Signal and Image Processing

The SignalProcessing and ImageTools packages have been expanded with new and updated commands along with enhanced tools in the Context Panel.

```
> with( ImageTools ):
```

```
> with( SignalProcessing ):
```

SignalProcessing

BandPower, MeanFrequency, and SpectralEntropy

The new BandPower, MeanFrequency, and SpectralEntropy commands are used to find the power spectral density of a signal, and compute, respectively, the band power, mean frequency, and spectral entropy, either for the entire signal, or a specific frequency band. For example:

```
> sample_rate := 5000;
```

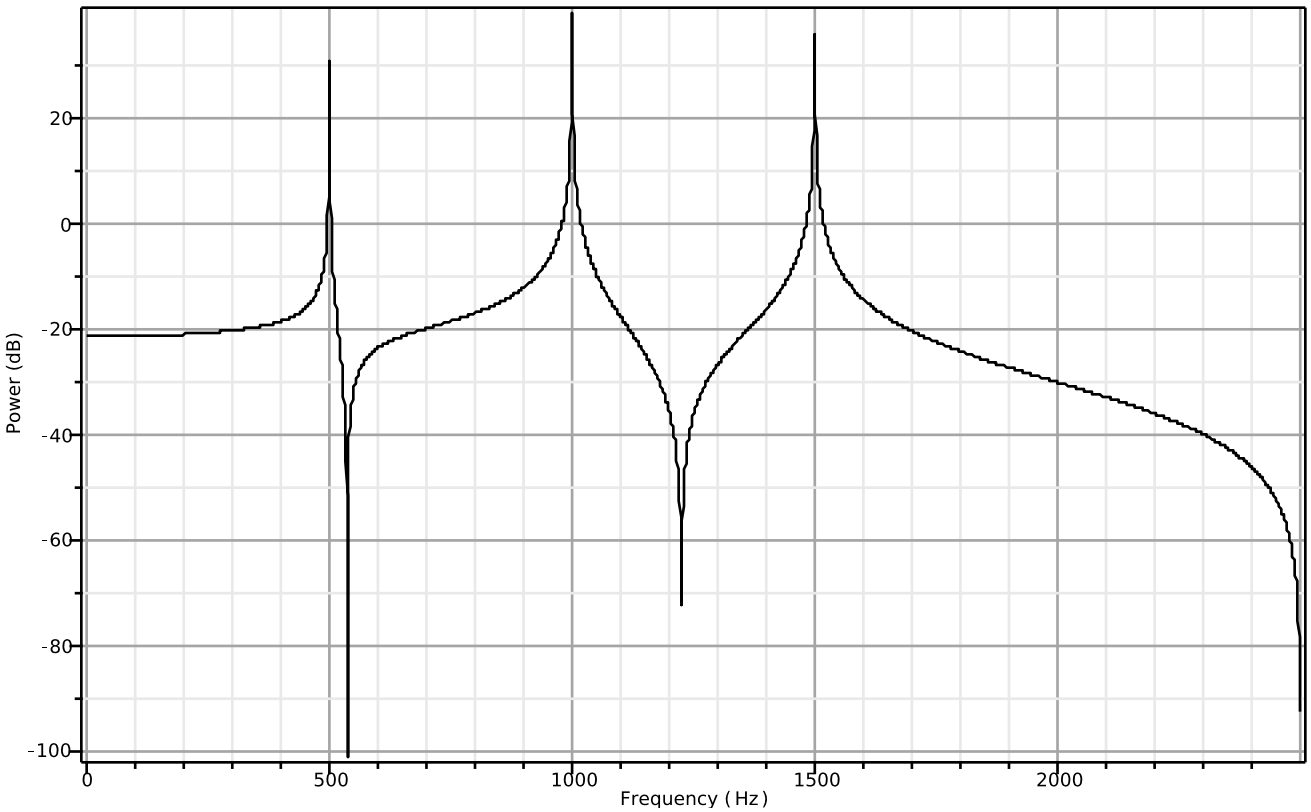
$$sample\_rate := 5000$$

```
> Times := Vector( [ seq ]( 0 .. 1, 1.0 / sample_rate ), datatype
  = float[8] ):
```

```
> g := t -> cos( 1000 * Pi * t ) + 3 * cos( 2000 * Pi * t ) + 2 *
  cos( 3000 * Pi * t );
  Signal := Vector( g~(Times), datatype = float[8] ):
```

$$g := t \mapsto \cos(1000 \cdot \pi \cdot t) + 3 \cdot \cos(2000 \cdot \pi \cdot t) + 2 \cdot \cos(3000 \cdot \pi \cdot t)$$

```
> Periodogram( Signal, samplerate = sample_rate, size = [800,400]
  );
```




```
> frequency_range := 250 * Unit( Hz ) .. 1.25 * Unit( kHz );
```

$$frequency\_range := 250\ \text{Hz}..1.25\ \text{kHz}$$

```
> BandPower( Signal, sample_rate, frequency_range );
  MeanFrequency( Signal, sample_rate, frequency_range );
  SpectralEntropy( Signal, sample_rate, frequency_range );
```

5.00688167653967309

949.412589381238149

1.34562172301835759

The commands can also return the power spectral density and frequencies:

```
> Power_Spectral_Density, Frequencies := BandPower( Signal,
  sample_rate, output = [ psd, frequencies ] );
```

$$Power\_Spectral\_Density, Frequencies := \begin{bmatrix} 2.87942720498115 \times 10^{-6} \\ 2.87943448146849 \times 10^{-6} \\ 2.87945631139094 \times 10^{-6} \\ 2.87949269545107 \times 10^{-6} \\ 2.87954363522815 \times 10^{-6} \\ 2.87960913240025 \times 10^{-6} \\ 2.87968918936491 \times 10^{-6} \\ 2.87978380934996 \times 10^{-6} \\ 2.87989299596818 \times 10^{-6} \\ 2.88001675264265 \times 10^{-6} \\ 2.88015508507797 \times 10^{-6} \\ 2.88030799748615 \times 10^{-6} \\ 2.88047549605390 \times 10^{-6} \\ 2.88065758716316 \times 10^{-6} \\ 2.88085427745574 \times 10^{-6} \\ \vdots \end{bmatrix}_{\text{2501 element Vector[column]}}, \begin{bmatrix} 0. \\ 0.999800039992002 \\ 1.99960007998400 \\ 2.99940011997600 \\ 3.99920015996801 \\ 4.99900019996001 \\ 5.99880023995201 \\ 6.99860027994401 \\ 7.99840031993601 \\ 8.99820035992802 \\ 9.99800039992002 \\ 10.9978004399120 \\ 11.9976004799040 \\ 12.9974005198960 \\ 13.9972005598880 \\ \vdots \end{bmatrix}_{\text{2501 element Vector[column]}}$$

PowerSpectrum

The PowerSpectrum command now accepts signals (in addition to FFTs) and other new options. For instance:

```
> n := 8192;
```

$$n := 8192$$

```
> fs := ( n - 1 ) / 2.0 / Pi;
```

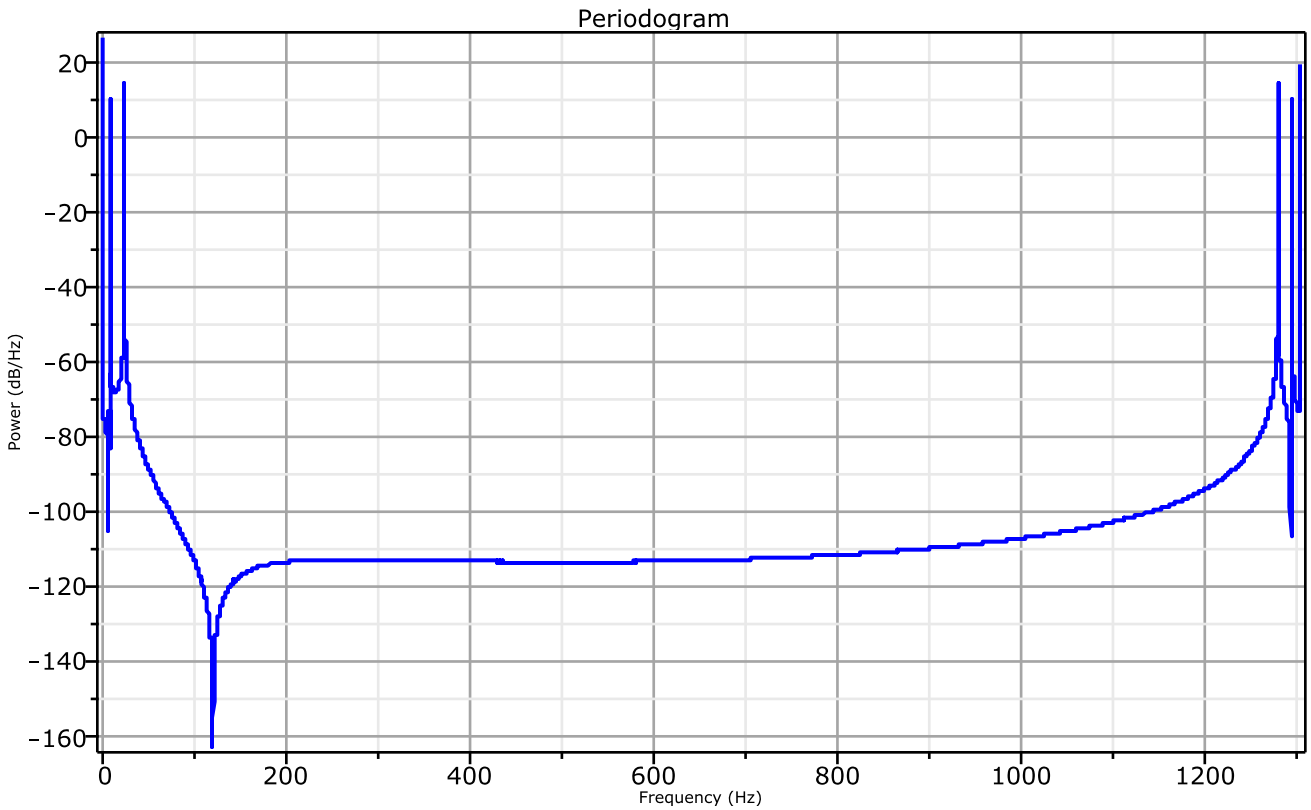
$$fs := 1303.638139$$

```
> T := Vector( n, k -> (k-1) / fs, datatype = float[8] ):

> X := Vector( n, k -> 3 * sin( 50 * T[k] ) + 5 * I * cos( 150 * T
  [k] ) + 10, datatype = complex[8] ):

> R := PowerSpectrum( X, samplerate = fs, variety = signal, window
  = Hamming, powerscale = dB/Hz, output = record ):

> 'power' = R[ power ];
```

$$power = \begin{bmatrix} 26.6375438109732 \\ 19.2256585408436 \\ -62.2889408858280 \\ -69.4683960561654 \\ -72.4884755068700 \\ -73.7649615862321 \\ -74.3657048170721 \\ -74.7040672866454 \\ -74.9334054839930 \\ -75.1143006501009 \\ -75.2728945621864 \\ -75.4216679891008 \\ -75.5671753931988 \\ -75.7132512080228 \\ -75.8624219197153 \\ \vdots \end{bmatrix}$$

8192 element Vector[column]

```
> R[ periodogram ];
```



Welch

The Welch command estimates the power spectrum of a signal, while attenuating the effect of noise at the expense of frequency resolution. To this end, the signal is divided into overlapping segments of equal (or nearly equal) length, and for each segment, a window is applied and the power spectrum found. The overall power spectrum is determined by averaging all the segment power spectra.

For an example, consider the following signal:

```
> num_points := 8192;
```

$$num\_points := 8192$$

```
> sample_rate := ( n - 1 ) / 2.0 / Pi;
```

$$sample\_rate := 1303.638139$$

```
> Times := Vector( num_points, k -> (k-1) / sample_rate, datatype
  = float[8] ):
```

```
> g := t -> cos( 10 * Pi * t ) + 3 * cos( 20 * Pi * t ) + 2 * cos(
  60 * Pi * t );
  Original_Signal := Vector[column]( g~( Times ), datatype = float
  [8] ):
```

$$g := t \mapsto \cos(10 \cdot \pi \cdot t) + 3 \cdot \cos(20 \cdot \pi \cdot t) + 2 \cdot \cos(60 \cdot \pi \cdot t)$$

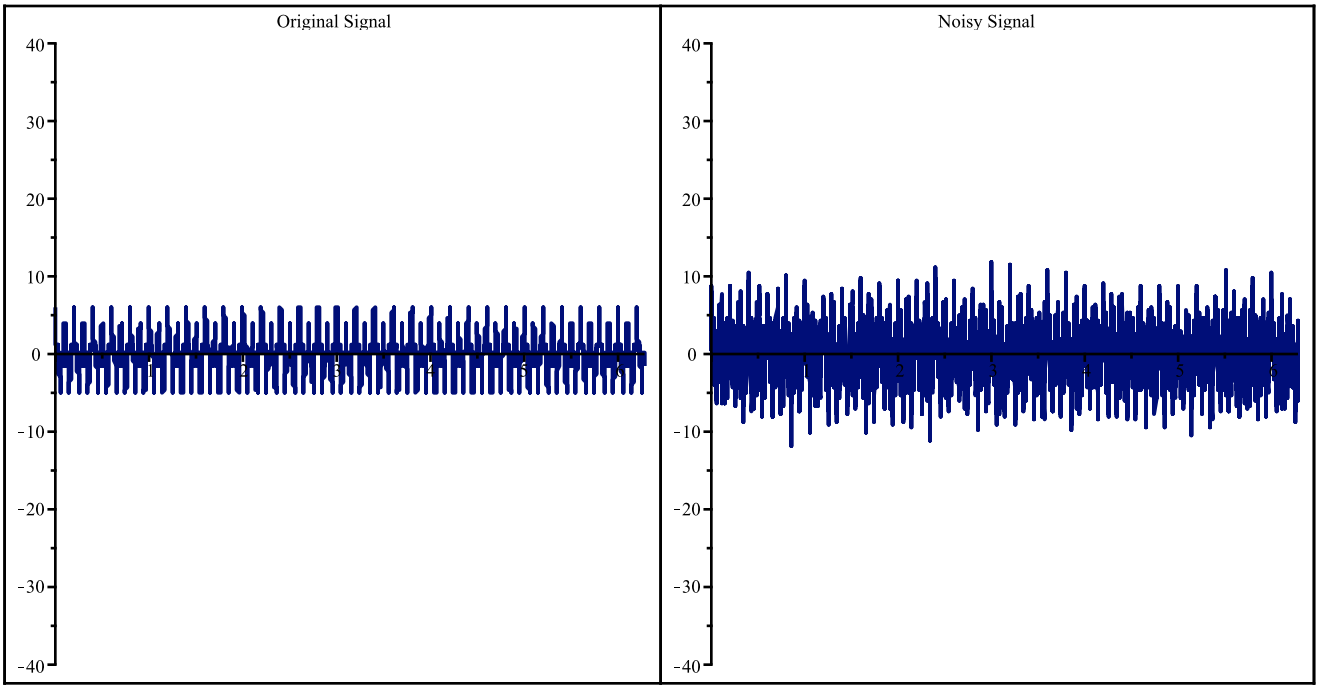
Now, add noise:

```
> use Statistics in
            Noise := Vector[column]( Sample( RandomVariable(
  Normal( 0, 2 ) ), num_points ) );
  end use:
```

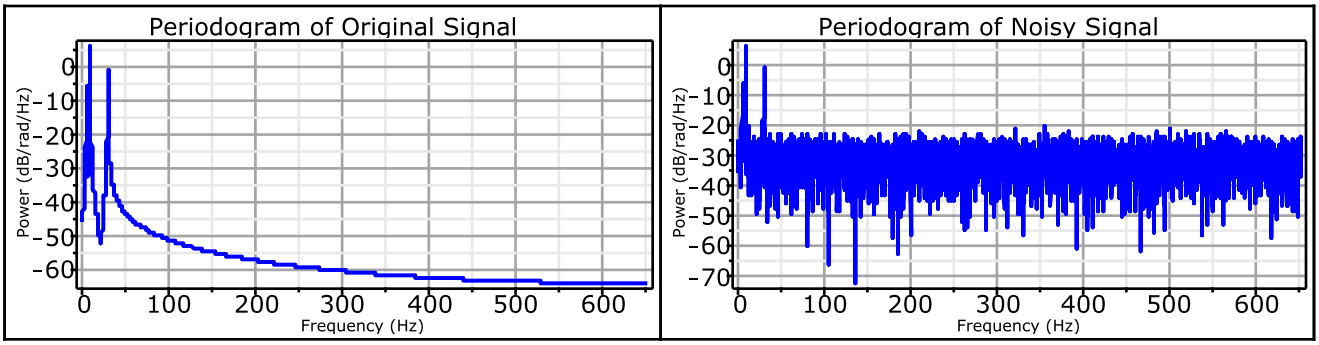
```
> Noisy_Signal := Original_Signal + Noise:
```

Now, let's compare plots of the signals and power spectra:

```
> plots:-display( Array( [
      dataplot( Times, Original_Signal, style = line, view = [0.
  .2*Pi,-40..40], title = "Original Signal" ),
      dataplot( Times, Noisy_Signal, style = line, view = [0..2*
  Pi,-40..40], title = "Noisy Signal" )
  ] ) );
```




```
> plots:-display( Array( [
      Welch( Original_Signal, samplerate = sample_rate,
  segmentsize = num_points, output = periodogram,
  periodogramoptions = [ title = "Periodogram of Original Signal"
  ] ),
      Welch( Noisy_Signal, samplerate = sample_rate, segmentsize =
  num_points, output = periodogram, periodogramoptions = [ title =
  "Periodogram of Noisy Signal" ] )
  ] ) );
```



Now, apply Welch's method with an appropriate choice of parameter values to return a record with the Vectors for the power spectrum and frequencies, along with the periodogram:

```
> R := Welch(
```

```
      Noisy_Signal,
      overlapsize = 512,
      segmentsize = 1024,
      window  = "Hamming",
      samplerate = sample_rate,
      temperendpoints = true,
      datapowerscale = 1/Hz,
      plotpowerscale = dB/rad/Hz,
      output = record
  ):
> 'power' = R[power];
```

$$power = \begin{bmatrix} 0.00346665141949456 \\ 0.00538579785665287 \\ 0.00586950629385681 \\ 0.0807403618411378 \\ 0.302631983733609 \\ 0.0527610429701862 \\ 0.00525081213890429 \\ 0.790269791677775 \\ 2.59105503877552 \\ 0.280851459621314 \\ 0.00688292331389145 \\ 0.00557150605725930 \\ 0.00556297954981199 \\ 0.00485358568424033 \\ 0.00544170357567522 \\ \vdots \end{bmatrix}$$

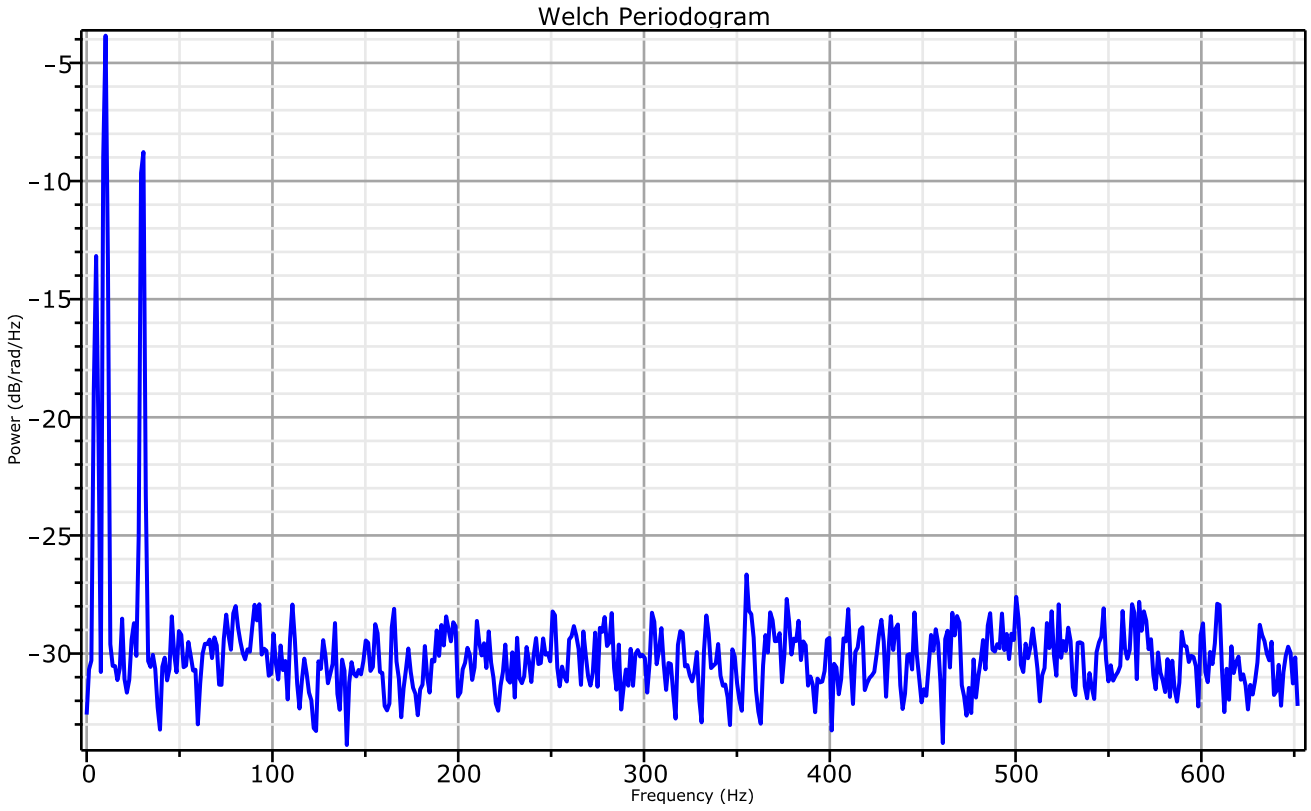
513 element Vector[column]

```
> 'frequencies' = R[frequencies];
```

$$frequencies = \begin{bmatrix} 0. \\ 1.27308412011719 \\ 2.54616824023437 \\ 3.81925236035156 \\ 5.09233648046875 \\ 6.36542060058594 \\ 7.63850472070312 \\ 8.91158884082031 \\ 10.1846729609375 \\ 11.4577570810547 \\ 12.7308412011719 \\ 14.0039253212891 \\ 15.2770094414062 \\ 16.5500935615234 \\ 17.8231776816406 \\ \vdots \end{bmatrix}$$

513 element Vector[column]

```
> R[periodogram];
```



MUSIC

The MUSIC command performs the Multiple Signal Classifier (MUSIC) Method on a signal, which estimates frequencies present in a noisy signal by computing the eigenvalues of the autocorrelation matrix and separating them into signal-subspace eigenvalues and noise-subspace eigenvalues. Consider, for an example, the following real-valued signal:

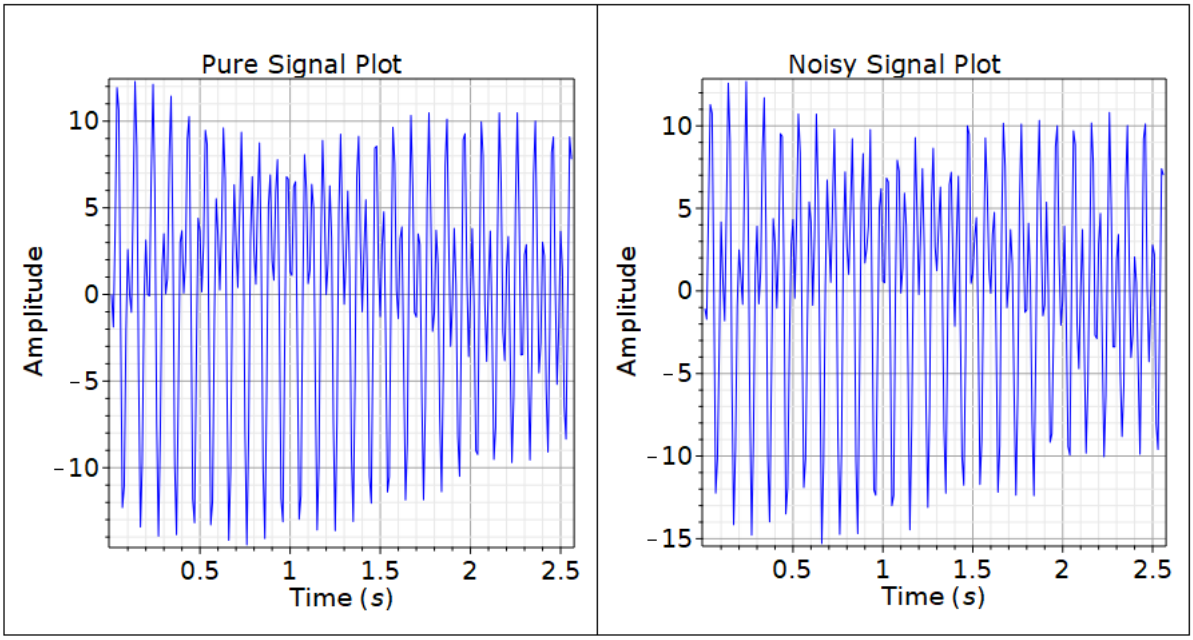
```
> # number of points in the signal
  m := 2^8:

> # sample rate
  fs := 100.0:

> # times
  T := Vector( m, k -> 2 * Pi * (k-1) / fs, 'datatype' = 'float[8]
  ' ):

> # pure signal
  X := Vector( m, k -> 5.00 * sin( 10.25 * T[k] ) + 3.00 * sin(
  10.40 * T[k] ) - 7.00 * sin( 20.35 * T[k] ), 'datatype' = 'float
  [8]' ):
```

To this, we add some noise:

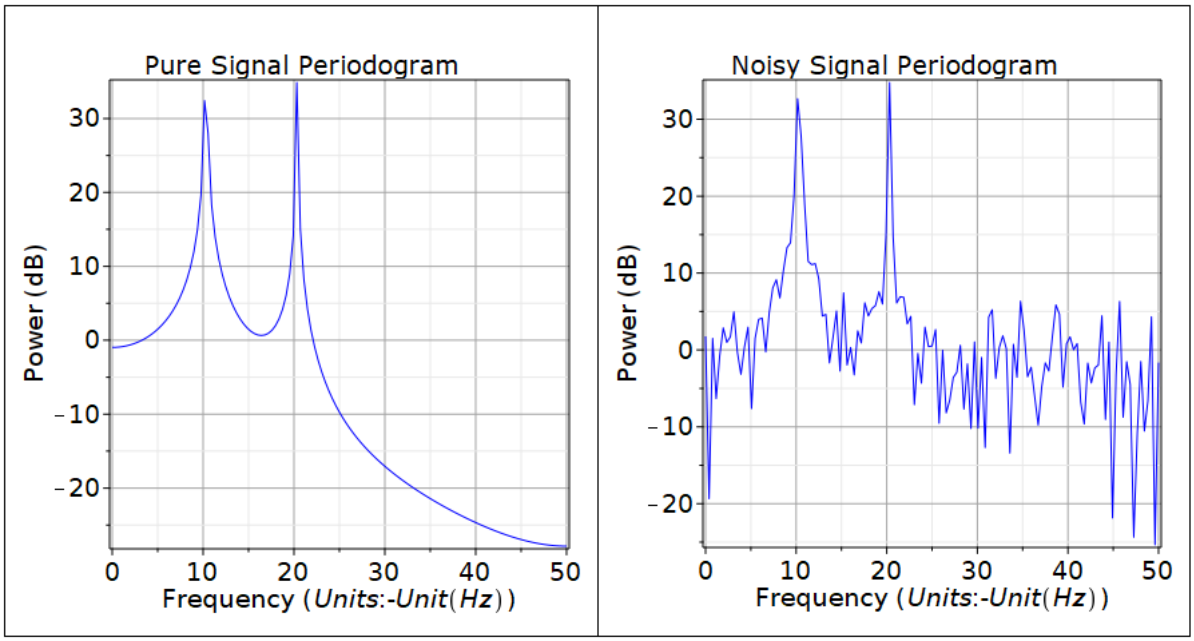
```
> # noise
  use Statistics in
            N := Vector[column]( Sample( RandomVariable( Normal(
  0, 1 ) ), m ) ):
  end use:

> # noisy signal
  Y := X + N:
```

Let's compare plots of the signals and power spectra for both the original and noisy versions:

```
> DocumentTools:-Tabulate( Array( [
            SignalPlot( X, 'samplerate' = fs, 'color' = 'blue',
  'title' = "Pure Signal Plot", 'font' = [ 'Verdana', 15 ] ),
            SignalPlot( Y, 'samplerate' = fs, 'color' = 'blue',
  'title' = "Noisy Signal Plot", 'font' = [ 'Verdana', 15 ] )
  ] ) ):
```

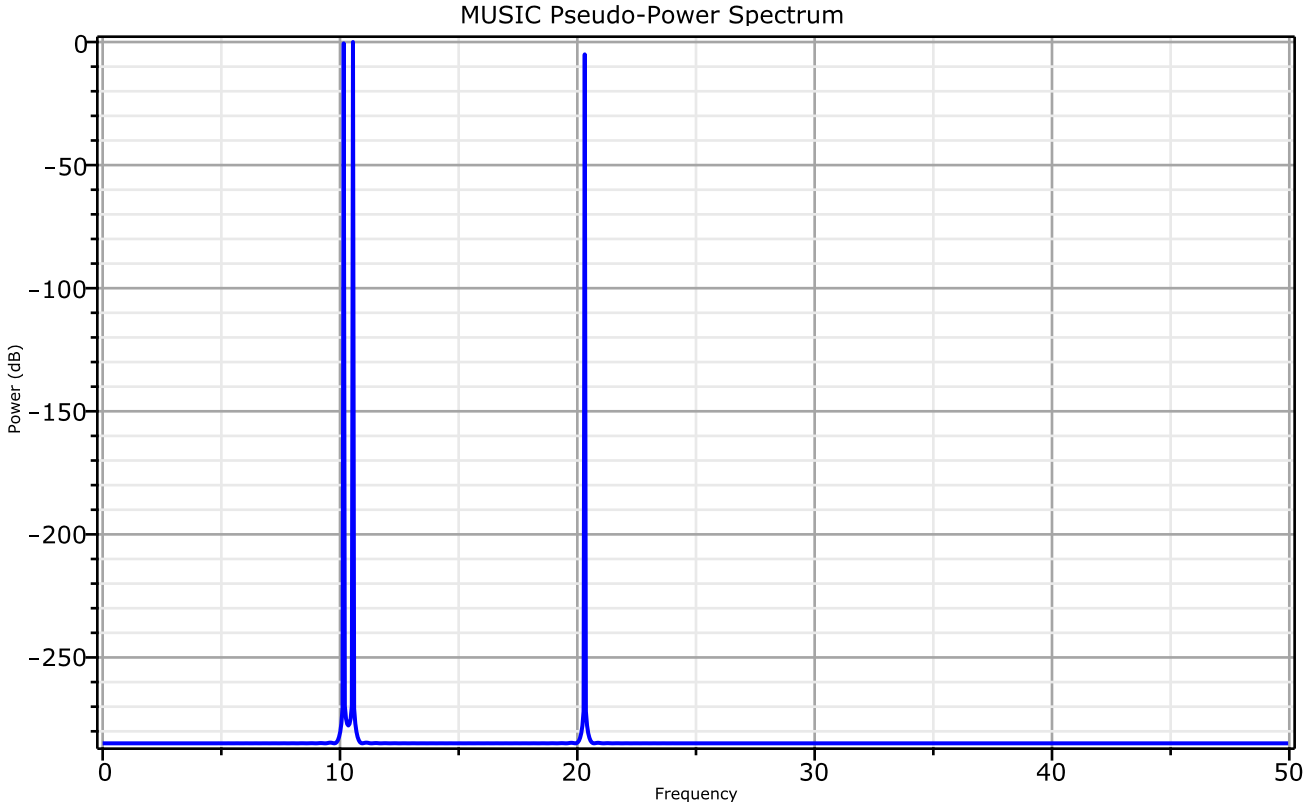



```
> DocumentTools:-Tabulate( Array( [
          Periodogram( X, 'samplerate' = fs, 'color' = 'blue',
  'title' = "Pure Signal Periodogram", 'font' = [ 'Verdana', 15 ]
  ),
          Periodogram( Y, 'samplerate' = fs, 'color' = 'blue',
  'title' = "Noisy Signal Periodogram", 'font' = [ 'Verdana', 15 ]
  )
  ] ) ):
```



The power spectra for both the clean and noisy signals fails to discriminate the two smaller frequencies. With the **MUSIC** command, however, we can detect all the frequencies:

```
> MUSIC( Y, 'samplerate' = fs, 'dimension' = 6, 'output' = 'plot'
  );
```



The peaks can also be returned, with the dominant peaks corresponding to the

frequencies of the sinusoids in the clean signal:

```
> MUSIC( Y, 'samplerate' = fs, 'dimension' = 6, 'output' = 'peaks'
  );
```

$$\begin{bmatrix} 10.5468750000000 & 1. \\ 10.1562500000000 & 0.881269822689514 \\ 20.3125000000000 & 0.314567906139360 \\ 11.1328125000000 & 3.45849990198759 \times 10^{-29} \\ 9.57031250000000 & 3.45836305214661 \times 10^{-29} \\ 19.7753906250000 & 3.40254084261547 \times 10^{-29} \\ 20.8496093750000 & 3.40232715742633 \times 10^{-29} \\ 11.5234375000000 & 3.32818070232832 \times 10^{-29} \\ 9.17968750000000 & 3.32798685701327 \times 10^{-29} \\ 19.3359375000000 & 3.30069539811017 \times 10^{-29} \\ 21.2890625000000 & 3.30026143196606 \times 10^{-29} \\ 11.9140625000000 & 3.29042011041287 \times 10^{-29} \\ 8.78906250000000 & 3.29016043634738 \times 10^{-29} \\ 18.9453125000000 & 3.27437738168937 \times 10^{-29} \\ 12.3046875000000 & 3.27414833617662 \times 10^{-29} \\ \vdots & \vdots \end{bmatrix}$$

126 × 2 Matrix

ShortTimeFourierTransform

The new ShortTimeFourierTransform command takes a signal, and, similar to the **Welch** command, divides it into overlapping segments of equal (or nearly equal) length, and for each segment, a window is applied and the Fourier Transform computed. For example, here we find the Short-Time Fourier Transform of a violin recording along with the spectrogram (which plots the corresponding Short-Time Power Spectrum versus time):

```
> with( AudioTools ):
```

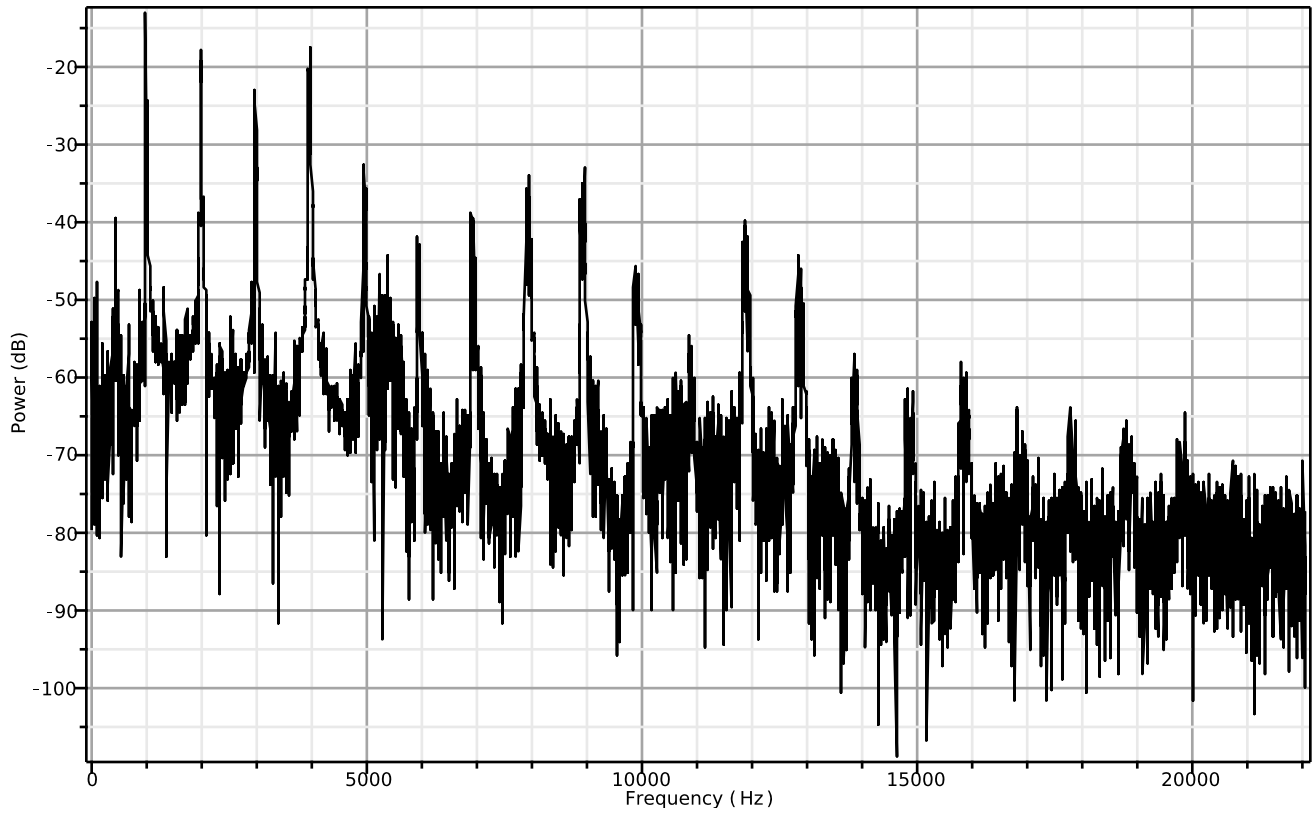
```
> file := cat( kernelopts( datadir ), kernelopts( dirsep ),
  "audio", kernelopts( dirsep ), "ViolinThreePosVibrato.wav" );
```

$$file := \text{"C:\Program Files\Maple Main\data\audio\ViolinThreePosVibrato.wav"}$$

```
> violin := ToMono( Read( file, samples = 1000 .. 10000 ) );
```

$$violin := \begin{bmatrix} \text{"Sample Rate"} & 44100 \\ \text{"File Format"} & PCM \\ \text{"File Bit Depth"} & 16 \\ \text{"Channels"} & 1 \\ \text{"Samples/Channel"} & 9001 \\ \text{"Duration"} & 0.20410\ s \end{bmatrix}$$

```
> Periodogram( violin, size = [800,400] );
```




```
> overlap_size := 128;
```

$$overlap\_size := 128$$

```
> segment_size := 256;
```

$$segment\_size := 256$$

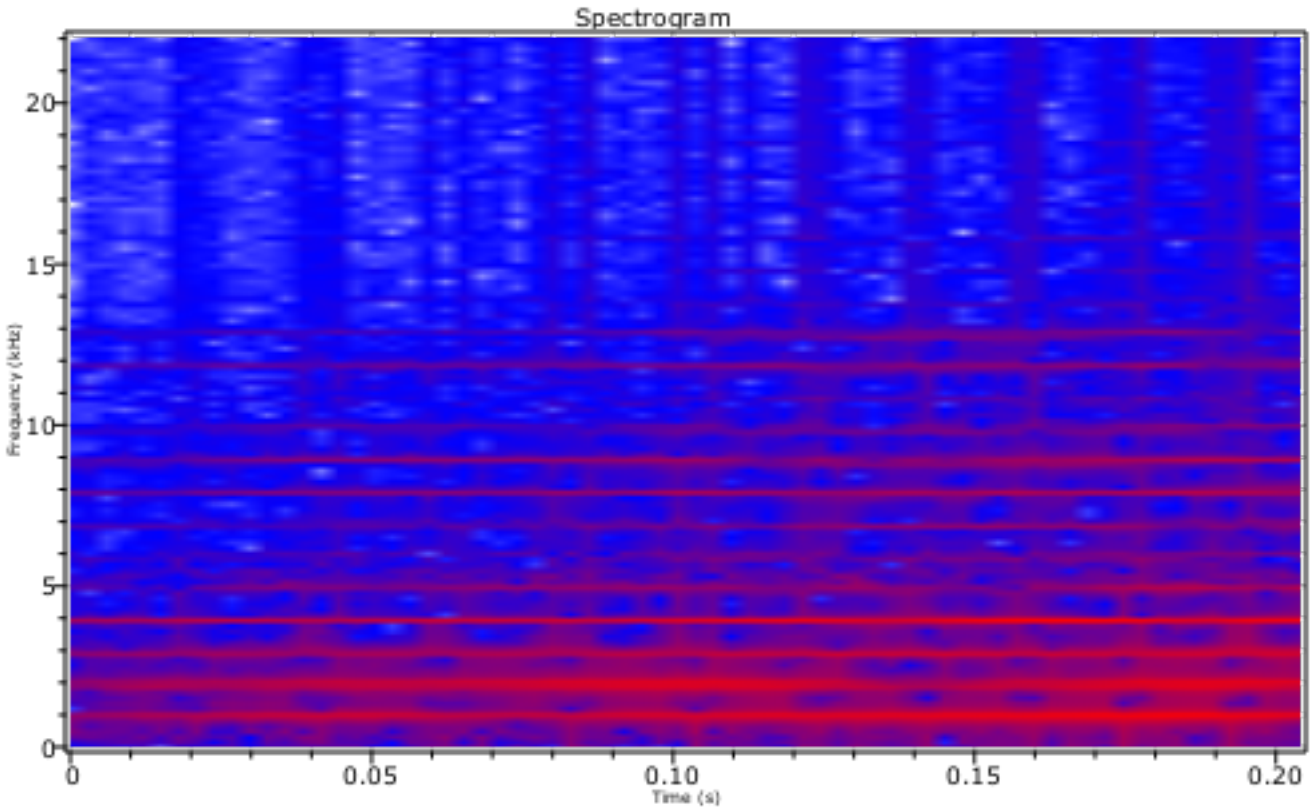
```
> stft_matrix, spectrogram_plot := ShortTimeFourierTransform(
      violin,
      overlapsize = overlap_size,
      segmentsize = segment_size,
      frequencyunit = kHz,
      powerscale = dB/Hz,
      output = [ stft, spectrogram ]
```

```
):
```

```
> stft_matrix;
```

$$\begin{bmatrix} -0.000284767843571211 + 0.\,\mathrm{I} & 0.000354240457761399 + 0.\,\mathrm{I} \\ 0.000405722563984783 + 0.000502560360044971\,\mathrm{I} & 0.000178136176584058 + 0.001044889312881 \\ 0.00240982051791025 - 0.000718320811860703\,\mathrm{I} & -0.000796655251439435 + 0.002829873561293 \\ -0.00461555825917711 - 0.00120171605264289\,\mathrm{I} & -9.99657404018396 \times 10^{-6} - 0.00272349372183 \\ -0.00354601852344845 - 0.000318707173647515\,\mathrm{I} & -0.00136534452681129 + 0.000412481249039 \\ -0.00428148108601939 - 0.00220915094833603\,\mathrm{I} & -0.00198610922431472 + 0.002112435577779 \\ 0.00690439080421307 + 0.000316734515267734\,\mathrm{I} & 0.00671272084833143 - 0.006945770467346 \\ 0.000843262274934301 + 3.13720999754288 \times 10^{-6}\,\mathrm{I} & 0.00130135793545737 - 0.001441838084164 \\ -0.00842335001995854 + 0.000621110443602900\,\mathrm{I} & 0.00165111272023407 + 0.000016253867467 \\ -0.00165600134787495 + 0.000920607382005798\,\mathrm{I} & 0.00123604826714257 + 0.000450408189386 \\ -0.00193335984967267 - 0.000545921025562424\,\mathrm{I} & 0.000472486137203692 + 0.001016423228481 \\ -0.00511304653266314 - 0.00129112911469463\,\mathrm{I} & -0.000379356517072000 + 0.004964359047382 \\ 0.00488297282645291 + 0.00176362059722691\,\mathrm{I} & 0.000394041589011674 - 0.006678064706198 \\ 0.00168616402984287 + 0.000870690180509329\,\mathrm{I} & 0.000607208575256186 - 0.002626313465626 \\ 0.000677076418487527 + 0.000562828549252059\,\mathrm{I} & 0.000817574941784041 - 0.001653544064523 \\ \vdots & \vdots \end{bmatrix}$$

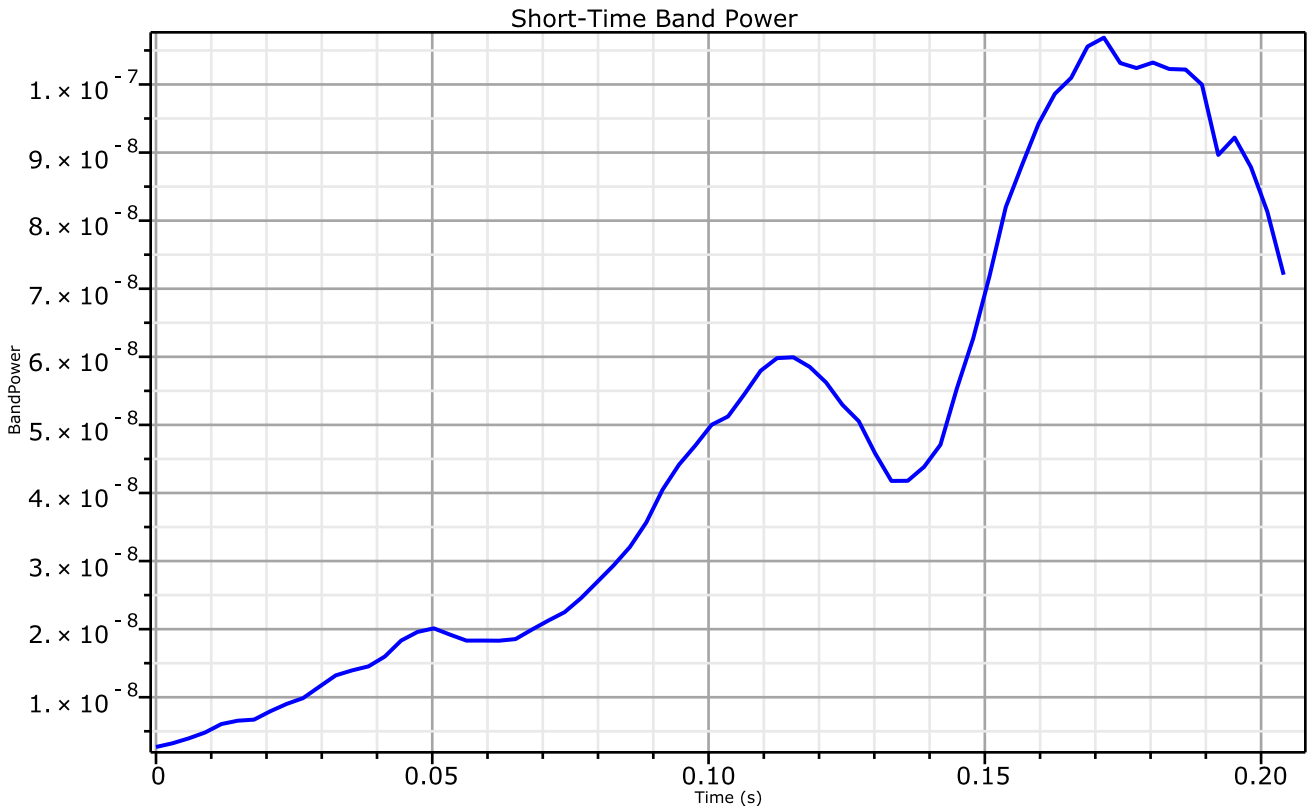
```
> spectrogram_plot;
```



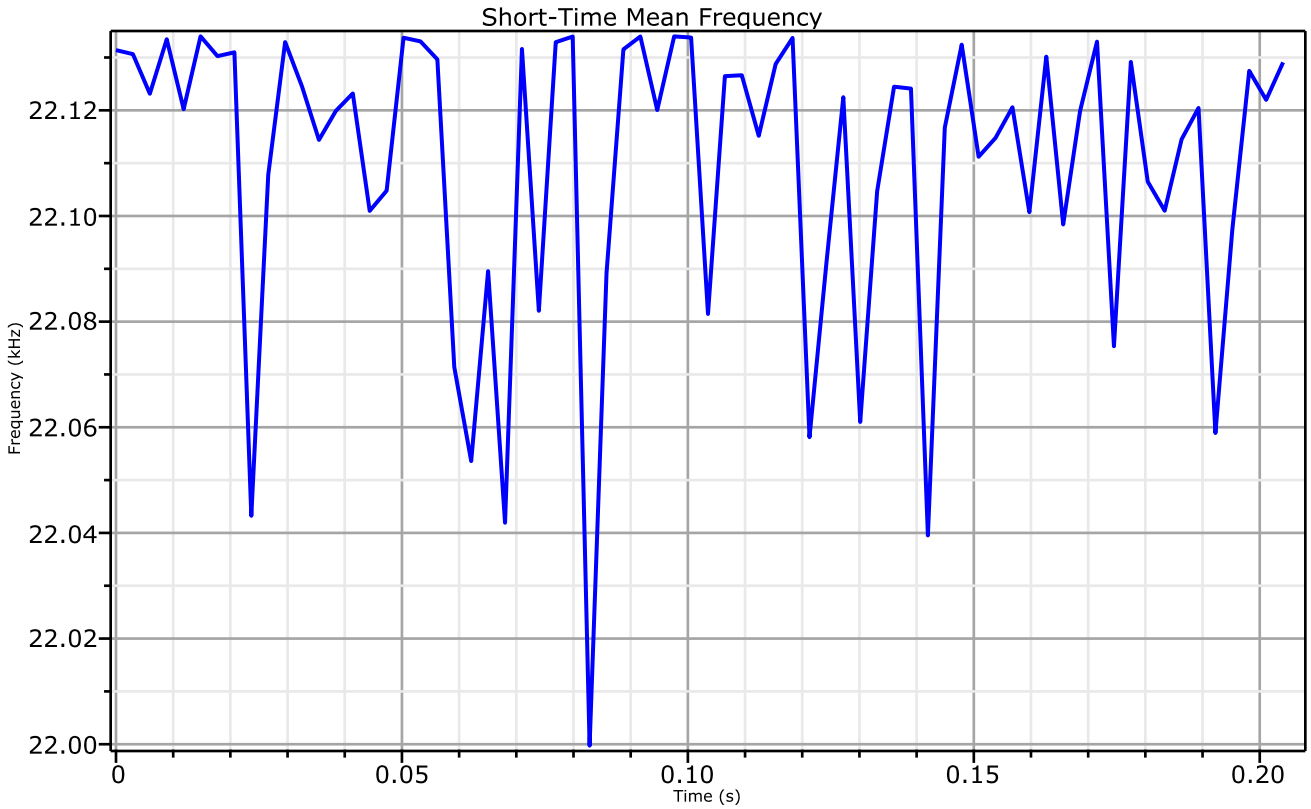
ShortTimeBandPower, ShortTimeMeanFrequency, and ShortTimeSpectralEntropy

The new commands ShortTimeBandPower, ShortTimeMeanFrequency, and ShortTimeSpectralEntropy apply the **ShortTimeFourierTransform** command, and compute the respective statistics for each short-time interval. Continuing the example above for the violin, we use the **ShortTimeBandPower** command to return everything, including plots, in a record:

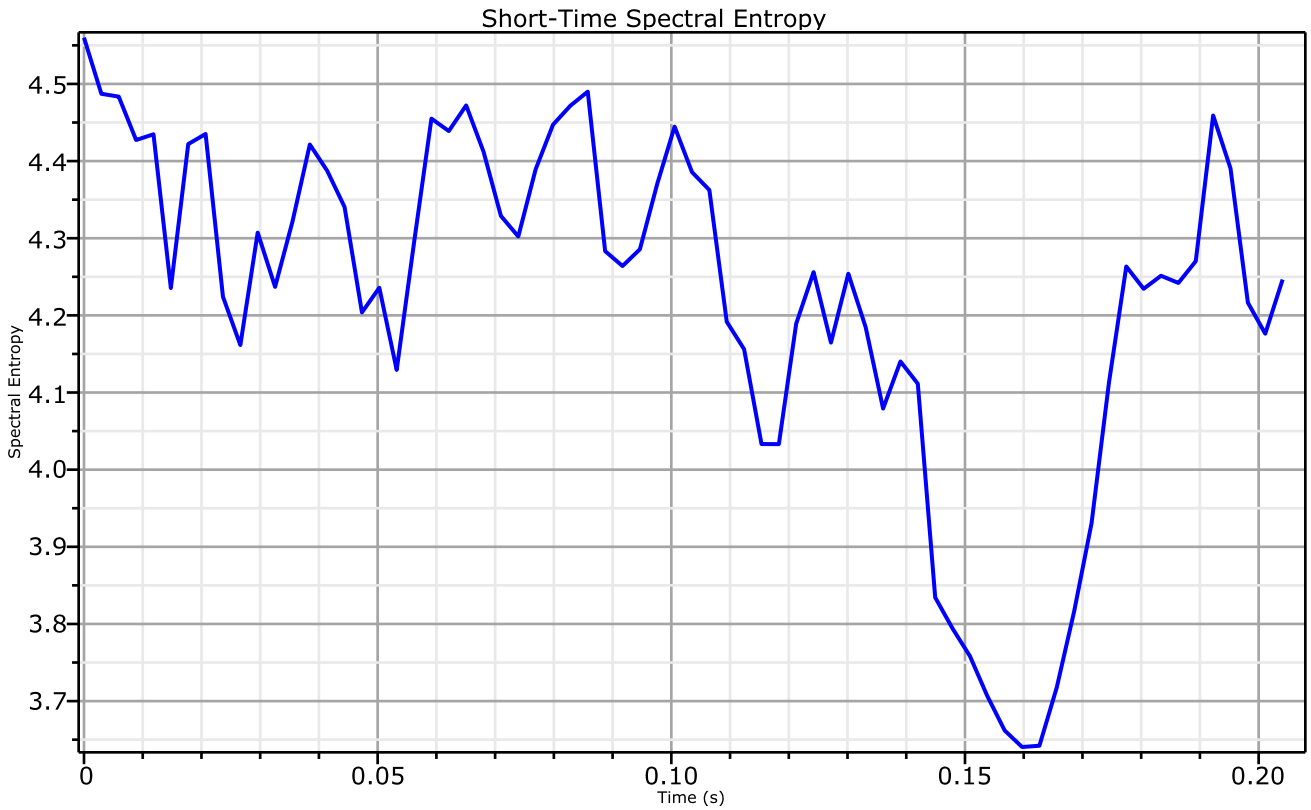
```
> R := ShortTimeBandPower(
      violin,
      overlapsize = overlap_size,
      segmentsize = segment_size,
      frequencyunit = kHz,
      output = record
  ):

> R[bandpowerplot];
```




```
> R[meanfrequencyplot];
```




```
> R[entropyplot];
```



FilterFrequencyResponse

The new FilterFrequencyResponse command takes one (for an FIR filter) or two (for an IIR filter) containers for taps, representing the coefficients of the transfer function, and returns the frequency response. For example:

```
> T := GenerateChebyshev1Taps( 5, 0.4, 29, filtertype = lowpass );
```

$$T := [0.0145256222951220, 0.0726281114756102, 0.145256222951220, 0.145256222951220, 0.0726281114756102, 0.0145256222951220, 1., 1.67764328574172, 1.12542851668154, -0.885659875792078, -1.51371985423053, -0.938872158956746]$$

```
> A, B := ListTools:-Slice( T, 2 );
```

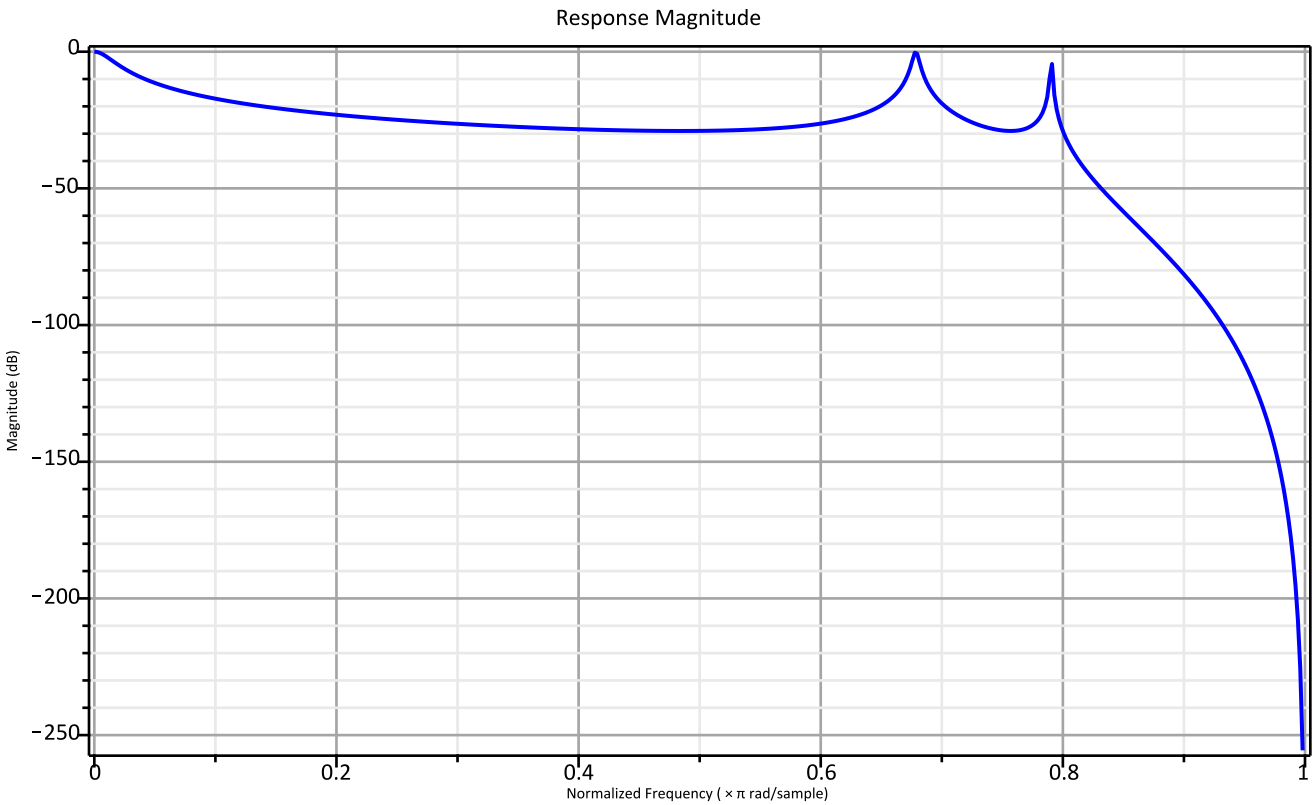
$$A, B := [0.0145256222951220, 0.0726281114756102, 0.145256222951220, 0.145256222951220, 0.0726281114756102, 0.0145256222951220], [1., 1.67764328574172, 1.12542851668154, -0.885659875792078, -1.51371985423053, -0.938872158956746]$$

```
> R := FilterFrequencyResponse( A, B, output = record ):

> 'response' = R[ response ];
```

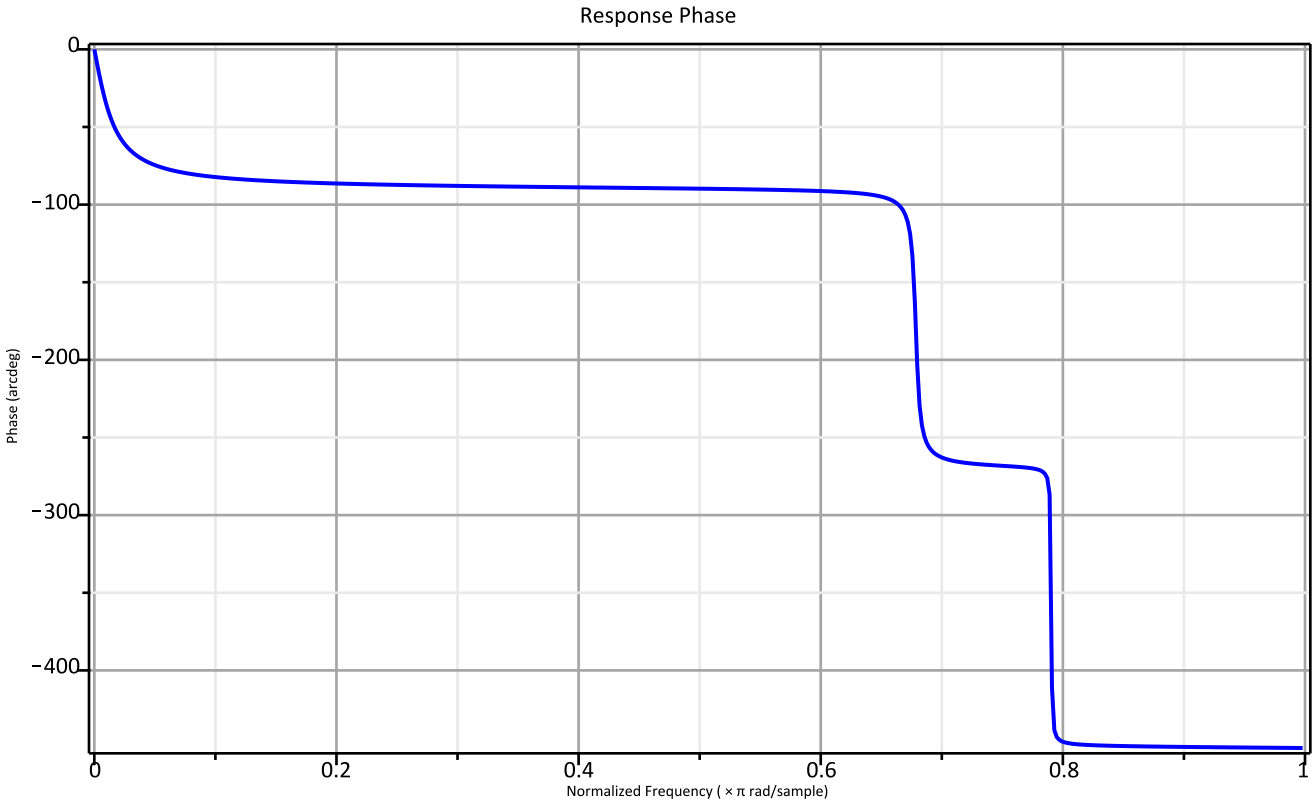
$$response = \begin{bmatrix} 1. + 0.\,I \\ 0.980663704326517 - 0.137737177143168\,I \\ 0.926893974330032 - 0.260376729044754\,I \\ 0.849279343943540 - 0.357875792846117\,I \\ 0.760157355320850 - 0.427119850276164\,I \\ 0.669779195091631 - 0.470458194661552\,I \\ 0.584787824361911 - 0.492957452192580\,I \\ 0.508513868401643 - 0.500159543609005\,I \\ 0.441982848190725 - 0.496887819131391\,I \\ 0.384896849200436 - 0.486869334770134\,I \\ 0.336332535023680 - 0.472785808653395\,I \\ 0.295158226411990 - 0.456478699328697\,I \\ 0.260251765981648 - 0.439169994190340\,I \\ 0.230598104989894 - 0.421646998035174\,I \\ 0.205321659671677 - 0.404401114549750\,I \\ \vdots \end{bmatrix}$$

512 element Vector[column]

```
> R[ magnitudeplot ];
```



```
> R[ phaseplot ];
```



EquivalentNoiseBandwidth

The EquivalentNoiseBandwidth command computes the equivalent-noise bandwidth of a window. For instance:

```
> EquivalentNoiseBandwidth( 10, ["Exponential",0.25], 1.5 );
```

$$0.173009806903817592$$

Hampel

The new Hampel command is useful when removing outliers from data. For an example, we will create a signal, add outliers, and then apply the filter:

```
> numpoints := 51;
```

$$numpoints := 51$$

```
> SignalOriginal := Vector( numpoints, i -> 5 + cos( 4 * Pi *
  (i-1) / (numpoints-1) ), datatype = float[8] ):

> SignalModified := copy( SignalOriginal ):
  SignalModified[3] := SignalOriginal[3] + 4.0:
  SignalModified[floor(numpoints/2)] := SignalOriginal[floor
```

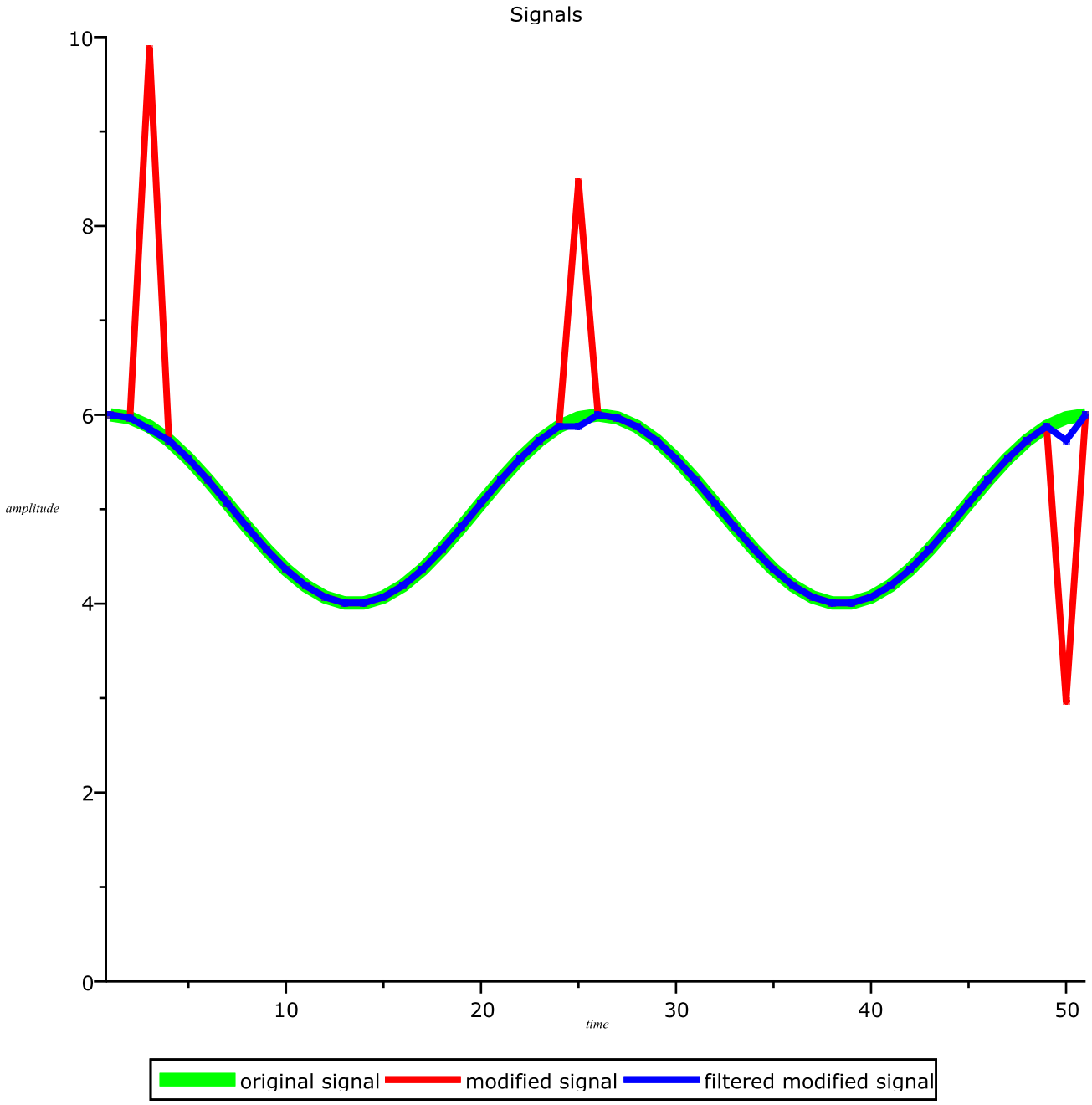
```
  (numpoints/2)] + 2.5:
  SignalModified[-2] := SignalModified[-2] - 3.0:
  'SignalModified' = SignalModified:
```

```
> SignalFiltered := Hampel( SignalModified, 3, 2.0, output =
  hampelsignal );
```

$$SignalFiltered := \begin{bmatrix} 6. \\ 5.96858316112863 \\ 5.84877589427502 \\ 5.72896862742141 \\ 5.53582679497900 \\ 5.30901699437495 \\ 5.06279051952931 \\ 4.81261868541428 \\ 4.57422070843493 \\ 4.36257601025131 \\ 4.19098300562505 \\ 4.07022351411175 \\ 4.00788529868552 \\ 4.00788529868552 \\ 4.07022351411175 \\ \vdots \end{bmatrix}$$

51 element Vector[column]

```
> dataplot(
      [ SignalOriginal, SignalModified, SignalFiltered ],
      view = [ 1 .. numpoints, 0 .. 10 ],
      labels = [time,amplitude],
      title = "Signals",
      font = [Verdana,15],
      legend = ["original signal","modified signal","filtered
  modified signal"],
      legendstyle = [font=[Verdana,15]],
      symbolsize = 5,
      color = ["green","red","blue"],
      thickness = [6,3,3]
  );
```



IntegrateData

The new IntegrateData command is used to estimate the area beneath a 1-D signal. For the example below, we start with a symbolic expression to create a signal, so that we can compare the symbolic integral with the numeric approximation:

```
> signal := t -> 5 + sin( t ) + 0.1 * sin( 3 * t ) + 0.01 * sin( 5
  * t );
```

$$signal := t \mapsto 5 + \sin(t) + 0.1 \cdot \sin(3 \cdot t) + 0.01 \cdot \sin(5 \cdot t)$$

```
> t1, t2 := 0.0, 10.0;
```

$$t1, t2 := 0., 10.0$$

```
> points := 100;
```

$$points := 100$$

```
> tValues := Vector( [ seq( t1 .. t2, numelems = points ) ],
  datatype = float[8] ):
```

```
> yValues := signal~( tValues ):
```

```
> area_symbolic := int( signal, t1 .. t2 );
```

$$area\_symbolic := 51.86733322$$

```
> area_numeric := IntegrateData( tValues, yValues, method =
  simpson );
```

$$area\_numeric := 51.8673373335523635$$

FindPeakPoints

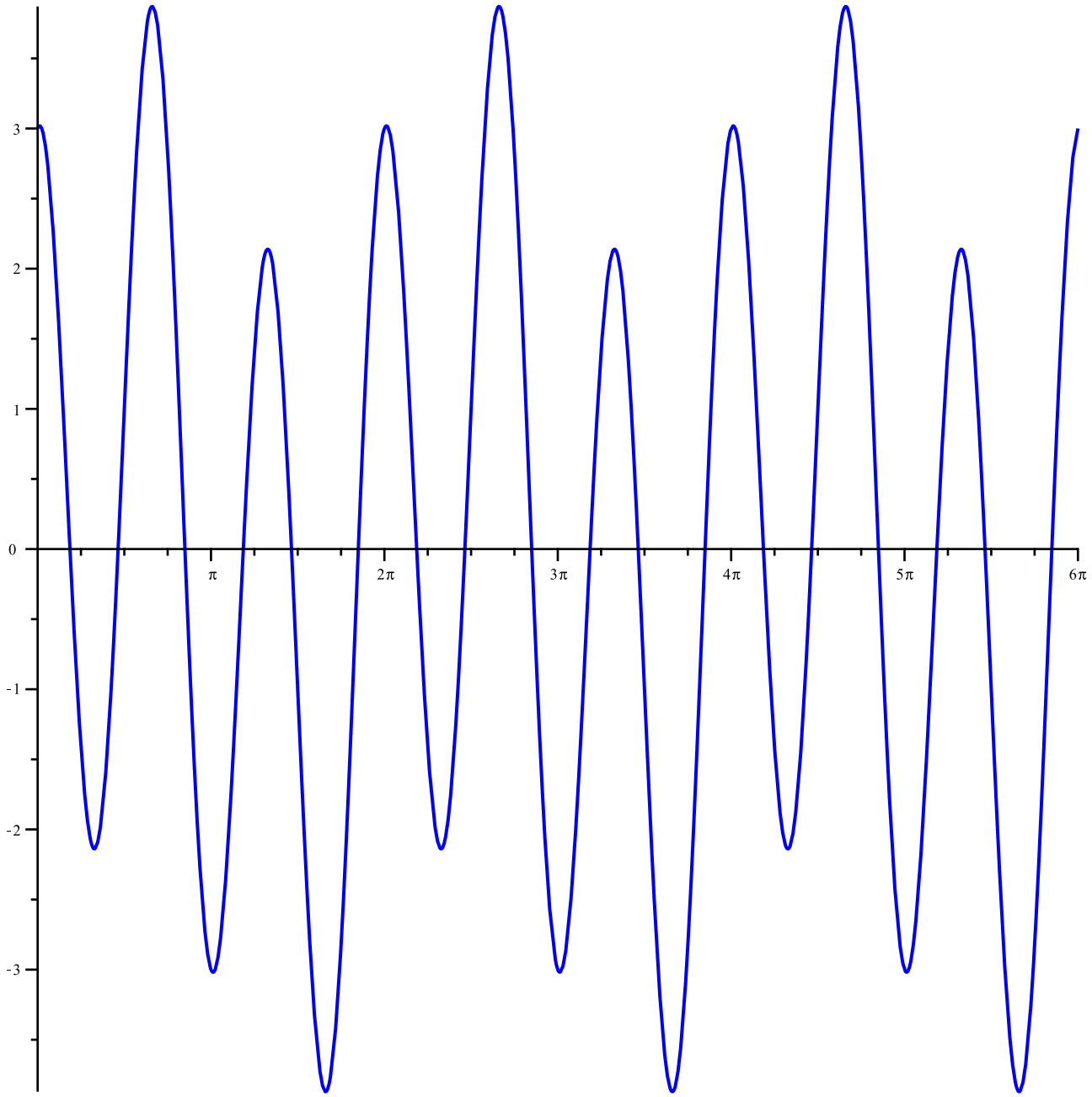
The FindPeakPoints command has been updated to include a new calling sequence, **FindPeakPoints(X,Y,...)**, and two new options, **maximumheight** and **sortdata** (which, when **false**, is used to skip sorting if the independent data are known to be sorted).

For example, consider the following signal:

```
> f := unapply( sin(t) + 3 * cos(3*t), t );
  a, b := 0, 6 * Pi;
  plot( f, a .. b, 'color' = 'blue' );
```

$$f := t \mapsto \sin(t) + 3 \cdot \cos(3 \cdot t)$$

$$a, b := 0, 6\,\pi$$



We will create data from this, and add noise:

```
> n := 10^4;
  T := Array( 1 .. n, i -> evalhf( (b-a) * (i-1) / (n-1) ),
  'datatype' = 'float[8]' );
  X := map['evalhf']( f, T );
  use Statistics in
            N := Array( Sample( RandomVariable( Normal( 0, 0.1 )
  ), n ) );
  end use;
```

$$n := 10000$$

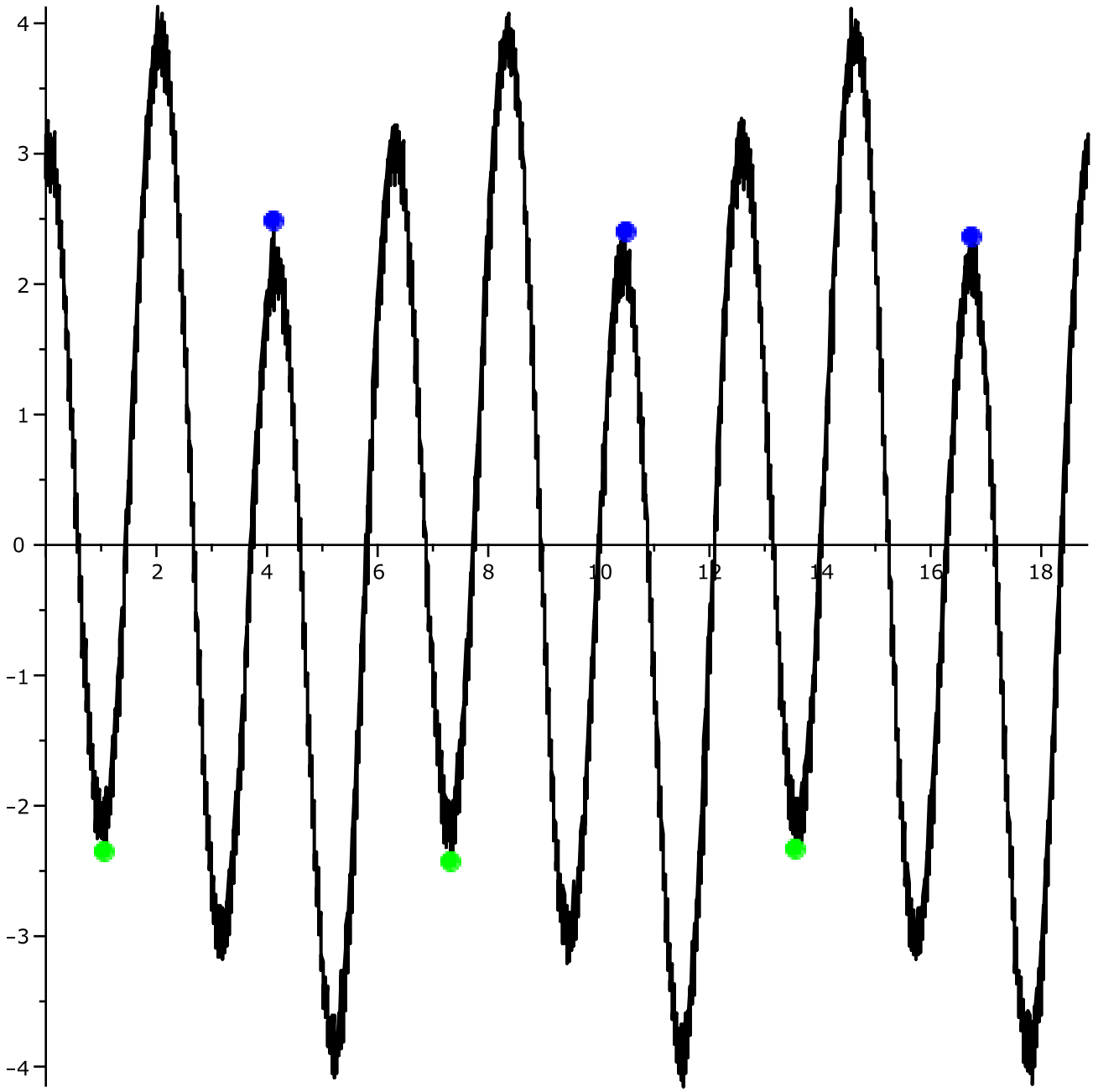
$T :=$ [0., 0.00188514410656453, 0.00377028821312906, 0.00565543231969360,
0.00754057642625813, 0.00942572053282266, 0.0113108646393872, 0.0131960087459517,
0.0150811528525163, 0.0169662969590808, 0.0188514410656453, 0.0207365851722099,
0.0226217292787744, 0.0245068733853389, 0.0263920174919035,…,
⋯ 9985 Array entries not shown]

$X :=$ [3., 3.00183716724579, 3.00357837783825, 3.00522362968132, 3.00677292374773,
3.00822626407870, 3.00958365778366, 3.01084511503986, 3.01201064909178,
3.01308027625058, 3.01405401589338, 3.01493189046239, 3.01571392546410,
3.01640014946819, 3.01699059410647,…, ⋯ 9985 Array entries not shown]

$N :=$ [−0.0694815815002697, −0.0981127030334508, 0.0353556486970789,
−0.0416781424326070, 0.0529635817228773, 0.0255379094424754, 0.148803714420798,
−0.0379943039798480, −0.197647289992132, 0.0613583653411600,
−0.0316518739512218, −0.00645411216724587, −0.0776930001573345,
−0.0369122603779730, 0.235359838126552,…, ⋯ 9985 Array entries not shown]

The original signal has peaks of three different heights, and valleys of three different heights. Here, let's identify the smaller peaks and larger valleys in the noisy data:

```
> FindPeakPoints(
            T,
            X + N,
            'sortdata' = 'false',
            'maximumheight' = 2.5,
            'minimumheight' = -2.5,
            'minimumprominence' = 0.75,
            'output' = 'plot',
            'plotincludepoints' = ['peaks','valleys'],
            'font' = ['Verdana',15]
  );
```



ComplexToReal and RealToComplex

The new ComplexToReal and RealToComplex commands, respectively, combine containers with the real and imaginary parts into a single container with the complex values, and decompose a container with complex values into separate containers with the real and imaginary parts. For instance:

```
> RealParts := LinearAlgebra:-RandomVector( 3, datatype = float[8]
  );
  ImaginaryParts := LinearAlgebra:-RandomVector( 3, datatype =
  float[8] );
```

$$RealParts := \begin{bmatrix} 4. \\ 92. \\ -17. \end{bmatrix}$$

$$ImaginaryParts := \begin{bmatrix} 59. \\ -20. \\ -99. \end{bmatrix}$$

```
> ComplexValues := RealToComplex( RealParts, ImaginaryParts );
```

$$ComplexValues := \begin{bmatrix} 4. + 59.\,\mathrm{I} \\ 92. - 20.\,\mathrm{I} \\ -17. - 99.\,\mathrm{I} \end{bmatrix}$$

```
> ComplexToReal( ComplexValues );
```

$$\begin{bmatrix} 4. \\ 92. \\ -17. \end{bmatrix}, \begin{bmatrix} 59. \\ -20. \\ -99. \end{bmatrix}$$

The operations are optimized for hardware floats and large rtables, and existing containers can be passed for storage (using the **container** option for **RealToComplex**, and **containers** option for **ComplexToReal**).

RootMeanSquare

The RootMeanSquare command now supports multidimensional rtables and lists when computing the Root Mean Square (RMS). For instance:

```
> M := Matrix( [ [ 1, 2 ], [ 3, 4 ] ], datatype = float[8] );
```

$$M := \begin{bmatrix} 1. & 2. \\ 3. & 4. \end{bmatrix}$$

```
> RootMeanSquare( M );
```

$$2.73861278752583059$$

```
> L := [ 10, 15, 20 ];
```

$$L := [10, 15, 20]$$

```
> RootMeanSquare( L );
```

$$15.5456317551480261$$

RootMeanSquareError and RelativeRootMeanSquareError

The new RootMeanSquareError and RelativeRootMeanSquareError commands are useful for quantifying errors. Specifically, the Root Mean Square Error (RMSE) of two containers **X** and **Y** is the RMS of the difference **X-Y**, and the Relative Root Mean Square Error (RRMSE) is the RMS of **X-Y** divided by the RMS of **Y**. For example:

```
> RootMeanSquareError( < 1.1, 1.9, 3.1 >, [ 1, 2, 3 ] );
```

$$0.10000000000000103$$

```
> P := < 0.00003, 0.00004 >;
  Q := < 0.00001, 0.00002 >;
```

$$P := \begin{bmatrix} 0.00003 \\ 0.00004 \end{bmatrix}$$

$$Q := \begin{bmatrix} 0.00001 \\ 0.00002 \end{bmatrix}$$

```
> RootMeanSquareError( P, Q );
  RelativeRootMeanSquareError( P, Q );
```

$$0.000019999999999999982$$

$$1.26491106406735154$$

Mean

The Mean command in SignalProcessing now supports multidimensional containers and weights. For example:

```
> A := Matrix( [ [ 5, 2 ], [ -1, 8 ] ], datatype = float[8] );
```

$$A := \begin{bmatrix} 5. & 2. \\ -1. & 8. \end{bmatrix}$$

```
> Mean( A );
```

$$3.50000000000000000$$

```
> W := Matrix( [ [ 1, 2 ], [ 3, 4 ] ], datatype = float[8] );
```

$$W := \begin{bmatrix} 1. & 2. \\ 3. & 4. \end{bmatrix}$$

```
> Mean( A, W );
```

$$3.79999999999999982$$

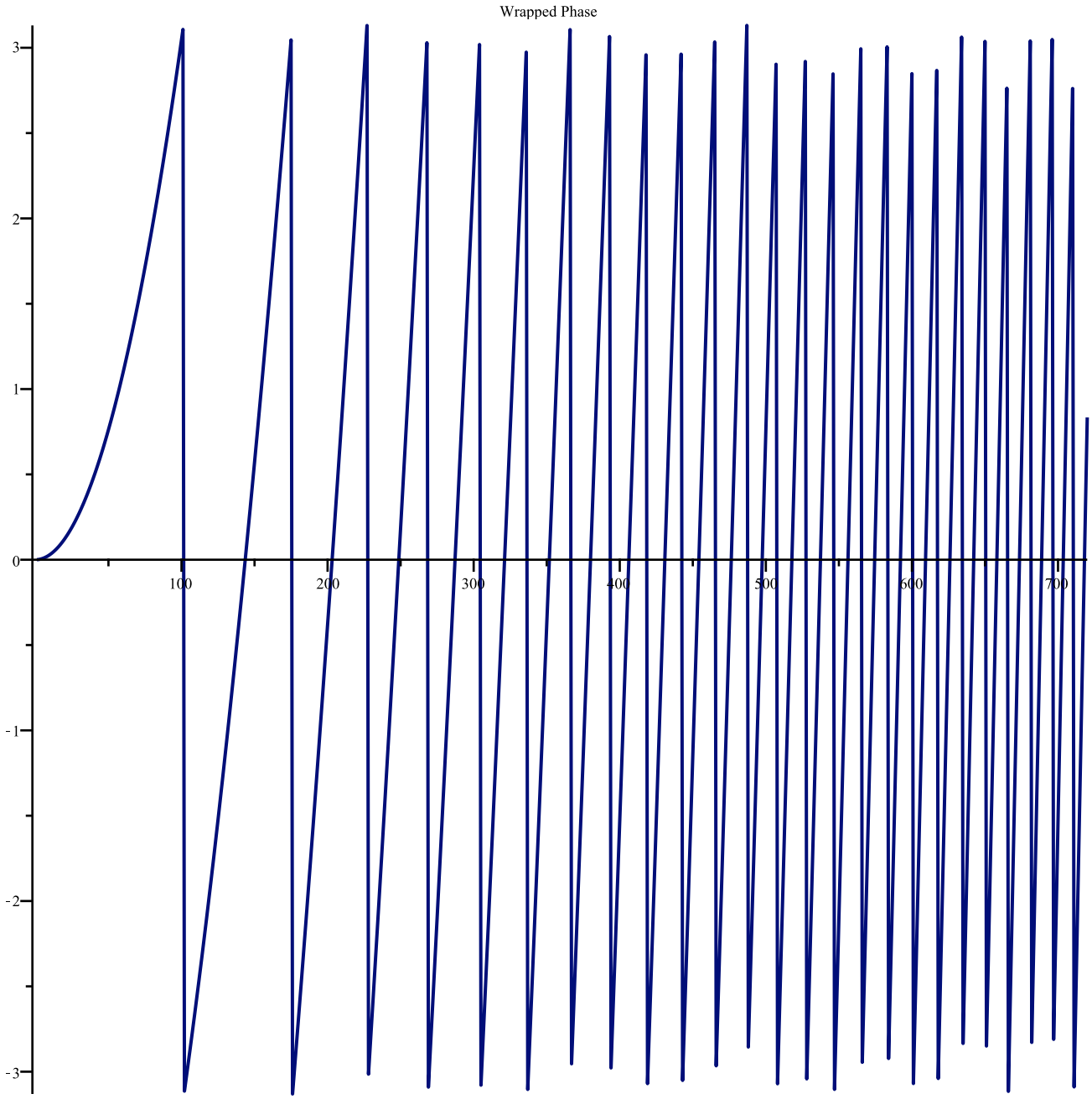
Phase

The Phase command in SignalProcessing now includes an option for unwrapping the phases, so that there are no large jumps:

```
> Z := Vector( 720, k -> k * exp( I * ( Pi / 180 * k )^2 ),
  datatype = complex[8] ):
```

```
> P := Phase( Z ):
```

```
> dataplot( P, style = line, title = "Wrapped Phase" );
```

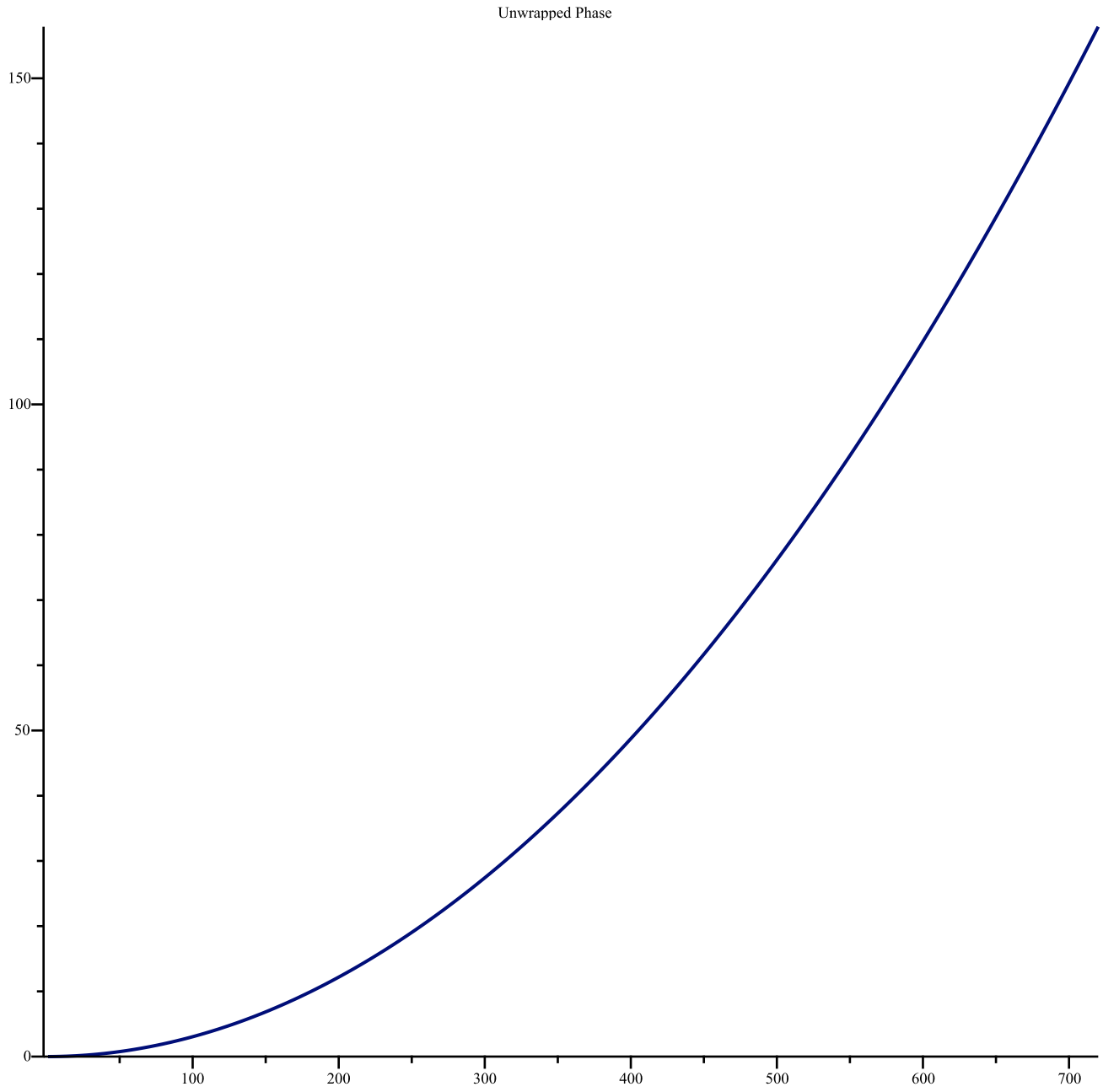



```
> Q := Phase( Z, unwrap );
```

$$Q := \begin{bmatrix} 0.000304617419786709 \\ 0.00121846967914683 \\ 0.00274155677808038 \\ 0.00487387871658734 \\ 0.00761543549466771 \\ 0.0109662271123215 \\ 0.0149262535695487 \\ 0.0194955148663493 \\ 0.0246740110027234 \\ 0.0304617419786709 \\ 0.0368587077941917 \\ 0.0438649084492860 \\ 0.0514803439439537 \\ 0.0597050142781949 \\ 0.0685389194520094 \\ \vdots \end{bmatrix}$$

720 element Vector[column]

```
> dataplot( Q, style = line, title = "Unwrapped Phase" );
```



ImageTools

The SampleImage command in the ImageTools package returns the requested image from a repository of sample images. For example:

```
> # Current number of available images in the repository.
  num_images := SampleImage( 0 );
```

$$num\_images := 5$$

```
> Embed( SampleImage( 1 ) );
```

```
> Embed( SampleImage( 5 ) );
```