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

\[ \text{restart;} \]

\[ \text{with( SignalProcessing );} \]

\[ \text{GenerateSignal} \]

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

\[ \text{X := GenerateSignal( 3 * \sin(t) + \cos(2*t), t = 0 .. 4 * \Pi, 100 );} \]

\[
X := \begin{bmatrix}
1.34772606211760 \\
1.62729333861302 \\
1.83872140508605 \\
1.98581567591191 \\
2.07564416249220 \\
2.11781895027008 \\
2.12363272481008 \\
2.10510967724277 \\
2.07403525211827 \\
\vdots
\end{bmatrix}
\]

100 element Vector[column]
The command also packages many related features and data for the signal:

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

sample_rate, Times, Signal := 199.800000000000011, 
```

```maple
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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:

```plaintext
> 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'];
```

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:

\[ R['rmse']; > 0.0220493272388582497 \]

\[ \textbf{DifferentiateData} \]

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

\[ > f := \text{piecewise}( t < \pi \text{ or } t > 3 \times \pi, \sin(4\times t) + 5, 1/10 \times \sin(40\times t) + 5 ); \]
\[ a, b := 0, 4 \times \pi; \]
\[ n := 2^{14}; \]
\[ f \begin{cases} 
\sin(4t) + 5 & t < \pi \text{ or } 3\pi < t \\
\frac{\sin(40t)}{10} + 5 & \text{otherwise}
\end{cases} 
\]

\[ a, b := 0, 4\pi \]
\[ n := 16384 \]

\[ > ( \text{dt, T, X } ) := \text{GenerateSignal}( f, t = a .. b, n, \text{'includefinishetime'} = \text{'false'}, \text{'output'} = \text{['timestep','times','signal'] } ) ; \]

\[ > \text{DX} := \text{DifferentiateData}( X, 1, \text{'step' = dt, 'method' = \text{'spectral'}, 'extrapolation' = \text{'periodic'} } ) ; \]

\[ > \text{dataplot}( T, [X,DX], \text{'style' = 'line', 'color' = ['red','blue'], 'legend' = ['Signal','First derivative']}, \text{'size' = [500,250]} ) ; \]
The `IntegrateData2D` command provides a fast and accurate way of finding the volume under a two-dimensional dataset. For example:

\[
f := (x, y) \rightarrow \sqrt{1 + \sin(\pi \cdot (x^2 + 3 \cdot y^2))};
\]

\[
a, b, c, d := 0.0, 1.0, -0.5, 1.5;
\]

\[
f := (x, y) \sqrt{1 + \sin(\pi \cdot (x^2 + 3 \cdot y^2))};
a, b, c, d := 0., 1.0, -0.5, 1.5
\]

\[
> \text{plot3d}(f, a .. b, c .. d);
\]
> 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]' );

Z :=

<table>
<thead>
<tr>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
<th>1.30656296487638</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
<td>1.35122406134324</td>
</tr>
<tr>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
<td>1.38295754218013</td>
</tr>
<tr>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
<td>1.40302880476168</td>
</tr>
<tr>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
<td>1.41276686457614</td>
</tr>
<tr>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
<td>1.41352474569771</td>
</tr>
<tr>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
<td>1.40664653803368</td>
</tr>
<tr>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
<td>1.39344080846178</td>
</tr>
<tr>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
<td>1.37515989082796</td>
</tr>
<tr>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
<td>1.35298447186640</td>
</tr>
</tbody>
</table>

> 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:

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

X :=
[ -0.170954922213783 - 0.07032011674972541 I
  0.99893240195409 - 0.4243013103697261 I
  0.0799641980758583 + 0.4138348386455261 I
 -0.763689603106579 + 0.9768358575699611 I
 -0.338342009593391 + 0.7969722756685991 I ]

> Y := RealPart(X);

Y :=
[ -0.170954922213783
  0.99893240195409
  0.0799641980758583
 -0.763689603106579
 -0.338342009593391 ]

> Z := ImaginaryPart(X);

Z :=
[ -0.0703201167497254
 -0.424301310369726
 0.413834838645526
 0.976835857569961
 0.796972275668599 ]
```

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

```maple
> ComplexToReal(X);

[ -0.170954922213783 - 0.07032011674972541 I
  0.99893240195409 - 0.4243013103697261 I
  0.0799641980758583 + 0.4138348386455261 I
 -0.763689603106579 + 0.9768358575699611 I
 -0.338342009593391 + 0.7969722756685991 I ]
```

▼ **Insert**

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

```maple
> 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:

```plaintext
> 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'] );

\[
\begin{bmatrix}
2 \\
4 \\
6
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 \\
3 \\
5 \\
7
\end{bmatrix}
\]
The RootMeanSquareError and RelativeRootMeanSquareError commands have been sped up for large data containers. For example:

```maple
> 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 ):

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

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
```