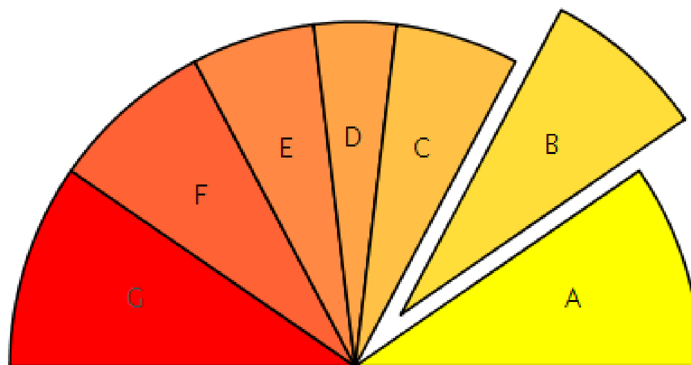
What's New in Maple 2016

Maple™

Component State

A new construct in the **DocumentTools:-Components** package called State allows you to manage computation state that will survive through a saved worksheet and through restart. Programming your applications using this construct also makes it easier to embed multiple different versions of the same application, with different data and options. Consider the following two interactive pie chart examples created using the package below.

$$\mathit{PieChartPlot}(["A" = 5, "B" = 4, "C" = 3, "D" = 2, "E" = 3, "F" = 4, "G" = 5]);$$



Options

Annular:

3-D:

Color: default

Explode: none

Labels: default

Outline:

Sector: 0..360

$$\mathit{PieChartPlot}([1, 1, 3, 4, 4, 5, 5, 5, 7]);$$

Options

Annular:

3-D:

Color:

default

Explode:

none

Labels:

default

Outline:

Sector:

0..360

The *PieChartPlot* command module is defined below. The *SetDefaults* command creates a *state variable*, which is used to hold all mutable information relating to this application, including the plot data and options settings. Additionally, the **`InsertContent`** command now accepts a **`state`** option that can be used with the state component variable name. When reexecuting—even with different data—the application can call **`GetProperty("", 'contentstate')`** in order to access the state of the application that is being replaced. In this way it can inherit whatever properties it wants from the previous application and preserve them when displaying the new data.

```
PieChartPlot := module()
    uses DT=DocumentTools, DL=DocumentTools:-Layout, DC=DocumentTools:-Components;

    # define the color options that will appear in the color combo box
    local colorschemes := table({
        "default"=NULL,
        "green/blue" = ('color'="Niagara_LeafGreen".."Niagara_DeepBlue"),
        "blue"=('color'="CornflowerBlue".."DarkBlue"),
        "red"=('color'="DarkRed".."Feldspar"),
        "green"=('color'="SeaGreen".."DarkSeaGreen"),
        "brown"=('color'="Niagara_Cinnamon".."Tan"),
        "purple"=('color'="DarkOrchid".."Plum"),
        "yellow"=('color'="YellowGreen".."Yellow"),
        "blue/rose"=('color'="Spring_Blue".."Spring_Rose"),
        "green/violet"=('color'="Spring_YellowGreen".."Spring_Violet"),
        "dark gray"=('color'="DarkGray".."Nautical_DarkGray"),
        "light gray"=('color'="Nautical_GrayViolet".."LightGray"),
        "yellow/red"=('color'="Yellow".."Red"),
        "bright"=('color'="Yellow"),
        "tan"=('color'="SaddleBrown".."Goldenrod"),
        "light blue"=('color'="SteelBlue".."Cyan")
    });

    # create and initialize the state variable
    local SetDefaults := proc( data )
        local stateVar, stateXML, prev;
        # check if there was prior state in the previous output
        prev := DT:-GetProperty("",'contentstate');
        if prev::`module`('signature') and prev:-signature = "PieChart" then
            prev:-xmap := NULL;
            prev:-Data := data;
            return prev;
        end if;

        stateXML := DC:-State('stateVar','Data'=data,'stateComponent','xmap','Options','signature'="PieChart");
        stateVar:-stateComponent := stateXML;

        stateVar:-Options := table({"annular"="false", "render3d"="false", "explode"=NULL, "outline"="true",
            "sector"="0..360", "datasetlabels"="default", "color"="default" });
        stateVar:-xmap := NULL;
        SetPlot(stateVar);
        return stateVar;
    end proc;

    # Use the state options to generate the pie chart.
    # Note that this procedure does not fetch the option values from the components -- those values are all
    # recorded in the state variable as the option control's action is triggered.
    local SetPlot := proc( sv )
        local p := Statistics:-PieChart(
            sv:-Data
            ,'explode'=[`if`(sv:-Data::list(equation),sv:-Options["explode"],parse(sv:-Options["explode"]))]
            ,'annular'=evalb(sv:-Options["annular"]="true")
            ,'render3d'=evalb(sv:-Options["render3d"]="true")
            ,'style'=`if`(sv:-Options["outline"]="true",'polygonoutline','polygon')
            ,'sector'=parse(sv:-Options["sector"])
            ,'datasetlabels'=parse(sv:-Options["datasetlabels"])
            ,colorschemes[sv:-Options["color"]]
        );
        if sv:-xmap <> NULL then
            DT:-SetProperty(sv:-xmap["piePlot"],'value',p);
        end if;
        return p;
    end proc;

    # Define the user-interface -- a table with a plot component on one side and options on the other.
    local GenApp := proc( sv )
        local xml;

        xml := DL:-Worksheet(
          DL:-Table( DL:-Column('weight'=70), DL:-Column('weight'=30),
            'interior'='none', 'exterior'='none', 'hiddenborderdisplay'='never', 'width'=800, 'widthmode'='pixels',
            DL:-Row(
              DL:-Cell(
                DL:-Textfield('alignment'='right',
                  DC:-Plot('identity'="piePlot",'showborders'=false,SetPlot(sv)))
                ),
              DL:-Cell(
                DL:-Section(
                  DL:-Title(DL:-Textfield("Options",'alignment'='left','style'='Heading3','layout'='Heading3')),
```

Small Example with Explanation

This example defines an application that uses a slider to move along the x-axis of a plot. When you drag the slider to the right, the plot view will move to the right along the x-axis. The slider will then snap back to the middle position, allowing you to grab it and move it to the right again. In this way you can move further and further along the x-axis.

The application can be described as follows:

- Highlighted in blue are the main components—Plot, Slider, and hidden State
- Highlighted in orange is the variable created as part of the State component description. Make sure this occurs earlier in the code than anything that uses it.

```
> moveLeftRightApp := proc( fn )
      local stateVariable, xmap;
      uses DocumentTools, DocumentTools:-Layout, DocumentTools:-
  Components;

      #define the application layout and components
      xmap := InsertContent( Worksheet( Table( Column(), Row( Cell(
  Textfield(
         State('stateVariable', position = 0, func = fn),
         Plot(identity="Plot0"),
         "\n",
         Slider(-10..10, 'identity'="Slider0", 'position'=0,
  'width'=400,
                'action'=sprintf("slideX(%s,\"Plot0\",\"Slider0\")
  ",stateVariable)))
      ) ) ) ),
      'output'=table );

      #set the initial plot (slideX is defined below)
      slideX(stateVariable,xmap["Plot0"],xmap["Slider0"]);
  end proc:
```

The action handler—the code that gets run when someone moves the slider—is defined below. It gets passed the state variable, from which it can see and update the current

position offset. The position records how far left or right we have scrolled along the x-axis. We get the slider value, adjust our position, redraw the moved plot, and snap the slider back to the middle.

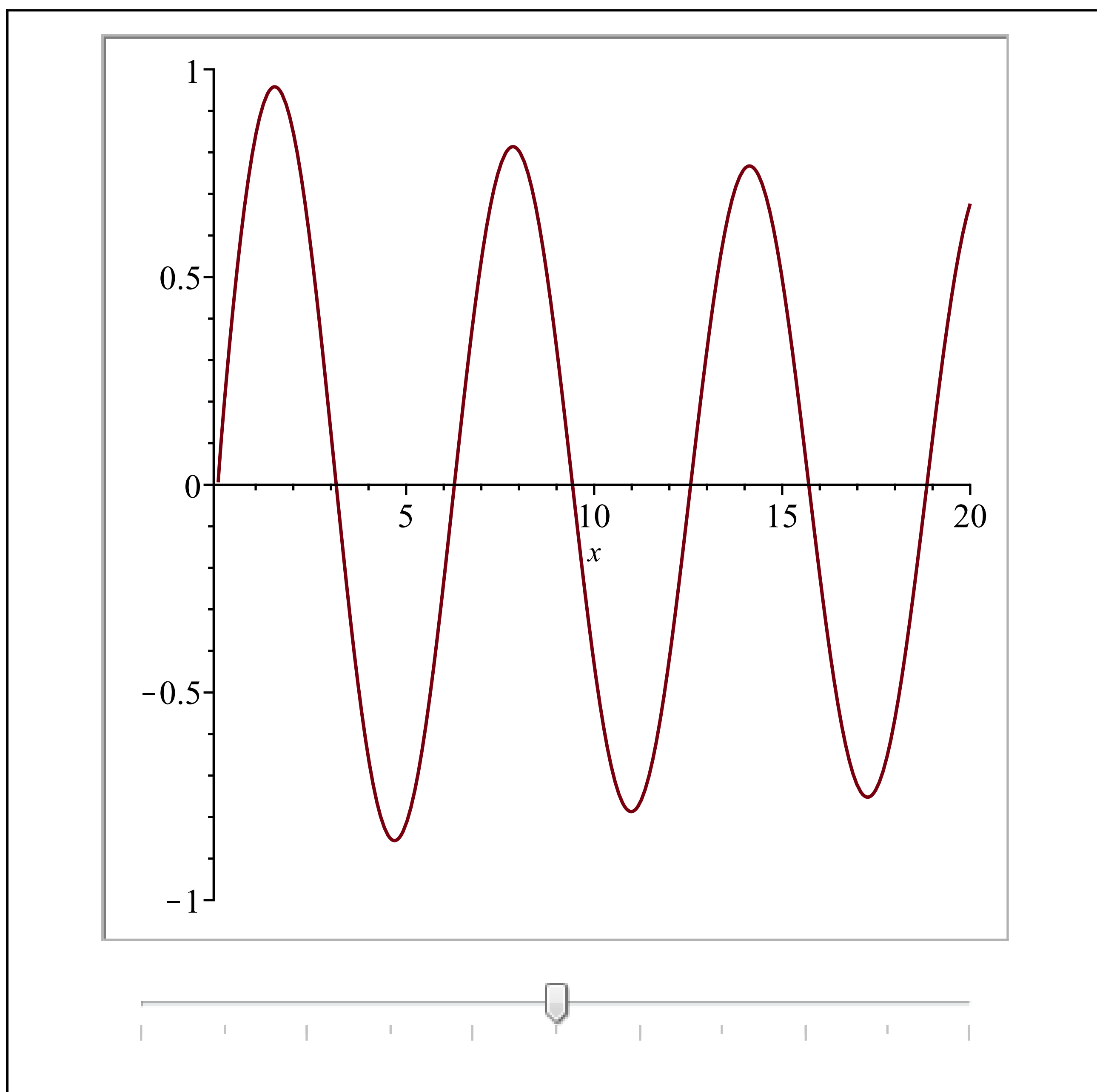
```
> slideX := proc( stateVar, plotCompName, sliderCompName )
      local delta, p;
      delta := DocumentTools:-GetProperty(sliderCompName,'value');
      stateVar:-position := stateVar:-position + delta;
      p := plot(stateVar:-func,x=stateVar:-position-10..stateVar:-
  position+10, 'view'=['default',-1..1]);

      DocumentTools:-SetProperty(plotCompName,'value', p);
      DocumentTools:-SetProperty(sliderCompName,'value',0);
  end proc:
```

Note that the above two procedures could be put in startup code (via **Edit>Startup Code**), or they can be auto-initialized by setting **Format>AutoExecute>Set**. If either is done, when this worksheet is loaded both of these procedure definitions will be executed.

Now that the app and handler are defined, we can run a command to insert the content:

```
> moveLeftRightApp( Re(x^(-.1))*sin(x) );
```



Peeking into Existing Content To "Remember" Settings

In the following modification to *moveLeftRightApp*, we call **`DocumentTools:-GetProperty("",'contentstate');`** This will return non-NULL if the content that we are inserting is going to replace existing content. We then additionally check if we are about to replace the same kind of content—by putting a unique signature on the state variable. If the signatures match, then we can initialize the new app based on some of the settings of the old app. For example, here we start at the same position that was previously left.

Two things need to happen:

- Check for previous content and access that content's state.
- Call **InsertContent** with the **state** option, giving it the name of the state variable.

```
> moveLeftRightApp2 := proc( fn )
     local stateVariable, prev, pos, xmap;
     uses DocumentTools, DocumentTools:-Layout, DocumentTools:-
  Components;

     # check if there was prior state in the previous output
                    prev := GetProperty("",    'contentstate');
                    if prev::`module`('signature') and prev:-
  signature = "moveLeftRight" then
         pos := prev:-position;
     else
         pos := 0;
     end if;

     #define the application layout and components
     xmap := InsertContent( Worksheet( Table( Column(), Row( Cell(
  Textfield(
        State('stateVariable', 'position' = pos, 'func' = fn,
  'signature'="moveLeftRight"),
        Plot(identity="Plot0"),
        "\n",
        Slider(-10..10, 'identity'="Slider0", 'position'=0,
  'width'=400,
               'action'=sprintf("slideX(%s,\"Plot0\",\"Slider0\")
  ",stateVariable)))
     ) ) ) ),
     'state'=convert(stateVariable,string),
     'output'=table );

     #set the initial plot (slideX is defined below)
     slideX(stateVariable,xmap["Plot0"],xmap["Slider0"]);
  end proc:
```

When you execute the following example with different values for the negative exponent, the x-axis offset remains the same between executions.

```
> moveLeftRightApp2( Re(x^(-.5))*sin(x) );
```