

# Learning PostScript by Doing

André Heck

© 2005, AMSTEL Institute, UvA

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Getting Started</b>	<b>3</b>
2.1	A Simple Example Using GHOSTSCRIPT . . . . .	4
2.2	A Simple Example Using GSVIEW . . . . .	5
2.3	Using PostScript Code in a Microsoft Word Document . . . . .	6
2.4	Using PostScript Code in a L <sup>A</sup> T <sub>E</sub> X Document . . . . .	7
2.5	Numeric Quantities . . . . .	8
2.6	The Operand Stack . . . . .	9
2.7	Mathematical Operations and Functions . . . . .	11
2.8	Operand Stack Manipulation Operators . . . . .	12
2.9	If PostScript Interpreting Goes Wrong . . . . .	13
<b>3</b>	<b>Basic Graphical Primitives</b>	<b>14</b>
3.1	Point . . . . .	15
3.2	Curve . . . . .	18
3.2.1	Open and Closed Curves . . . . .	18
3.2.2	Filled Shapes . . . . .	18
3.2.3	Straight and Circular Line Segments . . . . .	21
3.2.4	Cubic Bézier Line Segment . . . . .	23
3.3	Angle and Direction Vector . . . . .	25
3.4	Arrow . . . . .	27
3.5	Circle, Ellipse, Square, and Rectangle . . . . .	29
3.6	Commonly Used Path Construction Operators and Painting Operators . . . . .	31
3.7	Text . . . . .	33
3.7.1	Simple PostScript Text . . . . .	33
3.7.2	Importing L <sup>A</sup> T <sub>E</sub> X text . . . . .	38
3.8	Symbol Encoding . . . . .	40
<b>4</b>	<b>Style Directives</b>	<b>42</b>
4.1	Dashing . . . . .	42
4.2	Coloring . . . . .	43
4.3	Joining Lines . . . . .	45

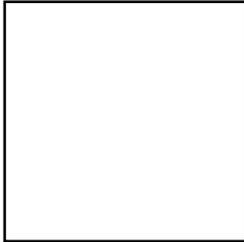
<b>5</b>	<b>Control Structures</b>	<b>48</b>
5.1	Conditional Operations . . . . .	48
5.2	Repetition . . . . .	50
<b>6</b>	<b>Coordinate Transformations</b>	<b>66</b>
<b>7</b>	<b>Procedures</b>	<b>75</b>
7.1	Defining Procedure . . . . .	75
7.2	Parameter Passing . . . . .	75
7.3	Local variables . . . . .	76
7.4	Recursion . . . . .	76
<b>8</b>	<b>More Examples</b>	<b>82</b>
8.1	Planar curves . . . . .	83
8.2	The Lorenz Butterfly . . . . .	88
8.3	A Surface Plot . . . . .	89
8.4	Iterated Functions . . . . .	91
8.5	Marking Angles and Lines . . . . .	95

# 1 Introduction

PostScript is a graphic programming language developed by Adobe Systems that allows its user to produce high-quality graphics and text that can be printed. By high quality we mean that an author has full control over the graphical output and can produce clean computer code in exactly the way he or she wants it, in contrast with the PostScript code produced by high-end software packages like XFIG, PICTEX, METAPOST, ADOBE ILLUSTRATOR, and computer algebra systems like MAPLE and *Mathematica*. A drawback of this ‘do-it-yourself’ approach is that an author has to put effort in learning to write PostScript code and that he or she must accept less simplicity in return of high quality graphical output. This course is only meant as a short, hands-on introduction to PostScript for newcomers who want to produce rather simple graphics, say e.g. teachers who want to produce neat graphics for their personally created tutorials and tests. The main objective is to get students started with PostScript. A more thorough, but also much longer introduction that also discusses the mathematics behinds graphics is the textbook *Mathematical Illustrations* of Bill Casselman [Cas05]. For complete descriptions of the PostScript language we refer to the so-called red book [Pos99], green book [Pos88], and blue book [Pos86]. These are the ultimate reference manuals.

We follow a few didactical guidelines in writing the course. Learning is best done from examples, learning is done from practice. The examples are often formatted in two columns, as follows:<sup>1</sup>

---

	<pre>newpath   1 1 moveto 91 1 lineto 91 91 lineto   1 91 lineto   1 1 lineto stroke</pre>
---	--

---

The exercises give you the opportunity to practice PostScript; the answers to the exercises in which a picture or program must be created are provided separately as Encapsulated PostScript files on a cd-rom and electronically via Internet.

## 2 Getting Started

PostScript is not a WYSIWYG drawing tool like `xfig` or `paint`. It is a graphic document preparation system. First, you write a plain text containing graphic formatting commands into a file by means of your favorite editor. Next, an interpreter allows you to preview the graphical object. We will use two PostScript interpreters, viz. AFPL GHOSTSCRIPT and GSVIEW. Both interpreters can be downloaded without cost from [www.cs.wisc.edu/~ghost](http://www.cs.wisc.edu/~ghost). In this chapter we will describe the basics of this process of creating and viewing PostScript, as well as conversion into different graphic formats such as BITMAP, GIF, and JPEG.

---

<sup>1</sup>On the left is printed the graphic result of the PostScript code on the right. Here, a square is drawn.

## 2.1 A Simple Example Using Ghostscript

GHOSTSCRIPT is an interactive PostScript interpreter. This means that you can type PostScript commands in a command line interface and see the effect of it in a separate image window. In this way you can have a quick start with learning the effect of PostScript commands. To invoke GHOSTSCRIPT on a Unix platform, enter the `gs` command. On a Windows PC, double-click the `gswin32.exe` application.

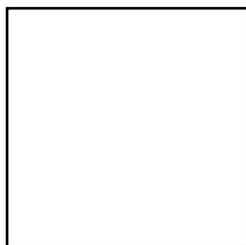
### EXERCISE 1

Do the following steps:

1. Install the AFPL GHOSTSCRIPT interpreter on your PC, if necessary, and start the program by double-clicking the `gs.exe` application. You will get a command window that looks like this:

```
AFPL Ghostscript 8.00 (2002-11-21)
Copyright (C) 2002 artofcode LLC, Benicia, CA. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>
```

2. Enter the PostScript statements on the right-hand side and verify in the Ghostscript Image window that the picture on the left-hand side appears:<sup>2</sup>



```
GS>newpath
GS> 1 1 moveto
GS>91 1 lineto
GS>91 91 lineto
GS> 1 91 lineto
GS> 1 1 lineto
GS>stroke
```

---

The example in the above exercise needs some explanation. The Ghostscript Image window opens with a default coordinate system, the origin of which is at the lower left corner of the page. The default unit of measurement, which is the same in both horizontal and vertical directions, is equal to a *point*, exactly equal to 1/72 of an inch. The size of the Ghostscript Image page is by default U.S. letter (8.5×11 inches, 216×279 millimeters, or 612×792 points), but it can be changed into European A4 paper size (8.3×11.7 inches, 210×297 millimeters, or 595×842 points) by editing the initialization file `GS_INIT.PS`. This file is usually in the `LIB` directory somewhere in the search path of Ghostscript. Find the consecutive lines

```
% Optionally choose a default paper size other than U.S. letter.
% (a4)
```

Then, to make A4 the default paper size, uncomment the second line by removal of the comment character `%`.

You draw things in PostScript by constructing *paths*. A path is a set of connected and disconnected points, lines and curves that together describes a geometric figure. A path is

---

<sup>2</sup>If necessary, scroll down to get the lower left corner of the page in view.

constructed out of simple graphics primitives such as `moveto` and `lineto`. The resulting path may then have paint applied to it with one of the painting operators such as `stroke` and `fill`. At any moment there is only one *current path* on the drawing surface called the *current page*.

The `newpath` operator on the first line empties the current path and declares we are starting a new path. We start building a path with a `moveto`. This operator treats the two numbers as  $x$  and  $y$  coordinates to which to move, in this case to the point (1, 1). The coordinates specified become the *current point*. The `lineto` operator on the third line, `91 1 lineto`, adds a line segment to the current path that connects the current point to the position specified by the numbers, in this case 91 and 1. The point specified as the argument to this operator becomes the new current point. Note that the `lineto` operator does not actually draw on the current page. It simply adds a line segment to the current path. The `stroke` operator in the last line in the program code causes the path we have constructed to be painted on the current page. Our path, in this case a square, becomes visible.

## 2.2 A Simple Example Using GSview

GSVIEW is an PostScript previewer built on top of GHOSTSCRIPT. In this tutorial we will use the GSview 4.4 PostScript previewer. It allows you to type PostScript commands in a file with your favorite editor, as long as it produces ordinary text without formatting, and see the effect of its contents in an preview window. Once you have become more familiar with PostScript it will be more convenient to use the PostScript previewer GSVIEW than the underlying PostScript interpreter GHOSTSCRIPT. To invoke GHOSTSCRIPT on a Unix platform, enter the `gsview` command. On a Windows PC, double-click the `gsview32.exe` application.

### EXERCISE 2

Do the following steps:

1. Install the PostScript previewer GSVIEW on your PC, if necessary, and start the program by double-clicking the `gsview32.exe` application.
2. Create a text file, say `exercise2.eps`, that contains the following PostScript statements:

---

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 5 5 105 105
%%BeginProlog
%%EndProlog
newpath
 10 10 moveto
 90 0 rlineto
 0 90 rlineto
-90 0 rlineto
closepath
5 setlinewidth
stroke
showpage
%%EOF
```

---

3. Open the PostScript file in GSVIEW and verify that a square is drawn with thick border lines.<sup>3</sup>
4. Select the EPS Clip item in the Option menu and verify that a square is drawn with a little bit of white around it (1 pt to be precise).

The example in the above exercise needs some explanation. At the very beginning of a PostScript file you must have the characters `%!PS` to identify it as a *conforming* PostScript language program. The longer line `%!PS-Adobe-3.0 EPSF-3.0` specifies the version of PostScript to which the program is supposed to conform, viz. *Encapsulated PostScript* Adobe level 3. The next few lines beginning with `%%` are *document structure comments*, which are mostly ignored by the PostScript interpreter, but may be parsed by other programs, e.g. while incorporating PostScript graphics. For example, `%%EOF` informs the previewer or printer that the end of the document has been reached. The `%%BoundingBox: 5 5 105 105` statement, which is required for an Encapsulated PostScript file, informs the previewer or printer about the boundaries of the drawing area, in this case a rectangular box with lower left corner at position (5,5) and upper right corner at position (105, 105). As you will see later on in this tutorial, the *prolog* may also contain definitions of variables and procedures that are used in the program code. The `%%BeginProlog` and `%%EndProlog` statements mark the beginning and end of the prolog, respectively. The `showpage` statement near the end of the file is only needed in case you want to print the PostScript picture. It can be used elsewhere in a PostScript program to transmit the contents of the current page to the output device.

The `rlineto` operator on the third line of the main program, `90 0 rlineto`, adds a line segment to the current path extending 90 units to the right from the current point. In other words, `rlineto` means a *relative lineto*. Similarly, the `rmoveto` operator means motion relative to the current point. The `setlinewidth` operator allows you to specify the width of the line that is stroked onto your path. By default, the linewidth is 1 unit; in this case, a linewidth of 5 units (here, 5 points) is specified. The `closepath` operator in the program code causes the current path to be closed by extending it with a straight line to the last `moveto` point.

### EXERCISE 3

See what happens with the picture when you do not use the `closepath` operator to ensure a closed path, but replace this statement by `0 -90 rlineto`.

### EXERCISE 4

You can fill the interior of a closed path with some color or shade of gray, using the `fill` operator instead of the `stroke` operator. The gray shading is obtained by the statement `c setgray`, where `c` is a number between 0 and 1 (0 corresponds with black and 1 means white). Draw a black square box that contains a smaller gray square box centered in the middle of the black one.

## 2.3 Using PostScript Code in a Microsoft Word Document

In MS word 2002, a picture can easily be inserted from a file in various formats, including the Encapsulated PostScript format. Just select the menu items INSERT, PICTURE, FROM FILE, and browse for files of type ALL PICTURES. In older versions of MS Word, the insertion of

<sup>3</sup>If necessary, scroll down to get the lower left corner of the page in view.

an Encapsulated PostScript file is less easy. In this case, you better open the EPS file within GSVIEW, select the **EPS Clip** item in the **Option** menu, copy the picture to the clipboard (by selecting the **Copy** item in the **Edit** menu or by using the shortcut **CTRL+C**), and paste into the Word document (by selecting the **Paste** item in the **Edit** menu or by using the shortcut **CTRL+C**). In this way, a preview of the display will be copied into the Word document so that the printout will also not look optimal. You have more control on the resolution of the preview when you add it explicitly to the EPS file; select for example **Edit, Add to EPS, TIFF 6 packbits**, and save to a file. Insert this EPS file with Preview by selecting the menu items **INSERT, PICTURE, FROM FILE**, and browse for files of type **ALL PICTURES**. The printout will also be of good quality.

## 2.4 Using PostScript Code in a L<sup>A</sup>T<sub>E</sub>X Document

### EXERCISE 5

Do the following steps:

1. Create a file, say `sample.tex`, that contains the following lines of L<sup>A</sup>T<sub>E</sub>X commands:

---

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
Figure~\ref{fig:square} is an example of a picture in a \LaTeX\ text.

\begin{figure}[hbt]
\begin{center}
\includegraphics{exercise2.eps}
\caption{A square.}
\label{fig:square}
\end{center}
\end{figure}
\end{document}
```

---

Above, we use the extended `graphicx` package for including the external graphic file that was prepared by us in the second exercise.

2. Typeset the L<sup>A</sup>T<sub>E</sub>X-file:

```
latex sample
```

When typesetting is successful, the device independent file `sample.dvi` is generated.

3. Convert the dvi-file `sample.dvi` into PostScript code:

```
dvips sample
```

4. Preview the PostScript file `sample.ps`, e.g., by `GSview`:
5. On a Unix platform, you can convert the PostScript document `sample.ps` into a printable PDF document as follows:

```
ps2pdf sample.ps
```

It creates the file `sample.pdf`.

6. You can avoid the intermediate PostScript generation. Just convert the DVI file immediately into a PDF document via the `dvipdf` command:

```
dvipdf sample
```

On a Windows PC, similar conversions from PostScript or DVI format to PDF format are possible.

## 2.5 Numeric Quantities

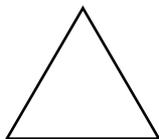
Numeric quantities in PostScript are represented by integers or real numbers, with a limited number of decimals of accuracy — 5 digits, to be precise. The PostScript interpreter GHOSTSCRIPT and the previewer GSVIEW both understand exponential notation such as  $1.23E4$  and  $1e-2$ . Assignment of numeric values can be done with statements like this

```
/pi 3.14159265 def
```

which sets the variable `pi` to be 3.14159265. The `/pi` is the *name* of the variable. In the definition and assignment you cannot use the name `pi`, but after a variable is defined in your program, any occurrence of that variable will be replaced by the last value assigned to it. For example, you may define variables  $\text{deg} = \frac{180}{\pi}$  and  $\text{rad} = \frac{\pi}{180}$ , which are useful in conversions between degrees and radians, as follows:

```
/deg 180 pi div def  
/rad pi 180 div def
```

The following program will draw an equilateral triangle with edges of size 2 cm.



```
%!PS-Adobe-3.0 EPSF-3.0  
%%BoundingBox: -2 -2 59 52  
%%BeginProlog  
% define and assign the scale  
% note: 1 cm = 28.34645 points,  
%       1 inch = 72 points.  
/s 28.34645 def  
%%EndProlog  
s s scale  
1 s div setlinewidth  
newpath  
0 0 moveto  
2 0 rlineto  
-1 3 sqrt rlineto  
closepath  
stroke  
showpage  
%%EOF
```

---

In the above example, the scale `s` has been stored in a single variable that can be used elsewhere in the program in order that a change of the scale only requires a change in a single line of the PostScript program. In this case, the statement `s s scale` scales up horizontal and vertical units by a factor of 28.34645, which corresponds with a unit of 1 cm. Because the default line width is 1 unit, lines would be drawn 1 cm wide. The statement `1 s div setlinewidth` resets the width of line to 1 point. Be warned, you cannot use `1/s setlinewidth`, but must stick to the division operator `div`.

You may already have noticed it, but PostScript uses *post-fix notation* of putting the operator after its arguments. This convention is also called *Reverse Polish Notation*. Thus, `1 s div` means ‘1 divided by s’, the product of the integers 2 and 3 is denoted by `2 3 mul`, and `3 sqrt` stands for ‘the square root of 3’.

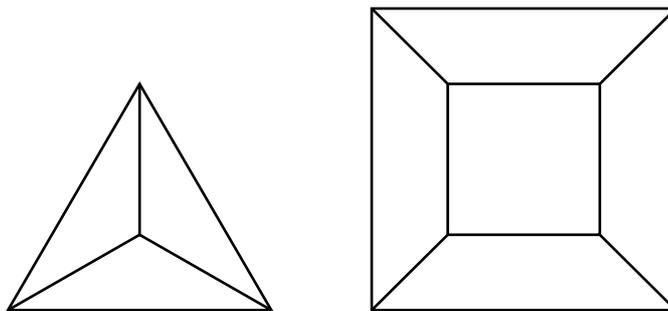
#### EXERCISE 6

Create the following red arrow with length 4 inch and arrowhead length 1 inch against a light gray, rectangular background of size 5" × 1". The red color can be defined by the `setrgbcolor` operator that expects three real numbers between 0 and 1, inclusively, representing fractional intensity of red, green and blue, respectively.



#### EXERCISE 7

Create the following two pictures:



## 2.6 The Operand Stack

The PostScript programming language is based on stacks. A *stack* is a fixed data structure in which objects are placed temporarily, one on top of another, and items are removed according to the “last in, first out” principle.

#### EXERCISE 8

Do the following steps:

1. Start the AFPL GHOSTSCRIPT interpreter and enter the statement

2 3

The command window will look like this:

```
AFPL Ghostscript 8.00 (2002-11-21)
Copyright (C) 2002 artofcode LLC, Benicia, CA. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>2 3
GS<2>
```

The <2> in the last line means that there are 2 items on the so-called *operand stack*. Henceforth, we will call this simply the stack.

2. View the contents of the stack by entering the `pstack` operator. Note that first 2 is pushed onto the stack. When 3 is placed on the stack, the number 2 is pushed to the next position down, leaving 3 on top of the stack.
3. Enter the `add` operator. When the PostScript interpreter encounters this operator and executes it as defined in the PostScript dictionary, two operands are taken off the stack and the sum of the numbers is placed on top of the stack. Verify that indeed the stack contains hereafter one item, viz. the number 5.

The effect of the statement `2 3 add` on the stack is illustrated by the following diagram:

2	3	add
<b>2</b>	<b>3</b>	<b>5</b>
	<b>2</b>	

PostScript Stack of statement `2 3 add`

The `sub` operator works in a similar manner, with the program line `3 2 sub` having the results diagrammed below. The numbers 3 and 2 are pushed on the stack. The `sub` operator subtracts the top number on the stack from the number below it. Finally, the result of subtraction is pushed on the stack.

3	2	sub
<b>3</b>	<b>2</b>	<b>1</b>
	<b>3</b>	

PostScript Stack of statement `3 2 sub`

Henceforth, we will adopt the stack notation of ‘the blue book’ [Pos86]: the effects of an operator on the stack is indicated by showing the stack’s initial condition (before the operator is executed), the operator’s name, and then the contents of the stack after the operator was executed. In this notation the example is written as: `3 2 add  $\implies$  1`.

## 2.7 Mathematical Operations and Functions

The complete list of arithmetic and mathematical operators in PostScript is given below.

Arithmetic and Mathematical Operators		
<i>Arguments</i>	<i>Operator</i>	<i>Left on Stack; Side Effects</i>
$x y$	add	$x + y$
$x y$	sub	$x - y$
$x y$	mul	$xy$
$x y$	div	$x/y$
$x y$	idiv	the integral part of $x/y$
$x y$	mod	the remainder of $x$ after division by $y$
$x$	abs	the absolute value of $x$
$x$	neg	$-x$
$x$	ceiling	the integer just above $x$
$x$	floor	the integer just below $x$
$x$	round	$x$ rounded to nearest integer
$x$	truncate	$x$ with fractional part chopped off
$x$	sqrt	square root of non-negative $x$
$y x$	atan	the polar argument of the point $(x, y)$ , in degrees between 0 and 360
$x$	cos	$\cos x$ ( $x$ in degrees)
$x$	sin	$\sin x$ ( $x$ in degrees)
$x y$	exp	$x^y$
$x$	ln	$\ln x$
$x$	log	$\log x$ (base 10)
	rand	a random integer in the range 0 to $2^{31} - 1$

### EXERCISE 9

Use GHOSTSCRIPT to compute  $(\frac{1}{2} + \frac{1}{2}\sqrt{3})^2$

### EXERCISE 10

Use GHOSTSCRIPT to verify that the following four PostScript statements yield the same result:

```
2 2 mul 1 sub
2 dup mul 1 sub
1 2 2 mul exch sub
```

The second line introduces the `dup` operator that duplicates the top item on the stack. The `exch` operator exchanges the top two items on the stack, i.e.,  $x y \text{ exch} \implies y x$ . So, the results of the third line, `1 2 2 mul exch sub`, can be diagrammed as follows:

1	2	2	mul	exch	sub
<b>1</b>	<b>2</b>	<b>2</b>	<b>4</b>	<b>1</b>	<b>-3</b>
	<b>1</b>	<b>2</b>	<b>1</b>	<b>4</b>	
		<b>1</b>			

You may wonder for what purpose the PostScript programming language contains the stack operator `exch`. The main application is the removal of arguments of a PostScript procedure from the stack. As an example, we define the arccosine function as follows: `h x acos` will leave on the stack the polar argument (in degrees between  $0^\circ$  and  $180^\circ$ ) of the point  $(x, \sqrt{h^2 - x^2})$ . Think of a triangle with hypotenuse  $h$  and horizontal side of length  $x$ . The implementation of `acos` is based on the fact that `atan` is an existing function in PostScript, which also returns its answer in degrees rather than radians.

```
GS>/acos {
  /h exch def
  /x exch def
  h dup mul x dup mul sub sqrt x atan
} def
```

Here, the first line in the definition of `acos` takes care of the following at runtime: it exchanges the top of the stack, say  $t$ , and the variable `h`, effectively builds up the definition `/h t def`, and stores the value of  $t$  in the variable `h`. The second line in the definition does a similar thing with the second item on the stack when the function is called and stores the value of the second stack element in the variable `x`. The two variables `x` and `h` can be used in the program: here, the expression  $\arctan(\sqrt{h^2 - x^2}, x)$  is computed. Let us have a look at few results to see if the function works properly.

```
GS>1 2 acos ==
60.0
GS>0.5 1 acos ==
60.0
GS>-1 2 acos ==
120.0
GS>3 sqrt neg 2 acos ==
150.0
GS>1 1 acos ==
0.0
GS>-1 1 acos ==
180.0
```

Here we have used the stack operator `==` that displays the top item on the stack and then takes it off the stack.

### EXERCISE 11

Use GHOSTSCRIPT to define the `cosine` function that computes the cosine of its argument in radians rather than degrees.

## 2.8 Operand Stack Manipulation Operators

PostScript has several operators to add, remove or rearrange items on the operand stack. The complete list of operand stack manipulation operators is given below.<sup>4</sup>

<sup>4</sup>The symbol `+` denotes the empty stack and it marks the bottom of the stack.

Operand Stack Manipulation Operators		
Operator	Meaning	Effects on Stack
clear	remove all stack items	$\vdash obj_1 \cdots obj_n \text{ clear} \implies \vdash$
cleartomark	remove all stack items down through <i>mark</i>	$mark\ obj_1 \cdots obj_n \text{ cleartomark} \implies -$
copy	duplicate top <i>n</i> stack items	$obj_1 \cdots obj_n\ n\ \text{copy} \implies$ $obj_1 \cdots obj_n\ obj_1 \cdots obj_n$
count	count the stack items	$\vdash obj_1 \cdots obj_n\ \text{count} \implies \vdash\ obj_1 \cdots obj_n\ n$
counttomark	count all stack items down through <i>mark</i>	$mark\ obj_1 \cdots obj_n\ \text{counttomark} \implies$ $mark\ obj_1 \cdots obj_n\ n$
dup	duplicate the top stack item	$obj\ \text{dup} \implies obj\ obj$
exch	exchange top two stack items	$obj_1\ obj_2\ \text{exch} \implies obj_2\ obj_1$
index	push a copy of the <i>n</i> -th stack item on the stack	$obj_n \cdots obj_0\ n\ \text{index} \implies$ $obj_n \cdots obj_0\ obj_n$
mark	push mark on stack	$\text{mark} \implies mark$
pop	remove the top stack item	$obj\ \text{pop} \implies -$
roll	roll <i>n</i> stack items up <i>j</i> times	$obj_{n-1} \cdots obj_0\ n\ j\ \text{roll} \implies$ $obj_{(j-1)\ \text{mod}\ n} \cdots obj_0\ obj_{n-1} \cdots obj_{j\ \text{mod}\ n}$

Because we will quite often use the `roll` operator, an illustrative example of its effect:

(a) (b) (c) (d) 3 1 roll  $\implies$  (a) (d) (b) (c)  
(a) (b) (c) (d) 3 -1 roll  $\implies$  (a) (c) (d) (b)

## 2.9 If PostScript Interpreting Goes Wrong

If you make a mistake in the source file and the PostScript interpreter or viewer cannot process your document without any trouble, a bunch of error statements appear. Each error message consists of two elements:

- The *Error* name, e.g. `stackunderflow`.
- The *OffendingCommand*. This is the PostScript object that was being executed at the time of the error. In general, it takes time and experience to identify and correct errors on the basis of the error messages.

The PostScript error message gives details about *what* happened, but may not tell you *why*. This makes it cumbersome to understand the errors and to handle them. In practice, it never pays to try hard to interpret the error messages of the PostScript interpreter

### EXERCISE 12

1. Start the AFPL GHOSTSCRIPT interpreter and deliberately make a mistake by entering the following statement

```
0 moveto
```

The command window will look like this:

```
AFPL Ghostscript 8.14 (2004-02-20)
Copyright (C) 2004 artofcode LLC, Benicia, CA. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>0 moveto
```

```
Error: /stackunderflow in --moveto--
Operand stack
  0
Execution stack:
  %interp_exit   .runexec2   --nostringval--  --nostringval--
  --nostringval-- 2   %stopped_push   --nostringval--
  --nostringval-- %loop_continue 2   3   %oparray_pop
  --nostringval-- --nostringval-- false 1   %stopped_push
  --nostringval-- --nostringval-- --nostringval--
  2   %stopped_push   --nostringval--
Dictionary stack:
  --dict:1112/1686(ro)(G)--  --dict:0/20(G)--  --dict:70/200(L)--
Current allocation mode is local
Last OS error: No such file or directory
Current file position is 9
GS<1>
```

Do you really understand the error message?

2. Recover from the error by clearing the stack with the `clear` statement and then enter the next statement:

```
1 0 div
```

Do you understand the error message that will appear?

3. One of the tricks to get more insight in *where* an error occurs is the introduction of debug statements. Create the file `exercise12.ps` containing the following lines of code:

```
%!PS-Adobe-3.0
0 0 moveto 10 0 lineto
(correct until here) ==
0 lineto
showpage
```

View this file with the PostScript previewer `GSVIEW` and look at the text in the error message window. Does this give you enough information about the location of the error?

### 3 Basic Graphical Primitives

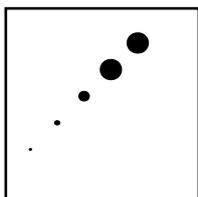
In this chapter you will learn how to build up a picture from basic graphical primitives such as lines and text objects.

### 3.1 Point

PostScript does not have a data type for a point. But you can render a point as a dot at the specified location and with specified diameter via a simple trick: draw a line of zeroth length starting at the requested point with a specified `linewidth` and with `linecap` set equal to 1, which means that the line segment (of length 0) ends with a circular caps with diameters equal to the width of the line.

#### EXERCISE 13

Try to understand the following result:



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 74 74
newpath 1 1 moveto 72 0 rlineto
0 72 rlineto -72 0 rlineto
closepath stroke
%
1 setlinecap
newpath 10 20 moveto 0 0 rlineto stroke
2 setlinewidth
newpath 20 30 moveto 0 0 rlineto stroke
4 setlinewidth
newpath 30 40 moveto 0 0 rlineto stroke
8 setlinewidth
newpath 40 50 moveto 0 0 rlineto stroke
newpath 50 60 moveto 0 0 rlineto stroke
showpage
%%EOF
```

If you really wish, you can implement points, coordinate pairs or vectors in a two-dimensional space as arrays of length 2. Using the `put` and `get` operators to store and fetch array information, respectively, you can easily implement vector calculus. The following example of a GHOSTSCRIPT session illustrates this.

- First we define a vector, say `v`, as an array and we define the procedures `xcoord v` and `ycoord v` that will allow us to refer to the first and second coordinate of a vector, respectively.

```
GS>% define the vector v
GS>/v [10 20] def
GS>% display the vector v
GS>v ==
[30 40]
GS>% define xcoord and ycoord
GS>/xcoord 0 get def
GS>/ycoord 1 get def
```

We verify that this works:

```

GS>v xcoord ==
30
GS>v ycoord ==
40

```

- Scalar multiplication can be defined as follows:

```

GS>/scalarmul {3 dict begin % c [a b] on stack
  aload pop % c a b on stack
  /b exch def
  /a exch def
  /c exch def
  [c a mul c b mul]
  end } def
GS>v ==
[30 40]
GS>2 v scalarmul ==
[60 80]

```

The procedure `scalarmul` needs some explanation: we want the variables that we use in the procedure to be local to this procedure, so that they do not interfere other variables with the same name outside the procedure. To do this, we add in the first line `3 dict begin` that sets up a mechanism of three local variables and we restore the original environment when the `end` clause is encountered. The `3` in the statement `3 dict begin` means that we reserve space for 3 local variables.

The second statement `aload pop` does the following: the `aload` operator takes an array as its argument and places the individual elements of that array, and then the array itself, on the stack. The `pop` operator pushed the top item off the stack. Thus, the combination of these two operators replaces the array on top of the stack by the individual elements.

Our procedure `scalarmul` returns an array of length 2 on the stack.

- The scalar multiplication can also be defined without local variables, but the following code will be less easy to understand. That is why we add comments that describe the stack at each stage. The main advantage of this procedure is that it is more efficient: computer timing reveals that the version without local variables is about three times faster than the one that uses local variables.

```

GS>/scalarmul {      % c [a b] on stack
  aload pop        % c a b
  3 -1 roll        % a b c
  dup              % a b c c
  3 1 roll         % a c b c
  mul              % a c bc
  3 1 roll         % bc a c
  mul              % bc ac
  exch             % ac bc
  2 array astore % [ac bc] on stack

```

```

    } def
GS>v ==
[30 40]
GS>2 v scalarmul ==
[60 80]

```

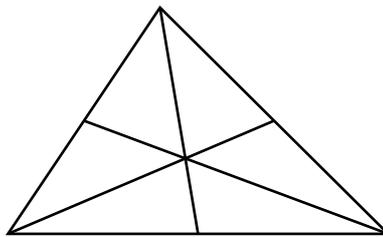
The statement `2 array astore` takes two items and a 2-dimensional array off the stack and places all of the items into an array, which is left on the stack.

#### EXERCISE 14

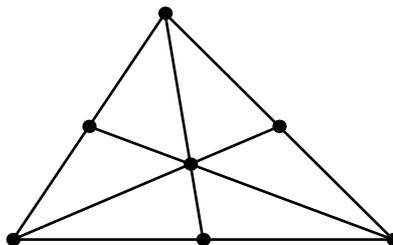
Write a procedure, say `addvec`, that adds two vectors. Make two versions of your procedure: one with local variables and one without local variables.

#### EXERCISE 15

1. Create the following geometrical picture of an acute-angled triangle together with its three medians in an Encapsulated PostScript file:<sup>5</sup>

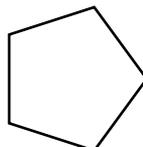


2. Mark the vertices of the triangle and the intersection points of the medians with the sides of the triangle and with each other with a dot, as shown in the picture below:



#### EXERCISE 16

Define the procedure `dir` command that gives a point on the unit circle. For example, `30 dir` should generate the point  $(0.86603, 0.5)$  ( $= (\frac{1}{2}\sqrt{3}, \frac{1}{2})$ ). Use your `dir` procedure to generate a regular pentagon, like in the picture below.

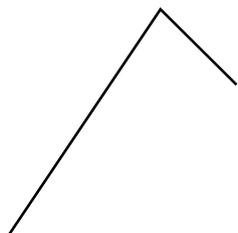


<sup>5</sup>The *A*-median of a triangle *ABC* is the line from *A* to the midpoint of the opposite edge *BC*.

## 3.2 Curve

### 3.2.1 Open and Closed Curves

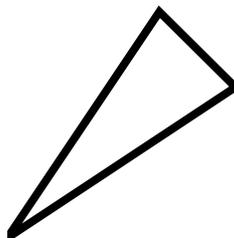
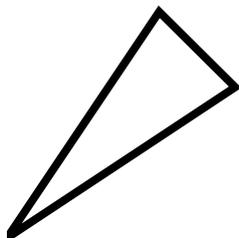
PostScript can draw straight lines as well as curved ones. You have already seen that a `lineto` statement draws a straight lines connecting the current point with the specified one. An example



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -1 -1 86 86
%%BeginProlog
/cm {28.34645 mul} def
%%EndProlog
newpath
0 0 moveto
2 cm 3 cm lineto
3 cm 2 cm lineto
stroke
showpage
%%EOF
```

---

Closing the above path is done either by inserting the statement `0 0 lineto` or the statement `closepath` before the `stroke` command. The difference between these two methods is that the path extension with the starting point via a `lineto` command only has the optical effect of closing the path. This means that only with the `closepath` extension it really becomes a closed path. The difference is illustrated by the following two pictures, where we have made the lines thicker with the statement `3 setlinewidth`.



---

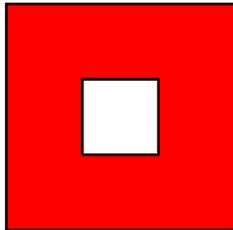
### 3.2.2 Filled Shapes

The `fill` command paints the area inside the current path with the current color. If the path consists of several disconnected subpaths, `fill` paints the insides of all subpaths, considered together. Any subpaths that are open are implicitly closed before being filled. The *nonzero winding number rule* is used to determine what points lie inside the path. It works as follows: draw a ray from the point of interest in any direction and examine the places where a segment of the path crosses the ray. Counting from 0, add 1 each time a path segment crosses the ray from left to right and subtract 1 each time a segment crosses from right to left. After counting all crossings, if the result is 0 then the point is outside the path; otherwise it is inside.

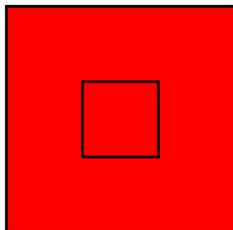
An alternative is the *even-odd rule*: draw a ray from the point of interest in any direction and count the places where a segment of the path crosses the ray. If the number of crossings is even, then the point is outside the path; otherwise it is inside.

**EXERCISE 17**

Explain the difference in filling of the following two pictures on the basis of the given PostScript programs.



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -5 -5 90 90
%%BeginProlog
/cm {28.34645 mul} def
%%EndProlog
newpath
  0 0 moveto
  3 cm 0 rlineto
  0 3 cm rlineto
-3 cm 0 rlineto
  0 -3 cm rlineto
  1 cm 1 cm moveto
  0 1 cm rlineto
  1 cm 0 rlineto
  0 -1 cm rlineto
-1 cm 0 rlineto
gsave 1 0 0 setrgbcolor fill grestore
stroke
showpage
%%EOF
```



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -5 -5 90 90
%%BeginProlog
/cm {28.34645 mul} def
%%EndProlog
newpath
  0 0 moveto
  3 cm 0 rlineto
  0 3 cm rlineto
-3 cm 0 rlineto
  0 -3 cm rlineto
  1 cm 1 cm moveto
  1 cm 0 rlineto
  0 1 cm rlineto
-1 cm 0 rlineto
  0 -1 cm rlineto
gsave 1 0 0 setrgbcolor fill grestore
stroke
showpage
%%EOF
```

The PostScript code in the previous exercise needs more explanation: we must still discuss for what purpose the `gsave` and `grestore` operators are used here. What would have happened if

we had left these operators out of the program code? To answer this question it is important to realize that after filling the current path, the `fill` operator clears it with an implicit `newpath` operation. So, without the `gsave` and `grestore` operators, the operator `stroke` would have nothing to stroke after the clearance of the current path with `fill`! Under these circumstances we would have to build the path used for the filling a second time in order to outline it. This would lead to ugly code as

```
newpath
 0 0 moveto
 3 cm 0 rlineto
 0 3 cm rlineto
-3 cm 0 rlineto
 0 -3 cm rlineto
 1 cm 1 cm moveto
 1 cm 0 rlineto
 0 1 cm rlineto
-1 cm 0 rlineto
 0 -1 cm rlineto
1 0 0 setrgbcolor
fill
 0 0 moveto
 3 cm 0 rlineto
 0 3 cm rlineto
-3 cm 0 rlineto
 0 -3 cm rlineto
 1 cm 1 cm moveto
 1 cm 0 rlineto
 0 1 cm rlineto
-1 cm 0 rlineto
 0 -1 cm rlineto
0 setgray
stroke
```

In order to eliminate redundancy, PostScript offers the programmer the opportunity to save a copy of the current graphics state and come back to this copy at a later time. The graphics state holds data about the current path, the current color and line width, the current user coordinate system, and more. The `gsave` and `grestore` operators save and retrieve the current graphics state, respectively. In our example in the previous exercise, the `gsave` operator saves a copy of the current graphics state on a *graphics state stack*. So, a copy of the current path is set apart for later usage. Hereafter we choose a red color to fill the current path. The `grestore` operator takes the most recently gsaved graphics state from the graphics state stack. In our case, the path used for the filling is back again and can be stroked.

Let us summarize: after filling the current path, the `fill` operator clears it with an implicit `newpath` operation. To preserve the current path across a `fill` operation, use the sequence

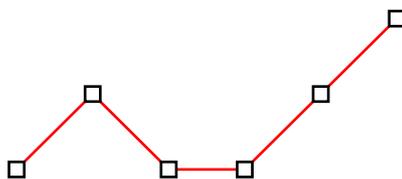
```
gsave
fill
grestore
```

The same holds for the `stroke` operator: after painting the current path, the `stroke` operator clears it with an implicit `newpath` operation. To preserve the current path across a `stroke` operation, use the sequence

```
gsave
  stroke
grestore
```

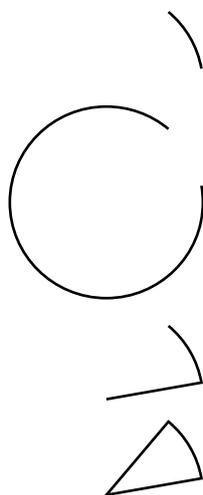
### EXERCISE 18

Make a procedure box that allows you to draw a white square box of user specified width with a black outline at any given position and use it to create the following picture.



### 3.2.3 Straight and Circular Line Segments

As we have seen many times before, the `lineto` and `rlneto` commands can be used to draw straight line segments. But curved segments are possible as well, the simplest one being a circular one. The following example gives you an idea how the `arc` operator works.




---

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 80 196
newpath 40 160 36 10 50 arc stroke
newpath 40 116 36 50 10 arc stroke
newpath
40 42 moveto
40 42 36 10 50 arc
stroke
newpath
40 6 moveto
40 6 36 10 50 arc
closepath
stroke
showpage
%%EOF
```

---

The statement

```
 $x$   $y$   $r$   $angle_1$   $angle_2$  arc
```

appends a circular arc to the current path, possibly preceded by a straight line segment. The arc is centered at coordinates  $(x, y)$  in user space, with radius  $r$ . The operands  $angle_1$  and

$angle_2$  define the endpoints of the arc by specifying the angles of the vectors joining them to the center of the arc. The angles are measured in degrees counter-clockwise from the positive  $x$  axis of the current user coordinate system (see Figure 1).

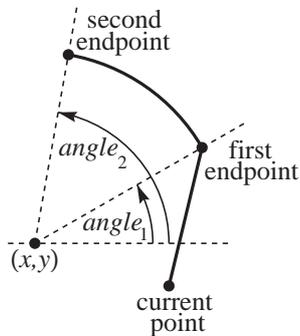
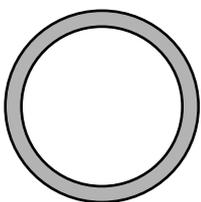
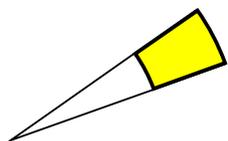


Figure 1: The arc operator.

If  $angle_2$  is less than  $angle_1$ , it is increased by multiples of 360 until it becomes greater than or equal to  $angle_1$ . No other adjustments are made to the two angles. In the example, the effect of the statement `40 116 36 50 10 arc` will be the same as of the statement `40 116 36 10 50 arcn`. The `arcn` operator (`arc` negative) is similar to `arc`, differing only in that it constructs an arc in a clockwise direction instead of the default counter-clockwise direction.

**EXERCISE 19**

Create the following pictures in Encapsulated PostScript files:

1. 
2. 
3. 

There is a third operator for appending a circular arc to the current path, possibly preceded by a straight line, viz. `arct`. It is mainly used for making bends in a path. The statement

$$x_1 y_1 x_2 y_2 r \text{ arct}$$

appends the circular arc that is defined by a radius  $r$  and two tangent lines, drawn from the current point  $(x_0, y_0)$  to  $(x_1, y_1)$  and from  $(x_1, y_1)$  to  $(x_2, y_2)$ , as in Figure 2.

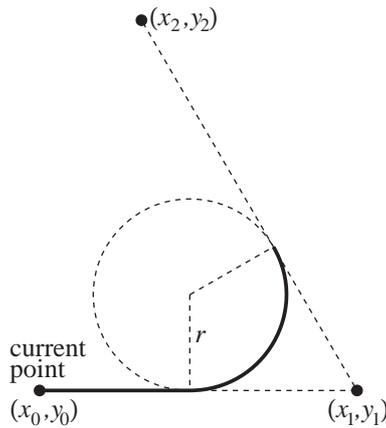


Figure 2: The arct operator.

**EXERCISE 20**

Create the following picture:



**3.2.4 Cubic Bézier Line Segment**

Two points can also be connected by a cubic Bézier curve, which needs, in order to be determined, two intermediate control points in addition to the end points. The points on the curved segment from points  $P_0$  to  $P_3$  with post control point  $P_1$  and pre control point  $P_2$  are determined by the formula

$$P(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t) t^2 P_2 + t^3 P_3,$$

where  $t \in [0, 1]$ . Each point of the cubic Bézier line segment is a weighted average of the four points  $P_0, \dots, P_3$ . This implies that this line segment will stay inside the convex hull of the defining points. In Figure 3, the control points are drawn as open dots and connected to their ‘parent point’ (marked with a black dot) with gray line segments.

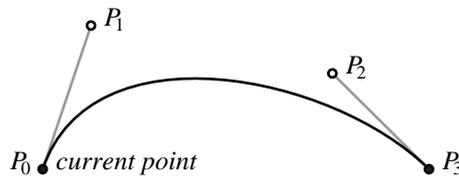
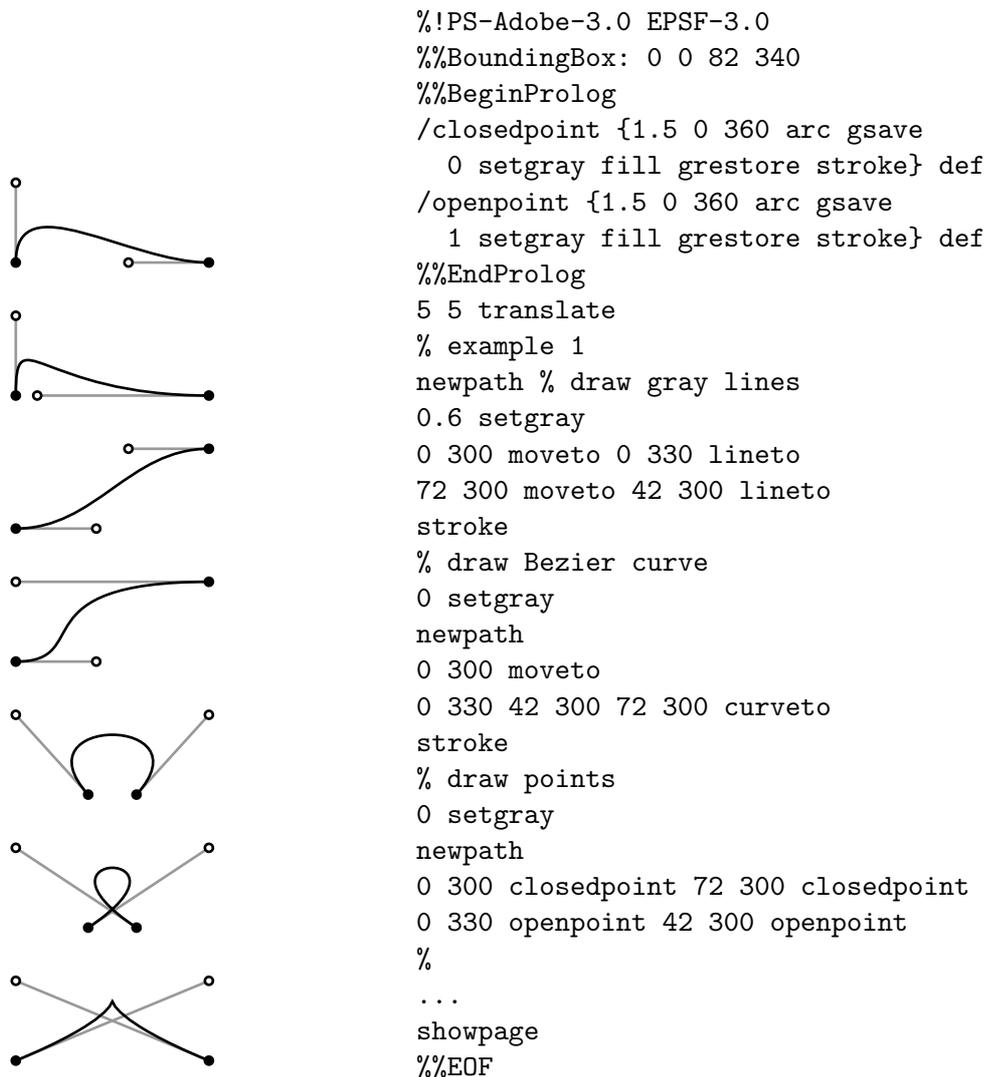


Figure 3: The curveto operator.

The statement

`x1 y1 x2 y2 x3 y3 curveto`

makes a cubic Bézier segment starting at the current point  $P_0 = (x_0, y_0)$  in the path to the point  $P_3 = (x_3, y_3)$  with post control point  $P_1 = (x_1, y_1)$  and pre control point  $P_2 = (x_2, y_2)$ . The curve moves from the starting point in the direction of the post control point, but possibly bends after a while in another direction. The further away the post control point is, the longer the curve keeps this direction. Similarly, the curve arrives at a point coming from the direction of the pre control point. The further away the pre control point is, the earlier the curve gets this direction. It is as if the control points pull their parent point in a certain direction and the further away a control point is, the stronger it pulls. The examples in figure below give you an idea of how cubic Bézier line segments may look like. We only show the PostScript code that creates the top picture; the other pictures are created similarly.



Let us summarize a few facts about a Bézier segment with end points  $P_0$  and  $P_3$ , and with control points  $P_1$  and  $P_2$ .

- The curve starts at  $P_0$  straight in the direction of the post control point  $P_1$ .

- The curve ends at  $P_3$ , coming straight from the direction of the pre control point  $P_2$ .
- The longer the line from a control point to its parent end point, the tighter the curve sticks to that line.
- The whole cubic Bézier line segment is contained inside the convex hull of the defining four points  $P_0, \dots, P_3$ .



#### EXERCISE 21

Draw the Yin-Yang symbol<sup>6</sup> that looks like

### 3.3 Angle and Direction Vector

It is not difficult to define your own PostScript procedure, say `angle`, that takes a point, interprets it as a vector, and computes the two-argument arctangent, i.e., gives the angle corresponding with the vector. The definition of the inverse procedure, say `dir`, that generates a point on the unit circle at a given angle with the horizontal axis is as easy.

#### EXERCISE 22

Verify that the implementation of a direction vector and the angle of a vector can be made as follows:

```
/angle {exch atan} def
/dir {dup cos exch sin} def
```

In the example below we use these procedures to draw a bisector of a triangle. We will assume that the following auxiliary procedures `addvec` and `subvec` for sum and difference of two vectors are defined in the prolog of the PostScript code, as well as the procedure `scalarvec` for a scalar product.

```
/addvec { % a b c d on stack
  3 -1 roll % a c d b
  add % a c b+d
  3 1 roll % b+d a c
  add % b+d a+c
  exch % a+c b+d
} def
/subvec { % a b c d on stack
  3 -1 roll % a c d b
  sub neg % a c b-d
  3 1 roll % b-d a c
  sub % b-d a-c
  exch % a-c b-d
} def
```

<sup>6</sup>See [www.chinesefortunecalendar.com/YinYang.htm](http://www.chinesefortunecalendar.com/YinYang.htm) for details about the symbol.

```

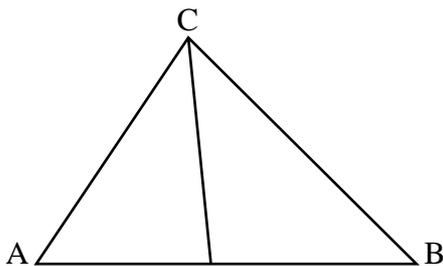
/scalarvec{ % c a b on stack
  3 -1 roll % a b c
  dup      % a b c c
  3 1 roll % a c b c
  mul      % a c bc
  3 1 roll % bc a c
  mul      % bc ac
  exch     % ac bc
} def

```

### EXERCISE 23

Verify that the above procedures `addvec`, `subvec`, and `scalarvec` make sense.

In the example of a bisector of an acute-angled triangle below we use the `clip` operator to restrict the drawing area to the triangle. In general, this operator clips the drawing to the region outlined by the current path. The nonzero winding number rule is used to determine what points lie inside the current path.



```

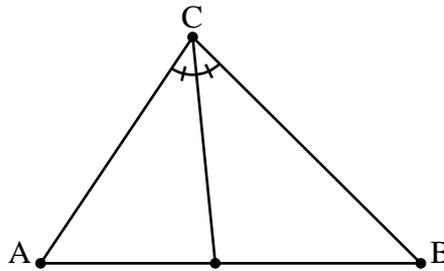
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 155 95
%%BeginProlog
% ...
/s 28.34645 def
/A {0 0} def /B {5 0} def /C {2 3} def
/trianglepath {newpath A moveto B lineto
  C lineto closepath} def
%%EndProlog
s dup scale 1 s div setlinewidth
12 s div 5 s div translate
% draw the triangle with vertices A, B, C
trianglepath stroke
gsave % clip drawing
  trianglepath clip
  % gamma is the bisector angle at C
  /gamma {A C subvec angle 2 div
    B C subvec angle 2 div add} def
  newpath C moveto
  10 gamma dir scalarvec rlineto stroke
grestore
% draw labels
/Times-Roman findfont 12 s div scalefont setfont
A 0.4 0 subvec moveto (A) show
B 0.1 0 addvec moveto (B) show
C -0.15 0.1 addvec moveto (C) show
showpage
%%EOF

```

In section 3.7 we will explain the way text has been added to the triangle. For the moment, you can just mimic this labeling in the exercises below.

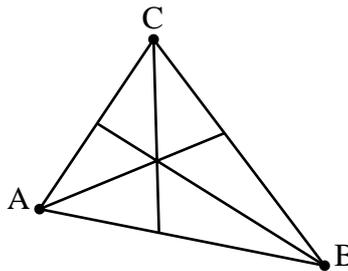
**EXERCISE 24**

Extend the above picture to the following geometric diagram, which illustrates better that a bisector is actually drawn for the acute-angled triangle.<sup>7</sup>



**EXERCISE 25**

Consider the triangle with vertices  $A = (0, 0)$ ,  $B = (3.75, -0.75)$ , and  $C = (1.5, 2.25)$ . Create the following picture that illustrates all bisectors of the triangle  $ABC$ .



### 3.4 Arrow

PostScript has no built-in commands for drawing arrows. Program 4 in the cookbook section of the blue book [Pos86] is a sample program for drawing arrows. Instead of using this program we will discuss a simple program for drawing the arrowhead alone, taken from listing 3.4 in the green book [Pos88]. Let us first have a look at our PostScript procedure `arrowhead` and see how it works. Suppose that current point has coordinates  $(x_0, y_0)$ , then the following call of our procedure

`s x1 y1 arrowhead`

draws an arrowhead with the tip at point  $(x_0, y_0)$ , coming from the direction of point  $(x_1, y_1)$ , and scaled by a factor  $s$ . The comments in the program code, which describe the stack at each moment during execution of the procedure, illustrate this behavior. One thing to learn from this is that the `currentpoint` operator places the coordinates of the current point on the stack. The transformations `rotate` and `scale` change the current coordinate system. We let `arrowhead` clear the current path with an `newpath` operation in order to avoid unwanted side effects.

<sup>7</sup>You may use the fact that segment AB is horizontal in the computation of the intersection point of the bisector and its opposite edge.

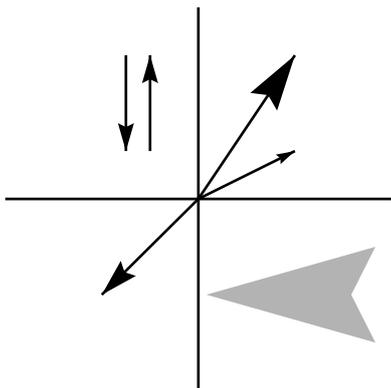
```

/arrowhead {% stack: s x1 y1, current point: x0 y0
  gsave
    currentpoint    % s x1 y1 x0 y0
    4 2 roll exch  % s x0 y0 y1 x1
    4 -1 roll exch % s y0 y1 x0 x1
    sub 3 1 roll   % s x1-x2 y0 y1
    sub exch       % s y0-y1 x1-x2
    atan rotate    % rotate over arctan((y0-y1)/(x1-x2))
    dup scale      % scale by factor s
    -7 2 rlineto 1 -2 rlineto -1 -2 rlineto
    closepath fill % fill arrowhead
  grestore
  newpath
} def

```

### EXERCISE 26

Assuming that the above procedure `arrowhead` is present in the prolog, verify the outcome of the following PostScript code.



```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 150 150
%%BeginProlog
/arrowhead {...} def
%%EndProlog
75 75 translate
newpath % draw axes
-72 0 moveto 144 0 rlineto
0 -72 moveto 0 144 rlineto
stroke
newpath % draw black arrows
0 0 moveto 36 18 lineto stroke
newpath 36 18 moveto 1 0 0 arrowhead
0 0 moveto 36 54 lineto stroke
newpath 36 54 moveto 3 0 0 arrowhead
0 0 moveto -36 -36 lineto stroke
newpath -36 -36 moveto 2 0 0 arrowhead
-18 18 moveto -18 54 lineto stroke
newpath -18 54 moveto 1.5 -18 18 arrowhead
-27 18 moveto -27 54 lineto stroke
newpath -27 18 moveto 1.5 -27 54 arrowhead
% draw gray arrowhead
0.7 setgray
36 54 moveto 3 -36 moveto 9 4 -36 arrowhead
showpage
%%EOF

```

### 3.5 Circle, Ellipse, Square, and Rectangle

You have already seen that you can draw a circular arc with center  $x, y$ , radius  $r$ , starting angle  $angle_1$ , and ending angle  $angle_2$  by the statement

```
x y angle1 angle2 arc
```

Program 3 in the cookbook section of the blue book [Pos86] is a sample program for drawing elliptical arcs (there is no built-in PostScript operator for this purpose) from basic PostScript graphic primitives. This procedure, called `ellipse`, takes six operands: the  $x$  and  $y$  coordinates of the center of the ellipse,<sup>8</sup> the ‘radius’ of the ellipse in the horizontal direction, the ‘radius’ of the ellipse in the vertical direction, the starting angle of the elliptical arc, and the ending angle of the elliptical arc. The implementation is based on translation to the center of the ellipse, scaling the coordinate system by the horizontal and vertical radius values, and addition of a circular arc that is centered at the origin with a 1 unit radius to the current path.

```
/ellipse {7 dict begin
  /endangle exch def
  /startangle exch def
  /yradius exch def
  /xradius exch def
  /yC exch def
  /xC exch def
  /savematrix matrix currentmatrix def % save current transformation matrix
  xC yC translate % translate to center of ellipse
  xradius yradius scale % scale by radius values
  0 0 1 startangle endangle arc % add arc to path
  savematrix setmatrix % restore the transformation matrix
end
} def
```

The program code can only be fully understood once transformations have been discussed; at that time we will come back to this example. Nevertheless we give an example that also illustrates why we cannot simply use `gsave` and `grestore` to make scaling a local operation. We assume that the above procedure `ellipse` is present in the prolog of the PostScript program on the next page.

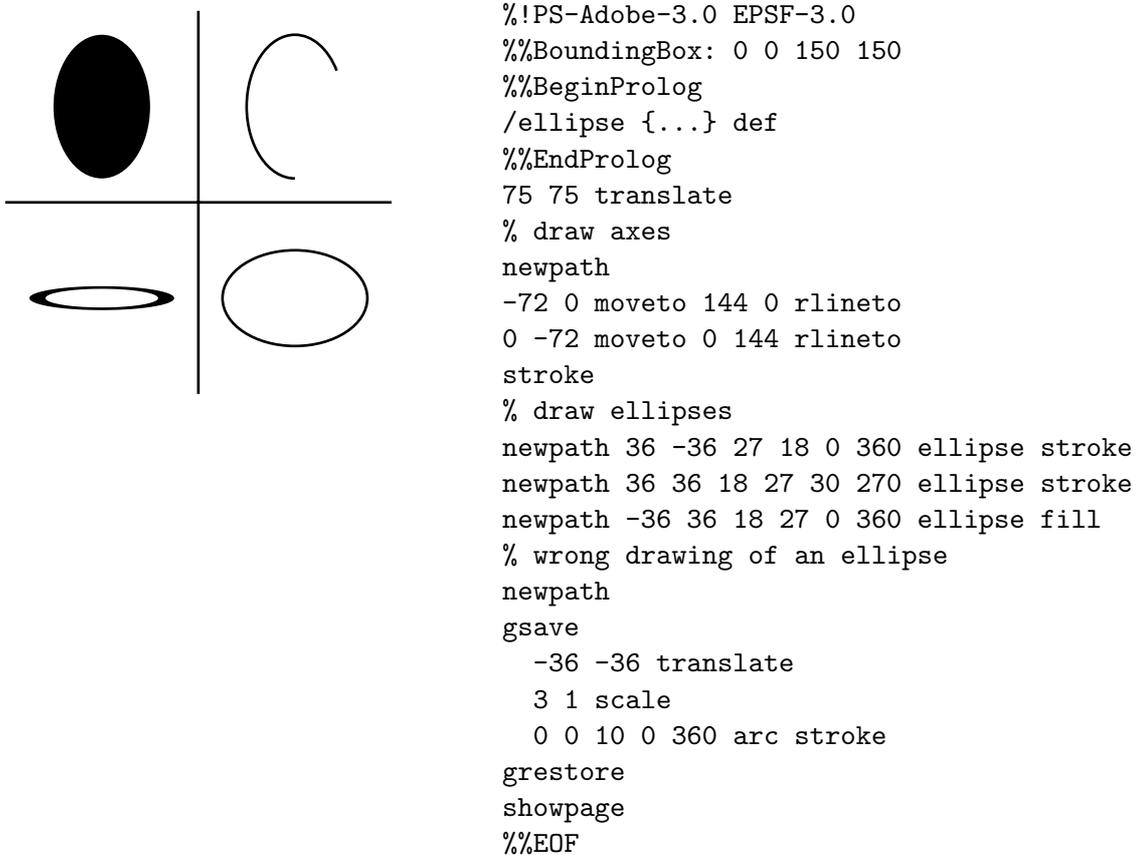
There are two problems in the example with the use of `gsave` and `grestore` for drawing an ellipse:

1. Non-uniform scaling will make the thickness of the elliptical arc not equal for each point on the curve.
2. The arc segment that is added to the path is not saved.

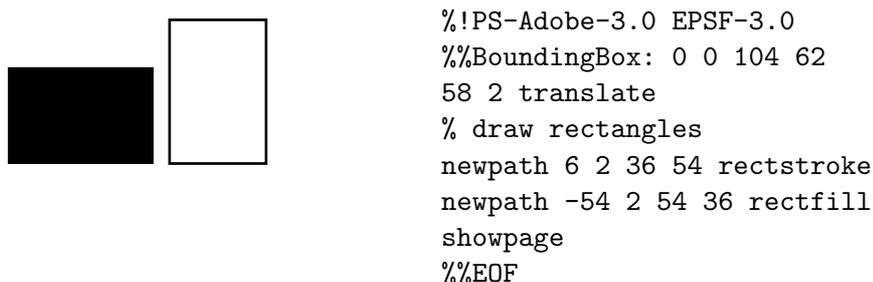
These problems are resolved by saving the current transformation matrix and restoring it explicitly after adding the elliptical arc to the path.

---

<sup>8</sup>the center of an ellipse is defined as the intersection point of the major and minor axes.



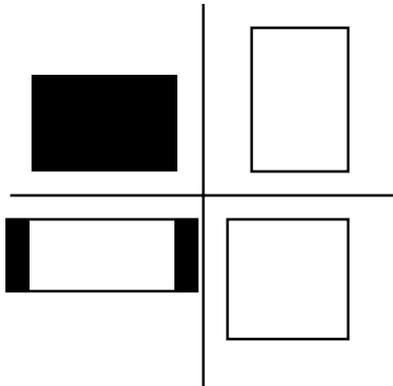
PostScript has built-in commands for drawing open or filled rectangles, viz., `rectstroke` and `rectfill`. The following example illustrate the most common use of these painting operators.



But again, if these operators were not present in PostScript, it would not have been too difficult to construct them yourself from basic PostScript graphic primitives. In the next exercise we will define another operator `rectangle` that constructs a rectangle with the current point as lower left corner and that takes two arguments, viz., the width  $w$  and the height  $h$  of the rectangle. So, the arguments are similar to the ones of `rectstroke` and `rectfill`, but our procedure just constructs the rectangular path and does not paint it.

**EXERCISE 27**

Try to understand the outcome of the PostScript program on the right.



```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 150 150
%%BeginProlog
/rectangle {% stack: w h
             % current point: x y
             currentpoint % w h x y
             moveto exch % h w
             dup          % h w w
             3 1 roll    % w h w
             0 rlineto   % w h
             0 exch rlineto % w
             neg 0 rlineto % -
             closepath
           } def
%%EndProlog
75 75 translate
% draw axes
newpath
-72 0 moveto 144 0 rlineto
0 -72 moveto 0 144 rlineto
stroke
% draw rectangles
newpath
9 -54 moveto 45 45 rectangle
18 9 moveto 36 54 rectangle
stroke
newpath -64 9 moveto 54 36 rectangle fill
newpath
gsave
  9 1 scale
  -0.7 -9 moveto
  0 -27 rlineto
  -7 0 rlineto
  0 27 rlineto
  closepath stroke
grestore
stroke
showpage
%%EOF

```

### 3.6 Commonly Used Path Construction Operators and Painting Operators

Below we list commonly used path construction operators and painting operators. Some of these operators have not been discussed before and are underlined in the first column, but their meaning is hopefully clear from the description in the tables.

Path Construction Operators		
<i>Operator</i>	<i>Meaning</i>	<i>Effects on Stack</i>
<code>arc</code>	append counter-clockwise an arc from angle $a$ to angle $b$ , center $(x, y)$ , and radius $r$	$x\ y\ r\ a\ b\ \text{arc} \implies -$
<code>arcn</code>	append clockwise an arc from angle $a$ to angle $b$ , center $(x, y)$ , and radius $r$	$x\ y\ r\ a\ b\ \text{arcn} \implies -$
<code>arct</code>	append a tangent arc defined by radius $r$ and tangent lines from the current point to $(x_1, y_1)$ and $(x_2, y_2)$	$x_1\ y_1\ x_2\ y_2\ r\ \text{arct} \implies -$
<code>clip</code>	clip drawing to the region outlined by the current path using the nonzero winding number rule	$- \text{ clip} \implies -$
<code>closepath</code>	connect subpath back to starting point	$- \text{ closepath} \implies -$
<code>currentpoint</code>	return coordinates of current point	<code>currentpoint</code> $\implies x\ y$
<code>curveto</code>	append cubic Bézier line segment with the current point and $(x_3, y_3)$ as end points and with control points $(x_1, y_1)$ and $(x_2, y_2)$	$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ <code>curveto</code> $\implies -$
<code>eoclip</code>	clip drawing to the region outlined by the current path using the even-odd rule	$- \text{ eoclip} \implies -$
<code>initclip</code>	set clipping path to device default	$- \implies -$
<code>lineto</code>	append straight line to $(x, y)$	$x\ y\ \text{lineto} \implies -$
<code>moveto</code>	set current point to $(x, y)$	$x\ y\ \text{moveto} \implies -$
<code>newpath</code>	initialize current path to be empty	$- \text{ newpath} \implies -$
<code>pathbbox</code>	return bounding box of current path defined by coordinates of lower left point and upper right point	$- \text{ pathbbox} \implies$ $ll_x\ ll_y\ ur_x\ ur_y$
<code>rcurveto</code>	perform relative <code>curveto</code> operands are relative displacements from the current point $(x_0, y_0)$ instead of absolute coordinates, i.e., $dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$ <code>rcurveto</code> is the same as $x_0 + dx_1\ y_0 + dy_1\ x_0 + dx_2\ y_0 + dy_2\ x_0 + dx_3\ y_0 + dy_3$ <code>curveto</code>	$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$ <code>rcurveto</code> $\implies -$
<code>rectclip</code>	clip drawing to the region outlined by the current rectangular path of given width $w$ and height $h$ , and with lower left corner $(x, y)$	$x\ y\ w\ h\ \text{rectclip} \implies -$
<code>rlineto</code>	perform relative <code>lineto</code> , i.e.	$dx\ dy\ \text{rlineto} \implies -$
<code>rmoveto</code>	perform relative <code>moveto</code> , i.e.	$dx\ dy\ \text{rmoveto} \implies -$
<code>setbbox</code>	set bounding box for current path specifying the coordinates of the lowerleft point and upper right point	$ll_x\ ll_y\ ur_x\ ur_y$ <code>setbbox</code> $\implies -$
<code>strokepath</code>	replace current path with its outline	$- \text{ strokepath} \implies -$

Painting Operators		
<i>Operator</i>	<i>Meaning</i>	<i>Effects on Stack</i>
<code>eofill</code>	fill current path with current color using the even-odd rule	<code>- eofill</code> $\implies$ <code>-</code>
<code>fill</code>	fill current path with current color using the nonzero winding number rule	<code>- fill</code> $\implies$ <code>-</code>
<code>rectstroke</code>	draw the rectangular path of given width $w$ and height $h$ , and with lower left corner $(x, y)$	<code>x y w h rectstroke</code> $\implies$ <code>-</code>
<code>rectfill</code>	fill the rectangular path of given width $w$ and height $h$ , and with lower left corner $(x, y)$	<code>x y w h rectfill</code> $\implies$ <code>-</code>
<code>stroke</code>	draw line along current path	<code>- stroke</code> $\implies$ <code>-</code>

Recall that the operators `stroke` and `fill` destroy the current path; the behavior of the `eofill` operator is identical to `fill` in this respect. All other painting operators listed above do not alter the current path.

### 3.7 Text

In this section we will discuss ways to add text to a picture and how to import a mathematical formula created by  $\text{\LaTeX}$  in a picture.

#### 3.7.1 Simple PostScript Text

In exercises 23 to 25 on pages 26–27 you have already used the basic mechanism to put text on a page. Here we will explain some basic methods.

Putting text on a page consists basically of three main steps:

1. Set up a font to use.
2. Set the current point to where the lower left corner of the text will be.
3. Give the string to print to the `show` operator.

Let us have a close look at the three steps. The process of specifying the desired font consists of three steps:

1. Find the information describing the font in a dictionary called `FontDirectory`. For example,

```
/Times-Roman findfont
```

looks up the name `FontDirectory`, retrieves the information about the Times Roman font, and leaves it on the stack.

2. Set the size of the font with `scafont`. The size is specified by the minimum vertical separation necessary between lines of text. Thus,

*font dictionary* 12 **scalefont**

will scale the font such that successive lines of text are twelve points apart. A new font dictionary will be the result of **scalefont**

3. Establish the scaled font as the *current font*, in which all text is to be printed. This is done with the **setfont** operator, which takes the font dictionary off the stack and makes it the current font.

Putting all things together, suppose that we want to start typesetting in Times Roman and that we want it to be 12 points, then the following PostScript code would set up the correct font:

```
/Times-Roman findfont % get the basic font
12 scalefont          % scale the font to 12 points
setfont               % make it the current font
```

Once you have set the current font and are ready to print something, you can use the **moveto** operator to set the current point. Then place the text that you want to render on the stack as a string by enclosing it by rounded brackets and call the **show** operator. The **show** operator is the basic operator for printing strings of text. It takes a string and prints it out in the current font and with the lower left corner at the current point. PostScript considers text just like graphical objects and it allows transformations to be applied to it or different colorings to be used. For example, running the following code

```
newpath
1 0 0 setrgbcolor
72 72 moveto
(Hello world!) show
```

right after the font selection code above, you would get the string **Hello world!** printed an inch in from the lower left corner in a 12 point Times-Roman and in red.

The standard PostScript fonts that you can choose are listed below:

Times-Roman	Times-Italic	Times-Bold	Times-BoldItalic
Helvetica	Helvetica-Oblique	Helvetica-Bold	Helvetica-BoldOblique
Courier	Courier-Oblique	Courier-Bold	Courier-BoldOblique
Symbol			

The following example (on the next page) prints text in four different sizes of Times Roman. It needs some explanation. The variable **vpos** is used to keep track of the current point's vertical position. The **newline** moves the current point down twenty-five points by decreasing **vpos** and using it with a **moveto**. The **showtext** procedure takes three argument: the coordinates of the lower left corner where the text should start and the requested font size. We use this font size to compute the horizontal displacement between the printed word **Times-Roman** and the number that stands for the size of the font. In order to print this font size, the number must first be converted into a string. Here, the **cvs** operator is used for this purpose. It takes two arguments off the stack: a string object and the object that must be converted into text and that is stored in the string argument. **cvs** returns the string object on the stack so

that it can be printed with the `show` operator. The string object that we need on top of the stack when we call the `cvs` operator is created by the `string` operator. We do this in the prolog with the statement `/temp 2 string`, i.e., we store a string object of length two in the variable `temp`, with each element initialized with the integer 0.

---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 145 85
%%BeginProlog
/vpos 77 def % vertical position
/newline { /vpos vpos 25 sub def
           2 vpos moveto} def
/temp 2 string def
/showtext {% stack: x, y, size
           /Times-Roman findfont % x, y, size, font
           exch dup 3 1 roll % x, y, size, font, size
           scalefont setfont % x, y, size
           3 1 roll % size, x, y
           moveto (Times-Roman) show % size
           dup % size size
           3 div 0 rmoveto % size
           temp cvs show
        } def
%%EndProlog
% draw texts in different font sizes
newpath
Times-Roman 5 2 vpos 5 showtext newline
Times-Roman 10 2 vpos 10 showtext newline
Times-Roman 15 2 vpos 15 showtext newline
Times-Roman 20 2 vpos 20 showtext
showpage
%%EOF

```

---

In practice, when you put text on a page, you will use several fonts in one PostScript program. Then it is convenient to define operators for choosing the font; see the following example.

---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 35 13
%%BeginProlog
/symbol10 {/Symbol findfont 10 scalefont
           setfont} bind def
/symbol7 {/Symbol findfont 7 scalefont
           setfont} bind def
%%EndProlog

```

---

---

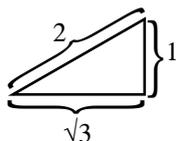
$(\alpha_1 + \alpha_2)$	<pre> newpath 2 5 moveto symbol10 (\050a) show 0 -3 rmoveto symbol17 (1) show 0 3 rmoveto symbol10 ( + a) show 0 -3 rmoveto symbol17 (2) show 0 3 rmoveto symbol10 (\051) show %%EOF </pre>
-------------------------	---

---

The above example needs some explanation. In the first place, there is the last part `bind def` in the font procedures. Without `bind` the code also works well. So what is the purpose of using the operator `bind`? A short answer is: the effect of executing `bind` is to perform *early name binding*, i.e., to replace executable operator names in a procedure by their values, so that the procedure works more efficiently.

A second aspect in the application of the Symbol font is the character code, which is a bijection between numeric codes and characters recognized by the computer system. For example, the code `\050` and `\051` stand for the open and close round brackets, respectively. The standard encoding for the alphanumeric fonts, such as Times, Helvetica, and Courier, is similar to the ASCII standard and is a number in octal notation. If a character has a corresponding key on a computer keyboard, you can simply use the key. In all other cases you must use the character code. Thus, in the symbol font, the letters `a` and `W` will be printed as  $\alpha$  and  $\Omega$ , respectively. The square root symbol, for example, has no key equivalent and must be entered as `\326`. At the end of this chapter, we will list the encoding for the Symbol font.

We end this section with an example that illustrates that PostScript does not really distinguish between graphical objects and textual objects. In the example, we scale the font and transform the coordinate system as we wish; the numbers required for a nice display are found by trial and error.




---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 70 55
%%BeginProlog
/cm {28.34645 mul} def
/mf {/Symbol findfont 10 scalefont
  setfont} bind def
/s1 {/Symbol findfont 1.2 cm scalefont
  setfont} bind def
/s2 {/Symbol findfont 2.4 cm scalefont
  setfont} bind def
/s3 {/Symbol findfont 2.1 cm scalefont
  setfont} bind def
%%EndProlog
5 20 translate
newpath % triangle
0 0 moveto 3 sqrt cm 0 rlineto
0 1 cm rlineto closepath stroke

```

---

---

```

newpath % the braces
49 5 moveto
gsave
  0.5 1 scale s1 (\175) show
grestore
39 24 moveto
gsave
  120 rotate 0.18 1 scale s2 (\175) show
grestore
9 -1 moveto
gsave
  -90 rotate 0.25 1 scale s3 (\175) show
grestore
% the numbers
57 13 moveto mainfont (1) show
15 20 moveto mainfont (2) show
19 -18 moveto mainfont (\3263) show
showpage
%%EOF

```

---

Another example to illustrate that PostScript treats characters as path is the following outlining:




---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 166 35
/Helvetica-BoldOblique findfont
40 scalefont setfont
newpath
-2 3 moveto
(Outlined) true charpath
stroke
showpage
%%EOF

```

---

The actual path of the word *outlined* is appended to the current path by the `charpath` operator. It takes two arguments: a string and a boolean. Use the boolean value `true` if you want to stroke the outline and `false` if you intend to fill or clip.

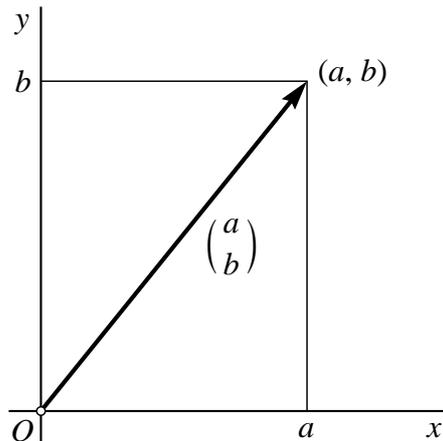
#### EXERCISE 28

Create the following picture, which consists of an outlined word fill with yellow.



#### EXERCISE 29

Create the following picture:



### 3.7.2 Importing L<sup>A</sup>T<sub>E</sub>X text

When it comes to complicated mathematical formulas and manual typesetting becomes a burden, it is useful to create Encapsulated PostScript files for the formulas with L<sup>A</sup>T<sub>E</sub>X and import these files when needed. The only drawback of this is the size of the PostScript file that is generated: in our example below, the file with the PostScript code of the formula is 14KB, whereas most of the PostScript examples in this book only take about 1KB. In Appendix 7 of the textbook *Mathematical Illustrations* of Bill Casselman [Cas05], the author gives a detailed description of how to import L<sup>A</sup>T<sub>E</sub>X text. Here we will only sketch the idea and give a working example.

#### EXERCISE 30

Consider the formula

$$\frac{2x^2 + 6x + 1}{x(x + 1)^2}$$

and do the following steps:

1. The first difficulty is to create via L<sup>A</sup>T<sub>E</sub>X an Encapsulated PostScript file that only contains the code for printing the formula and nothing else. The first thing to do is avoiding a page number. Create a L<sup>A</sup>T<sub>E</sub>X document, say `formula.tex` with the following contents:

```
\documentclass[12]{article}
\begin{document}
\pagestyle{empty}
\Huge $\frac{2x^2+6x+1}{x(x+1)^2}$
\end{document}
```

2. Typeset the document `formula.tex`. This will provide you with the `formula.dvi` file.
3. Create the Encapsulated PostScript file `formula.eps` by applying the command

```
dvips -E formula.dvi -o formula.eps
```

Preview the file to check that it is as expected.

4. Open the file `formula.eps` to check that the bounding box information is present. In our case, this is `%%BoundingBox: 149 621 240 668`. Without the `-E` option in the `dvips` command in the previous step, the bounding box would have been such that a full page with the formula in it were produced.
5. Create a PostScript file with the following contents.

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 200 50
%%BeginProlog
% define and assign the scale
% note:  1 cm = 28.34645 points,
%        1 inch = 72 points.
/s 28.34645 def
/BeginImport {
  save /SavedState exch def
  count /OpStackSize exch def
  /DictStackSize countdictstack def
  /showpage {} def
  0 setgray 0 setlinecap
  1 setlinewidth 0 setlinejoin
  10 setmiterlimit [] 0 setdash newpath
  /languagelevel where
  {pop languagelevel 1 ne
   {false setstrokeadjust false setoverprint} if
  } if
} bind def
/EndImport {
  count OpStackSize sub
  dup 0 gt { {pop} repeat } {pop} ifelse
  countdictstack DictStackSize sub
  dup 0 gt { {end} repeat } {pop} ifelse
  SavedState restore
} bind def
%%EndProlog
s s scale
1 s div setlinewidth
newpath
0.07 0.07 moveto
gsave
  1 s div 1 s div scale
  /TimesRoman findfont 20 scalefont setfont
  0 20 rmoveto (The formula:) show
  currentpoint pop 0 translate
  -149 -621 translate
  BeginImport
  (formula.eps) run

```

```

EndImport
stroke
grestore
showpage
%%EOF

```

The `BeginImport` and `EndImport` are taken literally from [Cas05]. `BeginImport` takes care of turning off a possible `showpage` in the imported file and sets up a default graphics state. `EndImport` restore the graphics state to what it was just before importing the Encapsulated PostScript file. The line `-149 -621 translate` in the above code sets the lower left corner of the imported image at the origin of the figure it is being imported to. This origin has been translated in the line before in the horizontal direction such that the imported formula will appear just after the introductory text. The `run` operator executes the contents of the specified Encapsulated PostScript file as if the command in it were written in the file to which the figure is imported. When the `run` operator encounters `%%EOF` (end-of-file), it closed the file.

6. Preview the PostScript document that you created in the previous step. It should look like

The formula: 
$$\frac{2x^2+6x+1}{x(x+1)^2}$$

### EXERCISE 31

Create a PostScript document in which you import the formula

$$f(x) = \frac{e^x}{1 + e^{2x}},$$

the PostScript code of which is created via L<sup>A</sup>T<sub>E</sub>X.

### 3.8 Symbol Encoding

The following table lists the character encoding for the Symbol font.

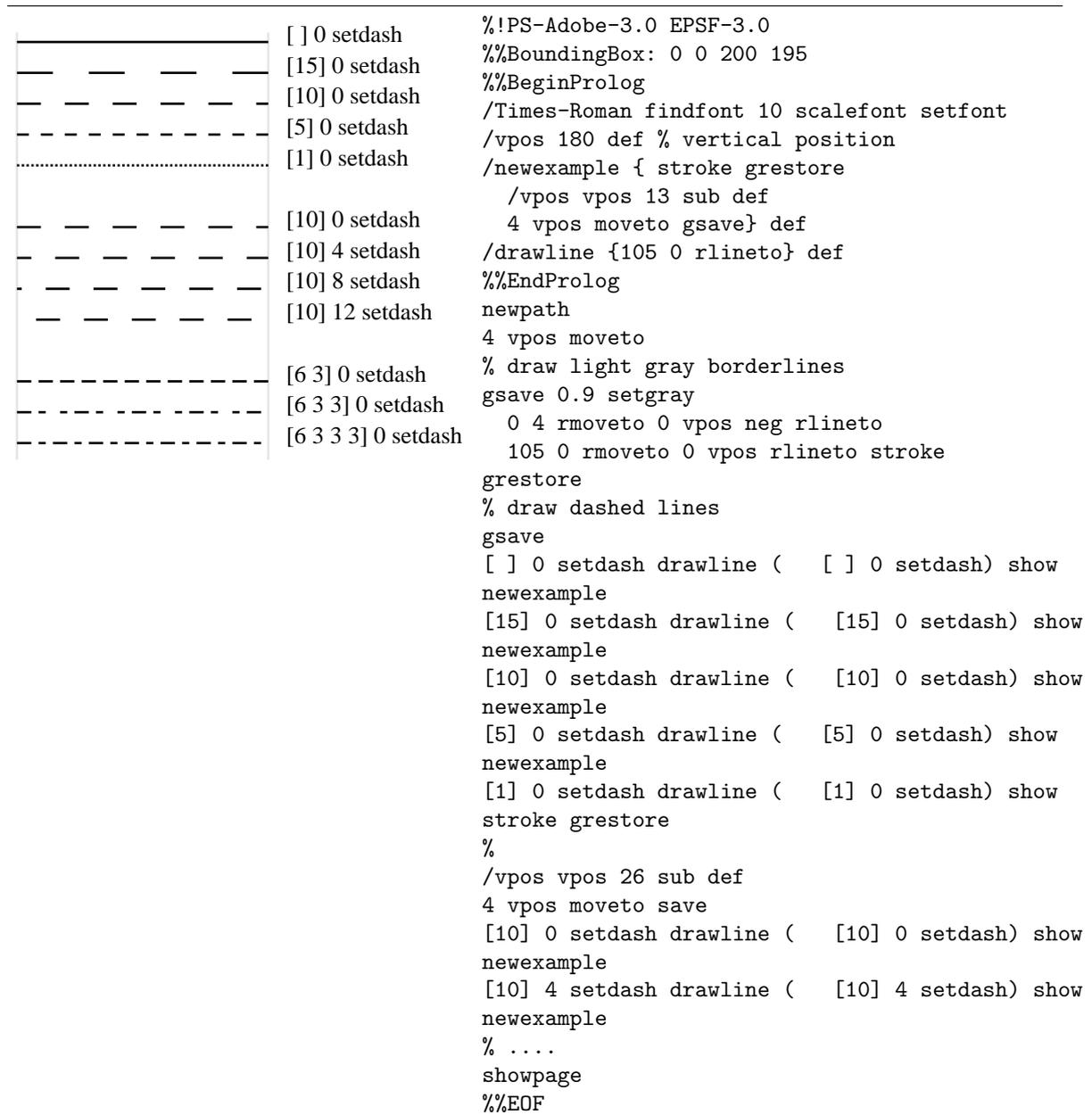
<i>octal</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>
\00x								
\01x								
\02x								
\03x								
\04x		!	∇	#	∃	%	&	ə
\05x	(	)	*	+	,	-	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	≅	A	B	X	Δ	E	Φ	Γ
\11x	H	I	∅	K	Λ	M	N	O
\12x	Π	Θ	P	Σ	T	Υ	ς	Ω
\13x	Ε	Ψ	Z	[	∴	]	⊥	—
\14x	—	α	β	χ	δ	ε	φ	γ
\15x	η	ι	φ	κ	λ	μ	ν	ο
\16x	π	θ	ρ	σ	τ	υ	ϖ	ω
\17x	ξ	ψ	ζ	{		}	~	
\20x								
\21x								
\22x								
\23x								
\24x	€	Υ	'	≤	/	∞	f	♣
\25x	♦	♥	♠	↔	←	↑	→	↓
\26x	◦	±	”	≥	×	∞	∂	•
\27x	÷	≠	≡	≈	...		—	↙
\30x	ℵ	ℑ	℔	℘	⊗	⊕	∅	∩
\31x	∪	⊃	⊇	∩	⊂	⊆	∈	∉
\32x	∠	∇	®	©	™	∏	√	·
\33x	¬	∧	∨	↔	⇐	↑↑	⇒	↓↓
\34x	◇	⟨	®	©	™	Σ	∫	
\35x	⌊	⌈		⌊	⌈	{	⌊	
\36x		⟩	⌋	⌋		⌋	⌋	
\37x	⌋	⌈		⌋	⌈	⌋	⌋	

## 4 Style Directives

In this chapter we explain how you can alter the appearance of graphics primitives, e.g., allowing certain lines to be thicker and others to be dashed, using different colors, and shading the interior of a closed path.

### 4.1 Dashing

Examples show you best how to specify a dash pattern when drawing a line or curve. We do not show the complete code of producing the figure to the left, because it is clear from the given examples and the labels added in the picture how the code looks like.



In general, the syntax for dashing is

```
[ dash pattern ] offset setdash
```

The dash pattern is a sequence of nonnegative numbers and not all of them may be zero. An empty dash pattern denotes solid, unbroken lines. In a nonempty dash pattern, the numbers represent the lengths of alternating dashes and gaps that make up the stroked line. The **stroke** operator uses the elements of the dash pattern cyclicly, i.e., when it reaches the end of the dash pattern, it continues from the beginning. Where the **stroke** operator is to start when it prints the line is determined by the offset.

## 4.2 Coloring

In PostScript, colors can be described in any of a variety of color models, including grayscale, RGB (red-green-blue), HSB (hue-saturation-brightness), and CMYK (cyan-magenta-yellow-black). The operators for color specifications in the listed models are **setgray**, **setrgbcolor**, **sethsbcolor**, and **setcmykcolor**, respectively. Below we will only discuss the gray and RGB color model.

In the gray model, a gray level is specified by a real number in the range 0 to 1, with 0 denoting black and 1 denoting white. So, the statement

```
.75294118 setgray
```

sets the current color to a particular kind of gray.

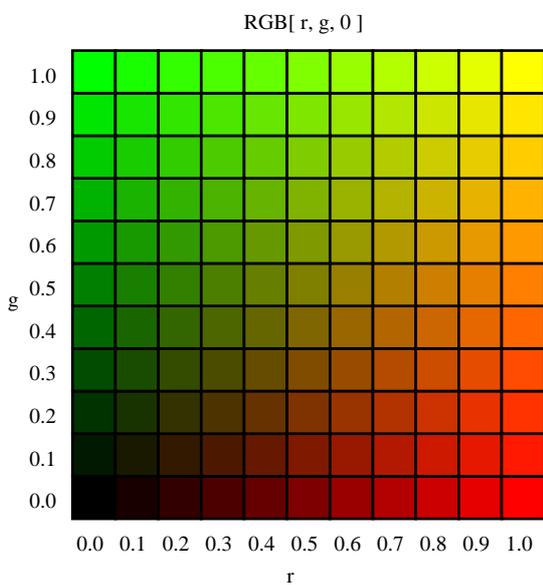
In the RGB model, a color is represented as a triple  $(r, g, b)$ . Each of  $r$ ,  $g$ , and  $b$  must be a number between 0 and 1, inclusively, representing fractional intensity of red, green, or blue, respectively. The statement

```
1 0 0 setrgbcolor
```

sets the current color to red. The following color table lists some color encodings.

<b>Color</b>	<b>RGB specification</b>		
<i>Name</i>	<i>red</i>	<i>green</i>	<i>blue</i>
black	0	0	0
blue	0	0	1
brown	0.647	0.165	0.165
cyan	0	1	1
green	0	1	0
magenta	1	0	1
orange	0.8	0.196	0.196
red	1	0	0
white	1	1	1
yellow	1	1	0

Let us draw one color chart:




---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 225 225
%%BeginProlog
/u 16 def % 16 points per unit
/lw 1 u div def % default linewidth
/Times-Roman findfont 8 u div scalefont setfont
/display {dup 3 1 roll 10 exp mul round
          10 3 -1 roll exp div} def
/str 10 string def
/n 10 def
/dp 2 def % display precision
%%EndProlog
u dup scale
lw setlinewidth
1.8 1.8 translate
%
0 1 n {
  /i exch def
  % draw horizontal labels
  i 0.1 add -0.75 moveto
  i n div dp display str cvs show
  % draw vertical label
  -1 i 0.25 add moveto
  i n div dp display str cvs show
  0 1 n {
    /j exch def
    gsave % draw filled rectangle
      i n div j n div 0 setrgbcolor
      newpath i j 1 1 rectfill
      0 setgray
      newpath i j 1 1 rectstroke
    grestore
  } for
} for
n 2 div -1.5 moveto (r) show
-1.5 n 2 div moveto (g) show
n 2 div 1 sub n 1.5 add moveto
(RGB[ r, g, 0 ]) show
showpage
%%EOF

```

---

We want to mention two things about the above PostScript code. Firstly, PostScript has no built-in methods to control the number of displayed decimal places of a floating-point number. This is why we define our own `display` routine, which takes a real number and the number of decimal places that are shown, with removal of trailing zeros. Secondly, we use a

repetition control structure, viz., a `for` loop. We will come to control structures of imperative programming in Chapter 5. But to understand the above PostScript code, it suffices to know that the following basic structure of the counted `for` loop

```

start stepsize finish {
  /counter exch def
  ...
} for

```

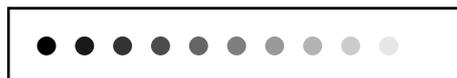
means that a `counter` with initial value `start` is incremented at the end of each step in the repetition by the value of `stepsize` until it passes the value of `finish`. Then the repetition stops and the PostScript interpreter continues with what comes after the `for` part. Actually, the `for` loop involves a hidden and nameless variable with initial value `start` whose value is put on the stack just before each step in the repetition. The statement of the form `/counter exch def` takes this hidden variable off the stack and assigns it to the variable `counter`. In the above PostScript code we have a nested counted `for` loop

#### EXERCISE 32

1. Verify that the above `display` procedure indeed displays a given real number in the specified number of decimal places (Hint: write down what the stack looks like at each step of the computation).
2. Change the integer `n` from 10 into 4 and verify that the program still works correctly.

#### EXERCISE 33

Create the following graylevel chart:



#### EXERCISE 34

Using color charts, compare the linear conversion from color to gray, defined by the function

$$(r, g, b) \mapsto \frac{(r + g + b)}{3} \times (1, 1, 1)$$

with the following conversion formula used in black and white television:

$$(r, g, b) \mapsto (0.30r + 0.59g + 0.11b) \times (1, 1, 1).$$

### 4.3 Joining Lines

In PostScript, lines are joined by default such that line joints are normally rounded. You can influence the appearances of the lines by the operators `setlinejoin` and `setlinecap`, which give the internal variables `linejoin` and `linecap` a value 0, 1 or 2, respectively. The default values of these variables is 0. The pictures below show the possibilities.

---

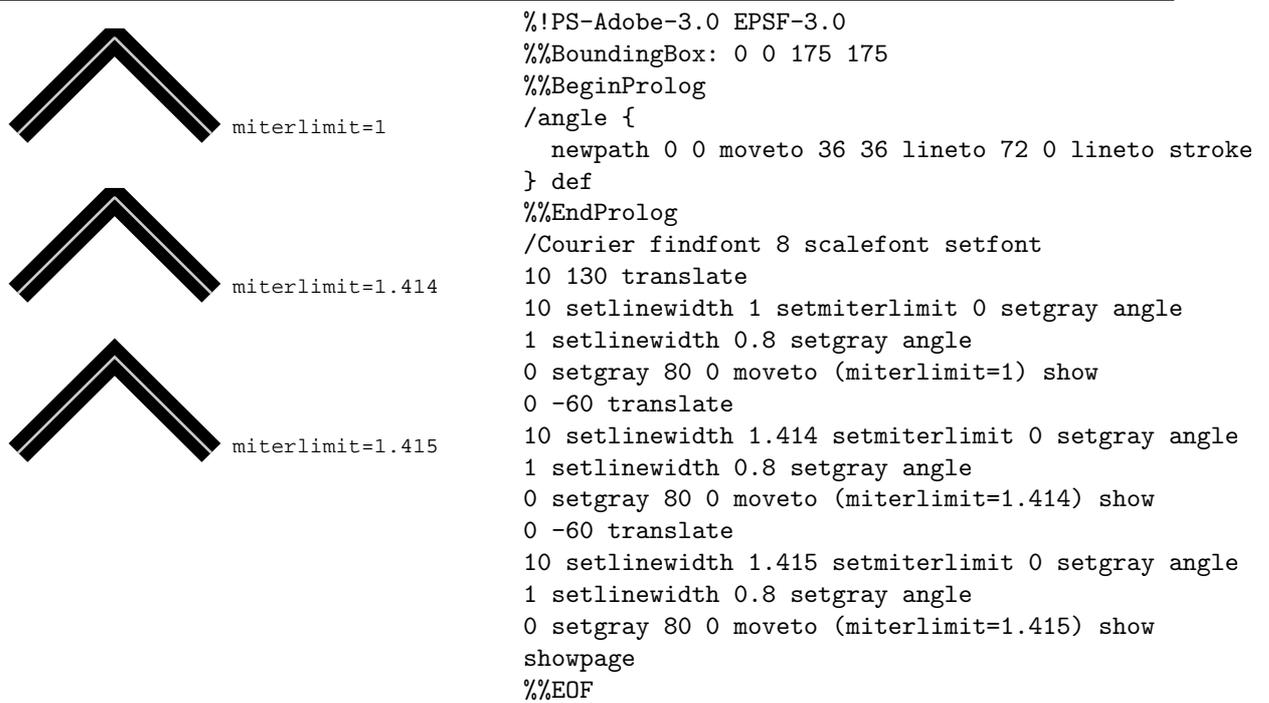
	butt: linecap=0	<pre> %!PS-Adobe-3.0 EPSF-3.0 %%BoundingBox: 0 0 180 60 %%BeginProlog /line {newpath 0 0 moveto 72 0 lineto stroke} def %%EndProlog /Courier findfont 8 scalefont setfont 10 50 translate 10 setlinewidth 0 setlinecap 0 setgray line 1 setlinewidth 1 setgray line 0 setgray 82 -2 moveto (butt: linecap=0) show  0 -20 translate 10 setlinewidth 1 setlinecap 0 setgray line 1 setlinewidth 1 setgray line 0 setgray 82 -2 moveto (rounded: linecap=1) show  0 -20 translate 10 setlinewidth 2 setlinecap 0 setgray line 1 setlinewidth 1 setgray line 0 setgray 82 -2 moveto (squared: linecap=2) show showpage %%EOF </pre>
	rounded: linecap=1	
	squared: linecap=2	

---

	mitered: linejoin=0	<pre> %!PS-Adobe-3.0 EPSF-3.0 %%BoundingBox: 0 0 165 185 %%BeginProlog /angle {   newpath 0 0 moveto 36 36 lineto 36 0 lineto stroke } def %%EndProlog /Courier findfont 8 scalefont setfont 10 130 translate 10 setlinewidth 0 setlinejoin 0 setgray angle 1 setlinewidth 0.8 setgray angle 0 setgray 56 0 moveto (beveled: linejoin=0) show  0 -60 translate 10 setlinewidth 1 setlinejoin 0 setgray angle 1 setlinewidth 0.8 setgray angle 0 setgray 56 0 moveto (rounded: linejoin=1) show  0 -60 translate 10 setlinewidth 2 setlinejoin 0 setgray angle 1 setlinewidth 0.8 setgray angle 0 setgray 56 0 moveto (mitered: linejoin=2) show showpage %%EOF </pre>
	rounded: linejoin=1	
	beveled: linejoin=2	

---

By setting the variable `miterlimit`, you can influence the mitering of joints. The next example demonstrates that the value of this variable, a number greater than or equal to 0, acts as a trigger.



At any given corner, the *miter length* is the distance from the point at which the inner edges of the strokes intersect to the point at which their outer edges intersect (see Figure 4). This distance depends on the angle between the segments. If the ratio of the miter length to the line width exceeds the specified miter limit, the `stroke` operator treats the corner as a beveled join instead of a mitered join.

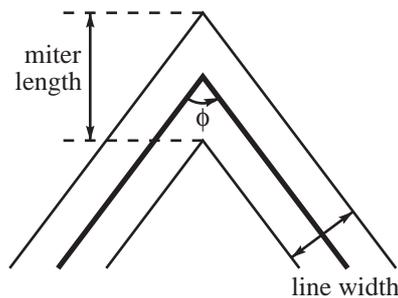


Figure 4: Mitter length.

The ratio of miter length to line width is directly related to the angle  $\phi$  between the segment by the following formula:

$$\frac{\text{miterlength}}{\text{linewidth}} = \frac{1}{\sin\left(\frac{\phi}{2}\right)}$$

Example miter limit values are:

- 1.414 cuts off miters at angles less than 90 degrees.
- 2.0 cuts off miters at angles less than 60 degrees.
- 10.0 cuts off miters at angles less than 11 degrees.
- 1.0 cuts off miters at any angle.

The default value of the miter limit is 10.0.

## 5 Control Structures

In this chapter we will look at two commonly used control structures of imperative programming languages: condition and repetition. Unlike common programming languages, the control constructs are specified by means of operators that take procedures as operands. By the way, a PostScript procedure is nothing else than an executable array, i.e., an array whose contents are to be executed by the PostScript interpreter. Let us exemplify the meaning of this. If a series of objects is enclosed in braces, it is not immediately executed, but is stored in an array and placed on the stack. Thus, the line `1 2 add` causes the interpreter to add the numbers and put the sum `3` on the stack, while the line `{1 2 add}` places the numbers and the `add` in an array, which is then placed on the stack. As we have seen already in many examples, an executable array is often preceded by a literal name (a name beginning with a slash `/`) and followed by a `def` operator, which associates it with a name in the current dictionary

### 5.1 Conditional Operations

In PostScript, there are two operators for building a conditional statement: `if` and `ifelse`. The simplest form of a conditional statement is this:

```
condition
{ true clause }
if
```

where *condition* represents an object that leaves a boolean value (i.e., true or false) on the stack. The `if` operator takes two operands: a boolean value and an executable array, denoted here as *true clause* because `if` causes this procedure to be executed in case the boolean value is equal to *true*. The next example of an `if` statement prints the stack at some step in the PostScript program provided that the `debug` variable has been given the value `true`.<sup>9</sup>

```
/debug true def % debugging on
% ... some code
debug {pstack} if
% ... some more code
```

The most general conditional statement in PostScript uses the `ifelse` operator and has the following form:

```
condition
{ true clause }
{ false clause }
ifelse
```

where *condition* represents a procedure that leaves a boolean value (i.e., true or false) on the stack. The `ifelse` operator is used for branching: depending on some condition, one procedure is executed or another. Nesting of conditional operations is allowed, but there is no shortcut notation. For example, nesting of two basic `ifelse` operations looks as follows:

---

<sup>9</sup>Use the GhostScript interpreter to see the debugging printout and the PostScript picture in different windows.

```

1st condition
{ 1st procedure }
{ 2nd condition
  { 2nd procedure }
  { 3rd procedure }
  ifelse
}
ifelse

```

**EXERCISE 35**

Start the GHOSTSCRIPT interpreter and enter the following statements:

```

/i -1 def
/j 1 def
i j gt {i}{j} ifelse

```

Which number do you expect on top of the stack? Verify your answer.

In the above exercise, the condition is a command sequence in PostScript that returns a boolean value. In the sample code, the sequence `i j gt` puts `true` on the stack if  $i > j$ ; otherwise, the boolean value `false` will be put on the stack. The command sequence that returns a boolean value and forms the condition can be built up with the following relational and logical operators.

Relational Operators		
<i>Arguments</i>	<i>Operator</i>	<i>Meaning</i>
<i>any</i> <sub>1</sub> <i>any</i> <sub>2</sub>	<code>eq</code>	test equal
<i>any</i> <sub>1</sub> <i>any</i> <sub>2</sub>	<code>ne</code>	test not equal
<i>num</i> <sub>1</sub> <i>num</i> <sub>2</sub>	<code>ge</code>	test greater than or equal (of numbers)
<i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub>	<code>ge</code>	test lexically greater than or equal (of strings)
<i>num</i> <sub>1</sub> <i>num</i> <sub>2</sub>	<code>gt</code>	test greater than (of numbers)
<i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub>	<code>gt</code>	test lexically greater than (of strings)
<i>num</i> <sub>1</sub> <i>num</i> <sub>2</sub>	<code>le</code>	test less than or equal (of numbers)
<i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub>	<code>le</code>	test lexically less than or equal (of strings)
<i>num</i> <sub>1</sub> <i>num</i> <sub>2</sub>	<code>lt</code>	test less than (of numbers)
<i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub>	<code>lt</code>	test lexically less than (of strings)

Logical Operators	
<i>Operator</i>	<i>Meaning</i>
<code>and</code>	test if all conditions hold
<code>not</code>	negation of condition
<code>or</code>	test if one of many conditions hold
<code>xor</code>	test exclusive or

These operators can be used to create your own test procedures. For example, the following procedure checks whether a given integer is even. If yes, `true` is put on the stack; if not, `false` is put on the stack.

```

/isEven {dup 2 idiv 2 mul eq} def

```

**EXERCISE 36**

Write a procedure that takes three number as arguments, considers them as lengths of sides of a triangle, and decides whether such a triangle can be constructed. If so, a sample triangle is drawn.

**5.2 Repetition**

In the construction of the color chart on page 45 you have already seen the counted `for` loop of the form

```

start stepsize finish {
  /counter exch def
  ...
} for

```

means that a *counter* with initial value *start* is incremented at the end of each step in the repetition by the value of *stepsize* until it passes the value of *finish*. Then the repetition stops and the PostScript interpreter continues with what comes after the `for` part. Actually, the `for` loop involves a hidden and nameless variable with initial value *start* whose value is put on the stack just before each step in the repetition. The statement of the form `/counter exch def` takes this hidden variable off the stack and assigns it to the variable *counter*. Instead of revealing the hidden variable in the `for`, you can also ignore it in all of the code of the loop text but the first line, in which you `pop` it off the stack. Nesting of counted `for` loops is of course possible, but there are no abbreviations like in other programming languages.

The counted `for` loop is an example of an *unconditional repetition*, in which a predetermined set of actions are carried out. The simplest repetition structure of this type is however the `repeat` loop of the the form

```

count {
  ...
} repeat

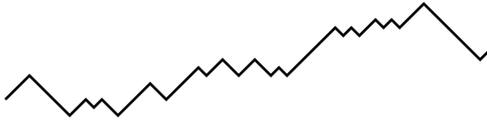
```

Here *count* is an integer. The lines of the procedure are repeated *count* times.

Below, we will give some examples of a counted `for` loop and a `repeat` loop.

**A Bernoulli walk**

A 1-dimensional Bernoulli walk is a random walk in which a person takes at regular time intervals randomly a step to the left or to the right. Each step is assumed to be of the same length and to be stochastically independent of the previous one. In the following diagram, the position of the random walker (in number of steps to the right (positive value) or to the left (negative value)) is plotted against the number of steps made since he or she left the origin. We define the `randstep` operator to generate randomly a number 1 or -1. It uses the `rand` operator that returns a random integer in the range 0 to  $2^{31} - 1$ , produced by a pseudo-random number generator. Applying the modular arithmetic with modulus 2, we get a random number 0 or 1. Multiplying this number by 2 and subtracting one convert the number to -1 or 1.

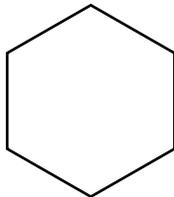


```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 190 190
%%BeginProlog
/u 3 def % 3 points per unit
/lw 1 u div def
/oneOrZero 2 rand 2 mod mul 1 sub def
/n 60 def
%%EndProlog
u dup scale
lw setlinewidth
1 30 translate
%
newpath
0 0 moveto
n { 1 oneOrZero rlineto} repeat
stroke
showpage
%%EOF
```

---

### A Regular Polygon

Regular polygons are easily drawn via a `repeat` loop. In the program below we assume that one of the vertices of the  $n$ -gon is  $(0, 1)$ , which implies that we can take a starting angle  $\alpha_0$  of 90 degrees. If we set  $\alpha = 360/n$ , then the other vertices will be  $(\cos(\alpha_0 + \alpha), \sin(\alpha_0 + \alpha))$ ,  $(\cos(\alpha_0 + 2\alpha), \sin(\alpha_0 + 2\alpha))$ , and so on.



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 76 76
%%BeginProlog
/u 36 def % unit: 0.5 inch
/lw 1 u div def
/n 6 def
/da 360 n div def % angle increment
/a 90 def % initial angle
%%EndProlog
u dup scale
lw setlinewidth
1.05 1.05 translate
newpath
0 1 moveto
n 1 sub {/a a da add def % increase angle
  a cos a sin lineto % draw line
} repeat
closepath
stroke
showpage
%%EOF
```

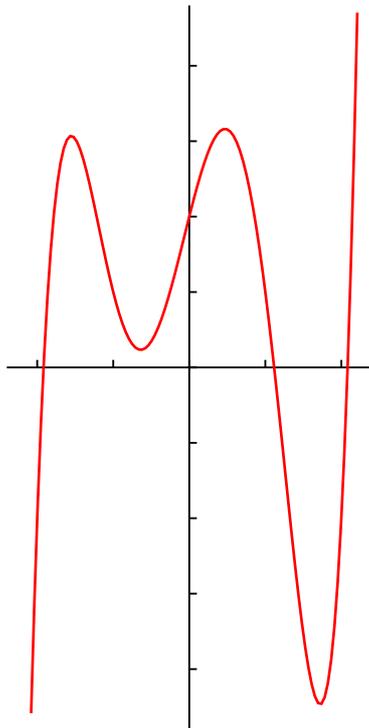
---

## The graph of a function

In this example we plot the graph of the function

$$f(x) = x^5 - 5x^3 - x^2 + 4x + 2$$

on the interval  $(-2.4, 2.4)$ ; we restrict the vertical range to the interval  $(-4.8, 4.8)$ . First we give a simple implementation that illustrates the idea of drawing a function. The graph of the function is drawn in red and tick marks on the axes indicate the unit scaling.



```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 75 150
%%BeginProlog
/u 14.173225 def % unit: 0.5 cm
/lw1 1 u div def
/lw2 0.7 u div def
/f { % the function f(x)
  1 dict begin
  /x exch def
  x 5 exp      % x^5
  x 3 exp -5 mul add % -5x^3
  x 2 exp sub  % -x^2
  x 4 mul add  % +4x
  2 add       % +2
  end } bind def
/n 100 def
%%EndProlog
u dup scale
lw1 setlinewidth
2 setlinejoin % beveled joins
% translate origin
2.5 4.9 translate
% draw axes
lw2 setlinewidth
newpath
-2.4 0 moveto 2.4 0 lineto
0 -4.8 moveto 0 4.8 lineto
stroke
% draw ticks
newpath
-2 1 2 {/i exch def
  i 0 moveto 0 0.1 rlineto
} for
-4 1 4 {/i exch def
  0 i moveto 0.1 0 rlineto
} for
stroke
% draw graph
lw1 setlinewidth
1 0 0 setrgbcolor
newpath
```

---

```

/x -2.08 def
/dx 4.29 n div def
x x f moveto
n {
/x x dx add def
x x f lineto
} repeat
stroke
showpage
%%EOF

```

---

Let us comment on the above code: In the definition of the function  $f(x)$  we make the variable  $x$  local to the procedure via the `1 dict begin ... end` construct. Furthermore, we use the `bind` for efficiency reasons. The comments in the PostScript code makes it for the reader more easy to verify that the intended function is indeed defined.

The heart of the code is the last part:

```

/x -2.08 def
/dx 4.29 n div def
  x x f moveto
  n {
    /x x dx add def
    x x f lineto
  } repeat

```

First we choose the smallest  $x$  value (-2.08) and the length (4.29) of the domain that we will actually use for drawing the graph. The numbers are chosen such that the graph fits in the vertical range  $-4.8 \dots 4.8$ . If  $n$  is the number straight line segments that are used to graph the function  $f$ , then the step size will be  $length\_of\_domain / n$ . After moving to the starting point  $(-2.08, f(-2.08))$ , we cross in  $n$  steps to the end point  $(2.21, f(2.21))$ . At each step in the `repeat` loop, we draw a straight line from the current point  $(x, f(x))$  to the next point  $(x + dx, f(x + dx))$ . If the number of segments ( $n$ ) is large enough, the graph of the function will look smooth and you will not really see that it is actually built up from small straight line segments. We use beveled joins (`2 setlinejoin`) to make breaks between segments less sharp.

In fact, the graph of a function is a *polygonal approximation* to it. In practice, the graphical result will be often quite acceptable provided that you can choose a sufficiently large number of segments. Remaining drawback is that the picture created in this way is not scalable: if the approximation looks smooth at one scale, it may not look good at another. The only remedy to this is the use of curve segments that join smoothly. The *cubic Bézier approximation* of the graph of a function is a good solution, but as we will see, it only works if you know the derivative of your function or can approximate it well enough. The only change that we basically make to PostScript code is that we draw cubic Bézier segments between the points that we use to build up the graph. The control points are determined as follows: suppose that

$$P_0 = (x, f(x))$$

and

$$P_3 = (x + dx, f(x + dx))$$

are the end points of our small Bézier curve. Then set the post and pre control points as

$$P_1 = \left(x + \frac{1}{3} dx, f(x) + \frac{1}{3} f'(x) dx\right)$$

and

$$P_2 = \left(x + \frac{2}{3} dx, f(x + dx) - \frac{1}{3} f'(x + dx) dx\right)$$

so that the direction of the cubic Bézier curve with end points  $P_0$  and  $P_3$ , post control point  $P_1$ , and pre control point  $P_2$  equals the direction of the graph of the function  $f$  at the end points. All Bézier segments together build up a smooth approximation of the graph of our function.

The changes to the above PostScript code that will produce a cubic Bézier approximation of the graph of the function

$$f(x) = x^5 - 5x^3 - x^2 + 4x + 2$$

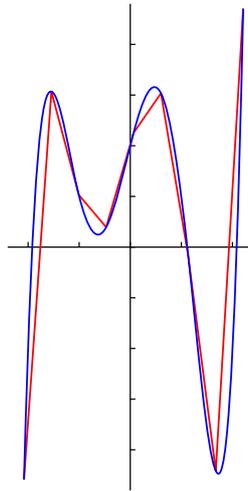
are: the introduction of the derivative of  $f$

```
/f' { % the function f'(x)
  1 dict begin
  /x exch def
  x 4 exp 5 mul      % 5x^4
  x 2 exp -15 mul add % -15x^2
  x 2 mul sub        % -2x
  4 add              % +4
end } bind def
```

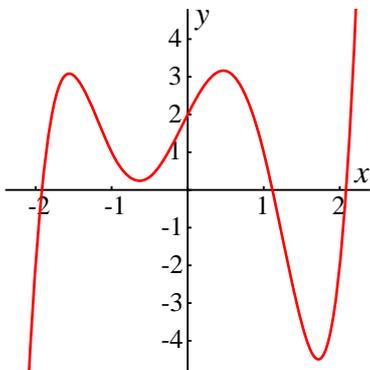
and the change in the repeat loop to cubic Bézier segments

```
x x f moveto % P0
/f'x x f' def
n {
  x dx 3 div add
  x f dx 3 div f'x mul add % P1
  /x x dx add def
  /f'x x f' def
  x dx 3 div sub
  x f dx 3 div f'x mul sub % P2
  x x f % P3
  curveto
} repeat
```

To see the difference between the polygon approximation and the cubic Bézier approximation, we choose a small number of segment, viz.,  $n = 8$ , and draw both approximations in one picture. Believe it or not, but the cubic Bézier approximation shown in the picture below can hardly be distinguished from the real graph on the given domain, range, and scaling.



The graph of the function  $f(x) = x^5 - 5x^3 - x^2 + 4x + 2$  can be produced in a more general way. Below we will give an idea how to do this. We deliberately choose different scaling of the axes. We also put much effort in automatic placing of the labels near the tick marks and in making the code working properly for any horizontal and vertical ranges that contain 0, provided that the bounding box of the picture is chose large enough to contain the diagram. This complicates the code, but makes it more general. In the code listing below, we repeatedly show the picture that is created, so that you can compare PostScript code with the result. Of course, the PostScript code will generate only one picture.



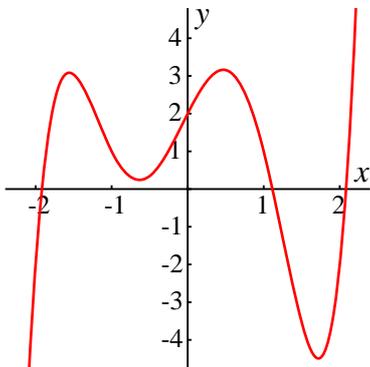

---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 150 150
%%BeginProlog
/u 14.173225 def % unit: 0.5 cm
/lw1 1 u div def
/lw2 0.7 u div def
/f1 {
  /Times-Roman findfont 10 u div scalefont setfont
} bind def
/f2 {
  /Times-Italic findfont 12 u div scalefont setfont
} bind def
/str 2 string def
/sx 0.5 def % scale factors for x, y
/sy 1 def
/f { % the function f(x)
  1 dict begin
  /x exch def
  x 5 exp          % x^5
  x 3 exp -5 mul add % -5x^3
  x 2 exp sub      % -x^2
  x 4 mul add      % +4x
  2 add            % +2
  end
} bind def

```

---



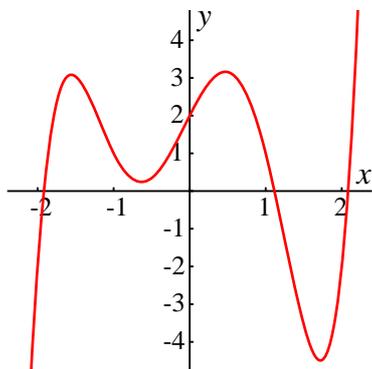

---

```

/f' {1 dict begin % the function f'(x)
  /x exch def
  x 4 exp 5 mul      % 5x^4
  x 2 exp -15 mul add % -15x^2
  x 2 mul sub        % -2x
  4 add end          % +4
} bind def
/scoord {% scales coordinates
  % stack:      x y
  exch sx div % y x/sx
  exch sy div % x/sx y/sy
} bind def
/x0 -2.4 def % set x range that
/x1 2.4 def  % contains 0
/y0 -4.8 def % set yrange that
/y1 4.8 def  % contains 0
/n 10 def
%%EndProlog
u dup scale lw1 setlinewidth
% translate origin
x0 sx div neg 0.5 add
y0 sy div neg 0.5 add translate
% draw axes
lw2 setlinewidth
newpath
x0 0 scoord moveto x1 0 scoord lineto
0 y0 scoord moveto 0 y1 scoord lineto
stroke
x1 sx div 0.4 sub 0.2 moveto f2 (x) show
0.1 sx div y1 sy div 0.4 sub moveto f2 (y) show
% draw ticks
newpath
/start x0 ceiling cvi def
/finish x1 floor cvi def
start 1 finish {/i exch def
  i 0 scoord moveto 0 0.1 rlineto
  i 0 ne {-0.2 -0.75 rmoveto f1 i str cvs show} if
} for
/start y0 ceiling cvi def
/finish y1 floor cvi def
start 1 finish {/i exch def
  0 i scoord moveto 0.1 0 rlineto
  i 0 lt {-0.8 -0.2 rmoveto f1 i str cvs show} if
  i 0 gt {-0.6 -0.2 rmoveto f1 i str cvs show} if
} for
stroke
% restrict drawing area
x0 y0 scoord moveto x1 y0 scoord lineto
x1 y1 scoord lineto x0 y1 scoord lineto
closepath clip
% draw graph

```

---




---

```

lw1 setlinewidth 1 0 0 setrgbcolor
/x x0 def
/dx x1 x0 sub n div def
newpath
x x f scoord moveto % P0
/f'x x f' def
n {
  x dx 3 div add
  x f dx 3 div f'x mul add scoord % P1
  /x x dx add def
  x dx 3 div sub
  /f'x x f' def
  x f dx 3 div f'x mul sub scoord % P2
  x x f scoord % P3
  curveto
} repeat
stroke
showpage
%%EOF

```

---

The really new things in the above PostScript code are:

1. Variables are introduced for the scale factors (`sx`, `sy`), and for the start and end of axes (`x0`, `x1`, `y0`, `y1`). Just by changing the values of these variable you can select a new domain or range, or choose a different scaling.
2. The diagram is clipped to the rectangular area with lower left corner (`x0`, `y0`) and upper right corner (`x1`, `y1`). In this way we take care of function values outside our vertical range.
3. The origin of the coordinate system is translated such that the drawing area will be placed in the lower left corner.
4. Scaling is done via the `scoord` operator. This has the advantage that we can define points with respect to the normal coordinate system and hereafter transform them to positions in the scaled coordinate system.<sup>10</sup>
5. Tick marks and text labels are automatically placed in the diagram with respect to the given domain and range. String conversion via the `cvs` operator is used to create a string representation of a given integer. For this purpose, the auxiliary variable `str` is defined to contain words consisting of at most two symbols. Drawback of automatic labeling is that the graph may go through a label. Therefore, an ad hoc style of labeling is not a bad idea.

### EXERCISE 37

Verify that the above code keeps generating nice diagrams when the horizontal and vertical ranges are changed and when the scaling is changed (provided that the bounding box is large enough to contain the complete picture).

Especially through the last example of drawing the graph of a mathematical function we have drifted away from our topic: the control structure called *repetition*. There are still more forms of repetition in PostScript to discuss.

<sup>10</sup>By the way, use of the `scale` operator with different horizontal and vertical scaling has the disadvantage that it also changes the line width in a non-uniform way.

Another popular type of repetition is the *conditional loop*. PostScript does not have a pre- or post conditional loop (in many imperative programming language called a `while` loop and `until` loop, respectively) built in. You must create one by an endless loop and an explicit jump outside this loop. First the endless loop: this is created by

```
{ loop text } loop
```

To terminate such a loop when a boolean condition becomes true, use the `exit` operator:<sup>11</sup>

```
condition {exit} if
```

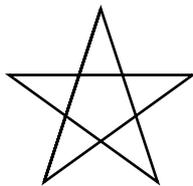
When the boolean expression evaluates to `true`, then the PostScript interpreter encounters the `exit` operator and exits the current loop. One thing you must keep in mind is that any objects pushed on the stack during execution of the loop remain after the loop is exited. The PostScript version of a `until` loop is

```
{ loop text
  condition {exit} if
} loop
```

If it is more convenient to exit the loop when an expression becomes false, then the test must appear at the beginning of the loop. Thus, the PostScript version of a `while` loop is:

```
{ condition {exit} if
  loop text
} loop
```

Below, we use the `loop` operator to create a star shaped figure. In the program we assume that one of the vertices of the  $n$ -star is  $(\cos \alpha_0, \sin \alpha_0)$ , where  $\alpha_0$  is the starting angle in degrees. If we set  $\alpha = 720/n$ , then the other vertices will be  $(\cos(\alpha_0 + \alpha), \sin(\alpha_0 + \alpha))$ ,  $(\cos(\alpha_0 + 2\alpha), \sin(\alpha_0 + 2\alpha))$ , and so on.  $\alpha_0$  and  $n$  can be changed to get a different shape.




---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 76 76
%%BeginProlog
/u 36 def % unit: 0.5 inch
/n 5 def % choose an odd number
/da 720 n div def % angle increment
/a0 90 def % initial angle
%%EndProlog
u dup scale 1 u div setlinewidth
1.05 1.05 translate newpath
a0 cos a0 sin moveto /a a0 def
n {a 720 a0 add gt {exit} if % test
  /a a da add def % increase angle
  a cos a sin lineto % draw line
} loop
closepath stroke
showpage
%%EOF

```

---

<sup>11</sup>An alternative operator for leaving a loop is the `stop` operator, which works in this context the same as the `exit` operator.

Another application of the conditional loop You can combine a counted for loop and a conditional loop. For example,

```

start stepsize finish {
  /counter exch def
  condition {exit} if
  ...
} for

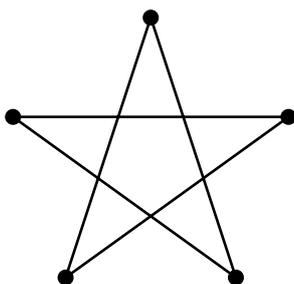
```

is the PostScript equivalent of a combination of a *for* and *while* loop.

The fourth kind of repetition, viz., the *forall* operator, only works for arrays. Recall that an array is denoted in PostScript by a sequence of objects placed between square brackets. We used implemented the graphical primitive *point* as an array of two numbers  $[x, y]$ . A *polygon* is a sequence of points  $P_0, P_1, \dots, P_{n-1}$ , called its *vertices*. It is natural to implement a polygon as an array of points  $[P_0, P_1, \dots, P_{n-1}]$ .

### EXERCISE 38

Below we draw a polygon and its vertices.



```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 120 112
%%BeginProlog
/u 54 def % unit: 0.5 inch
/lw 1 u div def
/point {2 lw mul 0 360 arc
  gsave 0 setgray fill grestore stroke
} bind def
/makepolygon {2 dict begin
  /vertices exch def
  /n vertices length def
  n 1 gt {
    vertices 0 get aload pop moveto
    vertices 1 n 1 sub getinterval
    { aload pop lineto} forall
  } if
end} bind def
/drawvertices {
  {aload pop point newpath} forall
} bind def
/vertices [ [0 1] [234 cos 234 sin]
  [18 cos 18 sin] [162 cos 162 sin]
  [306 cos 306 sin] ] def
%%EndProlog
u dup scale 1.1 0.95 translate
lw setlinewidth 2 setlinejoin
newpath vertices makepolygon closepath stroke
vertices drawvertices
%%EOF

```

Try to understand the above PostScript code using the following hints about arrays in PostScript :

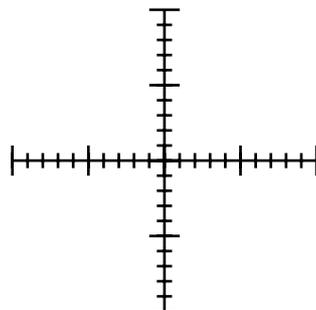
- The numbering of elements of arrays starts at 0.
- If `a` is an array, then `a length` returns the number of elements of the array.
- If `a` is an array, then `a i get` returns the *i*th element on the stack.
- You create an array on the stack by entering `[, a few items, then ]`.
- If `a` is an array, then `a aload`, pushes all elements of the array on the stack plus the array itself. For example, after `[1 2] aload` the top of the stack looks like `1 2 [1 2]`.
- You create a *subarray* of a given array `a` via the `getinterval` operator. Besides the array, the `getinterval` needs as argument the index to start at and the number of items to take. For example, `[0 2 4 6 8] 1 3 getinterval` returns `[2 4 6]` on the stack.
- If `a` is an array, then `a { ... } forall` executes the procedure `{ ... }` for each element of the array `a`.

The following table summarizes the various forms of control operators in PostScript .

Control Operators		
<i>Operator</i>	<i>Calling Sequence</i>	<i>Meaning</i>
<code>for</code>	<code>start increment finish { proc } for</code>	execute <i>proc</i> with a counter running from <i>start</i> to <i>finish</i> by steps of <i>increment</i>
<code>forall</code>	<code>array { proc } forall</code>	execute <i>proc</i> for each element of <i>array</i>
<code>if</code>	<code>bool { proc } if</code>	execute <i>proc</i> if <i>bool</i> is <i>true</i>
<code>ifelse</code>	<code>bool { proc<sub>1</sub> } { proc<sub>2</sub> } ifelse</code>	execute <i>proc<sub>1</sub></i> if <i>bool</i> is <i>true</i> , <i>proc<sub>2</sub></i> if <i>false</i>
<code>loop</code>	<code>{ proc } loop</code>	execute <i>proc</i> an indefinite number of times
<code>repeat</code>	<code>count { proc } repeat</code>	execute <i>proc</i> <i>count</i> times

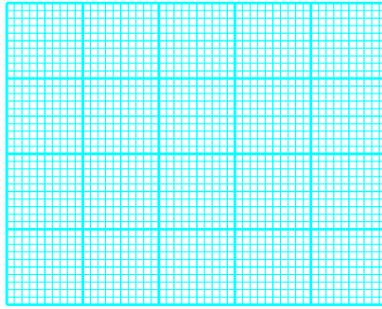
#### EXERCISE 39

Create the following coordinate system:



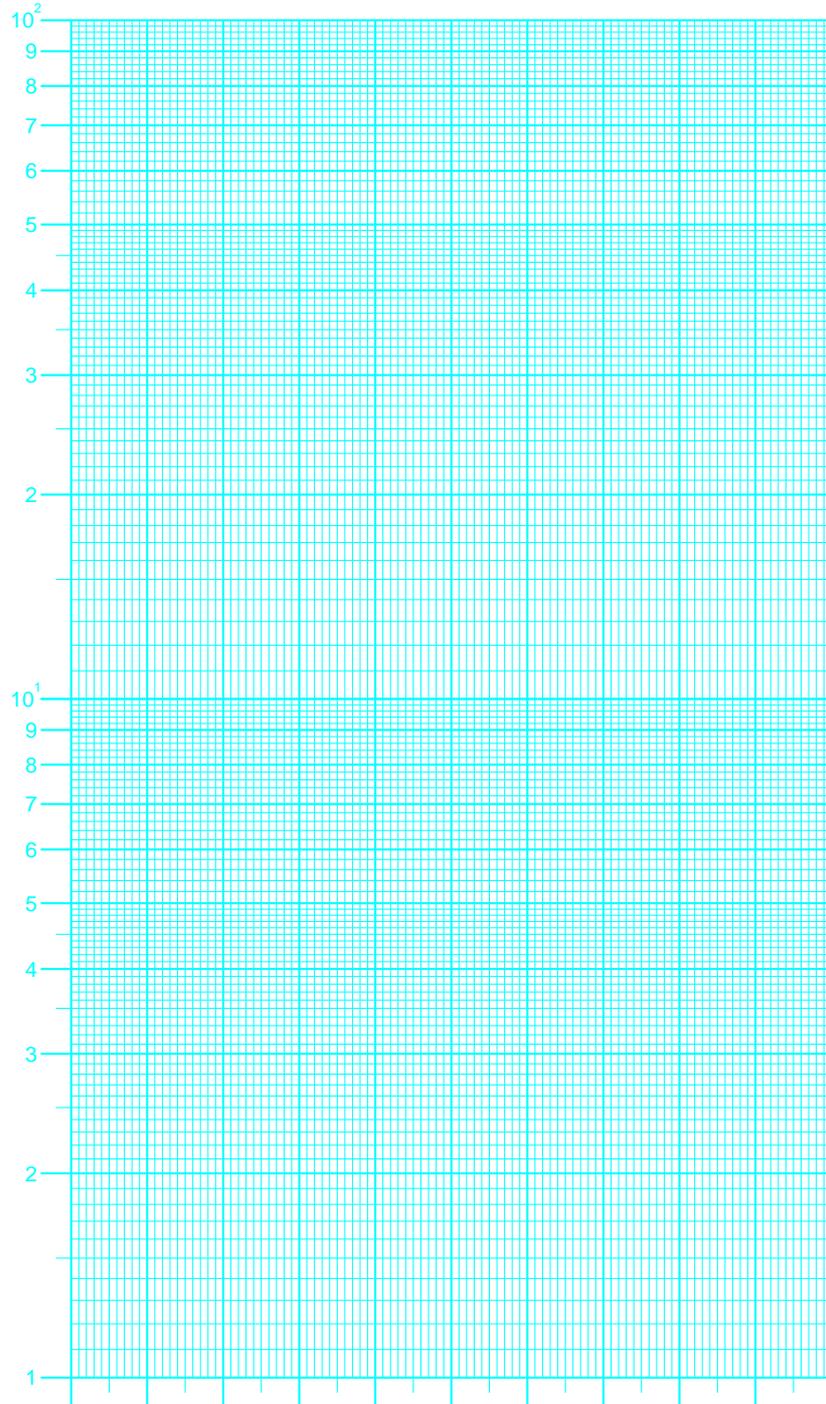
#### EXERCISE 40

Create the following piece of millimeter paper. Write your code such that you can easily generate graph paper of any number of horizontal and vertical units.



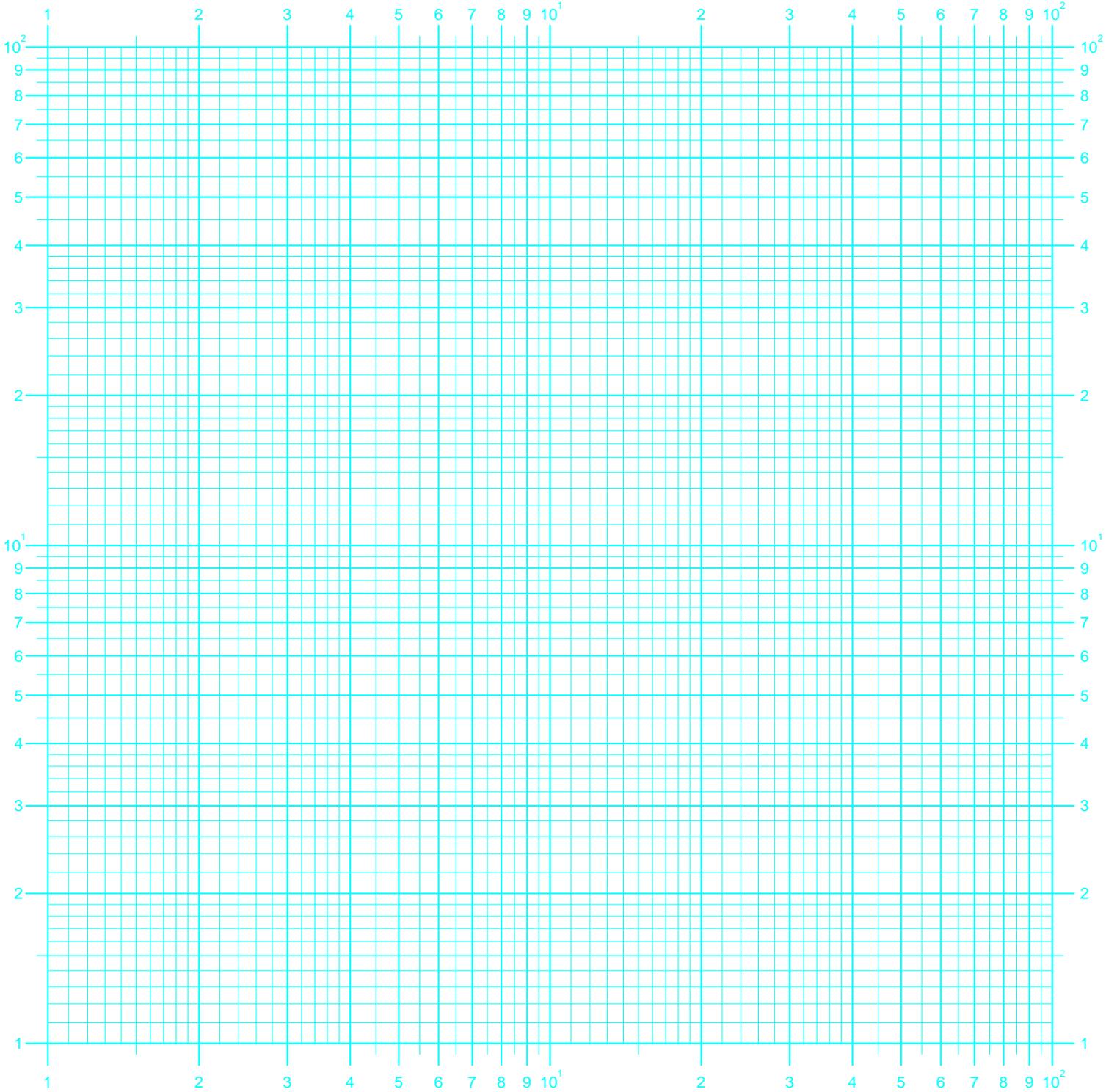
**EXERCISE 41**

Create the following piece of logarithmic paper. Write your code such that you can easily generate graph paper of any number of horizontal boxes of unit size and any number of vertical cycles.



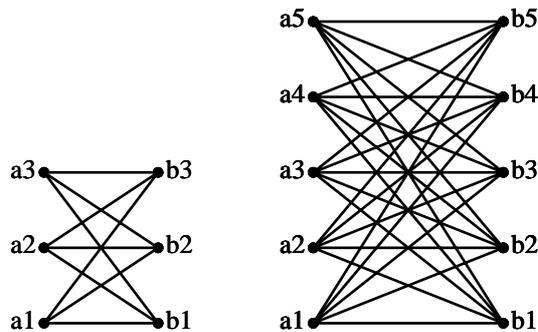
**EXERCISE 42**

Create the following piece of double logarithmic paper. Write your code such that you can easily generate graph paper of any number of horizontal and vertical cycles.

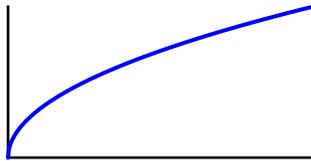


**EXERCISE 43**

A graph is bipartite when its vertices can be partitioned into two disjoint sets  $A$  and  $B$  such that each of its edges has one endpoint in  $A$  and the other in  $B$ . The most famous bipartite graph is  $K_{3,3}$  show below to the left. Write a program that draws the  $K_{n,n}$  graph for any natural number  $n > 1$ . Show that your program indeed creates the graph  $K_{5,5}$ , which is shown below to the right.

**EXERCISE 44**

1. Try to understand how the polygon approximation of the graph of the function  $x \mapsto \sqrt{x}$  on the interval  $(0,2)$  is obtained in the following PostScript program.

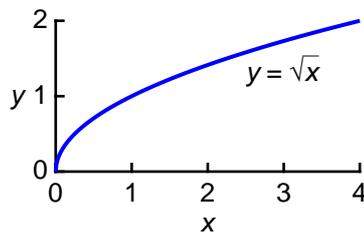


```

%%BoundingBox: 0 0 140 100
%%BeginProlog
/u 28.343452 def % unit: 1 cm
/lw1 1 u div def /lw2 1.5 u div def
/xmin 0 def /xmax 4 def
/ymin 0 def /ymax 2 def
/dx 0.01 def % stepsize
%%EndProlog
u dup scale 0.75 1 translate
2 setlinejoin 2 setlinecap
lw1 setlinewidth
newpath % make axes
xmin 0 moveto xmax 0 lineto
0 ymin moveto 0 ymax lineto
stroke
0 0 1 setrgbcolor
lw2 setlinewidth 1 setlinecap
/x xmin def
newpath % make graph
x x sqrt moveto
{/x x dx add def x xmax 0.0001 add gt {exit} if
  x x sqrt lineto } loop
stroke
showpage
%%EOF

```

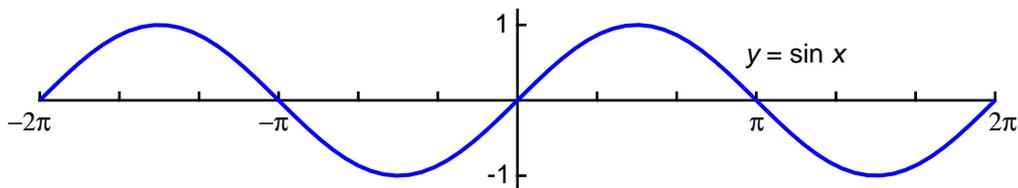
2. Add tick marks, axes labels and text so that your picture looks like



**EXERCISE 45**

1. Create a polygon approximation of the graph of the function  $x \mapsto \sin x$  on the interval  $(-2\pi, 2\pi)$  that looks like the following picture. How many line segments do you need for making a faithful representation of the sine graph on the interval  $(-2\pi, 2\pi)$ ?

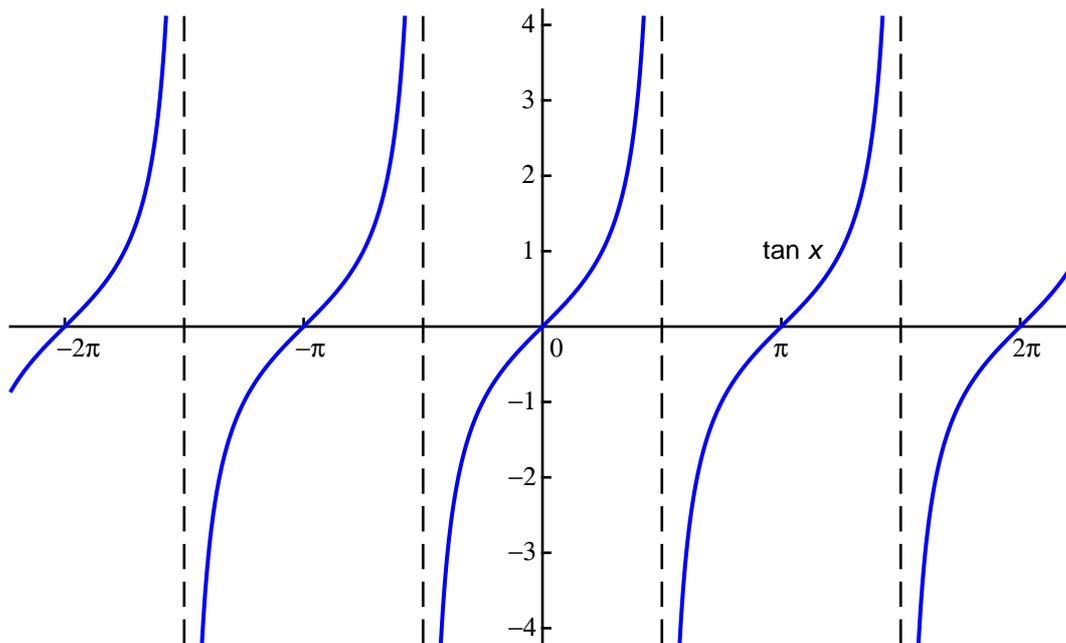
Try to make your code general enough so that it is easy to adapt the PostScript program to draw a graph for any domain  $(m\pi, n\pi)$ , for nonnegative integer  $m$  and natural number  $n$  (ignoring the labels, which you may set manually).



2. Change your polygon approximation into a cubic Bézier approximation. How many Bézier segments are required for a faithful representation of the sine graph on the interval  $(-2\pi, 2\pi)$ ? Is this change worth the effort?

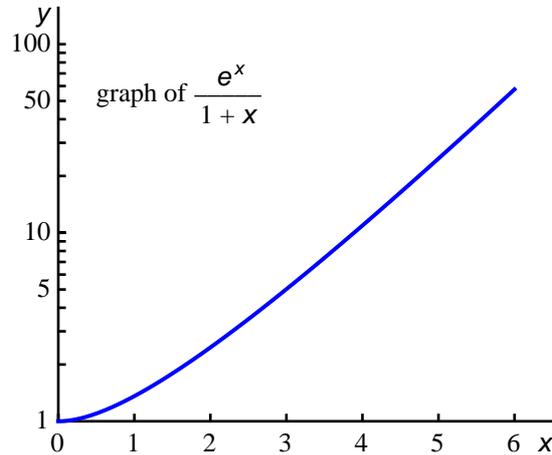
**EXERCISE 46**

Draw a polygon approximation of the graph of the function  $x \mapsto \tan x$  that looks like the following picture.

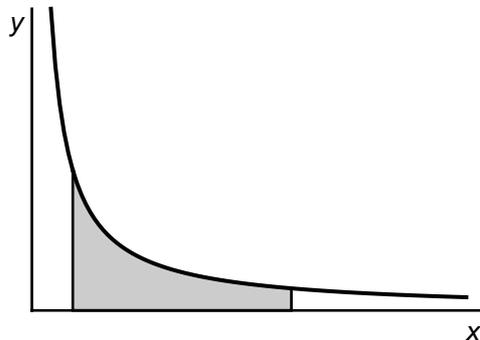


**EXERCISE 47**

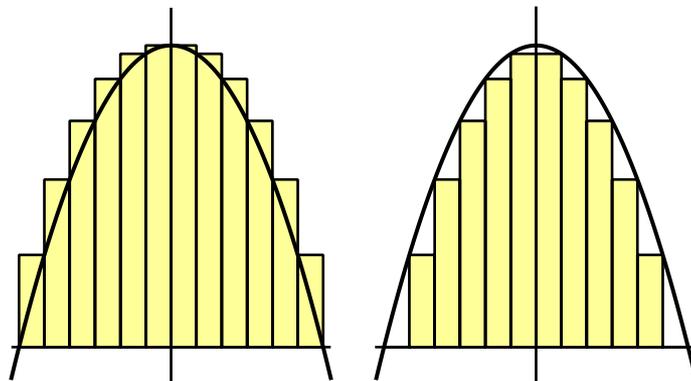
Draw a polygon approximation of the graph of the function  $x \mapsto \frac{e^x}{1+x}$  from 0 to 6 with the vertical axis in a logarithmic scale (in other words, create a logplot of the function). Your picture must look similar to the figure below. Also put some effort in making your code so general that you can easily change the horizontal range or choose another power of 10 as vertical range.

**EXERCISE 48**

Create the picture below, which has a shaded area enclosed by the horizontal axis and the graph of the function  $f(x) = 1/x$ .

**EXERCISE 49**

Create the picture below, which illustrates the upper and lower Riemann sum for the area enclosed by the horizontal axis and the graph of the function  $f(x) = 4 - x^2$ .



## 6 Coordinate Transformations

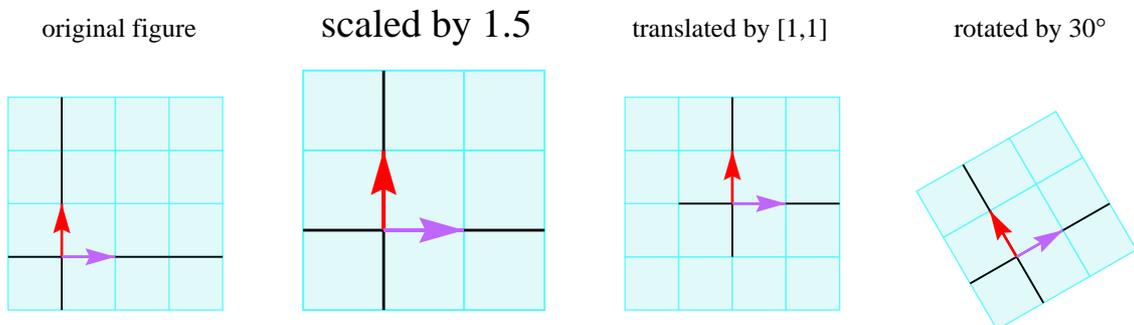
In many examples we already used coordinate transformation like translation and scaling to place the origin of our coordinate system at another location and to use another unit of scale for drawing a picture. In this chapter we will go into details of coordinate transformations.

PostScript distinguishes the *device coordinate system* and the *user coordinate system* (also called *user space*). The device may be a computer display or a printer. When you write a PostScript program, you only have to think about the user coordinate system except in a few rare cases. The operands of path operators are always coordinates in user space, which are then automatically transformed into device coordinates by the PostScript interpreter. Initially the user space origin is located at the lower-left corner, with the positive  $x$  axis extending horizontally to the right and the positive  $y$  axis extending vertically upward. The length of the unit along both  $x$  and  $y$  axis is  $1/72$  inch. We refer to this default unit as a ‘PostScript point’ (or shortly, ‘point’) to distinguish it from the ‘classical printer’s point’, which is  $1/72.27$  of an inch.<sup>12</sup> The coordinate system just described is the *default user space*. We adopt the convention used in [Cas05] and refer to this coordinate system as the *page*. A coordinate system can be defined with respect to the page by stating:

- The length of the units along each axis (**scale**).
- The location of the origin (**translate**).
- The orientation of the  $x$  and  $y$  axes (**rotate**).

In the above listing we have put between brackets the name of the operator that takes care of the coordinate transformation. To understand the working of *coordinate transformation operators* it is convenient to think of *coordinate frames* and to imagine the effect of transformations on a coordinate frame. The following sequence of pictures illustrates that

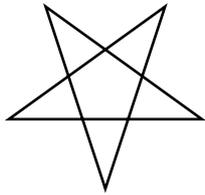
- **scale** modifies the unit lengths independently along the current  $x$  and  $y$  axes, leaving the origin location and the orientation unchanged.
- **translate** moves the origin of the user space to a new position with respect to the current coordinate system, leaving the orientation and the unit lengths unchanged.
- **rotate** turns the user space axes about the current user space origin by some angle, leaving the origin location and the unit lengths unchanged.



---

<sup>12</sup>This illustrates the difference that can exist between device and user coordinates. For example, a high-resolution printer may have a unit like  $1/2400$  of an inch.

In short, coordinate transformation affect the current frame in the way you expect. For example, `1.5 1.5 scale` scales the current frame by a factor of 1.5, `1 1 translate` translates the current frame origin one unit left and upward, and so on. Drawing commands take effect relative to the current frame as can be seen in the following example of a star shaped picture.

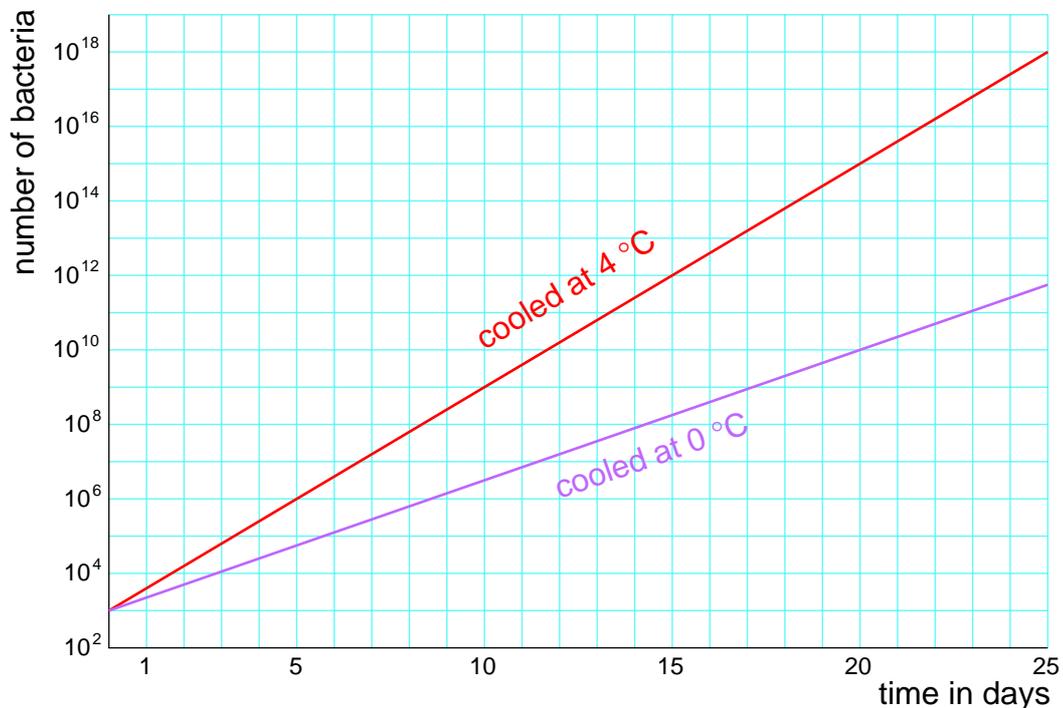


```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 85 75
%%BeginProlog
/side {72 0 lineto
      currentpoint translate
      144 rotate
} bind def
%%EndProlog
newpath
5 30 translate 0 0 moveto
4 { side } repeat closepath
stroke
showpage
%%EOF

```

With the `gsave` and `grestore` commands you can temporarily change the coordinate system. The following example with rotated text illustrates this.



We only show the code that generates to purple line with label. You can easily figure out how the rest of the picture has been created.

```

% ...
% draw line with rotated text label
newpath
0.7490 0.4 0.9843 setrgbcolor
0 1 moveto 25 9.75 lineto 12 4 moveto
gsave
  8.75 25 atan rotate fh0 (cooled at 0) show
  fs (\260) show fh0 (C) show
grestore
stroke

```

The operators `scale`, `translate` and `rotate` can be combined, or *concatenated*, to yield a single transformation with the same effect as the sequential application of the original operators. For example, a rotation of the coordinate system counter-clockwise around the point  $(a, b)$  by angle  $\theta$  can be performed by first translating the coordinate system by the vector  $(a, b)$ , then rotating around  $(0, 0)$  by  $\theta$ , and finally by translating the system by the vector  $(-a, -b)$ . To understand better how transformations work in PostScript, let us work out this example in more detail.

When PostScript starts up, the origin of the current coordinate frame is  $(0, 0)$  and the user coordinates  $(x_0, y_0)$  of some fixed point  $P$  are the same as the page coordinates  $(x_{page}, y_{page})$  of this point. Let us identify a point  $(x, y)$  with the 3-dimensional column vector<sup>13</sup>  $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$ ,

then we have

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

If we perform `a b translate`, i.e., if we translate the coordinate frame by the vector  $(a, b)$ , we find ourselves with new coordinates  $(x_1, y_1)$  of the point  $P$ . The page coordinates of the new origin are  $(a, b)$ , but the page coordinates of the point  $P$  have not changed. Thus,

$$(x_0, y_0) = (x_1, y_1) + (a, b)$$

or

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} .$$

If we now perform `\theta rotate`, the origin does not change and we find ourselves with new coordinates  $(x_2, y_2)$  of the point  $P$ . They are related with the previous coordinates by

$$(x_2, y_2) = (x_1 \cos \theta + y_1 \sin \theta, -x_1 \sin \theta + y_1 \cos \theta)$$

or

$$(x_1, y_1) = (x_2 \cos \theta - y_2 \sin \theta, x_2 \sin \theta + y_2 \cos \theta) .$$

---

<sup>13</sup>In computer graphics and in PostScript it is customary to use row vectors. We use the mathematical notation of column vectors. All our results can be converted into computer graphics notation by transposing matrices and vectors.

In matrix notation, we get

$$\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}$$

and thus

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}.$$

If we now perform  $-a -b$  **translate**, i.e., if we translate the coordinate frame by the vector  $(-a, -b)$ , we find ourselves with new coordinates  $(x_3, y_3)$  of the point  $P$ . They are related with the previous coordinates by

$$(x_2, y_2) = (x_3, y_3) - (a, b).$$

In matrix notation, we get

$$\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix}$$

and thus

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix}.$$

What we see, is that each of the coordinate transformations is described analytically by a matrix:

$$\text{rotate around origin by } (\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \text{translate by } (a, b) = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}.$$

A concatenation of coordinate transformations corresponds with a multiplication of the corresponding matrices. Thus if a point has coordinates  $(x, y)$  in a coordinate system that is rotated counter-clockwise around the point  $(a, b)$  by angle  $\theta$ , then its coordinates  $(x_{page}, y_{page})$  in the current page are given by

$$\begin{aligned} (x_{page}, y_{page}) &= (x \cos \theta - y \sin \theta + a(1 - \cos \theta) + b \sin \theta, \\ &\quad x \sin \theta + y \cos \theta + b(1 - \cos \theta) - a \sin \theta). \end{aligned}$$

or in matrix notation by

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta & a(1 - \cos \theta) + b \sin \theta \\ \sin \theta & \cos \theta & b(1 - \cos \theta) - a \sin \theta \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

As a matter of fact, any coordinate transformation that we have seen until now can be described by an affine matrix  $T$  that has the form

$$T = \begin{pmatrix} T_{xx} & T_{xy} & T_x \\ T_{yx} & T_{yy} & T_y \\ 0 & 0 & 1 \end{pmatrix}.$$

The corresponding change from user space coordinates  $(x, y)$  to page coordinates  $(x_{page}, y_{page})$  is given by

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} T_{xx} & T_{xy} & T_x \\ T_{yx} & T_{yy} & T_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

or

$$(x_{page}, y_{page}) = (T_{xx}x + T_{xy}y + T_x, T_{yx}x + T_{yy}y + T_y).$$

This mapping is completely determined by the sextuple  $(T_x, T_y, T_{xx}, T_{xy}, T_{yx}, T_{yy})$  and this is the way PostScript stores information about affine coordinate transformations.

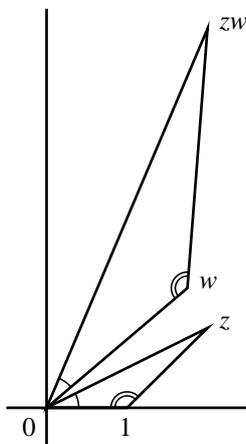
The matrix formulation of a coordinate transformation suggest that there are other transformations that we have not considered yet, e.g., the transformation

$$(x_{page}, y_{page}) = (ax - by, bx + ay)$$

or

$$\begin{pmatrix} x_{page} \\ y_{page} \\ 1 \end{pmatrix} = \begin{pmatrix} a & -b & 0 \\ b & a & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

This transformation is called *zscaling*. The effect of this mapping is to rotate and scale so as to map  $(1, 0)$  into  $(a, b)$ . This transformation can also be thought of as multiplication of complex numbers as the following example illustrates.



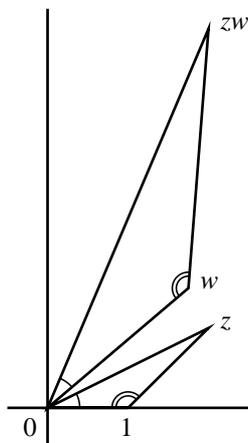

---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 95 170
%%BeginProlog
/u 30 def % 30 points per unit
/lw1 1 u div def
/lw2 0.6 u div def
/fh {
  /Times-Roman findfont 10 u div scalefont setfont
} bind def
/fi {
  /Times-Italic findfont 10 u div scalefont setfont
} bind def
/zscale {% stack: x y a b
          % output: a*x-b*y b*x+a*y
  4 dict begin
    /b exch def
    /a exch def
    /y exch def
    /x exch def
    a x mul b y mul sub b x mul a y mul add
  end} def
/z {2 1} def
/w {1.75 1.5} def
/zw {z w zscale } def
%%EndProlog

```

---



```

17 17 translate
u dup scale
lw1 setlinewidth
newpath
% draw lines
-0.5 0 moveto 2.5 0 lineto
0 -0.5 moveto 0 5 lineto
0 0 moveto 1 0 lineto z lineto closepath
0 0 moveto w lineto zw lineto closepath
-0.3 -0.35 moveto fh (0) show
0.9 -0.35 moveto fh (1) show
z exch 0.15 add exch moveto fi (z) show
w exch 0.15 add exch moveto fi (w) show
zw exch 0.15 add exch moveto fi (zw) show
stroke
% draw arcs
lw2 setlinewidth
/argz z exch atan def
/argw w exch atan def
newpath 0 0 0.4 0 argz arc stroke
newpath 0 0 0.4 argw argw argz add arc stroke
/angle z exch 1 sub atan def
newpath 1 0 0.2 angle 180 arc stroke
newpath 1 0 0.15 angle 180 arc stroke
newpath w 0.2 argw angle add argw 180 add arc stroke
newpath w 0.15 argw angle add argw 180 add arc stroke
showpage
%%EOF

```

An alternative definition of the `zscale` is: specify the sextuple  $(T_{xx}, T_{yx}, T_{xy}, T_{yy}, T_x, T_y)$  that defines the affine coordinate transformation  $T$  and provide this sextuple as argument to the `transform` operator, together with the coordinates of the point that must be transformed. The code is as follows:

```

/zscale {% stack: x y a b
        % output: a*x-b*y b*x+a*y
4 dict begin
/b exch def
/a exch def
/y exch def
/x exch def
x y [a b b neg a 0 0] transform
end} def

```

### EXERCISE 50

1. Verify that a scaling transformation is described by an affine matrix of the form

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

2. Determine the matrix that describes zooming with fixed point  $(a, b)$  and zoom factor  $c$ .

**EXERCISE 51**

Characterize the transformation  $(x_{page}, y_{page}) = (x + a y, y)$

The `zscale` operator that we defined in the above example is just an operator that maps one point into another. It does not affect the current coordinate system. A change of the coordinate system can only be performed by a change of the *current transformation matrix* or *CTM*, which contains the current affine transformation from user space to device space. If you want to apply the affine transformation

$$T = \begin{pmatrix} T_{xx} & T_{xy} & T_x \\ T_{yx} & T_{yy} & T_y \\ 0 & 0 & 1 \end{pmatrix}$$

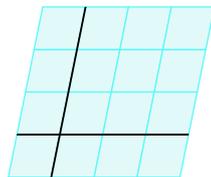
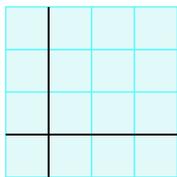
then you must enter the statement

`( $T_{xx}, T_{yx}, T_{xy}, T_{yy}, T_x, T_y$ ) concat`

The CTM is multiplied by  $T$ , with the effect that the transformation from user space to device space is adjusted to the new situation. The default CTM can always got back by entering

`matrix defaultmatrix setmatrix`

Let us illustrate a slanted coordinate system:




---

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 210 84
%%BeginProlog
/u 20 def % 20 point per unit
/lw1 0.5 u div def
/lw2 0.7 u div def
%
/grid {4 dict begin % m x n grid
  /n exch def
  /m exch def
  lw1 setlinewidth
  0 1 n 1 sub {/i exch def
    0 1 m 1 sub {/j exch def
      newpath
      gsave
        .88235 0.97647 0.97647 setrgbcolor
        i j 1 1 rectfill
      grestore
        .36040 0.97647 0.97647 setrgbcolor
        i j 1 1 rectstroke
      } for
    } for
  } for

```

---

---

```

newpath
lw2 setlinewidth 0 setgray
1 0 moveto 0 m rlineto
0 1 moveto n 0 rlineto
stroke
end } bind def
%%EndProlog
u dup scale
lw2 setlinewidth
0.1 0.1 translate
4 4 grid
5.5 0 translate
[1 0 0.2 1 0 0] concat
4 4 grid
showpage
%%EOF

```

---

### EXERCISE 52

1. Enter in the GHOSTSCRIPT interpreter the following statement

```
matrix currentmatrix ==
```

to find out what is the default coordinate transformation from user space to device space, i.e., to your computer display.

2. Enter the following statement

```
matrix defaultmatrix ==
```

to find out what is the default coordinate transformation from the page to the device space. Why is this transformation the same as the answer to the previous subquestion?

3. Reflect your coordinate system in the line through the origin and the point (2, 1). Check that the CTM has changed. Also verify with the statement `matrix defaultmatrix ==` that the default matrix is not affected (that is why it is called the 'default CTM').
4. Enter the following statement

```
matrix defaultmatrix setmatrix
```

and verify that you are back in the initial state.

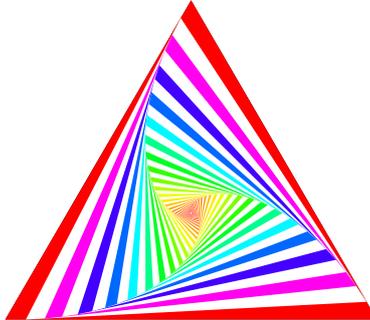
### EXERCISE 53

Using transformations, construct the following picture:



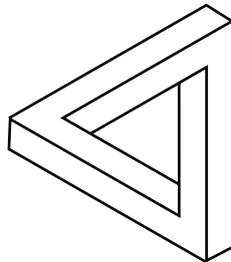
**EXERCISE 54**

Using transformations, construct the following picture (Hint: our colors are  $(b, 1, 1)$  in the HSB color scheme, where  $b$  is a number between zero and one).

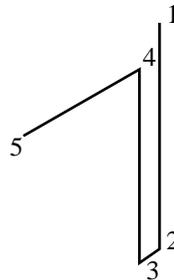


**EXERCISE 55**

Using transformations, construct Escher's impossible triangle:

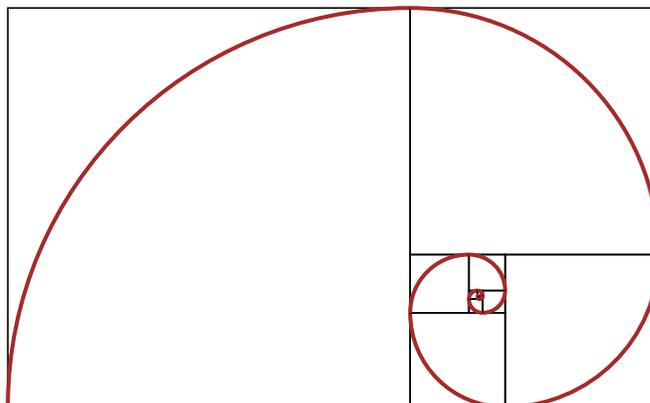


*Hint:* identify the base element shown to the right in the picture and find out how Escher's impossible triangle can be built up from copies of the base element.



**EXERCISE 56**

Using transformations, construct the following 'golden picture'. Following the curve from the lower left corner, the path moves through neighboring squares that have as a ratio of size  $\phi : 1$ , where  $\phi$  is the golden ratio ( $\phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618034$ ).



## 7 Procedures

In the examples so far we have already used many procedures that we defined ourselves. In this chapter we will teach you the tricks of the programming trade

### 7.1 Defining Procedure

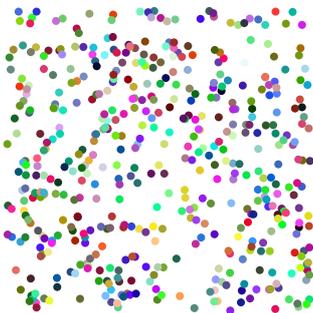
The most common form of a procedure or operator definition is:

```
name { procedure body } def
```

The `def` operator takes two arguments: an object that must evaluate to a name<sup>14</sup> and an object that consist of a collection of instructions in a executable array. All the PostScript interpreter does is to make an association between these objects in the current dictionary.

### 7.2 Parameter Passing

Unlike many imperative programming languages, there is in PostScript no procedure declaration to give names to input and output parameters. A PostScript procedure or operator passes information through the operand stack. In general, an operator takes all of its operands off the stack and returns its computed value(s) on the stack. In the following example, the procedure `randpoint` generates randomly a point within the given boundaries of region and with a random color.



```
!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 120 120
%%BeginProlog
/u 28.34645 def % unit: 1 cm
/point {1.5 u div 0 360 arc gsave fill grestore stroke} def
/ranndum {% stack: l r
          % output: random number between 1 and r
          dup 3 1 roll sub % stack: r l-r
          rand 1000 mod 1000 div mul add
} bind def %
%
/ranndpoint {% stack: l r b u
             % output random point in region [l,r] x [b,u]
             % in random color
/savecolor { currentrgbcolor } def
0 1 ranndum 0 1 ranndum 0 1 ranndum setrgbcolor
ranndum 3 1 roll ranndum % random point on stack
gsave newpath 1.5 u div 0 360 arc fill grestore
savecolor setrgbcolor
} bind def
%%EndProlog
%%Page: 1 1
u dup scale 0.1 0.1 translate
500 {0 4 0 4 ranndpoint} repeat
showpage
%%EOF
```

---

<sup>14</sup>Here we are sure to start with a name because of slash, but ( *name* ) cvn would do equally well.

It is possible to test the type of parameters via the `type` operator. For example, our procedure `randnum` can be made more robust by checking that the arguments `l` and `r` are real numbers such that  $l < r$ . In the program below, we introduce local variables to make the code more readable, but we could have stuck to the pure operator based programming style.

```

/randnum {% stack: l r
          % output: random number between l and r
  2 dict begin
  /r exch def
  /l exch def
  r type /realtyp e eq r type /integertype eq or
  l type /realtyp e eq r type /integertype eq or and
  l r lt and {
  r l r sub rand 1000 mod 1000 div mul add }
  { 0 } ifelse %
end } bind def

```

The trick is to ask the type of the object via the `type` operator and compare the answer with known types. A minor complication is that the type checking is based on internal representation and not in mathematical terms. For example, `0` has `integertype` and not `realtyp e`. This makes it a bit more work to test whether a parameter is a number.

The possible names that the `type` operator may return are listed below:

arraytype	filetype	integertype	nulltype	realtyp e
booleantyp e	fonttype	marktype	operatortyp e	savetyp e
dicttype	gstatetype	nametype	packedarraytype	stringtype

In practice, it is only worth to do type checking if you intend to reuse the procedure in many other programs and store it in a file, say `procs.inc`, that you then can include in PostScript file with the statement `(procs.inc) run`.

### 7.3 Local variables

To avoid name conflicts it is wise to make variables that you use only inside a procedure local to that procedure. PostScript provides a *dictionary* mechanism for this purpose. In the example of the procedure `randnum` in the previous section, we have a local dictionary with initial capacity for two items, which has been created by the statement `2 dict`. The `begin` operator pushes the dictionary on the so-called *dictionary stack*, making it the current dictionary and installing it as the first dictionary to use for name lookup. The `end` takes the current dictionary off the dictionary stack and makes the dictionary below it on the stack the current dictionary.

### 7.4 Recursion

A procedure is defined recursively if in its definition, it makes a call to itself. Recursive definition of a procedure is possible in PostScript, but it is inherently stack-based. The reason why is illustrated by the following PostScript program that computes the factorial of a natural number. Recall that  $n$  factorial,  $n!$ , is defined for natural numbers  $n$  by

$$n! = n \times (n - 1) \times \cdots \times 3 \times 2 \times 1$$

```

/! { 1 dict begin
  /n exch def
  n 1 eq {
    1
  }{
    n n 1 sub ! mul
  } ifelse
end } bind def

```

The code is correct, but only if  $n$  is not too large. For large values of  $n$  you get an error message about a dictionary stack overflow. At each call of the `!` operator you place a local dictionary on the dictionary stack because of the `1 dic begin` statement. Only when the `end` command is reached, the local dictionary is taken off the stack. But the dictionary stack has limited size and this is exactly the problem with our PostScript code.

As we have just seen, recursive definition of a procedure is possible in PostScript, but it is inherently stack-based. You can either use the dictionary stack or the operand stack for storing intermediate results until the recursion is unwound back to the original invocation level. The trick to avoid stack overflow as much as possible is to avoid open dictionaries across the recursive procedure calls. We will introduce both methods in computing factorials and Lucas numbers.

### Recursion Using the Operand Stack

The following PostScript program for computing factorials is taken from the blue book [Pos86]. We add comments to show what is on the stack during execution.

```

/fac {          % stack: n
  dup          % n n
  1 gt { % check n>1; % n
    dup        % n n
    1 sub fac  % n (n-1)!
    mul        % n*(n-1)!
  } if
} bind def

```

This procedure duplicates the number on the stack and if it is greater than 1, it is multiplied by the result of calling `fac` with its numeric predecessor. If then number is less or equal than 1, then no action is taken and the function returns with that number. You can easily check that the statement `5 fac` indeed leaves the value 120 on the stack.

The Lucas numbers  $L_n$  are defined by the following linear recurrence

$$L_1 = 1, F_2 = 3, \text{ and } L_n = L_{n-1} + L_{n-2}, \text{ for } n > 2.$$

The PostScript program for computing Lucas number on the basis of the recursion formula is as follows:

```

/L {          % stack: n
  dup 2 gt { % n

```

```

    1 sub dup      % n-1 n-1
    1 sub L       % n-1 L(n-2)
    exch L        % L(n-2) L(n-1)
    add           % L(n-2)+L(n-1)
  }{
    2 eq {3} {1} ifelse
  } ifelse
} bind def

```

This procedure first duplicates the given number on the stack. If it is greater than 2, we subtract 1, duplicate this number, subtract 1 from the top element of the stack, apply the procedure `L` to the two items on top of the stack and add the result. If the number is equal to 2, then we put 3 on the stack; otherwise we put 1 on the stack. You can easily check that the statement `6 L` indeed leaves the value 18 on the stack.

## Recursion Using the Dictionary Stack

The PostScript program:

```

/factorial {
  save
  2 dict begin
    saveobj exch def
    /n exch def
    n 1 gt {
      n n 1 sub factorial mul
    }{
      n
    } ifelse
    saveobj
  end
  restore
} bind def

```

If you want to store the procedure's argument in a dictionary, you must create a new dictionary and push it on the dictionary stack each time the function is called, to maintain the name local to that instance of the procedure. In this example, the memory allocated by the dictionary is reclaimed by `save` and `restore`, putting each save object into the recursion dictionary until it is needed.

The Lucas numbers can be computed in PostScript as follows:

```

/Lucas {
  save
  2 dict begin
    /saveobj exch def
    /n exch def
    n 2 gt {
      n 1 sub Lucas n 2 sub Lucas add
    }
  end
} bind def

```

```

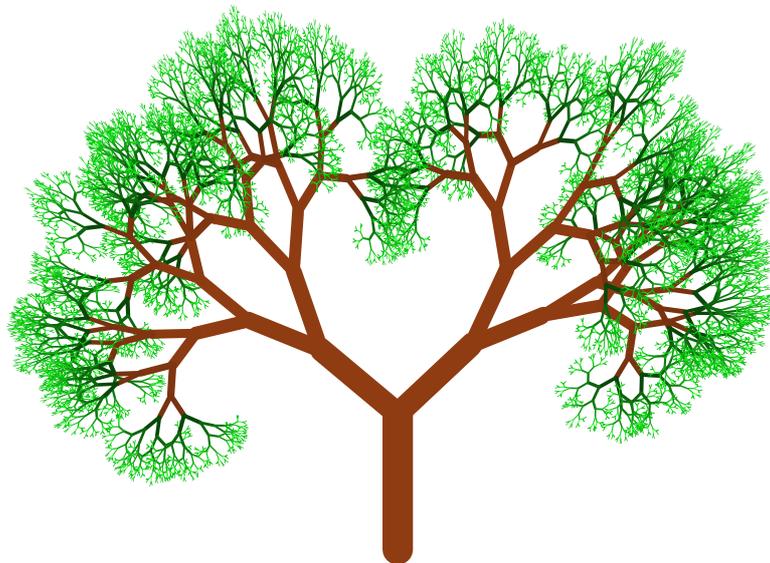
    }{
      n 2 eq {3} {1} ifelse
    } ifelse
    saveobj
  end
  restore
} bind def

```

Both methods of implementing recursive procedures work equally well. It is more a matter of personal taste which one prefers to use.

### A Graphical Example

Let us end the section on recursion with a graphical example: the computation of a Pythagorean tree. The picture for a recursion depth of 14 levels is:



The PostScript program:

```

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 300 215
%%BeginProlog
/u 28.34568 def % unit: cm
/size 1.8 def
/branchrotation 50 def %
/thinning 0.7 def
/shortening 0.8 def
/width 0.4 def% starting line width
/min {2 copy gt {exch pop}{pop} ifelse }def
/randdeviate {rand 1000 mod 1000 div mul} bind def
/colortable [ [0.013 0.750 0.028]
              [0.109 0.937 0.118]
              [0.085 0.727 0.092]

```

```

        [0.070 0.602 0.076]
        [0.055 0.469 0.059]
        [0.046 0.395 0.050]
        [0.033 0.281 0.035]
        [0.469 0.196 0.059]
        [0.500 0.210 0.063]
        [0.547 0.229 0.069]
        [0.562 0.236 0.071]
    ] def
/tree{
  save
  3 dict begin % stack: n
  /saveobj exch def
  /n exch def
  n 0 gt {
    newpath
    gsave
    0 0 moveto
    colortable n 1 sub 10 min get aload pop setrgbcolor
    width setlinewidth
    0 size rlineto
    currentpoint currentpoint % store current point twice
    stroke
  grestore
  %%%% draw right branch from current point
  newpath
  gsave
  translate
  thinning shortening scale
  -10 branchrotation randdeviate sub rotate
  0 0 n 1 sub tree
  stroke
  grestore
  %%%% draw left branch from current point
  newpath
  gsave
  translate
  thinning shortening scale
  10 branchrotation randdeviate add rotate
  n 1 sub tree
  stroke
  grestore
  } if
  saveobj
end
restore
} bind def

```

```

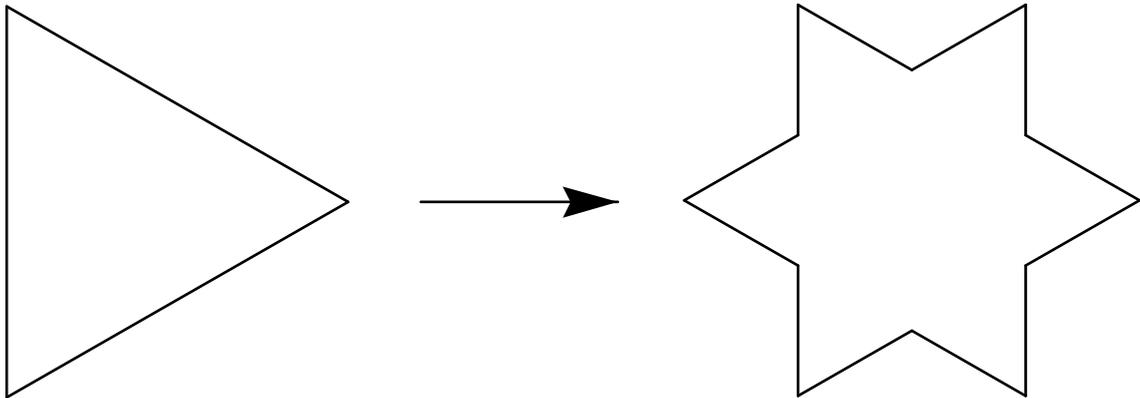
%%EndProlog
150 8 translate
u dup scale
1 setlinecap
14 tree
showpage
%%EOF

```

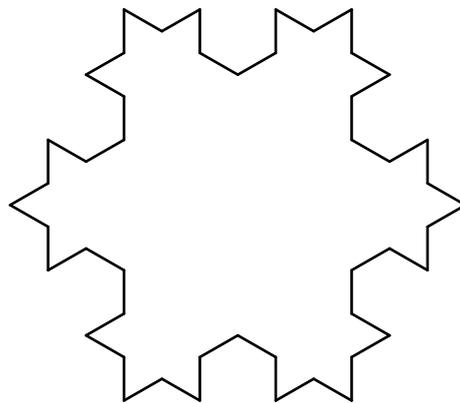
The only issue that you must keep in mind when using recursion is the danger of running out of space for the dictionary stack or operands stack during execution of the recursive computation. In this example, trying to draw the Pythagorean tree with a recursion depth of 15 levels fails because of stack overflow.

### EXERCISE 57

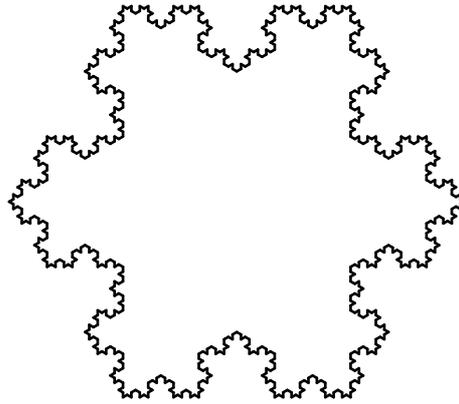
The Koch snowflake is constructed as follows: start with an equilateral triangle. Break each edge into four straight pieces by adding a bump as shown below.



You can then repeat the process of breaking a line segment into four pieces of length one-fourth of the segment that is broken up. Below you see the next iteration.

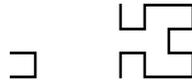


Write a PostScript program that can compute the picture after  $n$  iterations. After four iterations, the Koch snowflake should look like

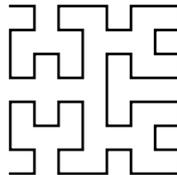


**EXERCISE 58**

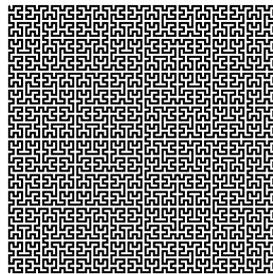
A Hilbert curve consists of 4 rotated and translated copies of a basis element connected by three straight lines.  $H_0$  is a dot. The base element of the Hilbert curve  $H_n$  of order  $n$  is  $H_n$ , for  $n = 1, 2, \dots$ . Below  $H_1$  and  $H_2$  are drawn:



1. Write a PostScript program that draws the Hilbert curve  $H_n$  of order  $n$ . After three iterations, your curve should look like



2. When you not only rotate and translate the base element, but also scale it to an appropriate smaller scale that makes the overall picture of constant size, then you get a space filling curve. Such curves are known as Peano curves. Adapt your program for drawing Hilbert curves such that it can be used to create the following picture (a scaled Hilbert curve of order 6):



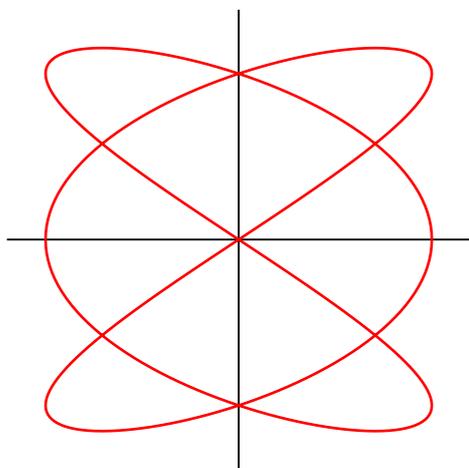
## 8 More Examples

The examples in this chapter are meant to give you an idea of the strength of PostScript for making nice mathematical illustrations in exactly the way that you want them to look like.

## 8.1 Planar curves

When a parametrization of a plane curve is given, then it is straightforward to draw a polygon approximation of the curve. The algorithm is as follows: suppose that the parameter interval is given by  $(t_0, t_1)$  and that we want to draw a curve  $P(t)$ . First, we set a step size  $h = (t_1 - t_0)/n$  so that we cross in  $n$  steps from  $t_0$  to  $t_1$ . We choose  $n$  large enough so that the straight lines connecting the neighboring points of  $P(t_0), P(t_0 + h), P(t_0 + 2h), \dots, P(t_0 + nh) = P(t_1)$  together build up a curve that looks smooth to the eye.

A concrete example of a parameterized curve is the following Lissajous figure, which is defined in parameterized form by  $t \mapsto (\sin(pt), \sin(qt))$ , for relatively prime numbers  $p$  and  $q$ . Here we take  $p = 3, q = 2$



---

```
%%BoundingBox: 0 0 180 180
%%BeginProlog
/u 72 def % unit: 1 inch
/lw1 1 u div def
/lw2 0.7 u div def
/n 360 def % number of line segments
/t0 0 def /t1 360 def
/p 3 def /q 2 def
/X {p mul sin} bind def % sin(p*t)
/Y {q mul sin} bind def % sin(q*t)
%%EndProlog
90 90 translate
u dup scale
lw2 setlinewidth 2 setlinejoin
% draw axes
newpath
-1.2 0 moveto 1.2 0 lineto
0 -1.2 moveto 0 1.2 lineto
stroke
newpath % polygon approximation of curve
lw1 setlinewidth 1 0 0 setrgbcolor
/dt t1 t0 sub n div def
/t t0 def
t X t Y moveto
n {
  /t t dt add def
  t X t Y lineto
} repeat
stroke
showpage
%%EOF
```

---

As you see, if  $n$  is large enough, then the polygon approximation of the curve is smooth. In practice, you do not often have to use a cubic Bézier approximation for a nice graph. If the

derivatives of the coordinate functions  $x(t)$  and  $y(t)$  are known, it is not much work to build up the Bézier approximation. The control points are determined as follows: suppose that

$$P_{0,i} = (x(t_i), y(t_i))$$

and

$$P_{0,i+1} = (x(t_{i+1}), y(t_{i+1}))$$

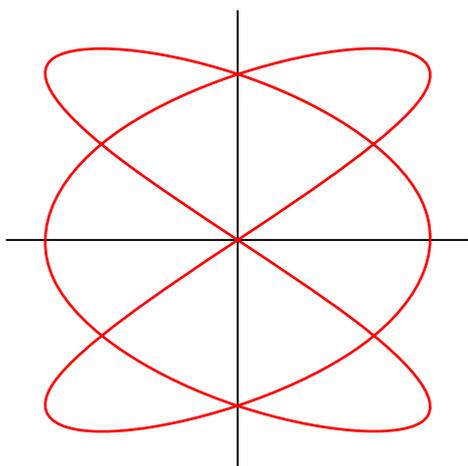
are the end points of the  $i$ th cubic Bézier line segment, where  $t_{i+1} = t_i + dt$ , then

$$P_{1,i} = (x(t_i) + \frac{1}{3}x'(t_i) dt, y(t_i) + \frac{1}{3}y'(t_i) dt)$$

and

$$P_{2,i} = (x(t_{i+1}) - \frac{1}{3}x'(t_{i+1}) dt, y(t_{i+1}) - \frac{1}{3}y'(t_{i+1}) dt)$$

The PostScript code is shown below and the example shows that 15 cubic Bézier segments suffice to present a faithful graph of the Lissajous figure. But we cannot stress it too much, a polygon approximation is in most practical cases more than good enough if you choose a step size that is small enough.



```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 180 180
%%BeginProlog
/u 72 def % unit: 1 inch
/lw1 1 u div def
/lw2 0.7 u div def
/n 15 def % number of line segments
/t0 0 def /t1 360 def
/p 3 def /q 2 def
/X {p mul sin} bind def % sin(p*t)
/Y {q mul sin} bind def % sin(q*t)
% Note: sin and cos use degrees in Postscript
/X' {p mul cos p mul 3.14159 mul 180 div} bind def
/Y' {q mul cos q mul 3.14159 mul 180 div} bind def
%%EndProlog
90 90 translate
u dup scale
lw2 setlinewidth 2 setlinejoin
newpath %%%% draw axes
-1.2 0 moveto 1.2 0 lineto
0 -1.2 moveto 0 1.2 lineto
stroke
newpath % Bezier approximation of curve
lw1 setlinewidth 1 0 0 setrgbcolor
/dt t1 t0 sub n div def
/dt3 dt 3 div def
/t t0 def
t X t Y moveto % P0
/X't t X' def
/Y't t Y' def

```

---

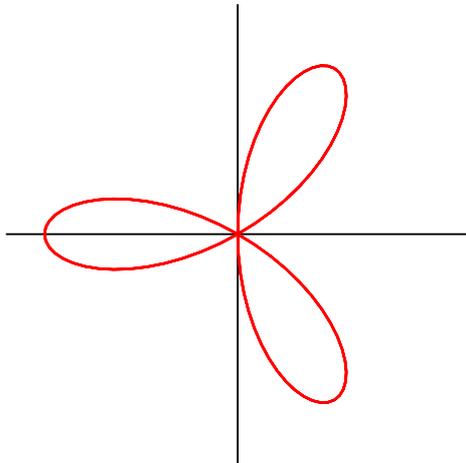
```

n {
  t X dt3 X't mul add
  t Y dt3 Y't mul add % P1
  /t t dt add def
  /X't t X' def
  /Y't t Y' def
  t X dt3 X't mul sub
  t Y dt3 Y't mul sub % P2
  t X t Y % P3
  curveto
} repeat
stroke
showpage
%%EOF

```

---

When the curve is specified via polar coordinates, the creation of the graph is as easy as in the Cartesian coordinates. The following picture of the Folium, defined by the polar equation  $r = \cos \phi(4 \sin^2 \phi - 1)$ , illustrates this. It is basically a parameterized curve  $\phi \mapsto (r \cos \phi, r \sin \phi)$ , where  $r = \cos \phi(4 \sin^2 \phi - 1)$  is a function of the parameter  $\phi$




---

```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 180 180
%%BeginProlog
/u 72 def % unit: 1 inch
/lw1 1 u div def /lw2 0.7 u div def
/n 180 def % number of line segments
/pi 3.1415062 def
/deg {180 pi div mul} bind def
/r {dup deg cos exch
  deg sin dup mul 4 mul 1 sub mul} bind def
/X {dup r exch deg cos mul} bind def
/Y {dup r exch deg sin mul} bind def
%%EndProlog
90 90 translate
u dup scale
lw2 setlinewidth 2 setlinejoin
newpath % draw axes
-1.2 0 moveto 1.2 0 lineto
0 -1.2 moveto 0 1.2 lineto
stroke
newpath % Bezier approximation of curve
lw1 setlinewidth
1 0 0 setrgbcolor
/dphi 2 pi mul n div def % step size
/phi 0 def
phi X phi Y moveto

```

---

---

```

n {
  /phi phi dphi add def
  phi X phi Y lineto
} repeat
stroke
showpage
%%EOF

```

---

A curve might also be a solution curve of a system of differential equations. As an example we take the van der Pol equation

$$\begin{aligned}
 x'(t) &= y(t), \\
 y'(t) &= -x(t) + \mu(1 - x(t)^2) y(t),
 \end{aligned}$$

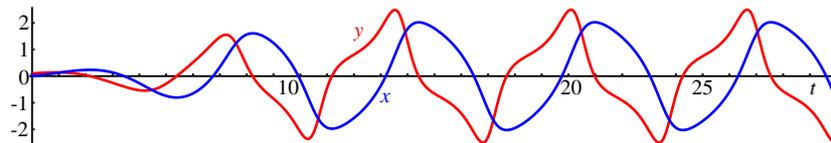
where  $\mu$  is the relaxation parameter. In the example below, we take values  $\mu = 0.8$ ,  $x(0) = 0$ ,  $y(0) = 0.1$  and approximate the solution curve by Euler's method. It works as follows: suppose that we already have computer part of the curve and that  $(x(t), y(t))$  is the last computed point at time  $t$ . At time  $t + dt$  the coordinates of the point of the solution curve are approximated by

$$\begin{aligned}
 x(t + dt) &\approx x(t) + x'(t) dt, \\
 y(t + dt) &\approx y(t) + y'(t) dt.
 \end{aligned}$$

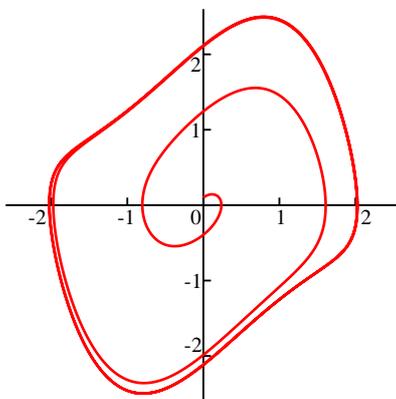
In other words,

$$\begin{aligned}
 x(t + dt) &\approx x(t) + y(t) dt, \\
 y(t + dt) &\approx y(t) + \left(-x(t) + \mu(1 - x(t)^2)\right) dt.
 \end{aligned}$$

For small step size  $dt$ , the error is small. First the plot of  $x(t)$  and  $y(t)$  against time:



We omit the PostScript code because it is similar to the one for plotting the parameterized curve  $t \mapsto (x(t), y(t))$ .




---

```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 150 150
%%BeginProlog
/u 28.34645 def % unit: 1 cm
/lw1 1 u div def /lw2 0.7 u div def
/f1 {
  /Times-Roman findfont 8 u div scalefont setfont
} bind def
/f2 {
  /Times-Italic findfont 8 u div scalefont setfont
} bind def

```

---

---

```

/n 3000 def % number of line segments
/dt 0.01 def
/mu 0.8 def
%%EndProlog
% x(t) and y(t) vs. t
75 75 translate
u dup scale
lw2 setlinewidth 2 setlinejoin
% draw axes
newpath
-2.6 0 moveto 2.6 0 lineto
0 -2.6 moveto 0 2.6 lineto
-2 1 2 { /i exch def
  i 0 moveto 0 0.1 rlineto
} for
-2 1 2 { /i exch def
  0 i moveto 0.1 0 rlineto
} for
stroke
% draw labels
-2.3 -0.25 moveto f1 (-2) show
-1.13 -0.25 moveto f1 (-1) show
  0.95 -0.25 moveto f1 (1) show
  2.07 -0.25 moveto f1 (2) show
-0.25 -1.95 moveto f1 (-2) show
-0.25 -1.1 moveto f1 (-1) show
-0.17 -0.25 moveto f1 (0) show
-0.17  0.9 moveto f1 (1) show
-0.17  1.77 moveto f1 (2) show
% approximation of x(t) and y(t)
newpath
lw1 setlinewidth 1 0 0 setrgbcolor
/t 0 def /x 0 def /y 0.1 def
x y moveto % initial position
n {
  /x' y def %  $x' = y$ ,  $y' = -x + \mu(1-x^2)y$ 
  /y' x neg mu 1 x x mul sub mul y mul add def
  /x x x' dt mul add def
  /y y y' dt mul add def
  /t t dt add def
  x y lineto
} repeat
stroke
showpage
%%EOF

```

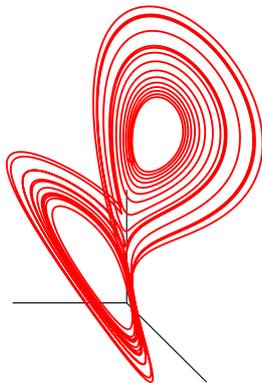
---

## 8.2 The Lorenz Butterfly

We do not have to restrict ourselves to planar curves. As an example we take the Lorenz butterfly, a curve in 3-dimensional space that is projected on a 2-dimensional image plane. The Lorenz attractor is described by the differential equations

$$x' = \sigma(y - x), \quad y' = \rho x - y - xz, \quad z' = xy - \beta z,$$

with  $x, y, z$  real functions of time  $t$ , and with  $\beta, \rho$ , and  $\sigma$  positive constants. We choose the standard values  $\sigma = 10$ ,  $\rho = 28$ , and  $\beta = 8/3$ . We take as initial values  $x(0) = 12.5$ ,  $y(0) = -17.4$ , and  $z(0) = 26$ .



```
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 105 150
%%BeginProlog
/u 2.834645 0.5 mul def % unit: 0.5 mm
/lw1 0.7 u div def /lw2 0.4 u div def
/n 20000 def % number of line segments
/dt 0.001 def
/Xr [-1 0] def
/Yr [0.7 -0.7] def
/Zr [0 1] def
/project { 3 dict begin % stack x y z
  /z exch def /y exch def /x exch def
  x Xr 0 get mul y Yr 0 get mul add z Zr 0 get mul add
  x Xr 1 get mul y Yr 1 get mul add z Zr 1 get mul add
end } bind def
%%EndProlog
% t -> x(t), y(t), z(t)
50 35 translate
u dup scale
lw2 setlinewidth 2 setlinejoin
% draw reference frame
gsave
  newpath
  30 30 scale lw2 30 div setlinewidth
  0 0 moveto Xr aload pop lineto
  0 0 moveto Yr aload pop lineto
  0 0 moveto Zr aload pop lineto
  stroke
grestore
% approximation of x(t), y(t), z(t)
newpath
lw1 setlinewidth 1 0 0 setrgbcolor
/t 0 def /x -12.5 def /y -17.4 def /z 26 def
x y z project moveto % initial position
```

---

```

n { % x'=10(y-x), y'=28x-xz-y, z'=xy-8z/3
/x' 10 y x sub mul def %
/y' 28 x mul x z mul sub y sub def
/z' x y mul 8 div 3 z mul sub def
/x x x' dt mul add def
/y y y' dt mul add def
/z z z' dt mul add def
/t t dt add def
x y z project lineto
} repeat
stroke
showpage
%%EOF

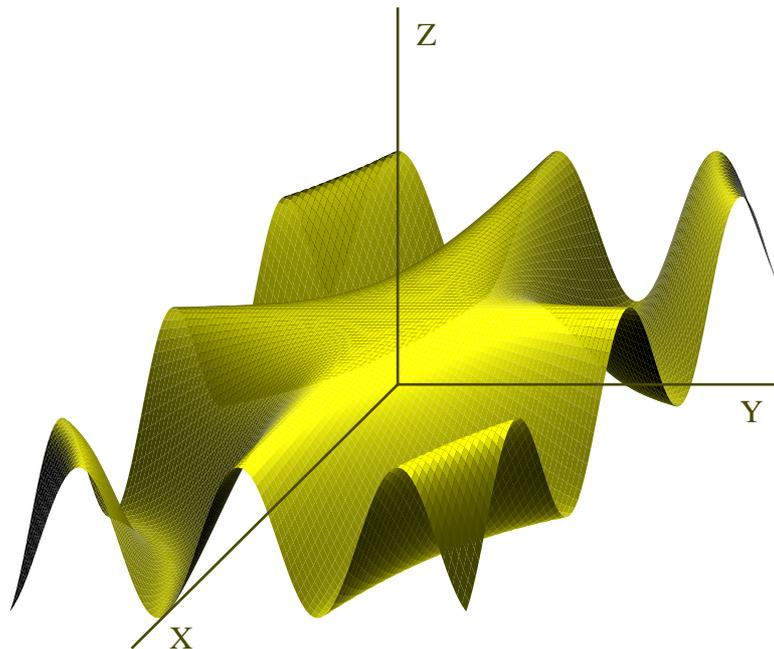
```

---

### 8.3 A Surface Plot

You can draw surface plots from basic principles. We give one example: the surface  $z = \cos(xy)$ .

---



```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 300 250
%%BeginProlog
/u 28.34645 def % unit: 1cm
/pi 3.1415927 def
/cosine {pi div 180 mul cos} bind def
/f {mul cosine} bind def % f(x,y) = cos(x*y)
%
```

```

/xp 3 def /yp 3 def /zp 10 def /bf 100 def
/Xr [-0.7 -0.7] def /Yr [1 0] def /Zr [0 1] def
%
/project { 3 dict begin % stack x y z
  /z exch def /y exch def /x exch def
  x Xr 0 get mul y Yr 0 get mul add z Zr 0 get mul add
  x Xr 1 get mul y Yr 1 get mul add z Zr 1 get mul add
end } bind def
%
% numerical derivatives by central differences
/xderiv {3 dict begin % stack: x y
  /y exch def /x exch def /h 0.01 def
  x h add y f x h sub y f sub 2 div h div
end } bind def
/yderiv {3 dict begin % stack: x y
  /y exch def /x exch def /h 0.01 def
  x y h add f x y h sub f sub 2 div h div
end } bind def
%
% compute brightness factor at a point
/brightnessfactor {8 dict begin % stack: x y z
  /z exch def /y exch def /x exch def
  /dfdx x y xderiv def /dfdy x y yderiv def
  /ca zp z sub dfdy yp y sub mul sub dfdx xp x sub mul sub def
  /cb 1 dfdx dup mul add dfdy dup mul add sqrt def
  /cc z zp sub dup mul y yp sub dup mul add x xp sub dup mul add sqrt def
  bf ca mul cb div cc div cc div cc div
end } bind def
%
/nx 100 def /ny 100 def % grid size
/xmin -3 def /xmax 3 def /ymin -3 def /ymax 3 def
%%EndProlog
150 105 translate u dup scale
% compute colors and draw patches
0 setlinewidth 1 0 0 setrgbcolor
/dx xmax xmin sub nx div def
/dy ymax ymin sub ny div def
0 1 nx 1 sub {/i exch def
  /xt xmin i dx mul add def
  0 1 ny 1 sub {/j exch def
    /yt ymin j dy mul add def
    /zt xt yt f def
    newpath
    xt yt zt brightnessfactor dup 0 setrgbcolor
    xt yt zt project moveto
    xt yt dy add 2 copy f project lineto
    xt dx add yt dy add 2 copy f project lineto
  }
}

```

```

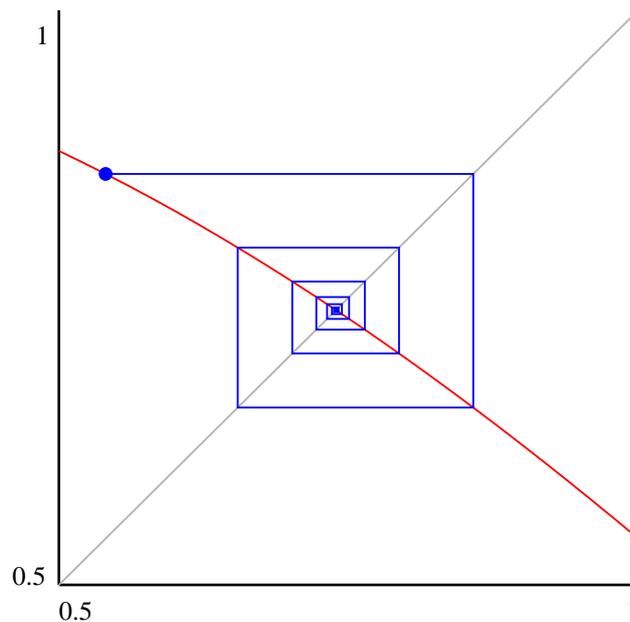
    xt dx add yt 2 copy f project lineto
    closepath
    fill
  } for
} for
stroke
1 u div setlinewidth
% draw reference frame
gsave
  newpath
  5 5 scale 1 u div 5 div setlinewidth
  0 0 moveto Xr aload pop lineto
  0 0 moveto Yr aload pop lineto
  0 0 moveto Zr aload pop lineto
  stroke
grestore
% labels
/Times-Roman findfont 12 u div scalefont setfont
-3 -3.5 moveto (X) show
4.5 -0.5 moveto (Y) show
0.25 4.5 moveto (Z) show
showpage
%%EOF

```

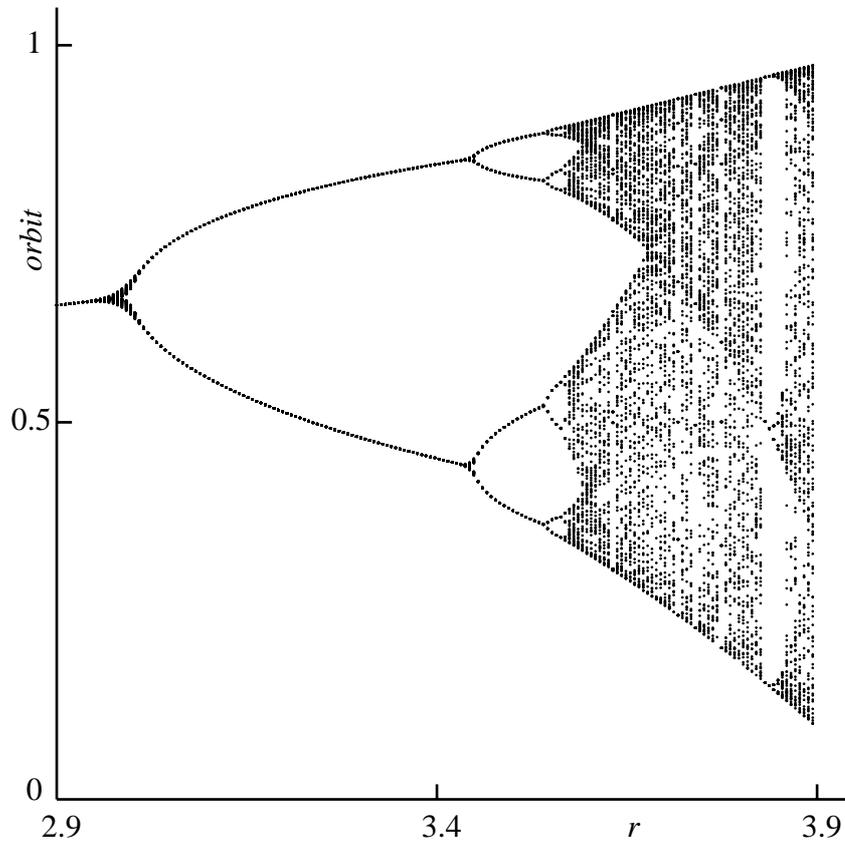
## 8.4 Iterated Functions

The following diagrams are standard in the theory of iterative processes:

- The cobweb-graph of applying the cosine function iteratively.



- The bifurcation diagram of the logistic function, i.e., of  $f(x) = rx(1 - x)$  for  $0 < r < 4$ .



The code that produced these diagrams is shown below.

```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 250 250
%%BeginProlog
/u 432 def % unit: 6 inch
/lw1 1 u div def /lw2 0.7 u div def
/f1 {/Times-Roman findfont 10 u div scalefont setfont} bind def
/pi 3.1415927 def
/cosine {pi div 180 mul cos} bind def
/f {mul cosine} bind def % f(x,y) = cos(x*y)
/xmin 0.5 def /xmax 1 def
/ymin xmin def /ymax xmax def
/dx 0.02 def
%%EndProlog
u dup scale
% drawaxes
0.05 xmin sub 0.05 ymin sub translate
lw1 setlinewidth

```

```

xmin ymin moveto xmax ymin lineto
xmin ymin moveto xmin ymax lineto
stroke
% labels near axes
xmin ymin 0.03 sub moveto f1 xmin 4 string cvs show
xmax 0.01 sub ymin 0.03 sub moveto f1 xmax 4 string cvs show
xmin 0.04 sub ymin moveto f1 ymin 4 string cvs show
xmin 0.02 sub ymax 0.03 sub moveto f1 ymax 4 string cvs show
% draw identity graph
newpath
lw2 setlinewidth 0.7 setgray
xmin ymin moveto xmax ymax lineto
stroke
% draw cosine graph
newpath
lw2 setlinewidth 1 0 0 setrgbcolor
/x xmin def
x x cosine moveto
{/x x dx add def
  x xmax 0.001 add gt {exit} if
  x x cosine lineto
} loop
stroke
% compute and draw orbit
newpath
lw2 setlinewidth 0 0 1 setrgbcolor
/initial 1 def % first some initial iterations
/x 1.0 def % the starting point
/orbitlength 15 def
1 1 initial {
  /x x cosine def
} for
x x cosine moveto
currentpoint
gsave newpath 0.006 0 360 arc fill grestore
1 1 orbitlength {
  /x x cosine def
  x x lineto x x cosine lineto
} for
stroke
showpage
%%EOF

```

```

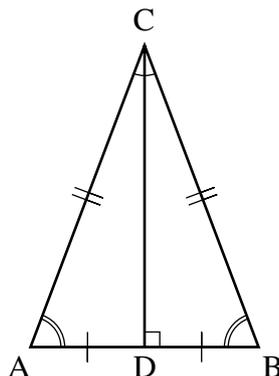
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 400 400
%%BeginProlog
/u 283.4645 def % unit: 10 cm
/lw1 1 u div def /lw2 0.7 u div def
/f1 {/Times-Roman findfont 12 u div scalefont setfont} bind def
/f2 {/Times-Italic findfont 12 u div scalefont setfont} bind def
/n 175 def
/rmin 2.9 def /rmax 3.9 def
/dr rmax rmin sub n div def
%%EndProlog
u dup scale
% drawaxes
0.1 rmin sub 0.1 translate
lw1 setlinewidth
rmin 0 moveto rmax 0.05 add 0 lineto
rmin 0 moveto rmin 1.05 lineto
% ticks and labels near axes
/rmid rmax rmin add 2 div def
rmid 0 moveto 0 0.02 rlineto
rmax 0 moveto 0 0.02 rlineto
rmin 0.5 moveto 0.02 0 rlineto
rmin 1 moveto 0.02 0 rlineto
stroke
rmin 0.02 sub -0.05 moveto f1 rmin 4 string cvs show
rmid 0.02 sub -0.05 moveto f1 rmid 4 string cvs show
rmax 0.02 sub -0.05 moveto f1 rmax 4 string cvs show
rmin 0.04 sub 0 moveto f1 (0) show
rmin 0.06 sub 0.49 moveto f1 (0.5) show
rmin 0.04 sub 0.99 moveto f1 (1) show
rmid rmax add 2 div -0.05 moveto f2 (r) show
gsave rmin 0.7 translate 90 rotate 0 0.02 moveto f2 (orbit) show grestore
% draw bifurcation diagram
/r rmin def /dr rmax rmin sub n div def
n {
  /x 0.5 def % our starting point
  75 { % initial iterations
    /x r x mul 1 x sub mul def % x := r*x*(1-x)
  } repeat
  150 {% the next 150 iterations
    /x r x mul 1 x sub mul def % x := r*x*(1-x)
    newpath r x 0.5 u div 0 360 arc fill
  } repeat
  /r r dr add def
} repeat
showpage
%%EOF

```

## 8.5 Marking Angles and Lines

In geometric pictures, line segments of equal length are often marked by an equal number of ticks and equal angles are often marked the same, too. In the following example, the procedures `tickmark`, `markangle`, and `markrightangle` mark lines and angles. We put effort in making our procedures `markangle`, and `markrightangle` independent of the order in which the non-common points of the angle are specified.

---



```

%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 115 150
%%BeginProlog
/u 28.34645 def % unit: 1cm
/lw1 1 u div def /lw2 0.5 u div def
/f1 {/Times-Roman findfont 12 u div scalefont setfont} bind def
/f2 {/Times-Italic findfont 12 u div scalefont setfont} bind def
/angleradius 0.4 def /angledelta 0.05 def /marksize 0.2 def
/angle {exch atan} def
%
/markangle {4 dict begin % stack: P common Q n
  /n exch def /Q exch def /common exch def /P exch def
  P common Q angleradius drawangle
  n 1 gt {P common Q angleradius angledelta add drawangle } if
  n 2 gt {P common Q angleradius angledelta sub drawangle } if
  n 3 gt {P common Q angleradius 2 angledelta mul add drawangle } if
end } bind def
/drawangle {7 dict begin % a c b r
  /r exch def /b exch def /c exch def /a exch def
  /beta b 0 get c 0 get sub b 1 get c 1 get sub angle def
  /alpha a 0 get c 0 get sub a 1 get c 1 get sub angle def
  /phi beta alpha sub def
  gsave
  lw2 setlinewidth
  c aload pop translate
  alpha rotate
  phi 0 gt {
    phi 180 lt {
      newpath 0 0 r 0 phi arc stroke
    }{
      /phi phi 360 sub def
    }
  }

```

```

        newpath 0 0 r 0 phi arcn stroke
    } ifelse
  }{
    phi -180 lt {
      /phi phi 360 add def
      newpath 0 0 r 0 phi arc stroke
    }{
      newpath 0 0 r 0 phi arcn stroke
    } ifelse

  } ifelse
  grestore
end } bind def
%
/markrightangle {6 dict begin % stack: a c b
  /b exch def /c exch def /a exch def
  /beta b 0 get c 0 get sub b 1 get c 1 get sub angle def
  /alpha a 0 get c 0 get sub a 1 get c 1 get sub angle def
  /phi beta alpha sub def
  gsave
  lw2 setlinewidth
  c aload pop translate marksize dup scale
  alpha rotate
  phi 0 gt {
    phi 180 lt {
      newpath 1 0 moveto 0 1 rlineto -1 0 rlineto stroke
    }{
      newpath 1 0 moveto 0 -1 rlineto -1 0 rlineto stroke
    } ifelse
  }{
    phi -180 lt {
      newpath 1 0 moveto 0 1 rlineto -1 0 rlineto stroke
    }{
      newpath 1 0 moveto 0 -1 rlineto -1 0 rlineto stroke
    } ifelse
  } ifelse
  grestore
end } bind def
%
/tickmark { 5 dict begin % stack: a b n
  /n exch def /b exch def /a exch def
  /phi b 0 get a 0 get sub b 1 get a 1 get sub angle def
  /s a 0 get b 0 get sub dup mul a 1 get b 1 get sub dup mul add sqrt 2 div def
  gsave
  lw2 setlinewidth
  a aload pop translate phi rotate
  n 1 eq {
    newpath s 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto stroke
  } if
  n 2 eq {
    newpath
    s marksize 4 div sub 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    s marksize 4 div add 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto

```

```

    stroke
  } if
  n 3 eq {
    newpath
    s marksize 2 div sub 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    s 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto stroke
    s marksize 2 div add 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    stroke
  } if
  n 4 ge {
    newpath
    s marksize 3 mul 4 div sub 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    s marksize 4 div sub 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    s marksize 4 div add 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    s marksize 3 mul 4 div add 0 moveto 0 marksize neg rlineto 0 2 marksize mul rlineto
    stroke
  } if
  grestore
end } bind def
%%EndProlog
u dup scale
0.5 0.5 translate
lw1 setlinewidth
% points of triangle ABC and base point D
/A [0 0] def
/B [3 0] def
/C [1.5 4] def
/D [1.5 0] def
A 0 get 0.3 sub A 1 get 0.4 sub moveto f1 (A) show
B 0 get 0.05 add B 1 get 0.4 sub moveto f1 (B) show
C 0 get 0.15 sub C 1 get 0.2 add moveto f1 (C) show
D 0 get 0.15 sub D 1 get 0.4 sub moveto f1 (D) show
% labeling
newpath
A aload pop moveto
B aload pop lineto
C aload pop lineto
closepath
C aload pop moveto
D aload pop lineto
stroke
% annotation
A D 1 tickmark
B D 1 tickmark
A C 2 tickmark
B C 2 tickmark
C A B 2 markangle
A B C 2 markangle
B C A 1 markangle
C D B markrightangle
showpage
%%EOF

```

---

## References

- [Cas05] Casselman, B. *Mathematical Illustrations*, Cambridge University Press, 2005. Downloadable at [www.math.ubc.ca/~cass/graphics/manual/](http://www.math.ubc.ca/~cass/graphics/manual/)
- [Pos99] *PostScript Language Reference Manual*, Adobe Systems, 3rd printing, 1999. Known informally as ‘the red book’. Downloadable at [www-cdf.fnal.gov/offline/PostScript/PLRM3.pdf](http://www-cdf.fnal.gov/offline/PostScript/PLRM3.pdf)
- [Pos88] *PostScript Language Program Design*, Adobe Systems, 1st printing, 1988. Known informally as ‘the green book’. Downloadable at [www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF](http://www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF)
- [Pos86] *PostScript Language Tutorial and Cookbook*, Adobe Systems, 2nd printing, 1986. Known informally as ‘the blue book’. Downloadable at [www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF](http://www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF)