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 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 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 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}
```

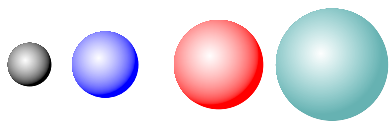
in the document prologue for LATEX, and

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

in "plain" TEX.

3 New macro and fill styles

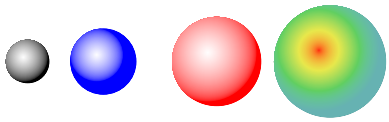
\psBall It takes the (optional) coordinates of the ball center, the color and the radius as parameter and uses \pscircle for painting the bullet.



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

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

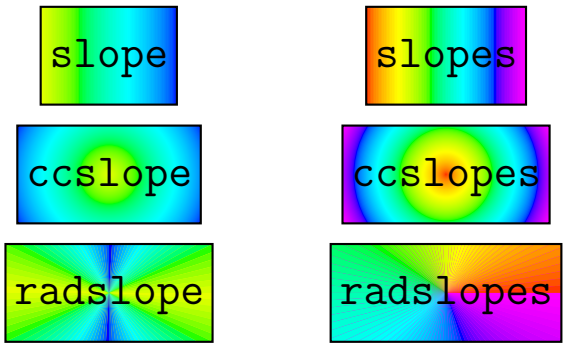
The predinied options can be overwritten in the usual way:



```
\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
slopes
ccslope
ccslopes
radslope
radslopes

`pst-slpe` provides six new fill styles called `slope`, `slopes`, `ccslope`, `ccslopes`, `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
slopeend

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

[1] By 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:

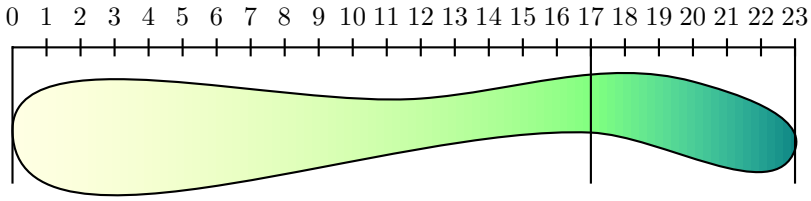
The same settings of `slopebegin` and `slopeend` for the `ccslope` and `radslope` fillstyles produce

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.

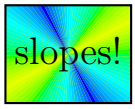
The default value for `slopecolors` specifies a rainbow.

`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.

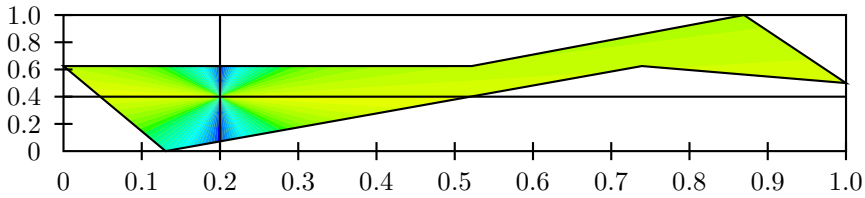
`slopeangle` The `slope(s)` and `radslope(s)` patterns may be rotated. As usual, the angles are given anti-clockwise. Eg, an angle of 30 degrees gives

and



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:

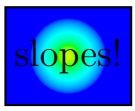


was achieved with

```
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 LaTeX 2_ε.

```
⟨*driver⟩
\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
}
%\OnlyDescription
\begin{document}
\DocInput{pst-slpe.dtx}
\end{document}
⟨/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.

```
⟨*stylefile⟩
\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’ (Martin Giese)]
⟨/stylefile⟩
```

4.3 The pst-slpe.tex file

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.

```
⟨*texfile⟩
\message{ v\pstslpefileversion, \pstslpefiledate}
\csname PstSlopeLoaded\endcsname
\let\PstSlopeLoaded\endinput
\ifx\PSTricksLoaded\endinput\else
  \def\next{\input pstricks.tex }\expandafter\next
\fi
\ifx\PSTXKeyLoaded\endinput\else\input pst-xkey \fi % --> hv
\edef\TheAtCode{\the\catcode`\@}
\catcode`\@=11
\pst@addfams{pst-slpe}                                % --> hv
\pstheader{pst-slpe.pro}
```

slopebegin
slopeend
slopesteps
slopeangle

4.3.1 New graphics parameters

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.

```
\newrgbcolor{slopebegin}{0.9 1 0}
\define@key[psset]{pst-slpe}{slopebegin}{\pst@getcolor{#1}\psslopebegin}% --> hv
\psset[pst-slpe]{slopebegin=slopebegin}                                   % --> hv

\newrgbcolor{slopeend}{0 0 1}
\define@key[psset]{pst-slpe}{slopeend}{\pst@getcolor{#1}\psslopeend}% --> hv
\psset[pst-slpe]{slopeend=slopeend}% --> hv

\define@key[psset]{pst-slpe}{slopesteps}{\pst@getint{#1}\psslopesteps}% --> hv
\psset[pst-slpe]{slopesteps=100}% --> hv

\define@key[psset]{pst-slpe}{slopeangle}{\pst@getangle{#1}\psx@slopeangle}% --> hv
\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.

```
\define@key[psset]{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}}
```

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

```
\define@key[psset]{pst-slpe}{slopecenter}{\def\psx@slopecenter{#1}}% --> hv
\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.

```
\define@key[psset]{pst-slpe}{sloperadius}{\pst@getlength{#1}\psx@sloperadius}% --> hv
\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.

```
\def\psfs@slopes{%
 \addto@pscode{
  \psx@slopecolors\space
  \psslopesteps
  \psx@slopeangle
  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.

```
\def\psfs@slope{%
 \addto@pscode{%
  gsave
    0 \pst@usecolor\psslopebegin currentrgbcolor
    1 \pst@usecolor\psslopeend currentrgbcolor
    2
  grestore
  \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

```
\def\psfs@ccslopes{%
 \addto@pscode{%
  \psx@slopecolors\space
  \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
  tx@PstSlopeDict begin CcSlopesFill end}}
\def\psfs@ccslope{%
 \addto@pscode{%
  gsave 0 \pst@usecolor\psslopebegin currentrgbcolor
    1 \pst@usecolor\psslopeend currentrgbcolor
    2 grestore
  \psslopesteps \psx@slopecenter\space \psx@sloperadius\space
  tx@PstSlopeDict begin CcSlopesFill end}}
\def\psfs@radslopes{%
 \addto@pscode{%
  \psx@slopecolors\space
  \psslopesteps\psx@slopecenter\space\psx@sloperadius\space\psx@slopeangle
  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.

```
\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
  tx@PstSlopeDict begin RadSlopesFill end}}
```

\psBall

```
\def\psBall{\pst@object{psBall}}
\def\psBall@i{\@ifnextchar(\psBall@ii{\psBall@ii(0,0)}}
\def\psBall@ii(#1,#2)#3#4{%
  \pst@killglue
  \pst@dima=#4%
  \pst@dimb=#4%
  \advance\pst@dima by 0.075\pst@dimb%
  \begingroup%
  \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}%
  \endgroup\ignorespaces%
}

\catcode`\@=\TheAtCode\relax
⟨/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.

119 ⟨∗prolog⟩
120 `/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.

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

`Iterate` p_1 r_1 g_1 b_1 $\ldots$ 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.

122 `/Iterate {`
123 `  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.

124 `  dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll`
125 `  setrgbcolor currenthsbcolor`
126 `  /ThisB ED`
127 `  /ThisS ED`
128 `  /ThisH ED`
129 `  /ThisPt ED`

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

130 `  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.

131 `  NumSegs {`
132 `    dup mul 3 1 roll dup mul 3 1 roll dup mul 3 1 roll`
133 `    setrgbcolor currenthsbcolor`
134 `    /NextB ED`
135 `    /NextS ED`
136 `    /NextH ED`
137 `    /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.

```
138     ThisPt NextPt sub ThisPt div NumSteps mul cvi /SegSteps exch def
139     /NumSteps NumSteps SegSteps sub def
```

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

```
140     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)

```
141       ThisS 0 eq {/ThisH NextH def} if
142       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.

```
143       ThisH NextH sub 0.5 gt
144         {/NextH NextH 1.0 add def}
145         { NextH ThisH sub 0.5 ge {/ThisH ThisH 1.0 add def} if }
146       ifelse
```

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

```
147       /B ThisB def
148       /S ThisS def
149       /H ThisH def
150       /BInc NextB ThisB sub SegSteps div def
151       /SInc NextS ThisS sub SegSteps div def
152       /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.

```
153       SegSteps {
154         H dup 1. gt {1. sub} if S B sethsbcolor
155         currentrgbcolor
156         sqrt 3 1 roll sqrt 3 1 roll sqrt 3 1 roll
157         setrgbcolor
158         DrawStep
159         /H H HInc add def
160         /S S SInc add def
161         /B B BInc add def
162       } bind repeat
```

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

```
163       /ThisH NextH def
```

```
164       /ThisS NextS def
165       /ThisB NextB def
166       /ThisPt NextPt def
167     } if
168   } bind repeat
169 } 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.

```
170 /PatchRadius {
171   Radius 0 eq {
172     /UpdRR { dup mul exch dup mul add RR max /RR ED } bind def
173     gsave
174     flattenpath
175     /RR 0 def
176     {UpdRR} {UpdRR} {} {} pathforall
177     grestore
178     /Radius RR sqrt def
179   } if
180 } def
```

SlopesFill p_1 r_1 g_1 b_1 ... p_n r_n g_n b_n n s α 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 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.

```
181 /SlopesFill {
182   gsave
183   180 add rotate
184   /NumSteps ED
185   clip
186   pathbbox
187   /h ED /w ED
188   2 copy translate
189   h sub neg /h ED
190   w sub neg /w ED
191   /XInc w NumSteps div def
192   /DrawStep {
193     0 0 XInc h rectfill
```

```
194     XInc 0 translate
195   } bind def
196   Iterate
197   grestore
198 } def
```

`CcSlopesFill` p_1 r_1 g_1 b_1 ... 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.

```
199 /CcSlopesFill {
200   gsave
201   /Radius ED
202   /CenterY ED
203   /CenterX ED
204   /NumSteps ED
205   clip
206   pathbbox
207   /h ED /w ED
208   2 copy translate
209   h sub neg /h ED
210   w sub neg /w ED
211   w CenterX mul h CenterY mul translate
212   PatchRadius
213   /RadPerStep Radius NumSteps div neg def
214   /Rad Radius def
215   /DrawStep {
216     0 0 Rad 0 360 arc
217     closepath fill
218     /Rad Rad RadPerStep add def
219   } bind def
220   Iterate
221   grestore
222 } def
```

`RadSlopesFill` p_1 r_1 g_1 b_1 ... p_n r_n g_n b_n n c_x c_y r α `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` × tan(`AngleIncrement`).

```
/RadSlopesFill {
  gsave
  rotate
  /Radius ED
  /CenterY ED
  /CenterX ED
  /NumSteps ED
  clip
  pathbbox
  /h ED /w ED
  2 copy translate
  h sub neg /h ED
  w sub neg /w ED
  w CenterX mul h CenterY mul translate
  PatchRadius
  /AngleIncrement 360 NumSteps div neg def
  /dY AngleIncrement sin AngleIncrement cos div Radius mul def
  /DrawStep {
    0 0 moveto
    Radius 0 rlineto
    0 dY rlineto
    closepath fill
    AngleIncrement rotate
  } bind def
  Iterate
  grestore
} def
```

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

```
end
⟨/prolog⟩
```