1 Introduction

As of the 97 release, PSTricks contains the \texttt{pst-grad} package, which provides a gradient fill style for arbitrary shapes. Although it often produces nice results, it has a number of deficiencies:

1. It is not possible to go from a colour $A$ to $B$ to $C$, etc. The most evident application of such a multi-colour gradient are of course rainbow effects. But they can also be useful in informative contexts, eg to identify modes of operation in a scale of values (normal/danger/overload).

2. Colours are interpolated linearly in the RGB space. This is often OK, but when you want to go from red $(1,0,0)$ to green $(0,1,0)$, it looks much better to get there via yellow $(1,1,0)$ than via brown $(0.5,0.5,0)$. The point is, that to get from one saturated colour to another, the colours on the way should also be saturated to produce an optically pleasing result.

3. \texttt{pst-grad} is limited to linear gradients, ie there is a (possibly rotated) rectilinear coordinate system, such that the colour at every point depends only on the $x$ coordinate of the point. In particular, there is no way to get circular patterns.

\texttt{pst-slpe} solves all of the mentioned problems in one package.

Problems 1. is addressed by permitting the user to specify an arbitrary number of colours, along with the points at which these are to be reached. A special form of each of the fill styles is provided, which just needs two colours as parameters, and goes from one to the other. This makes the fill styles easier to use in that simple case.

Problem 2. is solved by interpolating in the hue-saturation-value colour space. Conversion between RGB and HSV is done behind the scenes. The user specifies colours in RGB.
Finally, \texttt{pst-slpe} provides \textit{concentric} and \textit{radial} gradients. What these mean is best explained with a polar coordinate system: In a concentric pattern, the colour of a point depends on the radius coordinate, while in a radial pattern, it depends on the angle coordinate.

As a special bonus, the PostScript part of \texttt{pst-slpe} is somewhat optimized for speed. In \texttt{ghostscript}, rendering is about 30\% faster than with \texttt{pst-grad}.

For most of these problems, solutions have been posted in the appropriate \TeX{} newsgroup over the years. \texttt{pst-slpe} has however been developed independently from these proposals. It is based on the original PSTricks 0.93 \texttt{gradient} code, most of which has been changed or replaced. The author is indebted to Denis Girou, whose encouragement triggered the process of making this a shipable package instead of a private experiment.

The new fill styles and the graphics parameters provided to use them are described in section 2 of this document. Section 3, if present, documents the implementation consisting of a generic \TeX{} file and a PostScript header for the \texttt{dvi}-to-PostScript converter. You can get section 3 by calling \LaTeX{} as follows on most relevant systems:

\begin{verbatim}
l \texttt{latex \{AtBeginDocument\{\AlsoImplementation\}\input{pst-slpe.dtx}\}}
\end{verbatim}

2 Package Usage

To use \texttt{pst-slpe}, you have to say

\begin{verbatim}
\usepackage{pst-slpe}
\end{verbatim}

in the document prologue for \LaTeX{}, and

\begin{verbatim}
\input pst-slpe.tex
\end{verbatim}

in “plain” \TeX{}.

3 New macro and fill styles

\texttt{\psBall} It takes the (optional) coordinates of the ball center, the color and the radius as parameter and uses \texttt{\pscircle} for painting the bullet.

\begin{verbatim}
\psBall{black}{2ex}
\end{verbatim}
The predefined options can be overwritten in the usual way:

\psBall(1,0){blue}{3ex}
\psBall(2.5,0){red}{4ex}
\psBall(4,0){green!50!blue!60}{5ex}

\psBall{black}{2ex}
\psBall[sloperadius=10pt](1,0){blue}{3ex}
\psBall(2.5,0){red}{4ex}
\psBall[slopebegin=red](4,0){green!50!blue!60}{5ex}

\texttt{pst-slpe} provides six new fill styles called \texttt{slope}, \texttt{slopes}, \texttt{ccslope}, \texttt{ccslopes}, 
\texttt{slope} \texttt{slopes}, \texttt{ccslope} \texttt{ccslopes} and \texttt{radslope} \texttt{radslopes}. These obviously come in pairs: The ...\texttt{slope}-styles are simplified versions of the general ...\texttt{slopes}-styles.\footnote{The way, I use \texttt{slope} as a synonym for gradient. It sounds less pretentious and avoids name clashes.} The \texttt{cc}... styles paint concentric patterns, and the \texttt{rad}... styles do radial ones.

Here is a little overview of what they look like:

\begin{center}
\begin{tabular}{ccc}
\texttt{slope} & \texttt{slopes} \\
\texttt{ccslope} & \texttt{ccslopes} \\
\texttt{radslope} & \texttt{radslopes}
\end{tabular}
\end{center}

These examples were produced by saying simply

\texttt{\psframebox[fillstyle=slope]{...}}

etc. without setting any further graphics parameters. The package provides a number of parameters that can be used to control the way these patterns are painted.

\begin{itemize}
\item \texttt{slopebegin} and \texttt{slopeend} set the colours between...
\end{itemize}
which the three \texttt{...slopes} styles should interpolate. Eg,

\psframebox[fillstyle=slope,slopebegin=red,slopeend=green]{...}

produces:

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{example_slope}
  \caption{Example of slope style}
\end{figure}

The same settings of \texttt{slopebegin} and \texttt{slopeend} for the \texttt{ccslope} and \texttt{radslope} fillstyles produce

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{example_slope_ccslope}
  \caption{Example of ccslope style}
\end{figure}

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{example_slope_radslope}
  \caption{Example of radslope style}
\end{figure}

The default settings go from a greenish yellow to pure blue.

\begin{itemize}
  \item \texttt{slopecolors} - If you want to interpolate between more than two colours, you have to use the \texttt{...slopes} styles, which are controlled by the \texttt{slopecolors} parameter instead of \texttt{slopebegin} and \texttt{slopeend}. The idea is to specify the colour to use at certain points ‘on the way’. To fill a shape with \texttt{slopes}, imagine a linear scale from its left edge to its right edge. The left edge must lie at coordinate 0. Pick an arbitrary value for the right edge, say 23. Now you want to get light yellow at the left edge, a pastel green at 17/23 of the way and dark cyan at the right edge, like this:

  \begin{figure}[h]
    \centering
    \includegraphics[width=0.5\textwidth]{example_slope_slopecolors}
    \caption{Example of slope style with slopecolors}
  \end{figure}

  The RGB values for the three colours are (1, 1, 0.9), (0.5, 1, 0.5) and (0, 0.5, 0.5). The value for the \texttt{slopecolors} parameter is a list of ‘colour infos’ followed by the number of ‘colour infos’. Each ‘colour info’ consists of the coordinate value where a colour is to be specified, followed by the RGB values of that colour. All these values are separated by white space. The correct setting for the example is thus:

  \texttt{slopecolors=0 1 1 9 17 .5 1 5 23 0 .5 .5 3}

  For \texttt{ccslopes}, specify the colours from the center outward. For \texttt{radslopes} (with no rotation specified), 0 represents the ray going ‘eastward’. Specify the colours anti-clockwise. If you want a smooth gradient at the beginning and starting ray of \texttt{radslopes}, you should pick the first and last colours identical.

  Please note, that the \texttt{slopecolors} parameter is not subject to any parsing on the \TeX{} side. If you forget a number or specify the wrong number of segments, the PostScript interpreter will probably crash.
\end{itemize}
The default value for \texttt{slopecolors} specifies a rainbow.

\textbf{slopesteps} The parameter \texttt{slopesteps} controls the number of distinct colour steps rendered. Higher values for this parameter result in better quality but proportionally slower rendering. Eg, setting \texttt{slopesteps} to 5 with the \texttt{slope} fill style results in

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{slopes.png}
\caption{Example of \texttt{slope} fill style with \texttt{slopesteps}=5.}
\end{figure}

The default value is 100, which suffices for most purposes. Remember that the number of distinct colours reproducible by a given device is limited. Pushing \texttt{slopesteps} to high will result only in loss of performance at no gain in quality.

\textbf{slopeangle} The \texttt{slope(s)} and \texttt{radslope(s)} patterns may be rotated. As usual, the angles are given anti-clockwise. Eg, an angle of 30 degrees gives

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{slopes.png}
\caption{Example of \texttt{slope} fill style with \texttt{slopeangle}=30.}
\end{figure}

with the \texttt{slope} and \texttt{radslope} fillstyles.

\textbf{slopecenter} For the \texttt{cc...} and \texttt{rad...} styles, it is possible to set the center of the pattern. The \texttt{slopecenter} parameter is set to the coordinates of that center relative to the bounding box of the current path. The following effect:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{slopes.png}
\caption{Example of \texttt{cc...} fill style with \texttt{slopecenter}=0.2 0.4.}
\end{figure}

was achieved with

\texttt{fillstyle=radslope,slopecenter=0.2 0.4}

The default value for \texttt{slopecenter} is 0.5 0.5, which is the center for symmetrical shapes. Note that this parameter is not parsed by \TeX, so setting it to anything else than two numbers between 0 and 1 might crash the PostScript interpreter.

\textbf{sloperadius} Normally, the \texttt{cc...} and \texttt{rad...} styles distribute the given colours so that the center is painted in the first colour given, and the points of the shape furthest from the center are painted in the last colour. In other words the maximum radius to which the \texttt{slopecolors} parameter refers is the maximum distance from the center (defined by \texttt{slopecenter}) to any point on the periphery of the shape. This radius can be explicitly set with \texttt{sloperadius}. Eg, setting \texttt{sloperadius}=0.5cm gives
Any point further from the center than the given \texttt{sloperadius} is painted with the last colour in \texttt{slopeclours}, resp. \texttt{slopeend}.

The default value for \texttt{sloperadius} is 0, which invokes the default behaviour of automatically calculating the radius.

The optional boolean keyword \texttt{fading} allows a transparency effect of the filled area, starting with the opacity value \texttt{startfading} and ending with the value of \texttt{endfading}. Both values must be of the interval $[0...1]$, with 0 for total opacity and 1 for no opacity. The values are preset by 0 and 1.

Here is a little overview of what they look like:

These examples were produced by saying simply

\texttt{\psframebox[fading,fillstyle=...]{...}}

These examples were produced by saying simply

\texttt{\psframebox[fading,startfading=0.3,endfading=0.8,fillstyle=...]{...}}
4 The Code

4.1 Producing the documentation

A short driver is provided that can be extracted if necessary by the docstrip program provided with \LaTeXε.

\begin{verbatim}
\NeedsTeXFormat{LaTeX2e}
\documentclass{ltxdoc}
\usepackage{pst-slpe}
\usepackage{pst-plot}
\DisableCrossrefs
\MakeShortVerb{\|}
\newcommand\Lopt[1]{\textsf{#1}}
\newcommand\file[1]{\texttt{#1}}
\AtEndDocument{
\PrintChanges
\PrintIndex
}
\begin{document}
\DocInput{pst-slpe.dtx}
\end{document}
\end{verbatim}

4.2 The \texttt{pst-slpe.sty} file

The \texttt{pst-slpe.sty} file is very simple. It just loads the generic \texttt{pst-slpe.tex} file.

\begin{verbatim}
\RequirePackage{pstricks}
\ProvidesPackage{pst-slpe}[2005/03/05 package wrapper for 'pst-slpe.tex']
\input{pst-slpe.tex}
\ProvidesFile{pst-slpe.tex}
[\pstslpefiledate\space v\pstslpefileversion\space
\'pst-slpe' (mg,hv)]
\IfFileExists{pst-slpe.pro}{%
\ProvidesFile{pst-slpe.pro}{%
[2008/06/19 v. 0.01, PostScript prologue file (hv)]
\addtofilelist{pst-slpe.pro}}{}}%
\end{verbatim}

4.3 The \texttt{pst-slpe.tex} file

\texttt{pst-slpe.tex} contains the \TeX-side of things. We begin by identifying ourselves and setting things up, the same as in other PSTricks packages.

\begin{verbatim}
\message{ v\pstslpefileversion, \pstslpefiledate}
\csname PstSlopeLoaded\endslname
\end{verbatim}
4.3.1 New graphics parameters

We now define the various new parameters needed by the \texttt{slope} fill styles and install default values. First come the colours, i.e., graphics parameters \texttt{slopebegin} and \texttt{slopeend}, followed by the number of steps, \texttt{slopesteps}, and the rotation angle, \texttt{slopeangle}.

\begin{verbatim}
\newrgbcolor{slopebegin}{0.9 1 0}
\define@key[pst-slpe]{slopebegin}{\pst@getcolor{#1}\psslopebegin}% --> hv
\psset[pst-slpe]{slopebegin=slopebegin} % --> hv
\newrgbcolor{slopeend}{0 0 1}
\define@key[pst-slpe]{slopeend}{\pst@getcolor{#1}\psslopeend}% --> hv
\psset[pst-slpe]{slopeend=slopeend}% --> hv
\define@key[pst-slpe]{slopesteps}{\pst@getint{#1}\psslopesteps}% --> hv
\psset[pst-slpe]{slopesteps=100}% --> hv
\define@key[pst-slpe]{slopeangle}{\pst@getangle{#1}\psx@slopeangle}% --> hv
\psset[pst-slpe]{slopeangle=0}% --> hv
\end{verbatim}

\texttt{slopecolors} The value for \texttt{slopecolors} is not parsed. It is directly copied to the PostScript output. This is certainly not the way it should be, but it’s simple. The default value is a rainbow from red to magenta.

\begin{verbatim}
\define@key[pst-slpe]{slopecolors}{\def\psx@slopecolors{#1}}% --> hv
\psset[pst-slpe]{slopecolors={% --> hv
0.0 1 0 0
0.4 0 1 0
0.8 0 0 1
1.0 1 0 1
4}}
\end{verbatim}

\texttt{slopecenter} The argument to \texttt{slopecenter} isn’t parsed either. But there’s probably not much that can go wrong with two decimal numbers.

\begin{verbatim}
\define@key[pst-slpe]{slopecenter}{\def\psx@slopecenter{#1}}% --> hv
\psset[pst-slpe]{slopecenter=0.5 0.5}% --> hv
\end{verbatim}

\texttt{sloperadius} The default value for \texttt{sloperadius} is 0, which makes the PostScript procedure \texttt{PatchRadius} determine a value for the radius.
fading The default value for fading is false, which is no transparency effect at all. With fading=true the package takes the values startfading and endfading into account for the opacity effect of the filled area.

startfading The relative number for the starting value (0...1), preset by 0.

endfading The relative number for the end value (0...1), preset by 1.

4.3.2 Fill style macros

Now come the fill style definitions that use these parameters. There is one macro for each fill style named \psfs@style. PSTricks calls this macro whenever the current path needs to be filled in that style. The current path should not be clobbered by the PostScript code output by the macro.

slopes For the slopes fill style we produce PostScript code that first puts the slopecolors parameter onto the stack. Note that the number of colours listed, which comes last in slopecolors is now on the top of the stack. Next come the slopesteps and slopeangle parameters. We switch to the dictionary established by the pst-slop.pro Prolog and call SlopesFill, which does the artwork and takes care to leave the path alone.

slope The slope style uses parameters slopebegin and slopeend instead of slopecolors. So the produced PostScript uses these parameters to build a stack in slopecolors format. The \ps@usecolor generates PostScript to set the current colour. We can query the RGB values with currentrgbcolor. A gsave/grestore pair is used to avoid changing the PostScript graphics state. Once the stack is set up, SlopesFill is called as before.

72 \def\psfs@slopes{
73 \addto@pscode{
74 \psx@slopecolors\space
75 \psx@slopesteps
76 \psx@slopeangle
77 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
78 tx\PstSlopeDict begin SlopesFill end})

79 \def\psfs@slope{
80 \addto@pscode{
81 gsave
The code for the other fill styles is about the same, except for a few parameters more or less and different PostScript procedures called to do the work.

```plaintext
\def\psfs@ccslopes{% 
  \addto@pscode{% 
    \psx@slopecolors\space \psslopesteps \psx@slopecenter\space \psx@sloperadius\space 
    \ifPST@fading \psk@startfading \psk@endfading true \else false \fi 
    tx@PstSlopeDict begin CcSlopesFill end}}
\def\psfs@ccslope{% 
  \addto@pscode{% 
    gsave 0 \pst@usecolor\psslopebegin currentrgbcolor 
    1 \pst@usecolor\psslopeend currentrgbcolor 
    \ifPST@fading \psk@startfading \psk@endfading true \else false \fi 
    tx@PstSlopeDict begin CcSlopesFill end}}
\def\psfs@radslopes{% 
  \addto@pscode{% 
    \psx@slopecolors\space \psslopesteps \psx@slopecenter\space \psx@sloperadius\space \psx@slopeangle 
    \ifPST@fading \psk@startfading \psk@endfading true \else false \fi 
    tx@PstSlopeDict begin RadSlopesFill end}}
\def\psfs@radslope{% 
  \addto@pscode{% 
    gsave 0 \pst@usecolor\psslopebegin currentrgbcolor 
    1 \pst@usecolor\psslopeend currentrgbcolor 
    2 \pst@usecolor\psslopebegin currentrgbcolor 
    3 \pst@usecolor\psslopeend currentrgbcolor 
    4 \pst@usecolor\psslopebegin currentrgbcolor 
    5 grestore 
    \psslopesteps \psx@slopecenter\space \psx@sloperadius\space \psx@slopeangle 
    \ifPST@fading \psk@startfading \psk@endfading true \else false \fi 
    tx@PstSlopeDict begin RadSlopesFill end}}
```

\psBall

```plaintext
\def\psBall\@i{% 
  \@ifnextchar\psBall@ii\psBall@ii(0,0)}
\def\psBall@ii{% 
  \psBall}
```

```plaintext
\psBall\@i(%)
```

```plaintext
\psBall\@i(0,0))
```
\def\psBall@ii(#1,#2)#3#4{\%  
\pst@killglue  
\pssetlength\pst@dima{#4}%%%%% 20111025 hv  
\pst@dimb=\pst@dima%%%%%%%%%%% 20111025 hv  
\advance\pst@dima by 0.075\pst@dimb%  
\addbefore@par{sloperadius=\the\pst@dima,fillstyle=ccslope,  
slopebegin=white,slopeend=#3,slopecenter=0.4 0.6,linestyle=none}\%  
\use@par%  
\pscircle(#1,#2){#4}\%  
}\ignorespaces\%  
\catcode'@=\TheAtCode\relax  
⟨/texfile⟩

4.4 The \texttt{pst-slpe.pro} file

The file \texttt{pst-slpe.pro} contains PostScript definitions to be included in the PostScript output by the \texttt{dvi}-to-PostScript converter, eg \texttt{dvips}. First thing is to define a dictionary to keep definitions local.

\begin{verbatim}
\langle∗prolog⟩
/tx@PstSlopeDict 60 dict def tx@PstSlopeDict begin

Opacity++ This macro increments the Opacity index

/Opacity 1 def % preset, no transparency
/Opacity++ { Opacity dOpacity add /Opacity ED } def

max \texttt{x1 \texttt{x2 max max}
max is a utility function that calculates the maximum of two numbers.

/\texttt{max}{2 copy lt \{exch\} if pop} bind def

Iterate \texttt{p1 r1 g1 b1 \ldots pn rn gn bn n Iterate}
This is the actual iteration, which goes through the colour information and plots the segments. It uses the value of \texttt{NumSteps} which is set by the wrapper procedures. \texttt{DrawStep} is called all of \texttt{NumSteps} times, so it had better be fast.

First, the number of colour infos is read from the top of the stack and decremented, to get the number of segments.

/\texttt{Iterate}\{
1 sub /\texttt{NumSegs} ED

Now we get the first colour. This is really the \textit{last} colour given in the \texttt{slopecolors} argument. We have to work \textit{down} the stack, so we shall be careful to plot the segments in reverse order. The \texttt{dup mul} stuff squares the RGB components. This does a kind-of-gamma correction, without which primary colours tend to take up too much space in the slope. This is nothing deep, it just looks better in my opinion. The following lines convert RGB to HSB and store the resulting components, as well as the \texttt{Pt} coordinate in four variables.

dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
setrgbcolor currentshsbcolor
/ThisB ED
/ThisS ED
/ThisH ED
/ThisPt ED

To avoid gaps, we fill the whole path in that first colour.

Opacity .setopacityalpha
gsave
fill
grestore

The body of the following outer loop is executed once for each segment. It expects a current colour and Pt coordinate in the This* variables and pops the next colour and point from the stack. It then draws the single steps of that segment.

NumSegs {
  dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
  setrgbcolor currentshsbcolor
  /NextB ED
  /NextS ED
  /NextH ED
  /NextPt ED
}
NumSteps always contains the remaining number of steps available. These are evenly distributed between Pt coordinates ThisPt to 0, so for the current segment we may use NumSteps * (ThisPt - NextPt)/ThisPt steps.

ThisPt NextPt sub ThisPt div NumSteps mul cvi /SegSteps exch def
/NumSteps NumSteps SegSteps sub def

SegSteps may be zero. In that case there is nothing to do for this segment.

SegSteps 0 eq not {
  ThisS 0 eq {/ThisH NextH def} if
  NextS 0 eq {/NextH ThisH def} if
}

If one of the colours is gray, i.e. 0 saturation, its hue is useless. In this case, instead of starting off with a random hue, we take the hue of the other endpoint. (If both have saturation 0, we have a pure gray scale and no harm is done)

ThisS 0 eq {ThisH NextH def} if
NextS 0 eq {NextH ThisH def} if

To interpolate between two colours of different hue, we want to go the shorter way around the colour circle. The following code assures that this happens if we go linearly from This* to Next* by conditionally adding 1.0 to one of the hue values. The new hue values can lie between 0.0 and 2.0, so we will later have to subtract 1.0 from values greater than one.

ThisH NextH sub 0.5 gt
  {NextH ThisH sub 0.5 ge {/ThisH ThisH 1.0 add def} if }
ifelse

We define three variables to hold the current colour coordinates and calculate the corresponding increments per step.

/B ThisB def
The body of the following inner loop sets the current colour, according to \( H \), \( S \) and \( B \) and undoes the kind-of-gamma correction by converting to RGB colour. It then calls \texttt{DrawStep}, which draws one step and maybe updates the current point or user space, or variables of its own. Finally, it increments the three colour variables.

\begin{verbatim}
SegSteps {
  H dup 1. gt {1. sub} if
  S B sethsbcolor
  currentrgbcolor
  sqrt 3 1 roll sqrt 3 1 roll sqrt 3 1 roll
  setrgbcolor
  DrawStep
  /H H HInc add def
  /S S SInc add def
  /B B BInc add def
} bind repeat
\end{verbatim}

The outer loop ends by moving on to the \texttt{Next} colour and point.

\begin{verbatim}
/ThisH NextH def
/ThisS NextS def
/ThisB NextB def
/ThisPt NextPt def
} if
} bind repeat
} def
\end{verbatim}

\texttt{PatchRadius} — PatchRadius —

This macro inspects the value of the variable \texttt{Radius}. If it is 0, it is set to the maximum distance of any point in the current path from the origin of user space. This has the effect that the current path will be totally filled. To find the maximum distance, we flatten the path and call \texttt{UpdRR} for each endpoint of the generated polygon. The current maximum square distance is gathered in \texttt{RR}.

\begin{verbatim}
/PatchRadius {
  Radius 0 eq {
    /UpdRR { dup mul exch dup mul add RR max /RR ED } bind def
    gsave
    flattenpath
    /RR 0 def
    {UpdRR} {UpdRR} {} {} pathforall
    grestore
    /Radius RR sqrt def
} if
} def
\end{verbatim}

\texttt{SlopesFill} \( p_1 \ r_1 \ g_1 \ b_1 \ldots \ p_n \ r_n \ g_n \ b_n \ n \ s \ a \) — SlopesFill —

Fill the current path with a slope described by \( p_1, \ldots, b_n, n \). Use a total of \( s \) single
steps. Rotate the slope by $\alpha$ degrees, 0 meaning $r_1, g_1, b_1$ left to $r_n, g_n, b_n$ right.

After saving the current path, we do the rotation and get the number of steps, which is later needed by \texttt{Iterate}. Remember, that iterate calls \texttt{DrawStep} in the reverse order, ie from right to left. We work around this by adding 180 degrees to the rotation. Filling works by clipping to the path and painting an appropriate sequence of rectangles. \texttt{DrawStep} is set up for \texttt{Iterate} to draw a rectangle of width $\texttt{XInc}$ high enough to cover the whole clippath (we use the Level 2 operator \texttt{rectfill} for speed) and translate the user system by $\texttt{XInc}$.

\begin{verbatim}
202 /SlopesFill { 
203    /Fading ED % do we have fading? 
204    Fading { 
205      /FadingEnd ED % the last opacity value 
206      dup /FadingStart ED % the first opacity value 
207      /Opacity ED % the opacity start value 
208    } if 
209    gsave 
210    180 add rotate 
211    /NumSteps ED 
212    Fading { /dOpacity FadingEnd FadingStart sub NumSteps div def } if 
213    clip 
214    pathbbox 
215    /h ED /w ED 
216    2 copy translate 
217    h sub neg /h ED 
218    w sub neg /w ED 
219    /XInc w NumSteps div def 
220    /DrawStep { 
221      Fading { % do we have a fading? 
222        Opacity .setopacityalpha % set opacity value 
223        Opacity++ % increase opacity 
224      } if 
225      0 0 XInc h rectfill 
226      XInc 0 translate 
227    } bind def 
228    Iterate 
229    grestore 
230 } def
\end{verbatim}

\texttt{CcSlopesFill} $p_1 \ r_1 \ g_1 \ b_1 \ ... \ p_n \ r_n \ g_n \ b_n \ n \ c_x \ c_y \ r$ \texttt{CcSlopesFill} --
Fills the current path with a concentric pattern, ie in a polar coordinate system, the colour depends on the radius and not on the angle. Centered around a point with coordinates $(c_x, c_y)$ relative to the bounding box of the path, ie for a rectangle, $(0, 0)$ will center the pattern around the lower left corner of the rectangle, $(0.5, 0.5)$ around its center. The largest circle has a radius of $r$. If $r = 0$, $r$ is taken to be the maximum distance of any point on the current path from the center defined by $(c_x, c_y)$. The colours are given from the center outwards, ie $(r_1, g_1, b_1)$ describe the colour at the center.

The code is similar to that of \texttt{SlopesFill}. The main differences are the call
to PatchRadius, which catches the case that \( r = 0 \) and the different definition for DrawStep, Which now fills a circle of radius \( \text{Rad} \) and decreases that Variable. Of course, drawing starts on the outside, so we work down the stack and circles drawn later partially cover those drawn first. Painting non-overlapping, ‘donut-shapes’ would be slower.

\[
\text{RadSlopesFill} \quad p_1, r_1, b_1, \ldots, p_n, r_n, b_n \quad n \quad c_x, c_y, r, \alpha \quad \text{CcSlopesFill} \quad -
\]

This fills the current path with a radial pattern, ie in a polar coordinate system the colour depends on the angle and not on the radius. All this is very similar to CcSlopesFill. There is an extra parameter \( \alpha \), which rotates the pattern.

The only new thing in the code is the DrawStep procedure. This does not draw a circular arc, but a triangle, which is considerably faster. One of the short sides of the triangle is determined by Radius, the other one by dY, which is calculated
as \( dY := \text{Radius} \times \tan(\text{AngleIncrement}) \).

266 /RadSlopesFill {
267 /Fading ED % do we have fading?
268 Fading {
269 /FadingEnd ED % the last opacity value
270 dup /FadingStart ED % the first opacity value
271 /Opacity ED % the opacity start value
272 } if
273 gsave
274 rotate
275 /Radius ED
276 /CenterY ED
277 /CenterX ED
278 /NumSteps ED
279 Fading { /dOpacity FadingEnd FadingStart sub NumSteps div def } if
280 clip
281 pathbbox
282 /h ED /w ED
283 2 copy translate
284 h sub neg /h ED
285 w sub neg /w ED
286 w CenterX mul h CenterY mul translate
287 PatchRadius
288 /AngleIncrement 360 NumSteps div neg def
289 /dY AngleIncrement sin AngleIncrement cos div Radius mul def
290 /DrawStep {
291 Fading { % do we have a fading?
292 Opacity .setopacityalpha % set opacity value
293 Opacity++ % increase opacity
294 } if
295 0 0 moveto
296 Radius 0 rlineto
297 0 dY rlineto
298 closepath fill
299 AngleIncrement rotate
300 } bind def
301 Iterate
302 grestore
303 } def

Last, but not least, we have to close the private dictionary.

304 end
305 ⟨/prolog⟩