

pst-slpe package

version 1.3

Martin Giese*and Herbert Voß†

2008/09/20

1 Introduction

As of the 97 release, PSTricks contains the `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. `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.

`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.

*giese@ira.uka.de

†voss@pstricks.de

Finally, `pst-slpe` provides *concentric* and *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 `pst-slpe` is somewhat optimized for speed. In `ghostscript`, rendering is about 30% faster than with `pst-grad`.

For most of these problems, solutions have been posted in the appropriate \TeX newsgroup over the years. `pst-slpe` has however been developed independently from these proposals. It is based on the original PSTricks 0.93 `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 shippable 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 `dvi-to-PostScript` converter. You can get section 3 by calling \LaTeX as follows on most relevant systems:

```
latex '\AtBeginDocument{\AlsoImplementation}\input{pst-slpe.dtx}'
```

2 Package Usage

To use `pst-slpe`, you have to say

```
\usepackage{pst-slpe}
```

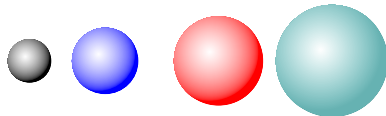
in the document prologue for \LaTeX , and

```
\input pst-slpe.tex
```

in “plain” \TeX .

3 New macro and fill styles

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



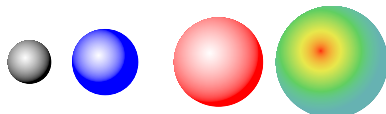
```
\psBall{black}{2ex}
```

```

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

```

The predefined options can be overwritten in the usual way:

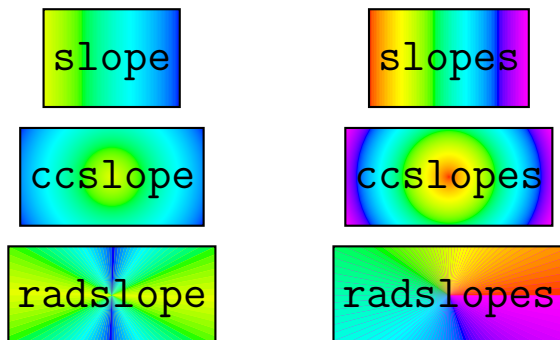


```

\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}

```

`slope` `\pst-slope` provides six new fill styles called `slope`, `slopes`, `ccslope`, `ccslopes`,
`slopes` `radslope` and `rad slopes`. These obviously come in pairs: The `...slope`-styles
`ccslope` are simplified versions of the general `...slopes`-styles.¹ The `cc...` styles paint
`ccslopes` concentric patterns, and the `rad...` styles do radial ones.
`radslope` Here is a little overview of what they look like:
`rad slopes`



These examples were produced by saying simply

```
\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.

`slopebegin` The graphics parameters `slopebegin` and `slopeend` set the colours between
`slopeend`

¹By the way, I use `slope` as a synonym for gradient. It sounds less pretentious and avoids name clashes.

which the three `...slope` styles should interpolate. Eg,

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

produces:



The same settings of `slopebegin` and `slopeend` for the `ccslope` and `radslope` fillstyles produce



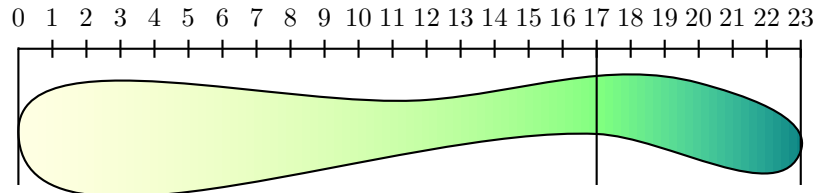
resp.



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

`slopecolors`

If you want to interpolate between more than two colours, you have to use the `...slopes` styles, which are controlled by the `slopecolors` parameter instead of `slopebegin` and `slopeend`. The idea is to specify the colour to use at certain points ‘on the way’. To fill a shape with `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:



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 `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:

```
slopecolors=0 1 1 .9 17 .5 1 .5 23 0 .5 .5 3
```

For `ccslopes`, specify the colours from the center outward. For `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 `radslopes`, you should pick the first and last colours identical.

Please note, that the `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.

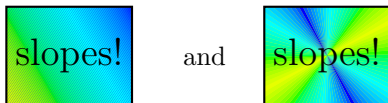
The default value for `slopecolors` specifies a rainbow.

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



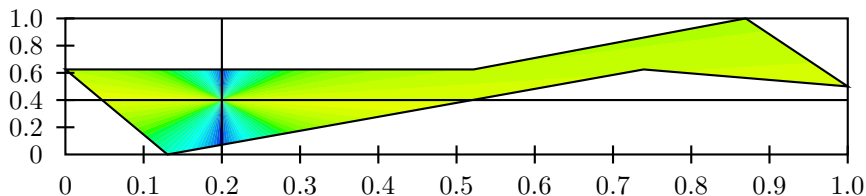
The default value is 100, which suffices for most purposes. Remember that the number of distinct colours reproducible by a given device is limited. Pushing `slopesteps` to high will result only in loss of performance at no gain in quality.

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



with the `slope` and `radslope` fillstyles.

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

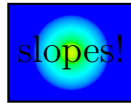


was achieved with

```
fillstyle=radslope,slopecenter=0.2 0.4
```

The default value for `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.

`sloperadius` Normally, the `cc...` and `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 `slopecolors` parameter refers is the maximum distance from the center (defined by `slopecenter`) to any point on the periphery of the shape. This radius can be explicitly set with `sloperadius`. Eg, setting `sloperadius=0.5cm` gives



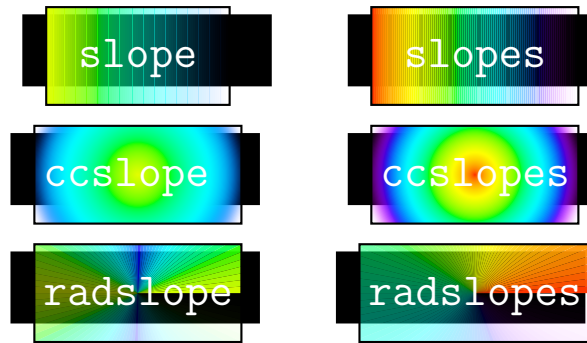
Any point further from the center than the given `sloperadius` is painted with the last colour in `slopeclours`, resp. `slopeend`.

The default value for `sloperadius` is 0, which invokes the default behaviour of automatically calculating the radius.

`fading`
`startfading`
`endfading`

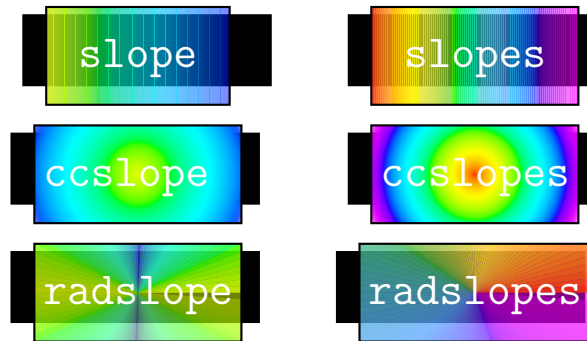
The optional boolean keyword `fading` allows a transparency effect of the filled area, starting with the opacity value `startfading` and ending with the value of `endfading`. Both values must be of the intervall $[0 \dots 1]$, with 0 for total opacity and 1 for no opacity. The values are preset by 0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

Here is a little overview of what they look like:



These examples were produced by saying simply

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



These examples were produced by saying simply

```
\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 L^AT_EX 2_ε.

```
1 <*driver>
2 \NeedsTeXFormat{LaTeX2e}
3 \documentclass{ltxdoc}
4 \usepackage{pst-slpe}
5 \usepackage{pst-plot}
6 \DisableCrossrefs
7 \MakeShortVerb{\|}
8 \newcommand\Lopt[1]{\textsf{#1}}
9 \newcommand\file[1]{\texttt{#1}}
10 \AtEndDocument{
11 \PrintChanges
12 \PrintIndex
13 }
14 %\OnlyDescription
15 \begin{document}
16 \DocInput{pst-slpe.dtx}
17 \end{document}
18 </driver>
```

4.2 The pst-slpe.sty file

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

```
19 <*stylefile>
20 \RequirePackage{pstricks}
21 \ProvidesPackage{pst-slpe}[2005/03/05 package wrapper for 'pst-slpe.tex']
22 \input{pst-slpe.tex}
23 \ProvidesFile{pst-slpe.tex}
24 [\pstslopefiledate\space v\pstslopefileversion\space
25 'pst-slpe' (mg,hv)]
26 \IfFileExists{pst-slpe.pro}{%
27 \ProvidesFile{pst-slpe.pro}
28 [2008/06/19 v. 0.01, PostScript prologue file (hv)]
29 \@addtofilelist{pst-slpe.pro}}{}%
30 </stylefile>
```

4.3 The pst-slpe.tex file

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

```
31 <*texfile>
32 \message{ v\pstslopefileversion, \pstslopefiledate}
33 \csname PstSlopeLoaded\endcsname
```

```

34 \let\PstSlopeLoaded\endinput
35 \ifx\PSTricksLoaded\endinput\else
36   \def\next{\input pstricks.tex }\expandafter\next
37 \fi
38 \ifx\PSTXKeyLoaded\endinput\else\input pst-xkey \fi % --> hv
39 \edef\TheAtCode{\the\catcode'\@}
40 \catcode'\@=11
41 \pst@addfams{pst-slpe} % --> hv
42 \pstheader{pst-slpe.pro}

```

slopebegin 4.3.1 New graphics parameters

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

```

43 \newrgbcolor{slopebegin}{0.9 1 0}
44 \define@key[psset]{pst-slpe}{slopebegin}{\pst@getcolor{#1}\psslopebegin}% --> hv
45 \psset[pst-slpe]{slopebegin=slopebegin} % --> hv
46
47 \newrgbcolor{slopeend}{0 0 1}
48 \define@key[psset]{pst-slpe}{slopeend}{\pst@getcolor{#1}\psslopeend}% --> hv
49 \psset[pst-slpe]{slopeend=slopeend}% --> hv
50
51 \define@key[psset]{pst-slpe}{slopesteps}{\pst@getint{#1}\psslopesteps}% --> hv
52 \psset[pst-slpe]{slopesteps=100}% --> hv
53
54 \define@key[psset]{pst-slpe}{slopeangle}{\pst@getangle{#1}\psx@slopeangle}% --> hv
55 \psset[pst-slpe]{slopeangle=0}% --> hv

```

slopecolors The value for `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.

```

56 \define@key[psset]{pst-slpe}{slopecolors}{\def\psx@slopecolors{#1}}% --> hv
57 \psset[pst-slpe]{slopecolors={% --> hv
58 0.0 1 0 0
59 0.4 0 1 0
60 0.8 0 0 1
61 1.0 1 0 1
62 4}}

```

slopecenter The argument to `slopecenter` isn't parsed either. But there's probably not much
that can go wrong with two decimal numbers.

```

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

```

sloperadius The default value for `sloperadius` is 0, which makes the PostScript procedure
`PatchRadius` determine a value for the radius.


```
65 \define@key[psset]{pst-slope}{sloperadius}{\pst@getlength{#1}\psx@sloperadius}% --> hv
66 \psset[pst-slope]{sloperadius=0}% --> hv
```

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.

```
67 \define@boolkey[psset]{pst-slope}[PST@]{fading}[true]{}% --> hv
68 \psset[pst-slope]{fading=false}% --> hv
```

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

```
69 \define@key[psset]{pst-slope}{startfading}{\pst@checknum{#1}\psk@startfading }% --> hv
```

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

```
70 \define@key[psset]{pst-slope}{endfading}{\pst@checknum{#1}\psk@endfading }% --> hv
71 \psset[pst-slope]{startfading=0,endfading=1}% --> hv
```

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.

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

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 `\pst@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.

```
79 \def\psfs@slope{%
80 \addto@pscode{%
81 gsave
```

```

82 0 \pst@usecolor\psslopebegin currentrgbcolor
83 1 \pst@usecolor\psslopeend currentrgbcolor
84 2
85 grestore
86 \psslopesteps \psx@slopeangle
87 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
88 tx@PstSlopeDict begin SlopesFill end}}

```

ccslopes The code for the other fill styles is about the same, except for a few parameters
ccslope more or less and different PostScript procedures called to do the work.

```

radslopes 89 \def\psfs@ccslopes{%
90 \addto@pscode{%
91 \psx@slopecolors\space
92 \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
93 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
94 tx@PstSlopeDict begin CcSlopesFill end}}
95 \def\psfs@ccslope{%
96 \addto@pscode{%
97 gsave 0 \pst@usecolor\psslopebegin currentrgbcolor
98 1 \pst@usecolor\psslopeend currentrgbcolor
99 2 grestore
100 \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
101 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
102 tx@PstSlopeDict begin CcSlopesFill end}}
103 \def\psfs@radslopes{%
104 \addto@pscode{%
105 \psx@slopecolors\space
106 \psslopesteps\psx@slopecenter\space\psx@sloperadius\space\psx@slopeangle
107 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
108 tx@PstSlopeDict begin RadSlopesFill end}}

```

radslope **radslope** is slightly different: Just going from one colour to another in 360 degrees is usually not what is wanted. **radslope** just does something pretty with the colours provided.

```

109 \def\psfs@radslope{%
110 \addto@pscode{%
111 gsave 0 \pst@usecolor\psslopebegin currentrgbcolor
112 1 \pst@usecolor\psslopeend currentrgbcolor
113 2 \pst@usecolor\psslopebegin currentrgbcolor
114 3 \pst@usecolor\psslopeend currentrgbcolor
115 4 \pst@usecolor\psslopebegin currentrgbcolor
116 5 grestore
117 \psslopesteps\psx@slopecenter\space\psx@sloperadius\space\psx@slopeangle
118 \ifPST@fading \psk@startfading \psk@endfading true \else false \fi
119 tx@PstSlopeDict begin RadSlopesFill end}}

```

\psBall

```

120 \def\psBall{\pst@object{psBall}}
121 \def\psBall@i{\ifnextchar(\psBall@ii{\psBall@ii(0,0)}}

```

```

122 \def\psBall@ii(#1,#2)#3#4{%
123   \pst@killglue
124   \pst@dima=#4%
125   \pst@dimb=#4%
126   \advance\pst@dima by 0.075\pst@dimb%
127   \begingroup%
128   \addbefore@par{sloperadius=\the\pst@dima,fillstyle=ccslope,
129     slopebegin=white,slopeend=#3,slopecenter=0.4 0.6,linestyle=none}%
130   \use@par%
131   \pscircle(#1,#2){#4}%
132   \endgroup\ignorespaces%
133 }

134 \catcode'\@=\TheAtCode\relax
135 </texfile>

```

4.4 The pst-slpe.pro file

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

```

136 <*prolog>
137 /tx@PstSlopeDict 60 dict def tx@PstSlopeDict begin

```

Opacity++ This macro increments the Opacity index

```

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

```

max x_1 x_2 **max** *max*

max is a utility function that calculates the maximum of two numbers.

```

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

```

Iterate p_1 r_1 g_1 b_1 ... p_n r_n g_n b_n n **Iterate** -

This is the actual iteration, which goes through the colour information and plots the segments. It uses the value of `NumSteps` which is set by the wrapper procedures. `DrawStep` is called all of `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.

```

141 /Iterate {
142   1 sub /NumSegs ED

```

Now we get the first colour. This is really the *last* colour given in the `slopecolors` argument. We have to work *down* the stack, so we shall be careful to plot the segments in reverse order. The `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 `Pt` coordinate in four variables.

```

143  dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
144  setrgbcolor currenthsbcolor
145  /ThisB ED
146  /ThisS ED
147  /ThisH ED
148  /ThisPt ED

```

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

```

149  Opacity .setopacityalpha
150  gsave
151  fill
152  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.

```

153  NumSegs {
154    dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
155    setrgbcolor currenthsbcolor
156    /NextB ED
157    /NextS ED
158    /NextH ED
159    /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 $\text{NumSteps} * (\text{ThisPt} - \text{NextPt}) / \text{ThisPt}$ steps.

```

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

```

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

```

162    SegSteps 0 eq not {

```

If one of the colours is gray, ie 0 saturation, its hue is useless. In this case, instead of starting of 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)

```

163      ThisS 0 eq {/ThisH NextH def} if
164      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.

```

165      ThisH NextH sub 0.5 gt
166      {/NextH NextH 1.0 add def}
167      { NextH ThisH sub 0.5 ge {/ThisH ThisH 1.0 add def} if }
168      ifelse

```

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

```

169     /B ThisB def
170     /S ThisS def
171     /H ThisH def
172     /BInc NextB ThisB sub SegSteps div def
173     /SInc NextS ThisS sub SegSteps div def
174     /HInc NextH ThisH sub SegSteps div 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 `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.

```

175     SegSteps {
176         H dup 1. gt {1. sub} if S B sethsbcolor
177         currentrgbcolor
178         sqrt 3 1 roll sqrt 3 1 roll sqrt 3 1 roll
179         setrgbcolor
180         DrawStep
181         /H H HInc add def
182         /S S SInc add def
183         /B B BInc add def
184     } bind repeat

```

The outer loop ends by moving on to the `Next` colour and point.

```

185     /ThisH NextH def
186     /ThisS NextS def
187     /ThisB NextB def
188     /ThisPt NextPt def
189     } if
190 } bind repeat
191 } def

```

`PatchRadius` — `PatchRadius` —

This macro inspects the value of the variable `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 `UpdRR` for each endpoint of the generated polygon. The current maximum square distance is gathered in `RR`.

```

192 /PatchRadius {
193     Radius 0 eq {
194         /UpdRR { dup mul exch dup mul add RR max /RR ED } bind def
195         gsave
196         flattenpath
197         /RR 0 def
198         {UpdRR} {UpdRR} {} {} pathforall
199         grestore
200         /Radius RR sqrt def
201     } if
202 } def

```

SlopesFill $p_1 r_1 g_1 b_1 \dots p_n r_n g_n b_n n s \alpha$ **SlopesFill** –
 Fill the current path with a slope described by p_1, \dots, b_n, n . Use a total of s single steps. Rotate the slope by α 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 **Iterate**. Remember, that **iterate** calls **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. **DrawStep** is set up for **Iterate** to draw a rectangle of width **XInc** high enough to cover the whole clippath (we use the Level 2 operator **rectfill** for speed) and translate the user system by **XInc**.

```

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

```

CcSlopesFill $p_1 r_1 g_1 b_1 \dots p_n r_n g_n b_n n c_x c_y r$ **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 `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 `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.

```

232 /CcSlopesFill {
233   /Fading ED % do we have fading?
234   Fading {
235     /FadingEnd ED % the last opacity value
236     dup /FadingStart ED % the first opacity value
237     /Opacity ED % the opacity start value
238   } if
239   gsave
240   /Radius ED
241   /CenterY ED
242   /CenterX ED
243   /NumSteps ED
244   Fading { /dOpacity FadingEnd FadingStart sub NumSteps div def } if
245   clip
246   pathbbox
247   /h ED /w ED
248   2 copy translate
249   h sub neg /h ED
250   w sub neg /w ED
251   w CenterX mul h CenterY mul translate
252   PatchRadius
253   /RadPerStep Radius NumSteps div neg def
254   /Rad Radius def
255   /DrawStep {
256     Fading { % do we have a fading?
257       Opacity .setopacityalpha % set opacity value
258       Opacity++ % increase opacity
259     } if
260     0 0 Rad 0 360 arc
261     closepath fill
262     /Rad Rad RadPerStep add def
263   } bind def
264   Iterate
265   grestore
266 } def

```

`RadSlopesFill` $p_1 r_1 g_1 b_1 \dots p_n r_n g_n b_n n c_x c_y r \alpha$ `CcSlopesFill` –
 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 α , 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 $dY := \text{Radius} \times \tan(\text{AngleIncrement})$.

```

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

```

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

```

305 end
306 </prolog>

```