Signal Processing

The SignalProcessing package has been expanded with new and updated commands.

```
> restart;
```

SignalProcessing

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

GenerateSignal

The new GenerateSignal command is useful for creating signals, filters, and windows from algebraic expressions. For example:

```
> X := GenerateSignal( 3 * sin(t) + cos(2*t), t = 0 .. 4 * Pi, 100
  );
```

$$X := \begin{bmatrix} 1. \\ 1.34772606211760 \\ 1.62729333861302 \\ 1.83872140508605 \\ 1.98581567591191 \\ 2.07564416249220 \\ 2.11781895027008 \\ 2.12363272481008 \\ 2.10510967724277 \\ 2.07403525211827 \\ \vdots \end{bmatrix}$$

100 element Vector[column]

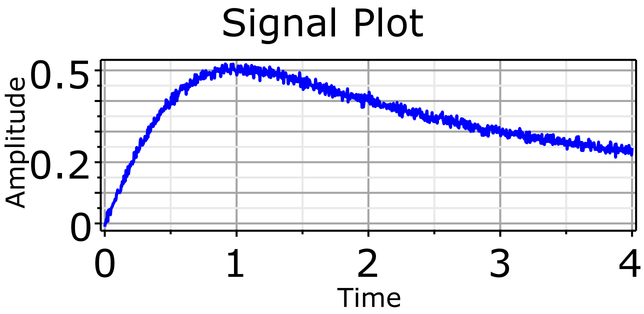
The command also packages many related features and data for the signal:

```
> sample_rate, Times, Signal := GenerateSignal( t * exp(-t), t = 0
  .. 5, 10^3, 'output' = ['samplerate','times','signal'] );
```

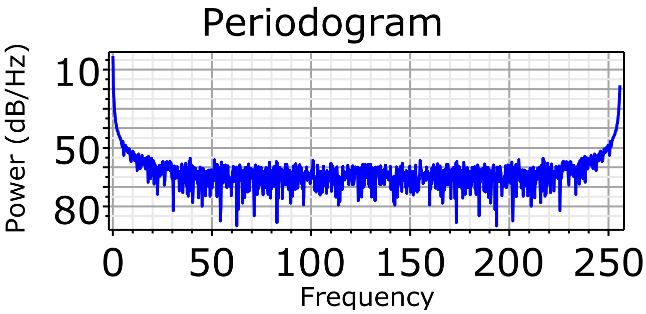
```
                                                          [         0.          ]
                                                          [ 0.00500500500500500 ]
                                                          [ 0.0100100100100100  ]
                                                          [ 0.0150150150150150  ]
                                                          [ 0.0200200200200200  ]
sample_rate, Times, Signal := 199.80000000000011,         [ 0.0250250250250250  ] ,
                                                          [ 0.0300300300300300  ]
                                                          [ 0.0350350350350350  ]
                                                          [ 0.0400400400400400  ]
                                                          [ 0.0450450450450450  ]
                                                          [          ⋮          ]
                                                          1000 element Vector[column]

    [         0.          ]
    [ 0.00498001751332686 ]
    [ 0.00991030954344368 ]
    [ 0.0147912484721253  ]
    [ 0.0196232042023248  ]
    [ 0.0244065441736429  ]
    [ 0.0291416333777046  ]
    [ 0.0338288343734441  ]
    [ 0.0384685073022971  ]
    [ 0.0430610099033036  ]
    [          ⋮          ]
    1000 element Vector[column]
```

```
> R := GenerateSignal( t / (1+t^2), t = 0 .. 4, 2^10, 'noisetype' =
  'additive', 'noisedeviation' = 0.01, 'output' = 'record' ):
```

```
> R['signalplot'];
```



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



DynamicTimeWarping

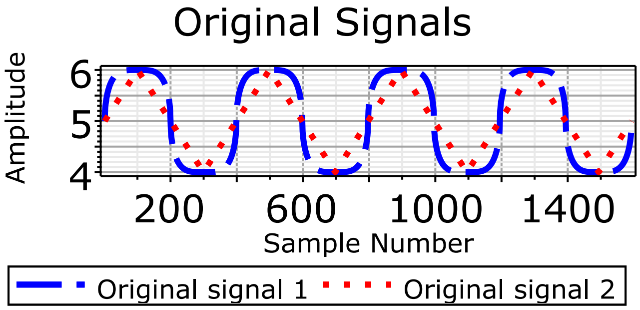
The DynamicTimeWarping command, which has many applications including speech recognition and genetic sequencing, determines the best match between two signals by varying the sampling rates dynamically. For example:

```
> X := GenerateSignal( 5 + sqrt(1-(t-1)^4), t = 0 .. 2, 200,
  'mirror' = 'antisymmetric', 'copies' = 4 ):
```

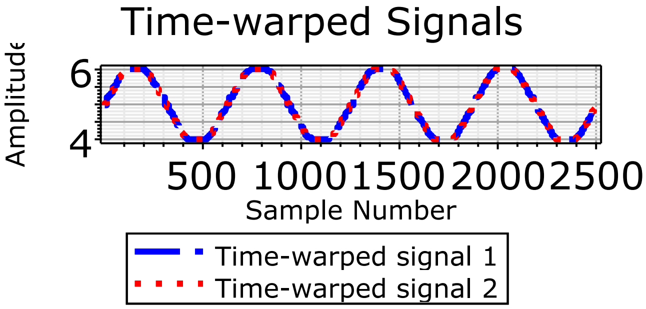
```
> Y := GenerateSignal( 6 - abs(t-1), t = 0 .. 2, 200, 'mirror' =
  'antisymmetric', 'copies' = 4 ):
```

```
> R := DynamicTimeWarping( X, Y, 'compiled', 'output' = 'record' ):
```

```
> R['unwarpedplot'];
```



```
> R['warpedplot'];
```



The match is computed by inserting zero or more copies of each sample for both signals in such a way that the cost with respect to the metric (taxicab, by default) is minimal.

For this example:

```
> R['cost'];
```

$$28.7505811441736654$$

In terms of the root mean square error:

```
> R['rmse'];
```

$$0.0220493272388582497$$

DifferentiateData

The new DifferentiateData command offers the standard methods, namely Backward, Central, and Forward Difference, and also features spectral differentiation. For example:

```
> f := piecewise( t < Pi  or t > 3 * Pi, sin(4*t) + 5, 1/10 * sin
  (40*t) + 5 );
  a, b := 0, 4 * Pi;
  n := 2^14;
```

$$f := \begin{cases} \sin(4\,t) + 5 & t < \pi \textbf{ or } 3\,\pi < t \\ \frac{\sin(40\,t)}{10} + 5 & \textit{otherwise} \end{cases}$$

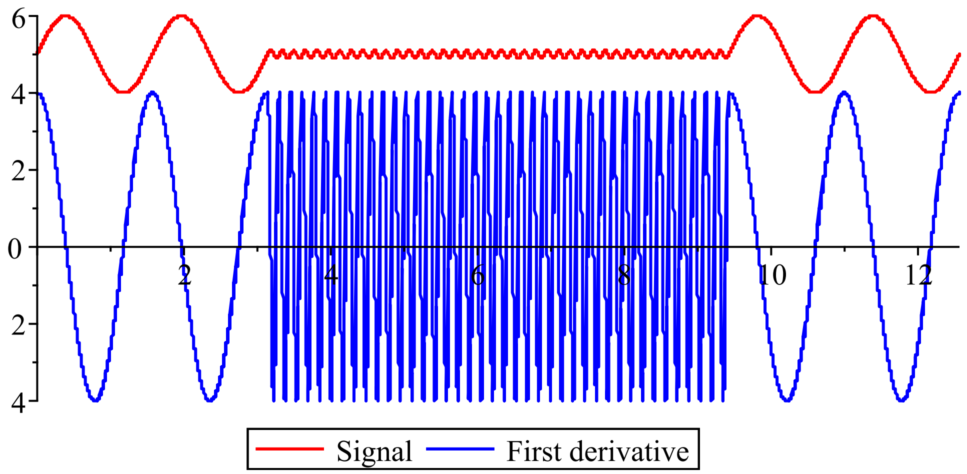
$$a, b := 0, 4\,\pi$$

$$n := 16384$$

```
> ( dt, T, X ) := GenerateSignal( f, t = a .. b, n,
  'includefinishtime' = 'false', 'output' = ['timestep','times',
  'signal'] ):
```

```
> DX := DifferentiateData( X, 1, 'step' = dt, 'method' =
  'spectral', 'extrapolation' = 'periodic' ):
```

```
> dataplot( T, [X,DX], 'style' = 'line', 'color' = ['red','blue'],
  'legend' = ["Signal","First derivative"], 'size' = [500,250] );
```



IntegrateData and IntegrateData2D

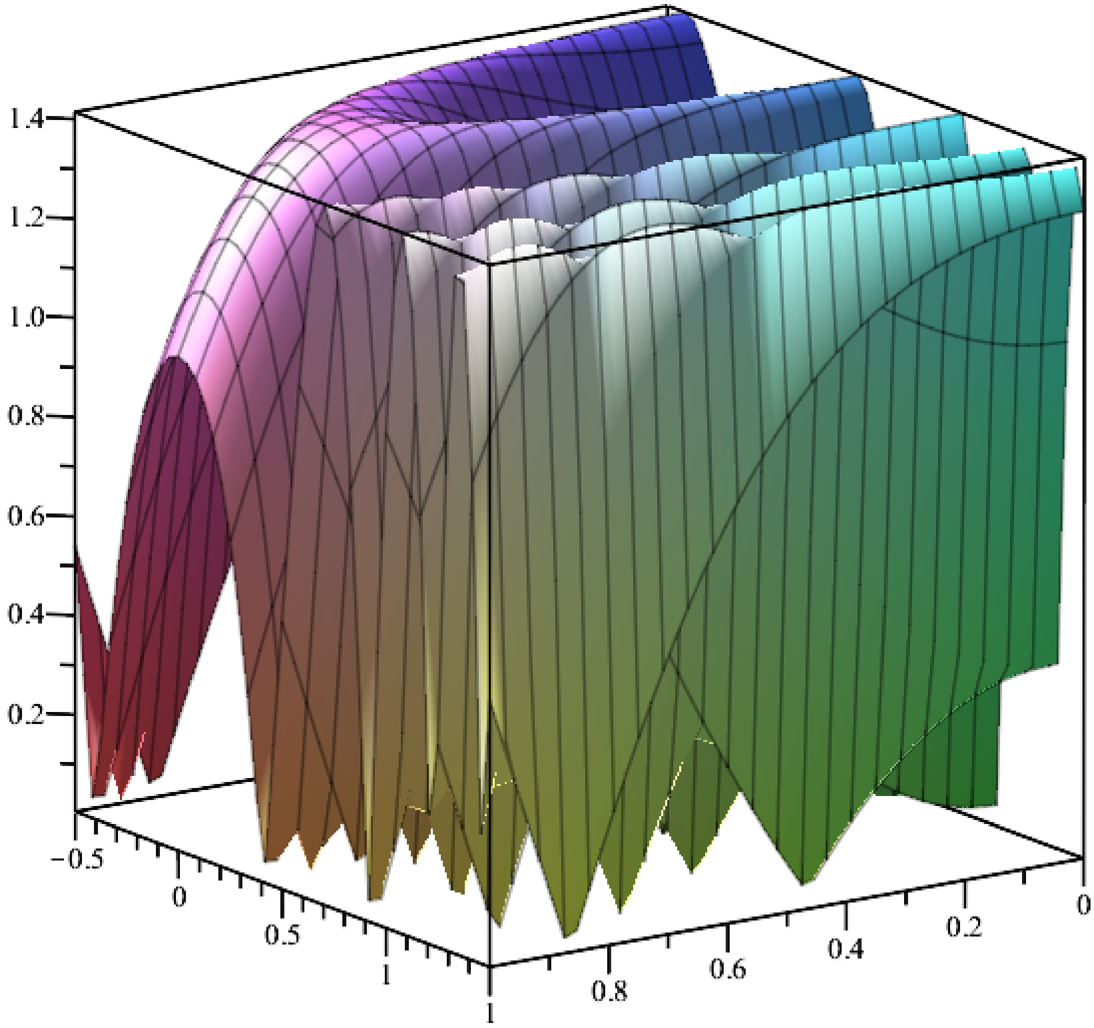
The IntegrateData2D command provides a fast and accurate way of finding the volume under a two-dimensional dataset. For example:

```
> f := (x,y) -> sqrt( 1 + sin( Pi * ( x^2 + 3 * y^2 ) ) );
  a, b, c, d := 0.0, 1.0, -0.5, 1.5;
```

$$f := (x, y) \quad \sqrt{1 + \sin\left(\pi\cdot\left(x^2 + 3\cdot y^2\right)\right)}$$

$$a, b, c, d := 0., 1.0, -0.5, 1.5$$

```
> plot3d( f, a .. b, c .. d );
```



```
> m, n := 100, 100;
  dx, dy := evalhf( (b-a) / (m-1) ), evalhf( (d-c) / (n-1) );
```

$$m, n := 100, 100$$

$$dx, dy := 0.0101010101010101019, 0.0202020202020202037$$

```
> X := Vector( m, i -> evalhf( a + (i-1) * dx ), 'datatype' =
  'float[8]' ):
  Y := Vector( n, j -> evalhf( c + (j-1) * dy ), 'datatype' =
  'float[8]' ):
  Z := Matrix( m, n, (i,j) -> evalhf( f( X[i], Y[i] ) ), 'datatype'
  = 'float[8]' );
```

```
       [ 1.30656296487638  1.30656296487638  1.30656296487638  1.30656296487638  1.30656296487638  1.3
       [ 1.35122406134324  1.35122406134324  1.35122406134324  1.35122406134324  1.35122406134324  1.3
       [ 1.38295754218013  1.38295754218013  1.38295754218013  1.38295754218013  1.38295754218013  1.3
       [ 1.40302880476168  1.40302880476168  1.40302880476168  1.40302880476168  1.40302880476168  1.4
       [ 1.41276686457614  1.41276686457614  1.41276686457614  1.41276686457614  1.41276686457614  1.4
Z :=   [ 1.41352474569771  1.41352474569771  1.41352474569771  1.41352474569771  1.41352474569771  1.4
       [ 1.40664653803368  1.40664653803368  1.40664653803368  1.40664653803368  1.40664653803368  1.4
       [ 1.39344080846178  1.39344080846178  1.39344080846178  1.39344080846178  1.39344080846178  1.3
       [ 1.37515989082796  1.37515989082796  1.37515989082796  1.37515989082796  1.37515989082796  1.3
       [ 1.35298447186640  1.35298447186640  1.35298447186640  1.35298447186640  1.35298447186640  1.3
       [        ⋮                 ⋮                 ⋮                 ⋮                 ⋮
```

```
> IntegrateData2D( X, Y, Z, 'uniform', 'compiled', 'method' =
  'simpson' );
                                   2.09312887784632640
```

This command supports both uniform (regular) and non-uniform (irregular) data. The option **uniform** tells the algorithm to skip the check for uniformity and use the uniform version of Simpson's Rule, which is quicker. The existing (added in Maple 2021) one-dimensional version of the command, IntegrateData, has been updated to include the **compiled** and **uniform** options.

RealPart and ImaginaryPart

The new RealPart and ImaginaryPart commands complement the existing ComplexToReal command. They, respectively, take a **complex[8]** container and quickly create a **float[8]** container for the real part and imaginary part.

For example:

```
> X := LinearAlgebra:-RandomVector( 5, 'generator' = -1.0 - 1.0 * I
  .. 1.0 + 1.0 * I, 'datatype' = 'complex[8]' );
```

$$X := \begin{bmatrix} -0.170954922213783 - 0.0703201167497254\,\mathrm{I} \\ 0.998983240195409 - 0.424301310369726\,\mathrm{I} \\ 0.0799641980758583 + 0.413834838645526\,\mathrm{I} \\ -0.763689603106579 + 0.976835857569961\,\mathrm{I} \\ -0.338342009593391 + 0.796972275668599\,\mathrm{I} \end{bmatrix}$$

```
> Y := RealPart( X );
```

$$Y := \begin{bmatrix} -0.170954922213783 \\ 0.998983240195409 \\ 0.0799641980758583 \\ -0.763689603106579 \\ -0.338342009593391 \end{bmatrix}$$

```
> Z := ImaginaryPart( X );
```

$$Z := \begin{bmatrix} -0.0703201167497254 \\ -0.424301310369726 \\ 0.413834838645526 \\ 0.976835857569961 \\ 0.796972275668599 \end{bmatrix}$$

Of course, if you need both the real and imaginary parts, it is best to use the **ComplexToReal** command:

```
> ComplexToReal( X );
```

$$\begin{bmatrix} -0.170954922213783 \\ 0.998983240195409 \\ 0.0799641980758583 \\ -0.763689603106579 \\ -0.338342009593391 \end{bmatrix}, \begin{bmatrix} -0.0703201167497254 \\ -0.424301310369726 \\ 0.413834838645526 \\ 0.976835857569961 \\ 0.796972275668599 \end{bmatrix}$$

Insert

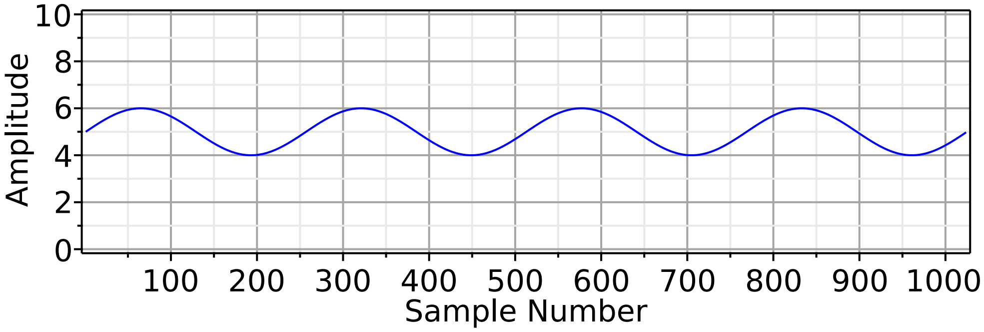
The new Insert command can take one signal, and insert it into another at any point. For example:

```
> n := 2^10;
```

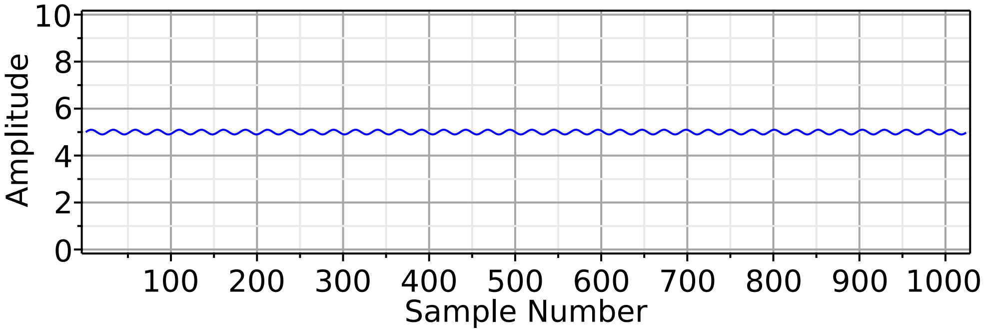
$$n := 1024$$

```
> X := GenerateSignal( sin(4*t) + 5, t = 0 .. 2 * Pi, n,
  'includefinishtime' = 'false' ):
```

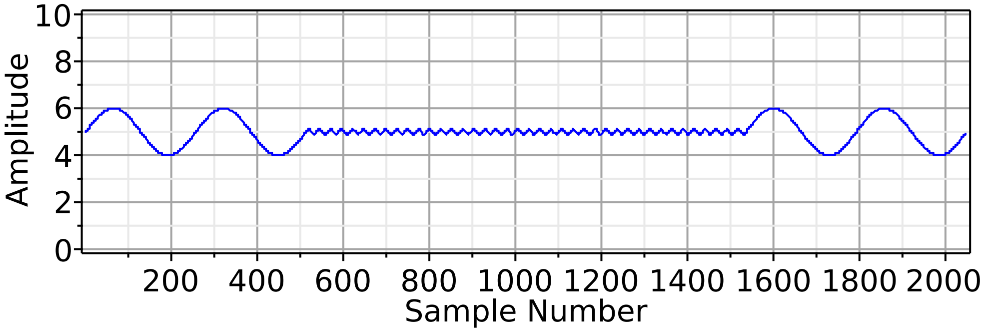
```
> SignalPlot( X, 'view' = ['DEFAULT',0..10], 'color' = 'blue' );
```



```
> Y := GenerateSignal( 1/10 * sin(40*t) + 5, t = 0 .. 2 * Pi, n,
  'includefinishtime' = 'false' ):
> SignalPlot( Y, 'view' = ['DEFAULT',0..10], 'color' = 'blue' );
```



```
> Insert( X, floor(n/2) + 1, Y, 'inplace' ):
> SignalPlot( X, 'view' = ['DEFAULT',0..10], 'color' = 'blue' );
```



FindPeakPoints

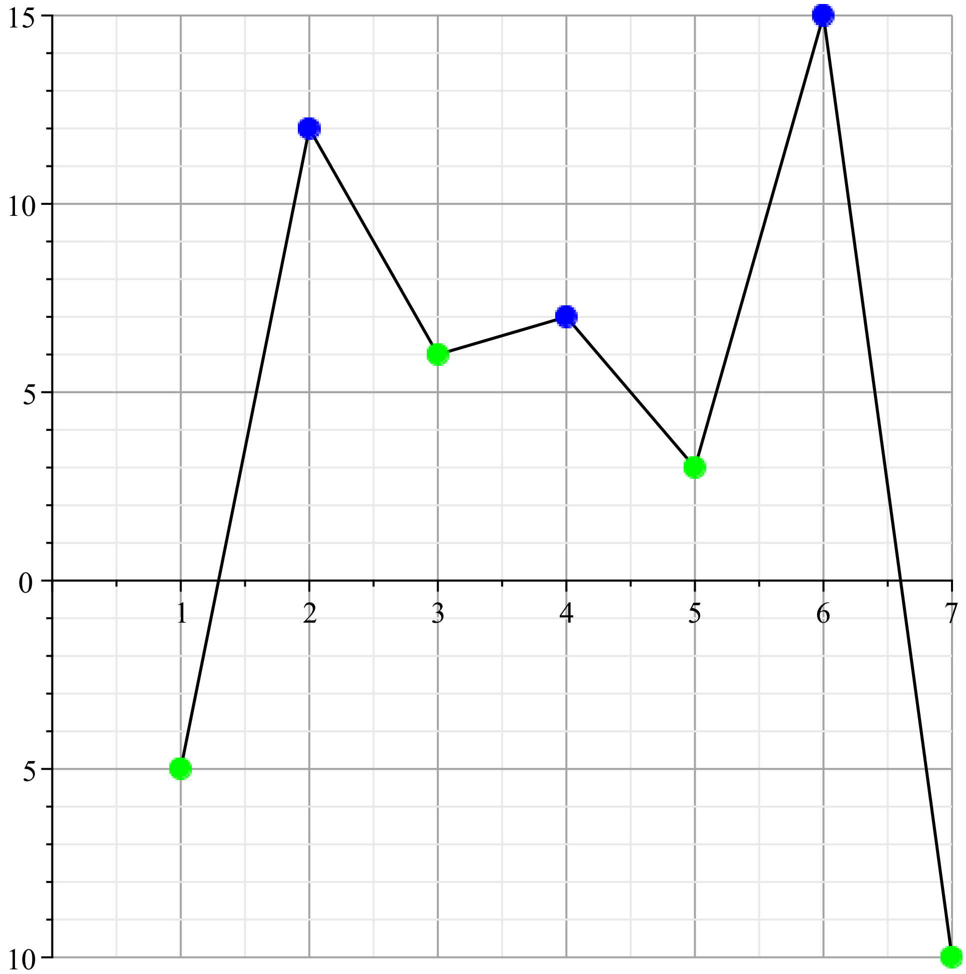
The FindPeakPoints command now has output options for the indices of the peaks and valleys.

For example:

```
> X := < -5, 12, 6, 7, 3, 15, -10 >;
```

$$X := \begin{bmatrix} -5 \\ 12 \\ 6 \\ 7 \\ 3 \\ 15 \\ -10 \end{bmatrix}$$

```
> FindPeakPoints( X, 'output' = 'plot', 'plotincludepoints' =
  ['peaks','regular','valleys'], 'gridlines' );
```



```
> FindPeakPoints( X, 'output' = ['peakindices','valleyindices'] );
```

$$\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 5 \\ 7 \end{bmatrix}$$

RootMeanSquareError and RelativeRootMeanSquareError

The RootMeanSquareError and RelativeRootMeanSquareError commands have been sped up for large data containers. For example:

```
> n := 10^5;
  r := 1.0 + 1.0 * I;
  dt := 'complex[8]';
  X := LinearAlgebra:-RandomVector( n, 'generator' = -r .. r,
  'datatype' = dt ):
  Y := LinearAlgebra:-RandomVector( n, 'generator' = -r .. r,
  'datatype' = dt ):
```

$$n := 100000$$

$$r := 1.0 + \mathrm{I}$$

$$dt := complex_8$$

```
> tests := 25;
  CodeTools:-Usage( RootMeanSquareError( X, Y ), 'iterations' =
  tests );
  CodeTools:-Usage( RelativeRootMeanSquareError( X, Y ),
  'iterations' = tests );
```

$$tests := 25$$

```
memory used=16.36KiB, alloc change=0 bytes, cpu time=3.12ms, real
time=3.56ms, gc time=0ns
```

$$1.15349306481481761$$

```
memory used=44.40KiB, alloc change=0 bytes, cpu time=7.48ms, real
time=7.20ms, gc time=0ns
```

$$1.41397255876189520$$