Supplementary material: FFTs With Perl
Chuck Connor January 24, 2014

Introduction

In practice, Fourier transforms are calculated using an algorithm called the fast Fourier transform (FFT), first developed in the 1960s by Cooley and Tukey. Most modern computer languages contain routines to calculate the FFT, and to perform filtering operations using transformed data sets and filters. The following examples show some FFT operations that are commonly used to process potential field data sets.

Perl Examples

The following examples calculate the FFT and perform filtering operations in PERL.

Example 1: The power spectrum

This first example plots the power spectrum of a function.

```
use Math::FFT;
  my $PI = 3.1415926539;
  my $N = 128;
  my ($series);
  for (my $k=0; $k<$N; $k++) {
      $series->[$k] = sin(4*$k*$PI/$N) + cos(6*$k*$PI/$N);
  }
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();
  my $spectrum = $fft->spctrm;

  for (my $k=0; $k<$N/2; $k++) {

     print "$k $series->[$k] $spectrum->[$k]\n";
  }
```

The first thing to notice is that the code uses a PERL module:

`use Math::FFT;`

This module might not be installed on your computer. If not, download and install the code. It is available from cpan, where you will also find some expla-

nation of its use:
http://search.cpan.org/ rkobes/Math-FFT-1.28/FFT.pm

The FFT module in PERL, like FFT code implemented in in other languages like Matlab and Python, contains a series of methods to ease mathematical operations involving Fourier transforms. In this code a series of data is created.

```
my $N = 128;
  my ($series);
  for (my $k=0; $k<$N; $k++) {
      $series->[$k] = sin(4*$k*$PI/$N) + cos(6*$k*$PI/$N);
  }
```

The number of points in the series is $\$N = 128$. This number must be 2^n, where n is an integer value. Data are then placed in a data structure called '$series'. Note that individual data within the structure are referenced using a pointer, $->$. This structure is important. Because this format is used, the FFT is actually just a few lines of code:

```
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();
  my $spectrum = $fft->spctrm;
```

These lines of code can be summarized as:

- define $fft as the Fourier transform of $series
- calculate the real discrete Fourier transform of series. Store the real and imaginary coefficients of the Fourier transform of $series in a new structure, referenced as $coeff.
- calculate the power spectrum of $series, using the Fourier transform, $fft.

After the spectrum is estimated, it is just a matter of printing out the spectrum in a format that can be plotted, say using gnuplot.

```
  for (my $k=0; $k<$N/2; $k++) {
     print "$k $series->[$k] $spectrum->[$k]\n";
  }
```

Note that only $N/2 lines are printed. The Fourier transform consists of $N/2 real (cosine) coefficients and $N/2 imaginary (sine) coefficients. The power spectrum, therefore, consists of $N/2 values. The index, $k, corresponds to the wavenumber. Exactly, the wavenumber = $\$k/L$, where L is the length of the series, N, times the sample spacing. The wavelength corresponding to, $k, is $= L/\$k$. $\$k = 0$ coincides with the mean value of the series. In taking the FFT of the series, this mean value is removed, so the value of the zero wavenumber is zero.

Although the command:

```
my $fft = new Math::FFT($series);
```

will remove the mean value of the series, it will not remove a trend (say a linear increase in the value of the series). So, it is usually necessary to pre-process the series to remove such trends.

Example 2: Box filter

Now let's expand on this theme and filter the series using a box filter. A box filter is a convolution that is multiplied with the Fourier transform of the original series, in which some coefficients are set to unity, and others are set to zero. Thus a box filter passes through some sine and cosine coefficients unchanged, and completely attenuates others.

```
use Math::FFT;
  my $PI = 3.1415926539;
  my $N = 128;
  my ($series);
  for (my $k=0; $k<$N; $k++) {
      $series->[$k] = sin(4*$k*$PI/$N) + cos(6*$k*$PI/$N);
  }
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();
  my $spectrum = $fft->spctrm;

  for (my $k=0; $k<7; $k++) {
      $$coeff[$k] = 0;
  }

  my $filtered_data = $fft->invrdft($coeff);

  for (my $k=0; $k<$N; $k++) {
    print "$k $series->[$k] $filtered_data->[$k]\n";
  }
```

This code is largely the same as Example 1, but some lines have been added to perform the filtering. Specifically,

```
for (my $k=0; $k<7; $k++) {
      $$coeff[$k] = 0;
  }
```

sets long wavelength, low wavenumber coefficients to zero. Since the longest wavelengths are cut, this is a high-pass filter. Exactly which wavelengths? The 'coeff' structure contains the list of real and imaginary coefficients. $$coeff[1] contains the first real coefficient – corresponding to the fundamental (longest) wavelength, $$coeff[2] contains the first imaginary coefficient – corresponding to the fundamental (longest) wavelength, $$coeff[3] contains the second real coefficient, etc. After the coefficients are changed, the inverse FFT creates the filtered series:

```
  my $filtered_data = $fft->invrdft($coeff);
```

and the results are printed so the filtered series can be plotted with gnuplot or a similar program.

Example 3: Ramped low-pass filter

Often a linearly ramped filter is used for low-pass, band-pass, or high-pass filters. Ramped filters set a threshold above or below which no filtering is done, an interval where there is a gradual filter, and a threshold above or below which coefficients are reduced to zero. An example PERL code for a low-pass ramped filter is:

```
use POSIX;
use Math::FFT;
  my $PI = 3.1415926539;
  my $sample_spacing = 1; #map distance between samples
  my $long_wave_cut=300;  #long wavelength ramp filter bound
  my $short_wave_cut=100;  #short wavelength ramp filter bound
  my ($series);


  open (IN, "<$ARGV[0]") || die ("Cannot open $ARGV[0]: $!");
  @MyMagneticDATA = <IN>;

  foreach $line (@MyMagneticDATA) {
   $N++;
   ($x, $series->[$N]) = split " ", $line;
  }

  @out_data=low_pass_filter ($z, $N, $sample_spacing,
        $long_wave_cut, $short_wave_cut);
  for ($i=0; $i<$N; $i++)
  {
     $profile_distance = $i*$sample_spacing;
     print "$profile_distance $series->[$i] $out_data[$i]\n ";
  }


sub low_pass_filter ($$$$$)
{
  my( $z, $N, $sample_spacing, $long_wave, $short_wave) = @_;
  my $i, $k, $k1, $k2, $filter;
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();

$k2 = floor(($N*$sample_spacing)/$long_wave);
$k1 = floor(($N*$sample_spacing)/$short_wave);


for ($k=1; $k<= $N/2; $k++)
  {
     if ($k<$k2) {$filter = 1;}
     elsif ($k>=$k1){$filter = 0;}
     else {$filter = 1.0 - (($k)-($k2))/(($k1) - ($k2));}

     $i++;
     $C_[$i] = $filter;
     $i++;
     $C_[$i] = $filter;
  }

  for ($i=0; $i<$N; $i++)
  {
       $$coeff[$i]=$$coeff[$i]*$C_[$i];
  }
  my $filtered= $fft->invrdft($coeff);

  for ($i=0; $i<$N; $i++)
  {
       $out[$i]=$$filtered[$i];
  }
  return @out;
}
```

This code reads a data file containing $\$N = 1024$ data values (distance along the profile and magnetic reading). Remember, there must be 2^n readings. The lines:

```
my $sample_spacing = 1; #map distance between samples
my $long_wave_cut=300;  #long wavelength ramp filter bound
my $short_wave_cut=100;  #short wavelength ramp filter bound
```

establish the sample spacing (actually redundant with the information in the data file), and the thresholds for the ramped filter, expressed as wavelength. So, if the 1024 measurements are spaced one meter apart, the units of $long_wave_cut and $short_wave_cut are in meters.

The line:

```
@out_data=low_pass_filter ($z, $N, $sample_spacing,
   $long_wave_cut, $short_wave_cut);
```

calls a subroutine called 'low_pass_filter' to actually do the filtering. Once this filtering is done, the results are printed to standard output, or redirected to a file. All of the action takes place in the subroutine. Most of the subroutine has the same form as given in previous examples. However, carefully note:

```
$k2 = floor(($N*$sample_spacing)/$long_wave);
$k1 = floor(($N*$sample_spacing)/$short_wave);
```

These lines translate the long- and short wavelength thresholds into integer values of the index, \$k. The values \$k1 and \$k2 are actually used as the thresholds. The actual ramped low-pass filter is defined by these lines of code:

```
    if ($k<$k2) {$filter = 1;}
    elsif ($k>=$k1){$filter = 0;}
    else {$filter = 1.0 - (($k)-($k2))/(($k1) - ($k2));}
```

which for each iteration of \$k, finds the value of the filter. Note the conditional structure (if...else) used and the mathematical formula for the linear ramp. The lines:

```
     $i++;
     $C_[$i] = $filter;
     $i++;
     $C_[$i] = $filter;
```

are used to make sure that both real and imaginary coefficient values are changed based on the filter. The Fourier transform of the original series is convolved with the filter:

```
 $$coeff[$i]=$$coeff[$i]*$C_[$i];
```

and, as before, the inverse FFT is calculated to obtain the filtered series of magnetic data.

Example 4: Upward continuation filter

Upward continuation attenuates the signal as a function of wavelength. An example of an upward continuation code is:

```
use Math::FFT;
  my $PI = 3.1415926539;
  my $N = 1024;  #number of samples
  my $sample_spacing = 1; #map distance between samples
  my $z = 100.0;  #map distance of upward continuation
  my ($series);

  for (my $k=0; $k<$N; $k++) {
   $series->[$k] = 4*cos(2*$k*$PI/$N) + sin(4*$k*$PI/$N)
              + cos(6*$k*$PI/$N) + 0.3*sin(5*$k*$PI/$N)
              + 1.1* cos(7*$k*$PI/$N)+0.25*sin(10*$k*$PI/$N)
              + 0.3* cos(9*$k*$PI/$N) + 0.1*sin(12*$k*$PI/$N)
              + 0.2* cos(15*$k*$PI/$N) + 0.4*sin(32*$k*$PI/$N)
              + 0.1* cos(17*$k*$PI/$N)  ;
  }

@out_data=upward_continuation_filter ($z, $N, $sample_spacing);
for ($i=0; $i<$N; $i++)
  {
     $profile_distance = $i*$sample_spacing;
     print "$profile_distance $series->[$i] $out_data[$i] ";
     print "\n";
  }


sub upward_continuation_filter ($$$)
{

  my( $z, $N, $sample_spacing) = @_;
  my $i, $k;
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();

for ($k=1; $k<= $N/2; $k++)
  {
    $i++;
    $C_[$i] = exp(-$z/($sample_spacing*$N)*$k);
    $i++;
    $C_[$i] = exp(-$z/($sample_spacing*$N)*$k);
  }

  for ($i=0; $i<$N; $i++)
  {
      $$coeff[$i]=$$coeff[$i]*$C_[$i];
  }


  my $filtered= $fft->invrdft($coeff);
 for ($i=0; $i<$N; $i++)
  {
   $out[$i]=$$filtered[$i];
  }
  return @out;
}
```

The structure of this code is similar to previous examples. The main difference lies in the filter:

```
$i++;
$C_[$i] = exp(-$z/($sample_spacing*$N)*$k);
$i++;
$C_[$i] = exp(-$z/($sample_spacing*$N)*$k);
```

which is the filter to perform the upward continuation as a function of height, $z, and wavenumber.

Example 5: Derivative filter

The Fourier transform is used to estimate the derivative of a function or data series using the same sort of filtering operation:

```
use Math::FFT;
  my $PI = 3.1415926539;
  my $N = 1024;  #number of samples (2^n)
  my $sample_spacing = 1; #map distance between samples

  my ($series);

  for (my $k=0; $k<$N; $k++) {
   $series->[$k] = 4*cos(4*$k*$PI/$N) + sin(4*$k*$PI/$N)
       + cos(6*$k*$PI/$N) + 0.3*sin(5*$k*$PI/$N)
       + 1.1* cos(7*$k*$PI/$N)+0.25*sin(10*$k*$PI/$N)
       + 0.3* cos(9*$k*$PI/$N) + 0.1* sin(12*$k*$PI/$N)
       + 0.2* cos(15*$k*$PI/$N) + 0.4 * sin(32*$k*$PI/$N)
       + 0.1* cos(17*$k*$PI/$N);
  }



@out_data= derivative_filter($N);
for ($i=0; $i<$N; $i++)
  {
     $profile_distance = $i*$sample_spacing;
    print "$profile_distance $series->[$i] $out_data[$i]\n";
  }
```

```
sub derivative_filter ($)
{

  my($N) = @_;
  my $i, $k;
  my $fft = new Math::FFT($series);
  my $coeff = $fft->rdft();
$i=1;
for ($k=1; $k< $N/2; $k++)
  {

    $i++;
    $tmp =  $$coeff[$i];
    $$coeff[$i]=$$coeff[$i+1];
    $$coeff[$i+1]=-$tmp;
    $i++;
  }

 my $filtered= $fft->invrdft($coeff);
 for ($i=0; $i<$N; $i++)
  {
   $out[$i]=$$filtered[$i];
  }
  return @out;
}
```

Again, the main difference with previous examples is in the filter design. Note how the real and imaginary coefficient are switched in order to calculate the derivative of the original function:

```
$i++;
$tmp =  $$coeff[$i];
$$coeff[$i]=$$coeff[$i+1];
$$coeff[$i+1]=-$tmp;
$i++;
```