

pst-slpe package

version 1.2

Martin Giese*

2008/06/19

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.

*email: giese@ira.uka.de Version 1.2 prepared by Herbert Voß 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 T_EX 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 T_EX file and a PostScript header for the dvi-to-PostScript converter. You can get section 3 by calling L^AT_EX 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}
```

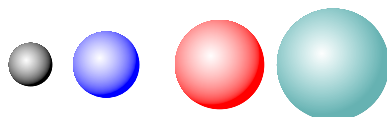
in the document prologue for L^AT_EX, and

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

in “plain” T_EX.

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.



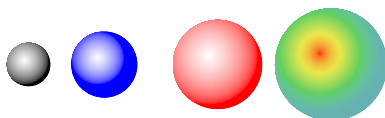
```
\psBall{black}{2ex}
```

```

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

```

The predinied options can be overwritten in the usual way:

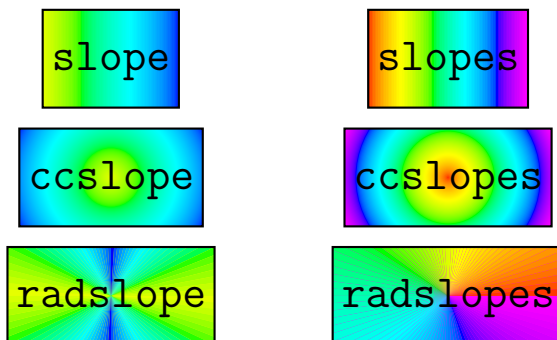


```

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



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]{...}
```

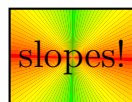
produces:



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



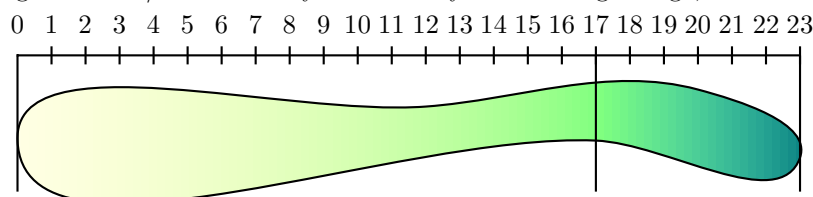
resp.



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.

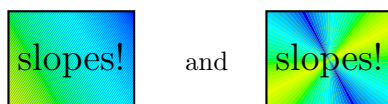
The default value for `slopecolors` specifies a rainbow.

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



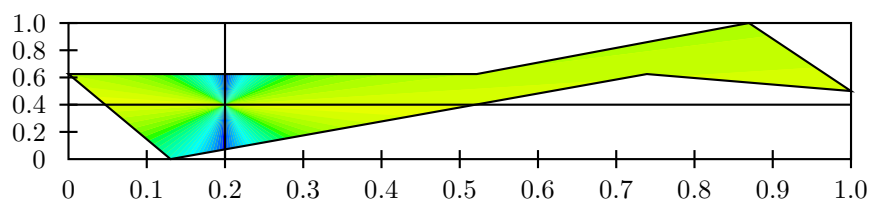
The default value is 100, which suffices for most purposes. Remember that the number of distinct colours reproducible by a given device is limited. Pushing `slopesteps` to high will result only in loss of performance at no gain in quality.

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:

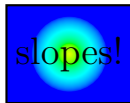


was achieved with

```
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.

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' (Martin Giese)]
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 \langle /stylefile \rangle

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  $\langle$ *texfile $\rangle$ 
32 \message{ v\pstslpefileversion, \pstslpefiledate}
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
slopesteps
slopeangle

We now define the various new parameters needed by the slope fill styles and install default values. First come the colours, ie graphics parameters slopebegin 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-slpe}{sloperadius}{\pst@getlength{#1}\psx@sloperadius}% --> hv
66 \psset[pst-slpe]{sloperadius=0}% --> 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.

```
67 \def\psfs@slopes{%
68 \addto@pscode{
69 \psx@slopecolors\space
70 \psslopesteps
71 \psx@slopeangle
72 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.

```
73 \def\psfs@slope{%
74 \addto@pscode{%
75 gsave
76 0 \pst@usecolor\psslopebegin currentrgbcolor
77 1 \pst@usecolor\psslopeend currentrgbcolor
78 2
79 grestore
80 \psslopesteps \psx@slopeangle 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

```

81 \def\psfs@ccslopes{%
82 \addto@pscode{%
83 \psx@slopecolors\space
84 \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
85 tx@PstSlopeDict begin CcSlopesFill end}}
86 \def\psfs@ccslope{%
87 \addto@pscode{%
88 gsave 0 \pst@usecolor\psslopebegin currentrgbcolor
89 1 \pst@usecolor\psslopeend currentrgbcolor
90 2 grestore
91 \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
92 tx@PstSlopeDict begin CcSlopesFill end}}
93 \def\psfs@radslopes{%
94 \addto@pscode{%
95 \psx@slopecolors\space
96 \psslopesteps\psx@slopecenter\space\psx@sloperadius\space\psx@slopeangle
97 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.

```

98 \def\psfs@radslope{%
99 \addto@pscode{%
100 gsave 0 \pst@usecolor\psslopebegin currentrgbcolor
101 1 \pst@usecolor\psslopeend currentrgbcolor
102 2 \pst@usecolor\psslopebegin currentrgbcolor
103 3 \pst@usecolor\psslopeend currentrgbcolor
104 4 \pst@usecolor\psslopebegin currentrgbcolor
105 5 grestore
106 \psslopesteps\psx@slopecenter\space\psx@sloperadius\space\psx@slopeangle
107 tx@PstSlopeDict begin RadSlopesFill end}}

```

\psBall

```

108 \def\psBall{\pst@object{psBall}}
109 \def\psBall@i{\@ifnextchar(\psBall@ii{\psBall@ii(0,0)}}
110 \def\psBall@ii(#1,#2)#3#4{%
111 \pst@killglue
112 \pst@dima=#4%
113 \pst@dimb=#4%
114 \advance\pst@dima by 0.075\pst@dimb%
115 \begingroup%
116 \addbefore@par{sloperadius=\the\pst@dima,fillstyle=ccslope,
117 slopebegin=white,slopeend=#3,slopecenter=0.4 0.6,linestyle=none}%
118 \use@par%
119 \pscircle(#1,#2){#4}%
120 \endgroup\ignorespaces%
121 }

```

```

122 \catcode'\@=\TheAtCode\relax
123 \</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.

```

124 <*prolog>
125 /tx@PstSlopeDict 60 dict def tx@PstSlopeDict begin

```

```

max x1 x2 max max
max is a utility function that calculates the maximum of two numbers.
126 /max {2 copy 1t {exch} if pop} bind def

```

```

Iterate p1 r1 g1 b1 ... 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 NumSteps which is set by the wrapper proce-
dures. 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.

```

127 /Iterate {
128 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.

```

129 dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
130 setrgbcolor currenthsbcolor
131 /ThisB ED
132 /ThisS ED
133 /ThisH ED
134 /ThisPt ED

```

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

```

135 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.

```

136 NumSegs {
137 dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll
138 setrgbcolor currenthsbcolor
139 /NextB ED
140 /NextS ED

```

```

141    /NextH ED
142    /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.

```

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

```

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

```

145    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)

```

146        ThisS 0 eq {/ThisH NextH def} if
147        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.

```

148        ThisH NextH sub 0.5 gt
149        {/NextH NextH 1.0 add def}
150        { NextH ThisH sub 0.5 ge {/ThisH ThisH 1.0 add def} if }
151        ifelse

```

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

```

152        /B ThisB def
153        /S ThisS def
154        /H ThisH def
155        /BInc NextB ThisB sub SegSteps div def
156        /SInc NextS ThisS sub SegSteps div def
157        /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.

```

158        SegSteps {
159            H dup 1. gt {1. sub} if S B sethsbcolor
160            currentrgbcolor
161            sqrt 3 1 roll sqrt 3 1 roll sqrt 3 1 roll
162            setrgbcolor
163            DrawStep
164            /H H HInc add def
165            /S S SInc add def
166            /B B BInc add def
167        } bind repeat

```

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

```

168      /ThisH NextH def
169      /ThisS NextS def
170      /ThisB NextB def
171      /ThisPt NextPt def
172    } if
173  } bind repeat
174 } 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**.

```

175 /PatchRadius {
176   Radius 0 eq {
177     /UpdRR { dup mul exch dup mul add RR max /RR ED } bind def
178     gsave
179     flattenpath
180     /RR 0 def
181     {UpdRR} {UpdRR} {} {} pathforall
182     grestore
183     /Radius RR sqrt def
184   } if
185 } 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**.

```

186 /SlopesFill {
187   gsave
188   180 add rotate
189   /NumSteps ED
190   clip
191   pathbbox
192   /h ED /w ED
193   2 copy translate
194   h sub neg /h ED
195   w sub neg /w ED
196   /XInc w NumSteps div def

```

```

197 /DrawStep {
198   0 0 XInc h rectfill
199   XInc 0 translate
200 } bind def
201 Iterate
202 grestore
203 } 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.

```

204 /CcSlopesFill {
205   gsave
206   /Radius ED
207   /CenterY ED
208   /CenterX ED
209   /NumSteps ED
210   clip
211   pathbbox
212   /h ED /w ED
213   2 copy translate
214   h sub neg /h ED
215   w sub neg /w ED
216   w CenterX mul h CenterY mul translate
217   PatchRadius
218   /RadPerStep Radius NumSteps div neg def
219   /Rad Radius def
220   /DrawStep {
221     0 0 Rad 0 360 arc
222     closepath fill
223     /Rad Rad RadPerStep add def
224   } bind def
225   Iterate
226   grestore
227 } 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 $dY := Radius \times \tan(\text{AngleIncrement})$.

```

228 /RadSlopesFill {
229   gsave
230   rotate
231   /Radius ED
232   /CenterY ED
233   /CenterX ED
234   /NumSteps ED
235   clip
236   pathbbox
237   /h ED /w ED
238   2 copy translate
239   h sub neg /h ED
240   w sub neg /w ED
241   w CenterX mul h CenterY mul translate
242   PatchRadius
243   /AngleIncrement 360 NumSteps div neg def
244   /dY AngleIncrement sin AngleIncrement cos div Radius mul def
245   /DrawStep {
246     0 0 moveto
247     Radius 0 rlineto
248     0 dY rlineto
249     closepath fill
250     AngleIncrement rotate
251   } bind def
252   Iterate
253   grestore
254 } def

      Last, but not least, we have to close the private dictionary.
255 end
256 </prolog>

```