately 1/3 of the members, and from the same random set as *RandomNotes* for the remainder.

3: Now, we introduce random MIDI instruments into our score: *RandomInstruments* is a `Table` of length 48, and each member is a list of a random number, between 1 and 5, of random instruments with random notes. These list-elements create a "chord." Rests are introduced randomly, at about 1/2 of the beats.

4: Here we play with random percussion MIDI instruments. Dancing to this is not necessarily advised.

A function that is periodic in a single variable can be expressed as:

$$\begin{aligned} f(x+\lambda) &= f(x) \\ f(t+\tau) &= f(t) \end{aligned} \tag{17-1}$$

The first form is a suggestion of a spatially periodic function with wavelength λ and the second form suggests a function that is periodic in time with period τ. Of course, both forms are identical and express that the function has the same value at an infinite number of points ($x = n\lambda$ in space or $t = n\tau$ in time where n is an integer.)

Specification of a periodic function, $f(x)$, within one period $x \in (x_o, x_o + \lambda)$ defines the function everywhere. The most familiar periodic functions are the trigonometric functions:

$$\sin(x) = \sin(x + 2\pi) \quad \text{and} \quad \cos(x) = \cos(x + 2\pi) \tag{17-2}$$

However, any function can be turned into a periodic function.

3.016 Home

Full Screen

Close

Quit

Using Mod to Create Periodic Functions

This section has been removed from the 2012 Notebook

Periodic functions are often associated with the "modulus" operation. $\texttt{Mod}[x, \lambda]$ is the remainder of the result of *recursively* dividing x by λ until the result lies in the domain $0 \leq \text{Mod}[x, \lambda] < \lambda$). Another way to think of modulus is to find the "point" where are periodic function should be evaluated if its primary domain is $x \in (0, \lambda)$.

Mod is a very useful function that can be used to force objects to be periodic. Mod[x,λ] return that part of x that lies within 0 and λ. Or, in other words if we map the real line x to a circle with circumference λ, then Mod[x,λ] returns were x is mapped onto the circle.

```
modmatdemo[n_Integer, λ_Integer] :=
  Table[{i, Mod[i, λ]}, {i, 1, n}] //
   MatrixForm;
modcircledemo[n_Integer, λ_Integer] :=
  Module[{xpos, angle, cpos},
   Graphics[
    Table[xpos = 3 Quotient[i - 1, λ];
     angle = 2 Pi Mod[i, λ] / λ;
     cpos = {Cos[angle], Sin[angle]};
     {Circle[{xpos, 0}],
      Text[i,
       Flatten[{{xpos, 0} + 1.2 * cpos}]],
      Text[Mod[i, λ], Flatten[{{xpos, 0} +
          0.8 * cpos}]]}, {i, 1, n}]]];
```

1

```
GraphicsColumn[
 {modmatdemo[13, 5], modcircledemo[26, 5]},
 ImageSize → Full]
```

2

Boomerang uses Mod to force a function, f, with a single argument, x, to be periodic with wavelength λ

```
Boomerang[f_, x_, λ_] := f[Mod[x, λ]]
```

3

```
AFunction[x_] := ((3 - x) ^ 3) / 27
```

4

*The following step uses **Boomerang** to produce a periodic repetition of **AFunction** over the range 0 < x < 6:*

```
Plot[Boomerang[AFunction, x, 6],
 {x, -12, 12}, PlotRange → All]
```

5

1: We create two visualization methods to show how Mod works: *modmatdemo* creates a matrix with two columns (i, Mod[i,λ]); *modcircledemo* wraps the the counting numbers and their moduli around a Graphics- Circle with a λ sectors, after each circle becomes filled a new circle is created for subsequent filling. *modcircledemo* should show how Mod is related to mapping to a periodic domain.

2: We show both visualization demonstrations in a GraphicsColumn.

3: *Boomerang* uses Mod on the argument of any function f of a single argument to map the argument into the domain $(0, \lambda)$. Therefore, calling *Boomerang* on any function will create a infinitely periodic repetition of the function in the domain $(0, \lambda)$.

4: *AFunction* is created as an example to pass to *Boomerang*

5: Plot called on the periodic extension of wavelength $\lambda = 6$ of *AFunction* . This illustrates how *Boomerang* uses Mod to create a periodic function with a specified period.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Odd and Even Functions

The trigonometric functions have the additional properties of being an *odd* function about the point $x = 0$: $f_{\text{odd}} : f_{\text{odd}}(x) = -f_{\text{odd}}(-x)$ in the case of the sine, and an *even* function in the case of the cosine: $f_{\text{even}} : f_{\text{even}}(x) = f_{\text{even}}(-x)$.

This can generalized to say that a function is even or odd about a point $\lambda/2$: $f_{\text{odd}\frac{\lambda}{2}} : f_{\text{odd}\frac{\lambda}{2}}(\lambda/2 + x) = -f_{\text{odd}\frac{\lambda}{2}}(\lambda/2 - x)$ and $f_{\text{even}\frac{\lambda}{2}} : f_{\text{even}\frac{\lambda}{2}}(\lambda/2 + x) = f_{\text{even}\frac{\lambda}{2}}(\lambda/2 - x)$.

Any function can be decomposed into an odd and even sum:

$$g(x) = g_{\text{even}} + g_{\text{odd}} \tag{17-3}$$

The sine and cosine functions can be considered the odd and even parts of the generalized trigonometric function:

$$e^{ix} = \cos(x) + \imath \sin(x) \tag{17-4}$$

with period 2π.

Representing a particular function with a sum of other functions

A Taylor expansion approximates the behavior of a suitably defined function, $f(x)$ in the neighborhood of a point, x_o, with a bunch of functions, $p_i(x)$, defined by the set of powers:

$$p_i \equiv \vec{p} = (x^0, x^1, \ldots, x^j, \ldots) \tag{17-5}$$

The polynomial that approximates the function is given by:

$$f(x) = \vec{A} \cdot \vec{p} \tag{17-6}$$

where the vector of coefficients is defined by:

$$A_i \equiv \vec{A} = (\frac{1}{0!} f(x_o), \frac{1}{1!} \left.\frac{df}{dx}\right|_{x_o}, \ldots, \frac{1}{j!} \left.\frac{d^j f}{dx^j}\right|_{x_o}, \ldots) \tag{17-7}$$

The idea of a vector of infinite length has not been formally introduced, but the idea that as the number of terms in the sum in Eq. 17-6 gets larger and larger, the approximation should converge to the function. In the limit of an infinite number of terms in the sum (or the vectors of infinite length) the series expansion will converge to $f(x)$ if it satisfies some technical continuity constraints.

However, for periodic functions, the domain over which the approximation is required is only one period of the periodic function—the rest of the function is taken care of by the definition of periodicity in the function.

Because the function is periodic, it makes sense to use functions that have the same period to approximate it. The simplest periodic functions are the trigonometric functions. If the period is λ, any other periodic function with periods $\lambda/2$, $\lambda/3$, λ/N, will also have period λ. Using these "sub-periodic" trigonometric functions is the idea behind Fourier Series.

Fourier Series

The functions $\cos(2\pi x/\lambda)$ and $\sin(2\pi x/\lambda)$ each have period λ. That is, they each take on the same value at x and $x+\lambda$.

There are an infinite number of other simple trigonometric functions that are periodic in λ; they are $\cos[2\pi x/(\lambda/2))]$ and $\sin[2\pi x/(\lambda/2))]$ and which cycle two times within each λ, $\cos[2\pi x/(\lambda/3))]$ and $\sin[2\pi x/(\lambda/3))]$ and which cycle three times within each λ, and, in general, $\cos[2\pi x/(\lambda/n))]$ and $\sin[2\pi x/(\lambda/n))]$ and which cycle n times within each λ.

The constant function, $a_0(x) = \text{const}$, also satisfies the periodicity requirement.

The superposition of multiples of any number of periodic function must also be a periodic function, therefore any function $f(x)$ that satisfies:

$$\begin{aligned} f(x) &= \mathcal{E}_0 + \sum_{n=1}^{\infty} \mathcal{E}_n \cos\left(\frac{2\pi n}{\lambda}x\right) + \sum_{n=1}^{\infty} \mathcal{O}_n \sin\left(\frac{2\pi n}{\lambda}x\right) \\ &= \mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \end{aligned} \tag{17-8}$$

where the k_i are the *wave-numbers* or *reciprocal wavelengths* defined by $k_j \equiv 2\pi j/\lambda$. The k's represent inverse wavelengths—

3.016 Home
Full Screen
Close
Quit

large values of k represent short-period or high-frequency terms.

If any periodic function $f(x)$ could be represented by the series in in Eq. 17-8 by a suitable choice of coefficients, then an alternative representation of the periodic function could be obtained in terms of the simple trigonometric functions and their amplitudes.

The "inverse question" remains: "How are the amplitudes $\mathcal{E}_{k_n}$ (the even trigonometric terms) and $\mathcal{O}_{k_n}$ (the odd trigonometric terms) determined for a given $f(x)$?"

The method follows from what appears to be a "trick." The following three integrals have simple forms for integers M and N:

$$\begin{aligned}
&\int_{x_0}^{x_0+\lambda} \sin\left(\frac{2\pi M}{\lambda}x\right)\sin\left(\frac{2\pi N}{\lambda}x\right)dx = \begin{cases} \frac{\lambda}{2} \text{ if } M = N \\ 0 \text{ if } M \neq N \end{cases} \\
&\int_{x_0}^{x_0+\lambda} \cos\left(\frac{2\pi M}{\lambda}x\right)\cos\left(\frac{2\pi N}{\lambda}x\right)dx = \begin{cases} \frac{\lambda}{2} \text{ if } M = N \\ 0 \text{ if } M \neq N \end{cases} \\
&\int_{x_0}^{x_0+\lambda} \cos\left(\frac{2\pi M}{\lambda}x\right)\sin\left(\frac{2\pi N}{\lambda}x\right)dx = 0 \text{ for any integers } M, N
\end{aligned} \tag{17-9}$$

The following shows a demonstration of this *orthogonality relation for the trigonometric functions.*

Orthogonality of Trigonometric Functions

This section has been removed from the 2012 Notebook

This is a Demonstration that the relations in Eq. 17-9 are true.

```
fassume = {Minteger ∈ Integers,
  Ninteger ∈ Integers, xo ∈ Reals, λ > 0}
coscos = Integrate[
  Cos[2 π Minteger x / λ] Cos[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
1

Demonstrating $\int_{xo}^{xo+\lambda} \cos(2\,m\pi x/\lambda)\cos(2\,n\pi x/\lambda)\,dx = 0$ *for* $m \neq n$

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[coscos /. {Minteger → mrand ,
     Ninteger → nrand}]], {20}]
```
2

when n=m give indeterminate values, for these we should use a Limit.

```
Limit[coscos, Minteger → Ninteger,
 Assumptions → fassume]
```
3

```
cossin = Integrate[
  Cos[2 π Minteger x / λ] Sin[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
4

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[cossin /. {Minteger → mrand ,
     Ninteger → nrand}]], {20}]
```
5

```
Limit[cossin, Minteger → Ninteger,
 Assumptions → fassume]
```
6

```
sinsin = Integrate[
  Sin[2 π Minteger x / λ] Sin[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
7

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[sinsin /. {Minteger → mrand ,
     Ninteger → nrand}]], {20}]
```
8

```
Limit[sinsin, Minteger → Ninteger,
 Assumptions → fassume]
```
9

1: Using `Integrate` for $\cos(2\pi Mx/\lambda)\cos(2\pi Nx/\lambda)$ over a definite interval of a single wavelength, does not produce a result that obviously vanishes for $M \neq N$.

2: However, random replacement of the symbolic integers with integers results in a zero. So, one the orthogonality relation is plausible.

3: Using `Assuming` and `Limit`, one can show that the relation ship vanishes for $N = M$. Although, it is a bit odd to be use continuous limits with integers.

4–6: This shows the same process for $\int \cos(2\pi Mx/\lambda)\sin(2\pi Nx/\lambda)dx$, which always returns zeroes.

7–9: And, $\int \sin(2\pi Mx/\lambda)\sin(2\pi Nx/\lambda)dx$ returns zeroes unless $M = N$.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Using this orthogonality trick, any amplitude can be determined by multiplying both sides of Eq. 17-8 by its conjugate trigonometric function and integrating over the domain. (Here we pick an arbitrary periodic domain $x \in (x_c + \lambda/2, x_c + \lambda/2)$, but any other starting point would work fine.)

$$\begin{aligned}
\cos(k_M x) f(x) &= \cos(k_M x) \left(\mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \right) \\
\int_{x_c-\lambda/2}^{x_c+\lambda/2} \cos(k_M x) f(x) dx &= \int_{x_c-\lambda/2}^{x_c+\lambda/2} \cos(k_M x) \left(\mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \right) dx \qquad (17\text{-}10) \\
\int_{x_c-\lambda/2}^{x_c+\lambda/2} \cos(k_M x) f(x) dx &= \frac{\lambda}{2} \mathcal{E}_{k_M}
\end{aligned}$$

This provides a formula to calculate the even coefficients (amplitudes) and multiplying by a sin function provides a way to calculate the odd coefficients (amplitudes) for $f(x)$ periodic in the fundamental domain $x \in (0, \lambda)$.

$$\begin{aligned}
\mathcal{E}_{k_0} &= \frac{1}{\lambda} \int_{x_c-\lambda/2}^{x_c+\lambda/2} f(x) dx \\
\mathcal{E}_{k_N} &= \frac{2}{\lambda} \int_{x_c-\lambda/2}^{x_c+\lambda/2} f(x) \cos(k_N x) dx \qquad k_N \equiv \frac{2\pi N}{\lambda} \qquad (17\text{-}11) \\
\mathcal{O}_{k_N} &= \frac{2}{\lambda} \int_{x_c-\lambda/2}^{x_c+\lambda/2} f(x) \sin(k_N x) dx \qquad k_N \equiv \frac{2\pi N}{\lambda}
\end{aligned}$$

The constant term has an extra factor of two because $\int_0^{\lambda} \mathcal{E}_{k_0} dx = \lambda \mathcal{E}_{k_0}$ instead of the $\lambda/2$ found in Eq. 17-9.

Other forms of the Fourier coefficients

Sometimes the primary domain is defined with a different starting point and different symbols, for instance Kreyszig uses a centered domain by using $-L$ as the starting point and $2L$ as the period, and in these cases the forms for the Fourier coefficients look a bit different. One needs to look at the domain in order to determine which form of the formulas to use.

Extra Information and Notes

Potentially interesting but currently unnecessary

The "trick" of multiplying both sides of Eq. 17-8 by a function and integrating comes from the fact that the trigonometric functions form an orthogonal basis *for functions with inner product defined by*

$$f(x) \cdot g(x) = \int_0^\lambda f(x)g(x)dx$$

Considering the trigonometric functions as components of a vector:

$$\begin{aligned}
\vec{e_0}(x) &= (1, 0, 0, \dots,) \\
\vec{e_1}(x) &= (0, \cos(k_1 x), 0, \dots,) \\
\vec{e_2}(x) &= (0, 0, \sin(k_1 x), \dots,) \\
\dots &= \quad \vdots \\
\vec{e_n}(x) &= (\dots\dots, \sin(k_n x), \dots,)
\end{aligned}$$

then these "basis vectors" satisfy $\vec{e_i} \cdot \vec{e_j} = (\lambda/2)\delta_{ij}$*, where* $\delta_{ij} = 0$ *unless* $i = j$*. The trick is just that, for an arbitrary function represented by the basis vectors,* $\vec{P}(x) \cdot \vec{e_j}(x) = (\lambda/2)P_j$*.*

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Calculating Fourier Series Amplitudes

Functions are developed which compute the even (cosine) amplitudes and odd (sine) amplitudes for an input function of one variable. These functions are extended to produce the first N terms of a Fourier series.

First we will "do it the hard way" and write short programs that evaluate Fourier coefficients; then we will demonstrate how to make use of built-in functions in Mathematica's ***FourierTransform*** *package...*

Define functions based on the formulas derived for the fourier amplitudes

The constant term:

```
EvenTerms[0, function_ , λ_] :=
 1/λ ∫_0^λ function[dum] ddum
```
1

A function that defines each even amplitude individually (this is not very efficient, it would be better to evaluate the integral once and use that result)

```
EvenTerms[SP_Integer, function_ , λ_] :=
 EvenTerms[SP, function , wavelength] =
  2/λ ∫_0^λ function[dum] Cos[2 SP π dum/λ] ddum
```
2

Define the zeroth odd term as zero for symmetry with the even terms:

```
OddTerms[0, function_ , λ_] := 0
```
3

```
OddTerms[SP_Integer, function_ , λ_] :=
 OddTerms[SP, function , λ] =
  2/λ ∫_0^λ function[dum] Sin[2 SP π dum/λ] ddum
```
4

A function to create a vector of amplitudes for the odd terms and one for the even terms

```
OddAmplitudeVector[
  NTerms_Integer, function_, λ_] :=
 Table[OddTerms[i, function, λ],
  {i, 0, NTerms}]
```
5

```
EvenAmplitudeVector[
  NTerms_Integer, function_, λ_] :=
 Table[EvenTerms[i, function, λ],
  {i, 0, NTerms}]
```
6

1–2: *EvenTerms* computes symbolic representations of the even (cosine) coefficients using the formulas in Eq. 17-11. The $N = 0$ term is computed with a supplemental definition because of its extra factor of 2. The domain is chosen so that it begins at $x = 0$ and ends at $x = \lambda$.

3–4: *OddTerms* performs a similar computation for the sine-coefficients; the $N = 0$ amplitude is set to zero explicitly. It will become convenient to include the zeroth-order coefficient for the odd (sine) series which vanishes by definition. *The functions work by doing an integral for each term—this is not very efficient. It would be more efficient to calculate the integral symbolically once and then evaluate it once for each term.*

5–6: *OddAmplitudeVector* and *EvenAmplitudeVectors* create *amplitude vectors* for the cosine and sine terms with specified lengths and domains.

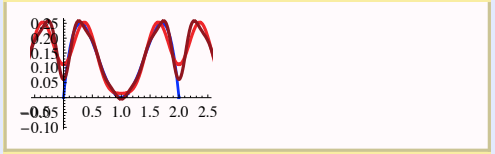
5: This function, $f(x) = x(1-x)^2(2-x)$, will be used for particular examples of Fourier series, note that it is an even function over $0 < x < 2$.

3.016 Home
Full Screen
Close
Quit

Approximations to Functions with Truncated Fourier Series

Example of using Eq. 17-11 to calculate a Fourier Series for a particular function.

```
myfunction[x_] := (x * (2 - x) * (1 - x)^2)                                    1
OriginalPlot = Plot[myfunction[x], {x, 0, 2},
  PlotStyle → {Hue[.66], Thickness[0.015]}]                                    2
EvenAmplitudeVector[6, myfunction, 2]                                          3
OddAmplitudeVector[6, myfunction, 2]                                           4
OddBasisVector[NTerms_Integer, var_, λ_] :=
 Table[Sin[2 π i var / λ], {i, 0, NTerms}]                                     5
OddBasisVector[6, x, 2]                                                        6
EvenBasisVector[NTerms_Integer, var_, λ_] :=
 Table[Cos[2 π i var / λ], {i, 0, NTerms}]                                     7
EvenBasisVector[6, x, 2]                                                       8
FourierTruncSeries[n_, function_, var_,
  λ_] := EvenAmplitudeVector[n, function, λ].
   EvenBasisVector[n, var, λ] +
  OddAmplitudeVector[n, function, λ].
   OddBasisVector[n, var, λ]                                                   9
FourierTruncSeries[6, myfunction, x, 2]                                        10
FPlot[n_Integer] := FPlot[n] = Plot[Evaluate[
    FourierTruncSeries[n, myfunction, x, 2]],
  {x, -2, 4}, PlotStyle →
   {Thick, ColorData[n, "ColorList"]}]                                         11
Show[OriginalPlot, FPlot[3], FPlot[6],
 PlotRange → {{-0.5, 2.5}, {-0.1, 0.26}}]                                      12
```



1: We introduce and example function $x(2-x)(1-x^2)$ that vanishes at $x = 0, 1, 2$ that will be used to produce a periodic function with $\lambda = 2$item We store the example function's graphical representation in `OriginalPlot`. Note that there will be a sharp discontinuity in the derivative at the edges of the periodic domain.

3–4: The Fourier coefficients, truncated at six terms, are computed with the functions that we defined above, *OddAmplitudeVector* and *EvenAmplitudeVector* . Note that because of the even symmetry of the function about the middle, all of the odd coefficients vanish.

5–8: *OddBasisVector* and *EvenBasisVector* , create vectors of *basis functions* of specified lengths and periodic domains.

9–10: The Fourier series up to a certain order can be defined as the sum of two inner (dot) products: the inner product of the odd coefficient vector and the sine basis vector, and the inner product of the even coefficient vector and the cosine basis vector.

11–12: This will illustrate the approximation for a truncated ($N = 6$) Fourier series. *FPlot* takes an integer truncation-argument and generates a plot for that truncation, but only for `myfunction`. It might be useful to extend this example so that it takes a function as an argument, but it makes more sense to leave this example and use MATHEMATICA® 's built-in Fourier series methods.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Demonstration of the use of the FourierSeries functions

This section has been modified in the 2012 Notebook

Fourier series expansions are a common and useful mathematical tool, and it is not surprising that MATHEMATICA® would have a package to do this and replace the inefficient functions defined in the previous example.

```
Needs["FourierSeries`"]                                                    1

AFunction[x_] := (x - 3)^3 / 27                                            2

Plot[AFunction[x], {x, 0, 6}]                                              3
```

Mathematica's Fourier Series functions are defined for function that are periodic in the domain x ∈ (-1/2,1/2). So we need to map the periodic functions to this domain

```
ReduceHalfHalf[f_, x_, λ_] :=
 f[(x + 1/2) * λ]                                                          4

ReducedFunction =
 ReduceHalfHalf[AFunction, x, 6] // Simplify                               5
```

$$8\,x^3$$

```
ExactPlot = Plot[ReducedFunction,
  {x, -1/2, 1/2}, PlotRange → All, PlotStyle →
   {Red, Opacity[0.5], Thickness[0.01]}]                                   6

FourierCosCoefficient[ReducedFunction, x, n]                               7

FourierSinCoefficient[ReducedFunction, x, n]                               8
```

$$\frac{2\,(-1)^n\,\left(6 - n^2\,\pi^2\right)}{n^3\,\pi^3}$$

```
FourierTrigSeries[ReducedFunction, x, 5]                                   9
```

$$\frac{2\,\left(-6 + \pi^2\right)\,\text{Sin}[2\,\pi\,x]}{\pi^3} + \frac{\left(3 - 2\,\pi^2\right)\,\text{Sin}[4\,\pi\,x]}{2\,\pi^3} + \frac{2\,\left(-2 + 3\,\pi^2\right)\,\text{Sin}[6\,\pi\,x]}{9\,\pi^3} + \frac{\left(3 - 8\,\pi^2\right)\,\text{Sin}[8\,\pi\,x]}{16\,\pi^3} + \frac{2\,\left(-6 + 25\,\pi^2\right)\,\text{Sin}[10\,\pi\,x]}{125\,\pi^3}$$

1: The functions in `FourierSeries` to operate on the unit period located at $x \in (-1/2, 1/2)$ by default. Therefore, the domains of functions of interest can be mapped onto this domain by a change of variables.

2–3: We introduce another function that will be approximated by a Fourier series. This function will be made periodic with $\lambda = 6$ in the untransformed variables.

4–6: *ReduceHalfHalf* is an example of a function design to do the required mapping. First the length of original domain is mapped to unity by dividing through by λ and then the origin is shifted by mapping the x (that the MATHEMATICA® functions will see) to $(-1/2, 1/2)$ with the transformation $x \to x + \frac{1}{2}$. *ReducedFunction* shows an example on the function defined above.

8–9: Particular amplitudes of the properly remapped function can be obtained with the functions `FourierCosCoefficient` and `FourierSinCoefficient`. In this example, a symbolic n is entered and a symbolic representation of the n^{th} amplitude is returned. Because the function is odd about the middle, all of the cosine-coefficients are zero.

9: A truncated Fourier series can be obtained symbolically to any order with `FourierTrigSeries`.

3.016 Home
Full Screen
Close
Quit

Recursive Calculation of a Truncated Fourier Series

This section has been removed from the 2012 Notebook

In this example, we build up a set of recursive function that will be utilized forefficient computation of a truncated Fourier series. These functions will be used in a subsequent visualization example.

```
ManipulateTruncatedFourierSeries[function_,
   {truncationstart_, truncationend_,
    truncjump_}] := Manipulate[Plot[Evaluate[
     FourierTrigSeries[function, x, itrunc]],
    {x, -1, 1}, PlotRange → {-2, 2}],
   {itrunc, {truncationstart,
     truncationend, truncjump}}];
```

1

The function above will work, but it is horribly inefficient! Because it asks FourierTrigSeries to calculate one more term each time, it is doing some redundant work. We can fix this up by having it calculate one new term and adding to the sum calculated previously. Here it is:

```
costerm[function_, x_, n_Integer] :=
 Simplify[FourierCosCoefficient[
    function, x, n]] Cos[2 π n x]
sinterm[function_, x_, n_Integer] :=
 Simplify[FourierSinCoefficient[
    function, x, n]] Sin[2 π n x]

TruncatedFourierSeries[function_, x_, 0] :=
 TruncatedFourierSeries[function, x, 0] =
  costerm[function, x, 0] +
   sinterm[function, x, 0]

TruncatedFourierSeries[
  function_, x_, n_Integer] :=
 TruncatedFourierSeries[function, x, n] =
  TruncatedFourierSeries[function, x, n - 1] +
   costerm[function, x, n] +
   sinterm[function, x, n]
```

2

1: *ManipulateTruncatedFourierSeries* is a simple example of visualization function for the truncated Fourier series. It uses the `Manipulate` function with three arguments in the iterator for the initial truncation `truncationstart`, final truncation, and the number to skip in between.

2: However, because the entire series is recomputed for each frame, the function above is not very efficient. In this second version, only two arguments are supplied to the iterator. At each function call, the two N^{th} Fourier terms are added to those computed in the $(N-1)^{th}$ and then stored in memory. The recursion stops at the defined $N = 0$ term.

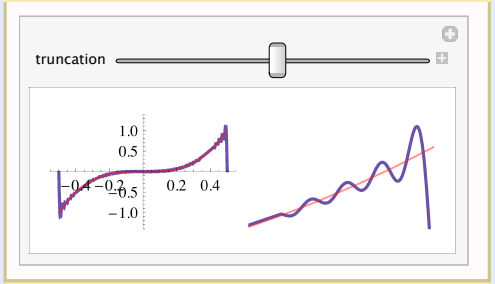
Visualizing Convergence of the Fourier Series: Gibbs Phenomenon

Functions that produce visualizations with `Manipulate` (each frame representing a different order of truncation of the Fourier series) are developed. This example illustrates *Gibbs phenomenon* where the approximating function oscillates wildly near discontinuities in the original function. In the `Manipulate` function, we use the option `Initialization` so that all evaluations during graphical output will be rapid.

The following will demonstrate how convergence is difficult where the function changes rapidly---this is known as Gibbs' Phenomenon

```
Manipulate[GraphicsRow[
  {plt = Show[Plot[theapprx[truncation],
      {x, -0.4999, 0.4999},
      PlotRange → {{-0.55, 0.55}, {-1.4, 1.4}},
      PlotStyle → {Thick, ColorData[1,
         truncation]}], ExactPlot], Show[plt,
    PlotRange → {{0.4, 0.5}, {0.5, 1.2}}]},
  ImageSize → Full],
 {{truncation, 100},
  1, 100, 1},
 Initialization → (Table[
     theapprx[i] = TruncatedFourierSeries[
       ReducedFunction, x, i], {i, 1, 100}];)]
```

1



1: Because *ReducedFunction* has a discontinuity (its end-value and its initial-value differ), this visualization will show Gibbs phenomena near the edges of the domain. The approximation is fine everywhere except in the neighborhood of the discontinuity. At the discontinuity, the oscillations about the exact value do not dampen out with increased truncation N, but the domain where the oscillations are ill-behaved shrinks with increased N.

Complex Form of the Fourier Series

The behavior of the Fourier coefficients for both the odd (sine) and for the even (cosine) terms was illustrated above. Functions that are even about the center of the fundamental domain (reflection symmetry) will have only even terms—all the sine terms will vanish. Functions that are odd about the center of the fundamental domain (reflections across the center of the domain and then across the x-axis.) will have only odd terms—all the cosine terms will vanish.

Functions with no odd or even symmetry will have both types of terms (odd and even) in its expansion. This is a restatement of the fact that any function can be decomposed into odd and even parts (see Eq. 17-3).

This suggests a short-hand in Eq. 17-4 can be used that combines both odd and even series into one single form. However, because the odd terms will all be multiplied by the imaginary number $\imath$, the coefficients will generally be complex. Also because $\cos(nx) = (\exp(inx) + \exp(-inx))/2$, writing the sum in terms of exponential functions only will require that the sum must be over both positive and negative integers.

For a periodic domain $x \in (-\lambda/2, \lambda/2)$, $f(x) = f(x+\lambda)$, the *complex form of the fourier series is given by:*

$$f(x) = \sum_{n=-\infty}^{\infty} \mathcal{C}_{k_n} e^{\imath k_n x} \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda}$$
$$\mathcal{C}_{k_n} = \frac{1}{\lambda} \int_{-\lambda/2}^{\lambda/2} f(x) e^{-\imath k_n x} dx \tag{17-12}$$

Because of the orthogonality of the basis functions $\exp(\imath k_n x)$, the domain can be moved to any wavelength, the following is also true: although the coefficients may have a different form.

Index